

# HGraphScale: Hierarchical Graph Learning for Autoscaling Microservice Applications in Container-based Cloud Computing

Zhengxin Fang, *Graduate Student Member, IEEE*, Hui Ma, *Senior Member, IEEE*, Gang Chen, *Senior Member, IEEE*, and Rajkumar Buyya, *Fellow, IEEE*

**Abstract**—Microservice architecture has become a dominant paradigm in application development due to its advantages of being lightweight, flexible, and resilient. Deploying microservice applications in the container-based cloud enables fine-grained elastic resource allocation. Autoscaling is an effective approach to dynamically adjust the resource provisioned to containers. However, the intricate microservice dependencies and the deployment scheme of the container-based cloud bring extra challenges of resource scaling. This article proposes a novel autoscaling approach named HGraphScale. In particular, HGraphScale captures microservice dependencies and the deployment scheme by a newly designed hierarchical graph neural network, and makes effective scaling actions for rapidly changing user requests workloads. Extensive experiments based on real-world traces of user requests are conducted to evaluate the effectiveness of HGraphScale. The experiment results show that the HGraphScale outperforms existing state-of-the-art autoscaling approaches by reducing at most 80.16% of the average response time under a certain VM rental budget of application providers.

**Index Terms**—Autoscaling, Microservice application, Container-based cloud, Graph neural network, Deep reinforcement learning

## I. INTRODUCTION

Microservice applications is marking a paradigm shift in how software systems are designed and managed [8]. These modern applications are composed of lightweight and scalable microservices, which improve scalability, agility, and resilience [7]. Each microservice is instantiated by one or more containers. Building on this paradigm, cloud computing serves as a critical enabler for hosting and scaling microservice applications [1].

The dynamic resource adjustment in cloud computing, known as *autoscaling* [4], [28], [30], [53], empowers microservice applications to efficiently handle fluctuating user requests [51] by leveraging *horizontal scaling* and *vertical scaling* techniques. Horizontal scaling creates or deletes replicas of containers, while vertical scaling adjusts the resources (e.g., CPU) provisioned to individual container.

The effectiveness of autoscaling is further enhanced by the container-based cloud [14], [21], [46], which offers fine-grained resource allocation tailored to dynamic workloads.

Z. Fang, H. Ma and G. Chen are with the School of Engineering and Computer Science & Centre for Data Science and Artificial Intelligence, Victoria University of Wellington, Wellington, New Zealand. E-mail: {zhengxin.fang, hui.ma, aaron.chen}@ecs.vuw.ac.nz.

R. Buyya is with the School of Computing and Information Systems, the University of Melbourne, Melbourne, Australia. Email: rbuyya@unimelb.edu.au

As illustrated in Fig. 1, microservice applications deployed in the container-based cloud follow a hierarchical structure: containers are hosted within Virtual Machine instances (VMs), which, in turn, are deployed on Physical Machine instances (PMs).

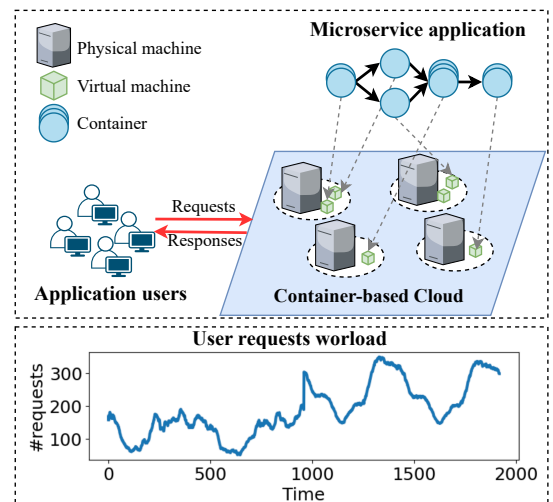


Fig. 1: Microservice application deployed in the container-based cloud with fluctuating user requests.

Within the container-based cloud, the Quality of Service (QoS) of microservice applications, such as their average response time [11], depends on the number of containers and the resources provisioned to them [4], [44], [57]. To maintain high QoS under fluctuating user requests workload (as shown in Fig. 1), this article investigates the critical problem of *Autoscaling Microservice applications in the Container-based cloud*, referred to as the AMC problem in the remaining of this paper.

In the AMC problem, provisioning excessive resources to containers may help meet the service level objectives (SLOs) of application providers but often leads to significant resource wastage [3], [37]. The cloud resources wastage increases cost for application providers due to unnecessary VM rentals [25], [44], [57]. Additionally, resource constraints at the PM and VM levels further increase the complexity of container resource provisioning and autoscaling.

Given the above challenges, an effective autoscaling approach is essential to enhance the QoS of applications while adhering to a defined cost budget. However, many existing

approaches rely on simple threshold-based mechanisms, such as Amazon auto-scaling service [2] and Horizontal Pod Autoscaler (HPA) [9]. These methods make scaling actions based on a pre-defined threshold. Nonetheless, manually selecting a threshold for changing workload is non-trivial. An inappropriate threshold can easily result in under-provisioning, leading to QoS degradation, or over-provisioning, causing unnecessarily high costs [26].

To address the limitations of threshold-based approaches, Deep Reinforcement Learning (DRL) is a promising approach for autoscaling [4], [31], [33], [34]. It can automatically learn generalizable scaling policies that adapt to dynamic environments. DRL-based methods employ deep neural networks, such as Graph Neural Networks (GNNs), to obtain container embeddings. These embeddings capture implicit characteristics and complex dependencies among containers, which are then used to guide scaling decisions. However, it is not intuitive to design an effective embedding learning approach for the AMC problem since the container-based cloud is a rather complex system. Two major issues have not been addressed by existing studies.

First, existing DRL-based autoscaling approaches [4], [34], [44] do not explicitly consider the *deployment scheme* of the container-based cloud [10]. The deployment scheme models the hierarchical placement of containers to VMs and VMs to PMs, together with network communication among containers. Ignoring such deployment information leads to suboptimal autoscaling decisions, increased physical network communication overhead, inefficient resource utilization, and ultimately degraded QoS. However, existing approaches only focus on the features of individual components, such as containers, VMs and PMs. The deployment scheme is critical because different deployment schemes result in varying resource constraints (e.g., the capacities of VMs and PMs), which directly influence the effectiveness of scaling actions. Ignoring this factor can lead to suboptimal scaling actions and resource utilization.

Second, GNNs employed in existing autoscaling approaches [33], [47] aggregate information in a *flat* way. In this flat structure, containers, VMs and PMs are modeled in a single layer. As a result, long-range dependencies (e.g., between containers on different PMs) require many message-passing steps to capture. Studies [32], [55], [61] have shown that the flat GNN structure cannot effectively capture long-range dependencies for learning node embeddings. This issue becomes more aggravated in the AMC problem with the increasing number of containers, VMs and PMs in the container-based cloud.

To address the above issues, this paper focuses on designing a novel embedding learning approach to improve the performance of the DRL-based autoscaling approach. For this purpose, we construct a three-layer hierarchical graph. It models both the dependencies among containers and the deployment scheme. The hierarchical structure of this graph, from bottom to top, is: *PM layer*, *VM layer* and *container layer*. Then, we design a novel Hierarchical Graph Neural Network (HGNN) to solve the issue of long-range information

aggregation [32], [61].

However, existing HGNN approaches [32], [59] mainly learn whole-graph embeddings by aggregating information in a fine-to-coarse manner. These approaches are effective for tasks that require holistic graph representations. However, it is not well-suited for the AMC problem, where precise scaling actions depend on embeddings at the granularity of individual containers rather than the entire graph.

To fill this gap, we proposed a *Cloud-oriented Hierarchical Graph Neural Network (CHGNN)*, which is an HGNN designed to effectively learn container embeddings from cloud environment. Unlike traditional methods, CHGNN first aggregates information locally within lower-layer nodes and then propagates it to higher layers. This *bottom-up information aggregation mechanism* establishes shortcut connections [22], [32], [55] between distant nodes in the graph, enabling effective processing of global context. Consequently, CHGNN can capture comprehensive global information from the container-based cloud for the container layer. This mechanism not only represents a departure from existing HGNN paradigms but also delivers a more precise and scalable solution for the AMC problem.

Through developing CHGNN, this paper makes the following main contributions:

- We represent the container-based cloud as a three-layer hierarchical graph. Meanwhile, we design a novel HGNN, i.e., CHGNN, to learn container embedding from the hierarchical graph. To our knowledge, this is the first work to learn embedding for autoscaling using HGNN, allowing to make more effective scaling actions for the AMC problem than existing approaches.
- We propose a novel bottom-up information aggregation mechanism for CHGNN to effectively capture thorough global information from the container-based cloud. This mechanism provides an accurate and scalable autoscaling solution to the AMC problem.
- We propose a novel DRL-based autoscaling approach that leverages CHGNN with a bottom-up information aggregation mechanism to effectively learn container embeddings. In addition, a newly designed *scaling policy network* is employed to make scaling decisions. We name this autoscaling approach as HGraphScale. Experiment results based on real-world traces indicate that HGraphScale can outperform five state-of-the-art autoscaling approaches.

The rest of this article is organized as follows. Section II presents the literature review of existing autoscaling approaches. Section III presents formal problem definitions of the problem. Section IV gives details of HGraphScale for autoscaling. The experiment designs, results and further analysis are shown in Section V. At last, Section VI makes conclusions and gives potential future directions.

## II. RELATED WORK

In this section, we review existing autoscaling approaches for microservice applications.

TABLE I: Comparison of HGraphScale with DRL-based and GNN-based autoscaling approaches

	Approaches										
	[31]	[19]	[34]	[44]	[4]	[20]	[37]	[33]	[38]	[31]	Ours
Vertical scaling				✓	✓						✓
Horizontal scaling	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
QoS improvement	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Cost saving	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓
High dimensional state			✓	✓	✓		✓	✓	✓	✓	✓
Microservice dependency	✓					✓	✓	✓	✓	✓	✓
Deployment scheme											✓

#### A. Heuristic-based Autoscaling

AWS-Scale [2] and Horizontal Pod Auto-scaler (HPA) [9] are autoscaling techniques that rely on manually determined thresholds. For example, the resources provisioned to containers are increased if the resource utilization is higher than a given threshold; otherwise, it decreases the resources provisioned to containers. However, manually designing a threshold for changing workload is challenging.

