IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 20 September 2022, accepted 12 October 2022, date of publication 14 October 2022, date of current version 20 October 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3214985

RESEARCH ARTICLE

Predictive Hybrid Autoscaling for Containerized Applications

DINH-DAI VU, MINH-NGOC TRAN, AND YOUNGHAN KIM, (Member, IEEE)
School of Electronic Engineering, Soongsil University, Seoul 06335, South Korea

Corresponding author: Younghan Kim (younghak@ssu.ac.kr)

This work was supported in part by the Institute of Information and Communications Technology Planning and Evaluation (IITP) Grant by the Korea Government Ministry of Science and ICT (MSIT) under Grant 2020-0-00946 (Development of Fast and Automatic Service recovery and Transition software in Hybrid Cloud Environment), and Grant 2022-0-01015 (6GRC).

ABSTRACT One of the main challenges in deploying container service is providing the scalability to satisfy the service performance and avoid resource wastage. To deal with this challenge, Kubernetes provides two kinds of scaling mode: vertical and horizontal. Several existing autoscaling methods make efforts to improve the default autoscalers in Kubernetes; however, most of these works only focus on one scaling mode at the same time, which results in some limitations. Only horizontal scaling may lead to low utilization of containers due to the fixed amount of resources for each instance, especially in the low-request period. In contrast, only vertical scaling may not ensure the quality of service (QoS) requirements in case of bursty workload due to reaching the upper limit. Besides, it is also necessary to provide burst identification for auto-scalers to guarantee service performance. This paper proposes a hybrid autoscaling method with burst awareness for containerized applications. This new approach considers a combination of both vertical and horizontal abilities to satisfy the QoS requirement while optimizing the utilization of containers. Our proposal uses a predictive method based on the machine learning technique to predict the future demand of the application and combines it with a burst identification module, which makes scaling decisions more effective. Experimental results show an enhancement in maintaining the response time below the QoS constraint whereas remaining high utilization of the deployment compared with existing baseline methods in single scaling mode.

INDEX TERMS Cloud computing, Kubernetes, autoscaling, machine learning, workload forecasting.

I. INTRODUCTION

Scalability is a crucial feature of cloud computing to automatically scale the resources to improve the service performance in response to dynamic workload [1]. The resource requirements are potentially influenced by complicated demand processes, which can be difficult to predict in advance. Resource under-provisioning leads to performance degradation and service unavailability, which may result in revenue loss for providers. In contrast, resource over-provisioning normally leads to low resource utilization of systems, which is cost-inefficient [2]. Therefore, autoscaling plays an important role in optimizing resources and avoiding service level objective (SLO) violations.

In recent years, containers have gradually replaced virtual machines as the runtime environment of applications. Kubernetes, the most popular container orchestrator, attempts to address the scaling problem by providing both Horizontal Pod Autoscaler (HPA [3]) and Vertical Pod Autoscaler (VPA [4]). A pod is the smallest entity that consists of one or more application containers in Kubernetes. HPA increases or decreases the number of pods (or ''replicas''), whereas VPA changes the size of the pod in terms of computing resources such as CPU and memory. Vertical and horizontal scalabilities are promising strategies to manage resources efficiently at runtime and in an adaptive means for containerized applications in Kubernetes.

HPA and VPA are the reactive autoscalers, which shows the limitations that slowly adapt to fluctuated workloads. Recently, many studies with various techniques in the autoscaling area mostly focus on improving the performance

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

of these two default Kubernetes scalers separately by predictive method to ensure application performance [5]. The existing predictive autoscaling methods are mostly based on diverse techniques, such as the statistical model [6], reinforcement learning [7], and machine learning [8] techniques. Despite the better performance of custom horizontal and vertical autoscalers, these two scaling types still have several constraints leading to the limitations of using only a single scaling type for the application at the same time.

The horizontal scaling could avoid SLO violations but has the limitation of optimizing resource utilization. The reason is that the operator needs to set a fixed size (CPU and memory resource) for a pod in the deployment in advance; then, it is unchangeable. It may lead to low utilization of pods during the low-request period from users, especially for some resource-intensive applications that initially require a larger amount of resources for one pod. If the Kubernetes cluster has many different services running on many pod instances like this, it may result in low cluster resource utilization, which is cost-inefficient [9]. In addition, it required a mechanism to assign appropriate initial resources for the deployment. The vertical scaling method will be essential to optimize resource utilization by dynamically resizing the fixed instance in this case and solving the problem of setting resources for the deployment.

On the other hand, only vertical scaling shows the limitation of satisfying the quality of service (QoS) despite optimizing resource utilization. Firstly, vertical scaling only works with the system usage metrics, which are not good scaling indicators for application performance [10]. In addition, vertical scaling causes service disruption because the pod must be restarted during the scaling duration. Finally, the performance of pod instances may reach the saturation point and does not increase significantly when the pod size sometimes increases [11], which leads to SLO violations, especially in heavy workloads. In this case, using horizontal scaling to assist the vertical one in scaling out the saturated pods will be more effective in ensuring the QoS requirements.

Most existing studies focus on only horizontal scaling or vertical scaling mode. With other approaches, we see that the problem of using a single scaling type can be solved effectively by the hybrid method, which is the combination of both vertical and horizontal scaling. Hybrid scaling means changing the number of pods and the assigned resource for a pod simultaneously. Currently, choosing only one type of autoscaling mode at a time in Kubernetes is highly recommended to avoid interference from others when they use the same type of metrics [4]. Only a few studies are nearest to the hybrid approach by cascading process that uses VPA to determine the resource configuration for the deployment before using HPA to scale horizontally at run time [11], [12]. Because the VPA does not run again once the HPA starts, the new pods may suffer from low resource utilization even though they can satisfy user requests. Therefore, it is necessary to make a hybrid scaling system that simultaneously decides both vertical and horizontal scaling actions.

Besides scaling type, the autoscaler is significantly affected by the accuracy in resource estimation to make scaling decisions. Recently, the effectiveness of machine learning techniques in time series forecasting can improve the accuracy of predicting future demand based on historical metrics [13]. By using machine learning to support scaling decisions, we may turn the default reactive method in Kubernetes into the predictive method and quickly satisfy user demand. In addition, because cloud applications normally experience burst workloads (a sudden sharp increase in workload rate) [14], a burst detection mechanism should be combined with workload forecasting. This combination can avoid the saturated pod performance problem of the vertical scaling part in hybrid scaling decisions.

In this paper, we propose the horizontal pod autoscaler with varying pod resources for Kubernetes, and the hybrid scaling method for containerized applications in Kubernetes to solve the above limitations. In summary, the main contributions of this paper include:

- Propose the novel hybrid autoscaling method that aims to satisfy QoS while optimizing resource utilization for containers in Kubernetes.
- Provide the rolling update deployment mechanism to avoid service disruption when changing both the number of pods and the resource per pod in hybrid scaling execution.
- Combine the predictive scaling method with burst identification to optimize hybrid scaling decisions.
- Implement the proposed hybrid scaling solution and compare our proposal with state-of-the-art reactive and proactive single autoscaling methods as the baselines.

