Auto-Scaling Containerized Applications in Geo-Distributed Clouds

Tao Shi, Hui Ma, Gang Chen, and Sven Hartmann

Abstract—As a lightweight and flexible infrastructure solution, containers have increasingly been used for application deployment on a global scale. By rapidly scaling containers at different locations, the deployed applications can handle dynamic workloads from the worldwide user community. Existing studies usually focus on the (dynamic) container scaling within a single data center or the (static) container deployment across geo-distributed data centers. This article studies an increasingly important container scaling problem for application deployment in geo-distributed clouds. Reinforcement learning (RL) has been widely used in container scaling due to its high adaptability and robustness. To handle high-dimensional state spaces in geo-distributed clouds, we propose a deep RL algorithm, named *DeepScale*, to auto-scale containerized applications. *DeepScale* innovatively utilizes multi-step predicted future workloads to train a holistic scaling policy. It features several newly designed algorithmic components, including a domain-tailored state constructor and a heuristic-based action executor. These new algorithmic components are essential to meet the requirements of low deployment costs and achieve desirable application performance. We conduct extensive simulation studies using real-world datasets. The results show that *DeepScale* can significantly outperform an industry-leading scaling strategy and two state-of-the-art baselines in terms of both cost-effectiveness and constraint satisfaction.

Index Terms—Auto-scaling, containerized application, geo-distributed clouds, workload management, safe reinforcement learning, time series analysis.

1 Introduction

Enterprise application deployment demands for elastically acquiring and releasing resources to handle dynamic workloads from the worldwide user community [1]. Currently, the elastic deployment and real-time management of applications increasingly rely on containers, an industry-leading lightweight virtualization technology [2], [3]. By bundling together an application with all its dependencies (e.g., libraries and code), the containerized application can realize fast deployment and migration in clouds [4].

Containers lay the technical foundation for elastic application deployment through *horizontal scaling* and *vertical scaling* [5]. Particularly, the horizontal scaling changes the number of containers for deployed application instances, i.e., containerized applications[1]. The vertical scaling changes the container configuration (i.e., the amount of provisioned resources) for application instances.

Since the horizontal scaling can be performed on containers in different cloud data centers, it has the advantage to address the workload changes across multiple geographical locations [6]. The vertical scaling can be realized with practically no downtime [7]. Therefore, it is favored for the localized workload variations [6].

Many recent efforts have been made in the literature for container scaling within a single data center [5], [8], [9], [10], [11]. To guarantee the application performance in terms of the average response time of global user requests [12], [13], [14], container scaling must work effectively across geo-distributed data centers. Besides, the pricing of containers in public clouds is based on the assigned resources, e.g., the number of vCPUs [15], [16]. Note that the prices of resources assigned to containers in different regions can vary substantially, as clearly evidenced in TABLE 1 for Amazon Web Services (AWS) across different regions. For example, the price of containers in Sao Paulo is $0.0696 per hour, while the prices in N.Virginia, Oregon, and Dublin are $0.04048 per hour. That is, the most expensive region can fetch up to 1.7 times the price of the cheapest regions. The significant impact of containers' locations on both the performance and cost of deployed applications must be considered when performing container scaling.

In the articles [17], [18], the problem of application deployment in geographically distributed data centers was studied in a *static* scenario to find the most cost-effective deployment while satisfying the performance requirement on the average response time. However, the location-aware application deployment problem does not consider the impact of current deployment decisions on the future workload variations, which is important when we minimize the cumulative cost for application deployment over a time span such as a billing day in practice.

In this paper, we study the *dynamic* problem of *location-aware container scaling (LACS)* in geo-distributed clouds from

- *Tao Shi is with the Science and Information College, Qingdao Agricultural University, China.*
  E-mail: shitao@qau.edu.cn
- *Hui Ma and Gang Chen are with the School of Engineering and Computer Science, Victoria University of Wellington, Wellington, New Zealand.*
  E-mail: {hui.ma, aaron.chen}@ecs.vuw.ac.nz
- *Sven Hartmann is with the Department of Informatics, Clausthal University of Technology, Germany.*
  E-mail: sven.hartmann@tu-clausthal.de

Manuscript received x x, x; revised x x, x.

1. In the remainder of this paper, we use application instances, containerized applications, and containers interchangeably.

TABLE 1
Unit cost (vCPU per hour) of containers at AWS across different regions

Continent	Region	Cost (USD)
North America	US East (N.Virginia)	$0.04048
	US East (Oregon)	$0.04048
	Montreal	$0.04456
South America	Sao Paulo	$0.0696
Asia	Singapore	$0.05056
	Tokyo	$0.05056
	Mumbai	$0.04256
	Hong Kong	$0.0556
	Osaka	$0.05056
	Seoul	$0.04656
Oceania	Sydney	$0.04856
Europe	Frankfurt	$0.04656
	Dublin	$0.04048
	London	$0.04656
	Milan	$0.0486
	Paris	$0.0486
	Stockholm	$0.0445
Africa	Cape Town	$0.0546

the perspective of application providers, to minimize the cumulative application deployment cost while satisfying the constraint on the average application response time.

In practice, enterprise application providers usually apply threshold-based rules to automate container scaling [19]. This strategy is efficient to make scaling decisions in real-time. However, manually choosing appropriate thresholds is difficult and often results in containers either over-utilized that slow down the request processing speed or under-utilized that waste money [20].

Reinforcement learning (RL) allows to express *what* an application provider aims to obtain, instead of *how* it should be obtained [5]. The nature of RL makes it very appealing to auto-scale containers for applications with dynamically changing and widely distributed workloads. For example, a deep Q-network (DQN) can be trained to approach an optimal policy for scaling containers in geo-distributed clouds to minimize the cumulative cost in the long run [21], [22], [23]. However, it is challenging to achieve both cost-effectiveness and constraint satisfaction by directly using DQN [24]. Therefore, new learning techniques must be developed to effectively solve the LACS problem.

Predictive strategies such as time series analysis have been used to improve the timeliness of container scaling within a single data center, which is important to achieve cost-effective application deployment [20]. In this paper, we seek to utilize prediction models to provide essential information on dynamic application workloads for DQN to make well-informed scaling decisions in geo-distributed clouds. Particularly, *multi-step* predicted future workloads are collected by a *domain-tailored state constructor* to ensure high cost-effectiveness.

The performance constraint of the LACS problem motivates us to adopt safe RL, which aims to learn a policy that maximizes the expected return while ensuring (or encouraging) the satisfaction of some safety constraints [25]. In this paper, we propose two safe RL approaches to increase the chance of satisfying the constraint on the average response time. On the one hand, we introduce a *penalty-based reward function* and a *heuristic-based action executor* to train *policies* towards the constraint-compliant scaling. On the other hand, the heuristics designed in the action executor can prevent scaling actions from prolonging the average response time beyond the acceptable level during the *learning* process, which has been verified in our experiments (Subsection 5.6).

In a nutshell, all the new algorithmic components jointly form a *holistic scaling policy* for the LACS problem. In particular, we show experimentally in Section 5 that the proposed domain-tailored state constructor and the heuristic-based action executor, are essential to meet the requirements of low deployment costs and achieve desirable application performance. The main contributions of this paper are summarized as follows.

Firstly, we identify and formulate the LACS problem to minimize the total deployment cost subject to the constraint on the average response time across widely distributed user communities. To the best of our knowledge, this is the first study in the literature on dynamic container scaling for application deployment with a realistic consideration of the location impact on both the performance and cost on the global scale.

Secondly, we propose a novel deep reinforcement learning (DRL)-based algorithm, namely *DeepScale*, to solve the LACS problem. *DeepScale* innovatively trains the holistic scaling policy. It features several newly designed algorithmic components, including a domain-tailored state constructor and a heuristic-based action executor. Our proposed multi-step predicted workloads, penalty-based reward function, and safety-aware heuristics are utilized by *DeepScale* to minimize the cost and satisfy the performance constraint of the LACS problem.

