Deep Surrogate Models of Serverless Batch Processing Services

Yicheng Gao[1], Roberto Sala[2], Danilo Ardagna[2], Giuliano Casale[1]

[1] Department of Computing, Imperial College London, UK
{y.gao20, g.casale}@imperial.ac.uk
[2] DEIB, Politecnico di Milano, Italy
{roberto.sala, danilo.ardagna}@polimi.it

Abstract. Serverless computing breaks down applications into workflows of stateless functions, where the output of each function serves as the input for the next. When these workflows are distributed across multiple processing nodes, managing the interactions between nodes is crucial to maintaining overall system performance. However, existing analytical performance models do not cope well with the dependencies involved in distributed workflows when the offloaded jobs arrive in batches. In this paper, we study the problem for two processing resources in tandem and develop a scalable surrogate modeling approach based on neural networks that can be used for serverless resource management purposes. We validate our performance model with both synthetic and real-world AI application traces, demonstrating that our surrogate model achieves a mean average percentage error of about 5%.

Keywords: Batch processing · Function-as-a-service · Stochastic model · Performance measures

1 Introduction

Edge computing has emerged in recent years as a key paradigm for distributing computing. Recently, Function as a Service (FaaS) has garnered significant attention and current research often focuses on designing platforms and architectures that adapt serverless computing to edge scenarios [15]. In this paper, we focus instead on the problem of performance characterization of serverless solutions for prediction tasks in resource management. Many studies adopt learning-based approaches to analyze and predict the performance of applications running in edge computing systems [9][23][13][2][20]. For example, in [23], a neural network model, built with denoising auto-encoder and fuzzy clustering for QoS prediction in mobile edge environments, is designed to improve the prediction accuracy while addressing the overfitting problem. In [2], five widely used machine learning models, such as support vector regression and random forest, are considered for predicting CNN inference execution time on edge GPUs. However, when applying learning-based methods in the design phase of serverless edge computing systems, extensive large-scale profiling experiments are required to collect

sufficient training data, leading to significant operational costs. Additionally, gathering adequate data in the design phase may be particularly challenging due to the limited availability of system performance metrics at this stage.

Alternatively, some researchers consider analytical performance models for FaaS platforms, allowing application developers and serverless operators to perform capacity planning and system analysis rapidly, without the need for costly and large-scale experimental setups [11]. The authors in [12] and [11] develop transient and steady-state analytical performance models based on M/G/m/m queueing systems to make capacity predictions for serverless computing platforms. The authors in [5] present a sizing method, called COCOA, leveraging the layered queueing network (LQN) model and $M/M/k$ setup models to derive performance estimates for provisioning FaaS systems that meet service-level agreements.

While these existing analytical models are well-suited for performance prediction of serverless computing, they overlook the fact that offloaded jobs often arrive in batches [8], leading to significant challenges due to the limited computational resources available at the edge. These batch arrivals complicate resource allocation and queueing dynamics, often resulting in performance bottlenecks. Therefore, efficient performance models are crucial to capture the constraints and operational dynamics of serverless edge computing systems.

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

- We investigate the workflow characteristics of application traces from AI-SPRINT, a real-world serverless platform, offering an in-depth analysis of its batch arrival patterns [15, 4].
- We formulate a performance model based on tandem queueing networks with batch arrivals to capture the impacts of batch job patterns and distributed workflow dependencies.
- To address the computational overhead of analytical methods, we propose a DNN-based surrogate model, henceforth referred to as a deep surrogate model, to approximate the performance metrics of tandem workflows.
- We validate our performance model through extensive simulations with both synthetic workloads and real-world traces, demonstrating that our surrogate model achieves strong alignment with actual performance data, yielding a mean average percentage error of 5.33%.

The rest of the paper is organized as follows. Section 2 introduces the AI-SPRINT applications and presents a detailed trace analysis. Section 3 gives

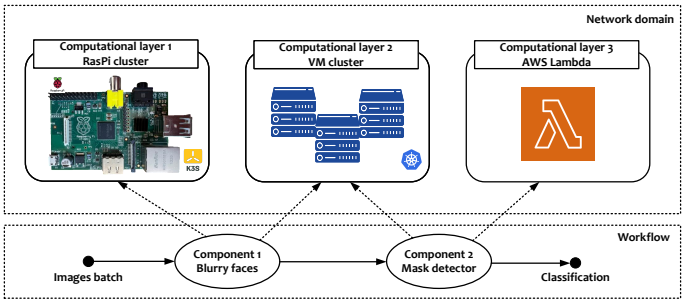

Fig. 1. An example of an AI-SPRINT application for mask detection. Edge nodes are managed through minified Kubernetes distributions (*e.g.,* K3s). Containers deployed in private and public VMs are orchestrated by K8s.

the motivation for modeling with tandem queueing networks and outlines the approach for their approximation. Section 4 presents the performance model and deep surrogate modeling approach. Experimental results are discussed in Section 5 prior to conclusions.

2 System Characteristics

2.1 AI-SPRINT Applications and Profiling

AI-SPRINT ("Artificial Intelligence in Secure PRIvacy-preserving computing coNTinuum") is a comprehensive framework designed to facilitate the development and deployment of AI applications across a distributed computing continuum, encompassing both edge devices and cloud infrastructures [18][3]. The framework addresses the critical need to balance performance, accuracy, and security within AI-driven workflows.

An AI-SPRINT application consists of multiple interconnected Python components, each performing a specific task within the workflow. The execution of these components is event-driven and orchestrated by OSCAR [14], an open-source platform that supports the FaaS model. Each component is triggered by file uploads, and the output of one component triggers the next in the sequence.

To ensure execution consistency and reliability, each component is encapsulated within a Docker container. The architecture supports a diverse range of computational resources, from edge devices like Raspberry Pis to cloud-based virtual machines and AWS Lambda, allowing for optimized resource allocation tailored to the computational demands of each component.

An example of an AI-SPRINT application is depicted in Fig. 1, which features a two-component workflow for mask detection executed across multiple computing layers. The first component, *Blurry Faces*, processes video input by extracting frames every five seconds, detecting faces using a neural network, and anonymizing them through pixel modification. The anonymized frames are then passed to the second component, *Mask Detector*, which detects the presence of masks on the faces and annotates the images with corresponding labels and confidence scores. This application exemplifies the use of edge computing, where

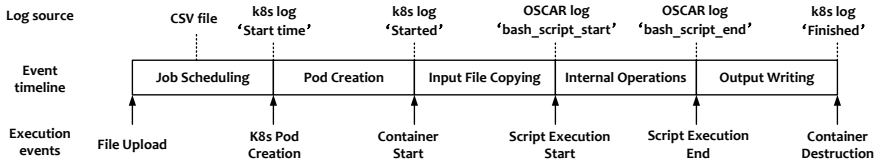

Fig. 2. Detailed logging process of OSCAR-P with execution stages and timestamps

video preprocessing is performed on a cluster of Raspberry Pi devices to ensure data privacy by keeping sensitive information at the edge. The anonymized frames are then uploaded and analyzed in the cloud, allowing for efficient and secure mask detection across various urban areas.

To systematically profile AI-SPRINT applications across diverse hardware configurations, an advanced auto-profiling tool integrated with the OSCAR framework named OSCAR-P is further employed [4]. The profiling process begins with specifying detailed hardware parameters and application requirements, including node configurations, memory capacity, processing power, Docker images, and levels of parallelism. OSCAR-P then conducts experiments by dynamically adjusting parameters and executing the application under various scenarios. After each run, it retrieves and processes execution logs, capturing timing, resource usage, and overall performance for each component.

Fig. 2 provides the detailed logging process managed by OSCAR-P, highlighting the execution stages and their corresponding timestamps. The logs, collected from both K8s and OSCAR, provide in-depth insights into the application performance across the different stages.

For such AI-SPRINT applications, predicting performance during the design phase is crucial, particularly when estimating the response time across various configurations, as it enables effective capacity planning, resource management, while maintaining quality of service. However, this task is challenging due to the complexity of distributed, multi-component workflows. Accurately predicting execution time for components across a computing continuum and understanding their complex interactions is inherently difficult. Moreover, the design phase often lacks sufficient data, making data-driven approaches both time-consuming and costly to implement [10]. To address these challenges, we propose an analytical approach that captures system performance. To reduce the computational overhead of the specific analytical method, we develop a deep surrogate model for efficient performance metric prediction.

2.2 Trace Collection and Analysis

In this section, we analyse the mask detection application execution traces collected through OSCAR-P. As depicted in Fig. 1, the components are profiled using multiple configurations on AWS EC2. The trace spans six days, during which a series of experiments are conducted to comprehensively assess the system under a wide range of operational conditions.

Input Traffic Configuration: The experiments consider different numbers of 5-second videos as input. Each set of input data is processed three times

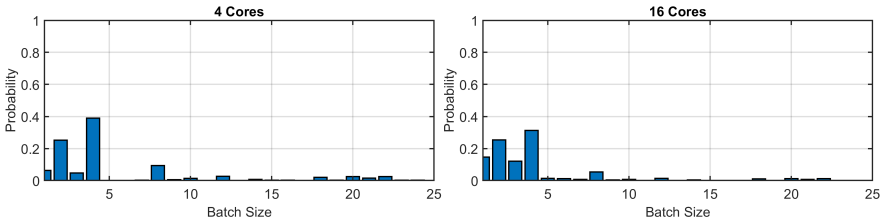

Fig. 3. Probability mass function of batch sizes across different core configurations

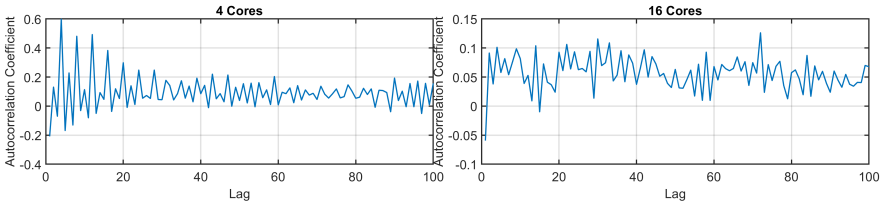

Fig. 4. Autocorrelation function of batch sizes across different core configurations

on VMs to ensure robust and consistent results. To simulate diverse real-world usage scenarios, the input traffic is configured to arrive in batches, with varying throughput rates ranging from 0.02 to 1.86 requests per second. This variability is achieved by maintaining a batch size of four files and altering the inter-arrival time between batches, allowing for a detailed analysis of the system performance under various load conditions.

VMs Configuration: The configuration of the VM cluster is critical for understanding how the number of cores affects the performance of the mask-detection application. In this context, each node refers to an individual VM within the cluster. For simplicity, we profiled the application on homogeneous AWS EC2 m4.large VMs, equipped with Intel Xeon processors, each featuring two cores and 8 GB of memory. This configuration allows a maximum of two instances of the application to run concurrently on each node. Each application component operates within an isolated, single-core container, requiring one core to function. Thus, the level of parallelism—the maximum number of concurrent containers that can run on a node—is directly determined by the number of available cores. In this setup, the VMs scale horizontally to configurations with 2, 4, 8, and 16 cores for profiling the application. Notably, the service times for all components remain relatively stable across these configurations.

With these predefined configurations, AI-SPRINT application traces are collected from logs across a varying number of cores. Both the full workflow and its individual components are profiled. Each log file records essential information about the jobs, including creation and completion times, as well as the parallelism level. To capture these characteristics, we analyze each single component by its probability distribution and temporal dependencies, enabling accurate performance predictions. Specifically, we utilize the Autocorrelation Function (ACF) to analyze the temporal dependencies within the traces.

For each individual component, Fig. 3 illustrates that the probability mass function (PMF) spans batch sizes from 1 to 25, with a notable concentration

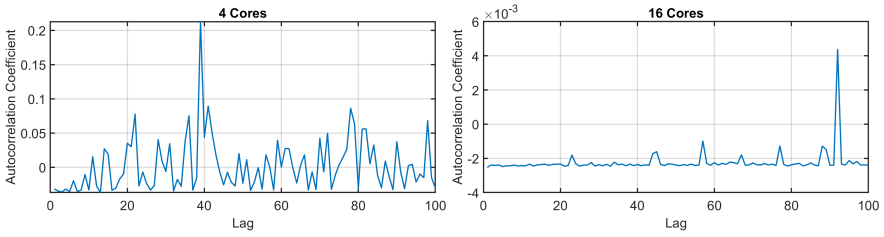

Fig. 5. Autocorrelation function of inter-arrival times between batches

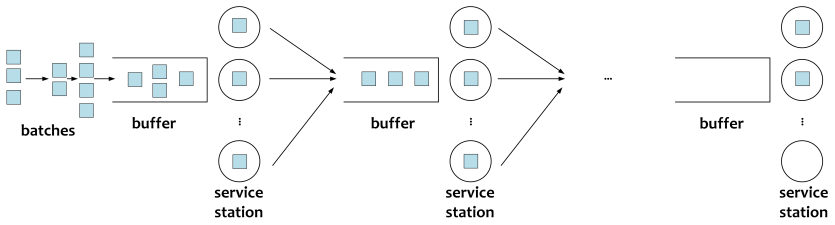

Fig. 6. An example of a tandem queueing network with batch arrivals

around 4. Although the system is configured for batches of 4 jobs, Kubernetes scheduling policies (which determine resource allocation, pod distribution, and job queueing) led to variations in the actual batch sizes. These factors can create discrepancies between the intended batch size and the one observed during execution. Furthermore, Fig. 4 shows that the autocorrelation of batch sizes is weak and diminishes rapidly, implying that the sizes of consecutive batches are independent. In Fig. 5, the periodic spikes observed in the ACF of the inter-arrival times suggest the presence of recurring scheduling patterns. However, these periodic spikes diminish rapidly, indicating that the inter-arrival times between batches are close to independent over lags.

3 Motivation

Given the characteristics of the AI-SPRINT application workflow, we propose modeling the system as a tandem queueing network, where each component operates as a sequential queueing system, an example is given in Fig. 6.

3.1 Traffic Decomposition Methods

To address state spaces explosion and complex interactions within such tandem networks, we apply the traffic decomposition approach [21]. This technique enables the analysis of each node in isolation by treating the output from one queue as the arrival process for the next. Each output is characterized in terms of its statistical moments, and in some works also its autocorrelation function [7], and then fit to a simplified stochastic process to describe arrivals. Within the traffic decomposition approach, we propose to capture the dynamics of each component by employing the BMMPP/GI/c model that we approximate by means of a neural surrogate. A BMMPP is a Batched Marked Modulated process that can be used to describe non-i.i.d. point processes with batch arrivals [22]. BMMPP/GI/c

queueing systems have general independent service and c servers. They are typically classified as M/G/c-type models and their underlying (infinite) continuous-time Markov chain is characterized by an upper block Hessenberg structure in their infinitesimal generator $\boldsymbol{Q}_{\text{M/G/c-type}}$, where only the first block below the main diagonal contains non-zero elements, while all subsequent lower blocks are zero matrices.

To approximate the BMMPP/GI/c model, which is difficult to analyze due to state-space explosion driven by its multiserver parameter c, we initially focus on the response time and model the system as an equivalent single-server BMMPP/GI/1 queue with an adjusted service rate to account for c. Our proposal is to adapt for the first time an approximation method in [19] proposed in the context of product-form queueing network theory to multiserver matrix analytic queueing systems.

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

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

as intended. Summarizing, the response time $\widetilde{R_m}$ can be approximated as [19]

$$\widetilde{R_m} = R_s + \frac{c-1}{c} \times \frac{1}{\mu}. \tag{1}$$

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

To determine the accuracy of the proposed approximations in a way that abstracts the accuracy of the associated single-server solver, we test the proposed method first using simulation. The approximation error σ is evaluated as

$$\sigma = \frac{|\widetilde{R_m} - R_m|}{R_m} \times 100\%, \tag{2}$$

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

As illustrated in Fig. 7, the approximation error remains minimal across different numbers of servers and mean batch sizes. For scenarios with a small number of servers (*e.g.,* $c \leq 5$), the error is consistently below 2%, demonstrating that the approximation is highly accurate in these settings. As the number of servers increases, the error gradually escalates but the deviation remains limited

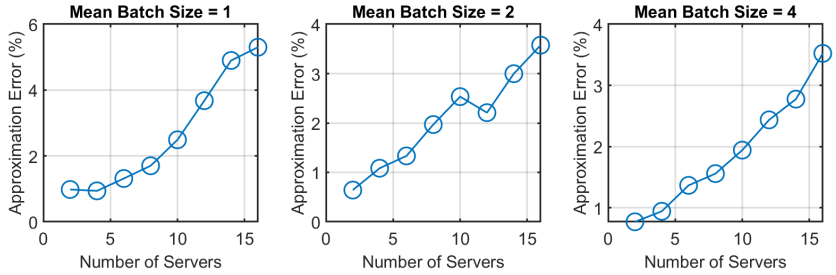

Fig. 7. Approximation error for multiserver scenarios

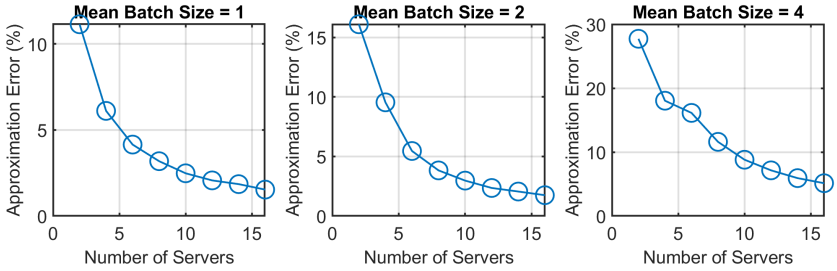

Fig. 8. Approximation error for complete tandem workflows

to 4-6%. These results indicate that the approximation provides a precise solution for a BMMPP/GI/c system.

3.2 Tandem Workflow Approximation

In the full application workflow modeled as a tandem queueing network, the departure process of one component directly serves as the arrival process for the subsequent component. To approximate the performance of the full tandem workflow, we initially leverage MAMsolver [16][17] to solve each individual component within the network. MAMsolver integrates the ETAQA technique to enhance computational efficiency in computing key stationary performance metrics M/G/1-type, GI/M/1-type, and QBD processes. In this initial evaluation, the departure process of each component is derived with simulation to analyze moments and ACF of the inter-departure times, and then passed as the arrival process for the subsequent component in the tandem workflow. For all subsequent components, the arrival process is defined by the departure process of the preceding component. This sequential analysis allows us to approximate the performance of the entire tandem workflow, despite the challenges posed by inter-component dependencies. Later on, our methodology will not require using simulation anymore, which is considered here only to illustrate the level of accuracy that could be achieved through moment-based traffic decomposition when the values of the moments are accurately estimated.

To simplify the analysis, we first approximate the inter-departure times by fitting them to an Erlang-k distribution, which captures the first two moments. The results in Fig. 8 show that the approximation error for the complete workflow

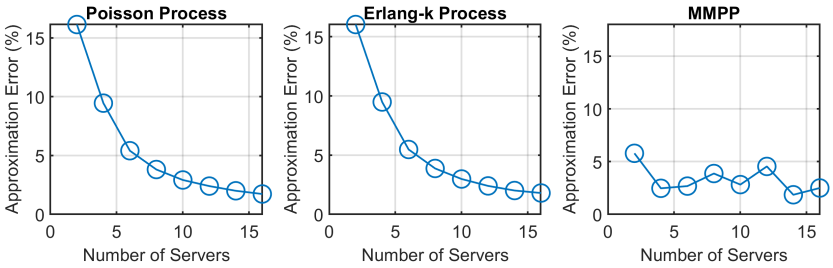

Fig. 9. Approximation error for tandem workflows with different arrival process fitting methods in the subsequent queues

remains below 27.73% across all configurations. As the mean batch size increases, the error tends to rise, particularly in scenarios with fewer servers. For smaller batch sizes, the error decreases rapidly, falling below 5% as the number of servers increases, indicating a high level of accuracy. However, as the batch size grows, the error increases to around 30%, reflecting the increased difficulty in modeling more complex batch dynamics.

Further, we compare different methods of fitting the arrival process for the subsequent queues within the tandem workflow, specifically under a batch size of 2. The results in Fig. 9 compare the approximation errors across tandem workflows modeled with three different types of arrival processes for the subsequent queues: Poisson, Erlang-k, and MMPP. The MMPP model, which accounts for the first three moments and the lag-1 ACF, demonstrates higher performance, with the approximation error consistently staying below 5.8% across different configurations. This enhanced accuracy highlights the capability of MMPP to capture the more complex dynamics of the departure process of batch queues compared to the Erlang-k and Poisson models, which ignore dependence between inter-departure times.

Issues with Approximation Methods. While existing BMMPP/GI/1 solvers provide accurate approximations of multiserver systems, they suffer from high computational complexity when applied to low variability arrivals as in the AI-SPRINT system. To demonstrate this, we employ MAMsolver as a case study for analyzing the BMMPP/GI/1 model. Specifically, we examine the time complexity associated with varying numbers of phases in the arrival process, averaging 1,000 samples for each configuration. Running experimental tests, we find that if both the arrival and service processes are considered, the computational burden can exceed 10^6 seconds, further illustrating the limitations of this approach in handling complex systems. This approach becomes particularly costly in design studies, where thousands of models may need to be solved. To address this issue, we propose leveraging surrogate models as discussed in the next section.

4 Deep Surrogate Model

4.1 Performance Model

Since we apply the decomposition technique to the entire application workflow, this section provides a detailed performance modeling of each individual component. Specifically, we utilize a two-state Batch Markov Modulated Poisson

Process, denoted as BMMPP(2), to model the batch arrival process of each component. The intensity matrices of BMMPP(2) can be characterized by $\mathbf{D}_b$, $b = 0, 1, \ldots, B$, where B is the maximum batch size, as follows.

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

$$\mathbf{D}_0 = \begin{bmatrix} -\lambda_1 & \lambda_{12} \\ \lambda_{21} & -\lambda_2 \end{bmatrix}, \tag{3}$$

where λ_{12} and λ_{21} represent the transition rates between the two states. The states map to different arrival rates for the incoming jobs. The intensity matrices $\mathbf{D}_b$, $b = 1, \ldots, B-1$, describe transitions associated with batch arrivals of size b and are given by $\mathbf{D}_b = \text{diag}(\lambda_1 p_{11b}, \lambda_2 p_{22b})$, where p_{11b} and p_{22b} are the probabilities of observing a batch arrival of size b in states 1 and 2, respectively. The intensity matrix $\mathbf{D}_B$ is defined as:

$$\mathbf{D}_B = \begin{bmatrix} -\lambda_1 - \lambda_{12} - \sum_{i=1}^{B-1} \lambda_1 p_{11i} & 0 \\ 0 & -\lambda_2 - \lambda_{21} - \sum_{i=1}^{B-1} \lambda_2 p_{22i} \end{bmatrix}. \tag{4}$$

Thus, the infinitesimal generator $\mathbf{Q}_{BMMPP(2)}$ of the underlying Markov process is given by $\mathbf{Q}_{BMMPP(2)} = \sum_{b=0}^{B} \mathbf{D}_b$.

Given the negligible correlation in the trace data, we consider a variant of the BMMPP(2) where the batch size probabilities are assumed to be identical across states, *i.e.,* $p_{iib} = p_{jjb}$, for any $i \neq j$. Under this assumption, the intensity matrices can be expressed as [22] $\mathbf{D}_0 = \mathbf{Q} - \boldsymbol{\Delta}(\boldsymbol{\delta})$, $\quad \mathbf{D}_b = \boldsymbol{\Delta}(\boldsymbol{\delta}) \cdot \boldsymbol{\Delta}(p_b)$, $\quad b \geq 1$, where $\boldsymbol{\Delta}(\delta)$ is a diagonal matrix with $\boldsymbol{\delta} = [\lambda_1, \lambda_2]^T$, and $\boldsymbol{\delta}(p_b)$ is a diagonal matrix with the batch probabilities p_b as its diagonal elements, satisfying $\sum_{b=1}^{B} p_b = 1$.

By assuming no correlation between inter-arrival times and batch sizes, this variant, denoted as BMMPP$_v$(2) and also known as a MAP with independent and identically distributed (i.i.d.) batch arrivals, provides a tractable approach for modeling the batch arrival process of each application component. Additionally, the service time distribution is modeled using a General Independent (GI) distribution, offering flexibility to capture a wide range of service behaviors, and the model incorporates c servers to account for the parallel processing capabilities of each component.

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

$$\boldsymbol{Q}_{\text{BMMPP}_v(2)/\text{GI}/c} = \begin{bmatrix} \boldsymbol{B}_0 & \boldsymbol{B}_1 & \boldsymbol{B}_2 & \boldsymbol{B}_3 & \cdots \\ \boldsymbol{C}_0 & \boldsymbol{A}_1 & \boldsymbol{A}_2 & \boldsymbol{A}_3 & \cdots \\ 0 & \boldsymbol{A}_0 & \boldsymbol{A}_1 & \boldsymbol{A}_2 & \cdots \\ 0 & 0 & \boldsymbol{A}_0 & \boldsymbol{A}_1 & \ddots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix} \tag{5}$$

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

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

$$\boldsymbol{A}_0 = \begin{bmatrix} 0 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ r\mu & \cdots & 0 \end{bmatrix}, \boldsymbol{A}_1 = \begin{bmatrix} -k\lambda + r\mu & r\mu & \cdots & 0 \\ 0 & -k\lambda + r\mu & r\mu & \cdots \\ 0 & 0 & -k\lambda + r\mu & r\mu \cdots \\ \vdots & \vdots & \ddots & \ddots \ddots \end{bmatrix} \tag{6}$$

$$\boldsymbol{A}_k = \begin{bmatrix} P_x\lambda & 0 & 0 & \cdots & & \\ 0 & P_x\lambda & 0 & 0 & \cdots & \\ 0 & 0 & P_x\lambda & 0 & 0 & \cdots \\ \vdots & \vdots & \vdots & \ddots & \ddots & \ddots \end{bmatrix} \tag{7}$$

where P_x denotes the probability mass function of the batch size. $\boldsymbol{C}_0 = [0, 0, \ldots, r\mu]^T$ represents the backward transition for the first repetitive block. Moreover, $\boldsymbol{B}_0$ and $\boldsymbol{B}_k$ denote the local and forward transitions for the boundary states, where $\boldsymbol{B}_0 = -\lambda$ and $\boldsymbol{B}_k = [\cdots 0, 0, P_k\lambda, 0, 0, \cdots]$.

4.2 Deep Surrogate Model

To solve the BMMPP$_v$(2)/GI/c queue in a more efficient way than with traditional matrix analytic methods, we further propose a Deep Neural Network (DNN)-based surrogate model. This model is tailored to learn the system behavior and approximate the performance metrics of each component in multiserver systems. Specifically, it employs a feedforward neural network architecture with multiple hidden layers of fully connected neurons. Each hidden layer applies Rectified Linear Unit (ReLU) activation functions to introduce nonlinearity, enhancing the capability to learn complex mappings between input features and performance metrics.

Training data for the proposed deep surrogate model is generated from simulations of BMMPP/PH/c systems. The dataset includes critical features such as the statistical moments of arrival processes E_{a1}, E_{a2}, E_{a3}, the PMF of batch arrival sizes P_{bs}, service process parameters E_{s1}, E_{s2}, and the number of servers c. The target outputs are performance metrics, including moments of response times. To ensure effective training and enhance model performance, both the input features and output targets are normalized using min-max scaling. The training process focuses on minimizing the Mean Absolute Percentage Error (MAPE) between the predicted and actual performance metrics. An adaptive optimization algorithm adjusts the learning rate dynamically to ensure efficient convergence. The network undergoes training for a specified number of epochs, with a batch size optimized for both computational efficiency and memory management, facilitating robust learning and accurate approximation of the performance metrics.

Table 1. Parameters for training data generation

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

5 Experimental Results

5.1 Simulation-based Evaluation

In this section, we evaluate our proposed deep surrogate model for full tandem workflows using data simulated by the JMT tool [1]. For each component in the tandem workflow, we model it as a BMAP/GI/c system, where BMAP generalizes BMMPP for more complex dynamics.

BMAP Sample Generation. First, we generate samples of BMAP to simulate the arrival process for each component in the tandem queueing network. For the first component, by configuring the mean inter-arrival time between batches and the squared coefficient of variation (SCV), we characterize the BMAP using the following parameters: the mean inter-arrival time E_1, which is the average time between consecutive batch arrivals; the second central moment $E_2 = (1 + \text{SCV}) \times E_1^2$; and the lag-2 autocorrelation γ_2. The third moment E_3 is set to the minimum value possible for the underlying MAP using the formulas in [6].

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

Training Data Generation. With the input of BMAP samples for each component, we further simulate the mean response time of each queue with the JMT tool. In parallel, we perform a full workflow simulation using the JMT tool, with the arrival process of the first component as input, allowing for assessing the overall system performance. To generate the training data for the deep surrogate model, we further detail the configuration by simulating a range of scenarios using the JMT tool in Table 1.

Training Process. We train the deep surrogate model using the following architecture and configurations. The DNN consists of four hidden layers with 128, 64, 32, and 16 neurons respectively, using ReLU activation functions for non-linearity. The final layer outputs the moments of response time prediction for each component of the tandem workflow. We implement the model using PyTorch and optimize it using the Adam optimizer. The learning rate is 0.001, the batch size is 512, and the number of epochs is 100.

Experimental Evaluation. The effectiveness of our proposed deep surrogate model is evaluated by comparing its aggregated predictions with the response times

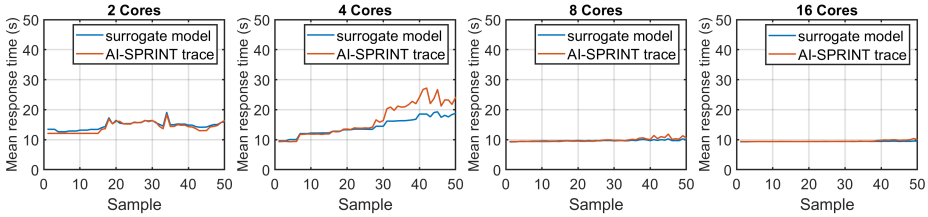

Fig. 10. Comparison of the response time between trace-based simulations and deep surrogate model

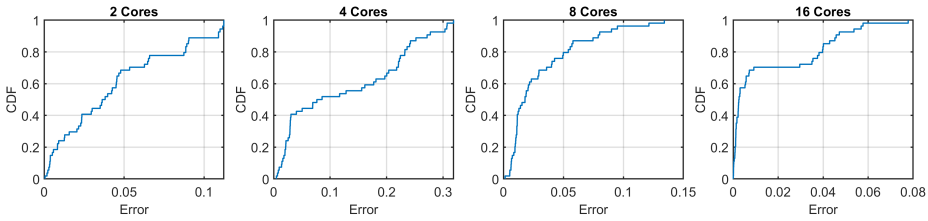

Fig. 11. CDF of the prediction error of the deep surrogate model

obtained from the full tandem workflow simulations conducted in the JMT tool. The evaluation results present an average MAPE of 19.04% across all samples, demonstrating the accuracy of our proposed surrogate model in predicting the performance of tandem queueing systems.

5.2 Mask Detection Application Trace

To evaluate the effectiveness of our proposed deep surrogate model for real-world tandem workflows, we compare its performance against the trace-driven simulation using traces collected from the mask detection applications operating within serverless edge computing systems. The experimental setup includes 216 different configurations, varying by the number of cores and job arrival rates. The trace data collected under each configuration spans a range of 32 to 3,968 requests. The inter-arrival times between batches within these traces exhibit low variability, allowing us to approximate them using an Erlang distribution. This approximation is well-suited for time-to-event random variables with low variability in Markovian modeling, as the Erlang distribution accurately captures the first two moments, *i.e.,* the mean and the variance, effectively in this context.

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

Fig. 10 presents the comparison of response times between the surrogate model and the AI-SPRINT trace across four core configurations. The surrogate model demonstrates a strong alignment with the mask detection trace, achieving an overall MAPE of 5.33%, indicating its effectiveness in capturing the system behavior. For the 2-core, 8-core, and 16-core configurations, response times are consistently below 20 seconds showing minimal deviation. In the 4-core configuration, a slight fluctuation is observed, where the application trace shows an increase in response times to around 20-25 seconds, while the surrogate model captures the overall trend but with a degree of error. This discrepancy is likely due to sudden bursts of high load in the real system, managed by Kubernetes, leading to temporary resource contention and scheduling delays.

We conjecture that during the 4-core experiments, the system may have entered an unusual state midway through the test, differing from the typical conditions observed in other tests. However, we could not firmly confirm this as we did not have direct access to the internals. Indeed, the strong alignment in the first half of the experiment with 4 cores shows that under normal conditions this case can also be matched fairly accurately. The overall alignment underscores the robustness of the model and its ability to generalize across diverse configurations, accurately predicting system performance in real-world scenarios.

Fig.e 11 shows the CDF of the prediction error of the proposed surrogate model under different core configurations. The CDF of the error across 2, 8, and 16-core configurations demonstrates that the surrogate model consistently achieves high accuracy, with most errors remaining below 0.1. These cases reflect strong alignment with the AI-SPRINT trace, indicating the robustness of the model under dynamic environments. The 4-core configuration stands out as an outlier, with maximum errors extending up to 30%, which are still acceptable from a practical standpoint. Nevertheless, the median error is far lower and equal to 10%. Overall, the CDF analysis demonstrates that our proposed surrogate model achieves high accuracy across different configurations.

6 Conclusion

In this paper, we propose a surrogate modeling approach for performance prediction in serverless edge computing systems. Focusing on real application traces, we show the need for capturing batch arrival patterns and propose a tandem BMMPP/GI/c performance model. Extensive simulations to validate the surrogate model, using both synthetic workloads and real-world traces, demonstrate that our proposed deep neural surrogate achieves high accuracy against real traces. Future work may look at extending our surrogate models to non-tandem topologies and multiclass batch arrivals.

References

1. Bertoli, M., et al.: JMT: Performance engineering tools for system modeling. ACM SIGMETRICS PER **36**(4), 10–15 (2009)
2. Bouzidi, H., et al.: Performance prediction for convolutional neural networks on edge GPUs. In: ACM CF. pp. 54–62 (2021)

3. Filippini, F., et al.: SPACE4AI-R: a runtime management tool for ai applications component placement and resource scaling in computing continua. In: IEEE/ACM UCC. pp. 1–7 (2023)
4. Galimberti, E., et al.: OSCAR-P and amllibrary: Performance profiling and prediction of computing continua applications. In: ACM/SPEC ICPE. pp. 139–146 (2023)
5. Gias, A.U., Casale, G.: Cocoa: Cold start aware capacity planning for function-as-a-service platforms. In: IEEE MASCOTS. pp. 1–8 (2020)
6. Heindl, A., et al.: Explicit inverse characterizations of acyclic maps of second order. In: EPEW 2006. pp. 108–122. Springer (2006)
7. Horváth, A., Telek, M.: A joint moments based analysis of networks of MAP/MAP/1 queues. PEVA **67**(9), 759–778 (2010)
8. Huang, J., et al.: Revenue-optimal task scheduling and resource management for iot batch jobs in mobile edge computing. Peer-to-Peer Networking and Applications **13**(5), 1776–1787 (2020)
9. Liu, Z., et al.: Context-aware and adaptive QoS prediction for mobile edge computing services. IEEE TSC **15**(1), 400–413 (2019)
10. Madougou, S., et al.: The landscape of GPGPU performance modeling tools. Parallel Computing **56**, 18–33 (2016)
11. Mahmoudi, N., Khazaei, H.: Performance modeling of serverless computing platforms. IEEE Trans. on Cloud Computing **10**(4), 2834–2847 (2020)
12. Mahmoudi, N., Khazaei, H.: Temporal performance modeling of serverless computing platforms. In: WoSC. pp. 1–6 (2020)
13. Miao, Y., et al.: Intelligent task prediction and computation offloading based on mobile-edge cloud computing. FGCS **102**, 925–931 (2020)
14. Pérez, A., et al.: On-premises serverless computing for event-driven data processing applications. In: IEEE CLOUD. pp. 414–421 (2019)
15. Risco, S., et al.: Serverless workflows for containerised applications in the cloud continuum. Journal of Grid Computing **19**, 1–18 (2021)
16. Riska, A., Smirni, E.: MAMsolver: A matrix analytic methods tool. In: International Conference on Modelling Techniques and Tools for Computer Performance Evaluation. pp. 205–211 (2002)
17. Riska, A., Smirni, E.: ETAQA solutions for infinite markov processes with repetitive structure. INFORMS J. on Computing **19**(2), 215–228 (2007)
18. Sedghani, H., et al.: Advancing design and runtime management of ai applications with ai-sprint (position paper). In: IEEE COMPSAC. pp. 1455–1462 (2021)
19. Seidmann, A., et al.: Computerized closed queueing network models of flexible manufacturing systems: A comparative evaluation. Large Scale Systems **12**(2), 91–107 (1987)
20. Tang, Q., et al.: Distributed task scheduling in serverless edge computing networks for the IoT: A learning approach. IEEE IoT Journal **9**(20), 19634–19648 (2022)
21. Whitt, W.: The queueing network analyzer. Bell System Tech. J. **62**(9), 2779–2815 (1983)
22. Yera, Y., et al.: Fitting procedure for the two-state batch markov modulated poisson process. EJOR **279**(1), 79–92 (2019)
23. Yin, Y., et al.: QoS prediction for service recommendation with features learning in mobile edge computing. IEEE TCCN **6**(4), 1136–1145 (2020)