IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 25 November 2022, accepted 31 December 2022, date of publication 3 January 2023, date of current version 9 January 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3234021

RESEARCH ARTICLE

Proactive Random-Forest Autoscaler for Microservice Resource Allocation

LAMEES M. AL QASSEM, THANOS STOURAITIS, (Life Fellow, IEEE), ERNESTO DAMIANI, (Senior Member, IEEE), AND IBRAHIM (ABE) M. ELFADEL, (Senior Member, IEEE)

Center for Cyber-Physical Systems (C2PS), Khalifa University, Abu Dhabi, United Arab Emirates
Department of Electrical Engineering and Computer Science, Khalifa University, Abu Dhabi, United Arab Emirates

Corresponding author: Ibrahim (Abe) M. Elfadel (ibrahim.elfadel@ku.ac.ae)

ABSTRACT Cloud service providers have been shifting their workloads to microservices to take advantage of their modularity, flexibility, agility, and scalability. However, numerous obstacles remain to achieving the most out of microservice deployments, especially in terms of a Quality of Service (QoS). One possible approach to overcoming these obstacles is to perform autoscaling, which is the ability of cloud infrastructure and services to scale themselves up or down by changing their resource pool. There are two major categories of autoscaling: reactive and proactive. In *reactive autoscaling*, a feedback loop based on current workload resource usage is implemented to guide resource scaling. One disadvantage of reactive autoscaling is that it may result in inconsistencies between workload demand and resource allocation. In *proactive autoscaling*, a prediction model is used to guide the future allocation of resources according to current workload metrics. In this paper, a novel *proactive autoscaling* method is introduced where a two-state, machine-learning Random Forest (RF) model is designed to forecast the future CPU and memory utilization values required by the microservice workload. These predicted values are then used to adjust the resource pool both vertically (hardware resources) and horizontally (microservice replicas). The RF proactive autoscaler has been implemented on a home-grown, open-source microservice prototyping platform and verified using real-world workloads. The experiments show that the RF proactive autoscaler outperforms state-of-the-art ones in terms of allocated resources and latency. The increase in the utilization of allocated resources can reach 90% and the improvement in end-to-end latency, measured by the 95^{th} percentile, can reach 95%.

INDEX TERMS Microservices, autoscalers, resource allocation, resource utilization, machine learning, random forest.

I. INTRODUCTION

The objective of autoscaling cloud resources is to dynamically modify allocated resources to meet Quality of Service (QoS) requirements. Resource allocation is the process of distributing available computing resources economically among competing services [1]. In other words, cloud resource allocation involves resource search, selection, provisioning, and management. In the absence of proper resource allocation management, services will starve. This issue is resolved by dynamic resource provisioning using for instance, autoscaling which allows service providers to manage the resources dynamically according to customer demand. An important trend in cloud computing is the containerization of cloud resources. Another important trend is the continued diversification of cloud workloads. Both of these trends pose stiff challenges to the application of autoscaling techniques. For example, in a diverse workload environment, sudden and abrupt fluctuations in the nature of the workload may cause the autoscaler to alternate between various resource configurations, but with a time lag that may result in a violation of the Service Level Agreement (SLA). An important SLA metric is tail latency, which is the long latency that clients experience with low probability. Although tail latency occurs infrequently, it is still an important metric, as it typically impacts users with the highest number of requests. In a

The associate editor coordinating the review of this manuscript and approving it for publication was Wanqing Zhao.

microservice architecture, the computation of tail latency may be quite complex, especially in use cases involving the serial chaining of services. Such case arises when a front-end service calls multiple back-end services, each one of which could, in turn, call other services. A time lag in any of these calls increases the end-to-end tail latency of the overall service. Autoscaling may help in addressing such worst-case scenarios by determining the amount of resources needed to eliminate local time lags in a service chain. Resource utilization is another important metric for cloud providers. It facilitates the fulfillment of QoS requirements of customers and providers [2].

There are two major categories of autoscaling: reactive and proactive. The state-of-the-art autoscalers [3], [4], [5] are reactive rule-based systems that scale resources in *reaction* to specific events. For example, Kubernetes (K8s) [5] is a well-known microservice autoscaler that tracks the average CPU utilization and moves containers in or out to reach the CPU and utilization thresholds specified by the end user. One major shortcoming of these reactive autoscalers is that they are burst-oblivious, that is, they do not consider workload bursts when determining and provisioning suitable resources. In particular, they cannot identify online surges in dynamic workloads and prevent service performance from degradation as a result. Furthermore, reactive autoscalers suffer from time lags between observations and responses, especially when workloads undergo drastic changes, resulting in non-optimal resource configurations.

The second autoscaling category is that of proactive methods whose goal is to anticipate resource demands and meet them in time so as to eliminate delayed responses. The core of the proactive autoscaler is a prediction model of future resource demands based on collected workload metrics. Recent trends have been to develop predictive autoscaling solutions based on modern machine learning (ML) methods [6], [7], [8], [9]. As pointed out in [10], the challenges posed by microservice management are best met by ML models that can estimate end-to-end latency, compute the likelihood of a QoS violation, and predict microservice resource demands. The autoscaler then uses the outputs of the ML models to optimize resource utilization under a given set of QoS specifications.

In this paper, we present a new proactive autoscaler with a random forest (RF) [11] demand prediction model. The RF model forecasts both CPU and memory usage under various cloud workloads. The proactive autoscaler leverages the outputs of the RF model to appropriately allocate future resources while continuously monitoring workload metrics. The novel contributions of this paper are as follows:

1) We develop a robust multi-variate predictive ML model based on the RF paradigm of ensemble machine learning. The model considers all essential measures of resource metrics for running microservices to predict their CPU and memory usage accurately.
2) The model is workload-independent and can accurately predict resource metrics of different workload types, including those that contain activity bursts.
3) We show how our proactive model can adapt to changes in the workload and be robust to outliers. We also use the tools of explainable Artificial Intelligence (AI) to trace the RF predictions to their root causes, thus facilitating supervisory insights into future resource demands.
4) The core of the autoscaler is a discrete-time, closed feedback loop with two states, CPU and memory usage, that is able to allocate resources in real time both horizontally and vertically.
5) The RF-based autoscaler is evaluated on a home-grown, open source, microservice prototyping platform with the microservices deployed on an AWS virtual machine and managed by the Docker Engine.

Recently, the cloud computing community has shown strong interest in applying RFs to various aspects of cloud resource management [8], [12], [13], [14], [15]. Applications to both virtual machines [14], [15] and microservices [8], [13] have been considered. This recent research and the encouraging results it has produced confirm that RF machine learning is very suitable for managing cloud workloads, be they run on traditional infrastructures or on microservices. The distinct novelty of our own contributions to this emerging research area is threefold. The first is that our RF model predicts both CPU and memory usage at once, while in prior work two RF models must be trained, one for each metric. The second is that our RF model is trained on publicly available datasets rather than privately generated ones, and therefore future RF models can use the same datasets to benchmark their results against ours. The third is that our RF models place no restrictions on the nature of the workloads that are run on the cloud cluster.

The remainder of this paper is organized as follows. In Section II, recent work on microservice proactive autoscaling is reviewed. In Section III, the system architecture of our proposed autoscaler is given, and in Section IV, the design, analysis, and evaluation of the RF prediction models are explained in detail and benchmarked against the prior art. The RF models are nothing but components withing the full autoscaler algorithm whose validation and evaluation are given in Section V. The paper is concluded in Section VI.

II. LITERATURE REVIEW

Autoscaling systems are generally used to enhance the performance of cloud-hosted applications. There are two main types of autoscaling: (1) vertical autoscaling in which the amount of hardware resources assigned to each microservice is changed and (2) horizontal autoscaling in which the number of microservice replicas is changed. Figure 1 illustrates how the two types of autoscaling work. Due to the migration of cloud services to microservice architectures, an increasing number of publications on microservice autoscaling has appeared [6], [8], [9], [12], [13], [14], [15], [16], [17], [18]. In this section, we will review these publications with focus on their use of machine learning in their microservice management models. We will then highlight those publications that have used RFs in their machine learning modules.

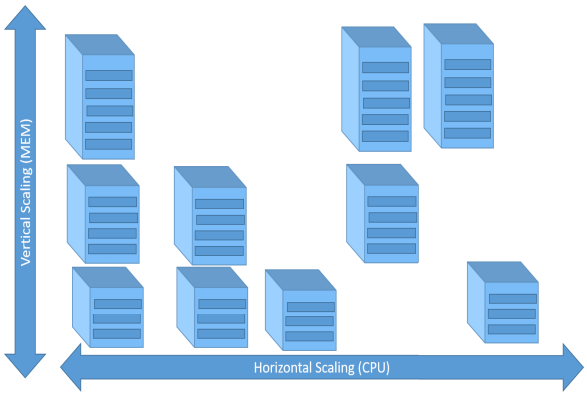


FIGURE 1. Vertical vs. horizontal scaling.

RScale [6] is an autoscaler that provides an end-to-end performance guarantee. It uses a predictive model based on Gaussian Process Regression (GP) to predict the end-to-end tail latency of microservice work?ows and provides confidence bounds on each prediction with minimal overhead. RScale periodically collects the resource usage metrics at two levels: the container level and the virtual-machine level. The periodically calculated utilization value is used as the threshold value in the Kubernetes autoscaler [19] to find the desired number of containers (see also Section III). The microservices are either scaled up by adding more containers or down by removing containers. The RScale evaluation results have shown that the proposed system can meet the system SLAs (e.g.,tail latency) even in the presence of varying interference and evolving system dynamics.

HyScale predicts future CPU utilization using a neural network (NN) model [16]. Four NNs have been trained offline: standard (baseline model), layer-normalized, long short term memory (LN-LSTM); multi-layered, LN-LSTM; and convolutional, multi-layered, LN-LSTM. These pretrained models are used as references and updated reactively to real-time data using the online technique recommended in [20]. The convolutional, multilayered, LN-LSTM has achieved the lowest error in the training phase and has been used as a reference for online training. The updated online model outperformed its offline version and reduces its prediction error to about 3.77%. Due to its up-to-date data processing, online learning is considered the preferred method for identifying time-series anomalies.

The autoscaling method introduced in [7] uses neural networks to predict the workload of microservices and execute advanced scaling to avoid SLA violations. The workload prediction is based on the HTTP request history. The method is focused on reducing microservice management costs as the system searches for cloud packages that have the lowest prices for a given task. As such, the system works as a cloud broker that continuously monitors the SLA's and notifies the scaling module whenever an SLA violation occurs so as scaling action is taken.

In [9], a horizontal autoscaling model based on predictive ML is proposed that uses the collected workload observations to predict future workload bursts. This predicted value is used to find the number of containers needed to meet the SLA latency requirements. Different state-of-the-art ML models were evaluated and the model with the least mean square error was selected, namely, the decision tree regression model. This predictive autoscaler targets CPU-bound microservices. The Fast Fourier Transformation (FFT) algorithm [21] is used to evaluate system performance and compare it with four previous autoscaling methods for virtual machines. The proposed model was found to outperform all of them under a variety of workloads.

