Neurocomputing 521 (2023) 99–112

Contents lists available at ScienceDirect

Neurocomputing

journal homepage: www.elsevier.com/locate/neucom

Stable and efficient resource management using deep neural network on cloud computing

Byeonghui Jeong, Seungyeon Baek, Sihyun Park, Jueun Jeon, Young-Sik Jeong *

Department of Multimedia Engineering, Dongguk University, Seoul 04620, Republic of Korea

ARTICLE INFO

Article history:
Received 21 February 2022
Revised 17 November 2022
Accepted 27 November 2022
Available online 5 December 2022

Keywords:
Cloud computing
Resource management
Hybrid pod autoscaling
Resource usage forecasting
Attention-based Bi-LSTM

ABSTRACT

Resource management autoscaling in a cloud computing service guarantees the high availability and extensibility of applications and services. Horizontal pod autoscaling (HPA) does not affect the executed tasks but also has the disadvantage that it cannot provide immediate scaling. Furthermore, scale down is not possible if excess resources are allocated, because it is difficult to identify the amount of resources required for applications and services; thus resources are wasted. Therefore, this study proposes Proactive Hybrid Pod Autoscaling (ProHPA), which immediately responds to irregular workloads and reduces resource overallocation. ProHPA uses a bidirectional long short-term memory (Bi-LSTM) model applied with an attention mechanism for forecasting future CPU and memory usage that has similar or different patterns. Reducing excessive resource usage with vertical pod autoscaling (ReVPA) adjusts the overallocation of resources within a pod by forecasted resource usage. Lastly, prevention overload with HPA (PoHPA) immediately performs resource scaling by using forecasted resource usage and pod information. When the performance of ProHPA was evaluated, CPU and memory average utilization were improved by 23.39% and 42.52%, respectively, compared with conventional HPA when initial resources were overallocated. In addition, ProHPA did not exhibit overload compared to conventional HPA when resources are insufficiently allocated.

1. Introduction

Many IT companies are adopting a cloud-native computing paradigm that pursues a container-based microservice architecture; therefore, the container has emerged as the standardized virtualization technology in cloud computing [1,2]. A container has less overhead than a hypervisor-based virtualization technology, and guarantees efficient portability and extensibility based on its light weight [3–6]. However, interdependence between containers becomes complicated as the number of containers soars.

Container orchestration platforms such as Kubernetes, Docker Swarm, and Apache Mesos provide various automated container management functions for managing large-scale containers, including scheduling, scaling, and failover. Specifically, Kubernetes offers scaling for the size and number of pods that include at least one container for utilizing resources elastically according to whether vertical pod autoscaling (VPA) and horizontal pod autoscaling (HPA) are available [7]. VPA adjusts the size of a pod by computing the recommended request based on past resource usage. However, VPA cannot guarantee the continuity of services because existing pods are removed and then regenerated. On the contrary, HPA adjusts the numbers of pods by duplicating and removing pods according to resource usage, which varies. However, HPA based on a reactive control mechanism is incapable of an immediate response due to a delay in applying the scaled resources [8]. Therefore, an overload occurs when resource utilization becomes too high when the workload suddenly rises. In addition, when a developer designates an excessive amount of initial resources, HPA cannot adjust the size of the resources, thus leading to waste [9,10].

This study, therefore, proposes Proactive Hybrid Pod Autoscaling (ProHPA), which stably and efficiently autoscales pods by forecasting future CPU and memory usage to overcome the drawbacks of the HPA provided by Kubernetes. ProHPA extracts workload data through monitoring and trains an attention-based bidirectional long short-term memory (Bi-LSTM) model to analyze irregular patterns. ProHPA also sequentially performs reducing excessive resource usage with VPA (ReVPA) and prevention overload with HPA (PoHPA) in order to adjust the size and number of pods based on the forecasted future resource usage. First, ReVPA adjusts the

* Corresponding author.
E-mail addresses: qudgml6323@dgu.ac.kr (B. Jeong), tmddusdls12@dongguk.edu (S. Baek), psh0430@dongguk.edu (S. Park), jry02107@dongguk.edu (J. Jeon), ysjeong@dongguk.edu (Y.-S. Jeong).

https://doi.org/10.1016/j.neucom.2022.11.089
0925-2312/© 2022 Elsevier B.V. All rights reserved.

amount of initial resources that have been overallocated in pods. PoHPA calculates the number of future replicas based on the regenerated pods and forecasted resource usage for scaling CPU, memory.

ProHPA adjusts the amount of initial resources that have been insufficiently allocated to resolve the issue of resources wasted by the HPA provided by Kubernetes. Furthermore, when resources are overallocated in the initial pods, ProHPA immediately allocates resources to stably manage the resources by preventing temporary service interruption due to an overload.

The main contributions in this study are summarized as follows:

- ProHPA, a new deep learning-based hybrid autoscaling technique is proposed.
- ProHPA accurately predicts workloads composed of various patterns by considering the main patterns of each time point related to future workloads through attention-based Bi-LSTM.
- ProHPA uses the predicted workload to improve the resource waste and overload problem of the reactive mechanism-based autoscaling technique caused by dynamic load.
- ProHPA has high resource utilization and low overload even in various workload composed of different patterns.

The remainder of this paper is organized as follows: related works on resource autoscaling are described in Section 2; the ProHPA scheme proposed in this study is explained in Section 3, and the overall architecture of ProHPA is presented in Section 4; ProHPA is implemented in Section 5, and the performance of ProHPA is evaluated in Section 6; lastly, the research findings on ProHPA are summarized and future research directions are proposed in Section 7.

2. Related works

Various studies have been conducted to efficiently manage resources in cloud computing [11–19]. In this study, several universal methods for managing container resources using horizontal, vertical and hybrid autoscaling techniques are described.

2.1. Horizontal autoscaling

Horizontal autoscaling provides service scalability without affecting running services. Studies using horizontal autoscaling are as follows:

Toka et al. [8] proposed HPA+, which reduces the number of rejected requests by performing autoscaling based on forecasted resource usage. HPA+ forecasts resources using various machine learning (ML) techniques such as autoregressive (AR), hierarchical temporal memory (HTM), LSTM, and reinforcement learning (RL). Additionally, an excess parameter for resource utilization balance has been proposed to provide efficient resource management.

Dang-Quang et al. [20] proposed an autoscaling technique based on a deep learning model to improve the problems of conventional HPA, which is generally carried out using a reactive method. Autoscaling using a Bi-LSTM model provides scaling faster than conventional HPA and efficiently processes a burst workload.

Yan et al. [21] proposed HANSEL, which provides autoscaling by integrating reactive and proactive methods for elastically managing resources according to the changes in a workload. The proactive HANSEL method guarantees service level agreement (SLA) of a microservice in edge computing and improves resource utilization using an attention-based Bi-LSTM model and RL to accurately forecast a microservice workload.

However, when the amount of initially allocated resources is inefficient, the resources are wasted or overloaded.

2.2. Vertical autoscaling

Vertical autoscaling utilizes resources elastically by adjusting the amount of allocated resources based on past resource usage without the expertise in applications and services. Studies using this vertical autoscaling are as follows:

Chouliaras et al. [14] proposed a performance-aware autoscaler for cloud elasticity (PACE) to efficiently manage containerized cloud applications and ensure quality of service (QoS) requirements. PACE prevents application failure by performing reactive autoscaling using threshold-based rules. In addition, after predicting the workload through convolutional neural network (CNN) and K-means, PACE managed resources elastically by utilizing proactive autoscaling.

Wang et al. [22] proposed proactive vertical autoscaling for solving various problems, such as wasted resources and service interruptions that are caused by inaccurate resource recommendations. Proactive vertical autoscaling forecasts a future workload using an LSTM model to more efficiently manage resources than conventional VPA.

Buchaca et al. [23] proposed the AI4DL framework to reduce out-of-memory error and over-provisioning caused by irregular workloads. AI4DL encodes multi-dimensional time-series data into feature vectors using conditional restricted Boltzmann machines (CRBM) and a clustering technique. The generated feature vectors are trained in a multi-layer perceptron (MLP) and LSTM model to forecast the future workload. AI4DL reduced the over-provisioning of a CPU by up to 38% and prevented out-of-memory error by up to 96%.

However, vertical autoscaling performs inefficient scaling when a pattern different from the past resource usage pattern occurs. In addition, the service is temporarily suspended to update the amount of the resource.

2.3. Hybrid autoscaling

Hybrid autoscaling is an effective autoscaling technique that combines horizontal and vertical autoscaling to improve the problems of conventional autoscaling. This hybrid autoscaling technique efficiently manages resources based on the advantages of horizontal and vertical autoscaling.

Vu et al. [17] proposed hybrid autoscaling that optimizes resource utilization and satisfies QoS to solve the problem of single autoscaling. The proposed hybrid autoscaling optimized hybrid scaling execution through predictive scaling method and burst identification after predicting workload through Bi-LSTM. The proposed method managed resources more efficiently than HPA based on reactive and proactive mechanisms.

Rossi et al. [24] proposed Elastic Docker Swarm (EDS), which uses threshold-based heuristics to improve the elasticity of container-based applications. EDS defines a threshold policy through various RL techniques, and elastically handles different workloads by performing horizontal and vertical autoscaling.

Rzadaca et al. [25] proposed autopilot to resolve performance degradation and wasted resources caused by inefficiently provisioned resources. Autopilot selects vertical and horizontal autoscaling according to the resource limit calculated by an ML-based recommender. Therefore, autopilot reduced the difference between resource usage and the limit by 23% and reduced the rate of out-of-memory error by 10-fold.

The ProHPA proposed in this study, which combines horizontal and vertical autoscaling, improves effectively the problems of reactive mechanism-based HPA. ProHPA consists of ReVPA, which adjusts overallocated resources through forecasted future workload, and PoHPA, which responds immediately to dynamic load. To accurately predict the workload composed of various patterns, ProHPA utilizes the attention-based Bi-LSTM model. ProHPA manages CPU and memory of the container reliably and efficiently.