To address the above issue, some heuristic-based autoscaling approaches are proposed to make scaling actions based on predicted future workload. ProScale [11] is a proactive autoscaling method that leverages the accurate and fast Simple Moving Average (SMA) to predict future request workloads. Then, the resource adjustment of containers is based on a greedy method. PBScaler [57] is proposed to detect the bottleneck microservices in an application. Subsequently, a genetic algorithm is applied to decide the number of containers required by bottleneck microservices. StatusScale [53] is a status-aware autoscaling approach that selects appropriate autoscaling strategies for resource scheduling based on load status.

The above autoscaling methods require substantial human efforts to design the heuristics or fine-tune the thresholds. Meanwhile, the heuristics methods exhibit poor generalization ability in dynamically changing environments [58].

#### B. Reinforcement Learning-based Autoscaling

Existing studies [4], [19], [20], [34], [44], [60] have shown that the RL-based autoscaling methods can effectively adjust the resource allocation to handle the changing workload. For example, A-SARSA is proposed [60] to combine neural network based workload prediction and the SARSA algorithm [52] to make scaling actions based on predicted workload.

A Q-learning based autoscaling approach [19] is proposed for workflow autoscaling, which considers the workflow structure when making scaling actions. [20] further compared the performance of Q-learning and SARSA for workflow autoscaling, considering the workflow structures. Their results show that SARSA can achieve significantly better performance in many scenarios compared to Q-learning.

The above approaches use table-based RL techniques, struggling to handle high-dimensional state spaces. To tackle this limitation, the DRL-based autoscaling approach has been gaining more attention in recent years. For instance, a Deep Q-Network (DQN) [35] based autoscaling method, called

HRA [34], is proposed to make holistic autoscaling actions for microservice applications. Similarly, DeepScale [44] integrates DQN and heuristics methods to make scaling actions for applications. DRPC [4] is a TD3 [18] based DRL approach to make scaling actions based on embedding learned by multiple distributed neural networks. ASTRA [30], a recently introduced approach, leverages an adversarial DRL algorithm for autoscaling.

The above DRL-based approaches fail to explicitly consider the deployment scheme, which impacts scaling actions. This hinders the effectiveness of these methods in addressing the AMC problem.

#### C. Graph Neural Network-based Autoscaling

Beyond heuristic and DRL-based autoscaling methods, GNN-based approaches have emerged as a popular solution for autoscaling. For instance, DeepScaler [33] is proposed to estimate resource utilization by GNN, which is further utilized to guide the autoscaling decisions. GRAF [37], [38] is proposed to predict tail latency of microservice applications. The predicted latency is leveraged for proactive autoscaling decision making. AGQ [31] is an autoscaling approach that utilizes a GNN-based resource usage predictor, which directly informs the autoscaler's scaling decisions.

In a summary, existing GNN-based autoscaling approaches are designed for prediction tasks, either forecasting resource usage or predicting latency. Such prediction tasks require large datasets for training, and these GNNs do not account for the deployment scheme in the cloud environment.

#### D. Summary

To address the above limitations of existing autoscaling approaches, this article proposes HGraphScale, a novel DRL-based autoscaling approach that incorporates a newly designed GNN and information aggregation mechanism. The details of comparison between HGraphScale and other DRL-based and GNN-based autoscaling approaches are shown in TABLE I

### III. PROBLEM DESCRIPTION

In this section, we formulate the problem of Autoscaling Microservice application in the Container-based cloud (the AMC problem).

### A. System Overview

Fig. 2 presents the system model of the container-based cloud, where containers of each microservice application are deployed on VMs, which are further hosted by PMs. A microservice application can be modeled as a Directed Acyclic Graph (DAG)  $App = \langle V_{app}, E_{app} \rangle$ , as shown in Fig. 2.  $V_{app} = \{ms_0, ms_1, \dots, ms_n\}$  represents  $n$  microservices.  $e_{ij}^{app} \in E_{app}$  denotes the execution dependency between a pair of adjacent microservices  $ms_i$  and  $ms_j$ . Following existing studies [5], [14], [50], [62], a microservice  $ms_i$  is instantiated by at least one container  $Con_i^j$ , where  $j$  denotes the index of the container.

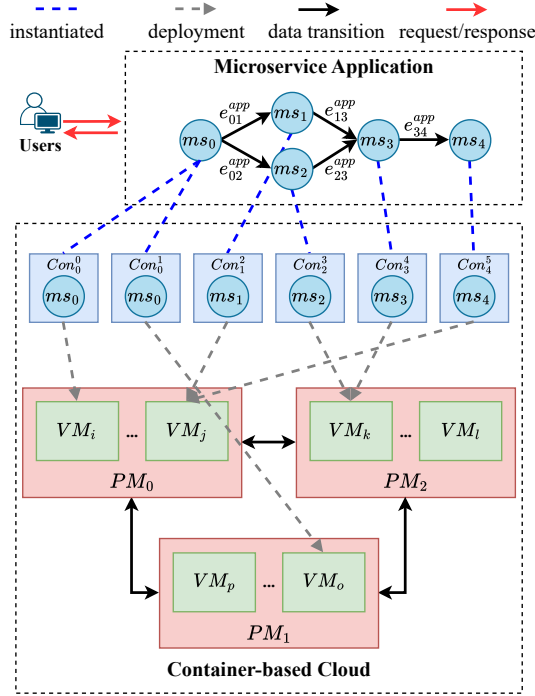


Fig. 2: An example microservice application

Each user request triggers the execution of a workflow instance  $WF = \langle V_{wf}, E_{wf} \rangle$  [50].  $V_{wf} = \{t_{start}, t_1, t_2, \dots, t_n, t_{end}\}$  denotes tasks in a user request, where  $t_{start}$  and  $t_{end}$  are dummy starting and ending tasks, respectively.  $e_{ij}^{wf} \in E_{wf}$  represents  $t_i$  is the predecessor task of  $t_j$  while  $t_j$  is the successor task of  $t_i$ . Task  $t_i$  can only be executed by a container of the corresponding microservice  $ms_i$ .

### B. QoS and Cost Modeling

Let  $et_i$  denote the execution time of  $t_i$  with one vCPU, and  $concpu_i^j$  denotes the amount of vCPU provisioned to  $Con_i^j$ . This study focus on resource adjustment of vCPU [12], [38], [44]. This is because existing studies showed that CPU is the dominant factor affecting microservice application response time [29], [38], [53]. Accordingly, the execution time of  $t_i$  in  $Con_i^j$  is

$$ET_i^j = \frac{et_i}{concpu_i^j}. \quad (1)$$

As assumed in existing studies [5], [42], [50], [58], a container can execute at most one task at any time. Meanwhile, each container maintains a pending queue of task, following [42], [44]. Each task starts execution only after the preceding task in the queue has been completed. As a result, the finish time  $FT$  of a task  $t_i$  in  $Con_i^j$  is calculated by:

$$FT_i^j = ST_i^j + ET_i^j, \quad (2)$$

where  $ST_i^j$  indicates the start time of  $t_i$  in  $Con_i^j$ . Particularly,  $ST_i^j$  is defined as

$$ST_i^j = WT_i^j + FT^{pre_i} + TT^{pre_i}, \quad (3)$$

where  $WT_i^j$  is the waiting time of  $t_i$  in the pending queue of  $Con_i^j$ .  $FT^{pre_i}$  denotes the finish time of predecessor tasks ( $t_{pre_i}$ ) of  $t_i$ .  $TT^{pre_i}$  is the data transmission time from predecessor task.

Let  $req_r$  represent a user request for a microservice application, the response time  $RT_r$  of  $req_r$  is calculated by:

$$RT_r = FT_{end} \quad (4)$$

As shown in Fig. 2, each container is deployed in a VM instance, while each VM is deployed in a PM instance [14], [15], [46], [50]. A VM/PM instance can host multiple container/VM instances.  $VM_k = (vmcpu_k, price_k)$  represents a VM instance, where  $vmcpu_k$  denotes the amount of vCPU provided by  $VM_k$  (i.e., CPU capacity) and  $price_k$  indicates the hourly rental fee. The CPU capacity of a PM instance constrains the total CPU capacity of the VMs deployed in it, which further limits the available amount of vCPU provisioned to containers deployed in those VMs. The rental fee  $Cost_k$  of any VM instance  $VM_k$  is calculated by:

$$Cost_k = price_k \times \frac{FT_{last}^k - ST_{first}^k}{3600}, \quad (5)$$

where  $FT_{last}^k$  and  $ST_{first}^k$  are the finish time and the start time of the last task and the first task executed in  $VM_k$ , respectively. The total cost  $Cost(T)$  of renting VMs over a period of time  $T$  is calculate by:

$$Cost(T) = \sum_{k \in ACT_{VM}(T)} Cost_k, \quad (6)$$

where  $ACT_{VM}(T)$  is the index set of active VMs over  $T$ .

### C. Optimization Objective

In this article, we evaluate the QoS of microservice applications by *Average Response Time (ART)* [4], [10], [44]. Therefore, the aim of the AMC problem is to autoscaling containers to minimize the *ART* over a time period  $T$  while the  $Cost(T)$  is under a cost budget, which is formulated as

$$\min ART(T) = \min \frac{\sum_{r \in REQ(T)} RT_r}{num} \quad (7)$$

$$\text{s.t. } Cost(T) \leq budget(T) \quad (8)$$

where  $REQ(T)$  is the index set of user requests over  $T$  and  $num$  is the number of requests.  $budget(T)$  is the cost budget of an application provider given over  $T$ .



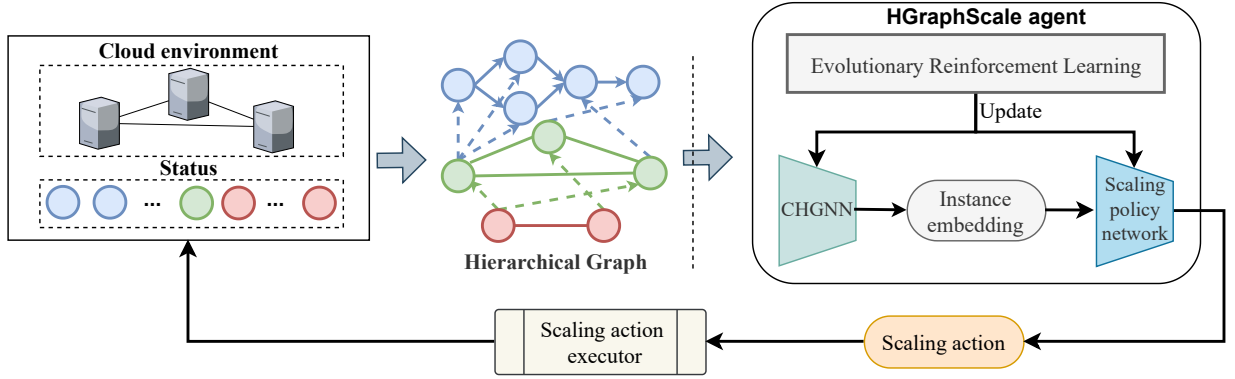


Fig. 3: The overall framework of HGraphScale for the AMC problem.