The rest of this paper is structured as follows: Section II mentions the related work. We present our autoscaling solution in section III. The experiment setup is described in section IV. Section V shows our evaluation and mentions the key points. We conclude the paper in Section VI.

II. RELATED WORK

In this section, we mention recent works that fall close to our proposal. Since scalability and dynamism are appealing features of cloud infrastructure, many previous studies work in this area to optimize the scaling strategy for both virtual machine and container environments. We mostly focus on studies related to the scaling method for containers in the Kubernetes environment.

For vertical scaling, the study [15] proposes a proactive method by using the statistic model Holt-Winters (HW) exponential smoothing and Long Short-Term Memory (LSTM) deep learning model to predict the CPU usage for improving the resource estimation in scaling decisions to optimize the pod resource. We argue that it still has the limitation that poorly satisfies QoS. Firstly, vertical scaling only works with low-level metrics (CPU, memory) for scaling indicators that cannot directly reflect the QoS requirement of applications. Secondly, they did not mention the negative effect of service disruption. Rattihalli et al. [16] presented RUBAS optimizing the default VPA in an effort to reduce the disruption time

of service during vertical scaling execution. RUBAS used a combination of container migration and checkpoint technologies to reduce the start-up time of the pod, but RUBAS still does not satisfy the QoS requirement. In general, few studies focus on vertical scaling type due to the limitation of service disruption even though it can optimize resource utilization. There is an ongoing work for in-place pod resource adjustment that aims to avoid restarting when updating pod resources in Kubernetes [17]. If this feature in Kubernetes is released, exiting vertical autoscaling studies would also become more viable.

In contrast, most studies focus on horizontal scaling aiming to avoid SLA violations in container services. Khazaei et al. [18] proposed Elascale, which enables autoscaling and monitoring services for cloud systems, making scaling decisions based on a set of resource usage, and network utilization. We argue that this simplification is a limiting indicator of the scaling performance. Instead of using default system metrics from the metrics server component in Kubernetes, Nguyen et al. [9] mentioned several application-level metric indicators that affected the performance of HPA, and the author also shows the direction for optimizing HPA. But the above studies are still threshold-based reactive methods.

According to the study [7], a threshold-based scaling policy, such as the default Kubernetes HPA, is not well suited to meet the QoS requirements of latency-sensitive applications. The author uses the reinforcement learning (RL) approach with model-free and model-based policies to compare with the default threshold-scaling policy of Kubernetes based on CPU utilization as the indicator. Khaleq and Ra [10] proposed a scaling module using RL agents on microservices log data for the real-time system with response time as a QoS metric.

Besides of RL approach, workload forecasting is an important research area that researchers use as a predictive method in various domains. There have been some efforts to use the forecasted workload to address the autoscaling problem. The authors of [6] proposed an adaptive prediction method using statistic models, such as the auto-regressive integrated moving average (ARIMA) model, simple and extended exponential smoothing. Iqbal et al. [19] proposed two different solutions, Ordinary Least Squares (OLS) and Non-Negative Least Square (NNLS) workload pattern prediction for proactive auto-scaling of web applications. Abdullah et al. [20], [21] presented the autoscaling method for microservice using machine learning techniques. The authors used different regression methods, such as linear regression (LR), and elastic net (EN) regression, to learn the predictive autoscaling model. The authors in [22] apply the machine learning LSTM model for proactive scaling policies instead of the default algorithm in HPA in Kubernetes. Toka et al. [8] proposed HPA+, which proposes predictive autoscaling to improve the default Kubernetes HPA by exploiting the multi-forecast, including auto-regressive (AR) statistic model, RL model, and LSTM deep learning model. Then HPA+ applies the best prediction result from these forecasting models to tune the number of pods.

Despite these existing proactive horizontal scaling methods outperforming default HPA due to the quick adaption to satisfy the QoS requirement, the general problem for these above studies is the limitation of optimizing resource utilization, especially in the low-request periods. The reason is that the horizontal scaling type cannot change the resource for the instances along with the number of instances during the scaling processes.

We see the advantage of vertical scaling when optimizing resource utilization but the limitation of avoiding SLO violations. In contrast, only horizontal scaling has the drawback of optimizing the pod utilization even if it can satisfy the QoS requirements well. It is necessary to achieve the hybrid scaling method, which can leverage the advantage of both horizontal and vertical scaling methods. Only a few existing studies are nearest to the hybrid approach. The authors in [14] propose the hybrid auto-scaled service cloud model to ensure (QoS) for smart campus-based applications. They vertically scale the server in normal load and switch to horizontal scaling mode in burst cases. However, it is not used in the container environment. Libra [11] and KOSMOS [12] are nearest to the hybrid approach but the whole process is cascaded. These works use vertical scaling to set a suitable limit for pods before using the horizontal scaling at run time, and KOSMOS still does not have evaluation results to prove. The cascading process may lead to low pod utilization for the newly created instances in fluctuated traffic cases. The reason is that all pods in a Kubernetes deployment have the same assigned resources, the decided amount of resources found by the vertical scaling process will be applied to all pods instantiated by the horizontal scaling process.

In the Kubernetes environment, no study proposes the hybrid scaling method in which hybrid scaling decisions are determined simultaneously for the deployment of the application at run time. Therefore, we propose the horizontal pod autoscaler with varying pod resources for Kubernetes, the hybrid autoscaling method, that can change both the number of pods and the resource per pod that aim to satisfy the QoS constraint and optimize the resource utilization. Dealing with the fluctuating workload, motivated by the huge success of deep learning, we use the Bidirectional Long Short-term Memory (Bi-LSTM) model and combine it with burst identification to improve the hybrid scaling decisions.

III. PROPOSED SYSTEM

In this section, we describe the components of our framework design. We present the set of requirements, machine learning model, and provision algorithm, which aims to provide the predictive hybrid scaling on Kubernetes system. Then, we describe the workflow, which automatically scales the number of application pods and the resource per pod according to real-time metrics.

A. COMPONENT OVERVIEW

Our autoscaler will run on top of the API of Kubernetes to provide an automated scaling mechanism for container applications. Kubernetes manages the container application in the deployment object and exposes service mapping with

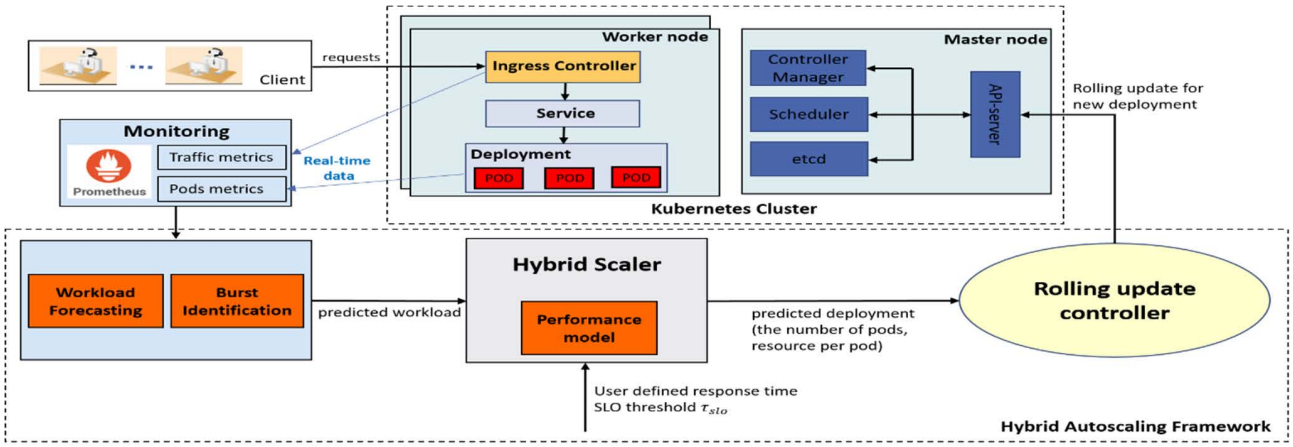


FIGURE 1. Proposed system architecture.

this deployment so that users can access the application. Like the default autoscaler in Kubernetes, we need to collect metrics to give scaling decisions. However, instead of using the default metrics server component, we use a custom monitoring framework, as shown in Fig. 1. The reason is that the metric server component in Kubernetes can only get the metrics related to the resource usage of the pod. Besides optimizing resource cost, our scaling goal is to avoid SLO violations; therefore, getting metrics at the application level is important. By using an ingress controller to control the traffic to the service, we can collect runtime metrics from the serving pod. Based on the capabilities of open-source monitoring tools, we use Prometheus [23] as our data scraper. It scrapes standard Kubernetes components through the default Kubernetes API. Besides the default metrics, with additional extensions, Prometheus can also scrape service performance metrics that expose from the ingress controller.

The overall proposed hybrid scaling framework is illustrated in Fig. 1. The flow and components of the framework are labeled to explain the working of the system. The general module workflow steps are described below:

1) Based on the real-time workload from the monitoring framework, the workload forecasting component firstly loads the deep learning model to predict future demand. It bases on time-series prediction to forecast the workload of the next interval. We describe this in section III-B.
2) Burst identification use last k workload observations with the predicted values to detect the burst. We also describe this in section III-B.
3) The hybrid scaler component, explained in section III-C, provides a provisioning algorithm with the support of the performance model for tuning the hybrid scaling decisions based on predicted workload and burst signal.
4) Rolling update controller provides the procedure to execute the hybrid scaling decisions (or change both the number of pods and resources per pod) without service disruption. We explain this in section III-D.

B. WORKLOAD FORECASTING AND BURST IDENTIFICATION

1) WORKLOAD FORECASTING

Short-term demand forecasting has been the focus of researchers of various application domains for many years, and machine learning techniques have become more prevalent, especially in time-series prediction. Serval existing studies use statistic models such as ARIMA, ARMA, Moving Average, OLS, NNLS, and regression (LR, EN) [6], [24], [19], [20], [21] to predict future workloads. However, statistical models are hard to predict the fluctuating workloads. When dealing with non-linear and non-stationary time-series datasets, advanced deep learning-based methods show better performance. Recurrent Neural Network (RNN) is a type of artificial neural network that can be used for time series forecasting. In RNN, the output of the several previous timesteps is used as input to the next, which allows them to keep a memory. LSTM is a type of RNN with the capability of learning long-term dependencies that can solve the disadvantage of the vanishing gradient problem in RNN. LSTM network has a similar architecture to RNN, with the exception that an LSTM cell replaces a neuron in the original RNN [25]. Each LSTM cell has three gates, input gate, output gate, and forget gate. The cell has the responsibility for managing the dependencies among the components in the input sequences. The gate input controls the flow of inputting a new value into the cell. The forget gate considers which data should be dropped from the cell. Finally, the output gate chooses which information from the cell is used to compute the output activation of the LSTM units.

Bidirectional LSTM (Bi-LSTM), an extension of LSTM, improves the understanding of sequence patterns by using both forward and backward dependencies. In contrast with [22] which uses LSTM, we use the Bi-LSTM model for our hybrid scaler. The architecture of the Bi-LSTM network includes two LSTM networks stacked on top of each other. The first LSTM is trained on the original input sequence (forward). The reverse form of the input sequence is given to the second LSTM (backward). This bidirectional learning enhances the learning ability of the model.

TABLE 1. Error metrics on test set for workload forecasting model.

Metric	Wikipedia trace							FIFA trace						
	ARIMA	LR	EN	OLS	NNLS	LSTM	Bi-LSTM	ARIMA	LR	EN	OLS	NNLS	LSTM	Bi-LSTM
RMSE	0.985	0.872	0.778	0.921	0.879	0.742	**0.655**	0.875	0.709	0.636	0.910	0.880	0.509	**0.436**
MAE	0.699	0.643	0.589	0.730	0.620	0.542	**0.466**	0.523	0.623	0.419	0.660	0.653	0.323	**0.289**

Algorithm 1 Online Burst Identification Algorithm

Require: Y list of k last points, *influence*, *threshold*, predicted workload ω_{pred}

```
burst_signal = []
avg = mean(Y[1],...,Y[k])
std = std(Y[1],...,Y[k])
while True do
  if (ω_pred − avg) > threshold ∗ std then
    burst_signal.append(ON)
    x = influence ∗ ω_pred + (1 - influence) ∗ Y[k]
    avg = mean(Y[2],...,Y[k], x)
    std = std(Y[2],...,Y[k], x)
  else
    burst_signal.append(OFF)
    x = ω_pred
    avg = mean(Y[2],...,Y[k], x)
    std = std(Y[2],...,Y[k], x)
  end if
  Y.append(ω_pred)
  Y.remove(Y[1])
  Waiting for the new predicted workload ω_pred
end while
```

We evaluate our proposed model using two real web server logs: Wikipedia Access trace [26] and FIFA World Cup 98 trace [27]. The request rate in these datasets is very large; hence, we scale them before training the model. We split each scaled dataset into two sets: training set and testing set with ratios of 80% and 20%, respectively, and normalized them to the range (0, 1) before training. To train Bi-LSTM models, we prepare historic sub-sequences from the dataset using the sliding window method as input sequences. This method uses n time steps as inputs to predict the next time step in a one-step-ahead prediction. We used 10 look-back steps in our work to predict the next time step. The 10 look-back steps and many other hyperparameters are fine-tuned during the training process. Our Bi-LSTM model shown in Table 1 achieves the smaller prediction error values on Root Mean Square Error (RMSE) and Mean Absolute Error (MAE) metrics compared to the ARIMA, LR, EN, OLS, NNLS, and LSTM on the test set.

2) BURST IDENTIFICATION

Cloud burstiness workload is one of the main reasons that may decrease the performance of real-time applications and lead to reduced QoS and service disruptions [28], [29]. Burst awareness will help to improve the scaling decisions to avoid SLO violations. About the burst-aware methodology, Abdullah et al. [21] use the number of calculated pods as the main parameter for burst detection due to their work only on horizontal scaling. In our work, the burst detection is based on the predicted workload, which is more accurate since we provide the hybrid scaling mechanism. In burst situations, the workload changes dramatically, and it is also heavy. Therefore, in our work, we prefer to maintain maximum resources for a pod and horizontally scale the number of pods to adapt in burst cases quickly; otherwise, we change both the number of pods and the resource per pod to satisfy the QoS, whereas optimizing the pod utilization. In this paper, we do not differentiate between different types of bursts, such as bursts of different duration or intensities. We only provide the burst signal directly on the predicted workload that needs to adapt quickly to satisfy the QoS.

Algorithm 1 shows our online burst identification algorithm. We use the standard deviation from the moving average for burst identification. We generate a moving baseline by calculating a new value for every point dependent on k precessed points and then get how far the current point varies from this baseline. It means that the algorithm compares the newest prediction value from the forecasting model to the average of the precessed k previous points. If a new prediction value is a given *threshold* input number of standard deviations away from some moving mean, the algorithm will return the ON burst signals. We also use *influence* input to adjust the values for any point that is considered to be a burst, because otherwise, they will distort the moving baseline.

C. HYBRID SCALER

In our paper, we aim to give hybrid scaling decisions to satisfy the response time threshold as the QoS constraint whereas optimizing the average pod utilization. With different configured resource levels, the deployment may need a different number of pods to avoid SLO violations and lead to the different average pod utilization also. Therefore, we first need to build an accurate performance model to tune both the number of pods with the assigned resource per pod at a time with any prediction workload from the forecasting model. The performance model plays an important role in our work. Then we provide the hybrid provisioning algorithm that leverages the prediction from the performance model to decide the best hybrid scaling scheme.

1) PERFORMANCE MODEL

The predicted workload ω, the pod resource level r, and the response time SLO threshold τ_{slo} are used as the input of the performance model to return the number of pods n and the pod average utilization u. We use the formula as follows:

$$\varphi : (\omega, r, \tau_{slo}) \rightarrow (n, u) \quad (1)$$

TABLE 2. Comparison of error metric on the test set for performance model.

Metric	LR	RDF	SVR	XGB	MLP	DTR
MSE	0.836	0.984	0.523	0.268	0.732	**0.109**
MAE	3.27	3.51	2.98	2.67	3.14	**1.81**

We treat the model as a multiple-output regression problem since it has two outputs. For this problem, we see the potential of using machine learning to solve it. To do it, at first, we need to have the performance dataset for the training model. The initial training data for the performance model is generated by leveraging the method presented in [30] that performed a trace-driven simulation to collect the dataset. In [30], the author generated the dataset for the benchmark application on the virtual machine (VM). With the same method in [30] but for pods in Kubernetes in our case, we use default reactive HPA and linearly increase the steady workload level for the initial benchmark to achieve the performance trace. We deployed the application and generated synthetic workloads using Locust [31]. HPA automatically adds the resource instance whenever the deployment overloads for the linearly increasing workload. The benchmark application, also used in our evaluation, is described in Section IV. The performance trace that included the request rate, the response time, the number of pods, the resource per pod of deployment, and the average pod utilization were recorded during this experiment. After finishing the process of generating data, we cleaned the data and removed all the rows which showed response time violations from the dataset.

We split the collected dataset into 2 sets: training set and test set with ratios of 80% and 20%, respectively, and normalized them to the range (0, 1) before training. We used several machine learning methods to train the performance model, including Linear Regression (LR), Random Decision Forests (RDF) Regression, Support Vector Regression (SVR), XGBoost (XGB) Regression, Multilayer Perceptron regression (MLP), and Decision Tree Regression (DTR). Since it is multiple-output regression, we use the custom average MSE and MAE as the loss functions during the training process. Table 2 shows the performance of the above regression models on the test set. DTR outperforms all other regression methods by a minimum MSE and MAE. DTR is a supervised machine learning algorithm that is usually used for regression problems. Therefore, we choose DTR as a learning method for our performance model.

2) PROPOSED HYBRID AUTOSCALING ALGORITHM

Algorithm 2 shows our method to give the hybrid scaling scheme. The algorithm requires the user-defined response time threshold τ_{slo}, and all components that are present in previous sections such as the forecasting model, the burst signal from burst identification, and the performance model. Besides, in our work, we need to bound the range of resources for pods with the set R. The reason is that a single instance may have an upper-performance limit. When the supply of resources reaches a certain threshold, its capacity will not be significantly improved [11]. Like the method in [10], we use vertical scaling processes with VPA in Kubernetes to get the recommendation value and set the upper limit r_{max}. For r_{min} value, it depends on the minimum requirement of the application owner to ensure the operation of the pod.

Algorithm 2 Proposed Hybrid Autoscaling Algorithm

Require: Responset time SLO threshold (τ_{slo}), workload forecasting model, burst_signal, performance model (φ), the set of resource per pod $R = [r_{min}, r_{max}]$

```
while True do
    ω_pred ← predicted value from workload forecasting model
    predValue ← []
    if burst_signal == ON then
        (n_pred, u_pred) ← φ : (ω_pred, r_max, τ_slo)
        predValue .append (n_pred, r_max)
    else if burst_signal == OFF then
        S ← []
        for each r_i in list R do
            (n_i, u_i) ← φ : (ω_pred, r_i, τ_slo)
            S.append (n_i, r_i, u_i)
        end for
        (n_pred, r_pred) ← get (n, r) in S with u_max
        predValue.append (n_pred, r_pred)
    end if
    (n_new, r_new) = predValue.pop()
    get(n_cur, r_cur) from running system
    if(n_cur, r_cur) != (n_new, r_new) then
        Rolling update new deployment with (n_new, r_new)
    else if
        Do nothing
    end if
    Waiting for next time interval
end while
```

In each time interval, our algorithm will consume the burst signal based on the predicted workload. If the burst signal is *ON*, we prefer to set r_{max} for each pod in the deployment and predict the number of pods corresponding with r_{max} configuration to adapt to the predicted workload ω_{pred} without SLO violations; otherwise, in non-burst cases, we tune the deployment with the number of pods and the resource per pod with the optimal utilization while remaining the response time under the threshold τ_{slo}. Due to the oscillation problem, the scaler performs opposite actions frequently within a short period [5], which wastes resources and costs. We choose the length of each time interval to 60 s (1 min) after each scaling execution, which is fine-grained compared to the container environment. In the last step, we use the rolling update controller to execute the hybrid scaling scheme. It is an important step to avoid disrupting the service during the scaling execution because we change the resource configuration of the pod. We describe it in the following section.

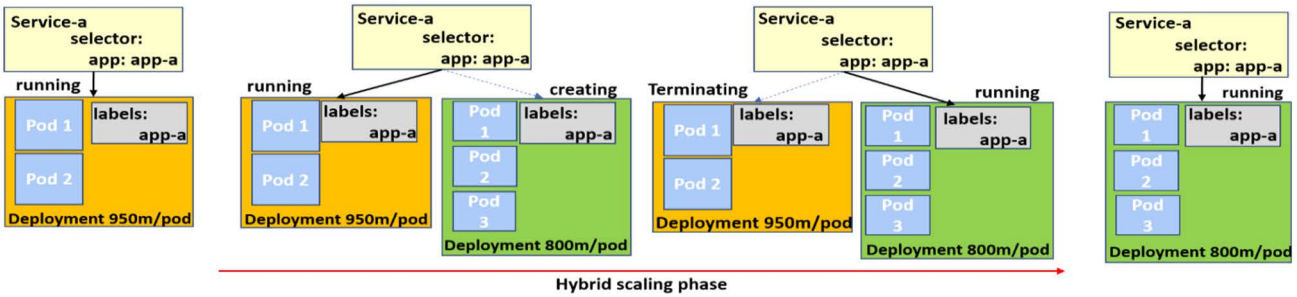


FIGURE 2. Example of the hybrid scaling procedure from 2 pods of 950 minicore CPU per pod deployment to 3 pods of 800 minicore CPU per pod deployment.

D. ROLLING UPDATE CONTROLLER

Our hybrid scaling method aims to change the number of pods and the assigned resource to the pod. This approach may lead to service disruption. To deal with this problem, we provide the rolling update controller with a non-disruptive procedure.

In Kubernetes, the application container can be deployed in a *pod*. The *deployment* object is used to ensure the desired number of pods (or the number of replicas) running and available at all times, which automatically manages ReplicaSet(RS). The scaling actions can be divided into two types. The first type is horizontal scaling, which changes the number of Pods under RS [3]. There is no disruption time of service during the horizontal scaling process. In contrast, the vertical scaling adjusts the assigned resources. Different from the horizontal scaling phase, once updating the Pod resources, the running pod will be evicted, and then Kubernetes restarts the new pod, which leads to downtime of service whenever executing the vertical scaling decisions.

Our proposed will run the hybrid scaling schemes which may include vertical updating processes. Therefore, we use the rolling update controller to avoid service disruption. We separate two main situations that may occur due to the result of the hybrid scaler component. The first situation is that the resources for each pod in the running deployment remain unchanged, but the number of pods changes. For this situation, we keep the running deployment and horizontally scale it. The second situation is that the resources for each pod need to be changed along with the number of pods in the running deployment. In this case, we will create the new deployment with the number of replicas and the resource configuration as the result of the hybrid scaler function.

During the time waiting for the new deployment up, we remain the running deployment. Once the new deployment is at running status, the controller immediately terminates the original deployment and finishes the hybrid scaling process. In Kubernetes, users access the application through the *service* object that points to the deployment by *label* [32]. When creating a new deployment, the controller will mark the same *label* as the running deployment (even though the deployment name can be different). Therefore, *service* with *selector* pointing to *label* can automatically route the user requests to the new deployment once it is running and avoid disruption time. Fig. 2 shows an example of our work when we need to execute a hybrid scaling decision. Since we do not focus on the placement problem in our work, we use the scheduler of Kubernetes with the default rule instead. Therefore, the new pods will be assigned to nodes in the cluster with the most available resource. However, the rule can be custom with the scheduler, and we also consider it for our future work.

IV. EXPERIMENTAL DESIGN AND SETUP

Our experiment uses version 1.23 of Kubernetes. We deploy the cluster with one master node and two worker nodes on three bare-metal servers that have 128-core CPU, and 64 GB memory as computational resources for each one.

A. BASELINE METHOD

We compare our hybrid scaling method with two baselines: the default HPA [3] in Kubernetes and the proactive method in [21] for only horizontal scaling baselines. For default HPA, we can directly use it since it is integrated into Kubernetes, and we refer to it as *reactive HPA*. We implemented the proactive autoscaling method similar to [21], and refer to it as *proactive HPA*. All other methods related to the vertical scaling type have trouble with service disruption; therefore, we do not use any vertical scaling method as the baseline.

B. BENCHMARK APPLICATION

We experiment in Kubernetes environment and at its current state, Kubernetes does not solve the bandwidth problem for containers, therefore we do not consider the bandwidth-intensive applications. For assigned memory, we assumed that need to configure the minimum memory requirement to ensure the performance of pods. Hence, we only run the experiment with a CPU-bound application. The applications that integrate the machine learning (ML) model are currently very popular. Therefore, in our experiment, we use the benchmark application that serves the regression model to return house price prediction. It is a web application that handles each request by first predicting the huge range of test data with large numbers and then returning HTML output. Our application is a CPU-bound application.

C. WORKLOAD GENERATION

We retrieve a subset from the Wikipedia Access trace [26] and FIFA World Cup 98 trace [27] as workloads in our

TABLE 3. Parameter setting.

Parameter	Value
Time interval	60 (s)
k last observations	10
threshold	5
influence	0.5
τ_{slo}	350 (ms)
r_{max}	950 (minicore CPU)
r_{min}	600 (minicore CPU)

experiment. We use Locust [31] as our workload generator to generate the HTTP requests with the pattern the same as the prepared subset to the serving application.

D. EXPERIMENTAL PARAMETERS

Table 3 lists the key parameters for our algorithm in the experiment. We choose the length of the time interval to 60 s (1 min) after each scaling execution, which is fine-grained compared to the container environment. If set time interval to a large value, scaling may skip some peak workloads; otherwise, the unnecessary scaling may occur frequently. Since we use the predicted workload as the indicator for the burst detection algorithm and the predicted workload is forecasted based on the time series prediction, the best window size k parameter for our algorithm should be synchronized with the number of past time steps of time series prediction in workload forecasting. Therefore, we choose $k = 10$ to synchronize with the number of past time steps of time series prediction in workload forecasting. Besides, we set *threshold* and *influence* to 5 and 0.5, respectively, due to the burstiness of workloads. We define the value for the user expected response time as 350 milliseconds. For the provisioning algorithm, we use the same method in [10] to achieve the recommendation value for pod resource configuration by using VPA. Applying this mechanism to our benchmark application, we achieve $r_{max} = 950$ minicore CPU, and this value is also used as resource configuration for baseline methods. We also set $r_{min} = 600$ minicore CPU as the minimum requirement.

V. RESULT AND DISCUSSION

Figure 3 and 4 illustrate the outstanding performance of our workload forecasting component with the Bi-LSTM model by the result of the predicted value versus the actual value on the fluctuated patterns. These figures also show the burst signal detected from our proposal. Wikipedia test trace has many periods of bursts, whereas FIFA test trace shows fewer bursts but a long period of bursts in the middle of the pattern.

Figure 5 shows the change in the deployment resource configuration and the performance of the application affected by the reactive HPA, proactive HPA, and our proposal during the experiment with Wikipedia test trace. Because two baseline methods only horizontally scale the deployment, the assigned CPU resource for the pod in the deployment remains unchanged in both burst and non-burst periods, whereas our method can update the new deployment with a new resource per pod in non-burst period. During the burst period,

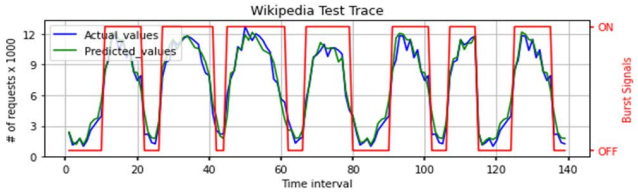


FIGURE 3. Workload prediction and burst identification on Wikipedia test trace.

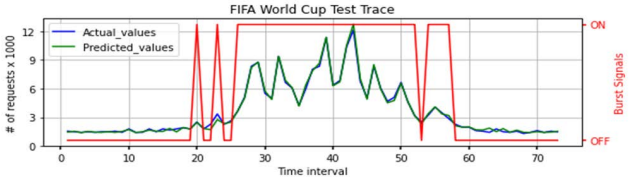


FIGURE 4. Workload prediction and burst identification on FIFA world cup test trace.

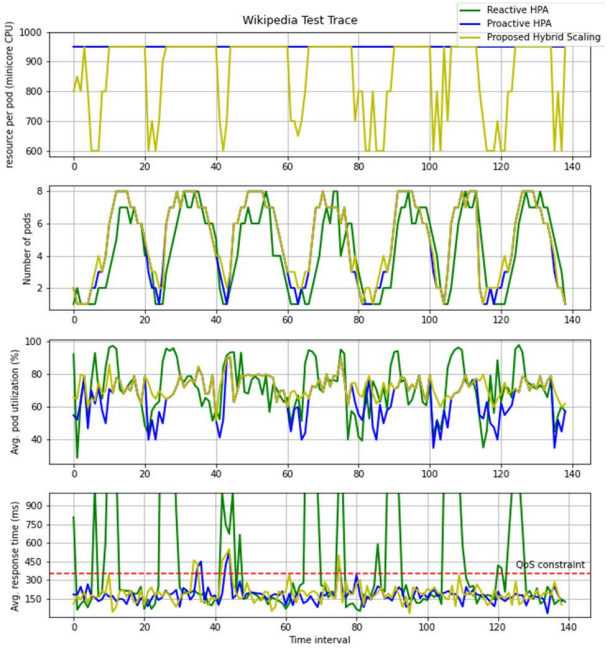


FIGURE 5. Change of resource per pod, number of pods, average pod utilization, and average response time for reactive HPA, proactive HPA, and our proposal during the experiment for Wikipedia test trace.

our method changes and remains the same 950 minicore CPU configuration as baseline methods. Otherwise, during non-burst period, only our method can change the assigned resources flexibly to optimize resource utilization.

The reactive HPA scales up or down the number of pods slower than the two other methods since it is based on the threshold. In contrast, the proactive HPA and our hybrid method achieve better performance that scales faster than reactive HPA because they can forecast the future workload and take the scaling up/down decisions more accurately and ahead of time, especially in burst workloads. Because of slow adaptation, the average response time of application frequently exceeds the QoS constraint for the reactive HPA, especially in burst cases, for example, from time intervals 12th to 19th.

TABLE 4. Percentage of total SLO violation.

Trace	Burst	SLO Violation (%)		
		Reactive HPA	Proactive HPA	Our proposal
Wikipedia	ON	22.14	2.87	2.96
	OFF	8.63	1.14	1.53
FIFA	ON	20.54	4.10	3.89
	OFF	5.47	1.57	1.86

TABLE 5. Percentage of average pod utilization.

Trace	Burst	Average pod utilization (%)		
		Reactive HPA	Proactive HPA	Our proposal
Wikipedia	ON	76.68	73.51	73.73
	OFF	61.78	55.26	69.65
FIFA	ON	78.3	72.68	72.58
	OFF	64.58	63.96	74.17

The proactive HPA and our method quickly adapt to the burst workload, therefore satisfying the QoS better than the reactive method. However, the different point of our proposal in comparison with the proactive HPA is that it can optimize the average utilization of pods. In burst cases, the utilization of both methods can be seen as equal since the workload is heavy in this case and our model also remains the same resource per pod as the proactive HPA. But, in the non-burst period or low-request period, our proposal frequently gets the higher pod utilization due to the ability of hybrid scaling instead of remaining the resource unchanged, for example, from time intervals 80th to 90th. The reactive HPA also performs similar poor pod utilization as the proactive HPA in the low-request period. In burst cases, sometimes the reactive HPA reaches very high pod utilization. Due to the reactive HPA's slow scaling up process, the serving pods are overloaded to serve the user requests and violate the SLO.

Figure 6 shows the performance of our proposal and baseline method for the FIFA test trace. Like the Wikipedia workload, our hybrid scaling method outperforms the other baselines in balancing between maintaining high pod utilization and avoiding SLO violations. The bursts in FIFA test trace do not frequently occur as in Wikipedia test trace. Its burst zone is mainly from the 27th to the 54th time interval. Therefore, the pods are frequently overloaded, and the application frequently violates the SLO with the reaction of reactive HPA. In contrast, the proactive HPA and our proposal remain the better response time under the QoS constraint. In other low-request periods, our method shows outstanding performance in optimizing the pod utilization. Because the low-request period remains longer than the case in the Wikipedia test trace, we can clearly observe the disadvantage of the two baseline methods in terms of optimizing pod utilization.

In both traces, our proposal and proactive HPA are better than the reactive HPA in satisfying the QoS due to the quick adaptation. However, the proactive HPA cannot optimize resource utilization like our hybrid scaling method.

Table 4 and Table 5 summarize the experimental results of the reactive HPA, the proactive HPA, and our proposal

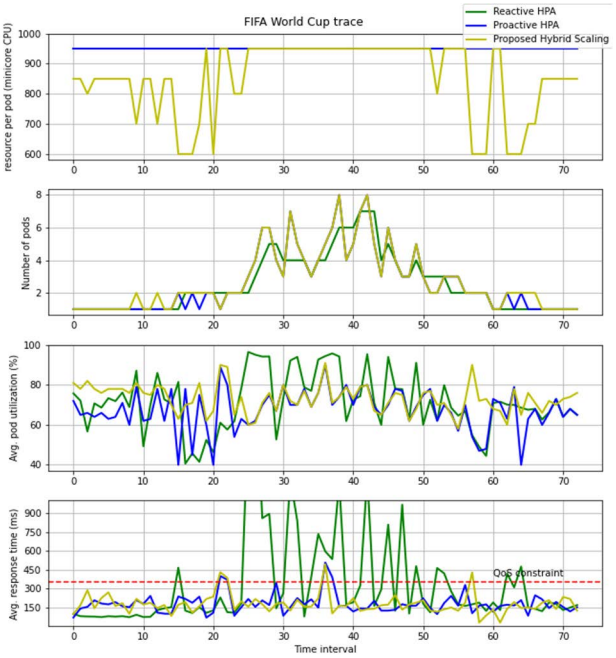


FIGURE 6. Change of resource per pod, number of pods, average pod utilization, and average response time for reactive HPA, proactive HPA, and our proposal during the experiment for under FIFA world cup test trace.

regarding average pod utilization and SLO violation. For each test trace, we separately calculate the percentage of SLO violations and the average pod utilization in burst zone (ON) and non-burst zone (OFF). Overall, the reactive HPA shows the worst performance, with over 20% of SLO violations in all burst cases. It also reaches the highest percentage of SLO violations even in non-burst zones with 8.63% and 5.47% in Wikipedia trace and FIFA trace respectively. In contrast, the proactive HPA and our hybrid method minimize the number of SLO violations in every situation of workload in both two test traces. With our method, we need to scale both the number of pods and the resource configuration for pods; therefore, the execution time is higher, leading to a slightly higher SLO violation percentage than the proactive HPA. There is ongoing work for in-place pod resource adjustment that aims to avoid restarting when updating pod resources in Kubernetes [17]. If this feature in Kubernetes is released, our proposal would also improve the scaling process and the SLO violation.

Regarding average pod utilization, our proposal is the same as the proactive HPA in the burst zones. We ignore the reactive HPA in this case because it mostly deals with SLO violations which lead to pod overloading and is not fair to compare by the average values. In non-burst zone or low-request periods, our proposal outperforms two baseline methods with 69.65% and 74.17% in Wikipedia trace and FIFA trace, respectively.

We also evaluate the effect of this procedure in our method on resource costs during the experiments. We refer to the pricing model [33] when formalizing the resource cost based on the assigned resource for the deployment and its usage time. Due to the operation of our hybrid method, we separate two situations that may occur to calculate the resource

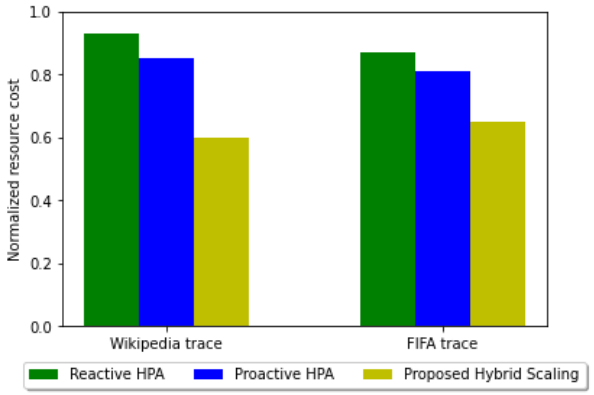


FIGURE 7. Performance evaluation at resource cost.

cost. The first situation includes all m time intervals that the resource for each pod in the running deployment remains unchanged but the number of pods changes. For this situation, the resource cost function is shown in (2).

$$rc = \sum [N_i * R_i] * pr * T_i, i = 1, 2, \ldots, m \tag{2}$$

where rc is the resource cost, which is calculated based on the total the price pr for the resource usage each time interval T_i second of the deployment with the number of pods N_i and the assigned resource per pod R_i.

The second situation includes all n time intervals that our method will replace the old deployment with the new deployment as the result of the hybrid scaler function. In this case, the pod in the old deployment will exist for a while t seconds. Therefore, we can estimate the resource cost in this situation as the following function:

$$rc = rc_{ru} + \sum [N_i * R_i] * pr * (T_i - t_i), i = 1, 2, \ldots, n \tag{3}$$

where rc_{ru} is the resource cost for all the rolling update processes calculated by (4).

$$rc_{ru} = \sum [No_i * Ro_i + N_i * R_i] * pr * t_i, i = 1, 2, \ldots, n \tag{4}$$

No is the number of pods and Ro is the assigned resource per pod of the old deployment that co-exists with the updated deployment in t seconds of each time interval T seconds.

The resource cost for the two baseline methods is calculated by (2) since they only use horizontal scaling processes. The resource cost for our method is estimated by the sum of all situations mentioned above.

Figure 7 compares the normalized resource cost results for the test workloads between the two baseline methods and our proposal. We can observe that our hybrid scaler achieves the lowest resource cost in the two experiments with Wikipedia and FIFA test traces despite the co-existing time of two deployments when executing the hybrid decisions. The reason is that the co-existing time is very short, and the resource for the new deployment is optimized. In contrast, the two baseline methods could not change the assigned resource for pods in the deployment; therefore, the resource cost is higher than our proposal. In general, our hybrid scaler achieves the best balance between avoiding SLO violations and optimizing the resource cost.

TABLE 6. SLO violations and resource cost comparison in burst case.

Trace	SLO Violation (%) / Normalized resource cost	
	Burst-aware HPA	Our proposal
Wikipedia	3.11 / 0.793	2.96 / 0.608
FIFA	4.05 / 0.809	3.89/ 0.612

We also compare the performance of our scaler affected by the burst mechanism to the method in [21], which also uses burst detection. We refer to it as burst-aware HPA. Table 6 shows the average SLO violation and the normalized resource cost of all workloads using different autoscaling methods. The result shows that our proposal achieves a lower percentage of SLO violation in both traces, with 2.96 and 3.89 percent for Wikipedia and FIFA traces, respectively, compared with burst-aware HPA. Specifically, the proposed method reduces over 7.5 percent for both two traces. The reason is that the burst-aware methodology in [21] is based on the number of calculated pods as the main parameter for burst detection, which may lead to higher cost for only horizontal scaling. In our work, the burst detection is based on the predicted workload, which is more accurate since we provide the hybrid scaling mechanism.

VI. FUTURE WORKS

Currently, there is ongoing work for in-place pod resource adjustment that aims to avoid restarting when updating pod resources in Kubernetes [17]. If this feature in Kubernetes is released, rolling update deployment processes in our proposal will become more viable. In the future, we also plan to enhance our proposal by considering the scheduling and placement of pods on the nodes in Kubernetes cluster. The combination of optimized hybrid scaling decisions and efficient scheduling will improve resource usage and ensure service performance. Besides, the scaling performance may be affected by monitoring interval; therefore, we also consider defining the mechanism for choosing the suitable length of monitoring interval as future work to enhance the performance of the scaler.

VII. CONCLUSION

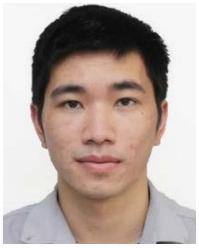
In summary, we present the limitations of existing studies that only focus on a single autoscaling method, only horizontal or only vertical scaling. Therefore, we see the potential of the hybrid autoscaling approach by providing a combination of vertical and horizontal scaling abilities.

Our paper proposes a design and implementation of a hybrid autoscaling framework for containerized applications in Kubernetes to satisfy QoS while optimizing resource utilization. Our proposal achieves this goal by changing both the number of pods and the resources of each pod without service disruption. We also combine the advantage of machine learning techniques with burst identification to make our

proposal predictive and optimize the hybrid scaling schemes. Moreover, the experimental and analytical results showed the effectiveness of our proposed method by outperforming the existing state-of-the-art autoscaling methods with the ability to optimize the average pod utilization whereas avoiding SLO violations. Therefore, we can conclude that it is necessary to provide the hybrid scaling method for containerized applications in Kubernetes.