A very recent predictive autoscaler that employs 2 ML models, a CPU model and a Request model, to predict the number of replicas needed for each microservice is proposed in [8]. The model also considers the impact of scaling a given microservice may have on other microservices under a given workload. The resource allocation is threshold-based as in [19]. The experimental results indicate that this autoscaler outperforms the Kubernetes horizontal autoscaler in terms of response time and throughput. Moreover, it takes fewer actions to achieve the desired efficiency and QoS standard for the target application.

The system of [17] is made up of two modules. The first uses a generic autoscaling algorithm installed on Google's Kubernetes Engine (GKE) to determine the microservice resource needs. The algorithm adjusts the Kubernetes autoscaler based on the microservice resource needs. The second module employs Reinforcement Learning (RL) agents to learn and select autoscaling threshold values depending on resource demands and QoS. The experimental findings demonstrate that the microservice response time can be improved by up to 20% compared to the Kubernetes default autoscaler. Furthermore, the RL agents can determine the threshold values without violating the response time SLA. The proposed system provides a customized autoscaling solution for microservices while adhering to QoS restrictions with little effort required from the system users.

One of the most recent autoscalers is SHOWAR [18], which uses empirical variance and average historical usage to estimate the optimal resources for the running microservices. It scales the microservices horizontally based on a given target latency, which is defined as the 95^{th} percentile of the observed latency during one minute. The measurement is the average observed latency during one minute. The difference between target and measurement is used in a proportional-integral-derivative (PID) controller.

The above-mentioned autoscalers use a variety of machine learning methods to forecast the workload. However, they do so only for the workloads of the interactive, latency-critical containers. These are the workloads required to satisfy the SLA requirement for response time and tail latency. The batch containers that are used in scientific computing or offline training are not considered, mainly due to the fact that their forecasts are less precise than what autoscalers require if they are to make fine-grained resource updates. Our approach addresses this modeling gap and offers predictive models

that work for both interactive *and* batch workloads. We now highlight the use of RFs in the machine learning components of the surveyed articles.

As mentioned in the previous section, the use of RF models has been steadily growing in cloud computing [8], [12], [13], [14], [15]. However, the RF models differ significantly in their input features and predicted outputs. For instance, [13] uses RF to predict the resource usage (CPU or memory) based on past resource usage and the number of message requests. On the other hand, [8] uses the number of replicas and the request rate to predict CPU utilization. In both [8] and [13], the RF models are trained on data from a specific microservice application. The direct comparison between these RF models and ours is hampered by the lack of common benchmarks, the widely differing feature sets, and the incompatibilities between the monitoring windows. One aspect of prior RF models is that they are single output, and so for multiple metrics of resource allocation, the RF model must be duplicated and retrained for each metric. On the other hand, the proposed RF model of this paper is a two-output model that predicts both CPU and memory usage with consistent recruiting of all the available input features. Our two-output RF model translates into a significant reduction in model complexity, memory footprint, and training requirements. Metric selection is another important aspect in RF prediction. Indeed, while existing RF models are restricted to few resource metrics, our RF model takes into account *all* the metrics provided by Docker. This enables a more comprehensive modeling methodology that can extend the validity of the model to a variety of workloads. Finally, it is important to note that our model is trained on a publicly available dataset, namely FastStorage [22], whose metrics are collected for different types of workloads over an extended period of time. This again helps in having a more comprehensive model that can be used with any microservice application without the need for training.

The closest work to ours is *RunWild*, a very recent industrial system from IBM Research [13] that was reported while our own academic research was in its final stages. The IBM system is based on commercial IBM tools such as IBM Cloud and AutoAI [23] while ours uses only open-source tools. *RunWild* is composed of three main modules: management, computation, and execution. The management module processes the microservice deployment specifications, such as past resource usage and expected workload. The computation module computes the deployment solution for every periodic window. Machine learning is heavily used in the computation module. For one thing, K-means clustering is used to group microservices based on the number of message requests, CPU usage, and memory usage. For another, a regression model is generated using AutoAI for each microservice cluster. The regression models are used to predict the resource usage of the microservices. The deployment plan generated by the computation module consists of the number of replicas, node placement, allocated resources, and the workload partitioning strategy. For each microservice group in *RunWild*, there are two ML models: one to predict CPU usage and the other to predict memory usage. RF models have been used as regressors for some of the microservice groups. Their highest R^2 scores are 0.655 for CPU utilization and 0.713 for memory usage. Our top two output RF models outperform those of *RunWild* as shown in Section IV.

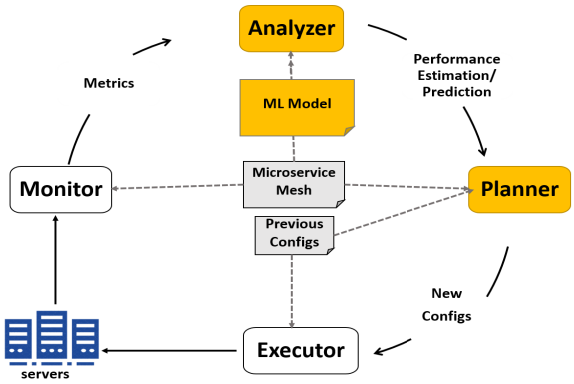


FIGURE 2. MAPE control loop autoscaler.

III. SYSTEM ARCHITECTURE OF THE PROACTIVE AUTOSCALER

In this section, we discuss the core concepts of our predictive ML autoscaler, whose main goal is to offer strong performance and SLA guarantees for containerized microservices. The autoscaler relies on an RF regression model that predicts near-future CPU and memory utilization for each running microservice based on previously collected metrics. These metrics are as follows: CPU usage, memory usage, and the throughputs of network input, network output, disk input, and disk output. The predicted values are leveraged to find the desired number of replicas (horizontal scaling) and resources (vertical scaling) needed to avoid SLA violations.

This core section of our paper is organized into three subsections. In Subsection III-A, the proactive autoscaler system architecture is described. Subsection III-B is devoted to the random-forest predictor. Finally, in Subsection III-C, the algorithms for horizontal and vertical scaling are described.

Note that Section IV of the paper will be devoted to the evaluation and explainability of the random-forest model itself while Section V will be devoted to numerical experiments on autoscaling using an e-commerce microservice mesh with real-world workloads.

A. MAPE CONTROL LOOP

Fig. 2 illustrates the main system components. The model is a "MAPE" control loop (i.e., Monitor, Analyze, Plan, and Execute). The *monitor* is responsible for periodically collecting the resource metrics of the running microservices. The collected metrics are then used by the *analyzer*, which is the RF regression model, to predict the running application performance. The predicted metrics are then sent to the *planner* to find the desired number of replicas for the running microservices and meet the SLAs. Finally, the *executor* is responsible for scaling the containers up or down. The

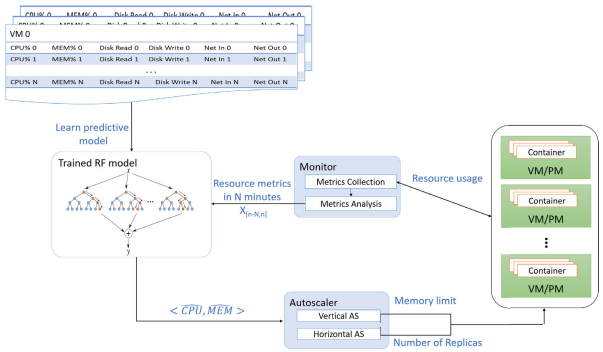


FIGURE 3. System architecture.

analyzer and the planner will be fully explained in Subsections III-B and III-C, respectively.

Fig. 3 further illustrates the system components and their interactions. The system workflow is as follows:

1) **Training:** Train the RF model on a large dataset collected from various applications running in the cloud. This helps to design a generalized training model that can accurately predict future CPU and memory usage for any type of workload. The training and evaluation of the model are discussed in Section IV.
2) **Monitoring:** The trained model periodically receives the performance metrics from the Monitor and uses them to predict future CPU and memory usages.
3) **Estimating:** The autoscaler then uses the predicted values to estimate the desired number of replicas and resources needed. This is explained in the autoscaling algorithms of Subsection III-C.
4) **Scaling:** The values calculated in the previous step are used to scale the containers.

B. THE ANALYZER: RANDOM FOREST PREDICTIVE MODEL

The analyzer is the random-forest (RF) regressor which uses an ensemble of decision trees to produce accurate predictions with a small amount of training data. RF uses the bagging technique to select the features for the best node splitting and to build multiple decision trees [24]. The average value across all decision tree predictions is the RF prediction. RF models can handle data with a mixed and large number of features, and are already widely used to predict the performance of virtual machines under a variety of workloads [15]. The evaluation results of [15] showed that the RF model outperformed both linear regression and plain decision trees. Furthermore, a very recent study [14] has conducted a comprehensive evaluation on a variety of workload predictors under four different workload patterns. The results showed that the RF model had good predictive accuracy compared with other models. All in all, RF models are scalable, largely resistant to overfitting [25], and well adapted to cloud computing tasks. Our work further supports these conclusions by showing that the RF regressor is an excellent fit for the microservice autoscaling task under unpredictable workloads.

1) MODEL FORMULATION

The RF model is an ensemble of decision trees. A decision tree with L leaves recursively partitions the K-dimensional feature space $\mathcal{F} \subset \Re^K$ into L regions: R_l, $1 \leq l \leq L$. For a feature vector $\mathbf{x} \in \mathcal{F}$, the general prediction function $f(x)$ of one decision tree is given by

$$f(\mathbf{x}) = \sum_{l=1}^{L} c_l I(\mathbf{x}, R_l)$$
$$I(\mathbf{x}, R_l) = \begin{cases} 1; & \mathbf{x} \in R_l \\ 0; & \mathbf{x} \notin R_l \end{cases} \quad (1)$$

where c_l is a constant calculated during the training phase and is equal to the average of the response variables in region R_l.

The above expression is compact and indicates how the decision tree returns the value of the leaf corresponding to the input feature vector. A more operational expression is given by

$$f(\mathbf{x}) = c_{full} + \sum_{k=1}^{K} C(\mathbf{x}, k) \quad (2)$$

where c_{full} is the mean of the response variables in the training set and $C(\mathbf{x}, k)$ the contribution of the k^{th} feature in the vector $\mathbf{x}$. Note that the contribution of each feature k is decided by *all* the features in vector $\mathbf{x}$, and these contributions correspond to a specific decision path that traverses the tree.

Using Eq. (2), one can write the prediction function $F(\mathbf{x})$ of an RF model as

$$F(x) = \frac{1}{J} \sum_{j=1}^{J} c_{jfull} + \sum_{k=1}^{K} \left(\frac{1}{J} \sum_{j=1}^{J} C_j(\mathbf{x}, k) \right) \quad (3)$$

where J is the number of decision trees, and $C_j(\mathbf{x}, k)$ the contribution of the k^{th} feature in feature vector $\mathbf{x}$ in the j-th tree. The above expression for $F(\mathbf{x})$ clarifies the fact that the RF prediction is simply the average of the tree predictions over all the trees.

In our context, the vector of features $\mathbf{x}$ has six components, which are the container resource metrics: CPU usage, memory usage, network input throughput, network output throughput, disk input throughput, and disk output throughput. These six metrics are gathered over a time interval of N minutes and stored in an array $\mathbf{X}_{[n-N,n]} \in \mathcal{F} \subset \Re^K$. The dimension of the feature space $K = 6N$. The values of J and K are decided at the training phase by performing hyper-parameter tuning with different sampling intervals. Finally, $F(\mathbf{x})$ is the 2D column vector $(\widehat{CPU}, \widehat{MEM})^T$ of the CPU utilization and memory usage estimates.

2) DATA NORMALIZATION

To efficiently scale microservices, different resource metrics should be considered. This aids in an accurate prediction of CPU and memory utilization. However, these metrics differ in scale and units. For example, the CPU usage ranges from 0 to 100 % while the network throughput ranges from 0 to 27727 KB/s. Re-scaling data before training

machine learning models can be beneficial as it reduces the sensitivity of the training process to the feature scale. This results in all features having the same influence on the model.

In addition to scale, the presence of outliers may impact the quality of the learned model. The workload metrics of the dataset used to train the RF model happen to contain many outliers, which usually compromises any advantages z-score scaling may provide. To address this aspect of the dataset, we have used *RobustScaler* [26] as a drop-in replacement for the z scoring. *RobustScaler* subtracts the median and scales the data according to the interquartile range (IQR). Recall that the IQR is given by IQR $= Q_3(x) - Q_1(x)$ where $Q_1(x)$ and $Q_3(x)$ are the first and third quartiles, i.e., the 25th and 75th quantiles, respectively. Centering and normalization are performed individually on each feature by computing the necessary statistics. The medians and IQRs are then saved to use with the validation data. The *RobustScaler* formula is given by

$$\frac{x - Q_2(x)}{\text{IQR}} \tag{4}$$

where $Q_2(x)$ is the median.

3) HYPERPARAMETER TUNING AND FEATURE SELECTION

While model parameters are learned during training, its hyperparameters are specified ahead of training. The RF model parameters include decision features and thresholds. Its hyperparameters include the number of decision trees and the depth of each tree. Machine learning libraries like *Scikit-Learn* provide a set of reasonable default hyperparameters for all models. Such default hyperparameters are not guaranteed to be optimal for all problems. Finding the optimal hyperparameters requires the formulation of an optimization problem whose solution may use a variety of deterministic, stochastic, or heuristic methods.

Hyperparameter tuning is based on experimental results that are obtained by means of a grid or random search. To determine the ideal hyperparameter values, a large number of various combinations are tested, and their impact on model accuracy using k-fold cross-validation is considered. In our case, we have used *RandomizedSearchCV* from *Sickit-learn* for hyperparameter tuning. The tuning results of the 5 hyperparameters considered are given below based on dataset with 84191 traces across 100 different combinations:

1) Number of decision trees in the forest: 1000.
2) Maximum number of features to be considered at each split: 6 (all features).
3) Maximum depth of each tree in the model: 100.
4) Minimum number of data points placed in a node before splitting: 5.
5) Minimum number of data points in a leaf node: 4.

These hyper-parameter values achieve an improvement of 20% in the RMSE with respect to the default values.

Once an RF is available, the importance of features can be calculated based on a test data set. In permutation importance, features are randomly shuffled and the change in model performance computed after each shuffle. The feature that impacts the most on the performance of the model is declared to be the most important. The following section will illustrate the impact of selecting the most important features not only on model accuracy, but also on its complexity.

C. THE AUTOSCALER

We have designed a full autoscaler model that bridges the gap between vertical and horizontal autoscaling. The model achieves competitive targets in resource utilization and end-to-end responsiveness.

The algorithm 1 shows how the horizontal and vertical autoscalers work together. The autoscaler's throughtput is one decision per minute. This is based on the fact that one minute is the shortest time interval needed to collect an adequate quantity of metric values in order to predict the resources and perform autoscaling. During this time interval, the monitoring program collects metric values at a sampling rate of 12 samples per minute with each sample containing values of 4 metrics for a total of 48 values. This is accomplished in parallel for all microservices (lines 5-7 of Algorithm 1). Horizontal scaling is based on the CPU usage metric (line 10 of Algorithm 1) while vertical scaling is based on the memory metric (Algorithm 2). In addition, each microservice is given a CPU limit of 1 CPU core, as recommended by Kubernetes best practices for most workloads on the Google Cloud Platform [19]. As a result, CPU's are not scaled vertically as each microservice will run on a single CPU. Furthermore, vertical scaling in memory is given priority over horizontal scaling. This is to avoid Out-of-Memory (OOM) errors. Such errors cannot be addressed by adding more microservice replicas. While CPU's can be scheduled to run heavy workloads and achieve full utilization at the price of larger latency, memory resources cannot. A container will be terminated if it exceeds its memory limit.

For horizontal scaling, the same formula used by the Kubenertes horizontal autoscaler, K8s-HA, is adopted to find the desired number of replicas:

$$R_d = \left\lceil R_c \frac{\mu_c}{\mu_d} \right\rceil \tag{5}$$

where $\lceil \cdot \rceil$ is the ceiling function, R_c and R_d are, respectively, the current and desired number of replicas, μ_c the predicted metric and μ_d the threshold value of the metric, assumed to be known for each container. The scaling is only performed if the ratio $\frac{\mu_c}{\mu_d}$ is less than 0.9 or larger than 1.1. Reactive autoscalers, such as K8s-HA, make scaling decisions whenever an SLA violation occurs or whenever the resource utilization of a container exceeds its threshold. On the other hand, in predictive autoscalers, such as the one proposed in this paper, the predicted value μ_c helps in taking the scaling decision well in advance of any SLA violation. In our context, μ_c is the predicted CPU value $\widehat{CPU}$, and μ_d is the CPU threshold value thr.

The industrial state of the art in vertical autoscaling, notably the Kubernetes vertical autoscaler [5] and the Google Autopilot [27], adopt a traditional approach to setting limits on microservice CPU and memory. In particular, they monitor

Algorithm 1 Autoscaling Algorithm for Each Microservice

Input MEM_limit_c, R_c, thr
MEM_limit_c: current MEM limit in MB
R_c: current number of replicas
thr: CPU threshold
Output R_d, MEM_limit_d
R_d: desired number of replicas
MEM_limit_d: desired MEM limit in MB
1: $\widehat{CPU}$: predicted CPU usage
2: $\widehat{MEM}$: predicted MEM usage
3: RF: Random Forest model
4: **while** True **do**
5: **for** a duration of 1 minute **do**
6: Collect metrics every 5 seconds
7: **end for**
8: $< \widehat{CPU}, \widehat{MEM} >= RF.predict(metrics)$
Vert. Scaling
9: $MEM\_limit_d \leftarrow mem\_scaling(MEM\_limit_c, \widehat{MEM})$
Horz. Scaling
10: $R_d = \left\lceil R_c \frac{\widehat{CPU}}{thr} \right\rceil$
11: **end while**

Algorithm 2 Memory Scaling Algorithm

Input $\widehat{MEM}$, MEM_limit_c
MEM_limit_c: current MEM limit in MB
$\widehat{MEM}$: predicted MEM usage
Output MEM_limit_d
MEM_limit_d: current MEM limit in MB
1: **if** $\widehat{MEM} > 80\%$ **then**
2: $MEM\_limit_d \leftarrow (1+\alpha) * MEM\_limit_c$
3: **end if**
4: **if** $\widehat{MEM} < 50\%$ **then**
5: $MEM\_limit_d \leftarrow (\alpha + \widehat{MEM}) * MEM\_limit_c$
6: **end if**
7: **return** MEM_limit_d

resource usage (CPU and memory) during a time window, from a few minutes to several days, and set the resource allocation for the next time window to be some percentile, typically between 90^{th} and 99^{th}, of the usage in the previous time window with a safety margin between 0.10 and 0.2. While this strategy addresses the under-provisioning problem and reduces the frequency of CPU throttling and OOM errors, it still leaves the issue of underutilized resources unaddressed, which is not cost-effective for cloud users.

In our vertical autoscaler, both under- and over-provisioning are considered along with their implications. This is embodied in Algorithm 2. The vertical scalar uses the predicted memory utilization to scale the memory limit of each container. The memory limit of the microservice is adjusted during the upcoming time window to $\widehat{MEM}(1+\alpha)*MEM\_limit$ if the predicted memory utilization is more than 80% of the current limit, lines 1-3 of Algorithm 2. The additional

TABLE 1. The schema of *fastStorage* dataset. The highlighted rows are the selected features in the RF model.

Index	Name	Description
0	Timestamp	Number of milliseconds from the start of the trace
1	CPU cores	Number of virtual CPU cores provided
2	CPU provisioned (MHZ)	CPU capacity in terms of MHZ
3	CPU usage (MHZ)	CPU utilization in MHZ
4	CPU usage (%)	CPU utilization in %
5	Memory provisioned (KB)	Memory capacity of the VMS
6	Memory usage (KB)	Memory utilization in KB
7	Disk Read (KB/s)	Disk read throughput
8	Disk write (KB/s)	Disk write throughput
9	Network in (KB/s)	Network input throughput
10	Network out (KB/s)	Network output throughput

α% of memory is used as a safety margin to prevent under-provisioning. A sweep over the safety margin $\alpha \in [0.1, 0.2]$ has been carried out and the value $\alpha = 0.15$ has been selected as it gives the best results with competitive resource utilization and no OOM errors. Details about these selections will be given in Section IV. To avoid over-provisioning, the autoscaler reduces the *mem_limit* to be at least $1-(\alpha+\widehat{MEM})$ less than the current limit if the predicted utilization is less than 50%, lines 4-6 of Algorithm 2. In other words, the reduction amount in the memory limit is $mem\_limit_c - mem\_limit_d = (1-(\alpha+\widehat{MEM}))mem\_limit_c$. This guarantees that memory utilization will always be more than 50%.

The proposed vertical autoscaler prevents under- and overprovisioning by considering the *predicted* memory usage instead of the *current tail percentile* usage. When there is a considerable fluctuation in resource utilization, the autoscaler supplies adequate resources, thus preventing both under-provisioning and performance deterioration. Furthermore, when resource utilization variance is small, the autoscaler does not over-provision, thus preventing over-provisioning and resource wastage.

IV. VALIDATION OF THE AUTOSCALER RANDOM-FOREST MODEL

This section is devoted to the evaluation and validation of the autoscaler predictive RF models, including the data set used in the evaluation (Subsection IV-A), the figures of merit (Subsection IV-B), and the design details of the RF models (Subsection IV-C). An important advantage of decision tree and random forest learning models is their explainability. This is addressed in Subsection IV-D for the autoscaler RF models. Finally, in Subsection IV-E, the autoscaler RF model is compared with some of the linear regression models that have appeared in the literature.

A. DATASET

To train and evaluate the RF model, we have used the *fastStorage* dataset [22]. *fastStorage* is comprised of workload traces of different software applications running on 1,250 VMs hosted within the Bitbrains data center. The workload traces have been collected as tenants join and leave the cluster and are, therefore, very dynamic. This is why *fastStorage* facilitates the creation of a very general model that is robust to variations in workload. The *fastStorage* traces are saved in CSV files, with each file containing one month worth of

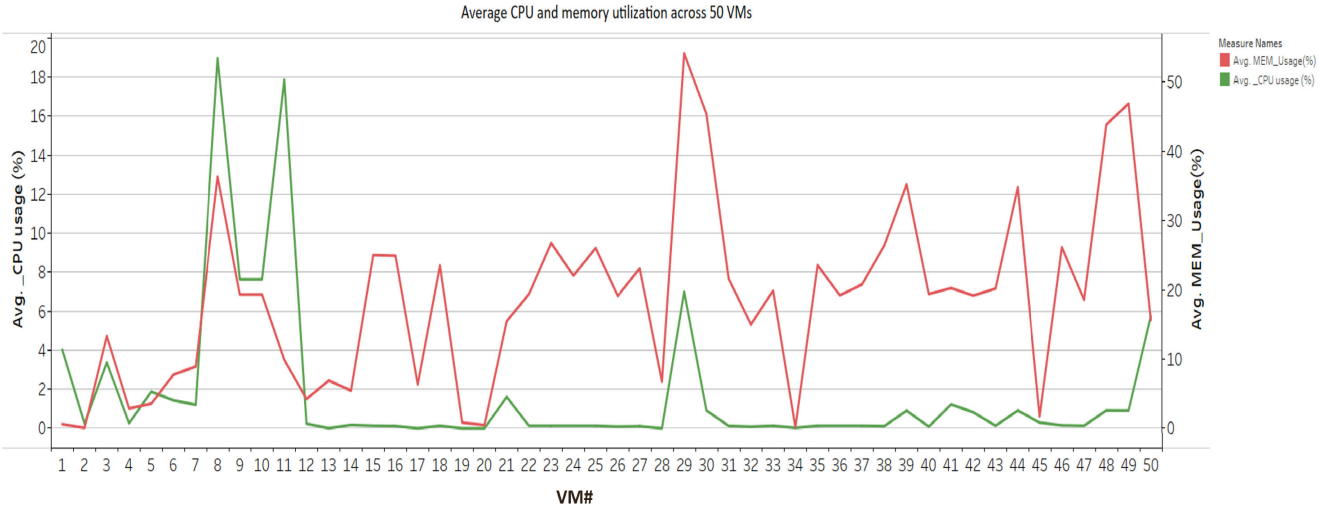


FIGURE 4. The average CPU and memory consumption across 50 Bitbrains *fastStorage* VMs.

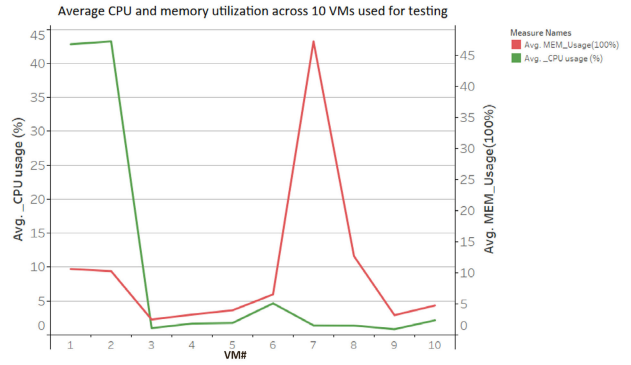


FIGURE 5. The average CPU and memory consumption across 10 Bitbrains *fastStorage* VMs.

traces collected at the sampling rate of one sample every five minutes. The schema of the dataset is shown in Table 1.

Of the 1,250 VMs, we randomly chose 50 VM traces (445,102 traces) for training and validation. We have also chosen another 10 VMs (170,00 traces) to test the accuracy and generality of the trained models. Figures 4 and 5 show the average CPU and memory usage in the 50 virtual machines used for training and the 10 virtual machines for testing, respectively.

These plots clearly show the workloads to be very bursty, which makes prediction and generalization quite challenging. The average CPU utilization is less than 10% in more than half of the virtual machines. In addition, 50% of the record have a very low (almost zero) CPU usage (see Table 2). On the other hand, memory has lower variance with most of the VMs having about the same average memory usage. In fact, the memory variance is the lowest among all resources, while network input throughout has the highest variance [28]. There is also a strong correlation between CPU and memory usage when CPU usage is high. However, the correlations between CPU usage and other resources are very low. According to the

TABLE 2. Statistical analysis of workloads (50 VMs).

Features	Count	mean	std	min	25%	50%	75%	max
CPU_usage(%)	445101	1.69	5.43	0.0	0.00	0.00	0.80	105.33
Memory_usage(%)	445101	17.241	15.00	0.0	4.62	14.03	26.90	1064.24
Disk_Read(KB/s)	445101	0.95	42.52	0.0	0.00	0.00	0.00	13977.50
Disk_Write(KB/s)	445101	13.74	356.32	0.0	0.00	0.27	1.13	56630.60
Network_In(KB/s	445101	154.20	1217.74	0.0	0.00	0.00	1.07	38371.73
Network_Out(KB/s)	445101	77.40	1155.89	0.0	0.00	0.00	1.13	38371.73

statistical analysis performed on *fastStorage* in [28], about half of VMs have stable CPU utilization centered on the mean.

Further statistical analysis is conducted on the features of the *fastStorage* dataset. Six features were selected and Table 2 highlights their statistical properties. All features, except memory utilization, have almost zero values most of the time. CPU and memory utilizations have values larger than 100% as highlighted in yellow in the table. These out-of-range values indicate over-utilization cases where the usage exceeds the provisioned resources, and are, therefore, considered outliers. To normalize the feature values, *RobustScaler* is used because of its robustness with respect to outliers. Note that a significant part of the monitored CPU usage in the Bitbrains dataset is 0%. Since a faithful model is supposed to capture this CPU behavior, the use of data filtering has been limited, and CPU usage that is zero or nearly zero is not filtered out.

One major criterion of feature selection is to avoid redundant selections. For example, in Table 1 the feature # 4 (CPU utilization) is nothing but the ratio of features # 2 and # 3. Feature # 4 is selected as it is the one provided by the Docker stats. Similarly, memory utilization (%) is calculated by dividing memory usage (KB) by provisioned memory (KB). All other features are used as given. The model features are, therefore, CPU utilization (%), memory utilization (%), and read, write, network in and network out performance. These features are also the metrics collected by the Docker engine and are the main metrics used to analyze the container workloads. For prediction, the data needs to be rearranged in a

time series. This is achieved using a sliding window of 1 hour with a sliding step of 5 minutes. As a result, the total number of feature samples is 6 *features* × (1 *hour*/5 *minutes*) = 72 values.

B. FIGURES OF MERIT

The figures of merit used for RF model evaluation are: Mean Absolute Error (MAE), Root Mean Squared Error (RMSE), and the coefficient of determination (R^2) whose expressions are

$$\text{MAE} = \frac{1}{N}\sum_{k=1}^{N}|y_k - \hat{y}_k| \tag{6}$$

$$\text{RMSE} = \sqrt{\frac{1}{N}\sum_{k=1}^{N}(y_k - \hat{y}_k)^2} \tag{7}$$

$$R^2 = 1 - \frac{\sum_{k=1}^{N}\left(y_k - \hat{y}_k\right)^2}{\sum_{k=1}^{N}(y_k - \bar{y})^2} \tag{8}$$

where N is the number of data points, y_k the k-th data sample, $\hat{y}_k$ the k-th predicted value, and $\bar{y}$ the statistical mean of the N data points.

RMSE and MAE are loss functions that measure prediction precision, MAE being the most resistant to outliers [29], and RMSE the most commonly used to compare regression models with others. Typically, small MAE and RMSE values indicate an accurate regression model.

On the other hand, R^2 is a statistical measure that quantifies the proportion of variance of a dependent variable that is attributed to the regression variable(s). In contrast, correlation quantifies the variation overlaps between two statistical variables. For example, a value of $R^2 = 0.79$ means that the regression model can explain about 79% of the variance of the dependent variable. The highest R^2 score is 1, which implies that the regression model fits the data perfectly.

C. RANDOM FOREST MODELS

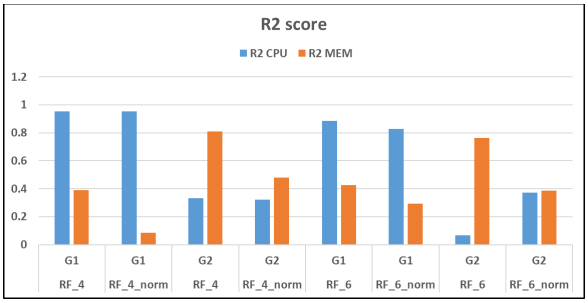
Our initial RF model has a single output and was geared toward the predictions of CPU utilization in compute-intensive microservices. As is clear from the statistical analysis of the *fastStorage* dataset, predicting CPU alone is not sufficient to avoid SLA violations. Memory utilization must also be taken into account. In the absence of prediction of memory utilization, OOM errors will result in increased response time, forcing microservice replication, which in turn translates into resource waste and increased cost. To address this common scenario, we have designed a two-output RF regression model that predicts both CPU and memory utilizations.

Multiple two-output RF models have been designed with different feature space dimensions and data normalization techniques. The best among these RF models has been compared with a reference linear regression model. The dataset selected from *fastStorage* (see IV-A) has been partitioned into two parts: 70% for training and validation and 30% for testing. The models have been validated using 10-fold cross-validation using the k-fold cross-validation API from Python's *scikit-learn* library. In the selection of the of the feature space dimension, both a dimension of 6 (all features) and 4 have been considered. An important aspect of the latter choice is the selection the top 4 most critical features. This has been accomplished using permutation feature importance [30], [31]. This approach randomly shuffles each feature j to break the associations between the feature and the output variables. It then calculates the difference between the original-model performance measure

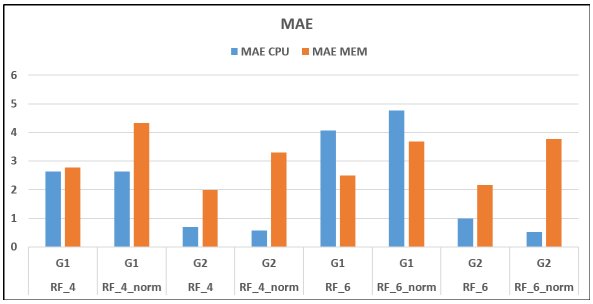
TABLE 3. The four RF models designed using two feature dimensions: 4 and 6, with and without normalization. The quality of the model is evaluated on the basis of the three criteria: R^2, RMSE, MAE.

RF Model	Features	Normalization	Model Quality: G1		Model Quality: G2	
			CPU	MEM	CPU	MEM
RF_6	6 features	No				
RF_6_norm	6 features	Yes				
RF_4	4 features	No				
RF_4_norm	4 features	Yes				

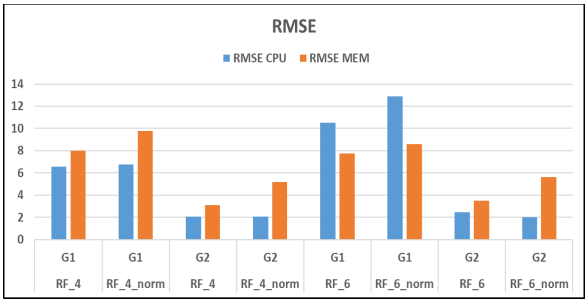
Best — Worst



(a) R^2 Score.



(b) MAE metrics.



(c) RMSE metrics.

FIGURE 6. Regression metrics for the RF models.

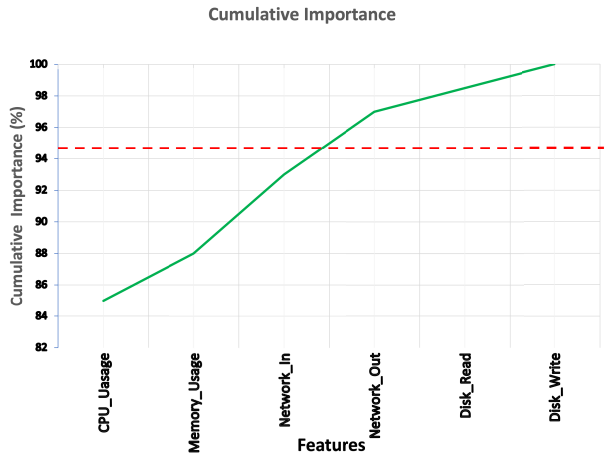


FIGURE 7. The cumulative feature importance graph.

(e.g., R^2) and the model performance with the shuffled data (e.g., $R^2(j)$). The most important features result in the most significant gain $R^2 - R^2(j)$ in the model performance measure. The cumulative feature importance graph is shown in Figure 7, where the *Disk_Read* and *Disk_Write* features are the least important and can therefore be removed. The remaining four features: *CPU_usage, Memory_usage, Network_In,* and *Network_Out* exceed 95% of cumulative importance and are therefore retained. Another possible selection that exceeds the 95% threshold is to replace *Network_Out* with *Disk_Read*. We have opted for the first selection as *Network_Out* turned out to be more important than *Disk_Read*.

Four RF models have been designed and their properties are summarized in Table 3. The MAE, RMSE and R^2 performance measures of these models are illustrated in Figure 6. The $G1$ and $G2$ testing sets are the traces of 10 different VMs. The traces in $G1$ have memory utilization outliers that range from 200 to 1750%, while $G2$ has no such outliers. In terms of CPU utilization, $G1$ has one CPU utilization outlier of value 120%. On the other hand, most of the $G2$ traces have very low CPU utilization ($< 20\%$). The main goal of using the $G1$ and $G2$ datasets is to illustrate the robustness of the RF models and their ability to accurately predict future resource utilization regardless of the type of workload.

From Table 3 and the graphs in Figure 6, the RF_6 model has a high R^2 value for the CPU in the $G1$ dataset, but a very low R^2 value for the CPU in the $G2$ dataset. This is due to the very long stretches of zero CPU utilization in this dataset. On the other hand, the R^2 of the RF_6 model is more than 0.7 for memory utilization when there are no outliers. Its performance is better than the RF_6_norm model for all three measures MAE, RMSE, R^2. On the other hand, models RF_4 and RF_4_norm compare more favorably than RF_6_norm. Note that RF_4 model has the lowest MAE and RMSE for CPU and memory utilization when there are no outliers. It also has a high R^2 score in all cases. The RF_4 model is the fastest in both the training (670 seconds) and scoring (1.02 seconds) phases. On the other hand, the training

TABLE 4. The feature vector of a sample, its actual CPU, and memory utilization.

Feature	Value
CPU_usage11	1.73
Network_IN11	4.60
mem_usage11	8.18
mem_usage4	4.62
Network_OUT11	0.00
Network_IN10	4.60
mem_usage7	7.82
Network_IN5	4.67
Network_OUT4	0.00
Network_OUT8	0.00
Ground Truth	<1.73, 8.18>

times of the other models range from 780 to 1046 seconds, and their prediction times range from 1.15 to 1.24 seconds.

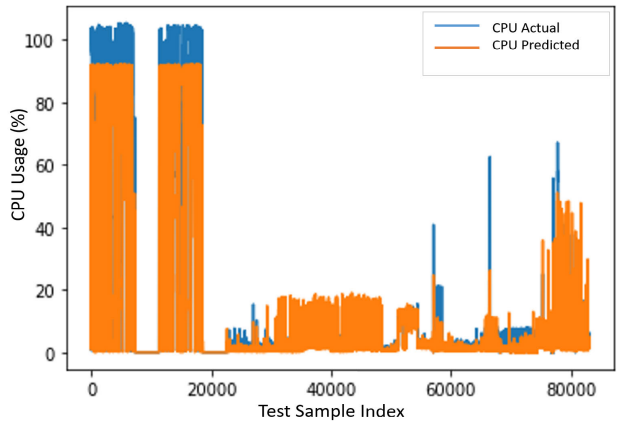
The evaluation results illustrate the negative impact of overfitting (6 features instead of 4) and the important fact that in RF regression, normalization may not have the same favorable impact as in linear regression.

The advantage of a compact RF model is the significant reduction in storage footprint, which has been reduced from 4 GB to 2 GB when the feature space dimension is reduced from 6 to 4. Note that normalization is a pre-processing step that increases the learning runtime by 15% and the prediction runtime by 4% with respect to non-normalized input. Normalization has no impact on the performance measures for the predicted CPU utilization. On the other hand, it negatively impacts predicted memory utilization, even though the memory utilization traces have more outliers compared with the CPU traces. The conclusion is that in RF regression, the RF models can be made sufficiently robust in the presence of outliers without any input normalization.

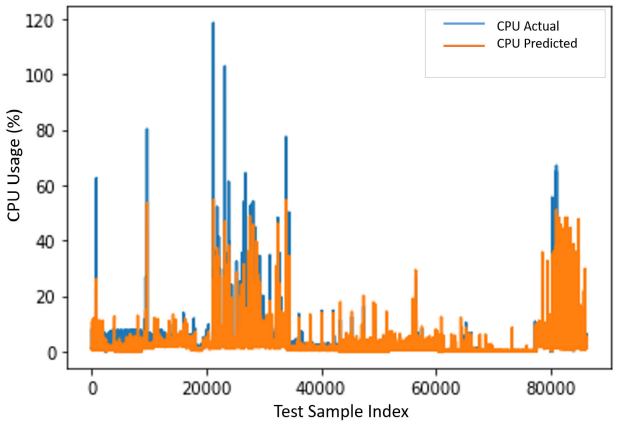
The predicted and actual values of the RF_4 model for CPU and memory utilization are shown in Figures 8 and 9, respectively, with (b) and without (a) outliers. The CPU outlier value is 120% while the memory's is 1750%. This latter value explains why memory outliers have such a negative impact on prediction accuracy. The range of the predicted CPU and memory utilization values is from 0 to 100%. Finally, note that there is no difference in the trends between the predicted and actual values. In other words, future CPU and memory utilizations are readily predictable from previous data traces.

D. MODEL EXPLAINABILITY

The decision trees of the RF predictor can be analyzed to gain more information on the relationship between the features and the predicted values. The RF autoscaler predictor is an ensemble of decision trees, the first of which is illustrated in Figure 10. The highlighted path is the decision path of the input sample shown in Table 4. The 12 samples collected during the one-hour sliding window are indexed from 0 to 11, and the notation *CPU_usage11* indicates a value of CPU utilization that corresponds to the very last sample. Similarly, *mem_usage7* indicates a value of memory utilization that corresponds to the 8th sample. For the RF_4 model, the total number of values is 48, all of which are used to predict CPU

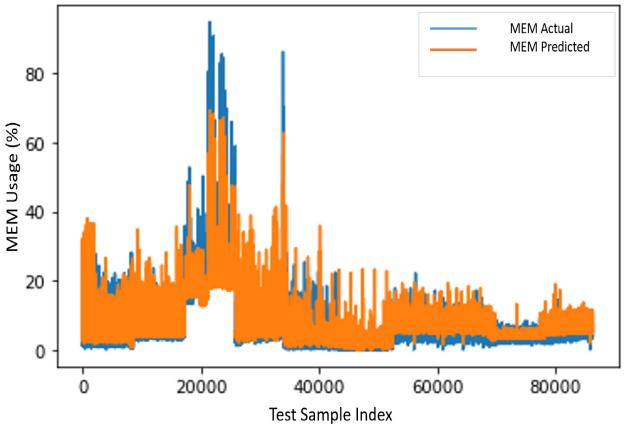


(a) The actual and predicted CPU usage values without outliers.

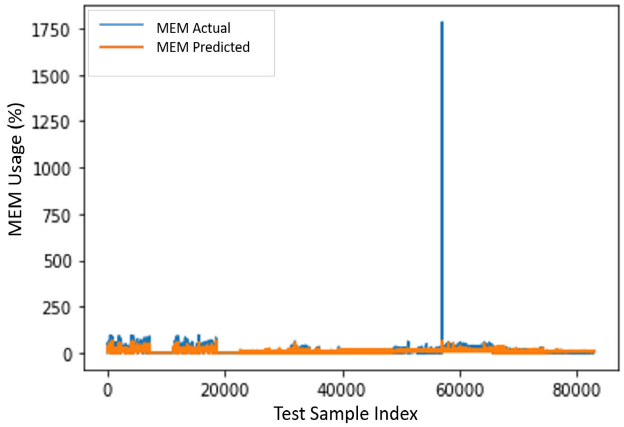


(b) The actual and predicted CPU usage values with outliers.

FIGURE 8. Actual and predicted CPU usage.



(a) The actual and predicted memory usage values without outliers.



(b) The actual and predicted memory usage values with outliers.

FIGURE 9. Actual and predicted memory usage.

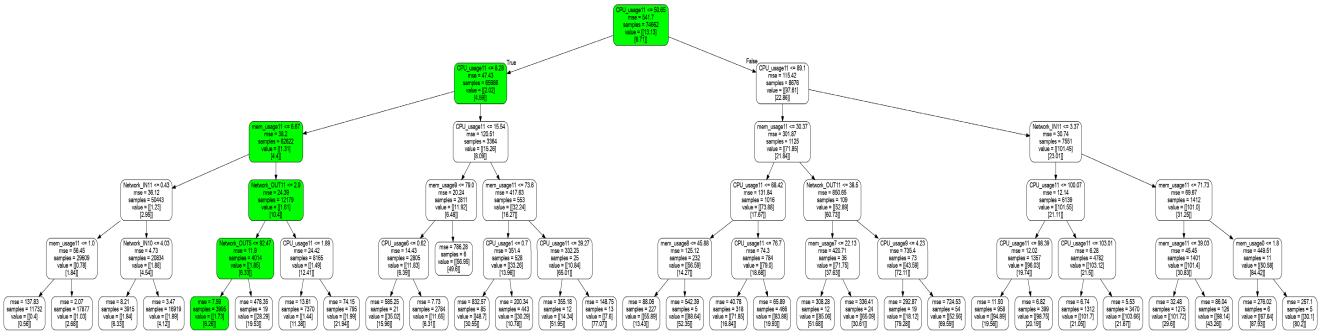


FIGURE 10. One of the decision trees of the RF model. The actual depth of the estimator is 100, but it is reduced to 5 in this graph for easier visualization. The highlighted path is the decision path taken for prediction based on the sample of Table 4.

and memory utilizations in the next 5 minutes. The details of the decision path are illustrated in Figure 11. The features shown in Table 4 and their value correspond to those used along the decision path of Figure 11. Note that the decision path corresponds to workload metrics collected in the middle and at the end of the sliding window. According to Eq. (1), the predicted values correspond to the internal state of the leaf node $(\widehat{CPU}, \widehat{MEM}) = (1.73, 6.26)$. It is important to note that the tree has been pruned for easier visualization. The actual predicted values are slightly different and are given by $(\widehat{CPU}, \widehat{MEM}) = (1.43, 8.01)$. These values are very close to the actual ones. Other decision paths can be traced as well across the trees of the ensemble with their leaf states averaged out to provide the RF predictions.

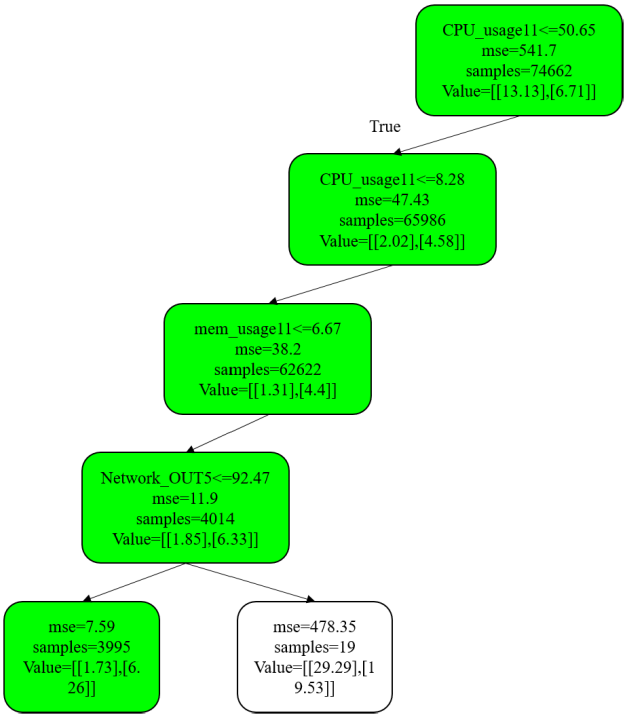


FIGURE 11. The decision path of the sample of Table 4.

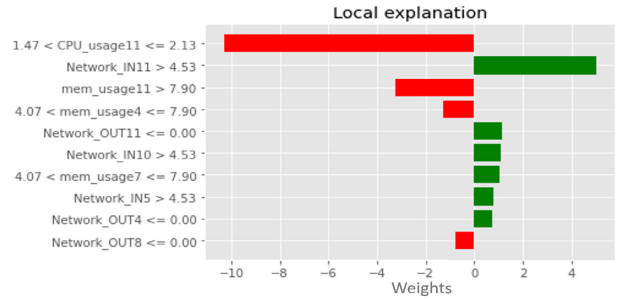


FIGURE 12. A LIME-generated bar chart of feature value contributions to the prediction of CPU utilization corresponding to the sample of Table 4.

One major advantage of using decision tree and random forest regression is their explainability. A given decision path in a tree can be readily used to trace back the prediction values to the input sample. When the RF model consists of a large number of decision trees (in our case 1000 trees) with high depth, a tool is needed to visualize the aggregate contributions of the trees to the RF predictions. We have used the LIME [32] tool for such visualizations. Figure 12 lists the top 10 regressor values, of the 48 collected in the one-hour time window, that contribute the most to the prediction. The vertical bar graph has the decision region of each regressor value on the y axis and the contribution weight of each regressor on the x axis. A positive (negative) weight is coded as a green (red) bar. It indicates an increase (decrease) in the predicted value when the regressor value increases. For the particular sample under consideration, *CPU_usage11, mem_usage11, mem_usage4,* and *Network_out8* all have negative weights and decrease the predicted CPU utilization when they increase. All other regressors increase the

FIGURE 13. The total number of requests per seconds and number of users of the generated loads.

predicted CPU utilization when they increase. Note that such explanations are for the RF model with 1000 trees. They could not have been made without LIME.

E. COMPARISON WITH LINEAR REGRESSION

The best RF model, the RF_4 model, was compared with a reference linear regression model, denoted LR. The regression variables in both models are the 48 variables collected during a time window of 1 hour at the sampling rate of 1 sample every 5 minutes. Compared with LR, RF_4 has lower MAE, MSE, RMSE, and a higher R^2 score. When examined further, many of the predicted values of LR are negative, possibly due to the presence of outliers in the dataset. Naturally, the learning and prediction run times of the linear regression model are much lower than those of RF_4.

The RF models have also been compared against recent state-of-the-art predictive autoscalers. In comparison with LSTM NN model proposed in [16] (3.9 average RMSE), RF_4 has achieved a better prediction accuracy with an average RMSE of 2.9 vs. 3.9. The LSTM NN model is more complex, requiring $12 \times 3 \times 6 = 216$ samples (i.e, 3-hour sliding window and 6 features) vs. 1 hour for RF_4. In addition, the LSTM NN model has used all 6 features vs. RF_4 which uses only the top 4 features. In other words, RF_4 achieves better prediction accuracy with a less complex model having a smaller number of features, a narrower data window, and a shorter training time. RF_4 learns thousands of samples in seconds and makes predictions in fractions of a second, allowing for frequent model re-training using the most recent traces. It is also much less prone to overfitting even in the presence of highly repetitive data sequences.

In [8], an RF model is proposed that uses the number of container replicas and the request rate to predict CPU utilization. The targeted application is interactive microservices and the dataset used for training, validation, and testing is *TeaStore*. On the other hand, RF_4 has been designed for arbitrary workloads and uses the most important resource metrics to predict CPU and memory utilization.

V. APPLICATION TO MICROSERVICE AUTOSCALING

In this section, a complete e-Commerce example of a microservice mesh is used to evaluate the proposed predictive autoscaler. The evaluation will exercise the complete MAPE

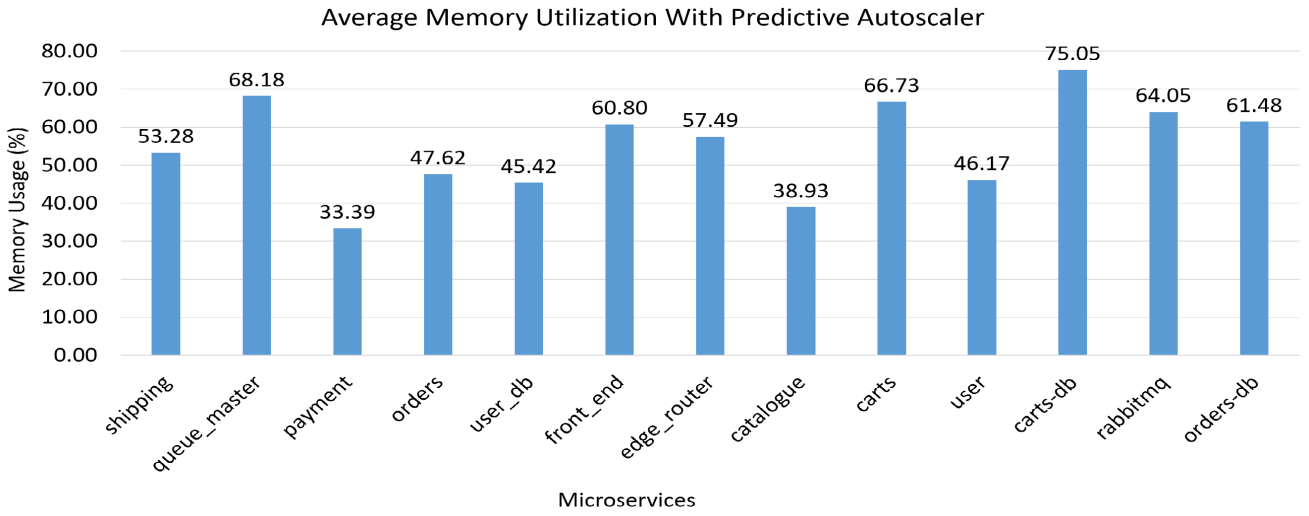


FIGURE 14. Average memory utilization during the load test with the proactive autoscaler (this work).

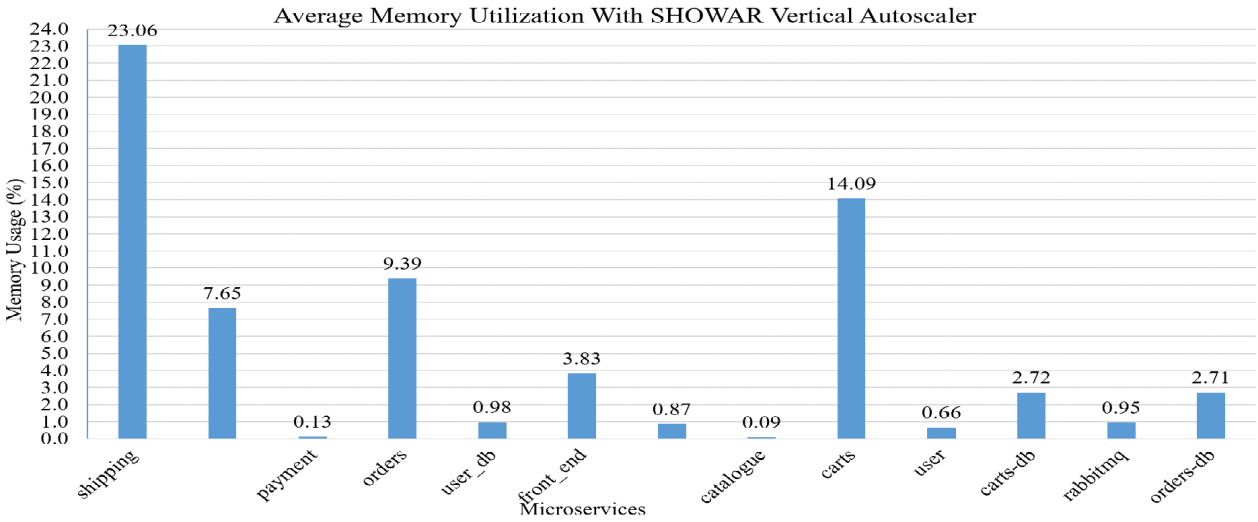


FIGURE 15. Average memory utilization during the load test with SHOWAR vertical autoscaler [18].

architecture of Fig. 2 and analyze the evaluation results for all mesh microservices. This is done on a home-grown, open-source platform for the fast prototyping of microservice meshes [33].

A. EXPERIMENTAL SETUP

The experiments are carried out on the AWS Cloud in the ap-south-1 region, using a t2.2xlarge instance (x86_64 architecture) with 8 vCPU (Intel Scalable Processor up to 3GHz) and 32 GB of memory, running Ubuntu 18.04 LTS. All services are running on a single host. Docker-compose is used to configure the services and create a network for inter-container communication. The Docker-compose version is 1.29.2 and the Docker-engine version is 20.10.21.

1) E-COMMERCE MICROSERVICE EXAMPLE

The full autoscaler is evaluated using *Socks Shop* [34], an e-commerce application implemented as a microservice mesh to demonstrate the advantages of using a container platform. It consists of 14 microservices built using *Spring Boot, Go kit,* and *Node.js*. It also contains a load test, which we utilize to benchmark a live example and quantify the consumption of computing resources for each one of its microservices. Each running microservice has a CPU limit of 1 core with no memory limit. Consequently, the initial memory limit of each container is that of the server physical memory, namely, 32GB. We have also used the load test to determine the threshold values of microservices in *Socks Shop*.

2) LOAD GENERATION

We have used *Locust* [35], an open-source load testing tool, as the workload generator. *Locust* has been extensively tested and successfully used to swarm systems with millions of simultaneous users. Our own load test has been performed

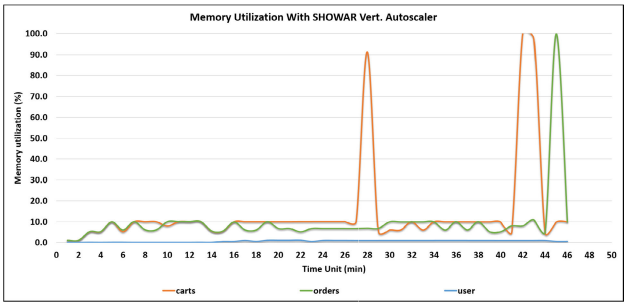


FIGURE 16. The average memory utilization for three microservices during the load test with the SHOWAR vertical autoscaler.

under the assumptions that the number of users ranges from 5 to 50 and that the spawn rate ranges from 10 to 100. This workload mimics the typical browsing behavior of most users visiting the socks-shop store. It adheres to a closed workload model and incorporates operations such as visiting the home page, logging in, adding items to the basket, going to checkout, and placing an order. The aggregated number of requests during load testing is in excess of 235,000 requests, and the number of requests per second (RPS) ranges from 18 RPS to 90 RPS. The total RPS and the number of users are shown in Figure 13. The workloads include activity bursts that are used to evaluate the robustness of the autoscaler.

B. REACTIVE AUTOSCALERS

The proactive horizontal autoscaler is compared with the reactive K8s-HA configured according to Eq. (5), by setting the target CPU utilization to 70%, and replacing the predicted metric with the 95^{th} percentile of the collected metrics in the data frame.

The vertical autoscaler is compared with one of the latest state-of-the-art vertical autoscalers, namely, SHOWAR [18]. SHOWAR is a reactive autoscaler that uses the running mean μ and standard deviation σ over a window of N samples of the memory usage time series to estimate the memory resource, M, a container needs according to the formula $M = \mu + 3 * \sigma$. The scaling decision is made only when M changes by more than 15% compared to the previous observation. The SHOWAR vertical autoscaler is reported to have very good performance with less memory allocation compared with K8s and Autopilot [18]. In the comparisons with our own autoscaler, we have used our own implementation of SHOWAR.

C. COMPARISONS

Throughout the load tests, the usage of resources of all running containers is collected and considered for normalization. CPU and memory utilizations are then analyzed along with the number of replicas and the other resources of the various workloads. Furthermore, the failure rates are logged to quantify the robustness of the autoscaler. Figures 14 and 15 illustrate the average memory utilization of microservices during load testing with our RF proactive autoscaler and the SHOWAR vertical autoscaler, respectively. Under the proactive autoscaler, all containers have an average usage of more than 30% regardless of how the load changes. On the other hand, with the SHOWAR vertical autoscaler, the average memory utilization is less than 25%.

Memory utilization for *payment*, *catalogue*, and *user* microservices is less than 50% under the proactive RF autoscaler. This is due to the fact that the autoscaler cannot set a memory limit less than 20 MB. Since these microservices are assigned the lowest possible memory limit, there is no way to increase memory utilization. The other microservices maintain a memory utilization of more than 50% most of the time.

With the SHOWAR vertical autoscaler, there are bursts in memory utilization during the load test, as shown in Figure 16. The memory utilization for the *carts* and *user* microservices increases for a few minutes to more than 90%, then drops to around 10%. These bursts cause OOM errors and increase the failure rate. Failure requests occur since the back-end services are not able to fulfill the requests even when the load is very low (10 users and 5 spawn rate) due to the small amount of memory assigned, which causes OOM errors. Such failure is the main drawback of reactive autoscalers. For example, under the SHOWAR autoscaler, the *post* and *delete* HTTP requests of the *cart* microservice have a failure rate of 12 failures/minute. On the other hand, under the proactive autoscaler of this work, the failure rate of the *delete* request is 0.0 failures/minute, and of the *post* request is 0.168 failures/minute. Finally, SHOWAR has $O(N)$ complexity due to the calculations of mean and standard deviations, while the complexity of the proactive vertical autoscaler is $O(1)$.

For the proactive horizontal autoscaling, one CPU core is found to be sufficient for most containers except for *shipping*. In addition, there is no need to add replicas to meet the high number of requests. For the *shipping* microservice, the horizontal autoscaler scales up when the load reaches the maximum of 50 users to meet the high number of requests. During this load, the CPU utilization of the *shipping* container is around 50%.

On the other hand, the reactive K8s Horizontal Pod Autoscaler (HPA) scales many microservices up and down based on the 95^{th} percentile. For instance, with only 30 users, K8s HPA adds many replicas for the *shipping* microservice even though the CPU utilization is not high and the load is considered moderate. In contrast, the proactive horizontal autoscaler adds more replicas only when the load is high, and the predicted CPU utilization is more than the threshold.

In terms of reliability and performance, the reactive autoscalers have suffered more failures and their response time has been longer when compared with the proactive autoscaler. Specifically, the system average response time is 22*ms* with the reactive autoscalers and 14*ms* with the proactive autoscaler. In addition, the max 95^{th} percentile of the response time under various loads is 250*ms* with the proactive autoscaler and 5100*ms* (more than 20X!) with the reactive ones.

In summary, we have found that scaling the containers proactively with the RF predictive model maximizes resource utilization and mitigates resource over- and under-provisioning in all load tests. The RF model can accurately predict the resource utilization of a microservice mesh and help to allocate them both in batch and in interactive workloads.

VI. CONCLUSION

In this paper, we have presented a robust, proactive, horizontal, and vertical autoscaler. The core of the proactive autoscaler is a predictive random forest (RF) model that predicts future CPU and memory utilization for containerized microservices. The forecast values are then used by the autoscaler to find the optimal number of microservice replicas and the amount of memory needed. Experimental evaluations have shown that the RF model can accurately predict CPU and memory utilization of dynamic microservice workloads. Compared to industry and state-of-the-art reactive autoscalers, the proposed proactive autoscaler has resulted in significantly improved response time (60%) and tail latency (19.6X). Furthermore, the proactive autoscaler has virtually eliminated out-of-memory failures, thus resulting in much improved memory utilization.

The proposed predictive RF model is a deterministic, supervised ML model that is used to predict the hardware resource usage of the underlying dynamic cloud workloads. Although the RF model used is very robust and accurate, its expansion to dynamically account for new features in cloud workloads is a challenge. One way to address such challenge is to use progressive learning, which is expected to converge faster to a better generalization model while preserving model accuracy.

REFERENCES

[1] A. Yousafzai, A. Gani, R. M. Noor, M. Sookhak, H. Talebian, M. Shiraz, and M. K. Khan, "Cloud resource allocation schemes: Review, taxonomy, and opportunities," *Knowl. Inf. Syst.*, vol. 50, no. 2, pp. 347–381, 2017.

[2] H. Singh, A. Bhasin, and P. R. Kaveri, "QRAS: Efficient resource allocation for task scheduling in cloud computing," *Social Netw. Appl. Sci.*, vol. 3, no. 4, pp. 1–7, Apr. 2021.

[3] W. Iqbal, M. Dailey, and D. Carrera, "SLA-driven adaptive resource management for web applications on a heterogeneous compute cloud," in *Proc. IEEE Int. Conf. Cloud Comput.* Berlin, Germany: Springer, Dec. 2009, pp. 243–253.

[4] R. Han, L. Guo, M. M. Ghanem, and Y. Guo, "Lightweight resource scaling for cloud applications," in *Proc. 12th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput.*, May 2012, pp. 644–651.

[5] Kubernetes. (2020). *Kubernetes*. Accessed: Feb. 16, 2020. [Online]. Available: https://kubernetes.io/

[6] P. Kang and P. Lama, "Robust resource scaling of containerized microservices with probabilistic machine learning," in *Proc. IEEE/ACM 13th Int. Conf. Utility Cloud Comput. (UCC)*, Dec. 2020, pp. 122–131.

[7] I. Prachitmutita, W. Aittinonmongkol, N. Pojjanasuksakul, M. Supattatham, and P. Padungweang, "Auto-scaling microservices on IaaS under SLA with cost-effective framework," in *Proc. 10th Int. Conf. Adv. Comput. Intell. (ICACI)*, Mar. 2018, pp. 583–588.

[8] A. Goli, N. Mahmoudi, H. Khazaei, and O. Ardakanian, "A holistic machine learning-based autoscaling approach for microservice applications," in *Proc. 11th Int. Conf. Cloud Comput. Services Sci. (CLOSER)*, 2021, pp. 190–198.

[9] M. Abdullah, W. Iqbal, J. L. Berral, J. Polo, and D. Carrera, "Burst-aware predictive autoscaling for containerized microservices," *IEEE Trans. Serv. Comput.*, vol. 15, no. 3, pp. 1448–1460, May 2022.

[10] Y. Zhang, W. Hua, Z. Zhou, E. Suh, and C. Delimitrou, "Sinan: Data-driven resource management for interactive microservices," ML Comput. Archit. Syst., Washington, DC, USA, Tech. Rep. NSF-PAR ID: 10165231, 2020.

[11] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, 2001.

[12] O. Anisfeld, E. Biton, R. Milshtein, M. Shifrin, and O. Gurewitz, "Scaling of cloud resources-principal component analysis and random forest approach," in *Proc. IEEE Int. Conf. Sci. Electr. Eng. Isr. (ICSEE)*, Dec. 2018, pp. 1–5.

[13] S. Choochotkaew, T. Chiba, S. Trent, and M. Amaral, "Run wild: Resource management system with generalized modeling for microservices on cloud," in *Proc. IEEE 14th Int. Conf. Cloud Comput. (CLOUD)*, Sep. 2021, pp. 609–618.

[14] H. Mohamed and O. El-Gayar, "End-to-end latency prediction of microservices workflow on kubernetes: A comparative evaluation of machine learning models and resource metrics," in *Proc. 54th Annu. Hawaii Int. Conf. Syst. Sci.*, 2021, p. 1717.

[15] Y. Li, D. Ou, C. Jiang, J. Shen, S. Guo, Y. Liu, and L. Tang, "Virtual machine performance analysis and prediction," in *Proc. Int. Conf. Commun., Comput., Cybersec., Informat. (CCCI)*, Nov. 2020, pp. 1–5.

[16] J. P. Wong, "HyScale: Hybrid scaling of dockerized microservices architectures," Ph.D. dissertation, Dept. Elect. Comput. Eng., Univ. Toronto, Toronto, ON, Canada, 2019.

[17] A. A. Khaleq and I. Ra, "Intelligent autoscaling of microservices in the cloud for real-time applications," *IEEE Access*, vol. 9, pp. 35464–35476, 2021.

[18] A. F. Baarzi and G. Kesidis, "SHOWAR: Right-sizing and efficient scheduling of microservices," in *Proc. ACM Symp. Cloud Comput.*, Nov. 2021, pp. 427–441.

[19] S. Dinesh. (2018). *Kubernetes Best Practices: Resource Requests and Limits*. Accessed: Feb. 5, 2022. [Online]. Available: https://cloud.google.com/blog/products/containers-kubernetes/kubernetes-best-practices-resource-requests-and-limits

[20] A. Kwan, J. Wong, H.-A. Jacobsen, and V. Muthusamy, "HyScale: Hybrid and network scaling of dockerized microservices in cloud data centres," in *Proc. IEEE 39th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jul. 2019, pp. 80–90.

[21] K. R. Rao, D. N. Kim, and J. J. Hwang, *Fast Fourier Transform: Algorithms and Applications*. Dordrecht, The Netherlands: Springer, 2010.

[22] The Grid Workloads Archive. *The Grid Workloads Datasets*. Accessed: Mar. 22, 2021. [Online]. Available: https://github.com/acmeair/acmeair

[23] D. Wang, P. Ram, D. K. I. Weidele, S. Liu, M. Müller, J. D. Weisz, A. Valente, A. Chaudhary, D. Torres, H. Samulowitz, and L. Amini, "AutoAI: Automating the end-to-end AI lifecycle with humans-in-the-loop," in *Proc. 25th Int. Conf. Intell. User Interfaces Companion*, Mar. 2020, pp. 77–78.

[24] J. Chen, K. Li, Z. Tang, K. Bilal, S. Yu, C. Weng, and K. Li, "A parallel random forest algorithm for big data in a spark cloud computing environment," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 4, pp. 919–933, Apr. 2017.

[25] R. Caruana and A. Niculescu-Mizil, "An empirical comparison of supervised learning algorithms," in *Proc. 23rd Int. Conf. Mach. Learn.*, 2006, pp. 161–168.

[26] (2021). *Scikit Learn*. Accessed: Oct. 20, 2021. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.RobustScaler.html#sklearn.preprocessing.RobustScaler

[27] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand, and J. Wilkes, "Autopilot: Workload autoscaling at Google," in *Proc. 15th Eur. Conf. Comput. Syst.*, Apr. 2020, pp. 1–16.

[28] S. Shen, V. Van Beek, and A. Iosup, "Statistical characterization of business-critical workloads hosted in cloud datacenters," in *Proc. 15th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput.*, May 2015, pp. 465–474.

[29] C. Pascual. (2018). *Tutorial: Understanding Regression Error Metrics in Python*. [Online]. Available: https://www.dataquest.io/blog/understanding-regression-error-metrics/

[30] 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 É. Duchesnay. (2022). *Permutation Feature Importance*. Accessed: Jan. 3, 2023. [Online]. Available: https://scikit-learn.org/stable/modules/permutation_importance.html

[31] 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 É. Duchesnay, "Scikit-learn: Machine learning in Python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, Jan. 2011.
[32] M. T. Ribeiro, S. Singh, and C. Guestrin, "'Why should I trust you?' Explaining the predictions of any classifier," in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2016, pp. 1135–1144.
[33] L. A. Qassem, "Microservice architecture and efficiency model for cloud computing services," Ph.D. dissertation, Dept. Elect. Eng. Comput. Sci., Khalifa Univ., Abu Dhabi, UAE, 2022.
[34] Weaveworks. (2016). *Containersolutions: Socks Shop—A Microservices Demo Application*. [Online]. Available: https://microservices-demo.github.io/
[35] Locust. (2020). *An Open Source Load Testing Tool*. Accessed: Feb. 11, 2020. [Online]. Available: https://locust.io/

LAMEES M. AL QASSEM received the M.Sc. and Ph.D. degrees from Khalifa University, Abu Dhabi, United Arab Emirates, in 2017 and 2022, respectively. She is currently a Postdoctoral Fellow at Khalifa University. Her M.Sc. thesis focused on Arabic natural language processing (NLP) and artificial intelligent. Her Ph.D. thesis focused on microservice architecture and efficiency models for cloud computing services. She has published in the areas of educational technology, including augmented and virtual reality, Arabic natural language processing, cloud computing, and microservice architectures. She received the Best Paper Award from the 2019 UAE Graduate Student Research Competition (GSRC) for her work on Arabic NLP.

THANOS STOURAITIS (Life Fellow, IEEE) received the Ph.D. degree from the University of Florida. He is currently a Professor with the Department of Electrical Engineering and Computer Science, Khalifa University, United Arab Emirates, and a Professor Emeritus of the University Patras. He has served on the faculties of The Ohio State University, the University of Florida, New York University, and The University of British Columbia. He also served on the National Scientific Board for Mathematics and Informatics of Greece. He was a Founding Council Member of the University of Central Greece. Along with several textbooks, he has authored about 200 technical papers, several book chapters, and holds one USA patent on DSP processor design. He has led several DSP processor design projects funded by the European Union, American organizations, and the Greek government and industry. His current research interests include AI hardware systems, signal and image processing systems, computer arithmetic, and design and architecture of optimal digital systems with emphasis on cryptographic systems. He is a fellow of IEEE for his contributions in digital signal processing architectures and computer arithmetic. He has served as an Editor or a Guest Editor for numerous technical journals, including IEEE TRANSACTIONS ON COMPUTERS, IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS, and IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS, and the General Chair and/or Technical Program Committee Chair for many international conferences, including IEEE ISCAS, AICAS, SiPS, ICECS, and GlobeCom. He received the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS SOCIETY Guillemin-Cauer Award. He has served IEEE in many ways, including as the President, from 2012 to 2013, of its Circuits and Systems Society.

ERNESTO DAMIANI (Senior Member, IEEE) is the Director of the Center for Cyber-Physical System at Khalifa University, Abu Dhabi, and a Full Professor at the Department of Computer Science, Università degli Studi di Milano, Italy, where he leads the SESAR Lab, and President of the National Interuniversity Consortium for Computer Science. He has hold a Visiting Positions at a number of international institutions, including George Mason University, Virginia, USA, Tokyo Denki University, Japan, La Trobe University, Melbourne, Australia, and the Institut National des Sciences Appliquées (INSA), Lyon, France. He is a Fellow of the Japanese Society for the Progress of Science. He has been the Principal Investigator in a number of large-scale research projects funded by the European Commission, the Italian Ministry of Research and by private companies such as British Telecom, Cisco Systems, SAP, Telecom Italia, Siemens Networks (now Nokia Siemens). He has coauthored more than 700 scientific articles and many books, including "Open Source Systems Security Certification" (Springer 2009). His research interests include artificial intelligence, machine learning, cyber-physical systems, secure service-oriented architectures, privacy-preserving big data analytics, and cyber-physical systems security. He serves as the Editorial Board of several international journals; among others, he is the EIC of the International Journal on Big Data and of the International Journal of Knowledge and Learning. He is an Associate Editor of the IEEE TRANSACTIONS ON SERVICE-ORIENTED COMPUTING and of the IEEE TRANSACTIONS ON FUZZY SYSTEMS. In 2008, he was nominated ACM Distinguished Scientist and received the Chester Sall Award from the IEEE Industrial Electronics Society.

IBRAHIM (ABE) M. ELFADEL (Senior Member, IEEE) received the Ph.D. degree from the Massachusetts Institute of Technology (MIT), in 1993. He is currently a Professor of electrical engineering and computer science at Khalifa University, Abu Dhabi, United Arab Emirates. Prior to his current academic position, he was with the corporate CAD organizations at IBM Research and the IBM Systems and Technology Group, Yorktown Heights, NY, USA, where he was involved in the research, development, and deployment of CAD tools and methodologies for IBM's high-end microprocessors. His current research interests include the IoT platform prototyping; energy-efficient edge and cloud computing; secure IoT communications; embedded digital-signal processing; and computer-aided design for VLSI, MEMS, and silicon photonics. He was a recipient of six Invention Achievement Awards, one Outstanding Technical Achievement Award, and one Research Division Award, all from IBM, for his contributions in the area of VLSI CAD. His other awards include the D. O. Pederson Best Paper Award from the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN, the SRC Board of Directors Special Award for pioneering semiconductor research in Abu Dhabi, the Best Paper Award from the IEEE Conference on Cognitive Computing, Milan, Italy, in July 2019 and 2022 Service Award from the International Federation of Information Processing (IFIP). He is an Associate Editor of the IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS. He has served on the Technical Program Committees of several leading conferences, including DAC, ICCAD, ASPDAC, DATE, ISCAS, VLSI-SoC, ICCD, ICECS, and MWSCAS. He was the General Co-Chair of the 25th IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC 2017), Abu Dhabi, UAE.