The number of user requests sent to microservice applications varies over time. Thus the autoscaling process needs to dynamically 1) identify the containers that require scaling, and 2) determine the optimal amount of scaling resources.

#### IV. PROPOSED AUTOSCALING APPROACH

The details of HGraphScale are introduced in this section. Specifically, we model the process of the AMC problem as a Reinforcement Learning (RL) problem in Section IV-A. The overall framework of HGraphScale is shown in Fig. 3. In each iteration, the state of the container-based cloud is extracted and represented as a hierarchical graph, detailed in Section IV-B. Then, a novel *Cloud-oriented Hierarchical Graph Neural Network (CHGNN)* is proposed to learn the embedding of every container, introduced in Section IV-C. The learned container embedding is fed into a newly designed scaling policy network to produce scaling actions, as described in Section IV-D. A *scaling action executor* performs either vertical or horizontal scaling in the container-based cloud based on the scaling actions, as outlined in Section IV-E. Section IV-F presents how HGraphScale handles load balancing.

Evolutionary Reinforcement Learning (ERL) [25], [40], a widely recognized and practically popular algorithm, is leveraged to train the neural networks of HGraphScale. This is because ERL demonstrates strong exploration ability, ensures a stable training process, and requires relatively few hyperparameters for fine-tuning [25], [40]. Moreover, recent studies have shown its effectiveness in several cloud-related applications [25], [41]. The detailed process of training by ERL is provided in Section IV-G.

##### A. RL Formulation

We formulate the process of solving AMC problems as an RL problem. Particularly, at each decision step  $t$ , the cloud environment provides the state  $s_t$  as a hierarchical graph. The HGraphScale agent in Figure 3 generates a scaling action  $a_t$  based on  $s_t$ . The environment then performs  $a_t$  and transitions to the next state  $s_{t+1}$ . The key components of this RL problem are outlined below.

1) *State*: Each state  $s_t$  is a snapshot of the status of the PMs, VMs and containers in the container-based cloud at a decision step  $t$ . We design a novel hierarchical graph  $\mathcal{H} = \langle \mathcal{V}, \mathcal{E} \rangle$  to represent the states  $s_t$ , detailed in Section IV-B. The number of nodes in this hierarchical graph dynamically changes with the addition or removal of containers.

The status of a PM instance  $p$  is defined as  $h_p^{pm} = \{\mu_{pm_p}, \Omega_{pm_p}\}$ , which denotes the resource utilization ( $\mu_{pm_p}$ ) and the capacity ( $\Omega_{pm_p}$ ) of  $pm_p$ , respectively. Status of a VM instance  $v$  is defined as  $h_v^{vm} = \{\mu_{vm_v}, \Omega_{vm_v}, price_{vm_v}, rental_{vm_v}, art_{vm_v}\}$ , which indicates the resource utilization ( $\mu_{vm_v}$ ), the capacity ( $\Omega_{vm_v}$ ), the per hour price ( $price_{vm_v}$ ), the current rental fees ( $rental_{vm_v}$ ) of  $vm_v$ , and the average response time ( $art_{vm_v}$ ) of containers that are deployed in  $vm_v$ .

Similarly, status of a container  $c$  is defined as  $h_c^{con} = \{\Omega_{con_c}, \zeta_{con_c}, d_{con_c}, pending_{con_c}, art_{con_c}, predicted_{con_c}\}$ . Specifically,  $\Omega_{con_c}$  denotes the resource capacity of  $con_c$ .  $\zeta_{con_c}$  represents the remaining resources of the VM that hosts  $con_c$ , indicating that container autoscaling is constrained by the resources of its hosting VM. The degree of  $con_c$  in the graph is denoted as  $d_{con_c}$ . Moreover,  $pending_{con_c}$  denotes pending requests,  $art_{con_c}$  the average response time, and  $predicted_{con_c}$  the future workload of  $con_c$ .

This article follows [11] to employ an effective and efficient workload predicting method, i.e., the SMA method, to predict the number of future requests  $predicted_{con_i}$  for a container  $con_i$ . The predicted future workload is based on the information from the historical workload.

2) *Action*: A scaling action  $a_t$  at time  $t$  of HGraphScale is represented as a 2-dimensional tuple:  $\langle Ind, Scale \rangle$ .  $Ind \in [0, n] \cap \mathbb{Z}^+$  denotes the index of the container that requires scaling. Here,  $n$  is the current number of containers, which changes dynamically over time.  $Scale \in [-m, +m] \cap \mathbb{Z}^+$  indicates the amount of resources for scaling. The sign of  $Scale$  determines whether to increase or decrease the provisioned resources for container  $Ind$ . If  $Scale$  equals 0, it indicates that the resource provisioned to the  $Ind$  container remains unchanged.

3) *Optimization objective*: To minimize the  $ART(t)$  and ensure the cost adheres to the budget, the optimization objec-

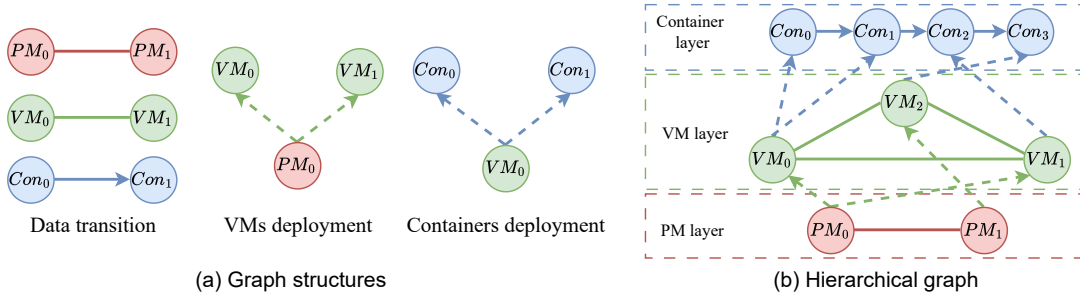


Fig. 4: An example of the hierarchical graph representation of the container-cloud.

tive of this RL problem is defined as

$$Obj(T) = -ART(T) - \rho \cdot \max(0, (Cost(T) - budget(T))) \quad (9)$$

where  $Obj(T)$  is the objective value over a time period  $T$  and  $\rho$  controls the penalty intensity when the cost exceeds the given budget.

### B. Hierarchical Graph Representation

To capture the hierarchical structure of containers, VMs and PMs in container-based clouds, as well as the communication relationships among containers, we represent the cloud state using a hierarchical graph  $\mathcal{H} = \langle \mathcal{V}, \mathcal{E} \rangle$ . Node set  $\mathcal{V} = \mathbb{C} \cup \mathbb{V} \cup \mathbb{P}$  consists of sets of container Nodes  $\mathbb{C}$ , VM nodes  $\mathbb{V}$  and PM nodes  $\mathbb{P}$ . Edge set  $\mathcal{E} = \mathbb{E}_{depvm} \cup \mathbb{E}_{depcon} \cup \mathbb{E}_{pm} \cup \mathbb{E}_{vm} \cup \mathbb{E}_{con}$ , where  $\mathbb{E}_{depvm}$  and  $\mathbb{E}_{depcon}$  represent the deployment scheme,  $\mathbb{E}_{pm}$ ,  $\mathbb{E}_{vm}$  and  $\mathbb{E}_{con}$  represent the data transition between machines.

In particular, Fig. 4(a) presents three main structures of  $\mathcal{H}$ . The edge between two PM nodes  $PM_0, PM_1 \in \mathbb{P}$  is undirected  $\{PM_0, PM_1\} \in \mathbb{E}_{pm}$ . It represents data transmission caused by interactions between containers running on different PMs, which causes additional communication overhead and thus affect QoS. Similarly, the edge between two VM nodes  $VM_0, VM_1 \in \mathbb{V}$  is also undirected  $\{VM_0, VM_1\} \in \mathbb{E}_{vm}$ . For container nodes, there exist execution orders between connected containers. Thus, the edge between two container nodes  $Con_0, Con_1 \in \mathbb{C}$  is directed, i.e.,  $(Con_0, Con_1) \in \mathbb{E}_{con}$ . Directed edges  $(PM_0, VM_0), (PM_0, VM_1) \in \mathbb{E}_{depvm}$  in the VMs deployment structures, indicating  $VM_0$  and  $VM_1$  are deployed in  $PM_0$ . Likewise, if  $Con_0$  and  $Con_1$  are deployed in  $VM_0$ , there are directed edges  $(VM_0, Con_0), (VM_0, Con_1) \in \mathbb{E}_{depcon}$ .

When new container is joined by autoscaling, a new container node is added to the container layer. If the new container requires new VM/PM for deployment, new VM/PM node are added to the VM/PM layer accordingly. Edges are then established among the container, VM, and PM nodes based on the deployment relationships.

At each decision step  $t$ , the hierarchical graph (Fig. 4(b)) serves as the system state  $s_t$  for autoscaling decisions. The hierarchical graph consists of the PM layer, VM layer and Container layer. We denote PM features as  $\mathbf{h}^{pm} = \{h_0^{pm}, \dots, h_P^{pm}\}$ , VM features as  $\mathbf{h}^{vm} = \{h_0^{vm}, \dots, h_V^{vm}\}$ ,

and container features as  $\mathbf{h}^{con} = \{h_0^{con}, \dots, h_C^{con}\}$ . The values  $P$ ,  $V$ , and  $C$  correspond to the numbers of PMs, VMs, and containers, respectively. Our newly designed CHGNN learns container embedding from this hierarchical graph and the raw features of each node.

### C. Cloud-Oriented Hierarchical Graph Neural Network

Given the hierarchical graph represented state, we proposed CHGNN to learn container embedding progressively through a bottom-up information aggregation mechanism, as shown in Fig. 5. Specifically, HCGNN first learns PM embedding in the PM layer, then PM embedding is propagated to the VM layer for VM embedding learning. At last, VM embedding is propagated to the Container layer for container embedding learning. Details of embedding learning in each layer and the bottom-up information aggregation are provided as follows.