Table 1 presents a comparative analysis of the proposed ProHPA for efficiently managing resources in a container and previous autoscaling studies. The comparison items include autoscaling methods used to improve the performance of autoscaling provided by Kubernetes, types of autoscaling used for efficiently managing resources, autoscaling resource objects that indicate the scaling subject, and time-series analysis techniques used for analyzing workloads.

3. ProHPA scheme

This study proposes ProHPA, which computes forecasted resource usage from the attention-based Bi-LSTM model and then sequentially applies to ReVPA and PoHPA to stably and efficiently manage resources. The overall scheme of ProHPA is as shown in Fig. 1.

The container orchestration platform mainly manages CPU and memory for creating and running applications and services. Therefore, the autoscaling resource object of ProHPA is designated as CPU and memory. First, the CPU and memory usage information of a pod are extracted and then trained in the attention-based Bi-LSTM model to forecast future CPU and memory usage. ProHPA sequentially performs ReVPA and PoHPA in order to manage inefficiently allocated CPUs and memory more stably and efficiently than does the HPA provided by Kubernetes.

3.1. Resource usage information extraction

Kubernetes monitors various metrics, such as disk I/O, HTTPS requests, CPU usage, and the memory usage of executing pods through cAdvisor and Prometheus in order to provide information on the status of resources [26]. Prometheus records the collected metrics in a chronological order using a time-series database (TSDB). Therefore, ProHPA extracts the CPU and memory usage metrics of pods recorded in TSDB by using *container_cpu_usage_seconds_total* and *container_memory_working_set_bytes* of Prometheus query for forecasting future CPU and memory usage. The extracted CPU and memory usage are configured as time-series data in JavaScript Object Notation (JSON) format.

3.2. Forecasted resource usage with attention-based Bi-LSTM model

The forecasted resource usage with attention-based Bi-LSTM model performs time-series data pre-processing and resource usage forecasting steps for analyzing and learning past CPU and memory usage patterns.

First, in the time-series data pre-processing step, the time-series data extracted in the resource usage information extraction step are converted to vectors. The CPU and resource usage data constituted with different vectors are combined into one vector as multivariate data in order to forecast future CPU and memory usage using the single time-series forecasting model.

In the resource usage forecasting step, resource usage is learned using a recurrent neural network (RNN)-based deep neural network to quickly and accurately forecast multivariate time-series data. Among the RNN-based deep neural networks, Bi-LSTM quickly analyzes sequences by bidirectionally extracting context information, and it is used [27,28]. In addition, the last hidden state of Bi-LSTM loses various information due to the gradient vanishing problem. Therefore, we add an attention mechanism that refers to all hidden states of Bi-LSTM [21].

The attention-based Bi-LSTM operating in ProHPA has an architecture as shown in Fig. 2, where vectorized CPU usage C_n and memory usage M_n are used as inputs to a deep neural network model. The Bi-LSTM layer analyzes the features of workload input in both directions, whereas the attention layer computes the importance of each element [29,30]. Here, w_t^f and w_t^b are the weights assigned to the cell in LSTM, and f and b indicate forward and backward directions, respectively. The exposure bias occurs because the teacher forcing technique used in the attention-based Bi-LSTM model, thus bias is not used to improve the stability and forecasting accuracy of the model [31]. The LSTM cell uses the cell state value from a previous time, c_{t-1}^f and c_{t-1}^b, as well as the output values of a previous time, h_{t-1}^f and h_{t-1}^b. To accurately analyze the patterns of CPU and memory usage, as shown in Eq. (1), the attention score is calculated by performing matrix multiplication on the hidden state V_n of each time point of the Bi-LSTM and the V_n^T transposed hidden state. Furthermore, the attention distribution α is calculated by applying the softmax function to the attention score. The attention value a which reflects the importance in the hidden state of Bi-LSTM is calculated by performing matrix multiplication on α and V_n as shown in Eq. (2). Lastly, we construct context vector which given importance by concatenating a.

$$\alpha = softmax\left(\left[V_1 \circ V_1^T, \cdots, V_n \circ V_n^T\right]\right) \tag{1}$$

$$a = \sum_{i=1}^{n} \alpha_i V_i \tag{2}$$

3.3. Reducing excessive resource usage with vertical pod autoscaling (ReVPA)

The amount of resources initially set by a developer for generating a pod may not be appropriate for future workloads [9,10].

Table 1
Comparison of ProHPA with previous research.

Related works	Autoscaling type	Autoscaling method	Autoscaling resource object	Time-series analysis technique
Toka et al. [8]	Horizontal	Proactive	CPU	AR, HTM, LSTM, RL
Dang-Quang et al. [20]	Horizontal	Proactive	CPU	Bi-LSTM
Yan et al. [21]	Horizontal	Reactive/Proactive	CPU, memory	Attention-based Bi-LSTM
Chouliaras et al. [14]	Vertical	Reactive/Proactive	CPU, memory	CNN, K-means
Wang et al. [22]	Vertical	Proactive	CPU	LSTM
Buchaca et al. [23]	Vertical	Proactive	CPU, memory	MLP, LSTM
Vu et al. [17]	Hybrid	Proactive	CPU	Bi-LSTM
Rossi et al. [24]	Hybrid	Proactive	CPU	RL
Rzadca et al. [25]	Hybrid	Reactive	CPU, memory	Argmin algorithm
ProHPA	**Hybrid**	**Proactive**	**CPU, memory**	**Attention-based Bi-LSTM**

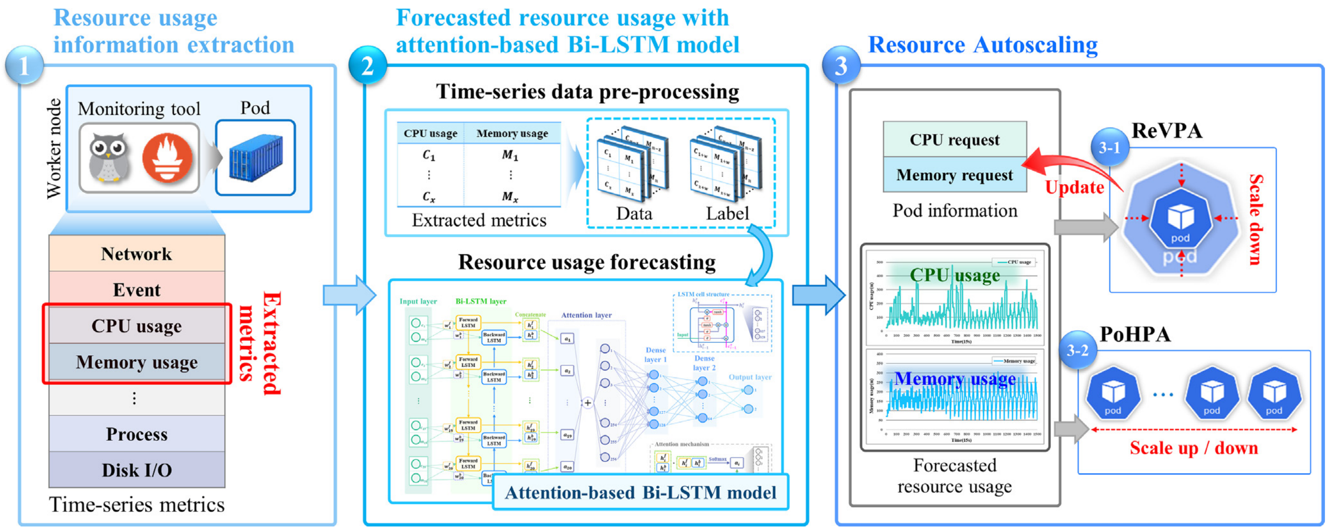


Fig. 1. Scheme of Proactive Hybrid Pod Autoscaling using attention-based Bi-LSTM (ProHPA).

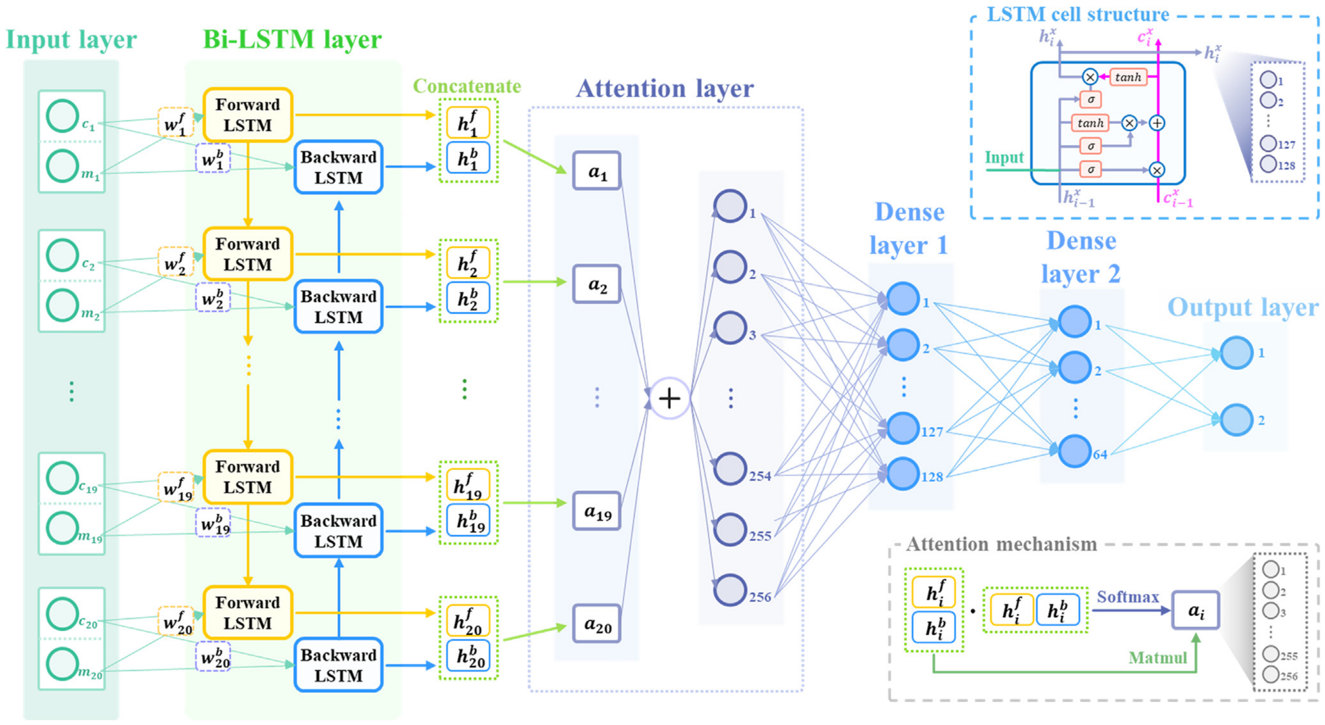


Fig. 2. Architecture of attention-based bidirectional long short-term memory (Bi-LSTM).

