Robust Resource Scaling of Containerized Microservices with Probabilistic Machine learning

Peng Kang
Department of Computer Science
University of Texas at San Antonio
San Antonio, Texas, USA
peng.kang@utsa.edu

Palden Lama
Department of Computer Science
University of Texas at San Antonio
San Antonio, Texas, USA
palden.lama@utsa.edu

***Abstract*—Large-scale web services are increasingly being built with many small modular components (microservices), which can be deployed, updated and scaled seamlessly. These microservices are packaged to run in a lightweight isolated execution environment (containers) and deployed on computing resources rented from cloud providers. However, the complex interactions and the contention of shared hardware resources in cloud data centers pose significant challenges in managing web service performance. In this paper, we present RScale, a robust resource scaling system that provides end-to-end performance guarantee for containerized microservices deployed in the cloud. RScale employs a probabilistic machine learning-based performance model, which can quickly adapt to changing system dynamics and directly provide confidence bounds in the predictions with minimal overhead. It leverages multi-layered data collected from container-level resource usage metrics and virtual machine-level hardware performance counter metrics to capture changing resource demands in the presence of multi-tenant performance interference. We implemented and evaluated RScale 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, Robot Shop, demonstrate the superior prediction accuracy and adaptiveness of our modeling approach compared to popular machine learning techniques. RScale meets the performance SLO (service-level-objective) targets for various microservice workflows even in the presence of multi-tenant performance interference and changing system dynamics.**

***Keywords*-Microservices, Performance modeling, Machine Learning.**

I. Introduction

Large-scale web services (e.g Netflix, Uber, Spotify) are increasingly adopting cloud-native design patterns such as microservices and containers [25]. In a microservice architecture, an application is built using a combination of loosely coupled and service-specific components that communicate via lightweight APIs, instead of using a single, tightly coupled monolith of code as shown in Fig. 1. Despite its many benefits [6], transitioning from monolithic to microservice architecture creates significant challenges for organizations. One of the primary challenges lies in managing the end-to-end tail latency (e.g 95^{th} percentile latency) of requests flowing through the microservice architecture, which results in poor user experiences and loss of revenue [12, 31]. Determining effective resource management policies to meet the end-to-end performance objectives in such a complex distributed system is difficult and error-prone [9, 22].

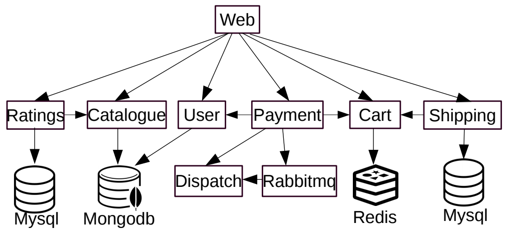


Figure 1: Microservice architecture of Robot Shop.

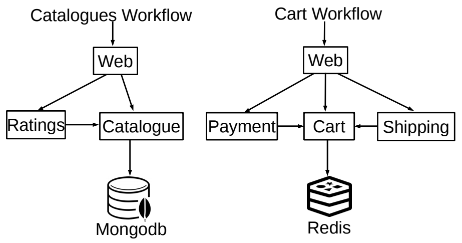


Figure 2: Workflow DAGs.

Performance modeling and dynamic resource provisioning of multi-tier monolithic applications have been well-studied during the past decade [11, 18, 23, 24, 30]. However, microservice architecture and its complex interplay with the cloud environment introduce new challenges in quality of service aware resource management. Request execution workflows in a microservice architecture can span numerous microservices forming a directed acyclic graph (DAG) with complex interactions across the service topology [9, 10, 22]. Fig. 2 shows the DAG structure of workflows involved with processing product Catalogues and shopping Carts in the Robot Shop benchmark. Furthermore, 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 [1, 3, 7, 28]. Tail latency is highly sensitive to any variance in the system which could be related to application, OS or hardware [12]. Together these issues pose significant challenges in modeling the system behavior and managing them effectively.

Recent works have developed machine learning models that relate observable resource usage metrics [21, 26] or resource allocation metrics [27] with application performance. As the model complexity grows (e.g from linear regression to deep neural networks), they can capture complex and non-linear system behavior more accurately as shown in our previous work [17]. However, these models are slow in adapting to changing dynamics of the cloud environment such as drastic variations in workload characteristics and performance interference patterns. Furthermore, these models are deterministic, i.e they provide point estimates only without expressing uncertainty associated with the prediction, which can be critical for providing robust performance guarantee. There are techniques such as bootstrapping [2], which can calculate the confidence bounds on each prediction made by a deterministic model. However, such techniques are associated with large overheads, which make them impractical for dynamic resource management.

In this paper, we present RScale, a robust resource scaling system that provides end-to-end performance guarantee for containerized microservices deployed in the cloud. RScale employs an adaptive performance model based on a probabilistic machine learning technique, Gaussian Process (GP) Regression [19]. In contrast to deterministic models, GP regression based performance models are highly adaptive to the changing dynamics of the cloud environment due to its data-efficiency (the ability to learn more from less data) and lazy learning technique, which delays the generalization of training data until the next inference interval, thus enabling local approximation of the target function. Furthermore, they directly provide confidence bounds on each prediction to assist in making better resource scaling decisions with minimal overhead. To capture the complex interplay between end-to-end tail latency of microservice workflows, inter-service performance dependencies, and cloud-induced performance variability, RScale leverages data collected from multiple layers of the cloud environment including container-level resource usage metrics and hardware performance counter based metrics associated with underlying VMs.

Our key contributions are as follows.