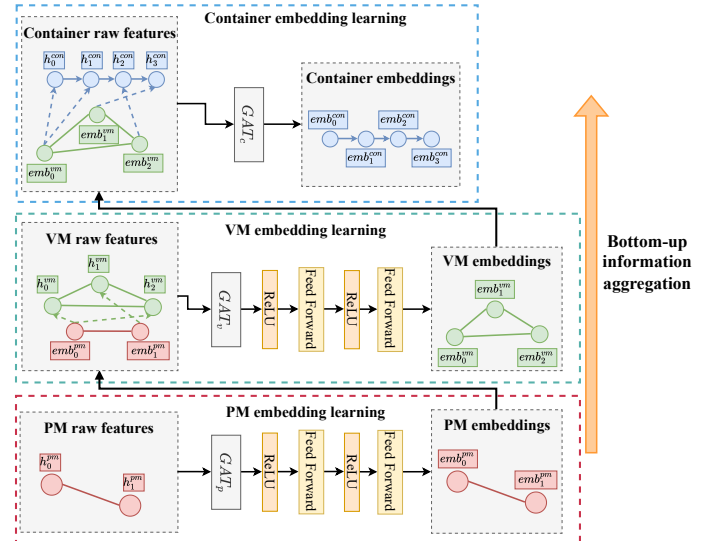


Fig. 5: The architecture of the CHGNN.

1) *Machine embedding Learning*: We stack graph attention layers [48] to construct Graph Attention Networks (GATs) (i.e.,  $GAT_p$ ,  $GAT_v$ , and  $GAT_c$  in Fig. 5) to learn embeddings of PMs, VMs and containers. The input to each graph attention layer consists of a graph and its node features. It then applies attention weights to aggregate neighbor information, resulting in updated node features.

Consider an example of learning PM embeddings by  $GAT_p$ . The  $GAT_p$  dynamically assigns attention weights  $\alpha_{i,j}$  to PM  $p_i$  and its neighbor  $p_j$  in the PM layer, indicating the importance of  $p_j$ 's features to  $p_i$  [48]. The  $\alpha_{i,j}$  is calculated by

$$\alpha_{i,j} = \frac{\exp(\text{LeakyReLU}(\vec{a}^T [\mathbf{W}_p h_i^{\vec{p}m} \parallel \mathbf{W}_p h_j^{\vec{p}m}]))}{\sum_{k \in \mathcal{N}_i} \exp(\text{LeakyReLU}(\vec{a}^T [\mathbf{W}_p h_i^{\vec{p}m} \parallel \mathbf{W}_p h_k^{\vec{p}m}]))}, \quad (10)$$

where  $\mathbf{W}_p \in \mathbb{R}^{F' \times F}$  is a learnable weight matrix in  $GAT_p$ , transforming the input features into high-level features.  $\parallel$  indicates the concatenation operation.  $\vec{a} \in \mathbb{R}^{2F'}$  is the learnable weight vector of a feedforward network. Following [48], LeakyReLU is applied for non-linear transformation.  $\mathcal{N}_i$  is the neighbor nodes of  $p_i$  in the PM layer.

The updated features  $h_i^{\vec{p}m'}$  of a  $p_i$  is generated by a graph attention layer according to

$$h_i^{\vec{p}m'} = \sigma\left(\sum_{j \in \mathcal{N}_i} \alpha_{i,j} \mathbf{W}_p h_j^{\vec{p}m}\right), \quad (11)$$

where  $\sigma$  is the sigmoid activation function that enables modeling of nonlinearity. The  $GAT_p$  outputs node embedding  $\mathbf{emb}^{\text{pm}} = \{emb_0^{\text{pm}}, \dots, emb_P^{\text{pm}}\}$  of the PM layer after passing through multiple stacked *graph attention layers*. The VM layer and container layer follow the same process of embedding learning by  $GAT_v$  and  $GAT_c$ , respectively.

2) *Bottom-Up Information Aggregation*: In our proposed bottom-up information aggregation mechanism, the learned PM embedding  $\mathbf{emb}^{\text{pm}} = \{emb_0^{\text{pm}}, \dots, emb_P^{\text{pm}}\}$  propagates to the VM layer. Thus, the inputs of  $GAT_v$  are the concatenation of VM raw features and PM embedding  $\mathbf{h}^{\text{vm}} \parallel \mathbf{emb}^{\text{pm}}$ . The VM embedding  $\mathbf{emb}^{\text{vm}} = \{emb_0^{\text{vm}}, \dots, emb_V^{\text{vm}}\}$  are learned by the  $GAT_v$  and feed forward networks.

Similarly, the inputs of the  $GAT_c$  are the concatenation of container raw features and VM embedding  $\mathbf{h}^{\text{con}} \parallel \mathbf{emb}^{\text{vm}}$ , which outputs the container embedding  $\mathbf{emb}^{\text{con}} = \{emb_0^{\text{con}}, \dots, emb_C^{\text{con}}\}$ . Through bottom-up information aggregation, the *Container layer* effectively incorporates global information of the container-based cloud into container embeddings. These embeddings allow the proposed *scaling policy network* to make system-aware scaling decisions.

#### D. Scaling Policy Network

To generate a scaling action, we design a *scaling policy network*, which takes the container embedding  $\mathbf{emb}^{\text{con}} = \{emb_1^{\text{con}}, emb_2^{\text{con}}, \dots, emb_C^{\text{con}}\}$  as input and outputs scaling actions, as illustrated in Fig. 6. A scaling action is defined as a tuple  $\langle Ind, Scale \rangle$ . To generate such actions, we design the scaling policy network with two MLPs: the *instance selector*  $MLP_\phi$  and the *scale selector*  $MLP_\omega$ .

$MLP_\phi$  is designed to calculate the priority values of each container  $i$ :

$$p_i = MLP_\phi(emb_i^{\text{con}}). \quad (12)$$

A container with a higher priority value implies a greater need for scaling. Thus, the index of the container to be scaled (e.g.,  $Ind$ ) is identified by

$$Ind = \arg \max_{i=1,2,\dots,C} (p_i), \quad (13)$$

where  $C$  is the current number of containers.

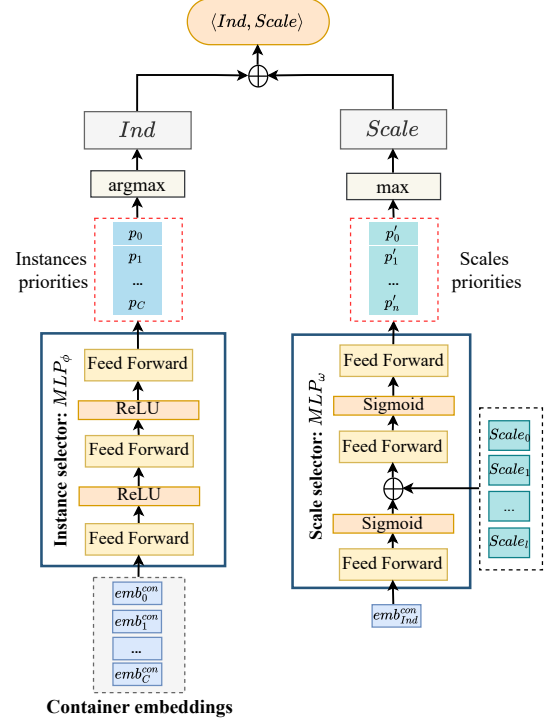


Fig. 6: The architecture of policy network

After identifying the container for scaling, the corresponding container embedding  $emb_{Ind}^{\text{con}} \in \mathbb{R}^{1 \times d}$  is selected and fed into  $MLP_\omega$ . As shown in Fig. 6,  $emb_{Ind}^{\text{con}}$  is passed through a feedforward network. The output is then concatenated with a vector  $\mathcal{S} = \{S_0, S_1, \dots, S_l\}$ , where  $S_j \in \mathbb{Z}$  indicates the amount of scaling resource, resulting in a new vector  $\mathcal{I} = \{emb_{Ind}^{\text{con}} \parallel S_0, emb_{Ind}^{\text{con}} \parallel S_1, \dots, emb_{Ind}^{\text{con}} \parallel S_l\}$ .  $\mathcal{I}$  is further processed by feed-forward networks. Finally,  $MLP_\omega$  outputs the priority  $p'_j$  of each  $Scale_j \in \mathcal{S}$ . Therefore, the amount of scaling resources is determined by

$$Scale = \max_{j=0,1,2,\dots,l} (p'_j). \quad (14)$$

Afterwards,  $Ind$  and  $Scale$  are combined to create a complete scaling action  $\langle Ind, Scale \rangle$ .

#### E. Scaling Action Executor

The scaling action executor transforms the scaling action  $\langle Ind, Scale \rangle$  to vertical scaling, horizontal scaling or both. Algorithm 1 summarizes the process of scaling action executor. Firstly, a container  $tar\_con$  is selected based on  $Ind$  (line 1).

If  $Scale > 0$ , the scaling action executor increases the resource provisioned to  $tar\_con$  (lines 3 to 12). To be specific, if the remaining CPU capacity  $max\_vcpu$  of the VM hosting

**Algorithm 1** Scaling action executor

---

**Input:** Scaling action:  $\langle Ind, Scale \rangle$   
**Output:** vertical scaling or horizontal scaling

```

1:  $tar\_con \leftarrow container\_list[Ind]$ 
2:  $tar\_vm \leftarrow$  the VM that hosts  $tar\_con$ 
3: if  $Scale > 0$  then
4:    $max\_vcpu \leftarrow$  the remaining vCPUs of  $tar\_vm$ 
5:   if  $max\_vcpu > Scale$  then  $\triangleright$  vertical scaling
6:     Increase  $Scale$  vCPUs to  $tar\_con$ 
7:   else  $\triangleright$  horizontal scaling
8:     Increase  $max\_vcpu$  vCPUs to  $tar\_con$ 
9:      $vcpu \leftarrow Scale - max\_vcpu$ 
10:    Create a new container with  $vcpu$  vCPUs
11:   end if
12: else
13:    $con\_vcpu \leftarrow$  number of vCPUs provisioned to  $tar\_vm$ 
14:   if  $con\_vcpu > Scale$  then  $\triangleright$  vertical scaling
15:     Decrease  $Scale$  vCPUs to  $tar\_con$ 
16:   else  $\triangleright$  horizontal scaling
17:     Delete  $tar\_con$ 
18:   end if
19: end if

```

---

the  $tar\_con$  is larger than the  $Scale$ , vertical scaling is applied to provision  $Scale$  vCPUs to the container (line 6); otherwise,  $max\_vcpu$  vCPUs are provisioned to  $tar\_con$ , then a new container is created by horizontal scaling. The newly created container is provisioned ( $Scale - max\_vcpu$ ) vCPUs (lines 7 to 11).