REFERENCES

[1] A. A. Khaleq and I. Ra, "Agnostic approach for microservices autoscaling in cloud applications," in *Proc. Int. Conf. Comput. Sci. Comput. Intell. (CSCI)*, Dec. 2019, pp. 1411–1415.
[2] C. Lu, K. Ye, G. Xu, C.-Z. Xu, and T. Bai, "Imbalance in the cloud: An analysis on Alibaba cluster trace," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Dec. 2017, pp. 2884–2892.
[3] *Horizontal Pod Autoscsaler—Kubernetes*. Accessed: Apr. 27, 2022. [Online]. Available: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/
[4] *Vertical Pod Autoscsaler—Kubernetes*. Accessed: Apr. 20, 2022. [Online]. Available: https://github.com/kubernetes/autoscaler/tree/master/verticalpod-autoscaler/
[5] C. Qu, R. N. Calheiros, and R. Buyya, "Auto-scaling web applications in clouds: A taxonomy and survey," *ACM Comput. Surv.*, vol. 51, no. 4, pp. 1–33, 2018.
[6] V. R. Messias, J. C. Estrella, R. Ehlers, M. J. Santana, R. C. Santana, and S. Reiff-Marganiec, "Combining time series prediction models using genetic algorithm to autoscaling web applications hosted in the cloud infrastructure," *Neural Comput. Appl.*, vol. 27, no. 8, pp. 2383–2406, Nov. 2016.
[7] F. Rossi, "Auto-scaling policies to adapt the application deployment in Kubernetes," in *Proc. ZEUS*, 2020, pp. 30–38.
[8] L. Toka, G. Dobreff, B. Fodor, and B. Sonkoly, "Machine learning-based scaling management for kubernetes edge clusters," *IEEE Trans. Netw. Service Manag.*, vol. 18, no. 1, pp. 958–972, Mar. 2021.
[9] T.-T. Nguyen, Y.-J. Yeom, T. Kim, D.-H. Park, and S. Kim, "Horizontal pod autoscaling in kubernetes for elastic container orchestration," *Sensors*, vol. 20, no. 16, p. 4621, Aug. 2020.
[10] 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.
[11] D. Balla, C. Simon, and M. Maliosz, "Adaptive scaling of kubernetes pods," in *Proc. IEEE/IFIP Netw. Oper. Manag. Symp.*, Apr. 2020, pp. 1–5.
[12] L. Baresi, D. Hu, G. Quattrocchi, and L. Terracciano, "KOSMOS: Vertical and horizontal resource autoscaling for kubernetes," in *Proc. Int. Conf. Service-Oriented Comput. (ICSOC)*, 2021, pp. 821–829.
[13] M. Masdari and A. Khoshnevis, "A survey and classification of the workload forecasting methods in cloud computing," *Cluster Comput.*, vol. 23, no. 4, pp. 2399–2424, Dec. 2020.
[14] M. A. Razzaq, J. A. Mahar, M. Ahmad, N. Saher, A. Mehmood, and G. S. Choi, "Hybrid auto-scaled service-cloud-based predictive workload modeling and analysis for smart campus system," *IEEE Access*, vol. 9, pp. 42081–42089, 2021.
[15] T. Wang, S. Ferlin, and M. Chiesa, "Predicting CPU usage for proactive autoscaling," in *Proc. 1st Workshop Mach. Learn. Syst.*, Apr. 2021, pp. 31–38.
[16] G. Rattihalli, M. Govindaraju, H. Lu, and D. Tiwari, "Exploring potential for non-disruptive vertical auto scaling and resource estimation in kubernetes," in *Proc. IEEE 12th Int. Conf. Cloud Comput. (CLOUD)*, Jul. 2019, pp. 33–40.
[17] *KEPS: In-Place Update of Pod Resources*. Accessed: Apr. 19, 2022. [Online]. Available: https://github.com/kubernetes/enhancements/tree/master/keps/sig-node/1287-in-place-update-pod-resources
[18] H. Khazaei, R. Ravichandiran, B. Park, H. Bannazadeh, A. Tizghadam, and A. Leon-Garcia, "Elascale: Autoscaling and monitoring as a service," in *Proc. 27th Annu. Int. Conf. Comput. Sci. Softw. Eng. (CASCON)*, 2017, pp. 234–240.
[19] W. Iqbal, A. Erradi, and A. Mahmood, "Dynamic workload patterns prediction for proactive auto-scaling of web applications," *J. Netw. Comput. Appl.*, vol. 124, pp. 94–107, Dec. 2018.
[20] M. Abdullah, W. Iqbal, A. Mahmood, F. Bukhari, and A. Erradi, "Predictive autoscaling of microservices hosted in fog microdata center," *IEEE Syst. J.*, vol. 15, no. 1, pp. 1275–1286, Mar. 2021.
[21] M. Abdullah, W. Iqbal, J. L. Berral, J. Polo, and D. Carrera, "Burst-aware predictive autoscaling for containerized microservices," *IEEE Trans. Services Comput.*, vol. 15, no. 3, pp. 1448–1460, May 2022.
[22] N. Marie-Magdelaine and T. Ahmed, "Proactive autoscaling for cloud-native applications using machine learning," in *Proc. IEEE Global Commun. Conf. (GLOBECOM)*, Dec. 2020, pp. 1–7.
[23] Prometheus. *A Monitoring Framework*. Accessed: Apr. 20, 2022. [Online]. Available: https://prometheus.io
[24] P. Singh, P. Gupta, and K. Jyoti, "TASM: Technocrat ARIMA and SVR model for workload prediction of web applications in cloud," *Cluster Comput.*, vol. 22, no. 2, pp. 619–633, Jun. 2019.
[25] S. Hochreiter and J. J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.
[26] *Wikipedia Pageview Trace*. Accessed: Aug. 5, 2021. [Online]. Available: https://old.datahub.io/dataset/wikistats
[27] (1998). *Worldcup Access Logs*. Accessed: Aug. 5, 2021. [Online]. Available: http://ita.ee.lbl.gov/html/contrib/WorldCup.html
[28] Q. Wang, Y. Kanemasa, M. Kawaba, and C. Pu, "When average is not average: Large response time fluctuations in n-tier systems," in *Proc. 9th Int. Conf. Autonomic Comput. (ICAC)*, 2012, pp. 33–42.
[29] A. Adegboyega, "Quantifying cloud workload burstiness: New measures and models," in *Proc. IFIP/IEEE Symp. Integr. Netw. Service Manag. (IM)*, May 2017, pp. 987–990.
[30] M. Abdullah, W. Iqbal, A. Erradi, and F. Bukhari, "Learning predictive autoscaling policies for cloud-hosted microservices using trace-driven modeling," in *Proc. IEEE Int. Conf. Cloud Comput. Technol. Sci. (CloudCom)*, Dec. 2019, pp. 119–126.
[31] Locust. *A Modern Load Testing Framework*. Accessed: Apr. 20, 2020. [Online]. Available: https://locust.io/
[32] *Labels and Selectors—Kubernetes*. Accessed: Apr. 20, 2022. [Online]. Available: https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/
[33] *AWS Resource Pricing*. Accessed: Apr. 20, 2022. [Online]. Available: https://aws.amazon.com/eks/pricing/



DINH-DAI VU received the B.E. degree from the Hanoi University of Science and Technology, Vietnam, in 2019. He is currently pursuing the master's degree in information and communication convergence with Soongsil University. He works on research projects related to autoscaling, cloud computing, microservices, and machine learning.

MINH-NGOC TRAN received the B.E. degree from the Hanoi University of Science and Technology, in 2018, and the M.S. degree from Soongsil University, in 2020, where he is currently pursuing the Ph.D. degree in information and communication convergence. His research interests include cloud computing, applied machine learning in cloud-edge computing, and 5G networking.

YOUNGHAN KIM (Member, IEEE) received the B.S. degree from Seoul National University and M.Sc. and Ph.D. degrees in electrical engineering from KAIST. He is currently a Full Professor with the Department of Electronic Engineering, Soongsil University. He used to be the President of the Korea Information and Communications Society (KICS). His current research interests include cloud computing, 5G networking, and next-generation networks.