Resources are wasted if the amount of resources allocated to a pod exceeds the actual amount needed for future work.

Kubernetes provides VPA to adjust the size of resources in a pod because it is difficult to improve the resource wasting issue using conventional HPA. VPA applies a decaying histogram to past resource usage to adjust the resource size of a pod based on a percentile. If the currently owned resources are less than 50%, VPA increases the size, as it judges that the resources are insufficiently allocated. In contrast, if the currently owned resources exceed 95%, VPA decreases the size as it judges that the resources are overallocated. The resource size is updated to a 90% value [22].

ReVPA applies the lower bound, upper bound, and target limit of conventional VPA based on past resource usage to future resource usage to reduce the overallocation of resources. Algorithm 1 reduces the overallocation of resources using the FR_{CPU}, FR_{memory} CPU and memory usage forecasted through the attention-based Bi-LSTM model. First, the 95th percentile $Percentile(FR_{CPU}, 95)$, $Percentile(FR_{memory}, 95)$ of FR_{CPU}, FR_{memory} and the amount of resources currently allocated to a pod are compared to examine whether resources are overallocated. If either CPU or memory is overallocated, the generated pod is removed and then regenerated with the 90th percentile of forecasted resource usage to stably manage the resources. In contrast, if the amount of initially allocated resources is between the 50th and 95th percentile or less than the 50th percentile, the case is not taken into consideration because PoHPA can be used for resource management.

Algorithm 1. *Reducing excess resources using with vertical pod autoscaling*

```
INPUT: $Pod^{request}_{CPU}$, $Pod^{request}_{memory}$, $FR_{CPU}$, $FR_{memory}$
OUTPUT: Pod scaling command
1:  while (Pod is running) & (ProHPA is running) do
2:    $Update \leftarrow False$
3:    if $Pod^{request}_{CPU} > Percentile(FR_{CPU}, 95)$ then
4:      $Pod^{request}_{CPU} \leftarrow Percentile(FR_{CPU}, 90)$
5:      $Update \leftarrow True$
6:    endif
7:    if $Pod^{request}_{memory} > Percentile(FR_{memory}, 95)$ then
8:      $Pod^{request}_{memory} \leftarrow Percentile(FR_{memory}, 90)$
9:      $Update \leftarrow True$
10:   endif
11:   if $Update == True$ then
12:     $SendComm\left(Scale\_down, Pod^{request}_{CPU}, Pod^{request}_{memory}\right)$
13:   else
14:     $SendComm(None)$
15:   endif
16: endwhile
```

ReVPA manages resources more efficiently than the HPA of Kubernetes by reducing the overallocation of resources within a pod with the resource amount that guarantees the continuity of future work.

3.4. Prevention overload with horizontal pod autoscaling (PoHPA)

The HPA conventionally provided by Kubernetes uses a reactive mechanism, thus it is incapable of immediate resource scaling. Therefore, a resource overload occurs when insufficiently allocated resources are scaled. Service quality is degraded due to throttling when a CPU is overloaded, and service interruption may occur due to an out-of-memory error if the memory is overloaded.

To improve the overload issue, PoHPA uses the forecasted resource usage calculated by the attention-based Bi-LSTM model and the resource capacity of a pod. The number of future replicas is calculated as shown in Eq. (3), where the ratio of the resource utilization threshold to forecasted resource utilization is considered. *Current_replicas* represents the number of current replicas, and *Target_value* represents the resource utilization threshold. *Forecasted_resource_utilization* is the ratio of forecasted resource usage to the resource capacity of a pod, where a ceiling is used to ensure the continuity and stability of future work.

$$Future\_replicas = \left\lceil \frac{Current\_replicas \times Forecasted\_resource\_utilization}{Target\_value} \right\rceil \tag{3}$$

The best performance is demonstrated when resource utilization is 80% or above, whereas the performance is degraded due to overload when resource utilization is 90% or above [32–34]. Therefore, the resource utilization threshold for the PoHPA proposed in this study is set to 80%, considering the forecasting error of the attention-based Bi-LSTM model. A Kubernetes pod typically executes a single instance of a specific application, where one pod generally consists of one container; hence, ProHPA only considers a singleton pod [35,36].

Algorithm 2 calculates the number of replicas needed for a future point in time using the forecasted CPU and usage per time of memory FR^t_{CPU}, FR^t_{memory}, thus demonstrating the process of immediate scaling for the varying workloads. First, the initial number of replicas is set to 1, and the CPU and memory utilization of a pod are calculated based on the request per resource. The number of replicas is calculated for CPU and memory utilization separately, and the greater value is used as the number of replicas for the respective point in time. The number of replicas is updated when the number of replicas at a future time differs from the number at the present time. For example, if the replica value at $t+1$ is smaller than the replica value at t, scale down is performed at $t+1$, or if it is greater than the replica value at t, scale up is performed at $t+1$.

Algorithm 2. *Prevention overload with horizontal pod autoscaling*

```
INPUT: $Pod^{request}_{CPU}$, $Pod^{request}_{memory}$, $FR^t_{CPU}$, $FR^t_{memory}$, $Target_{CPU}$, $Target_{memory}$
OUTPUT: Pod scaling command
1:  $Replica^t \leftarrow 1$
2:  while (Pod is running) & (ProHPA is running) do
3:      $ReUtil^{t+1}_{CPU} \leftarrow FR^{t+1}_{CPU} / \left(Pod^{request}_{CPU} \times Replica^t\right) \times 100$
4:      $Replica^{t+1}_{CPU} \leftarrow \lceil \left(Replica^t \times ReUtil^{t+1}_{CPU}\right) / Target_{CPU} \rceil$
5:      $ReUtil^{t+1}_{memory} \leftarrow FR^{t+1}_{memory} / \left(Pod^{request}_{memory} \times Replica^t\right) \times 100$
6:      $Replica^{t+1}_{memory} \leftarrow \lceil \left(Replica^t \times ReUtil^{t+1}_{memory}\right) / Target_{memory} \rceil$
7:      $Replica^{t+1} \leftarrow Max\left(Replica^{t+1}_{CPU}, Replica^{t+1}_{memory}\right)$
8:      if $Replica^t > Replica^{t+1}$ then
9:      $SendComm\left(Scale\_down, Replica^{t+1}\right)$
10:     else if $Replica^t < Replica^{t+1}$ then
11:     $SendComm\left(Scale\_up, Replica^{t+1}\right)$
12:     else
13:     $SendComm(None)$
14:     endif
15: endwhile
```

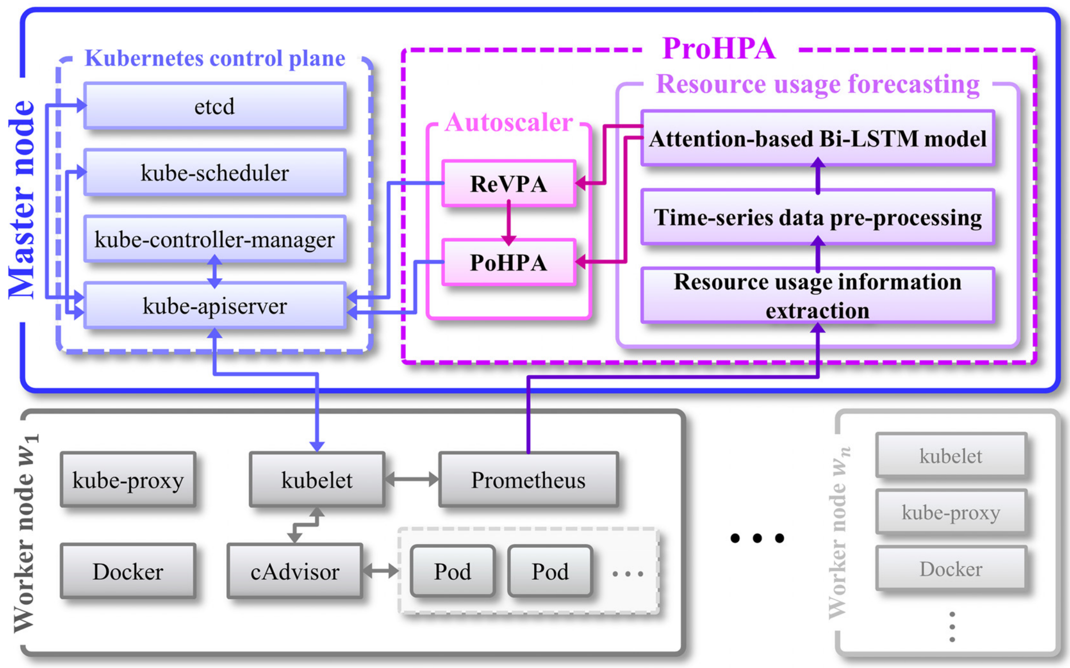


Fig. 3. Overall architecture of Kubernetes cluster and ProHPA.

4. Design of ProHPA

In this study, ProHPA, which uses a deep neural network model, is added to the Kubernetes cluster to manage the resources of a generated pod stably and efficiently. Fig. 3 shows the overall architecture of the Kubernetes cluster to which ProHPA is added.