The scaling action executor reduces the resource provisioned to  $tar\_con$  when  $Scale < 0$  (lines 12 to 19). If  $Scale$  is larger than the total vCPUs of  $tar\_con$ , the vCPUs of  $tar\_con$  are reduced by the  $Scale$  number (line 15). Otherwise, the container  $tar\_con$  is deleted as a result of horizontal scaling (line 17). Note that the scaling action executor allows a microservice to be encapsulated within containers with heterogeneous resources, which can reduce resource wastage [45], [54]. As a result, a load balancer is implemented to dispatch user requests among heterogeneous containers, as detailed in Section IV-F.

**F. Capacity-based Load Balancing**

HGraphScale applies Capacity-based Weighted Round-Robin (CWRR) [6], [24], [39], [44] to dispatch user requests to a suitable container for the purpose of load balancing. Specifically, the weight  $W_j$  of a container  $Con_i^j$  is determined by

$$W_i^j = \frac{\gamma_i^j}{\sum_{k \in set(ms_i)} \gamma_i^k}, \quad (15)$$

where  $\gamma_i^j$  indicates the resource allocation of  $Con_i^j$  and  $set(ms_i)$  denotes the container set of microservice  $ms_i$ .

The rationale for adopting CWRR in HGraphScale is three-fold. First, CWRR is widely employed in practice owing to its simplicity [39]. Second, CWRR demonstrates low computation overhead in handling load balancing. Third, it provides

effective load balancing by dispatching more user requests to containers with higher capacities. Thus, CWRR can prevent any container from being heavily utilized, reducing long tail response times [6], [24].

**G. Evolutionary Reinforcement Learning**

In this article, we adapt ERL [40] to train the neural networks of HGraphScale. ERL is a population-based approach to estimate the gradients of neural networks. Algorithm 2 presents the pseudo-code of the ERL.

**Algorithm 2** Evolutionary Reinforcement Learning (ERL)

---

**Input:** Population size:  $N$ , maximum generation:  $max\_gen$ , initial policy parameters:  $\hat{\theta}$ , learning rate:  $\eta$ , multi-variance gaussian noise standard deviation:  $\sigma$   
**Output:** Trained neural network

```

1:  $gen \leftarrow 0$ 
2: while  $gen \leq max\_gen$  do
3:   for  $i = 0$  to  $N$  do
4:     Sample perturbation  $\epsilon_i \sim \mathcal{N}(0, 1)$ 
5:     Update the neural network by using  $\theta_i \leftarrow \hat{\theta} + \sigma \epsilon_i$ 
6:     Calculate Fitness  $F(\theta_i)$  based on Eq. 9
7:   end for
8:   Estimate policy gradient  $\nabla_{\theta} \mathbb{E}_{\epsilon_i \sim \mathcal{N}(0, 1)} F(\hat{\theta} + \sigma \epsilon_i)$ 
9:    $\hat{\theta} \leftarrow \hat{\theta} + \sigma F(\hat{\theta} + \sigma \epsilon_i)$ 
10: end while

```

---

In particular, the CHGNN and scaling policy network of HGraphScale initial all trainable parameters  $\hat{\theta} = \{\vec{a}, W_p, W_v, W_c, \phi, \omega\}$ , randomly. Each iteration starts with sampling  $N$  perturbations  $[\epsilon_i]_{i=0,1,\dots,N}$  from standard gaussian distribution  $\mathcal{N}(0, 1)$  (line 4). Then, a population of  $N$  individuals  $[\theta_i]_{i=0,1,\dots,N}$  is generated by adding noise to  $\hat{\theta}$  (line 5).

The fitness of an individual  $\theta_i$  is evaluated based on the optimization objectives defined in Eq. (9) (line 6), which is calculated by

$$F(\theta_i) = Obj(T). \quad (16)$$

Then, the parameters of the policy network are updated by the estimated gradient, which is the expectation of individuals' fitness (line 8). Specifically, the gradient is estimated by

$$\begin{aligned} \nabla_{\theta} \mathbb{E}_{\epsilon \sim \mathcal{N}(0, 1)} F(\hat{\theta} + \sigma \epsilon) &= \frac{1}{\sigma} \nabla_{\theta} \mathbb{E}_{\epsilon \sim \mathcal{N}(0, 1)} [F(\hat{\theta} + \sigma \epsilon) \epsilon] \\ &\approx \frac{1}{N\sigma} \sum_{i=1}^N [F(\hat{\theta} + \sigma \epsilon_i) \epsilon_i]. \end{aligned} \quad (17)$$

Finally, the policy parameters are updated by gradient descent (line 9).

**V. PERFORMANCE EVALUATION**

In this section, we conduct comprehensive experiments to test the performance of our proposed HGraphScale. We first present the setup of experiments, the HGraphScale configuration and the competing approaches. Then, the experiment results are shown. Code of implementation, dataset and configuration are made publicly available<sup>1</sup>.

<sup>1</sup><https://github.com/sine-fandel/HGraphScale>



### A. Experiment Setup

All experiments are carried out in a simulator that implemented based on OpenAI Gymnasium [17]. The simulator models dynamic resource allocation across containers, VMs, and PMs, and reproduces fluctuating workload. It also simulates autoscaling behaviors with transient effects. The worst-case scenarios analysis in Appendix B enhances the fidelity of the simulator to real-world environments.

Three real-world traces of user requests, i.e., NASA<sup>2</sup>, Wiki<sup>3</sup> and Alibaba<sup>4</sup> are used to create workloads for our experiments. Fig. 7 illustrates the workload patterns over the 960-time-unit period (2 days) of NASA, Wiki and Alibaba, with each time unit representing a 3-minute interval. The workload trace patterns are shown in Fig. 7. The first 480 time units (one day) of workload from NASA or Wiki are extracted for training, while the remaining time units of workload are used for test [44]. In this article, a scaling action is made every 3 minutes, following [44].

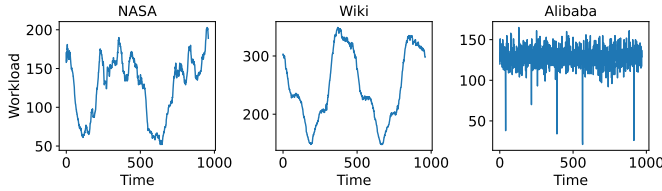


Fig. 7: Traces of user requests.

Four medium-scale microservice applications [23], [42], [43] and a large-scale microservice application [11] are used for our experiments, as summarized in Fig. 8. Each microservice application has a different number of microservices and application structures. For convenience, we denote them as “A11”, “A12”, “A13”, “A14” and “A30”, according to their microservices number.

Moreover, the cloud environment is equipped with 5 VM types from Amazon EC2<sup>5</sup>. The details of VM types are summarized in Tabel II. Each PM in the cloud environment has 64 vCPUs and 3200 GiB, following [46], [49].

TABLE II: Five VM types used in experiments

VM type	vCPU	Memory (GiB)	Hourly price (\$)
m5.xlarge	4	16	0.192
m5.2xlarge	8	32	0.384
m5.4xlarge	16	64	0.768
m5.8xlarge	32	128	1.536
m5.12xlarge	48	192	2.304

To sum up, there are 15 scenarios designed for experiments based on three real-world traces and five types of microservice applications. In the initial stage of each scenario, each microservice is instantiated with a container, allocated

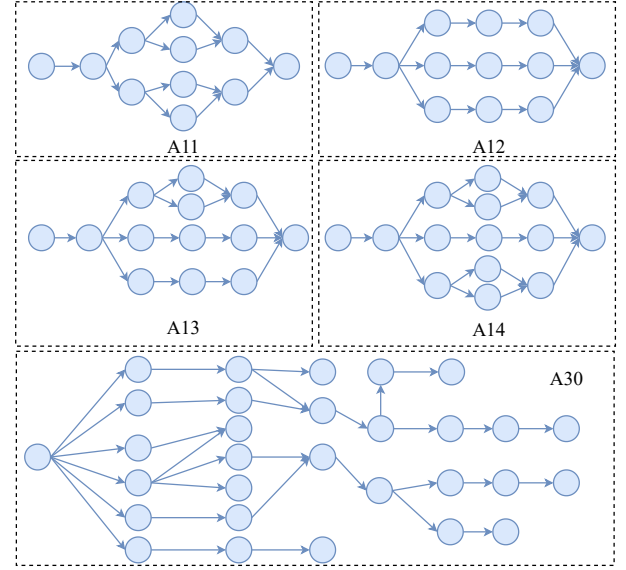


Fig. 8: Microservice applications used in experiments

with a vCPU and evenly deployed across three “m5.4xlarge” VMs [16], [46]. This setting allows each VM has enough remaining resources to support further vertical scaling.

### B. HGraphScale Configuration

This article sets the number of *graph attention layers* as: in container layer  $L_c = 2$ , in VM layer  $L_v = 1$  and PM layer  $L_p = 1$ , respectively. The dimension of GAT’s output is 64. The hidden dimension of each feedforward network is set as 64.

All the hyperparameter settings of the ERL follow existing studies [25] that are designed for practical application. Specifically, we set the population size of the ERL as 40. The maximum generation is set as 1000, while HGraphScale converges at about 400 generations in all scenarios. The learning rate  $\eta$  and the Gaussian noise standard deviation of ERL  $\sigma$  are set as 0.01 and 0.05, respectively. The parameters are updated by Adam Optimizer. The *budget*( $T$ ) of the optimization objective Eq. 9 is set as 200 USD per day [10], [57], while the performances under different budgets are evaluated in Section V-E. The penalty  $\rho$  is set as 100, following [44]. The performances of HGraphScale under different penalty settings are discussed in Section V-E4.

### C. Competing Approaches

HGraphScale is compared to two heuristic-based autoscaling approaches, two state-of-the-art DRL-based autoscaling approaches and a GNN-based autoscaling approaches. All competing approaches and HGraphScale share the same initial placement of containers. Moreover, they deploy newly created containers from horizontal scaling into suitable VMs using the Best-Fit heuristic [27]. With this heuristic, each new container is placed on the VM with the least remaining capacity that can still satisfy its demand. This strategy improves VM utilization and reduces the overall cost.

<sup>2</sup><http://ita.ee.lbl.gov/html/traces.html>

<sup>3</sup><http://www.wikibench.eu/wp-content/uploads/2010/10/vanbaaren-thesis.pdf>

<sup>4</sup><https://github.com/alibaba/clusterdata/tree/master/cluster-trace-microservices-v2021>

<sup>5</sup><https://aws.amazon.com/ec2/pricing/on-demand/>

TABLE III: Performance comparison in terms of ART and Vio, where Vio denotes the amount by which the cost exceeds the budget (200 USD).