Finally, we evaluate the effectiveness of *DeepScale* through extensive simulation studies using realistic container pricing offered by AWS and workloads for cloud applications, i.e., WikiBench [26] and NASA [27]. The results show that *DeepScale* can consistently satisfy the constraint on the average response time and achieve up to 39% savings in terms of the deployment cost, compared to the Amazon auto-scaling service [19] and two state-of-the-art baselines, i.e., A-SARSA [10] and DQLCM [28].

The remainder of this paper is organized as follows. Section 2 discusses the related work about containerized application deployment in clouds and container scaling with RL. Section 3 defines the LACS problem, including the system architecture for containerized application deployment in geo-distributed clouds. Section 4 presents the details of the *DeepScale* algorithm. Section 5 describes the design of simulation studies and analyses the evaluation results. Section 6 discusses the scope and limitations of this research. Section 7 concludes the paper.

2 Related Work

This section introduces the related works about containerized application deployment in clouds and container scaling with RL. The main challenges that need to be addressed in this paper are also highlighted.

2.1 Containerized Application Deployment in Clouds

Currently, the deployment and real-time management of applications increasingly rely on containers, an industry-

leading lightweight virtualization technology [4], [6]. Significant efforts have been made for containerized application deployment in clouds. Commercial container management platforms, e.g., Rancher [29] and OpenShift [30], facilitate container deployment in public clouds. In addition to cloud management and visibility, they empower application providers with the ability to easily adapt the deployment of containers across data centers through a unified user interface or API. However, these platforms do not support auto-scaling containers for applications to quickly respond to workload fluctuations, which are essential for application providers to achieve low deployment cost and satisfactory application performance [17].

The problem of containerized application deployment in clouds has been studied at different resource levels: container deployment [5], [8], [31], cluster deployment [32], [9], or both [33]. These works considered horizontal scaling [8], [9], [32], vertical scaling [31], or both [5] depending on different elasticity dimensions. However, they all focus on auto-scaling techniques within a single data center to handle workload variations of applications in a cost-effective manner. That is, these solutions neglect container locations, which can have a significant impact on both the deployment cost and the network latency of applications.

Predictive auto-scaling utilizes a workload forecast to derive scaling actions. Google uses Autopilot, as a vertical and horizontal auto-scaler, to reduce resource waste and increase reliability [34]. Focusing on vertical scaling, self-adaptive resource sharing was investigated in the article [35] for co-located applications in saturated containerized clouds. In the article [36], a hybrid auto-scaling mechanism, called Chameleon, combining proactive methods with a reactive fallback mechanism was proposed to deploy business applications. In the article [37], several predictive-based auto-scaling policies were designed for microservice applications both on VMs and containers level. However, they also focus on auto-scaling techniques within a single data center to handle workload variations of applications. Considering the locations of containers, latency-aware auto-scalers, i.e., Voilà [38] and Hona [39], were proposed for application replica placement in fog computing platforms. In the article [6], ge-kube was proposed as an extension of Kubernetes to introduce self-adaptation and network-aware scheduling capabilities among 4 different regions. A multi-level elastic deployment model for containerized applications was proposed in the article [7]. The model was evaluated in a simulated geo-distributed environments through uniformly distributing network delay. To support container scaling globally, the existing algorithms such as the meta-heuristics in the article [17] were proposed to make deployment decisions based on the current workload. However, these algorithms suffer from the high computational cost. Moreover, the current deployment may not work well in the future due to the dynamic nature of application workloads.

2.2 Container Scaling with RL

Existing research showed that RL is effective at scaling virtual machines (VMs) to handle dynamic application workloads [21], [40], [41], [42]. Due to the large difference between containers and VMs in terms of start-up, shut-down, and migration times, these RL-based algorithms that perform well in VM scaling cannot be effectively applied to container scaling with high demand for timeliness and accuracy [10]. Recently, some RL-based algorithms have been proposed for scaling containers for application deployment. For example, Rossi et al. [5] proposed a model-based RL method to control the horizontal and vertical elasticity of containers. Because the future trend of workload is not considered, these approaches fail to deal with highly dynamic application workloads, which causes resource wastage or compromised quality of service (QoS) [10].

As a prediction technique, time series analysis has been applied to handle dynamic application workloads [43]. The resulting workload prediction models can be further utilized by RL methods to ensure the timeliness and accuracy of scaling actions, such as A-SARSA [10]. Note that A-SARSA only considers one-step prediction, which may not be sufficient for RL to make well-informed scaling decisions as shown experimentally in Subsection 5.5. Besides, A-SARSA focuses on auto-scaling containers within a single data center. Due to its use of the Q-table, A-SARSA cannot scale well to large LACS problems involving many geo-distributed data centers.

In recent years, deep Q-network (DQN) was applied to resource allocation problems with high-dimensional state spaces [21], [23], [28]. In the article [21], a DQN-based resource provisioning and task scheduling system was proposed to minimize the energy cost for cloud service providers. Further considering the thermal effect of job allocation, Yi et al. [23] applied DQN to allocate compute-intensive jobs within the boundary of a single data center. Tang et al. [28] proposed a DQN-based container migration algorithm, i.e., DQLCM, to support mobility tasks with various application requirements, including the communication delay, power consumption, and migration cost.

Traditional DRL methods assume that agents are free to any policy for exploration [44], [45], [46], [47], [48]. For the location-aware container scaling (LACS) problem, it is unacceptable to give an agent complete freedom. For example, some scaling actions may cause containers to be heavily utilized. In that case, the application performance will drastically deteriorate such that the constraint on the average response time cannot be satisfied. Therefore, safe exploration, i.e., to control the damage caused by exploration [49], is important for solving the LACS problem. To guarantee constraint satisfaction, some policy-search-based DRL algorithms were proposed recently. For example, constrained policy optimization (CPO) was proposed to train neural network policies for robotic control tasks [50]. However, CPO may not be suitable for the LACS problem because it requires the learning process to start from a feasible policy. It is difficult to manually identify such a feasible stochastic policy modeled as a deep neural network[2] for any LACS problem instance. Moreover, DQN is often shown to be more effective than the policy-search-based method [46], [48].

2. A stochastic policy must assign a probability to perform each individual action, i.e., the outputs of the neural network are probability values.

TABLE 2
Mathematical notations

Notation	Definition
t	The t^{th} time period
u	The u^{th} user center
$\gamma_u(t)$	Application request rate from u during t
$\omega(t)$	The application workload during t
$n(t)$	Total number of containers during t
d	The d^{th} data center
$A_d(t)$	The application instance deployed in d during t
UC_d	The unit cost of vCPUs for containers in d
$x_d(t)$	Number of vCPUs provisioned to $A_d(t)$ during t
$CPU(t)$	Deployment of application instances during t
$DC(t)$	Application deployment cost during t
$T_d^{arp}(t)$	Average request processing time of $A_d(t)$
$\lambda_d(t)$	Workload of $A_d(t)$ during t
$\mu_d(t)$	Capacity of $A_d(t)$ during t
$T_{u,d}^{rtd}$	Round-trip delay between user region u and data center d
$ART(t)$	Average response time of application during t
m	Maximum acceptable average response time
$\sigma_{u,d}(t)$	Percentage of requests from u to A_d during t
$u(t)$	Average CPU utilization rate of containers during t
$ANL(t)$	Average network latency between user regions and application instances during t

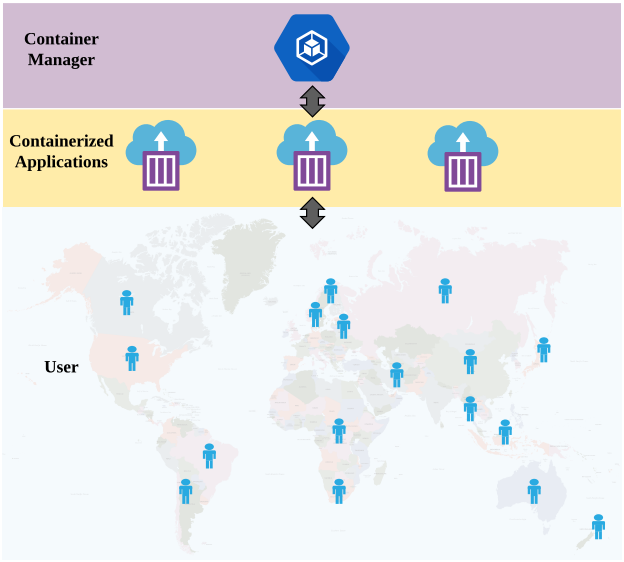


Fig. 1. A simplified system architecture of container scaling for application deployment in geo-distributed clouds based on [6].