Generally, a Kubernetes cluster consists of one master node and several worker nodes. A master node controls clusters, as it consists of control planes such as etcd, kube-scheduler, kube-controller-manager, and kube-apiserver [7]. etcd saves all configuration data to guarantee the consistency and high availability of clusters, while kube-scheduler allocates a newly generated pod or an unassigned pod to a node by considering specific scheduling requirements. kube-controller-manager converts the current cluster state to a desired state through the node controller, replication controller, or endpoint controller. kube-apiserver communicates with other configuration components using the Kubernetes cluster API.

A worker node consists of the kube-proxy, kubelet, container runtime, and container monitoring tools, arranges containers within a pod, and executes according to the commands of a master node. kube-proxy manages the operation of a virtual network based on the network rules of a node. kubelet is an agent that is executed in all nodes within a cluster, and manages containers according to the command of a kube-apiserver of a master node. Container runtime executes containers, and Docker is used in this study [7]. A container monitoring tool monitors the resource usage data of containers, and cAdvisor is a well-known monitoring agent. In this study, cAdvisor and Prometheus, which provide time-series data, are used for extracting resource usage by time.

To efficiently and stably manage resources in the Kubernetes cluster, we configured ProHPA in the master node and connected it with the container monitoring tool of the worker node. ProHPA consists of ReVPA and PoHPA that operate based on forecasted future resource usage.

First, ProHPA extracts the CPU and memory usage metrics among the resource usage metrics collected from cAdvisor and Prometheus in the worker node by performing resource usage information extraction step. The extracted resource usage metrics are converted into vectors through the time-series data preprocessing step, and then used as input data for the attention-based Bi-LSTM model. The attention-based Bi-LSTM model forecasts future resource usage with high accuracy by learning the patterns of collected resource usage.

ReVPA regenerates pods with the forecasted resource usage for stably adjusting the overallocated resources of pods to future workloads as well. PoHPA performs immediate resource scaling by pre-calculating the number of replicas required in the future through the forecasted resource usage and pod information generated in ReVPA.

5. ProHPA implementation

To verify the effectiveness of the proposed ProHPA, which stably and efficiently manages resources in Kubernetes, a Kubernetes cluster consisting of one master node and four worker nodes was constructed. Each node was generated using a virtual machine through Oracle VM VirtualBox with the specifications presented in Table 2.

To verify whether the proposed ProHPA stably and efficiently manages resources in Kubernetes, we used the Google cluster

Table 2
Kubernetes cluster configuration.

Name	Master node	Worker node
CPU	4 core	2 core
Memory	12 G	8 G
Storage	100 GB	
OS	Linux 20.04.3 LTS	
Kubernetes	v1.23.1	
Docker	v20.10.12	
Prometheus	v2.32.1	

workload traces 2019 dataset, which is a workload trace of May 2019 generated by Borg cell, the ancestor of Kubernetes. A pod of Kubernetes has features similar to that of the alloc of a Borg cell; thus, the workload data of a total of 35,243 allocs were used [37]. Here, a total of 467 allocs were used to extract workloads, excluding data with interrupted records or records of less than 30 days, to carry out training for the same period. In addition, the workload of an alloc has recorded in five-minute periods; thus, each workload consists of 8,972 data units. Considering that the basic control period and scraping period of HPA is 15 s, each workload was reconfigured with 178,540 data units.

5.1. Forecasted resource usage with attention-based Bi-LSTM model

To train the forecasted resource usage with the attention-based Bi-LSTM model, the training dataset used 142,832 data units, which represented 80% of the full dataset; the validation dataset used 17,854 data units to verify the forecasting accuracy of the constructed model, while 17,854 data units were used as the test dataset to test the model.

In the time-series data pre-processing step, the CPU and memory usage vectors were converted to one vector with respect to the timestamp, and labeling was performed using a sliding window technique with the window size of 1. Furthermore, the min–max normalization technique was applied to minimize the errors during data learning and improve the forecasting accuracy.

In the resource usage forecasting step, the attention-based Bi-LSTM model used CPU and memory usage as an input, and the sequence length was set to 20 to identify the patterns in five-minute periods. The hidden size of the LSTM cell was set to 128, and the mean squared error (MSE), which is a widely used regression analysis method, was applied as the loss function to extract the sequence features. Furthermore, Adam, which combines the RMSProp and momentum methods, was used as an optimizer for using the weight of the forecasting model; a deep neural network model applied early stopping with respect to the validation loss value to save the state before overfitting occurred. The average R^2 scores for the validation and testing of the attention-based Bi-LSTM model were 0.9573 and 0.9771, respectively.

5.2. Reducing excess resource usage with vertical pod autoscaling

ReVPA regenerated a pod with a size that could accommodate 90% of the future workload in order to stably and efficiently manage resources that were overallocated in a pod. To examine whether resources were overallocated in a pod, ReVPA first arranged the forecasted resource usage calculated from the attention-based Bi-LSTM model in an ascending order, and used the 50th, 90th, and 95th percentile values as thresholds. When the resource amount was greater than the 95th percentile value, it was deemed as overallocated and a pod having the 90th percentile value was regenerated. Stability was thus secured by maintaining service availability in the future at 90%. In contrast, it was deemed insufficiently allocated if the resource amount was less than the 50th percentile value, but scale up was feasible through PoHPA.

When the pod with overallocated resources is reconfigured through ReVPA and PoHPA is applied, the average CPU and memory utilization were improved by up to 23.39% and 42.52%, respectively, compared with the conventional HPA provided by Kubernetes.

5.3. Prevention overload with horizontal pod autoscaling

To immediately perform resource scaling, PoHPA calculates the number of replicas needed in the future from the resource allocation of the current pod and the forecasted resource usage. By applying the replica customized for the future workload, PoHPA stably and efficiently manages resources by improving the resource waste problem caused by the cooldown technique and the resource scaling delay problem that occurs in the HPA of Kubernetes. In particular, PoHPA provides immediate scaling for a workload that changes suddenly, thus guaranteeing stability by preventing an overload in a pod that has insufficiently allocated resources.

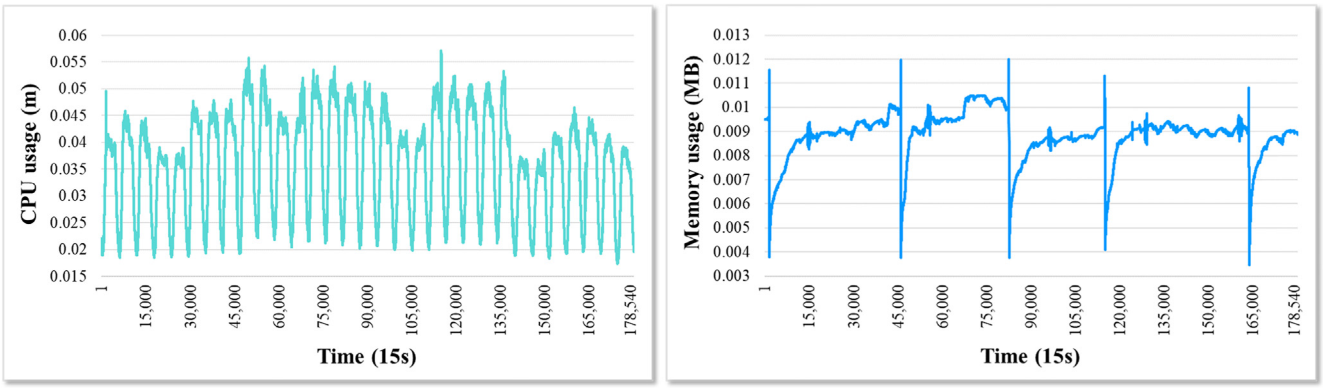
When the number of overloads in a pod with insufficient resources allocated from PoHPA was measured, CPU and memory overload occurred two or zero times, respectively, which is fewer than with the conventional HPA, which had up to eight and four overloads in CPU and memory, respectively.

6. Performance evaluation

To evaluate the forecasting accuracy of the attention-based Bi-LSTM model within the proposed ProHPA, the MSE, root mean squared error (RMSE), mean absolute error (MAE), and R^2 were employed as evaluation indices. Here, Y_i is an actual value, $\widehat{Y}_i$ is a forecasted value, and $\bar{Y}_i$ is the average value of Y_i.

MSE is measured as the mean square of errors between actual values and forecasted values, as shown in Eq. (4) and is sensitive to outliers because the values are squared.

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \widehat{Y}_i\right)^2 \tag{4}$$



(a) CPU usage (b) Memory usage

Fig. 4. Alloc-19519340051 workload composition.

RMSE, which is the root of MSE, is measured by applying a penalty to a large error value, as shown in Eq. (5).

$$\mathrm{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \widehat{Y}_i\right)^2} \quad (5)$$

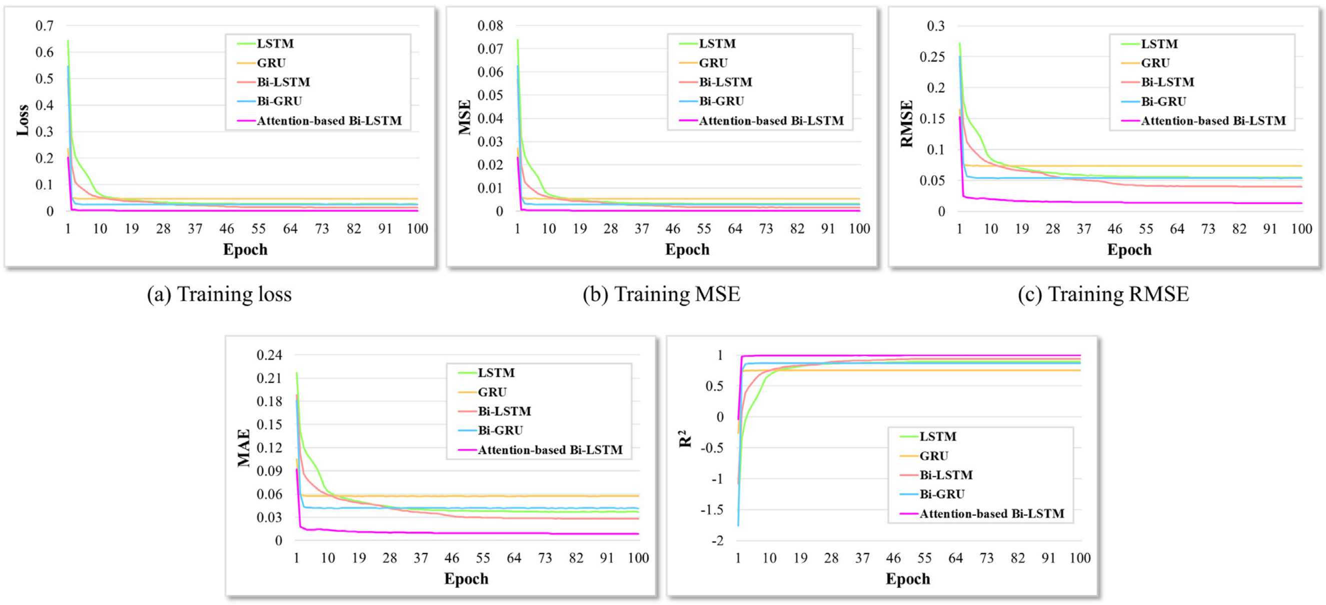
MAE, which is defined as the mean absolute value of errors, is less sensitive to outliers than is MSE, and it is calculated as shown in Eq. (6).