Scenario	AWS-Scale		ProScale		DeepScale		DRPC		AGQ		HGraphScale	
	ART	Vio	ART	Vio	ART	Vio	ART	Vio	ART	Vio	ART	Vio
NASA-11	410.4	0.00	305.5	104.1	306.6 ± 9.8	0.00	289.9 ± 7.3	0.00	278.1 ± 11.6	0.00	<b>255.1 ± 6.4</b>	0.00
NASA-12	688.5	0.00	387.7	45.7	532.6 ± 10.8	0.00	433.2 ± 8.1	20.8 ± 1.5	538.6 ± 15.1	26.7 ± 8.1	<b>268.4 ± 11.2</b>	0.00
NASA-13	899.0	0.00	406.8	1.6	493.6 ± 7.9	88.1 ± 11.0	532.3 ± 5.0	0.00	243.8 ± 6.7	0.00	<b>178.3 ± 5.4</b>	0.00
NASA-14	1022.1	0.00	532.3	56.7	348.0 ± 20.1	66.17%	510.7 ± 11.3	3.3 ± 2.0	336.4 ± 22.7	323.1 ± 60.1	<b>325.6 ± 16.0</b>	0.00
NASA-30	491.3	0.00	<b>303.4</b>	140.2	407.2 ± 7.1	0.00	391.9 ± 10.5	0.00	474.4 ± 8.8	3.39%	389.4 ± 17.2	0.00
Wiki-11	489.7	0.00	532.4	0.00	318.2 ± 11.2	69.4 ± 8.6	415.4 ± 7.7	56.3 ± 4.3	361.7 ± 21.1	82.0 ± 10.3	<b>307.7 ± 15.4</b>	0.00
Wiki-12	864.6	0.00	687.0	0.00	549.9 ± 19.0	0.00	512.4 ± 27.5	25.0 ± 2.0	457.0 ± 22.1	72.8 ± 16.6	<b>424.3 ± 12.3</b>	0.00
Wiki-13	1080.4	0.00	482.1	26.3	675.3 ± 21.1	0.00	491.6 ± 27.2	112.3 ± 24.3	<b>367.6 ± 15.0</b>	27.7 ± 5.5	369.1 ± 23.1	0.00
Wiki-14	1022.1	0.00	532.3	22.7	348.0 ± 19.1	53.2 ± 14.5	510.7 ± 20.1	0.00	520.2 ± 24.0	22.5 ± 4.4	<b>325.6 ± 14.1</b>	0.00
Wiki-30	395.6	0.00	426.6	102.1	388.3 ± 55.7	112.2 ± 7.8	374.8 ± 62.1	0.00	488.2 ± 34.2	20.5 ± 6.9	<b>350.9 ± 33.0</b>	0.00
Alibaba-11	395.8	0.00	476.6	0.00	307.4 ± 24.1	0.00	249.6 ± 31.2	0.00	295.9 ± 27.6	0.00	<b>222.1 ± 26.6</b>	0.00
Alibaba-12	665.5	0.00	654.0	0.00	312.2 ± 22.3	132.4 ± 20.1	291.8 ± 24.1	0.00	292.1 ± 33.6	24.1 ± 4.7	<b>283.7 ± 17.8</b>	0.00
Alibaba-13	525.7	0.00	281.6	0.00	212.4 ± 19.2	111.4 ± 22.8	251.7 ± 27.9	0.00	237.9 ± 21.2	0.00	<b>178.9 ± 20.1</b>	0.00
Alibaba-14	988.7	0.00	549.3	0.00	327.0 ± 41.8	40.5 ± 6.0	<b>277.0 ± 30.3</b>	113.6 ± 38.0	421.8 ± 31.2	25.4 ± 2.6	299.2 ± 36.1	0.00
Alibaba-30	474.7	0.00	237.7	315.7	210.1 ± 22.7	25.5 ± 16.2	191.3 ± 23.7	68.2 ± 24.1	442.8 ± 26.1	0.00	<b>183.9 ± 17.7</b>	0.00

**AWS-Scale** [2] is a threshold-based autoscaling approach. Referring to [36], [44], we set the upper threshold as 0.8 and the lower threshold as 0.6 for CPU utilization of each container.

**ProScale** [11] is a heuristic-based proactive autoscaling method that leverages the SMA to predict future request workloads of each container. The horizontal scaling is made according to the predicted future workload.

**DeepScale** [44] is an autoscaling approach based on DQN. Specifically, it uses a deep neural network to make high-level decisions, i.e., increase, decrease and maintain the amount of resources provisioned to containers. Then, heuristics based on queue theory is proposed to make low-level scaling actions, including horizontal scaling and vertical scaling.

**DRPC** [4] is a distributed reinforcement learning approach for autoscaling. It first trains a central module using TD3. After training the central module, multiple deployment units are trained to imitate the central module’s behaviors. Deployment units make scaling actions (horizontal scaling and vertical scaling) for each microservice in a distributed manner.

**AGQ** [31] applies Graph Convolution Network (GCN) for resource estimation. The predicted future resource demand is utilized to make horizontal scaling decisions, i.e., increase replicas, reduce replicas and no operation. The resource adjustment agent is trained by Q-learning.

We conduct 10 independent runs for each approach. Within each run, all approaches are evaluated sharing the same random seed, while different runs use different seeds.

#### D. Performance Comparison

TABLE III presents the test results on each scenario, where the best performance of ART in each scenario is highlighted in **bold**. Specifically, HGraphScale decreases from 37.17% to 80.16% of ART when compared to threshold-based AWS-Scale. This is because a fixed threshold setting cannot adapt effectively to workload changes across time. In the NASA-30 scenario, HGraphScale performs 28.32% worse than ProScale in terms of ART. However, in this scenario, ProScale exceeds

the budget by 70.11%. In other 14 scenarios, HGraphScale achieved 16.51% to 56.16% less ART than ProScale.

When compared to DeepScale, HGraphScale produces 3.33% to 63.9% less ART. As for DRPC, HGraphScale produces 7.42% larger ART than DeepScale in Alibaba-14, while producing 3.33% to 63.87% less ART in the remaining scenarios. Although the ART of Alibaba-14 produced by DRPC is slightly better than HGraphScale, the corresponding cost exceeds the budget by 56.84%. Although AGQ is also a GNN-based autoscaling method, it only shows a slight advantage over HGraphScale in Wiki-13. However, in this case, AGQ also exceeds the budget by 13.85%.

TABLE III also presents the violation degree (“Vio”) that quantifies the percentage of cost exceeding the predefined budget (200 USD). We can observe from this table that the total VM rental cost of HGraphScale is always kept under the budget. This indicates that HGraphScale can make suitable scaling actions to avoid resource wastage. In contrast, ProScale, DeepScale, DRPC and AGQ exceed the budget in multiple scenarios. Although AWS-Scale also prevents budget violation by removing containers promptly when their CPU utilization is under the lower threshold. However, this design makes AWS-Scale vulnerable to QoS degradation under dynamic workloads. Container removal during low request periods leads to increased ART when user demand rises abruptly.

The observed performance improvements of HGraphScale can be attributed to our hierarchical representation and CHGNN design. These designs enable each container to effectively capture global information from VMs and PMs. This delivers more precise autoscaling decision than prior studies [2], [4], [11], [31], [44], which improves QoS while simultaneously reducing resource waste and cost.

#### E. Further Analysis

1) *Tail Response Time*: Besides the ART, the tail response time also provides insights into the QoS of microservice applications in the industry [4], [13], [44], [56]. Fig. 9 shows the maximum response times at different percentiles of user

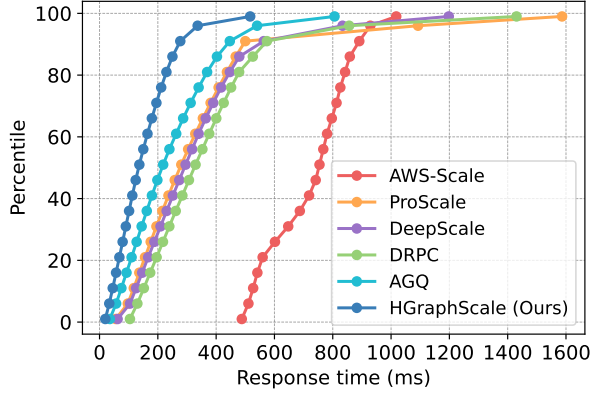


Fig. 9: Response times of NASA-13 at different percentiles for AWS-Scale, ProScale, DeepScale, DRPC and HGraphScale.

requests in NASA-13 (other scenarios have similar trends). We can see that HGraphScale achieves lower response times at all percentiles. Fig. 10 provides the details response time distribution of HGraphScale in NASA-13. The results show that HGraphScale ensures 95% of user requests are responded within 500 ms, and the maximum response time is 1.095s, showing stable performance and bounded worst-case latency. More details of response times analysis are provided in Appendix A.

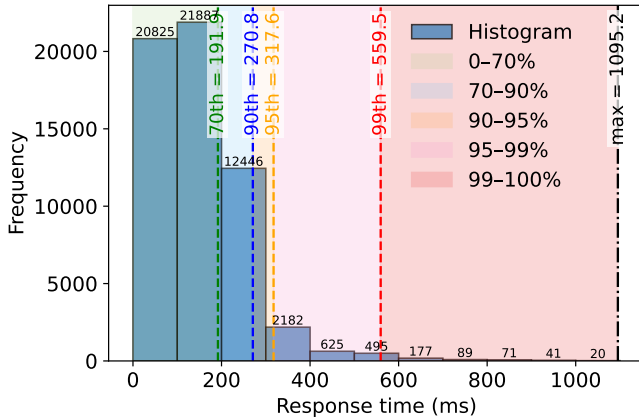


Fig. 10: Response time distribution of HGraphScale in NASA-13

2) *Ablation Studies*: To evaluate the effectiveness of the hierarchical graph learning, we conduct ablation studies by removing the PM layer of HGraphScale, giving rise to a variant named *w/o PM*. Moreover, we design another variant of HGraphScale without both VM and PM layers, named *w/o VM & PM*. HGraphScale is compared with *w/o PM* and *w/o VM & PM* on NASA-11, NASA-12, NASA-13 and NASA-14.

As shown in Fig. 11, both *w/o PM* and *w/o VM & PM* ensure the cost is not exceed the budget. However, *w/o PM* exhibits significantly inferior ART compared to HGraphScale, with *w/o VM & PM* performing even worse than *w/o PM*. These results indicate the effectiveness of both VM and PM embedding learning in HGraphScale.

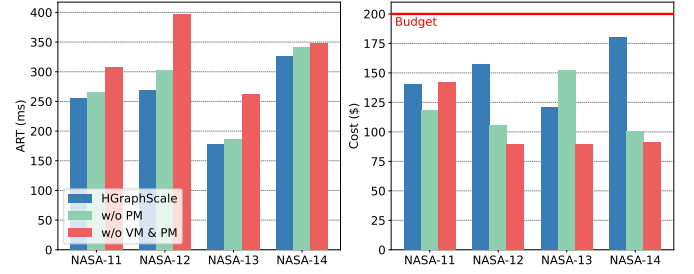


Fig. 11: The comparison results of ablation studies under NASA workload

TABLE IV: Performance Comparison With Different Budget: 150\$ and 250\$.

Scenario	150\$		250\$	
	ART (ms)	Cost (\$)	ART (ms)	Cost (\$)
NASA-11	237.4 ± 16.5	147.5 ± 15.1	219.7 ± 15.7	209.9 ± 23.6
NASA-12	271.7 ± 9.5	91.1 ± 5.1	245.0 ± 16.1	209.3 ± 11.5
NASA-13	430.8 ± 51.2	145.5 ± 21.1	162.9 ± 21.9	249.0 ± 12.7
NASA-14	408.9 ± 33.7	140.1 ± 12.2	349.8 ± 27.5	247.2 ± 15.5

3) *Performance Comparison with Different Budget*: We compare the performance of HGraphScale in solving the AMC problem with different cost budgets, that is, 150\$ and 250\$. TABLE IV presents the ART and cost under different budgets. Specifically, both the stringent and relaxed budgets of the AMC problem can be satisfied by HGraphScale. We observe that the HGraphScale achieves lower ART under 250\$ budgets than under 150\$ budgets. The reason is that a relaxed budget allows for the provision of more resources to the containers.

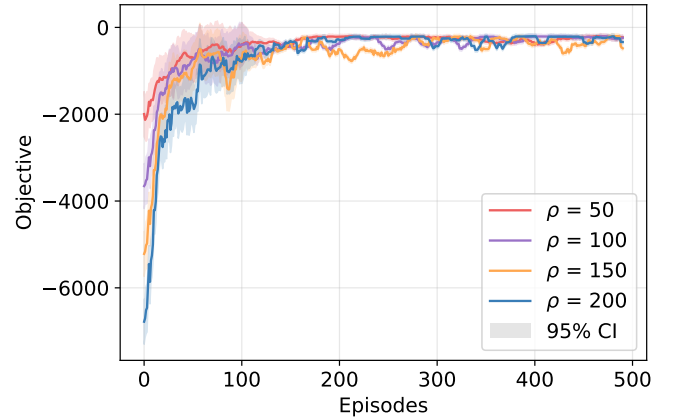


Fig. 12: Training curve of HGraphScale under different settings of penalty  $\rho$  on NASA-13

4) *Performance Comparison with Different Penalty*: The optimization objective of HGraphScale includes a penalty term ( $\rho$ ) for violating the budget. Therefore, we conduct sensitivity analysis on different penalty settings, that is  $\rho = 50$ ,  $\rho = 100$ ,  $\rho = 150$  and  $\rho = 200$ . Fig. 12 illustrates the training curves on NASA-13 obtained by different penalty settings. We can observe from this figure that the training process of HGraphScale is robust to different penalty settings, as they all achieve similar convergence stability.

TABLE V presents the test performance of HGraphScale. HGraphScale ensures the cost under 200\$/day with different settings of  $\rho$ . Moreover, when  $\rho = 50$ ,  $\rho = 100$ , and  $\rho = 150$ , HGraphScale achieves similar performances in terms of ART, while performance degradation occurs with  $\rho = 200$ . This is because the harsh penalty limits the exploration ability of HGraphScale during training.

TABLE V: Performance Comparison With Penalty Coefficient ( $\rho$ ).

$\rho$	NASA-11		NASA-13	
	ART (ms)	Cost (\$)	ART (ms)	Cost (\$)
50	273.5 $\pm$ 17.2	145.5 $\pm$ 9.3	169.4 $\pm$ 18.7	133.7 $\pm$ 20.2
100	255.1 $\pm$ 6.4	140.6 $\pm$ 7.2	178.3 $\pm$ 5.4	120.6 $\pm$ 11.9
150	256.4 $\pm$ 16.1	117.3 $\pm$ 8.8	180.1 $\pm$ 21.2	146.2 $\pm$ 15.5
200	324.0 $\pm$ 17.0	148.2 $\pm$ 8.2	203.6 $\pm$ 12.9	157.0 $\pm$ 5.5

5) *Quantitative Analysis of Scaling Actions*: To better understand the behavior of HGraphScale, we conduct a detailed analysis of quantitative breakdown of autoscaling actions. Fig. 13 demonstrates the frequencies of scaling actions generated by HGraphScale, including *vertical scaling*, *horizontal scaling*, and *no operation*. This figure provides evidences that HGraphScale tends to perform more vertical scaling than horizontal scaling in each scenario, resulting in fewer container replicas.

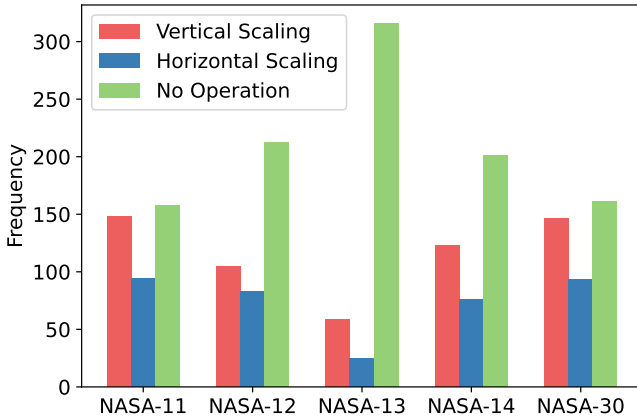


Fig. 13: Quantitative breakdown of HGraphScale's scaling actions.

Moreover, Fig. 13 also shows that *no operation* dominates in all scenarios. These results indicate that HGraphScale improves application performance while maintaining system stability without frequent scaling. It further demonstrates HGraphScale's ability to accurately identify containers requiring scaling and to determine appropriate scaling levels, thereby avoiding resource wastage.

## VI. CONCLUSION AND FUTURE WORK

In this article, we propose HGraphScale, a novel DRL-based autoscaling approach for microservice applications in container-based cloud. Particularly, We propose a hierarchical graph to capture dependencies in container-based clouds, a

CHGNN with bottom-up aggregation to learn container embeddings, and a scaling policy network that makes scaling decisions based on these embeddings. The experimental results indicate that HGraphScale reduces average response time compared to threshold-based, DRL-based, and graph-based autoscaling, without exceeding the cost budget. In future work, we will investigate multi-resource autoscaling to further enhance our method. Meanwhile, we will consider the container migration and respawning to further reduce cost by enabling more efficient VM utilization.

## REFERENCES

- [1] Al-Dhuraibi, Y., Paraiso, F., Djarallah, N., Merle, P.: Elasticity in cloud computing: state of the art and research challenges. *IEEE Transactions on services computing* **11**(2), 430–447 (2017)
- [2] Amazon: AWS Auto Scaling (2022), <https://aws.amazon.com/autoscaling/>, accessed: 2024-11-21
- [3] Baarzi, A.F., Kesidis, G.: Showar: Right-sizing and efficient scheduling of microservices. In: *Proceedings of the ACM Symposium on Cloud Computing*. pp. 427–441 (2021)
- [4] Bai, H., Xu, M., Ye, K., Buyya, R., Xu, C.: Drpc: Distributed reinforcement learning approach for scalable resource provisioning in container-based clusters. *IEEE Transactions on Services Computing* (2024)
- [5] Bao, L., Wu, C., Bu, X., Ren, N., Shen, M.: Performance modeling and workflow scheduling of microservice-based applications in clouds. *IEEE Transactions on Parallel and Distributed Systems* **30**(9), 2114–2129 (2019)
- [6] Baresi, L., Hu, D.Y.X., Quattrocchi, G., Terracciano, L.: KOSMOS: vertical and horizontal resource autoscaling for kubernetes. *ICSOC* **13121**, 821–829 (2021)
- [7] Berry, V., Castellort, A., Lange, B., Terihoania, J., Tibermacine, C., Trubiani, C.: Is it worth migrating a monolith to microservices? an experience report on performance, availability and energy usage. In: *2024 IEEE International Conference on Web Services (ICWS)*. pp. 944–954. IEEE (2024)
- [8] Blinowski, G., Ojdowska, A., Przybyłek, A.: Monolithic vs. microservice architecture: A performance and scalability evaluation. *IEEE Access* **10**, 20357–20374 (2022)
- [9] Burns, B., Beda, J., Hightower, K.: *Kubernetes: Up and Running: Dive into the Future of Infrastructure*. O'Reilly Media (2019)
- [10] Cheng, K., Zhang, S., Liu, M., Gu, Y., Wei, L., Cheng, H., Liu, K., Song, Y., Shi, X., Zhu, A., et al.: Geoscale: Microservice autoscaling with cost budget in geo-distributed edge clouds. *IEEE Transactions on Parallel and Distributed Systems* **35**(4), 646–662 (2024)
- [11] Cheng, K., Zhang, S., Tu, C., Shi, X., Yin, Z., Lu, S., Liang, Y., Gu, Q.: Proscale: Proactive autoscaling for microservice with time-varying workload at the edge. *IEEE Transactions on Parallel and Distributed Systems* **34**(4), 1294–1312 (2023)
- [12] Choularas, S., Sotiriadis, S.: An adaptive auto-scaling framework for cloud resource provisioning. *Future Generation Computer Systems* **148**, 173–183 (2023)
- [13] Dean, J., Barroso, L.A.: The tail at scale. *Communications of the ACM* **56**(2), 74–80 (2013)
- [14] Fang, Z., Ma, H., Chen, G., Hartmann, S.: Energy-efficient and communication-aware resource allocation in container-based cloud with group genetic algorithm. In: *International Conference on Service-Oriented Computing*. pp. 212–226. Springer (2023)
- [15] Fang, Z., Ma, H., Chen, G., Hartmann, S.: A group genetic algorithm for energy-efficient resource allocation in container-based clouds with heterogeneous physical machines. In: *Australasian Joint Conference on Artificial Intelligence*. pp. 453–465. Springer (2023)
- [16] Fang, Z., Ma, H., Chen, G., Hartmann, S., Chen, S.: A communication-aware and energy-efficient genetic programming based method for dynamic resource allocation in clouds. In: *International Conference on the Applications of Evolutionary Computation (Part of EvoStar)*. pp. 421–436. Springer (2025)
- [17] Foundation, F.: *Gymnasium documentation*. <https://gymnasium.farama.org/index.html>, accessed: 2025-09-16
- [18] Fujimoto, S., Hoof, H., Meger, D.: Addressing function approximation error in actor-critic methods. In: *International conference on machine learning*. pp. 1587–1596. PMLR (2018)



- [19] Garí, Y., Monge, D.A., Mateos, C.: A q-learning approach for the autoscaling of scientific workflows in the cloud. *Future Generation Computer Systems* **127**, 168–180 (2022)
- [20] Garí, Y., Pacini, E., Robino, L., Mateos, C., Monge, D.A.: Online rl-based cloud autoscaling for scientific workflows: Evaluation of q-learning and sarsa. *Future Generation Computer Systems* **157**, 573–586 (2024)
- [21] Hamzaoui, I., Duthil, B., Courboulay, V., Medromi, H.: A topical review on container-based cloud revolution: Multi-directional challenges, and future trends. *SN Computer Science* **5**(4), 416 (2024)
- [22] He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. pp. 770–778 (2016)
- [23] Huang, K.C., Shen, B.J.: Service deployment strategies for efficient execution of composite saas applications on cloud platform. *Journal of Systems and Software* **107**, 127–141 (2015)
- [24] Huang, V., Chen, G., Zhang, P., Li, H., Hu, C., Pan, T., Fu, Q.: A scalable approach to sdn control plane management: High utilization comes with low latency. *IEEE Transactions on Network and Service Management* **17**(2), 682–695 (2020)
- [25] Huang, V., Wang, C., Ma, H., Chen, G., Christopher, K.: Cost-aware dynamic multi-workflow scheduling in cloud data center using evolutionary reinforcement learning. In: *International Conference on Service-Oriented Computing*. pp. 449–464. Springer (2022)
- [26] Imdoukh, M., Ahmad, I., Alfaiakawi, M.G.: Machine learning-based auto-scaling for containerized applications. *Neural Computing and Applications* **32**(13), 9745–9760 (2020)
- [27] Jangiti, S., Vijayakumar, V., Subramaniaswamy, V.: Hybrid best-fit heuristic for energy efficient virtual machine placement in cloud data centers. *EAI Endorsed Transactions on Energy Web* **7**(26) (2020)
- [28] Jeong, B., Jeong, Y.S.: Autoscaling techniques in cloud-native computing: A comprehensive survey. *Computer Science Review* **58**, 100791 (2025)
- [29] Lee, S., Park, J.: Comparative performance analysis of i/o interfaces on different nvme ssds in a high cpu contention scenario. *IEEE TRANSACTIONS ON Information and Systems* **107**(7), 898–900 (2024)
- [30] Li, T., Ying, S., Tian, X., Zhang, T., Wang, Y.: Astra: Adversarial sim-to-real transfer reinforcement learning for autoscaling in cloud systems. *IEEE Transactions on Software Engineering* (2025)
- [31] Liang, P., Xun, Y., Cai, J., Yang, H.: Autoscaling of microservice resources based on dense connectivity spatio-temporal gnn and q-learning. *Future Generation Computer Systems* **174**, 107909 (2026)
- [32] Ma, Y., Gerard, P., Tian, Y., Guo, Z., Chawla, N.V.: Hierarchical spatio-temporal graph neural networks for pandemic forecasting. In: *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*. pp. 1481–1490 (2022)
- [33] Meng, C., Song, S., Tong, H., Pan, M., Yu, Y.: Deepscaler: Holistic autoscaling for microservices based on spatiotemporal gnn with adaptive graph learning. In: *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. pp. 53–65. IEEE (2023)
- [34] Meng, C., Tong, J., Pan, M., Yu, Y.: Hra: An intelligent holistic resource autoscaling framework for multi-service applications. In: *2022 IEEE International Conference on Web Services (ICWS)*. pp. 129–139. IEEE (2022)
- [35] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fiedelnd, A.K., Ostrovski, G., et al.: Human-level control through deep reinforcement learning. *nature* **518**(7540), 529–533 (2015)
- [36] Nouri, S.M.R., Li, H., Venugopal, S., Guo, W., He, M., Tian, W.: Autonomic decentralized elasticity based on a reinforcement learning controller for cloud applications. *Future Generation Computer Systems* **94**, 765–780 (2019)
- [37] Park, J., Choi, B., Lee, C., Han, D.: Graf: A graph neural network based proactive resource allocation framework for slo-oriented microservices. In: *Proceedings of the 17th International Conference on emerging Networking EXperiments and Technologies*. pp. 154–167 (2021)
- [38] Park, J., Choi, B., Lee, C., Han, D.: Graph neural network-based slo-aware proactive resource autoscaling framework for microservices. *IEEE/ACM Transactions on Networking* **32**(4), 3331–3346 (2024)
- [39] Saidu, I., Subramaniam, S., Jaafar, A., Zukarnain, Z.A.: A load-aware weighted round-robin algorithm for ieee 802.16 networks. *EURASIP Journal on Wireless Communications and Networking* **2014**(1), 226 (2014)
- [40] Salimans, T., Ho, J., Chen, X., Sidor, S., Sutskever, I.: Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864* (2017)
- [41] Shen, Y., Chen, G., Ma, H., Zhang, M.: Cost-aware dynamic cloud workflow scheduling using self-attention and evolutionary reinforcement learning. In: *International Conference on Service-Oriented Computing*. pp. 3–18. Springer (2024)
- [42] Shi, T., Ma, H., Chen, G., Hartmann, S.: Location-aware and budget-constrained service deployment for composite applications in multi-cloud environment. *IEEE Transactions on Parallel and Distributed Systems* **31**(8), 1954–1969 (2020)
- [43] Shi, T., Ma, H., Chen, G., Hartmann, S.: Cost-effective web application replication and deployment in multi-cloud environment. *IEEE Transactions on Parallel and Distributed Systems* **33**(8), 1982–1995 (2021)
- [44] Shi, T., Ma, H., Chen, G., Hartmann, S.: Auto-scaling containerized applications in geo-distributed clouds. *IEEE Transactions on Services Computing* (2023)
- [45] Srirama, S.N., Adhikari, M., Paul, S.: Application deployment using containers with auto-scaling for microservices in cloud environment. *Journal of Network and Computer Applications* **160**, 102629 (2020)
- [46] Tan, B., Ma, H., Mei, Y., Zhang, M.: A cooperative coevolution genetic programming hyper-heuristics approach for on-line resource allocation in container-based clouds. *IEEE Transactions on Cloud Computing* **10**(3), 1500–1514 (2020)
- [47] Tong, G., Meng, C., Song, S., Pan, M., Yu, Y.: Gma: graph multi-agent microservice autoscaling algorithm in edge-cloud environment. In: *2023 IEEE international conference on web services (ICWS)*. pp. 393–404. IEEE (2023)
- [48] Velickovic, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., Bengio, Y., et al.: Graph attention networks. *stat* **1050**(20), 10–48550 (2017)
- [49] Wang, C., Ma, H., Chen, G., Huang, V., Yu, Y., Christopher, K.: Energy-aware dynamic resource allocation in container-based clouds via cooperative coevolution genetic programming. In: *International Conference on the Applications of Evolutionary Computation (Part of EvoStar)*. pp. 539–555. Springer (2023)
- [50] Wang, S., Ding, Z., Jiang, C.: Elastic scheduling for microservice applications in clouds. *IEEE Transactions on Parallel and Distributed Systems* **32**(1), 98–115 (2020)
- [51] Wang, S., Li, X., Sheng, Q.Z., Beheshti, A.: Performance analysis and optimization on scheduling stochastic cloud service requests: A survey. *IEEE Transactions on Network and Service Management* **19**(3), 3587–3602 (2022)
- [52] Watkins, C.J., Dayan, P.: Q-learning. *Machine learning* **8**, 279–292 (1992)
- [53] Wen, L., Xu, M., Gill, S.S., Hilman, M., Srirama, S.N., Ye, K., Xu, C.: Statusscale: Status-aware and elastic scaling strategy for microservice applications. *ACM Transactions on Autonomous and Adaptive Systems* **20**(1), 1–25 (2025)
- [54] Wen, Z., Chen, Q., Deng, Q., Niu, Y., Song, Z., Liu, F.: Combofunc: joint resource combination and container placement for serverless function scaling with heterogeneous container. *IEEE Transactions on Parallel and Distributed Systems* (2024)
- [55] Wu, N., Zhao, X.W., Wang, J., Pan, D.: Learning effective road network representation with hierarchical graph neural networks. In: *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*. pp. 6–14 (2020)
- [56] Xie, J., Guo, D., Li, X., Shen, Y., Jiang, X.: Cutting long-tail latency of routing response in software defined networks. *IEEE Journal on Selected Areas in Communications* **36**(3), 384–396 (2018)
- [57] Xie, S., Wang, J., Li, B., Zhang, Z., Li, D., Hung, P.C.: Pbscaler: A bottleneck-aware autoscaling framework for microservice-based applications. *IEEE Transactions on Services Computing* (2024)
- [58] Yang, Y., Chen, G., Ma, H., Zhang, M.: Dual-tree genetic programming for deadline-constrained dynamic workflow scheduling in cloud. In: *International Conference on Service-Oriented Computing*. pp. 433–448. Springer (2022)
- [59] Ying, Z., You, J., Morris, C., Ren, X., Hamilton, W., Leskovec, J.: Hierarchical graph representation learning with differentiable pooling. *Advances in neural information processing systems* **31** (2018)
- [60] Zhang, S., Wu, T., Pan, M., Zhang, C., Yu, Y.: A-sarsa: A predictive container auto-scaling algorithm based on reinforcement learning. In: *2020 IEEE international conference on web services (ICWS)*. pp. 489–497. IEEE (2020)
- [61] Zhong, Z., Li, C.T., Pang, J.: Hierarchical message-passing graph neural networks. *Data Mining and Knowledge Discovery* **37**(1), 381–408 (2023)
- [62] Zolfaghari, R.: Energy-performance aware virtual machines migration in cloud network by using prediction and fuzzy approaches. *Engineering Applications of Artificial Intelligence* **131**, 107825 (2024)