The review above motivates us to develop a DQN-based algorithm to address the LACS problem. Moreover, to satisfy the constraint on the average response time, safe RL techniques, i.e., safe exploration, should be investigated.

3 Problem Description

The aim of the location-aware container scaling (LACS) problem is to scale containers globally for an application in response to a dynamically changing and widely distributed workload to minimize the total deployment cost (TDC) over a time span subject to the constraint on the average response time (ART). The response time is measured from the moment a user makes an application request to the moment when this user receives the corresponding response, taking into account both *request processing time* and *round-trip delay* (RTD) between the user and an application instance. The important notations for problem modeling are summarized in TABLE 2.

We consider container scaling for application deployment with three layers: container manager layer, application layer, and user layer, as shown in Fig. 1. In practice, an application involves a potentially large and dynamically changing number of requests from widely distributed users in global user regions $\mathcal{U}$. Suppose that the entire time span, e.g., a billing day, is divided into fixed-size execution periods. During time period t, we represent the workload from user region u ($u \in \mathcal{U}$) in terms of application request rate as $\gamma_u(t)$.

Multiple application instances can be deployed in parallel and each application instance independently processes a subset of the incoming requests [6]. Following [51], [52], the average resource consumption per request over a long sequence of requests is highly stable. Therefore, for an application instance, the variation of workload depends on the fluctuation of request rates from the corresponding user regions. Let $n(t)$ denote the number of application instances during time period t. Similar to [6], we adopt a hierarchical architecture to manage all application instances scalably, following the master-workers pattern [53].

We consider a set of geo-distributed data centers $\mathcal{D}$. In data center $d \in \mathcal{D}$, a collection of resources can be allocated to containers for application deployment. There are usually four main kinds of resources for containers in public clouds: CPU, memory, storage, and bandwidth [28]. Since the computing resource is the main factor, we assume that there is sufficient memory, storage and network capacity for the containerized application [54], [55]. During time period t we use $x_d(t)$ ($x_d(t) \in \mathbb{N}$) to denote the provisioned CPU, e.g., the number of vCPUs, to the container in data center d for application instance $A_d(t)$. Note that if there is not an application instance to be deployed in d, then $x_d(t) = 0$ holds. When an application instance is deployed in a single-vCPU container in d, then $x_d(t) = 1$ holds. The application deployment during t can be completely captured by the vCPU provision vector $CPU(t) = [x_d(t)]_{d\in\mathcal{D}}$, including the numbers of vCPUs provisioned in all cloud data centers.

Let UC_d denote the unit cost of vCPUs for containers in data center d. Based on the application deployment during t, i.e., $CPU(t)$, the deployment cost of all application instances during t can be calculated by:

$$DC(t) = \sum_{d\in\mathcal{D}} x_d(t) UC_d. \tag{1}$$

We use $\sigma_{u,d}(t) \in [0, 1]$ to denote the percentage of requests from user region u to application instance $A_d(t)$ during time period t. The *workload* of application instance $A_d(t)$ during t can be calculated by:

$$\lambda_d(t) = \sum_{u\in\mathcal{U}} \gamma_u(t)\sigma_{u,d}(t), \tag{2}$$

where $\sum_{d\in\mathcal{D}} \sigma_{u,d}(t) = 1$ guarantees that all application requests from user regions will be processed. The aggregated application workload from all user regions during t can be determined as:

$$\omega(t) = \sum_{u \in \mathcal{U}} \gamma_u(t). \quad (3)$$

Let $\mu_d(t)$ denote the *capacity* of $A_d(t)$, i.e., the maximum amount of requests processable by $A_d(t)$ per time unit. Following [56], we model each individual application instance as an $M/M/1$ queue. According to Little's Law [57], the average request processing time of $A_d(t)$ depends on both $\mu_d(t)$ and $\lambda_d(t)$:

$$T_d^{arp}(t) = \frac{1}{\mu_d(t) - \lambda_d(t)}, \quad (4)$$

where $\mu_d(t) > \lambda_d(t)$ guarantees that the capacity of $A_d(t)$ is greater than its workload.

Let $T_{u,d}^{rtd}$ denote RTD between user region u and data center d, we can calculate the average response time for all the user requests during time period t by:

$$ART(t) = \frac{\sum_{d \in \mathcal{D}} \sum_{u \in \mathcal{U}} \gamma_u(t) \sigma_{u,d}(t) (T_{u,d}^{rtd} + T_d^{arp}(t))}{\omega(t)}. \quad (5)$$

With the goal to minimize TDC, i.e., the cumulative deployment cost over the time span involving T consecutive periods, i.e., $t \in \{1, ..., T\}$, the LACS problem can be formulated as follows:

$$\min \quad TDC = \sum_{t=1}^{T} DC(t), \quad (6)$$

subject to:

$$ART(t) \leqslant m \ \ \forall t \in \{1, ..., T\}. \quad (7)$$

Constraint (7) guarantees that ART at any period over the entire time span does not exceed the acceptable threshold m set by the application provider. In the next section, we introduce our *DeepScale* algorithm to solve the LACS problem.

4 *DeepScale* FOR LACS

Our proposed algorithm *DeepScale* solves the location-aware container scaling (LACS) problem through auto-scaling containers both horizontally and vertically for application deployment in geo-distributed clouds. In this section, we start with a high-level overview of *DeepScale* and then describe the details on how *DeepScale* realizes container scaling by a deep reinforcement learning (DRL)-based policy.

4.1 Overview of *DeepScale*

To improve the effectiveness of the scaling actions, we include a workload prediction model in *DeepScale* to accurately predict future workloads of cloud applications [20]. *DeepScale* realizes container scaling by a DRL-based policy. Taking the predicted workloads and monitored container status as input, the scaling policy decides when and what scaling actions are performed to minimize the total deployment cost (TDC) subject to the constraint on the average response time (ART). Fig. 2 illustrates the workflow of *DeepScale*.

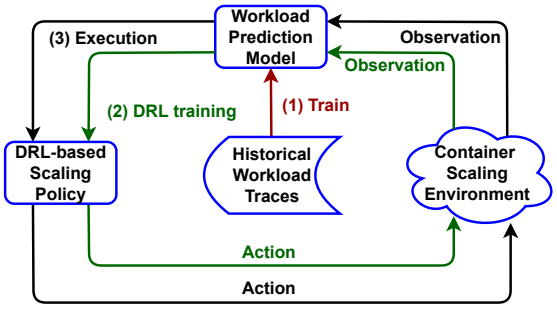


Fig. 2. Workflow of *DeepScale*.

Concretely, *DeepScale* first trains the workload prediction model based on historical workload traces. Different time series prediction methods, e.g., long short-term memory (LSTM) [58], can be used to built the workload prediction model. Afterwards, a DRL-based scaling policy is trained by utilizing the predicted future workloads. Finally, the trained prediction model and scaling policy can be commissioned to scale containers for serving incoming application requests. Before each time period, *DeepScale* first predicts the future application workloads using the prediction model. Based on the predicted workloads and the container status (e.g., resource utilization [6]), *DeepScale* makes container scaling decisions using the scaling policy.

Because different containers may have largely different capacities in terms of vCPU numbers, *DeepScale* applies capacity-based weighted Round-Robin (CWRR) [59] to dispatch requests among all application instances. CWRR can be implemented in the load balancing modules offered by cloud providers, e.g., AWS[3]. With CWRR, the percentage of requests from user region u to application instance $A_d(t)$, i.e., $\sigma_{u,d}(t)$, is determined by:

$$\sigma_{u,d}(t) = \frac{x_d(t)}{\sum_{d \in \mathcal{D}} x_d(t)}. \quad (8)$$

The rationale of CWRR-based request dispatching is two-fold. On the one hand, CWRR is commonly used in practice due to its simplicity and low computational cost [60]. On the other hand, CWRR can achieve effective load balancing by dispatching more user requests to the application instances with larger capacities [59]. Thus it can prevent any application instance from being heavily utilized, thereby reducing the risk of long queuing time. Next, we describe the details on how *DeepScale* trains the DRL-based scaling policy.