$$\mathrm{MAE} = \frac{1}{n}\sum_{i=1}^{n}\left(\left|Y_i - \widehat{Y}_i\right|\right) \quad (6)$$

The R^2 is an index for measuring forecasting accuracy by calculating the variance ratio of Y_i and $\widehat{Y}_i$. Unlike other indices, a value closer to 1 indicates that a model has good performance. The R^2 is calculated as shown in Eq. (7).

$$R^2 = 1 - \frac{\sum_{i=1}^{n}\left(Y_i - \widehat{Y}_i\right)^2}{\sum_{i=1}^{n}\left(Y_i - \bar{Y}_i\right)^2} \quad (7)$$

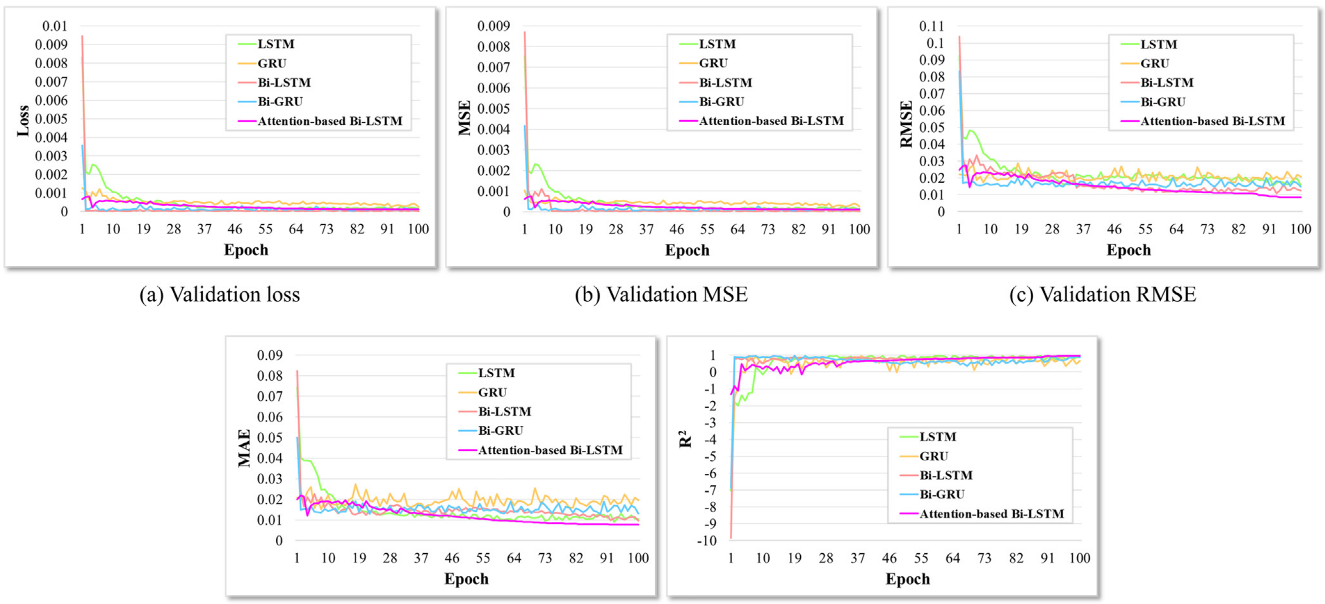
A comparison between the LSTM model, GRU model, Bi-LSTM model, and Bi-GRU model was performed to verify the forecasting accuracy for future workloads, which is calculated from the



(a) Training loss (b) Training MSE (c) Training RMSE

(d) Training MAE (e) Training R^2

Fig. 5. Training performance of LSTM, GRU, Bi-LSTM, Bi-GRU, and attention-based Bi-LSTM models.



(a) Validation loss (b) Validation MSE (c) Validation RMSE

(d) Validation MAE (e) Validation R^2

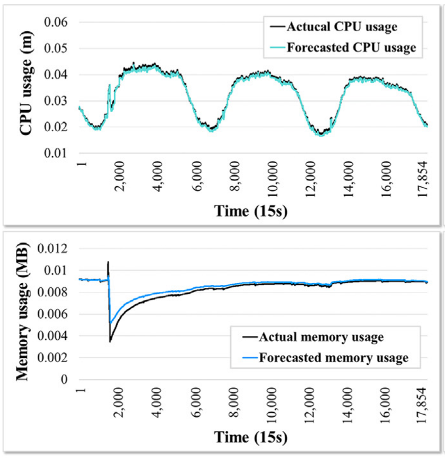
Fig. 6. Validation performance of LSTM, GRU, Bi-LSTM, Bi-GRU, and attention-based Bi-LSTM models.

attention-based Bi-LSTM model within ProHPA. The ARIMA model, which is a conventional time-series analysis technique, was not considered because it cannot perform immediate autoscaling due to its slow forecasting speed [20,38]. To compare the performance between deep neural network models, we used the workload of alloc-19519340051 in the Google cluster workload traces 2019. As shown in Fig. 4(a) and 4(b), the workload data consists of CPU and memory usage in 15-second period. Here, the CPU and memory usage records have a similar pattern for specific period.

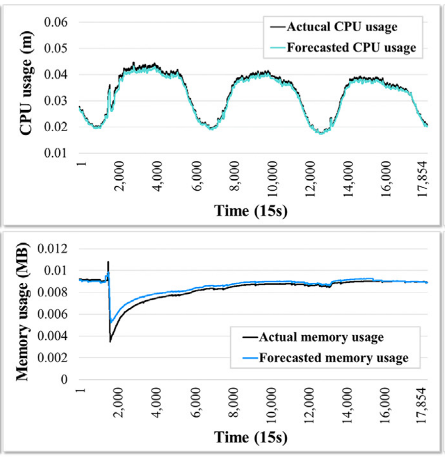
Figs. 5 and 6 illustrate the results of performing training and validation for the LSTM model, GRU model, Bi-LSTM model, Bi-GRU model, and attention-based Bi-LSTM model for 100 epochs. Fig. 5(a) and 6(a) show that the models are trained adequately, as the training and validation loss of deep neural network models decreases as the number of epochs increases. Fig. 5(b), 5(c), 5(d), 6(b), 6(c), and 6(d) show the MSE, RMSE, and MAE of each model according to the changes in epochs, where the attention-based Bi-LSTM model has the lowest MSE, RMSE, and MAE among the deep neural network models. In addition, Fig. 5(e) and 6(e) show that the attention-based Bi-LSTM model had an R^2 score closer to 1 than do the other models.

Fig. 7 and Table 3 present the test results of three trained deep neural network models. Fig. 7 shows that the workload forecasted by the attention-based Bi-LSTM model is more similar to the actual workload compared with other four models.

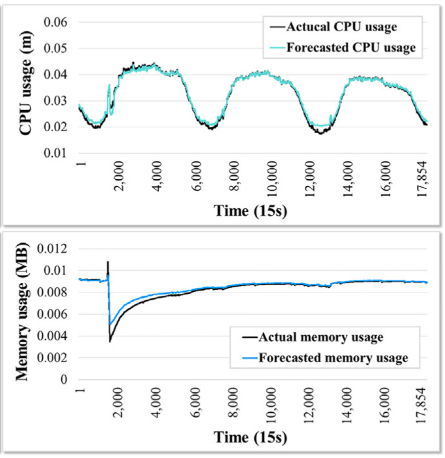
The attention-based Bi-LSTM model had MSE, RMSE, and MAE values closer to 0 than did the other models, as shown in Table 3, where the highest R^2 value was measured at 0.9947. Therefore, it is advisable to use the attention-based Bi-LSTM model in ProHPA to accurately forecast the future resource usage.



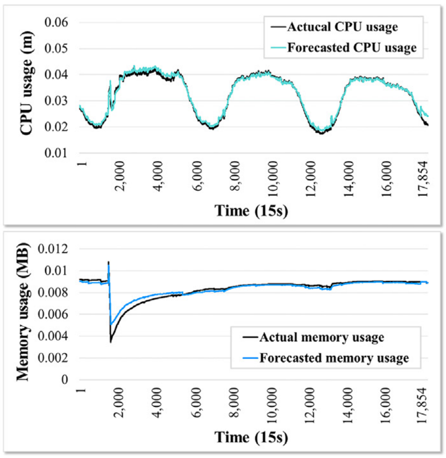
(a) Workload forecasting result of LSTM



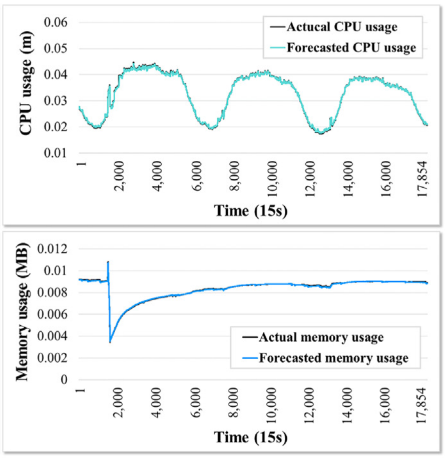
(b) Workload forecasting result of GRU



(c) Workload forecasting result of Bi-LSTM



(d) Workload forecasting result of Bi-GRU



(e) Workload forecasting result of attention-based Bi-LSTM

Fig. 7. Alloc-19519340051 forecasting results of LSTM, GRU, Bi-LSTM, Bi-GRU, and attention-based Bi-LSTM models.

Table 3
Test results of LSTM, GRU, Bi-LSTM, Bi-GRU, and attention-based Bi-LSTM models.

Deep neural network model	MSE	RMSE	MAE	R^2
LSTM	0.00079	0.0281	0.0172	0.9401
GRU	0.00088	0.0309	0.0193	0.9298
Bi-LSTM	0.00077	0.0277	0.0176	0.9537
Bi-GRU	0.00078	0.0279	0.0171	0.9429
Attention-based Bi-LSTM	**0.00011**	**0.0108**	**0.0066**	**0.9947**

To provide an optimized scaling technique for workloads with various patterns, we performed additional performance evaluation on the attention-based Bi-LSTM model in ProHPA using alloc workload data. We used two workload data which were randomly extracted from Google cluster workload traces 2019. Fig. 8 shows alloc-31881920623 workload data with similar CPU and memory usage patterns, and alloc-131695484619 workload data with different CPU and memory usage patterns.

Figs. 9 and 10 show the test results of the deep neural network model constructed by learning workload data with epoch 100. In Fig. 9, the attention-based Bi-LSTM model forecasted the actual workload pattern and peak value more closely than the LSTM model, GRU model, Bi-LSTM model, and Bi-GRU model. In addition, in Fig. 10, the other four models showed a large error between the forecasted value and the actual value. In contrast, the attention-based Bi-LSTM model accurately forecasted the actual value.

Table 4 shows the benchmark test results of the LSTM model, GRU model, Bi-LSTM model, Bi-GRU model, and attention-based Bi-LSTM model in terms of performance indices. The R^2 score of the attention-based Bi-LSTM model is approximately 0.96, and the MSE, RMSE, and MAE values are close to 0, thus demonstrating higher forecasting accuracy than other models. Therefore, the attention-based Bi-LSTM model within ProHPA forecasts with high accuracy CPU and memory usage showing various patterns.

The workload of Google cluster workload traces 2019 was simulated to verify the performance of ProHPA in an environment similar to that of an actual container-based cluster. A resource consumption image provided by Kubernetes was utilized to generate a load on the CPU and memory.

Table 5 presents the workload data configuration of alloc-19519340051. It was assumed that a user allocates the same CPU and memory request because it is difficult to judge the resource usage required in the future, and the maximum request was allocated as 2000 (m/MB) while the minimum request was allocated as 30 (m/MB) [7,8]. No limits were specified so that the number of overloads occurring for the workload could be measured.

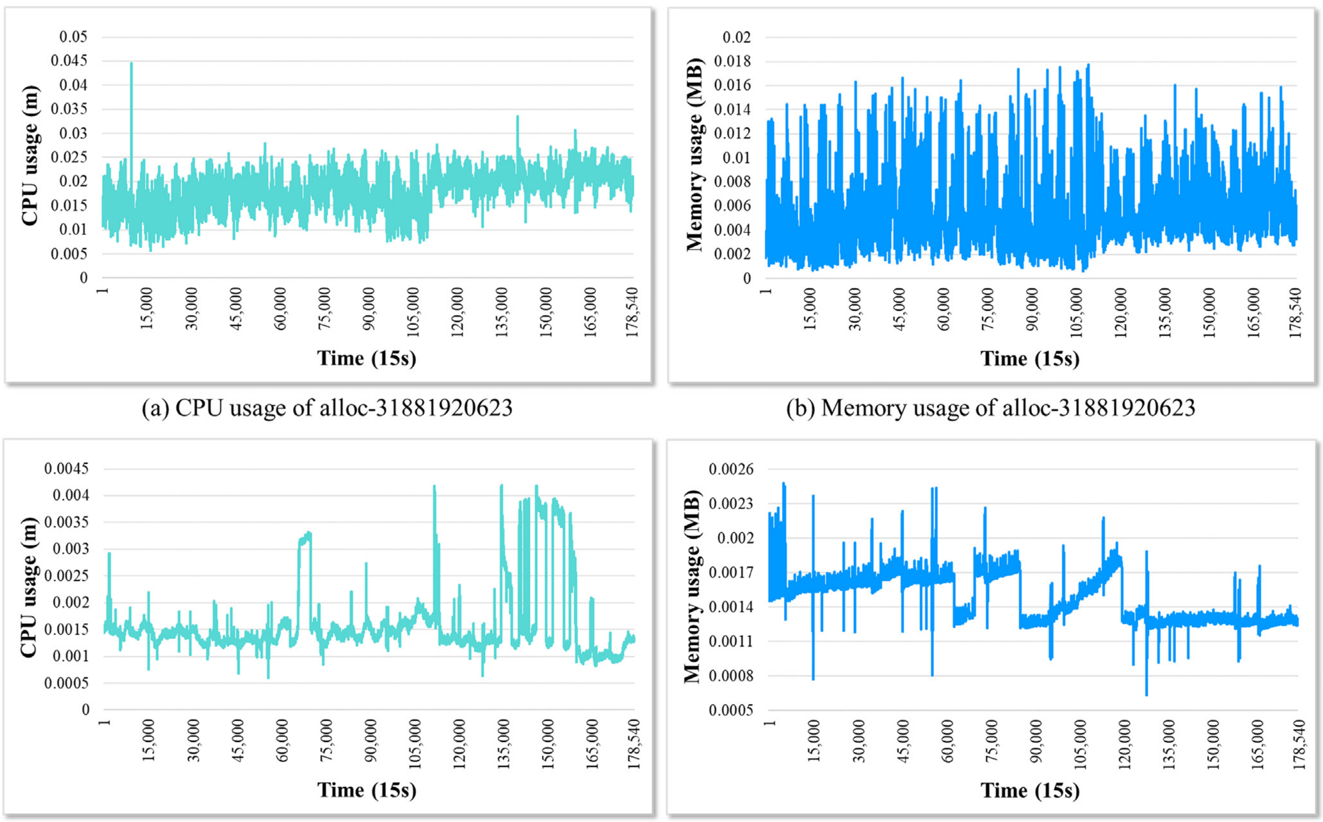
Table 6 presents a comparison of average resource utilization calculated from ProHPA and the conventional HPA of Kubernetes when CPU and memory requests are overallocated in a pod.

When HPA requests were each 400 (m/MB) and were overallocated to one resource, memory was reduced to 90 MB, which was at the 90th percentile of forecasted memory usage through ReVPA, and then PoHPA was performed. Because CPU and memory usage patterns varied, the average CPU utilization was reduced by around 10%, but the average memory utilization was increased by 30%.

When both CPU and memory requests were overallocated at 1,000, 1,500, and 2,000 (m/MB), each resource was reduced to 416 m and 90 MB, or the 90th percentile of forecasted resource usage, through ReVPA, and then autoscaling was performed by PoHPA. As the allocation of resources increased, autoscaling became impossible with conventional HPA, and thus the average resource utilization gradually decreased. To the contrary, the proposed ProHPA reduced the resource amount to stably carry out future workloads, and then performed proactive HPA to more stably and efficiently manage resources than conventional HPA.

Table 7 presents a comparison of maximum resource utilization and number of overloads between HPA and ProHPA when CPU and memory are insufficiently allocated.

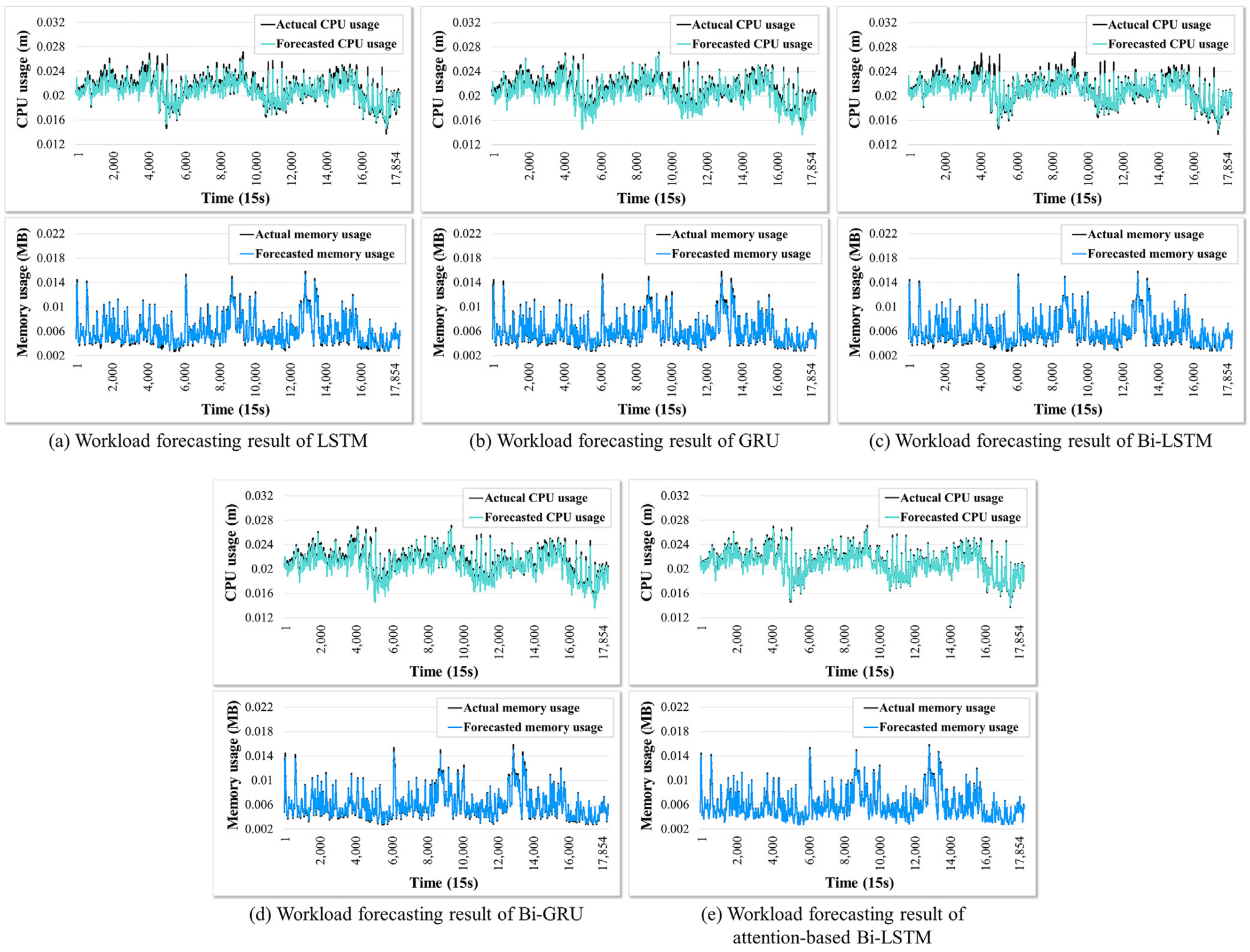
When the CPU and memory of a pod are insufficiently allocated, HPA has very high maximum resource utilization due to reactive scaling, which indicates that an overload has occurred. In contrast, ProHPA guarantees stable resource usage by providing proactive



(a) CPU usage of alloc-31881920623 (b) Memory usage of alloc-31881920623

(c) CPU usage of alloc-131695484619 (d) Memory usage of alloc-131695484619

Fig. 8. Two types of alloc workload data for a benchmark test.



(a) Workload forecasting result of LSTM (b) Workload forecasting result of GRU (c) Workload forecasting result of Bi-LSTM (d) Workload forecasting result of Bi-GRU (e) Workload forecasting result of attention-based Bi-LSTM

Fig. 9. Alloc-31881920623 forecasting results of LSTM, GRU, Bi-LSTM, Bi-GRU, and attention-based Bi-LSTM models.

scaling by forecasted resource usage through PoHPA. ProHPA experiences an overload in certain resources due to errors between predicted resource usage and actual resource usage, but significantly fewer overloads occur compared with HPA, because the attention-based Bi-LSTM model is used.

7. Conclusions

To manage resources more efficiently and stably than conventional HPA, which operates with a reactive method in Kubernetes, this study proposed ProHPA, which sequentially performs ReVPA and PoHPA. The attention-based Bi-LSTM model with built-in ProHPA performed hybrid autoscaling for resources in a pod by forecasting the future workload from a past workload having irregular patterns. When the performance of ProHPA was evaluated, the average utilization of CPU and memory was found to have improved by 23.39% and 42.52%, respectively, compared with conventional HPA when initial resources were overallocated. When initial resources were insufficiently allocated, overloads in CPU and memory rarely occurred in ProHPA compared with HPA provided by Kubernetes.

If HPA is performed when the usage patterns differ between autoscaling resource objects, or when errors are significant, specific resources may be wasted due to unnecessary scaling. Furthermore, it is challenging to elastically manage resources with conventional HPA because scaling is provided based on the threshold designated by a developer. Therefore, further research will be conducted to analyze techniques for elastically adjusting thresholds considering the changes in resource state and to add techniques that elastically provide resources to ProHPA.

CRediT authorship contribution statement

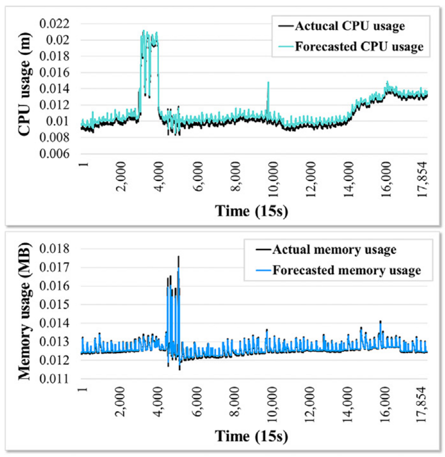
Byeonghui Jeong: Conceptualization, Methodology, Software, Validation, Writing – original draft, Writing – review & editing. **Seungyeon Baek:** Software, Validation. **Sihyun Park:** Validation, Investigation. **Jueun Jeon:** Methodology, Validation, Writing – review & editing, Supervision. **Young-Sik Jeong:** Writing – review & editing, Supervision, Project administration.

Data availability

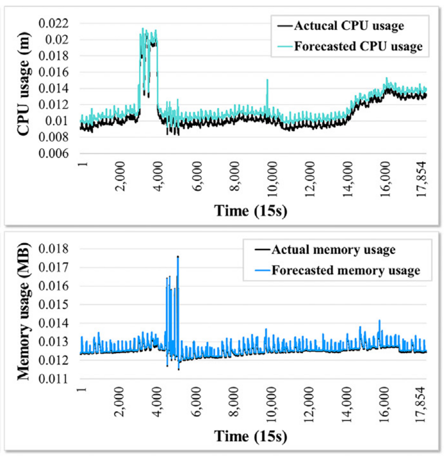
The authors do not have permission to share data.

Declaration of Competing Interest

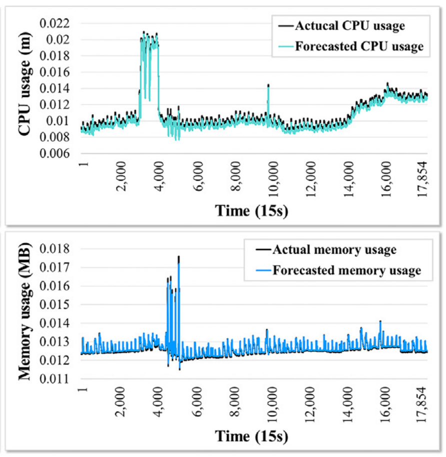
The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.



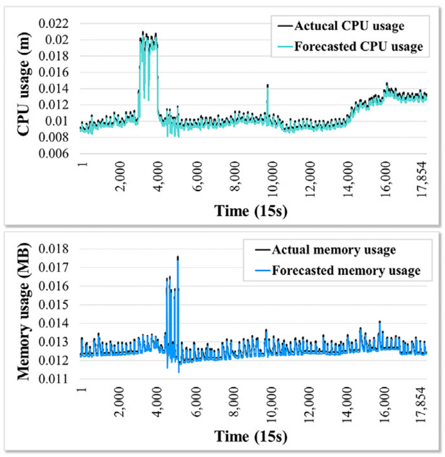
(a) Workload forecasting result of LSTM



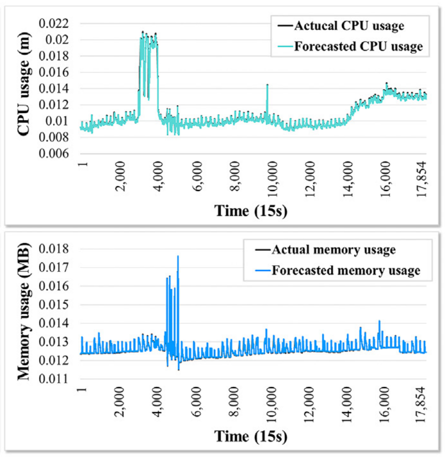
(b) Workload forecasting result of GRU



(c) Workload forecasting result of Bi-LSTM



(d) Workload forecasting result of Bi-GRU



(e) Workload forecasting result of attention-based Bi-LSTM

Fig. 10. Alloc-131695484619 forecasting results of LSTM, GRU, Bi-LSTM, Bi-GRU, and attention-based Bi-LSTM models.

Table 4
Benchmark test results of LSTM, GRU, Bi-LSTM, Bi-GRU, and attention-based Bi-LSTM models.

Alloc ID	Deep neural network model	MSE	RMSE	MAE	R^2
31881920623	LSTM	0.00041	0.02031	0.0115	0.9386
	GRU	0.00044	0.02103	0.0136	0.9305
	Bi-LSTM	0.00037	0.01928	0.0102	0.9458
	Bi-GRU	0.00035	0.01894	0.0108	0.9506
	Attention-based Bi-LSTM	**0.00029**	**0.01718**	**0.0069**	**0.9678**
131695484619	LSTM	0.00011	0.01063	0.00794	0.9235
	GRU	0.00012	0.01077	0.00816	0.9195
	Bi-LSTM	0.00009	0.00969	0.00751	0.9387
	Bi-GRU	0.00004	0.00657	0.00377	0.9473
	Attention-based Bi-LSTM	**0.00002**	**0.00502**	**0.00156**	**0.9687**

Table 5
CPU and memory usage configuration of the simulation workload data.

Resource	50th percentile	90th percentile	95th percentile	Maximum usage	Minimum usage	Average usage
CPU (m)	358	416	425	446	174	327
Memory (MB)	87	90	92	109	35	84

Table 6
Comparison of average resource utilization between HPA of Kubernetes and ProHPA when resources are overallocated.

HPA request		ProHPA request		HPA average resource utilization		ProHPA average resource utilization	
CPU (m)	Memory (MB)	CPU (m)	Memory (MB)	CPU (%)	Memory (%)	CPU (%)	Memory (%)
400	400	**400**	**90**	52.14	14.57	**41.27**	**46.69**
1000	1000	**416**	**90**	32.65	8.35	**39.71**	**46.69**
1500	1500	**416**	**90**	21.77	5.56	**39.71**	**46.69**
2000	2000	**416**	**90**	16.32	4.17	**39.71**	**46.69**

Table 7
Performance comparison between ProHPA and conventional HPA of Kubernetes when insufficient resources are allocated.

HPA & ProHPA request		HPA maximum resource utilization		ProHPA maximum resource utilization		Number of overloads in HPA		Number of overloads in ProHPA	
CPU	Memory	CPU	Memory	CPU	Memory	CPU	Memory	CPU	Memory
(m)	(MB)	(%)	(%)	(%)	(%)				
300	80	79.20	115.09	**63.81**	**67.57**	0	4	**0**	**0**
150	40	184.81	230.18	**81.34**	**85.55**	4	4	**0**	**0**
50	50	554.43	184.14	**93.43**	**34.77**	8	4	**2**	**0**
30	30	924.06	306.91	**88.23**	**36.22**	8	4	**0**	**0**

Acknowledgements

This work was partly supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2022-0-00047, Development of microservices development/operation platform technology that supports application service operation intelligence) and National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2019R1A2C1088383).

References

[1] J. Soldani, D.A. Tamburri, W.J. Van Den Heuvel, The pains and gains of microservices: A systematic grey literature review, J. Syst. Softw. 146 (2018) 215–232.
[2] P. Jamshidi, C. Pahl, N.C. Mendonca, J. Lewis, S. Tilkov, Microservices: The journey so far and challenges ahead, IEEE Softw. 35 (2018) 24–35.
[3] A.M. Potdar, D.G. Narayan, S. Kengond, M.M. Mulla, Performance evaluation of docker container and virtual machine, Procedia Comput. Sci. 171 (2020) 1419–1428.
[4] K.T. Seo, H.S. Hwang, I.Y. Moon, O.Y. Kwon, B.J. Kim, Performance comparison analysis of Linux container and virtual machine for building cloud, Adv. Sci. Technol. Lett. 66 (2014) 105–111.
[5] J.P. Martin, A. Kandasamy, K. Chandrasekaran, Exploring the support for high performance applications in the container runtime environment, Hum. -centric Comput. Inf. Sci. 8 (2018) 1–15.
[6] C. Yong, G.W. Lee, E.N. Huh, Proposal of container-based HPC structures and performance analysis, J. Inf. Process. Syst. 14 (2018) 1398–1404.
[7] Kubernetes, www.kubernetes.io/, (accessed 12 November 2021).
[8] L. Toka, G. Dobreff, B. Fodor, B. Sonkoly, Machine learning-based scaling management for Kubernetes edge clusters, IEEE Trans. Netw. Service Manag. 18 (2021) 958–972.
[9] C. Reiss, A. Tumanov, G.R. Ganger, R.H. Katz, M.A. Kozuch, Heterogeneity and dynamicity of clouds at scale: Google trace analysis, in: Proceedings of the 3rd ACM Symposium on Cloud Computing (2012) 1-13.
[10] X. Sun, C. Hu, R. Yang, P. Garraghan, T. Wo, J. Xu, J. Zhu, C. Li, Rose: Cluster resource scheduling via speculative over-subscription, in: Proceedings of the IEEE 38th International Conference on Distributed Computing Systems (2018) 949-960.
[11] H.W. Kim, J.H. Park, Y.S. Jeong, Human-intelligence workflow management for the big data of augmented reality on cloud infrastructure, Neurocomputing 279 (2018) 19–26.
[12] P. Sahu, S. Raghavan, K. Chandrasekaran, Ensemble deep neural network based quality of service prediction for cloud service recommendation, Neurocomputing 465 (2021) 476–489.
[13] J. Jeon, J.H. Park, Y.S. Jeong, Resource utilization scheme of idle virtual machines for multiple large-scale jobs based on OpenStack, Appl. Sci. (Basel) 9 (2019) 1–16.
[14] S. Chouliaras, S. Sotiriadis, Auto-scaling containerized cloud applications: A workload-driven approach, Simul. Model. Pract. Theory 121 (2022) 1–13.
[15] H.W. Kim, G. Yi, J.H. Park, Y.S. Jeong, Adaptive resource management using many-core processing for fault tolerance based on cyber–physical cloud systems, Future Gener. Comput. Syst. 105 (2020) 884–893.
[16] D. Saxena, A.K. Singh, A proactive autoscaling and energy-efficient VM allocation framework using online multi-resource neural network for cloud data center, Neurocomputing 426 (2021) 248–264.
[17] D.D. Vu, M.N. Tran, Y. Kim, Predictive hybrid autoscaling for containerized applications, IEEE Access 10 (2022) 109768–109778.
[18] J. Bi, S. Li, H. Yuan, M.C. Zhou, Integrated deep learning method for workload and resource prediction in cloud systems, Neurocomputing 424 (2022) 35–48.
[19] Y.S. Jeong, J.H. Park, Security, privacy, and efficiency of sustainable computing for future smart cities, J. Inf. Process. Syst. 16 (2020) 1–5.
[20] N.M. Dang-Quang, M. Yoo, Deep learning-based autoscaling using bidirectional long short-term memory for Kubernetes, Appl. Sci. (Basel) 11 (2021) 1–25.
[21] M. Yan, X.M. Liang, Z.H. Lu, J. Wu, W. Zhang, HANSEL: Adaptive horizontal scaling of microservices using Bi-LSTM, Appl. Soft Comput. 105 (2021) 1–12.
[22] T. Wang, S. Ferlin, M. Chiesa, Predicting CPU usage for proactive autoscaling, in: Proceedings of the 1st Workshop on Machine Learning and Systems (2021) 31-38.
[23] D. Buchaca, J.L. Berral, C. Wang, A. Youssef, Proactive container auto-scaling for cloud native machine learning services, in: Proceedings of the IEEE 13th International Conference on Cloud Computing (2020) 475-479.
[24] F. Rossi, M. Nardelli, V. Cardellini, Horizontal and vertical scaling of container-based applications using reinforcement learning, in: Proceedings of the IEEE 12th International Conference on Cloud Computing (2019) 329–338.
[25] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand, J. Wilkes, Autopilot: workload autoscaling at Google, in: Proceedings of the 15th European Conference on Computer Systems (2020) 1–16.
[26] T.T. Nguyen, Y.J. Yeom, T. Kim, D.H. Park, S. Kim, Horizontal pod autoscaling in Kubernetes for elastic container orchestration, Sensors 20 (2020) 1–18.
[27] K. Om, S. Boukoros, A. Nugaliyadde, T. McGill, M. Dixon, P. Koutsakis, K.W. Wong, Modelling email traffic workloads with RNN and LSTM models, Hum. -centric Comput. Inf. Sci. 10 (2020) 1–16.
[28] D.H. Kwon, J.B. Kim, J.S. Heo, C.M. Kim, Y.H. Han, Time series classification of cryptocurrency price trend based on a recurrent LSTM neural network, J. Inf. Process. Syst. 15 (2019) 694–706.
[29] J. Jeon, B. Jeong, S. Baek, Y.S. Jeong, Hybrid malware detection based on Bi-LSTM and SPP-net for smart IoT, IEEE Trans. Industr. Inform. 18 (2022) 4830–4837.
[30] S. Baek, J. Jeon, B. Jeong, Y.S. Jeong, Two-stage hybrid malware detection using deep learning, Hum. -centric Comput. Inf. Sci. 11 (2021) 1–15.
[31] S. Lee, D.B. Lee, S.J. Hwang, Contrastive learning with adversarial perturbations for conditional text generation, arXiv preprint arXiv:2012.07280 (2020).

[32] S. Taherizadeh, M. Grobelnik, Key influencing factors of the Kubernetes auto-scaler for computing-intensive microservice-native cloud-based applications, Adv. Eng. Softw. 140 (2020) 1–11.
[33] M.A.H. Monil, R.M. Rahman, Implementation of modified overload detection technique with VM selection strategies based on heuristics and migration control, in: Proceedings of the IEEE/ACIS 14th International Conference on Computer and Information Science (2015) 223–227.
[34] F. Al-Haidari, M. Sqalli, K. Salah, Impact of CPU utilization thresholds and scaling size on autoscaling cloud resources, in: Proceedings of the IEEE 5th International Conference on Cloud Computing Technology and Science (2013) 256–261.
[35] Z. Zhong, R. Buyya, A cost-efficient container orchestration strategy in Kubernetes-based cloud computing infrastructures with heterogeneous resources, ACM Trans. Internet Technol. 20 (2020) 1–24.
[36] F. Rossi, V. Cardellini, F.L. Presti, Hierarchical scaling of microservices in Kubernetes, in: Proceedings of the IEEE 1st International Conference on Autonomic Computing and Self-Organizing Systems (2020) 28-37.
[37] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, J. Wilkes, Large-scale cluster management at Google with Borg, in: Proceedings of the 10th European Conference on Computer Systems (2015) 1–17.
[38] M. Imdoukh, I. Ahmad, M.G. Alfailakawi, Machine learning-based auto-scaling for containerized applications, Neural Comput. Appl. 32 (2020) 9745–9760.

Byeonghui Jeong received his B.S. degree in computer science and engineering from Kongju National University in Cheonan, Korea in 2021. He is an M.S. student of the department of multimedia engineering at Dongguk University, Korea. His current research interests include cloud computing and information security for cloud computing.

Seungyeon Baek received his B.S. degree in multimedia engineering from Dongguk University in Seoul, Korea in 2021. He is an M.S. student of the department of multimedia engineering at Dongguk University, Korea. His current research interests include information security for IoT devices and malware detection using deep learning.

Sihyun Park received his B.S. degree in multimedia engineering from Dongguk University in Seoul, Korea in 2022. He is an M.S. student of the Department of Multimedia Engineering, Dongguk University, Korea. His current research interests include cloud computing, the Internet of Things, edge computing.

Jueun Jeon received her B.S. and M.S. degrees in multimedia engineering from Dongguk University in Seoul, Korea in 2018 and 2020. She is a Ph.D. student of the department of multimedia engineering at Dongguk University, Korea. Her current research interests include information security for cloud computing and the Internet of Things.

Young-Sik Jeong received his B.S. degree in mathematics and his M.S. and Ph.D. degrees in computer science and engineering from Korea University in Seoul, Korea in 1987, 1989, and 1993, respectively. He was a Professor in the Department of Computer Engineering at Wonkwang University, Korea from 1993 to 2012. He worked and conducted research at Michigan State University and Wayne State University as a Visiting Professor in 1997 and 2004. He currently works in the Department of Multimedia Engineering at Dongguk University, Korea. His research interests include multimedia cloud computing, information security for cloud computing, and the Internet of Things.