- We develop a robust performance model based on a probabilistic machine learning approach, Gaussian Process regression, which can predict the end-to-end tail latency of microservice workflows and efficiently estimate the confidence bounds of each prediction.
- We show how our performance models can adapt to changing system dynamics such as time-varying performance interference patterns arising from the contention of underlying shared resources in a cloud environment.
- We design a robust resource scaling system to manage the end-to-end tail latency of microservice workflows by explicitly incorporating predictive uncertainty obtained from probabilistic performance models into the resource allocation problem. This approach ensures system robustness and reduces the effects of modeling errors on the ability to meet end-to-end performance objectives.
- We implement and evaluate RScale on NSF Cloud's Chameleon testbed [1] using a representative microservices benchmark, Robot Shop[2], which is containerized with Docker and deployed in a cluster of VMs managed by Kubernetes [3] an open-source container orchestration engine. Experimental evaluation demonstrates how RScale can provide robust performance guarantee for containerized microservices in a cloud environment.

We outline the rest of this paper as follows. Section II presents related work. Section III elaborates RScale's key design and implementation details. Section IV gives the testbed setup and experimental results. Conclusion is in Section V.

II. Related Work

Dynamic resource provisioning. Dynamic resource provisioning of Internet applications has been an important research topic for many years [11, 18, 21, 23, 24, 27, 30]. There are traditional analytical modeling approaches based on queueing theory [23, 24], and hybrid approaches that combine queueing theory with machine learning techniques [20, 30]. Urgaonkar et al. [24] 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. Based on a G/G/1 queueing model, per-tier server provisioning is executed for the per-tier delay guarantees. Singh et al. [20] proposed a dynamic provisioning technique that handles both the non-stationarity in the workload and changes in request volumes when allocating server capacity in data centers. It is based on the 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 [3, 28] pose significant challenges in modeling the system behavior, identifying critical resource bottlenecks and managing them effectively.

Machine Learning Based Systems. Machine learning techniques have been widely adopted in cluster resource allocation and management [4, 8, 21, 17, 26, 27, 29]. Nguyen et al. [21] applied polynomial curve fitting to obtain a black-box performance model of an application's SLO violation rate for a given resource pressure. They used the model to

[1]https://www.chameleoncloud.org

[2]https://github.com/instana/robot-shop

[3]https://kubernetes.io/

dynamically adjust the number of VMs assigned to a cloud application. Wajahat et al. [26] presented an application-agnostic, neural network-based auto-scaler for minimizing SLA violations of diverse applications. Delimitrou et al. [4] used collaborative filtering to estimate the impact of resource scale-out (more servers) and scale-up (more resources per server) on application performance. Iqbal et al. [8] used supervised learning to predict the workload arrival rate and combined the observed response time of the last k intervals to make resource provisioning decisions. Yang, Z. [29] proposed a model-based reinforcement learning for microservice resource allocation. However, these works have mainly focused on applying deterministic machine learning models, which provide point estimates only without expressing uncertainty associated with the prediction. Recent work using Bayesian Neural Network [32] is able to estimate the uncertainty in predictions for deep regression models. However, it is challenging to quickly adapt such models to drastic variations in workload and performance interference patterns. In contrast, our work presents a robust resource scaling system that utilizes an adaptive performance model and directly incorporates predictive uncertainty into the resource allocation problem.

III. RSCALE DESIGN AND IMPLEMENTATION

In this section, we present the key design and implementation of RScale, a resource scaling system that aims to provide a robust performance guarantee for containerized microservices.

A. RScale Architecture

Fig. 3 shows the interaction between various components of RScale. The performance monitor is responsible for periodically collecting data from the application, container, and VM layers at each physical server that hosts the VMs belonging to an application owner (cloud user). The mapping of containers to VMs and VMs to the physical server can be obtained via commonly available APIs provided by the container orchestration engine (Kubernetes) and cloud management software (OpenStack) respectively. Probabilistic performance models based on Gaussian Process Regression for respective microservice workflows are adapted based on observed data samples. Finally, the auto-scalar component is responsible for adding or removing containers for various microservices to meet the end-to-end tail latency targets for microservice workflows, based on the optimization problem formulated in Section III-B.

B. Performance Modeling with Gaussian Process

GP regression is a non-parametric probabilistic modeling technique. In contrast to a deterministic ML-based performance modeling approach, which aims to approximate a particular function that can represent the relationship between independent variables and the predicted variable,

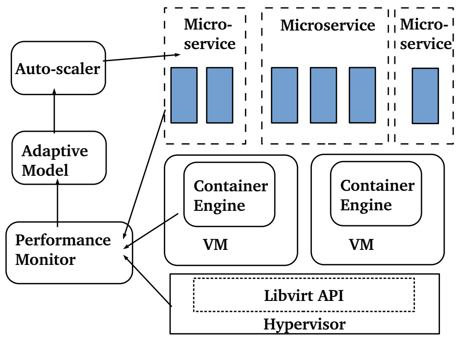


Figure 3: RScale Architecture.

GP regression utilizes the concept of Gaussian Process and Bayesian inference to find a distribution over all the possible functions that are consistent with the observed data. Hence, this probabilistic modeling approach can directly provide confidence bounds on its predictions, which is critical for making robust resource management decisions in an uncertain cloud environment.

Model Formulation. As with all Bayesian methods, GP regression begins with a prior distribution of functions, and updates this as data points are observed, producing the posterior distribution over functions. The prior distribution is defined by a GP, a collection of random variables, any finite number of which have (consistent) joint Gaussian distributions. For every input x there is an associated random variable $f(x)$, which is the value of the stochastic function f at that location. In this context of modeling containerized microservices, x represents a vector of performance metrics collected from the application-layer, container-layer, and VM-layer at a particular sampling interval. $f(x)$ represents the distribution of the end-to-end tail latency of a particular workflow for the given set of input data denoted by x. GP assumes that $p(f(x_1), \cdot\cdot\cdot, f(x_N))$ is jointly Gaussian, with a mean function $m(x)$ and covariance function $\Sigma(x) = k(x_i, x_j)$, where k is a positive definite kernel. The key idea is that if x_i and x_j are deemed by the kernel to be similar, then we expect the output of the function at those points to be similar, too. A commonly used kernel function is squared-exponential kernel given by $k_{SE}(x, x') = \sigma^2 exp\left(-\frac{(x-x')^2}{2l^2}\right)$. It calculates the squared distance between points and converts it into a measure of similarity, controlled by hyperparameters σ and l.

Let f be the function values that have been observed (training set output) for a training input set X, and let f_* be a set of function values that need to be predicted (test set output) corresponding to test input set X_*. The joint distribution of the function values can be expressed as:

$$\begin{bmatrix} f \\ f_* \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} m(X) \\ m(X_*) \end{bmatrix}, \begin{bmatrix} K(X,X) & K(X,X_*) \\ K(X_*,X) & K(X_*,X_*) \end{bmatrix}\right) \quad (1)$$

where $m(X)$ and $m(X_*)$ are the training and testing mean functions. Usually, for notational simplicity the prior means

Table I: Kernel Selection using log marginal likelihood.

kernel	log marginal likelihood	
	Catalogue	Cart
RBF	-3.153E+03	-2.153E+03
RationalQuadratic	-2.995E+03	-2.052E+03
Matern	-3.036E+03	-2.086E+03
DotProduct	-1.988E+15	-1.706E+15
RBF + RationalQuadratic	-2.971E+03	-2.027E+03
RBF + Matern	-2.974E+03	-2.027E+03
RationalQuadratic + Matern	-2.977E+03	-2.027E+03

are assumed to be constant and zero. This is also consistent with the practice of data normalization in machine learning. If there are n training points and n_* test points then $K(X, X_*)$ denotes the $n \times n_*$ matrix of the covariances evaluated at all pairs of training and test points, and similarly for the other entries $K(X, X)$, $K(X_*, X_*)$ and $K(X_*, X)$. Based on Bayesian inference, it is possible to get the conditional distribution of f_* given f as shown in Eq. (2).

$$\begin{aligned} f_* \mid f \sim N\left(\bar{m}, \bar{\Sigma}\right) \sim N(K\left(X_*, X\right) K\left(X, X\right)^{-1} f, \\ K\left(X_*, X_*\right) - K\left(X_*, X\right)_* K\left(X, X\right)^{-1} K\left(X, X_*\right)) \end{aligned} \tag{2}$$

This is the posterior distribution over target functions for a specific set of test cases X_*. The mean of this distribution, $\bar{m}$, will be used to predict the end-to-end tail latency of a microservice workflow. The standard deviation $\bar{\sigma}$ derived through Cholesky decomposition [5] of the variance $\bar{\Sigma}$, will be used to get the confidence bound on the prediction.

Kernel Selection. One of the key design for the generalization properties of a GP model is the choice of the kernel function. A multitude of possible families of kernel functions exists such as Matern kernel, Rational quadratic kernel, etc. New kernel can be derived by additive and multiplicative combination of existing kernels[15]. We systematically compare the log marginal likelihood (LML) of various kernels on our training data to select the best kernel for the GP models in Table I. We use the sum kernel RBF+RationalQuadratic for our GP model corresponding to the highest LML value for both Catalogue and Cart workflows.

GP Model Fitting. For a set of n observations, the hyperparameters associated with the kernel functions are estimated and optimized during the fitting of GP model. This is done by maximizing the log marginal likelihood, $\log p(f|X, \theta)$ i.e the probability of the data given the hyperparameters denoted by θ, as shown in Eq. (3).

$$\begin{aligned} L = \log p(f|X, \theta) = -\frac{1}{2} \log |K\left(X, X\right)| \\ -\frac{1}{2} f^T (K(X, X))^{-1} f - \frac{n}{2} \log(2\pi) \end{aligned} \tag{3}$$

C. Robust Resource Scaling.

Although existing cloud platforms[4] provide mechanisms for auto-scaling microservices, they expect application owners to specify thresholds for various microservices load

[4]https://aws.amazon.com/ecs/

Table II: Notation for Resource Scaling Optimization.

Symbol	Description
S	Set of microservices relevant to the target workflows
R	Set of container resource types(CPU, network)
SLO_j^{target}	tail latency target of workflow j
x_{ir}	Average utilization of resource r in microsevices i
$\bar{m}_j(x)$	Posterior mean function of workflow j from Eq.2
$\bar{\sigma}_j(x)$	Posterior standard deviation of workflow j from Eq.2
S_j	Set of microservices related to workflow j

metrics to enable auto-scaling features. For example, the auto-scaling feature in Kubernetes determines the allocation of containers to a microservice by using the formula:

$$desiredReplicas = \max_r\left(\left\lceil currentReplicas * \frac{currentMetric_r}{desiredMetric_r} \right\rceil \right) \tag{4}$$

If the $desiredMetric_r$ is specified as an average CPU utilization of 40% for and the $currentMetric_r$ for CPU utilization is 50% for a particular microservice, the number of containers allocated to that microservice will be doubled since the ratio is 1.2 (50%/40%). If multiple metrics are specified, this calculation is done for each metric, and then the largest of the desired replica counts is chosen. Any scaling is executed only if the ratio of $currentMetric_r$ and $desiredMetric_r$ drops below 0.9 or increases above 1.1 (10% tolerance). 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 now describe how RScale can enable cloud platforms to automatically determine these thresholds based on user-provided performance SLO targets.

1) Problem Formation: Consider that SLO targets in terms of the end-to-end tail latency for a set of workflows are 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 utilization values are calculated periodically and set as the thresholds (desiredMetricValue) for making resource scaling decisions. These dynamic thresholds help in determining which microservices should be scaled, and how many containers should be allocated to each microservice based on Eq. (4). This approach aims to avoid resource overprovisioning while providing a performance guarantee to the given workflows.

We formulate resource scaling as a constrained optimization problem as follows:

$$max \sum_{i \epsilon S} \sum_{r \epsilon R} x_{ir} \tag{5}$$

$$s.t. \forall j : \bar{m}_j\left(x\right) + \kappa \bar{\sigma}_j\left(x\right) \leq SLO_j^{target} \tag{6}$$

$$\forall i, r : x_{ir} \geq 0 \tag{7}$$

Algorithm 1: Online Adaptation and Resource Scaling

```
Require: collect training dataset D containing N samples;
  for $T = 1$ to $infinity$ do
      1: collect multi-layered data at sampling interval $T$;
      2: Insert new sample into dataset D;
      3: Remove the oldest sample from dataset D;
      4: Update model (Fit model to dataset D);
      5: Solve the optimization problem given by Eqs. (5) - (8);
      6: Calculate desiredReplicas from Eq. (4);
      7: Scale microservices
  end for
```

$$x = ((x_{ir})_{r \epsilon R})_{i \epsilon S_j} \tag{8}$$

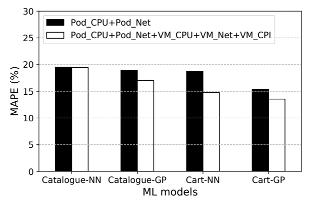
The symbol notations are described in Table II. The objective function in Eq. (5) aims to maximize the container-layer resource usage i.e the sum of average utilization of CPU and network resources in the set of microservices that are relevant to the target workflows. The relevance of a microservice to a workflow can be determined either by analyzing the workflow DAG or through machine learning-based feature selection[14].

Consider that $\bar{m}_j(x)$ and $\bar{\sigma}_j(x)$ are the posterior mean function and standard deviation function provided by the GP regression model for workflow j. Depending on the tunable parameter κ, the inequality constraint in Eq. (6) ensures with certain confidence bound that the SLO target of workflow j will not be violated. For example, if $\kappa = 2$, there is a 95% probability that the tail latency of workflow j will less than the SLO target. This is because point-wise mean plus and minus two times the standard deviation corresponds to the 95% confidence region in a Gaussian distribution.

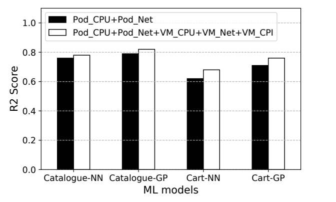
2) Solution: We apply a non-linear optimization technique, the interior-point method [16], to solve the resource scaling optimization problem. In the formulation of the proposed optimization problem, application-layer metrics (e.g workload intensity) is not included as variables, although $\bar{m}_j(x)$ and $\bar{\sigma}_j(x)$ depend on this metric as well. Instead, the values of these metrics will be fixed according to their observed values at the time of solving the optimization problem and treated as constants for the instance. Therefore, the resource scaling decision can be made directly based on the container-layer resource usage thresholds as determined by the optimization. This allows the resource scaling mechanism to be practical and manageable.

D. Putting It All Together

Algorithm 1 describes the overall operation of RScale system, including performance monitoring, online model adaptation, and resource scaling. In order to adapt the performance model in the face of changing system dynamics, RScale maintains a sliding window of data samples collected from the cloud environment. The use of the sliding window allows the GP models to forget old observations and utilize recent observations. At each sampling interval, RScale updates the GP models by fitting the models to the sliding window dataset (See GP Regression Model fitting in Section III-B). Then, it solves the resource scaling optimization problem given by Eqs. (5) - (8). During this process, the GP models are iteratively evaluated to obtain the optimal values of desired resource utilization (desiredMetrics). Then the desiredReplicas for individual microservices are calculated by Eq. (4). Finally, the microservices are scaled by adding or removing containers allocated to them.



(a) Prediction error.



(b) R^2 Score.

Figure 4: Prediction accuracy of Catalogue and Cart workflow models.

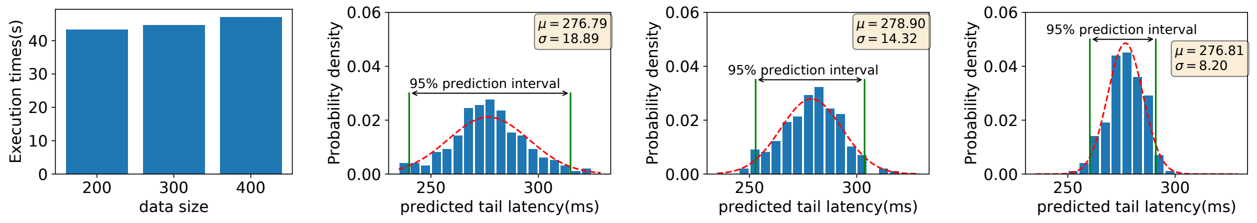
In this paper, we focus on coarse-grained resource scaling (scale-out approach) at the granularity of containers running on cloud VMs. We do not scale-up containers and VMs. This makes our approach transparently enforceable in existing cloud systems. We also make the assumption that when the resources available in a cluster of VMs are insufficient to accommodate the scaling of microservices, new VMs can be added to the cluster. Similarly, when VMs are underutilized they can be removed from the cluster. This assumption is in alignment with the VM auto-scaling feature of existing cloud platforms. Hence, avoiding overprovisioning of microservices based on the proposed techniques will contribute to resource efficiency in terms of the number of VMs used.

IV. Evaluation

A. Experimental TestBed

We set up a cloud prototype testbed closely resembling real-world cloud platforms, such as Google Kubernetes Engine and Amazon Elastic Container Services. The testbed includes four bare metal servers leased on NSF Chameleon Cloud. Each server was equipped with dual-socket Intel Xeon E5-2670 vs Haswell processors(each with 12 cores @ 2.3Ghz), 128 GB of RAM and connected by 10Gbps Ethernet. 16 VMs were hosted on the server cluster by using KVM for server virtualization. Each VM was configured with four vCPUs, 8 GB RAM and 30GB disk space. A Kubernetes cluster using these 16 VMs was built for container orchestration and management. Docker (Version 18.06.2-ce) was used as the container runtime engine.

As a representative microservice benchmark, we used Robot Shop which emulates an e-commerce website. The



(a) Bootstrap overhead(NN). (b) Data size is 200. (c) Data size is 300. (d) Data size is 400.

Figure 5: Distribution of predicted tail latency for Cart workflow obtained using the Bootstrap method and its associated overheads for various data sizes.

Locust tool[5] was used to generate user traffic composed of a number of concurrent clients that generate HTTP-based REST API calls. To introduce performance interference, memory-intensive STREAM[13] benchmark and the Iperf Network Performance benchmark[6] were executed on randomly select VMs in the testbed. The intensity of interference was varied by changing the number of containers for each interfering workload.

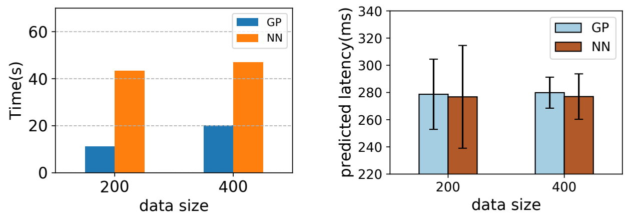
B. Machine Learning models.

We compare our GP regression based probabilistic model with a multi-layer perceptron based Neural Network (NN), which is a representative deterministic model with superior prediction accuracy. The input features of each model include the number of concurrent clients, pod-level resource metrics (the average CPU utilization and network throughput of load-balanced pods for each microservice), and VM-level resource metrics (the CPU utilization, network throughput and CPI of VMs that host the pods). We use the scikit-learn library's randomized lasso[15] technique to reduce our feature space and avoid potential over-fitting issues [14]. The hyperparameters of NN models are tuned by scikit-learn's GridSearchCV tuner. The prediction accuracy of NN model is highly sensitive to the number of hidden layers and the 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 (5, 3, 3) for Catalogue workflow and (3, 4, 6) for Cart workflow.

C. Prediction Accuracy.

We evaluate the prediction accuracy of NN and GP models with two different input feature space. The first input feature space includes only pod-level resource metrics and the second input feature space uses both pod-level and VM-level metrics. We also evaluate the models with 10-fold cross-validation on the collected dataset and compared the mean absolute percentage error (MAPE) and the coefficient of determination, R^2. An R^2 of 1 indicates that the regression predictions perfectly fit the data. Figs. 4 show that the GP regression models for the Catalogue and Cart workflows provided better prediction accuracy than NN models. To collect the training data, we conduct extensive experiments on our testbed by varying the number of concurrent clients and the performance interference levels experienced by different microservices in the Robot Shop benchmark. We also vary the number of containers (pods) allocated to the microservices, and measure the end-to-end tail latency of various workflows as reported by the Locust tool.

[5]https://locust.io
[6]https://iperf.fr/

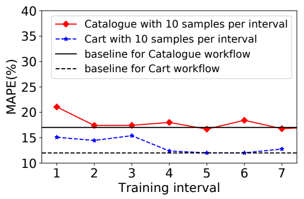


(a) Overheads in GP and NN models with bootstrapping. (b) Predicted tail latency with 95% prediction interval for NN and GP models.

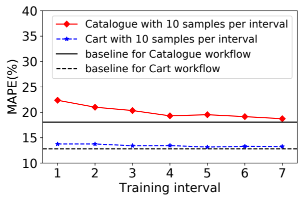
Figure 6: Comparing predictive uncertainty in GP and NN models (with bootstrapping) for Cart workflow. Measured tail latency is 280ms.

D. Estimation of Predictive Uncertainty.

Estimation of uncertainty for NN. Making robust resource management decisions in the face of changing dynamics of the cloud environment requires the ability to quantify predictive uncertainty of performance models. However, deterministic performance models, such as popular machine learning (ML) based models applied in recent works [21, 26, 29], provide point estimates only without confidence bounds. Although bootstrapping methods can calculate the confidence bounds on each prediction made by a deterministic model, they incur large overheads. Here, we evaluate the bootstrap technique using NN model to estimate the uncertainty in the prediction of tail latency for Cart workflow, when facing a workload of 55 concurrent clients and interference on Web microservice from STREAM benchmark. The NN model is trained with 80% of randomly sampled



(a) Case 1: Initial model trained with dataset that includes interference from memory intensive workload. Model is adapted online and tested with dataset that includes interference from network intensive workload.



(b) Case 2: Initial model trained with dataset that includes interference in Web and Cart microservices. Model is adapted online and tested with dataset that includes interference in Catalogue microservice.

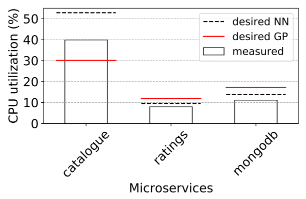
Figure 7: Prediction errors during online adaptation of GP model to changing interference pattern.

training data and used to predict the tail latency. This process is repeated for 200 iterations to obtain a distribution of predicted tail latency. The 95% confidence interval obtained from this distribution provides the predictive uncertainty of the model. We found that using 200 iterations for bootstrap provided the optimal estimation accuracy for a given data size, and iterations greater than 200 only increased the overhead. Due to space limitation, we only show the results obtained by using 200 bootstrap iterations for various data sizes as shown in Fig. 5. We observe that the predictive uncertainty reduces with increasing data size. However, there is a significant overhead of bootstrapping for all data sizes.

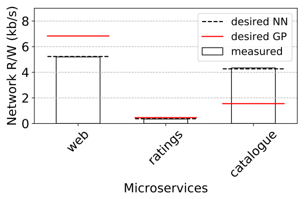
Comparison of uncertainty between NN and GP. Fig. 6 compares the predictive uncertainty and associated estimation overheads in case of GP and NN models. In contrast to NN models which require bootstrapping to estimate predictive uncertainty, GP models directly provide confidence bounds on each prediction. Fig. 6a compares the time taken by these two methods to estimate the 95% prediction interval. We observe that when the data size is 200, NN model with bootstrapping incurs 4X more overhead than GP model in estimating predictive uncertainty. This is despite the fact that we include the time taken for model fitting as a part of GP model's estimation overhead. However, model fitting may not be needed every time for GP to make a prediction. This can further reduce its overhead of uncertainty estimation. Fig. 6b shows that GP model is able to achieve smaller uncertainty in its prediction than the NN model for various data sizes.

E. *Model Adaptiveness.*

To evaluate the adaptiveness of GP regression models, we conduct experiments for two cases that require online adaptation of performance models in the face of varying interference patterns. As shown in Fig. 7, the prediction error of the GP models converges quickly and stays close to the the baseline value as the model is adapted over the training intervals. Here, the baseline is the prediction error measured

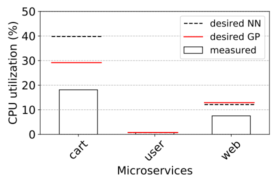


(a) Current vs Desired CPU utilization of various microservices using NN and GP models.

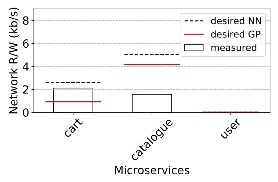


(b) Current vs Desired Network throughput of various microservices using NN and GP models.

Figure 8: Optimization of resource utilization thresholds for efficient resource scaling with a workload of 30 concurrent clients, and SLO target 170 ms for 95th percentile latency of Catalogue workflow.



(a) Current vs Desired CPU utilization of various microservices using NN and GP models.



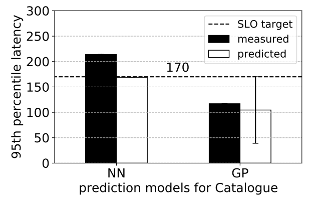
(b) Current vs Desired Network throughput of various microservices using NN and GP models.

Figure 9: Optimization of resource utilization thresholds for efficient resource scaling with a workload of 30 concurrent clients, and SLO target 220 ms for 95th percentile latency of Cart workflow.

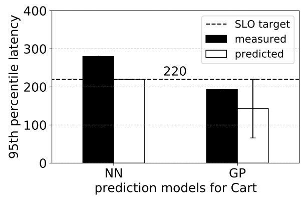
in the offline evaluation of the model with the offline training dataset. The adaptiveness of a GP regression model can be attributed to its ability to learn more from less data and lazy learning technique, which delays the generalization of training data until the next inference interval, thus enabling local approximation of the target function.

F. *Performance Guarantee.*

We now evaluate the effectiveness of RScale's resource scaling optimization in meeting the end-to-end tail latency targets of microservice workflows. For this purpose, we specify performance SLO targets of 170 ms and 220 ms for Catalogue and Cart workflows respectively, when a workload of 30 concurrent clients is applied to the Robot Shop benchmark. Furthermore, we compare the impact of the resource scaling optimization based on NN and GP regression models respectively. Fig. 8 compares the current (measured) resource utilization of the microservices relevant to Catalogue workflow and their desired resource utilization values, when only one pod is allocated to each microservice. In case of NN model based optimization, current CPU utilization of microservices including Catalogue, Ratings and MongoDB are less than their desired CPU utilization. Furthermore, the ratios of the current network read/write throughput and desired network read/write throughput for



(a) Catalogue workflow.



(b) Cart workflow.

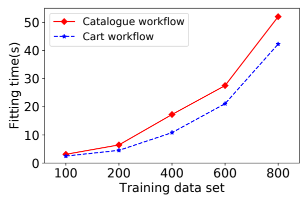
Figure 10: 95th percentile latency of microservice workflows with resource scaling optimization based on NN and GP.

microservices including Web, rating, and Catalogue are within the tolerance range from 0.9 to 1.1. Therefore, NN model based resource optimization does not suggest any change in resource allocation. Whereas in case of GP model based optimization, the ratio of current CPU utilization and desired CPU utilization and the ratio of current Network Read/Write throughput and desired Network Read/Write throughput for Catalogue microservice are 1.3 and 2.8 respectively. Hence, the desired number of replicas for Catalogue is three according to Eq. 4. The desired replicas stay the same as current replicas for Ratings, MongoDB, and Web. Thus, the optimal resource scaling option for GP model based optimization is to add two additional pods to the Catalogue microservice. Similarly, Fig. 9 shows the optimal resource scaling option relevant to Cart workflow for NN is to allocate an additional pod to Web and another new pod to Cart and for GP is to allocate an additional pod to Web and three new pods to Cart.

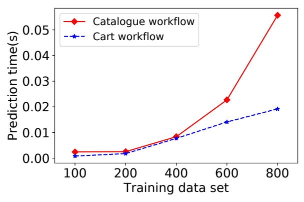
Finally, combining the two different scaling options for Catalogue workflow and Cart workflow, the optimal resource scaling option to satisfy both SLO targets at the same time according to NN, is to add one pod to Web, Cart, and Catalogue respectively. On the other hand, the optimal solution according to GP is to add one pod to Web and three pods to Cart and Catalogue respectively. Fig. 10 shows that when the microservices are scaled based on the NN's optimization model, the measured 95^{th} percentile latency for Catalogue and Cart workflows violates the SLO target. This is due to the fact that resource scaling optimization was performed without considering the uncertainty associated with performance prediction. On the other hand, GP model based resource scaling is able to meet the SLO targets since the 95% confidence interval of prediction is directly incorporated into the resource scaling optimization problem.

G. Overhead Analysis

A potential drawback of using GP regression lies in its computational complexity. A straightforward implementation of GP regression requires inversion of the covariance matrix, with a memory complexity of $\Theta\left(n^2\right)$ and a computational complexity of $\Theta\left(n^3\right)$. However, this is feasible even on a desktop computer for data sizes of n up to a few thousand. As GP models are non-parametric, they employ a lazy learning technique that delays the generalization of training data until the inference (prediction) is made. It means that they need to consider the training data each time they make a prediction. Hence, the size of training data determines both the time taken to fit GP models and the time taken to make a prediction as shown in Fig. 11.



(a) Fitting time.



(b) Prediction time.

Figure 11: Impact of training data size on GP.

Fig. 12 shows the impact of training data size on the time taken to solve the resource scaling optimization problem given by Eqs. (5) - (8). We observe that the optimization problem takes more time to converge in the case of Cart workflow since it has a much wider range of tail latency (100ms to 1000ms), compared to that of the Catalogue workflow (50ms to 250ms) for a given range of resource utilization values. Based on our overhead analysis, we limit the training data size to 200 samples so that the overhead of data collection(Line 1-3 in algorithm 1), the overhead of model fitting, prediction, and optimization(Line 4-6 in algorithm 1), and the overhead of scaling (Line 7 in algorithm 1) remain smaller than the data sampling interval of 30 seconds.

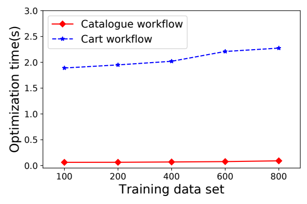


Figure 12: Optimization time for GP.

H. Provisioning under varying load intensity

We now illustrate the provisioning of resources under a dynamic workload shown in Fig. 13. We set the SLO target of Cart workflow in terms of the 95^{th} percentile latency to be 220ms. Figs. 14 (a) and (b) show the end-to-end tail latency of Cart workflow without interference and with interference respectively under a dynamic workload. In Fig. 14a, RScale is able to meet the SLO target for the Cart workflow by scaling the appropriate microservices at sampling interval 2 and 4 in response to increasing workload intensity. When the workload intensity decreases, pods are removed from appropriate microservices at sam-

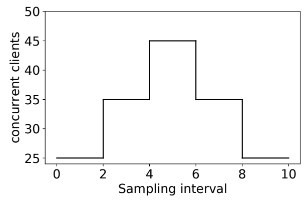
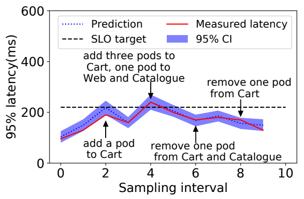
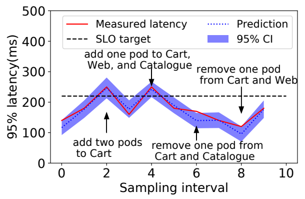


Figure 13: Dynamic workload for Cart workflow.

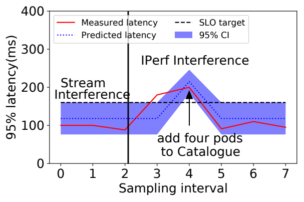


(a) 95th percentile latency without interference.

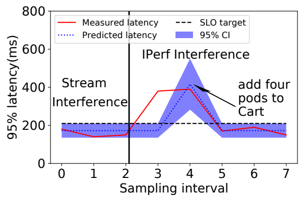


(b) 95th percentile latency with interference from iPerf.

Figure 14: Resource provisioning under dynamic workload for Cart workflow.



(a) Catalogue workflow.



(b) Cart workflow.

Figure 15: Resource provisioning under varying interference.

pling interval 6 and 8 in order to avoid overprovisioning of resources.

Fig. 14b shows similar results in the presence of performance interference from iPerf colocated with the microservices relevant to Cart workflow. We observe that RScale allocates more resources and decreases fewer resources when performance interference exists in the system.

I. Provisioning under varying interference.

We evaluate the effectiveness of RScale in providing performance guarantee to Catalogue and Cart workflows under varying interference patterns. In this experiment, the initial GP models are trained with dataset including interference from memory-intensive STREAM workload. As shown in Fig. 15, the interfering workload is changed from STREAM to network-intensive iPerf after sampling interval 2. As a result, the measured tail latency exceeds the SLO target at sampling interval 3. RScale is able to adapt the model quickly and scale the appropriate microservices at sampling interval 4. Finally, the measured tail latency meets the SLO target at sampling interval 5.

V. Conclusion

In this paper, we designed and implemented RScale, a robust resource scaling system that provides end-to-end performance guarantee for containerized microservices deployed in multi-tenant cloud. Unlike existing works, RScale ensures system robustness by explicitly incorporating predictive uncertainty estimated from probabilistic performance models into the resource allocation problem. Experimental evaluation demonstrates that RScale can meet the end-to-end tail latency of microservice workflows even in the face of multi-tenant performance interference and changing system dynamics. In future, we plan to develop interference-aware container scheduling technique that aims to minimize the resource contention experienced by the containerized microservices when they are placed on a cluster of VMs. We will further extend our work to include diverse microservice-based applications with different resource bottlenecks.

Acknowledgment

Results presented in this paper were obtained using the Chameleon testbed supported by the National Science Foundation. The research is supported by NSF CNS 1911012 grant.

References

[1] A. Balalaie, A. Heydarnoori, and P. Jamshidi. Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Software*, 33(3):42–52, May 2016.

[2] J. Brownlee. *Statistical Methods for Machine Learning: Discover how to Transform Data into Knowledge with Python*. Machine Learning Mastery, 2018.

[3] X. Chen, L. Rupprecht, R. Osman, P. Pietzuch, F. Franciosi, and W. Knottenbelt. Cloudscope: Diagnosing and managing performance interference in multi-tenant clouds. In *Proc. of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS '15, page 164–173, 2015.

[4] C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. *SIGARCH Comput. Archit. News*, 42(1):127–144, Feb. 2014.

[5] D. Dereniowski and M. Kubale. Cholesky factorization of matrices in parallel and ranking of graphs. In R. Wyrzykowski, J. Dongarra, M. Paprzycki, and J. Waśniewski, editors, *Parallel Processing and Applied Mathematics*, pages 985–992, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[6] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina. *Microservices: Yesterday, Today, and Tomorrow*, pages 195–216. Springer International Publishing, Cham, 2017.

[7] M. Fazio, A. Celesti, R. Ranjan, C. Liu, L. Chen, and M. Villari. Open issues in scheduling microservices in the cloud. *IEEE Cloud Computing*, 3:81–88, 09 2016.

[8] W. Iqbal, A. Erradi, M. Abdullah, and A. Mahmood. Predictive auto-scaling of multi-tier applications using performance varying cloud resources. *IEEE Transactions on Cloud Computing*, pages 1–1, 2019.

[9] V. Jalaparti, P. Bodik, S. Kandula, I. Menache, M. Rybalkin, and C. Yan. Speeding up distributed request-response workflows. *SIGCOMM Comput. Commun. Rev.*, 43(4):219–230, Aug. 2013.

[10] D. Jiang, G. Pierre, and C.-H. Chi. Autonomous resource provisioning for multi-service web applica-

tions. In *19th International World-Wide Web Conference (WWW)*, pages 471–480, 01 2010.

[11] 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*, 8:9:1–, 07 2013.

[12] 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 *Proc. of the ACM Symposium on Cloud Computing*, SOCC '14, page 1–14, 2014.

[13] J. McCalpin. Memory bandwidth and machine balance in high performance computers. *IEEE Technical Committee on Computer Architecture Newsletter*, pages 19–25, 12 1995.

[14] N. Meinshausen and P. Bühlmann. Stability selection. *Journal of the Royal Statistical Society Series B*, 72:417–473, 09 2010.

[15] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[16] F. A. Potra and S. J. Wright. Interior-point methods. *Journal of Computational and Applied Mathematics*, 124(1):281 – 302, 2000. Numerical Analysis 2000. Vol. IV: Optimization and Nonlinear Equations.

[17] J. Rahman and P. Lama. Predicting the end-to-end tail latency of containerized microservices in the cloud. In *Proc. of the IEEE International Conference on Cloud Engineering (IC2E)*, pages 200–210, 06 2019.

[18] J. Rao and C.-Z. Xu. Online measurement of the capacity of multi-tier websites using hardware performance counters. In *Proc. of the IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2008.

[19] C. Rasmussen and C. Williams. *Gaussian Processes for Machine Learning*. MIT Press, 01 2005.

[20] R. Singh, U. Sharma, E. Cecchet, and P. Shenoy. Autonomic mix-aware provisioning for non-stationary data center workloads. In *Proc. of the 7th International Conference on Autonomic Computing, ICAC '10*, 01 2010.

[21] S. Subbiah, j. wilkes, X. Gu, H. Nguyen, and Z. Shen. Agile: Elastic distributed resource scaling for infrastructure-as-a-service. In *Proc. of 10th International Conference on Autonomic Computing (ICAC 13)*, 2013.

[22] L. Suresh, P. Bodik, I. Menache, M. Canini, and F. Ciucu. Distributed resource management across process boundaries. In *Proc. of ACM Symposium on Cloud Computing (SoCC)*, 2017.

[23] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi. An analytical model for multi-tier internet services and its applications. *Sigmetrics Performance Evaluation Review*, 33:291–302, 06 2005.

[24] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood. Agile dynamic provisioning of multi-tier internet applications. *ACM Transactions on Autonomous and Adaptive Systems*, 3, 03 2008.

[25] M. Villamizar, O. Garcés, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano, and M. Lang. Infrastructure cost comparison of running web applications in the cloud using aws lambda and monolithic and microservice architectures. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 179–182, 2016.

[26] M. Wajahat, A. Gandhi, A. Karve, and A. Kochut. Using machine learning for black-box autoscaling. In *Proc. of IEEE International Green and Sustainable Computing Conference (IGSC)*, pages 1–8, 01 2016.

[27] L. Wang, J. Xu, H. Duran-Limon, and M. Zhao. Qos-driven cloud resource management through fuzzy model predictive control. In *Proc. of IEEE International Conference on Autonomic Computing*, pages 81–90, 07 2015.

[28] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey. Bobtail: Avoiding long tails in the cloud. In *Proc. of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[29] Z. Yang, P. Nguyen, H. Jin, and K. Nahrstedt. Miras: Model-based reinforcement learning for microservice resource allocation over scientific workflows. In *Proc. of IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2019.

[30] Q. Zhang, L. Cherkasova, and E. Smirni. A regression-based analytic model for dynamic resource provisioning of multi-tier applications. In *Fourth International Conference on Autonomic Computing, ICAC'07*, pages 27–27, 07 2007.

[31] Y. Zhang, D. Meisner, J. Mars, and L. Tang. Treadmill: Attributing the source of tail latency through precise load testing and statistical inference. *SIGARCH Comput. Archit. News*, 44(3):456–468, June 2016.

[32] L. Zhu and N. Laptev. Deep and confident prediction for time series at uber. In *IEEE International Conference on Data Mining Workshops*. IEEE, 2017.