We define a DRL-based scaling system for the LACS problem as follows.

- **Observation:** The agent observation includes the information about the current containers for application deployment and future application workloads.
- **Action:** To perform scaling actions, i.e., to adjust the number of containers among data centers (horizontal

3. https://aws.amazon.com/elasticloadbalancing/application-load-balancer. Accessed: May. 24, 2023.

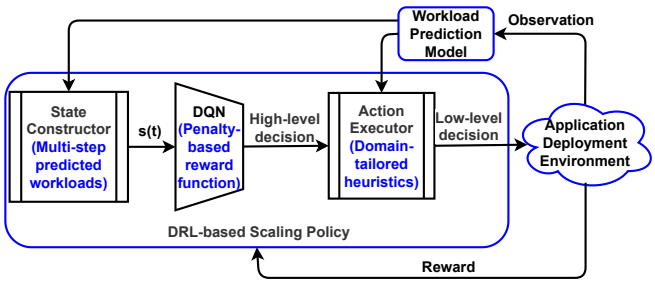


Fig. 3. Training DRL-based Scaling Policy.

scaling) and/or the number of vCPUs provisioned to current containers (vertical scaling).

The DRL design challenge is to effectively minimize TDC while handling constraint (7). Firstly, we introduce a *State Constructor* to collect multi-step future workloads using the workload prediction model. The essential information can help deep Q-network (*DQN*) to make well-informed scaling decisions. Secondly, the predicted workload is explicitly used by a heuristic-based *Action Executor* to ensure that any scaling decisions made by *DQN* will not prolong ART beyond the acceptable level. Finally, we propose a penalty-based reward function to guide constraint-aware Q-learning. The DRL-based scaling policy is shown in Fig. 3, which is composed of three components, i.e., a *State Constructor*, a *DQN*, and an *Action Executor*. In the following, we provide a detailed description of each component.

4.2 State Constructor

Three types of observation information are collected by the *State Constructor*.

The current deployment of application instances: We consider the current vCPU provision for application instances, i.e., $CPU(t)$, as the state feature because it decides the deployment cost during t, i.e., $DC(t)$, by eq. (1) and significantly affects application average response time $ART(t)$ by eq. (4), eq. (8), and eq. (5).

The current resource utilization: The current containers' CPU utilization rate is considered because it has a major impact on the application average response time [18]. With the help of geo-kube [6] for geo-distributed and elastic deployment of containers in Kubernetes, we can periodically monitor the CPU utilization rate of containers through RESTful APIs, e.g., the Metrics API in Kubernetes[4]. Because the CWRR-based request dispatching can approach the load balancing among all containers, we use the average utilization rate $u(t)$ as the state feature:

$$u(t) = \frac{\sum_{d \in \{i | x_i(t) > 0\}} u_d(t)}{n(t)}, \tag{9}$$

where $u_d(t)$ is the monitored average CPU utilization rate of application instance $A_d(t)$ during t.

The multi-step future workloads: To make well-informed scaling decisions in terms of cost-effectiveness, the *State Constructor* considers multi-step future workloads. Particularly, the *State Constructor* utilizes the trained workload prediction model to acquire the multi-step future workloads by a recursive strategy [20]. That is, the predicted workload is recursively used to predict the next workload. For example, after $w(t+1)$ is predicted, it is considered as input of the workload prediction model to predict the next workload $w(t+2)$. Similarly, $w(t+2)$ is used to predict $w(t+3)$, etc. We consider the difference between the predicted workloads in the future f time periods and the current workload, i.e., $\Delta w(t) = [w(t+1) - w(t), w(t+2) - w(t), ..., w(t+f) - w(t)]$, as the state feature. Here, we use the workload variations based on the aggregated request rate rather than the request rate from each user region since this is more relevant for the *DQN* to learn to make high-level scaling decisions, e.g., change the *total* number of vCPUs provisioned to containers. We will detail the high-level scaling decisions in the next subsection.

To sum up, the output of the *State Constructor* is $s(t) = [CPU(t), u(t), \Delta w(t)]$, which describes the current vCPU provision, average CPU utilization rate, and the predicted workload variations.

4.3 DRL for Training DQN

When performing both vertical and horizontal scaling in geo-distributed clouds, the number of potential scaling actions is indeterminate. For example, the possible actions for horizontal scaling depend on the locations of current application instances, not to mention the combination with vertical scaling. Therefore, it is infeasible to design a fixed-size action space in advance. We apply a novel method to let the *DQN* make *high-level* scaling decisions, i.e., change the total number of vCPUs provisioned to containers, and introduce a heuristic-based *Action Executor* to make *low-level* scaling decisions for concrete horizontal and vertical scaling actions. Formally, we define the action space of the high-level scaling decisions as $\mathcal{A} = \{\uparrow, \downarrow, \rightarrow\}$, where $\uparrow$, $\downarrow$, and $\rightarrow$ represent to increase (*scale-up*), decrease (*scale-down*), or *maintain* vCPU provision respectively.

We apply double Q-learning to learn the optimal Q-function. The DRL-based scaling policy applies the *DQN* in Fig. 3 as the function approximator. Following many existing research works [23], [61], we use experience replay [46] to stabilize Q-learning. To guide DRL to minimize TDC over the time span subject to the constraint on ART, we design a new reward function for DRL as follow:

$$r(s(t), a(t)) = -DC(t) - \max(0, \rho(ART(t) - m)), \tag{10}$$

where ρ is a penalty parameter for the scaling actions that cannot satisfy the performance requirement.

The detailed procedures for training the *DQN* are shown in Algorithm 1. At each time period t of an execution episode, based on the current $s(t)$, the DRL agent uses ϵ-greedy policy to choose an action randomly with probability ϵ and select the action with the maximum Q-value with probability $1 - \epsilon$ (step 4). The chosen action, i.e., $a(t)$, is performed through the *Action Executor* (step 5). At the next time period $t+1$, the DRL agent obtains $s(t+1)$, calculates the reward $r(s(t), a(t))$ defined in eq. (10) (step 6), and

4. https://kubernetes.io/docs/tasks/debug-application-cluster/resource-metrics-pipeline. Accessed: May. 24, 2023.

Algorithm 1 Training the *DQN*.

Initialize: Experience replay memory $\mathcal{D}$
Output: DQN parameters
1: **for** $episode = 1$ *to* $Max\_Episode$ **do**
2: **for** $t = 1$ *to* T **do**
3: Obtain $s(t)$ from the *State Constructor*
4: With probability ϵ select a random action, otherwise select an action with the maximum Q-value
5: Perform container scaling using the chosen action $a(t)$ through the *Action Executor*
6: Obtain $s(t+1)$, calculate $r(s(t), a(t))$ based on eq. (10)
7: Store transition $(s(t), a(t), r(s(t), a(t)), s(t+1))$ in $\mathcal{D}$;
8: Update Q-value
9: **end for**
10: Update DQN parameters using new Q-value estimates
11: **end for**

subsequently updates the Q-value (step 8). At the end of the execution episode, the DRL agent trains all the connection weight parameters of the *DQN*. The trained *DQN* will be utilized in the next execution episode (step 10).

4.4 Action Executor

Obeying the high-level scaling decisions from the *DQN*, i.e., to scale-up, scale-down, or maintain the total vCPU number, we design an *Action Executor* to make low-level scaling decisions. For example, if $a(t) = \uparrow$ is made by the *DQN*, the proposed *Action Executor* may add additional units of vCPU to existing containers and/or launch new containers in appropriate data centers. To closely follow large workload changes, three safety-aware heuristics are proposed to quickly change the deployed capacity of application instances. Note that the predicted application workload is also considered by the *Action Executor* to ensure that any low-level scaling decisions will not prolong the average response time beyond the acceptable level.

4.4.1 Scale-up

When a high-level decision of scale-up is made, i.e., $a(t) = \uparrow$, the *Action Executor* will iteratively choose the data center with the highest *scale-up benefit* to increase one vCPU unit. Particularly, the benefit of data center d in terms of performance improvement and cost-saving is estimated by:

$$benefit_d^+ = \frac{ANL(t) - ANL_d^+(t)}{DC_d^+(t) - DC(t)}, \tag{11}$$

where $ANL_d^+(t)$ and $DC_d^+(t)$ are the new average network latency and deployment cost provided that one vCPU unit is added in d. Concretely, the average network latency can be calculated by:

$$ANL(t) = \frac{\sum_{d\in\mathcal{D}}\sum_{u\in\mathcal{U}}\gamma_u(t)\sigma_{u,d}(t)T_{u,d}^{rtd}}{\omega(t)}. \tag{12}$$

Note that if there exists a container in data center d, the benefit of d is estimated provided that one vCPU unit is added to this container (vertical scaling). Otherwise, the benefit is estimated provided that a new container with one vCPU unit is launched in d (horizontal scaling). Next, we introduce a *safety-aware mechanism* to allocate a sufficient number of vCPUs in one scaling action. After increasing one vCPU unit, the new average CPU utilization rate can be estimated by:

$$u'(t) = \frac{w(t+1)}{\sum_{d\in\{i|x_i(t)>0\}}\mu'_d(t)}, \tag{13}$$

where $w(t+1)$ is the workload during the next time period predicted by the workload prediction model and $\mu'_d(t)$ denotes the new capacity of $A_d(t)$. If $u'(t) \geqslant 1$, one more vCPU unit is added following the above process. Otherwise, the *Action Executor* stops scaling up vCPUs. The safety-aware mechanism aims to handle the situation when there is a surge in workload. Finally, user requests are re-dispatched based on VWRR for load balancing. Algorithm 2 describes the overall procedure of scale-up vCPUs by the proposed *Action Executor*.

Algorithm 2 *Action Executor* performs scale-up.

Input: High-level decisions to scale-up vCPUs.
Output: Low-level scaling actions.
1: $Termination \leftarrow False$
2: **while** $Termination = False$ **do**
3: Decide the data center d with the highest $benefit_d^+$ evaluated by eq. (11) provided that one vCPU unit is added to the container or launch a new container with one vCPU unit
4: Evaluate the new average CPU utilization rate $u'(t)$ based on eq. (13)
5: **if** $u'(t) < 1$ **then**
6: $Termination \leftarrow True$
7: **end if**
8: **end while**
9: Re-dispatch requests based on CWRR

4.4.2 Scale-down

When a high-level decision of scale-down is made, i.e., $a(t) = \downarrow$, the *Action Executor* will attempt to reduce one vCPU unit from the existing container. Note that in order to modify the CPU resources assigned to containers, Kubernetes recreates the corresponding Pod (in *one-container-per-Pod* model). In practice, *connection draining* can be applied to ensure that all the requests are served. Through connection draining, the existing connection to application instances can be kept alive to complete in-flight requests during the process of starting up new containers. The mechanism is offered by many cloud providers, e.g., AWS [62]. Here, we introduce another *safety-aware mechanism* to avoid the potential deterioration of application performance. Particularly, the *Action Executor* first estimates the average CPU utilization rate $u'(t)$ after reducing one vCPU unit from the current containers defined eq. (13). If $u'(t) < 1$, the *Action Executor* reduces the vCPU unit from the existing container

Algorithm 3 *Action Executor* performs scale-down.

Input: High-level decisions to scale-down vCPUs.
Output: Low-level scaling actions.

```
Termination ← False
while Termination = False do
    Evaluate the new average CPU utilization rate u'(t) if one vCPU unit is reduced based on eq. (13)
    if u'(t) < 1 then
        Decide the data center d with the highest benefit_d^- evaluated by eq. (14) provided that one vCPU unit is reduced from the container
    else
        Termination ← True
    end if
end while
Re-dispatch requests based on CWRR
```

in the data center c with the largest *scale-down benefit*, which is estimated by:

$$benefit_d^- = \frac{DC(t) - DC_d^-(t)}{ANL_d^-(t) - ANL(t)}, \tag{14}$$

where $ANL_d^-(t)$ and $DC_d^-(t)$ are the new average network latency and deployment cost after decreasing one vCPU unit in d. In case that $u'(t) \geqslant 1$, the *Action Executor* aborts decreasing vCPUs to prevent containers from being heavily utilized and potentially resulting in a substantially prolonged response delay. Algorithm 3 presents the overall process to scale-down vCPUs by our proposed *Action Executor*.

4.4.3 Maintain

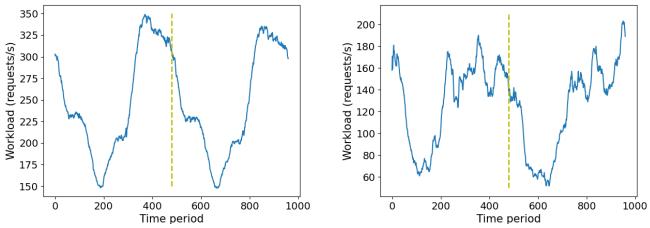
When a maintain decision is made, i.e., $a(t) = \rightarrow$, the *Action Executor* will attempt to reduce deployment cost and improve application performance by reconfiguring the current containers. Particularly, the *Action Executor* first determines the data center with the largest $benefit_d^+$ defined in eq. (11) for increasing one vCPU unit. Then the *Action Executor* determines the data center with the largest $benefit_d^-$ defined in eq. (14) for decreasing one vCPU unit. Let $ANL'(t)$ and $DC'(t)$ denote the new average network latency and deployment cost after the reconfiguration. In many cases, we have $ANL'(t) < ANL(t) \wedge DC'(t) > DC(t)$ or $DC'(t) < DC(t) \wedge ANL'(t) > ANL(t)$. The *Action Executor* only performs the low-level scaling when $ANL'(t) < ANL(t) \wedge DC'(t) < DC(t)$ to avoid reconfiguring the current containers too frequently.

In summary, as shown in Fig. 3 our newly designed multi-step predicted workloads, penalty-based reward function, and safety-aware heuristics are integrated into the three components of the DRL-based scaling policy to realize both the cost minimization and constraint satisfaction of the LACS problem.

5 Performance Evaluation

In the absence of a publicly available global cloud testbed, we conduct a series of simulation studies to examine the performance of *DeepScale* using the real-world datasets. By using realistic container pricing and workloads for cloud applications, we compare the performance of *DeepScale* with several state-of-the-art baselines. The highlights are:

- DeepScale can consistently satisfy the constraint on the average response time (ART), while some baselines cannot in some problem instances.
- For different problem instances in terms of application types and workloads, *DeepScale* achieves up to 39% savings in terms of the total deployment cost (TDC).



(a) WikiBench (b) NASA HTTP

Fig. 4. Application worklaod

5.1 Datasets

We collect the real container pricing schemes in April 2023 from AWS [15]. 18 major Amazon data centers (see TABLE 1) have been included in the experiments. Referring to [14], we adopt 82 user regions from 35 countries on 6 continents in the Sprint IP Network[5] to simulate the global user community. To evaluate the network latency between users and deployed services, we use the observation information in the Sprint[6] IP backbone network databases [18].

We use real traces of user requests based on the public benchmark WikiBench [26] and NASA HTTP [27] to create workloads for our experiments. WikiBench is a Web hosting benchmark allowing the stress-test of systems designed to host Web applications. Following [63], our experimental workload contains 1% of all user requests issued to Wikipedia (in all languages). NASA HTTP contains HTTP requests to the NASA Kennedy Space Center WWW server in Florida. Refer to [6], the request rate of NASA HTTP is increased by 180 times to replay millions of requests daily. Referring to [64], [65], we apply Facebook subscribers statistics[7] to simulate the distribution of application requests among different user regions.

We randomly extract one day's workload from WikiBench and NASA HTTP for training and use the workload on the following day for testing. The duration of each time period, i.e., the time interval for making scaling decisions, is set to 3 minutes as in the articles [5], [6]. Fig. 4 depicts the request rate (i.e., number of requests per second) during the two days. That is, the left 480 time periods are used for

5. https://www.sprint.net/tools/network-maps. Accessed: May. 24, 2023.
6. https://www.sprint.net/tools/ip-network-performance. Accessed: May. 24, 2023.
7. https://worldpopulationreview.com/country-rankings/facebook-users-by-country. Accessed: May. 24, 2023.

training and the right 480 time periods are used for testing. From Fig. 4, we can observe that the historical workloads variations match well with future workloads changes on the WikiBench dataset. As a result, the prediction model can predict future workloads with high accuracy. In comparison, on the NASA HTTP dataset, future workload trend deviates significantly from the historical track, making this dataset much harder to handle by the prediction model. We are tasked to deploy 3 applications reported in the article [18]. We follow [14] closely to calibrate the queuing model, particularly the request rate distribution based on real-world business web applications reported in the article [66], in term of resource demands for the individual applications. The application processing time for a single request is approximately 10ms (*application 1*), 15ms (*application 2*), and 20ms (*application 3*), running on the container with one vCPU unit. To sum up, six problem instances are included in our experiments. For convenience, we denote them as Wiki *app-1*, Wiki *app-2*, Wiki *app-3*, NASA *app-1*, NASA *app-2*, NASA *app-3*, respectively.

5.2 Algorithm Implementation

We implement *DeepScale* using PyTorch [67] and build a novel simulator using openAI's gym environment [68]. Following the basic principles for cloud experiment reproducibility in the article [69], we have made the representative code and data publicly available at: https://zenodo.org/record/8166602. For the workload prediction model, we apply a long short-term memory (LSTM) neural network because it can accurately predict cloud application workloads in many existing research works and production systems [20], [70]. We adopt the same parameter settings as [20] for the LSTM-based prediction model: a hidden layer with 20 LSTM units and the number of future time periods to be predicted $f = 5$. We use root-mean-square error (RMSE) as the loss function and Adam [71] as the optimizer for training the LSTM neural network. In our experiments, the LSTM neural network can converge within 100 episodes and provide good performance in predicting future workloads (see Subsection 5.6).

In our implementation, the *DQN* has two fully-connected hidden layers, each with 64 nodes. The input and hidden layers use rectified linear units (ReLUs). We also apply Adam as the optimizer. The initial and minimum ϵ, i.e., the probability that DRL randomly chooses an action (in step 4 of Algorithm 1), are set as 0.2 and 0.01, respectively [23]. Following [23], other algorithm settings of the *DQN* include: learning rate α is 0.001, and the mini-batch size is 32. The *DQN* is trained for 200 episodes because it always converges within 200 episodes. Refer to [12], the acceptable threshold of ART, i.e., m in constraint (7), is set to 150 ms, since a latency up to 200ms will deteriorate the user experience significantly [72].

5.3 Baselines

To evaluate the performance of *DeepScale*, we further implement an industry scaling strategy and two state-of-the-art baselines in our experiments.

Amazon auto-scaling service [19] provides many container scaling methods to handle the increasing or decreasing workload of an application. Referring to [73], we apply

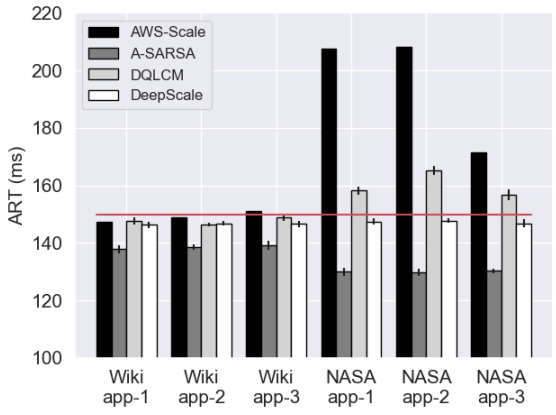


Fig. 5. ART of problem instances.

the rule-based auto-scaling method by setting an upper threshold (0.8) and a lower threshold (0.6) on the CPU utilization rate of containers. For convenience, we denote this baseline algorithm as *AWS-Scale*. Concretely, if the CPU utilization rate is above the upper threshold, *AWS-Scale* will scale-up the system. In case the CPU utilization rate is below the lower threshold, *AWS-Scale* will scale-down the system. To realize the location-aware container scaling, Algorithm 2 and Algorithm 3 are used in *AWS-Scale* to make low-level scaling decisions according to the threshold-based scale-up and scale-down decisions.

A-SARSA [10] is a recently proposed RL algorithm for auto-scaling containers. *A-SARSA* first combines ARIMA and a feedforward neural network to predict the CPU utilization rate and response time. Then the two predicted values are respectively discretized into different levels. Finally, a Q-table is trained by SARSA to make scaling decisions. For a fair comparison, we have fine-tuned the ARIMA model to achieve highly competitive accuracy for workload prediction with our LSTM-based model. Concretely, the RMSE of the predicted workload $w(t+1)$ on the testing day are 1.86 and 2.02 requests/s respectively (1.78 and 2.05 requests/s for our LSTM-based model detailed in Subsection 5.6). To control the constraint on the response time, *A-SARSA* applies a penalty-based reward function. Because *A-SARSA* only considers the container scaling within a single cloud data center, we also allow *A-SARSA* to use low-level scaling decisions made by our *Action Executor*.

The deep Q-learning container migration algorithm (*DQLCM*) [28] is proposed for delay-sensitive applications in fog computing. To select an appropriate action, *DQLCM* introduces problem-specific strategies for container migration. Particularly, two thresholds of CPU utilization rate, i.e., th_{under} and th_{over}, are predefined to classify fog nodes into different groups, i.e., under-utilized nodes and over-utilized nodes. For the nodes in different groups, different heuristics are proposed to determine the migrated containers and their destination. The action set of *DQLCM* is determined as optional container placement generated by these heuristics. To apply *DQLCM* to the LACS problem, we treat container migration as container scaling and fog nodes as application instances at different locations. th_{under} and th_{over} are set as 0.5 and 0.9 respectively because the combination of

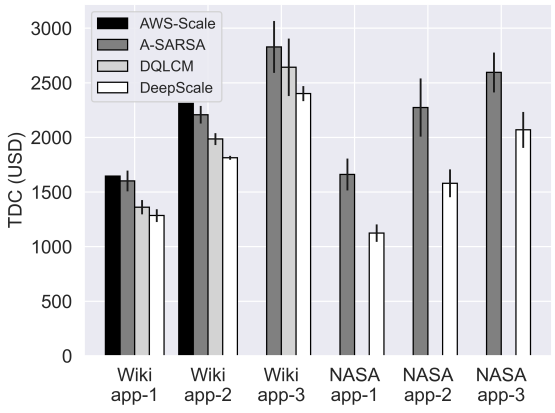


Fig. 6. TDC of constraint-compliant problem instances.

parameters demonstrates the best performance in terms of both the cost minimization and constraint satisfaction in our experiments.

5.4 Constraint Compliance

To compare algorithm performance, we run each experiment 30 times. Fig. 5 shows ART of different problem instances over the testing day achieved by *DeepScale* and the three baselines. The red line in Fig. 5 is the predefined constraint m, i.e., 150 ms.

For the WikiBench workload, ART of *app-3* by *AWS-Scale* is slightly over m. The three RL-based algorithms are capable of meeting the constraint on ART. Particularly, *A-SARSA* always has the lowest ART. Note that the LACS problem aims to minimize TDC subject to the constraint on ART. The lower ART obtained by *A-SARSA* is undesirable due to its high TDC (see Subsection 5.5). For *DQLCM*, ART is longer than *DeepScale* for *app-1* and *app-3*, while shorter than *DeepScale* for *app-2*.

For the NASA HTTP workload, the two threshold-based algorithms, i.e., *AWS-Scale* and *DQLCM*, significantly exceed the predefined constraint m for all applications. This indicates that using two fixed thresholds for container scaling is insufficient to satisfy the performance constraint. With the help of the penalty-based reward function and safety-aware heuristics, *A-SARSA* and *DeepScale* can satisfy constraints for all applications.

5.5 TDC Comparison

In this subsection, we compare *DeepScale* with the baselines for the *constraint-compliant* problem instances in terms of TDC as reported in Fig. 6.

TABLE 4
Performance of workload prediction model in terms of RMSE (requests/s)

	$w(t+1)$	$w(t+2)$	$w(t+3)$	$w(t+4)$	$w(t+5)$
WikiBench	1.78	2.76	3.60	4.35	5.08
NASA HTTP	2.05	3.38	4.51	5.44	6.34

DeepScale saves cost by 22% for *app-1* and 21% for *app-2* compared to *AWS-Scale* for the WikiBench workload. This shows that the industry-leading threshold-based methods may not be suitable for scaling cloud applications with dynamically changing and widely distributed workloads. The bad performance on the deployment cost in our experiments is consistent with previous observations reported in the article [73].

A-SARSA spends 21% more TDC for the WikiBench workload and 39% more TDC for the NASA HTTP workload than *DeepScale* on average over different applications. Through our newly proposed multi-step workload prediction, the scaling decisions made by *DeepScale* is more cost-effective. Beside, *DeepScale* can handle high-dimensional state space more effectively by using *DQN* comparing with the discretization technique adopted in *A-SARSA*.

DeepScale achieves on average 8% less TDC than *DQLCM* for the WikiBench workload, which also demonstrates the effectiveness of the workload prediction model.

The observed performance differences between *DeepScale* and all baselines are all verified through a statistical test (Wilcoxon Rank-Sum test) with a significance level of 0.05. From the above results, we can conclude that *DeepScale* can achieve the lowest TDC and highly likely satisfy constraints on ART. The mean and standard deviation of ART and TDC are presented in TABLE 3. The results are presented in italic for the problem instances on which the baseline algorithms cannot satisfy the constraint on ART. From TABLE 3 we observe that ART and TDC achieved by *DeepScale* has a small standard deviation over 30 repeated experiments, confirming its stability and reliability for the LACS problem.

5.6 Further Analysis

To verify the reliability of the workload prediction model used by *DeepScale*, we evaluate the performance of long short-term memory (LSTM)-based workload prediction model in terms of root-mean-square error (RMSE). The RMSE of the predicted workloads in future f time periods on the testing day are shown in TABLE 4. From TABLE 4, we can see the RMSE ranges from 1.78 to 5.08 requests/s for

TABLE 3
Algorithm performance comparison for the LACS problem with different problem instances (ART in ms, TDC per year in USD, the best is highlighted).

Problem Instance	*AWS-Scale*		*A-SARSA*		*DQLCM*		*DeepScale*	
	ART	*TDC*	*ART*	*TDC*	*ART*	*TDC*	*ART*	*TDC*
Wiki app1	147.34±0	1646.15±0	137.88±1.44	1602.35±94.9	147.61±1.27	1361.45±65.7	146.24±0.97	**1284.8±58.4**
Wiki app2	149±0	2310.45±0	138.45±0.91	2208.25±80.3	146.43±0.63	1985.6±54.75	146.85±0.84	**1814.05±18.25**
Wiki app3	*150.98±0*	*3095.2±0*	139.2±1.64	2828.75±237.25	148.81±1.14	2642.6±262.8	146.63±1.06	**2401.7±69.35**
NASA app1	*381.76±0*	*865.05.2±0*	130.77±1.56	1660.75±146	*242.26±0.44*	*824.9±160.6*	147.37±1.08	**1124.2±80.3**
NASA app2	*359.13±0*	*1178.95±0*	132.22±3.06	2273.95±266.45	*248.26±0.57*	*1084.05±208.05*	147.8±0.92	**1580.45±127.75**
NASA app3	*334.08±0*	*1543.95±0*	132.26±2.03	2595.15±182.5	*237.7±0.77*	*1715.5±281.05*	146.83±1.34	**2069.55±164.25**

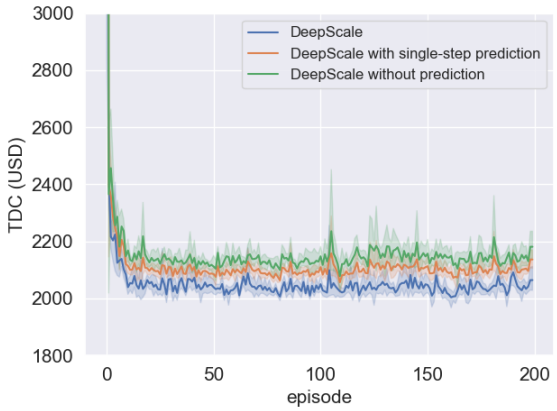


Fig. 7. TDC of NASA *app-3* on the testing day.

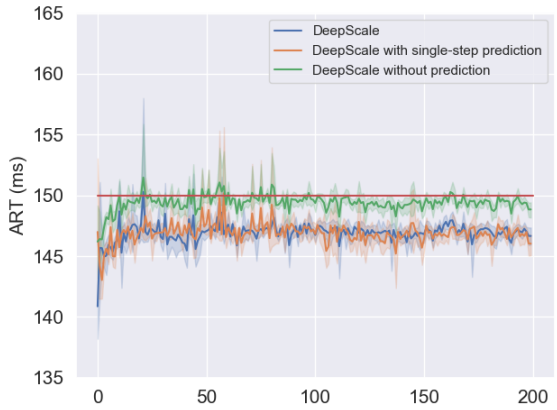


Fig. 8. ART of NASA *app-3* on the testing day.

the WikiBench workload and from 2.05 to 6.34 requests/s for the NASA HTTP workload. We can calculate a normalized RMSE by:

$$NRMSE = \frac{RMSE}{y_{max} - y_{min}}, \tag{15}$$

where y_{max} and y_{min} are the maximum and minimum workloads on the testing day. By calculation, the NRMSE of the workload prediction model for predicting the WikiBench workload and the NASA HTTP workload in the next time period are 0.95% and 1.36% respectively. The results are reliable for *DeepScale* to make scaling decisions [20].

Next, we depict the change of TDC and ART obtained by *DeepScale* on the testing day across all learning episodes in Fig. 7 and Fig. 8. For the ablation study, the performance of *DeepScale* with single-step predicted workload (*DeepScale with single-step prediction* for short) and *DeepScale* without using predicted workloads (*DeepScale without prediction* for short) are also included in Fig. 7 and Fig. 8. Concretely, the *State Constructor* of *DeepScale with single-step prediction* only utilizes $w(t+1)$ as the prediction information. Also, $w(t+1)$ is used by *Action Executor* to generate constraint-compliant scaling decisions. For *DeepScale without prediction*, the *State Constructor* does not collect the information of predicted workloads. Meanwhile, the *Action Executor* makes low-level scaling decisions based on the current CPU utilization rate. We only present the results with respect to the problem instance of NASA *app-3* while a similar trend has been observed for other problem instances. Fig. 7 shows that TDC of the three algorithms becomes flattened after about 50 episodes. By utilizing multi-step future workloads from the workload prediction model, TDC of *DeepScale* is 3% less than *DeepScale with single-step prediction* and 5% less than *DeepScale without prediction*, confirming the importance of considering future workloads on cost-saving. In Fig. 8, we can observe that ART of *DeepScale* and *DeepScale with single-step prediction* falls strictly under m (red line) after about 100 episodes, while *DeepScale without prediction* cannot effectively learn constraint-compliant policies. The results show that using the predicted workload in *Action Executor* is very helpful to reduce ART for meeting the performance constraint.

After training, *A-SARSA*, *DQLCM*, and *DeepScale* can scale containerized applications for incoming requests with trivial computational overhead. Particularly, the total time required to make a high-level scaling decision using *DQN* and a low-level scaling decision through our heuristic-based *Action Executor* is within 1 ms. The training time of *DeepScale* is within 30 minutes, which includes the training of the LSTM-based workload prediction model and the DRL-based scaling policy. Periodical use of *DeepScale* every day is highly feasible in practice. The training time of *A-SARSA* is similar to *DeepScale*. The training of *DQLCM* takes a longer time due to a more complex action space (about 5 hours).

TABLE 5
Performance comparison with different m (ART in ms, TDC per year in USD.

Problem Instance	140ms		160ms	
	ART	TDC	ART	TDC
Wiki app-1	138.42±0.28	1910.82±44.9	157.26±1.48	1255.25±45.28
Wiki app-2	138.96±0.45	2636.93±56.21	158.02±0.89	1795.01±66.29
Wiki app-3	139.19±0.74	3420.37±74.12	158.41±0.72	2334.77±50.37
NASA app-1	137.96±0.89	1255.95±65.92	158.83±0.49	999.45±71.24
NASA app-2	138.24±0.67	1709.69±83.65	157.72±0.84	1404.92±56.77
NASA app-3	138.55±0.98	2291.99±52.7	158.49±0.66	1837.4±99.23

Finally, we apply *DeepScale* to solve the LACS problem with a more stringent threshold, i.e., $m = 140ms$ and a more lax threshold, i.e., $m = 160ms$. The mean and standard deviation of ART and TDC are presented in TABLE 5. From TABLE, 5 we observe that *DeepScale* still can satisfy constraints on ART for all problem instances. A smaller m results in larger TDC while a larger m contributes to cost saving.

6 Discussion

From the experimental results, we can see the effectiveness of *DeepScale* for solving the LACS problem. The following issues deserve further investigations.

Scope. As confirmed by many existing studies [74], it is a common practice for modern applications to be architected as stateless as possible for container-level scalability, e.g., search engines and social media [75]. In line with this trending architectural style, we focus on scaling the stateless applications in the paper. Besides, we assume that each application instance, as a separate service, is deployed to one container. This is a common assumption in the literature [76], [77] and widely exercised in the industry for easier auto-scaling [78]. The proposed algorithm is inapplicable to the scenario where two or more applications running within the same container.

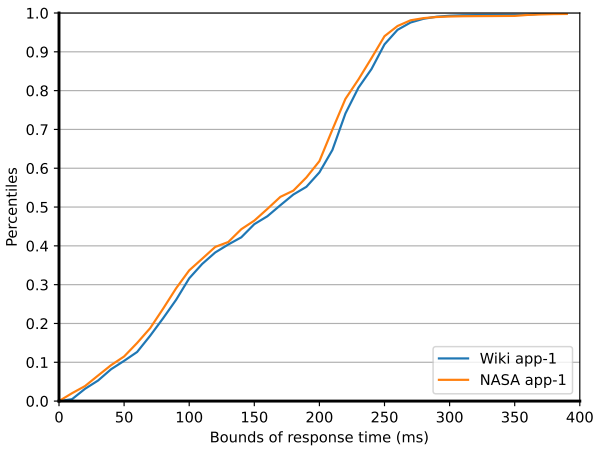


Fig. 9. Percentiles under the response time bounds for *app-1*.

In this paper, we focus on the application performance in terms of the response time. The response time is measured from the moment a user makes an application request to the moment when this user receives the corresponding response, taking into account both the request processing time and round-trip delay between the user and an application instance. Referring to [28], [54], [55], we assume that there is sufficient memory for the applications to support the temporary storage and fast access. Therefore, the request processing time is significantly affected by the CPU utilization of containers. That is, the experimental results may not apply directly to some memory-bound applications.

Tail response time. In this paper, we consider the constraint on the average response time with respect to widely distributed user requests, because it seriously affects the user satisfaction with applications [12], [13]. In the industry, tail response time is also considered to provide deep insights into the performance of some cloud applications under high load [79]. We measured the distribution of the response time achieved by *DeepScale* on the testing day. Subject to the constraint on the average response time, i.e., 150ms in our experiments, 99% of requests are served within 300ms for all the problem instances. For example, Fig. 9 demonstrates the percentiles under the response time bounds ranged from 0ms to 400ms for app-1. In Fig. 9, we can observe that 99% of user requests can be served within 285ms for the WikiBench workload and 292ms for the NASA workload. Several studies have proven that the performance is accepted by many applications [80], [81].

Limitations. In practice, workload contention may occur when an application imposes heavy workload on system resources [82]. In this paper, we apply the proactive strategy to ensure that the workload will not exceed to the critical level of the container capacity. In our experiments, the CPU utilization of containers is consistently below 80%, the previous study showed that the level of contention will not significantly affect the average request processing time [83].

Except for the number of allocated vCPUs, the capacity of an application instance is also impacted by the application caching, hyper-threading architecture, and potential contention. Even so, the number of vCPU is still an important measure of capacity for containerized applications [7], [33]. We will take the other effects on the capacity of application instances into account in our future work.

7 Conclusions and Future Works

In this paper, we proposed an effective deep reinforcement learning algorithm, *DeepScale*, to auto-scale containerized applications in geo-distributed clouds. We first formulated the location-aware container scaling (LACS) problem to minimize the total cost over a time span under the constraint on the average response time for containerized application deployment. By training the newly designed holistic scaling policy with newly designed algorithmic components, *DeepScale* can achieve both cost-effectiveness and constraint satisfaction. Finally, we evaluated the effectiveness of *DeepScale* through conducting extensive simulation studies on real-world datasets. The experiments with realistic application workloads showed that *DeepScale* can effectively satisfy the constraint on the average response time for a variety of applications under significantly different workloads. In the meantime, *DeepScale* can significantly reduce the deployment cost of applications compared with the state-of-the-art baselines, including Amazon auto-scaling service and recently proposed RL-based algorithms.

In this paper, we consider the constraint on the average response time because it seriously affects the user satisfaction of applications [12], [13]. We believe it is a promising future direction to simultaneously consider other constraints, such as service availability among different cloud data centers and data sovereignty. Designing novel safe RL algorithms for solving the LACS problem with multiple potentially conflicting constraints will be an interesting research topic. Furthermore, we will evaluate *DeepScale* on real-world geo-distributed cloud platforms.

References

[1] R. Xu, Y. Wang, H. Luo, F. Wang *et al.*, "A sufficient and necessary temporal violation handling point selection strategy in cloud workflow," *Future Generation Computer Systems*, 2018.

[2] S. Wang, Z. Ding, and C. Jiang, "Elastic scheduling for microservice applications in clouds," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, pp. 98–115, 2020.

[3] T. Shi, H. Ma, and G. Chen, "Energy-aware container consolidation based on pso in cloud data centers," in *IEEE Congress on Evolutionary Computation (CEC)*, 2018, pp. 1–8.

[4] T. Benjaponpitak, M. Karakate, and K. Sripanidkulchai, "Enabling live migration of containerized applications across clouds," in *IEEE Conference on Computer Communications (INFOCOM)*, 2020, pp. 2529–2538.

[5] F. Rossi, M. Nardelli, and V. Cardellini, "Horizontal and vertical scaling of container-based applications using reinforcement learning," in *IEEE International Conference on Cloud Computing (CLOUD)*, 2019, pp. 329–338.

[6] F. Rossi, V. Cardellini, F. L. Presti, and M. Nardelli, "Geo-distributed efficient deployment of containers with kubernetes," *Computer Communications*, vol. 159, pp. 161–174, 2020.

[7] M. Nardelli, V. Cardellini, and E. Casalicchio, "Multi-level elastic deployment of containerized applications in geo-distributed environments," in *IEEE International Conference on Future Internet of Things and Cloud (FiCloud)*, 2018, pp. 1–8.

[8] L. Toka, G. Dobreff, B. Fodor, and B. Sonkoly, "Adaptive AI-based auto-scaling for kubernetes," in *IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2020, pp. 599–608.

[9] Z. Zhong and R. Buyya, "A cost-efficient container orchestration strategy in kubernetes-based cloud computing infrastructures with heterogeneous resources," *ACM Transactions on Internet Technology (TOIT)*, vol. 20, pp. 1–24, 2020.

[10] S. Zhang, T. Wu, M. Pan, C. