2019 IEEE International Conference on Cloud Engineering (IC2E)

Predicting the End-to-End Tail Latency of Containerized Microservices in the Cloud

Joy Rahman
Dept. of Computer Science
University of Texas at San Antonio
San Antonio, Texas-78249
Email: joy.rahman@utsa.edu

Palden Lama
Dept. of Computer Science
University of Texas at San Antonio
San Antonio, Texas-78249
Email: palden.lama@utsa.edu

***Abstract*—Large-scale web services are increasingly adopting cloud-native principles of application design to better utilize the advantages of cloud computing. This involves building an application using many loosely coupled service-specific components (microservices) that communicate via lightweight APIs, and utilizing containerization technologies to deploy, update, and scale these microservices quickly and independently. However, managing the end-to-end tail latency of requests flowing through the microservices is challenging in the absence of accurate performance models that can capture the complex interplay of microservice workflows with cloud-induced performance variability and inter-service performance dependencies. In this paper, we present performance characterization and modeling of containerized microservices in the cloud. Our modeling approach aims at enabling cloud platforms to combine resource usage metrics collected from multiple layers of the cloud environment, and apply machine learning techniques to predict the end-to-end tail latency of microservice workflows. We implemented and evaluated our modeling approach on NSF Cloud's Chameleon testbed using KVM for virtualization, Docker Engine for containerization and Kubernetes for container orchestration. Experimental results with an open-source microservices benchmark, Sock Shop, show that our modeling approach achieves high prediction accuracy even in the presence of multi-tenant performance interference.**

***Keywords*-microservices; containers; cloud computing; performance modeling;**

I. Introduction

Large-scale web services (e.g Netflix, Microsoft Bing, Uber, Spotify etc.) are increasingly adopting cloud-native principles and design patterns such as microservices and containers to better utilize the advantages of the cloud computing delivery model, which includes greater agility in software deployment, automated scalability, and portability across cloud environments [24, 30]. In a micro-services architecture, an application is built using a combination of loosely coupled and service-specific software containers that communicate using APIs, instead of using a single, tightly coupled monolith of code. This development methodology combined with recent advancements in containerization technologies makes an application easier to enhance, maintain, and scale. However, it is challenging to manage the end-to-end tail latency (e.g 95^{th} percentile latency) of requests flowing through the microservice architecture, which could result in poor user experiences and loss of revenue [32, 46].

Containerized microservices deployed in a public cloud are scaled automatically based on user-specified static thresholds for *per-microservice* resource utilization [1, 2, 6]. However, this places a significant burden on application owners who are concerned about the end-to-end tail latency (e.g 95^{th} percentile latency) [28]. Setting appropriate resource utilization thresholds on various microservices to meet the end-to-end tail latency in such complex distributed system is difficult and error-prone in the absence of accurate performance models.

There are many challenges in modeling the end-to-end tail latency of containerized microservices. First, a microservice architecture is characterized by complex request execution paths spanning many microservices forming a directed acyclic graph (DAG) with complex interactions across the service topology [28, 29, 39]. Second, the tail latency is highly sensitive to any variance in the system which could be related to application, OS or hardware [32]. Third, in a cloud environment where microservices run as containers hosted on a cluster of virtual machines (VMs), application performance can degrade often in unpredictable ways [18, 21, 24, 44].

Traditionally, analytical models based on queuing theory have been widely applied for performance prediction and resource provisioning of monolithic (3-tier) applications [40, 41]. However, such techniques can become intractable when dealing with the scale and complexity of microservice architecture, and the presence of cloud-induced performance variability. Furthermore, analytical modeling is a white-box approach that often requires intrusive instrumentation of application code for workload profiling and expert knowledge about the application structure and data flow between various components [25]. Such approach can be impractical from a cloud provider's perspective since customer applications appear with limited visibility to the cloud providers.

There are black-box modeling approaches that relate observable resource usage metrics [36, 42] or resource allocation metrics [43] with the performance of monolithic applications hosted in virtualized computing environments. More recent studies [19, 26] focused on runtime trace anal-

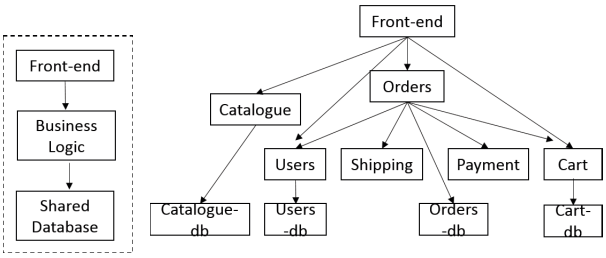
978-1-7281-0218-4/19/$31.00 ©2019 IEEE
DOI 10.1109/IC2E.2019.00034

ysis tools and simulation based approaches to analyze the performance of microservice-based applications. However, none of these works study the impact of cloud induced performance interference on microservice-based applications, and the resulting inaccuracies in performance modeling. In this paper, we observe that the end-to-end tail latency of microservice workflows are highly sensitivity to performance interference in the cloud. Furthermore, we show that the tail latency of microservice workflows can be accurately predicted even in the presence of performance interference, with the help of machine learning and multi-layer data collected from the cloud environment.

In particular, we make the following contributions.

1. We quantify the impact of resource utilization and performance interference experienced by various microservices on the end-to-end tail latency of various request workflows in a web application. Since CPU is a major bottleneck for most web applications, we use CPU utilization as a resource metric in this paper, and focus on the performance interference caused by the contention in shared processor resources such as LLC (last level cache) and memory bandwidth. However, our approach can be easily extended to include other resource metrics.
2. We propose a modeling approach that combines multi-layer data including container-level, VM level and a hardware performance counter based metric, CPI (clock cycles per instruction), to accurately predict end-to-end tail latency in the presence of performance interference in the cloud.
3. We apply several machine learning based modeling techniques, and compare their accuracy in predicting the end-to-end performance for containerized microservices.
4. We demonstrate the feasibility of utilizing the proposed performance models in making efficient resource scaling decisions. For this purpose, we formulate resource scaling of microservices as a constrained nonlinear optimization problem, and solve it to calculate appropriate resource utilization thresholds on various microservices, so that they can be scaled efficiently to meet a performance SLO (service level objective) target.
5. We implement and evaluate the proposed techniques using a representative microservices benchmark, Sock Shop [14], using the NSF Chameleon cloud [3] testbed. The Sock Shop benchmark is containerized with Docker [35] and deployed in a cluster of VMs managed by Kubernetes [8] an open-source container orchestration engine.

The rest of this paper is organized as follows. Section II provides the background on microservice archiecture. Related work are discussed in Section III. Section IV describes the testbed setup and benchmarks used. Section V presents the performance characterization of containerized microservices. Section VI provides the performance modeling approach. Section VII discusses resource scaling optimization based on the proposed models. Section VIII concludes the paper.



(a) Monolith. (b) Microservices.

Figure 1: Monolithic vs microservice architecture.

II. Background on Microservice Architecture

Microservice architecture aims to overcome various limitations of traditional monolithic architecture for software development [10, 22]. Figure 1 illustrates the difference between multi-tier monolithic architecture and microservice architecture in the context of an e-commerce application that takes orders from customers, verifies product catalogue, processes payment and ships orders. In monolithic architecture, the web application is divided into technology-specific tiers such as a frontend web tier for serving web contents, an application tier composed of numerous tightly coupled components for implementing the entire business logic, and a shared database tier for data persistence. A monolithic application is often simple to design. However, in order to update one component, the entire application has to be redeployed. Furthermore, each component within a tier cannot be scaled independently based on its resource requirements. On the other hand, microservice architecture splits the application into many smaller self-contained components, called microservices, that serve specific business functions and communicate with each other via lightweight language-agnostic APIs. Each microservice has its own code and database without any shared component with other services. This facilitates flexibility in application deployment and enhanced scalability since each component of an application can be updated and scaled independently. In essence, microservice architecture is a variant of the Service-Oriented Architecture (SOA) that emphasizes fine-grained services and lightweightness.

III. Related Work

Performance modeling and dynamic resource provisioning of Internet applications has been an important research topic for many years [31, 36, 37, 40, 41, 43, 45]. There are traditional analytical modeling approaches based on queueing theory [40, 41], and hybrid approaches that combine queueing theory with machine learning techniques [38, 45].

Urgaonkar *et al.* [41] designed a dynamic server provisioning technique on multi-tier server clusters. The technique decomposes the per-tier average delay targets to be certain percentages of the end-to-end delay constraint. Singh *et al.* [38] applied k-means clustering algorithm and a $G/G/1$ queuing model to predict the server capacity for a given workload mix. Although these approaches were effective for multi-tier monolithic applications, they can become intractable when dealing with complex microservice architecture in a cloud environment. The complexity introduced by having many moving parts with complex interactions and the presence of cloud-induced performance variability [21, 44] pose significant challenges in modeling the system behavior, identifying critical resource bottlenecks and managing them effectively.

Blackbox modeling techniques have been widely adopted in cluster resource allocation and management [31, 36, 42, 43]. Nguyen *et al.*[36] applied online profiling and polynomial curve fitting to provide a black-box performance model of the applications SLO violation rate for a given resource pressure. Wajahat *et al.* [42] presented an application-agnostic, neural network based auto-scaler for minimizing SLA violations of diverse applications. Wang *et al.* [43] applied fuzzy model predictive control and Lama *et al.* [31] proposed self-adaptive neural fuzzy control techniques for dynamic resource management of monolithic cloud applications. However, these studies do not address the modeling inaccuracies caused by the performance interference in the cloud, and the complexity introduced by microservice architecture.

A few studies have focused on managing the end-to-end performance objectives of large-scale web services and analyzing their complex performance behavior [27, 28, 39]. Guo *et al.* [27] highlighted how the complex interactions between various components of large-scale web services not only lead to sharp degradation in performance, but also trigger cascading behaviors that result in wide-spread application outages. Jalaparti *et al.* [28] presented Kwiken, a framework that decomposes the problem of minimizing latency over a general processing DAG in a large web service into a manageable optimization over individual stages. Suresh *et al.* [28] presented Wisp, a resource management framework that applies a combination of techniques, including estimating local workload models based on measurements of immediate neighborhoods, distributed rate control and metadata propagation to achieve end-to-end throughput and latency objectives in Service-Oriented architectures. These approaches are complimentary to our work as they focus on solutions that need to be adopted at the application layer in the context of cloud computing stack, and requires expert knowledge about the application. On the other hand, our performance modeling approach does not require intrusive instrumentation of application code for profiling or expert knowledge about the data flow between various components.

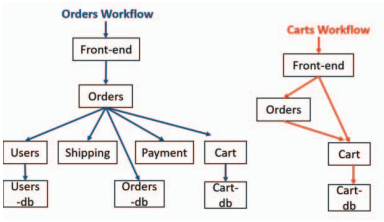


Figure 2: Workflow DAGs.

IV. Platform

A. Experimental Testbed

We set up a cloud prototype testbed, which closely resembles real-world cloud platforms such as Google Kubernetes Engine [6] and Amazon Elastic Container Services [2]. Our testbed consists of a physical layer of bare metal servers, a VM layer built on top of the physical layer and a container layer built on top of VM layer.

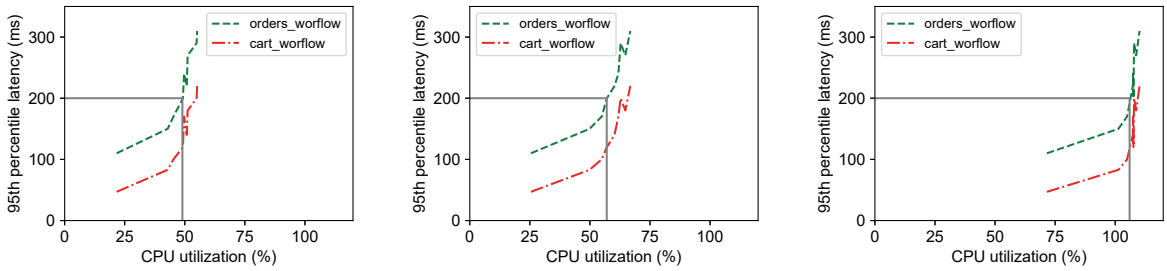
Physical Servers. We used four bare metal servers leased on NSF Chameleon Cloud[3] testbed. Each server was equipped with dual socket Intel Xeon E5-2670 v3 Haswell processors (each with 12 cores @ 2.3GHz) and 128 GiB of RAM. Each server was connected to a Dell switch at 10Gbp, with 40Gbps of bandwidth to the core network from each switch.

VMs. We setup 16 VMs on top of the bare metal servers by using KVM for server virtualization. Each VM was configured with four vCPUs, 8GB Ram and 30GB disk space.

Containers. We setup a 16 VM Kubernetes cluster for container orchestration and management. Docker (version 18.03.1-ce) was used as the container run time engine on each VM. Kubernetes pod networking was set up using the Calico CNI (Container Network Interface) network plugin [11]. We use the term pod and container interchangeably in this paper, since we use a one-container-per-Pod model, which is the most common Kubernetes use case.

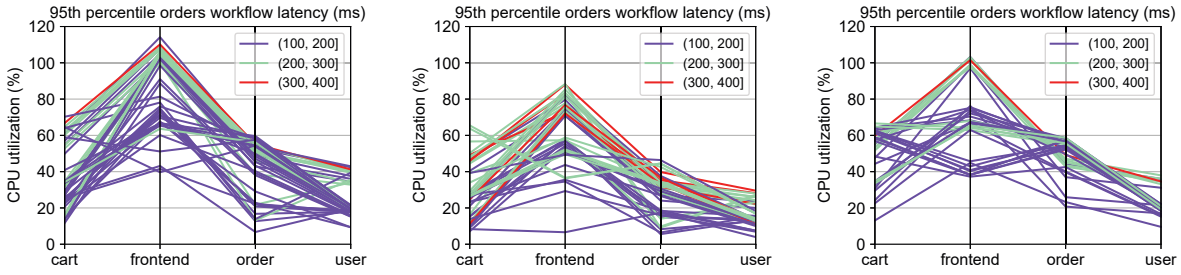
B. Workloads

For performance characterization, we used Sock Shop [14], an open-source microservices benchmark that is particularly tailored for container platforms. Sock Shop emulates an e-commerce website as shown in Figure 1 with the specific aim of aiding the demonstration and testing of existing microservice and cloud-native technologies. A recent study suggests that Sock shop closely reflects how typical microservices applications are currently being developed and delivered into production, as reported by practitioners and industry experts [17]. We used the Locust tool [9] to generate user traffic for the Sock Shop benchmark. The workload traffic is composed of a number of concurrent clients that generate HTTP-based REST API calls to Sock Shop. To create a controlled



(a) CPU utilization of orders microservice. (b) CPU utilization of cart microservice. (c) CPU utilization of frontend microservice.

Figure 3: Impact of CPU utilization on the tail latency of various workflows.



(a) without interference. (b) with interference on cart. (c) with interference on frontend.

Figure 4: Parallel coordinates plot showing the impact of performance interference on the multivariate relationship between CPU utilization and end-to-end tail latency of orders workflow.

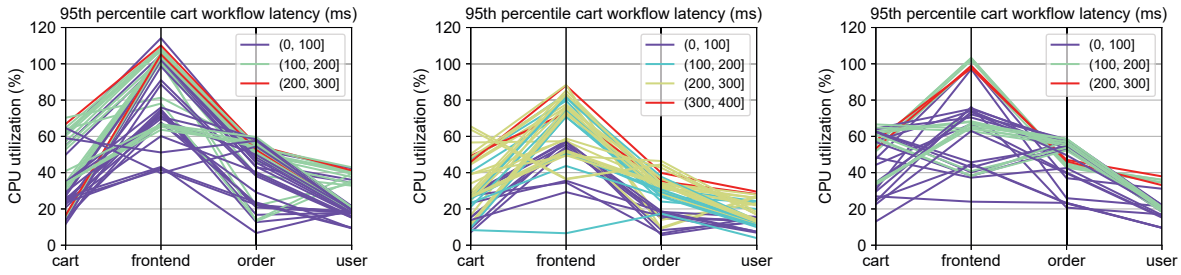
interference workload for our experiments, we used the STREAM Memory Bandwidth benchmark[33]. STREAM is a synthetic benchmark program geared towards measuring memory bandwidth (in MB/s) corresponding to computation rate for simple vector kernels. We run the benchmark inside a docker container and deploy it as a batch job in Kubernetes.

V. Performance Characterization

One of the challenges that complicate performance characterization of a microservice architecture is that request execution workflows can form directed acyclic graph (DAG) structures spanning across many microservices. As a result, the end-to-end latency of a workflow is impacted by the performance behavior of multiple microservices in a complex way. We use the term *workflow* to represent an application-specific group of requests that are associated with a particular API endpoint, which is usually in the form of an HTTP URI. For instance, in case of the Sock Shop benchmark shown in Figure 1, the HTTP URIs for workflows involved with processing orders are [base url: / GET / Orders] and [base url: / POST / Orders]. The exact structure of the DAG for request workflows is often unknown, since it depends on multiple factors such as the APIs invoked at each encountered microservice, the supplied arguments, the content of caches, as well as the use of load balancing along the service graph [39]. We used a visualization and monitoring tool, weavescope [16], to map the DAG structure of orders and cart workflows as shown in Figure 2.

A. End-to-end Tail Latency

First, we analyze the impact of CPU utilization of individual microservices on the end-to-end tail latency of two different workflows viz. orders and cart in the Sock Shop benchmark. For this purpose, we run experiments with various workload intensities by varying the number of concurrent clients in the workload generator from 5 to 50, while setting the total number of generated requests to be 50000. We also vary the number of pods allocated to cart, orders and frontend microservices to include various combination of scaling configurations. The CPU utilization of a particular microservice is measured as the average CPU utilization of all the pods allocated to that microservice. As shown in Figures 3 (a), (b) and (c) the end-to-end tail latency of various workflows have a non-linear relationship with the CPU utilization of individual microservices. We observe that the 95^{th} percentile latency of the two workflows



(a) without interference. (b) with interference on cart. (c) with interference on frontend.

Figure 5: Parallel coordinates plot showing the impact of performance interference on multivariate relationship between CPU utilization and end-to-end tail latency of cart workflow.

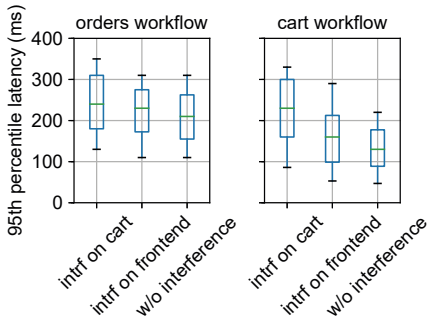


Figure 6: Impact of performance interference on the end-to-end tail latency of various workflows.

increase significantly even at low CPU utilization values of the orders and cart microservices. On the other hand, only high CPU utilization values (>70%) of the frontend microservice has significant impact on the 95^{th} percentile latency. For example, the tail latency of the orders workflow reaches 200 ms at 49%, 57% and 106% CPU utilizations of the orders, cart and frontend microservices respectively.

B. Impact of Performance Interference

Next, we analyze the impact of performance interference in a cloud environment on the multivariate relationship between CPU utilization of various microservices and the end-to-end tail latency of particular request workflows. For the sake of clarity, we present our analysis using top four microservices from the Sock Shop benchmark ranked according to their CPU utilization values. To induce performance interference, we colocate pods running the memory-intensive STREAM [33] benchmark on the VMs that host the pods running cart and frontend microservices respectively. The intensity of interference is fixed by running four pods for each interfering workload. The workload intensities and the scaling configurations for orders, cart and frontend microservices are varied similar to the previous experiment.

As shown in Figures 4 (a), (b) and (c), the end-to-end tail latency of the orders workflow is influenced by the CPU utilization of multiple microservices. However, their multivariate relationship changes significantly depending on the performance interference experienced by various microservices. For example, in the case of no interference, the 95^{th} percentile latency of orders workflow is greater than 300 ms when the CPU utilization measured at cart, frontend, orders and user microservices are 67%, 110%, 55% and 41% respectively. However, similar tail latency of orders workflow was observed at much lower CPU utilization values when one of the microservices experienced performance interference. Similar results were obtained for the cart workflow as shown in Figures 5 (a), (b) and (c). This implies that the CPU utilization of microservices measured at the pod level are insufficient in accurately predicting the end-to-end tail latency of various workflows.

Figure 6 shows the distribution of the 95^{th} percentile latency of various workflows under three different scenarios, i.e with interference on cart, interference on frontend and without interference. The variation in the latency observed within each case is mainly due to the varying workload intensities in these experiments. On average the performance degradation observed by orders and cart workflows due to interference on cart microservice are 22% and 79% respectively. On the other hand, the average performance degradation of the two workflows due to interference on frontend microservice are 6% and 18% respectively. These results demonstrate the complex interplay between performance interference, inter-service performance dependency and the end-to-end tail latency of various workflows.

VI. Performance Modeling with Machine Learning

In this section, we present our approach to address the challenges of predicting the end-to-end tail latency of complex workflows in a microservice architecture in the face of diverse performance interference patterns. Our approach

combines the resource usage metrics at the container/pod level with VM level resource usage and hardware performance counter values to construct machine learning (ML) based performance models for individual workflows. Our modeling approach does not rely on any expert application knowledge. Hence, it can be easily extended to fit the need of diverse applications.

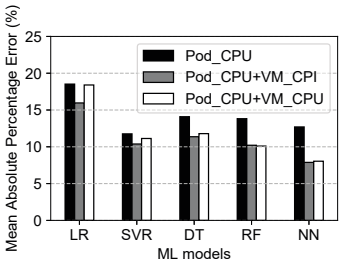
A. Data Collection

In this paper, we use CPU utilization as a resource metric for the microservices since CPU is a major resource bottleneck in most web applications. We use *docker stats* [4] to measure pod level CPU utilization. To capture the impact of performance interference due to the contention of processor resources, such as the last level cache (LLC) and memory bandwidth, we utilize the CPU utilization and CPI metric associated with the VMs that host the various microservices as pods. We use the *virt top* [15] tool to measure VM level CPU utilization. CPI is measured on a per cgroup basis by using the perf event [23] tool and each cgroup is mapped to a VM. For data collection, we conduct extensive experiments on our cloud prototype testbed by varying the number of concurrent clients, and the performance interference levels experienced by different microservices in the Sock Shop benchmark. We also vary the number of pods allocated to the microservices. For each experiment, we measure the end-to-end tail latency of various workflows as reported by the Locust [9] tool. The collected data is used to train our machine learning based performance models.

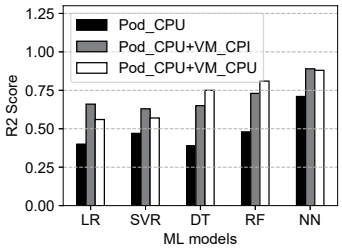
B. Machine Learning Models

We build performance models for predicting the end-to-end tail latency of each microservice workflow by applying various machine learning (ML) techniques including Linear Regression (LR), Support Vector Regression (SVR), Decision Tree (DT), Random Forrest (RF) and a deep Neural Network (NN) based regression (more specifically a multi-layer perceptron with multiple hidden layers). The ML models are built and trained by using scikit-learn [12], a machine learning library in Python.

Feature Selection. The input features of our ML models include the number of concurrent clients, pod-level resource metrics and VM-level resource metrics. The pod-level metrics include the average CPU utilization of load-balanced pods for each microservice. The VM-level metrics include the CPU utilization or the CPI of VMs that host the pods. To reduce our feature space and avoid potential overfitting issues, we apply a popular feature selection technique called stability selection [34]. In particular, we use scikit-learn [12] library's *randomized lasso* technique, which works by subsampling the training data and computing a Lasso estimate where the penalty of a random subset of coefficients has been scaled. By performing this operation several times, the method assigns high scores to features that are repeatedly selected across randomizations. The features selected for the orders workflow are the number of concurrent clients, pod-level CPU utilization of the microservices including front-end, orders, users, shipping, payment, cart, users-db, orders-db, cart-db, and the CPU utilization or CPI of the VMs that host these microservices. Similarly, the features selected for the cart workflow are the number of concurrent clients,

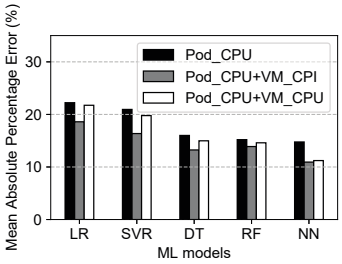


(a) Mean absolute percentage error.

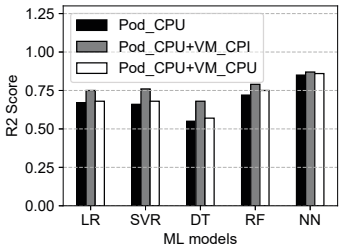


(b) R^2 Score.

Figure 7: Prediction accuracy of various ML models for orders.

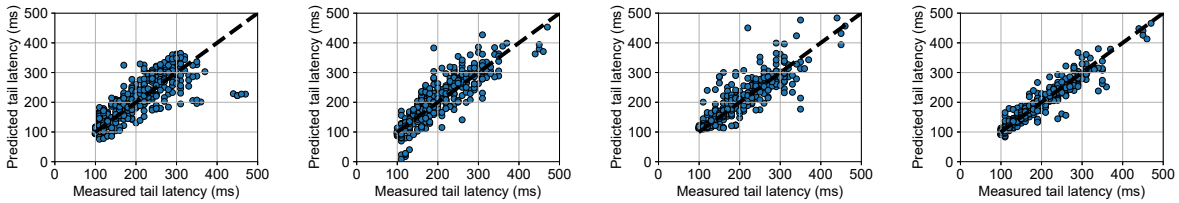


(a) Mean absolute percentage error.



(b) R^2 Score.

Figure 8: Prediction accuracy of various ML models for cart.



(a) Linear regression with Pod CPU. (b) Linear regression with Pod CPU and VM CPI. (c) Neural network with Pod CPU. (d) Neural network with Pod CPU and VM CPI.

Figure 9: Cross-validated predictions of tail latency in orders workflow.

Table I: Optimal number of neurons in the three hidden layers of NN models for orders and cart workflow.

Input Feature \ Workflow	orders	cart
Pod_CPU	(6,3,5)	(8,5,6)
Pod_CPU+VM_CPU	(4,6,3)	(3,6,8)
Pod_CPU+VM_CPI	(9,6,4)	(5,7,5)

the pod-level CPU utilization of the microservices including front-end, orders, cart, cart-db, and the CPU utilization or CPI of the VMs that host these microservices.

Hyper-parameters. The hyper-parameters of each model is set to the default values provided by scikit-learn. We observe that the prediction accuracy of the deep NN model is highly sensitive to the number of hidden layers and the size (number of neurons) in each hidden layer. Hence, we tuned these parameters through an exhaustive search for various combinations of input feature space and the targeted workflow for the prediction of end-to-end tail latency. The optimal number of hidden layers for our NN model is three, and the optimal number of neurons in these three hidden layers is summarized in Table I.

C. Prediction Accuracy

In this section, we evaluate the prediction accuracy of various ML models (LR, SVR, DT, RF, NN) and three modeling approaches. First, the Pod_CPU approach includes pod-level CPU utilization metrics in the input feature space. Second, the Pod_CPU+VM_CPU approach includes both pod-level and VM-level CPU utilization metrics. Third, the Pod_CPU+VM_CPI approach includes pod-level CPU utilization and VM-level CPI metrics in the input feature space. The models are evaluated with 10-fold cross validation on the collected dataset. As a result, 90% of data is used for training, 10% of data is used for testing in each of the 10 iterations of cross-validation. We utilize commonly used metrics such as the mean absolute percentage error (MAPE) and the coefficient of determination, R^2. MAPE is calculated as $\frac{1}{n}\sum_{i=1}^{n}\left|\frac{y-\hat{y}}{y}\right|$ where y and $\hat{y}$ are the measured and predicted values of the end-to-end tail latency respectively. R^2 is a statistical measure of how well the regression predictions approximate the real data points. An R^2 of 1 indicates that the regression predictions perfectly fit the data.

Figures 7 (a) and (b) show that, compared to the Pod_CPU based modeling approach, Pod_CPU+VM_CPU and Pod_CPU+VM_CPI approaches achieve significant improvement in the prediction accuracy of each ML model for the orders workflow. This is because VM-level CPU utilization can capture inter-pod CPU contention within a VM. Furthermore, VM-level CPI metric can capture the contention of shared processor resources between multiple pods within a VM as well as across VMs. Such inter-VM resource contention may arise when the concerned VMs are colocated in the same physical machine. The improvement in the prediction accuracy in terms of MAPE due to Pod_CPU+VM_CPU and Pod_CPU+VM_CPI approaches are up to 36% and 38% respectively. The largest improvement is observed in case of the NN model. We also observe that the NN model outperforms all other models in prediction accuracy since the Neural Network is a universal function approximator. On the other hand, the LR model shows the worst prediction accuracy. This is because a linear regression model can not capture the non-linearity of tail latency. Overall, we observed similar results in the latency prediction of cart workflow as shown in Figure 8.

Figure 9 plots the cross-validated predictions vs. the measured values of end-to-end tail latency of the orders workflow in order to graphically illustrate the different R^2 values for the LR and NN models. Theoretically, if a model could explain 100% of the variance in the observed data, the predicted values would always equal the measured values and, therefore, all the data points would fall on the fitted regression line. The more variance that is accounted for by the regression model the closer the data points will fall to the fitted regression line. The proportion of variance accounted for by the LR model with Pod_CPU , LR model with Pod_CPU+VM_CPI, NN model with Pod_CPU and NN model with Pod_CPU+VM_CPI approaches are 42%, 66%, 71% and 89% respectively.

VII. Optimization for Resource Scaling

Although existing cloud platforms [1, 2, 5, 6] provide mechanisms for auto-scaling microservices, they expect ap-

Table II: Notation used in Resource Scaling Optimization Problem

Symbol	Description
S_j	Set of microservices relevant to workflow j
SLO_j^{target}	Tail latency target of workflow j
x_i	Average pod-level CPU utilization in microservice i
x	A vector of average pod-level CPU utilizations of various microservices relevant to the target workflow
$r_j(x)$	Predicted tail latency of workflow j

plication owners to specify thresholds for various microservice load metrics to enable auto-scaling features. For example, the auto-scaling feature [7] in Kubernetes determines the allocation of containers/pods to a microservice by using the formula:

$$desiredReplicas = \lceil currentReplicas * \frac{currentMetricValue}{desiredMetricValue} \rceil \quad (1)$$

If the desiredMetricValue (threshold) is specified as an average CPU utilization of 50% for a particular microservice, and the current average CPU utilization is 100%, then the number of pods allocated to that microservice will be doubled. Furthermore, any scaling is performed only if the ratio of currentMetricValue and desiredMetricValue drops below 0.9 or increases above 1.1 (10% tolerance by default). It is challenging and burdensome for application owners to determine the resource utilization thresholds for various microservices in order to meet the application's end-to-end performance target. Setting inappropriate thresholds may lead to overprovisioning or underprovisioning of resources. We propose that cloud platforms should automatically determine these thresholds based on user-provided performance SLO targets. For this purpose, we study the feasibility of utilizing the proposed performance models in making efficient resource scaling decisions by formulating a constrained nonlinear optimization problem.

A) Problem Formulation. Consider that the performance SLO target in terms of the end-to-end tail latency for a workflow is specified. For a given workload condition, we aim to find the highest resource utilization values of the relevant microservices, at which the given SLO targets will not be violated. These optimal utilization values can be calculated periodically and set as the thresholds (desiredMetricValue) for making resource scaling decisions. These thresholds will help in determining which microservices should be scaled, and how many pods should be allocated to each microservice based on Equation 1. This approach aims to avoid resource overprovisioning while providing performance guarantee to the given workflow.

We formulate the optimization problem as follows:

$$\max \quad \sum_{i \in S_j} x_i \quad (2)$$

$$\text{s.t.} \quad r_j(x) \leq SLO_j^{target} \quad (3)$$

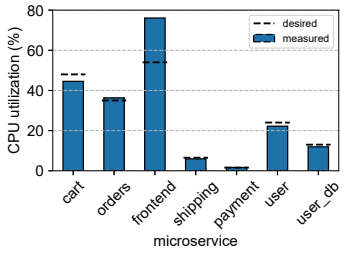
$$x = (x_i)_{i \in S_j} \quad (4)$$

where, the symbol notations are described in Table II. The objective function in Equation 2 aims to maximize the pod-level resource usage i.e the sum of average CPU utilization in the set of microservices that are relevant to the target workflow. The relevance of a microservice to a workflow can be determined either by analyzing the workflow DAG, or through machine learning based feature selection as described in Section VI-B. Consider that $r_j(x)$ is the tail latency predicted by machine learning model for workflow j. The inequality constraint in Equation 3 ensures that the SLO target of workflow j will not be violated. The optimization problem is nonlinear since the workflow tail latency $r_j(x)$ included in the constraint Equation 3 has a nonlinear relationship with the average CPU utilization of various microservices.

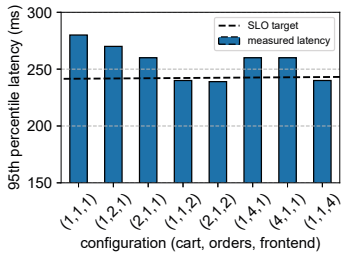
In the formulation of the optimization problem, application-layer metrics (e.g number of concurrent clients), VM-level CPU utilization and CPI metrics are not included as variables, although the tail latency prediction $r_j(x)$ depends on these metrics as well. Instead, the values of these metrics are fixed according to their observed values at the time of solving the optimization problem, and are treated as constants for that instance of optimization. As a result, the solutions to the optimization problem will only include pod-level CPU utilization values, which can be directly used as thresholds for making resource scaling decisions. This allows the resource scaling mechanism to be practical and simple to implement.

B) Solution. We apply a non-linear optimization technique, trust-region interior point method [13, 20], to solve this problem. This optimization technique provides two main benefits. First, it is efficient for large scale problems. Second, the gradient of the constraint function which is required for optimization, can be approximated through finite difference methods in this optimization technique [13]. This property is desirable since the machine learning models for workflow tail latency are blackbox functions, whose gradient can not be directly calculated.

C) Feasibility Study. As a case study, we apply the optimization technique to calculate the desired CPU utilization (thresholds) for various relevant microservices, when a workload of 30 concurrent clients is applied to the SockShop benchmark, and a performance SLO target of 240 ms is specified for the 95th percentile latency of orders workflow. For this optimization, we utilize our Neural Network model for orders workflow with pod-level CPU utilization, VM-level CPI metrics and the number of concurrent clients as the input



(a) Current vs desired average CPU utilization of various microservices. Here, one pod is allocated to each microservice.



(b) Tail latency of orders workflow for various resource scaling configurations. The configuration suggested by the optimization of CPU utilization thresholds is (1,1,2) i.e one pod for cart, one pod for orders and two pods for frontend. All other microservices are provisioned with one pod.

Figure 10: Optimization of CPU utilization thresholds for efficient resource scaling with a workload of 30 concurrent clients, and SLO target 240 ms for 95th percentile latency of orders workflow.

features. Figure 10 (a) compares the current (measured) CPU utilization of the microservices relevant to orders workflow and their desired CPU utilization values, when only one pod is allocated to each microservice. Based on Equation 1, the optimal resource scaling option is to allocate an additional pod to the frontend microservice. As shown in Figure 10 (b), we validate the optimality of this resource scaling option by comparing the tail latency of orders workflow for various possible resource scaling configurations. We observe that the resource scaling configuration suggested by our optimization technique is able to meet the performance SLO target while allocating minimum number of pods in total.

VIII. Conclusions And Future Work

We present the performance characterization and modeling of containerized microservices in the cloud. Our modeling approach utilizes machine learning and multi-layer data collected from the cloud environment to predict the end-to-end tail latency of microservice workflows even in the presence cloud induced performance interference. We also demonstrate the feasibility of utilizing the proposed models in making efficient resource scaling decisions. We envision that our performance modeling and resource scaling optimization approach can enable cloud platforms to automatically scale microservice-based applications based on user-provided performance SLO targets. This will remove the burden of determining resource utilization thresholds for numerous microservices from the cloud users, which is prevalent in existing cloud platforms. In future, we will extend our work to include diverse microservice-based applications with different resource bottlenecks. We will also evaluate the effectiveness of the proposed resource scaling system in the face of dynamic workloads.

Acknowledgment

Results presented in this paper were obtained using the Chameleon testbed supported by the National Science Foundation. The research is partially supported by NSF CREST Grant HRD-1736209. We thank the anonymous reviewers for their many suggestions for improving this paper. In particular we thank our shepherd, Prof. Maarten van Steen.

References

[1] Amazon elastic container service. https://aws.amazon.com/ecs/.
[2] Amazon elastic container service for kubernetes. https://aws.amazon.com/eks/.
[3] Chameleon: A configurable experimental environment for large-scale cloud research. https://www.chameleoncloud.org.
[4] Docker stats. https://docs.docker.com/engine/reference/commandline/stats/.
[5] Google app engine flexible environment. https://cloud.google.com/appengine/docs/flexible/.
[6] Google Kubernetes engine. https://cloud.google.com/kubernetes-engine/.
[7] Kubernetes horizontal autoscaling. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/#algorithm-details.
[8] Kubernetes: Production-grade container orchestration. https://kubernetes.io/.
[9] Locust: An open source load testing tool. https://locust.io.
[10] Microservices: an application revolution powered by the cloud. https://azure.microsoft.com/en-us/blog/microservices-an-application-revolution-powered-by-the-cloud/.
[11] Project calico. https://www.projectcalico.org/.
[12] Scikit-learn: Machine learning in python. http://scikit-learn.org/stable/.
[13] Scipy optimization library. https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html.
[14] Sockshop microservice demo application. https://microservices-demo.github.io.
[15] virt-top. https://linux.die.net/man/1/virt-top.

[16] Weave scope. https://www.weave.works/docs/scope/latest/introducing/.

[17] C. M. Aderaldo, N. C. Mendona, C. Pahl, and P. Jamshidi. Benchmark requirements for microservices architecture research. In *IEEE/ACM 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE)*, 2017.

[18] A. Balalaie, A. Heydarnoori, and P. Jamshidi. Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Software*, 33(3), 2016.

[19] S. Barakat. Monitoring and analysis of microservices performance. *Journal of Computer Science and Control Systems*, 10:19–22, 05 2017.

[20] R. H. Byrd, M. E. Hribar, and J. Nocedal. An interior point algorithm for large-scale nonlinear programming. *SIAM J. on Optimization*, 9(4):877–900, Apr. 1999.

[21] X. Chen, L. Rupprecht, R. Osman, P. Pietzuch, F. Franciosi, and W. Knottenbelt. Cloudscope: Diagnosing and managing performance interference in multi-tenant clouds. In *2015 IEEE 23rd International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2015.

[22] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina. Microservices: yesterday, today, and tomorrow. In *Present and Ulterior Software Engineering*, pages 195–216. Springer, 2017.

[23] S. Eranian. perfmon2: the hardware-based performance monitoring interface for linux. http://perfmon2.sourceforge.net/.

[24] M. Fazio, A. Celesti, R. Ranjan, C. Liu, L. Chen, and M. Villari. Open issues in scheduling microservices in the cloud. *IEEE Cloud Computing*, 3(5):81–88, 2016.

[25] I. Giannakopoulos, D. Tsoumakos, and N. Koziris. Towards an adaptive, fully automated performance modeling methodology for cloud applications. In *IEEE International Conference on Cloud Engineering (IC2E)*, 2018.

[26] M. Gribaudo, M. Iacono, and D. Manini. Performance evaluation of massively distributed microservices based applications. In *European Council for Modelling and Simulation (ECMS)*, 2017.

[27] Z. Guo, S. McDirmid, M. Yang, L. Zhuang, P. Zhang, Y. Luo, T. Bergan, M. Musuvathi, Z. Zhang, and L. Zhou. Failure recovery: When the cure is worse than the disease. In *Presented as part of the 14th Workshop on Hot Topics in Operating Systems*, Santa Ana Pueblo, NM, 2013. USENIX.

[28] V. Jalaparti, P. Bodik, S. Kandula, I. Menache, M. Rybalkin, and C. Yan. Speeding up distributed request-response workflows. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, 2013.

[29] D. Jiang, G. Pierre, and C.-H. Chi. Autonomous resource provisioning for multi-service web applications. In *Proceedings of the 19th ACM International Conference on World wide web (WWW)*, 2010.

[30] G. Kakivaya, L. Xun, R. Hasha, S. B. Ahsan, T. Pfleiger, R. Sinha, A. Gupta, M. Tarta, M. Fussell, V. Modi, M. Mohsin, R. Kong, A. Ahuja, O. Platon, A. Wun, M. Snider, C. Daniel, D. Mastrian, Y. Li, A. Rao, V. Kidambi, R. Wang, A. Ram, S. Shivaprakash, R. Nair, A. Warwick, B. S. Narasimman, M. Lin, J. Chen, A. B. Mhatre, P. Subbarayalu, M. Coskun, and I. Gupta. Service fabric: A distributed platform for building microservices in the cloud. In *Proceedings of the Thirteenth EuroSys Conference*, 2018.

[31] P. Lama and X. Zhou. Autonomic provisioning with self-adaptive neural fuzzy control for percentile-based delay guarantee. *ACM Transactions on Autonomous and Adaptive Systems, 31 pages, under 2nd reviewing after revision*, 2011.

[32] J. Li, N. K. Sharma, D. R. K. Ports, and S. D. Gribble. Tales of the tail: Hardware, os, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2014.

[33] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE computer society technical committee on computer architecture (TCCA) newsletter*, 2(19–25), 1995.

[34] N. Meinshausen and P. Bhlmann. Stability selection. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 72(4):417 – 473, 8 2010.

[35] D. Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

[36] H. Nguyen, Z. Shen, X. Gu, S. Subbiah, and J. Wilkes. AGILE: Elastic distributed resource scaling for infrastructure-as-a-service. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC)*, 2013.

[37] J. Rao and C.-Z. Xu. Online capacity identification of multi-tier Websites using hardware performance counters. *IEEE Trans. on Parallel and Distributed Systems*, 2009.

[38] R. Singh, U. Sharma, E. Cecchet, and P. Shenoy. Autonomic mix-aware provisioning for non-stationary data center workloads. In *Proc. IEEE Int'l Conf. on Autonomic Computing (ICAC)*, pages 21–30, 2010.

[39] L. Suresh, P. Bodik, I. Menache, M. Canini, and F. Ciucu. Distributed resource management across process boundaries. In *Proceedings of the 2017 Symposium on Cloud Computing-SoCC'17*. ACM Press, 2017.

[40] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi. An analytical model for multi-tier internet services and its applications. In *Proceedings*

of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, 2005.

[41] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood. Agile dynamic provisioning of multi-tier internet applications. *ACM Trans. Auton. Adapt. Syst.*, 3(1), Mar. 2008.

[42] M. Wajahat, A. Gandhi, A. Karve, and A. Kochut. Using machine learning for black-box autoscaling. In *2016 Seventh International Green and Sustainable Computing Conference (IGSC)*, 2016.

[43] L. Wang, J. Xu, H. A. Duran-Limon, and M. Zhao. Qos-driven cloud resource management through fuzzy model predictive control. In *IEEE International Conference on Autonomic Computing (ICAC)*, 2015.

[44] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey. Bobtail: Avoiding long tails in the cloud. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013.

[45] Q. Zhang, L. Cherkasova, and E. Smirni. A regression-based analytic model for dynamic resource provisioning of multi-tier Internet applications. In *Proc. IEEE Int'l Conference on Autonomic Computing (ICAC)*, 2007.

[46] Y. Zhang, D. Meisner, J. Mars, and L. Tang. Treadmill: Attributing the source of tail latency through precise load testing and statistical inference. In *ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016.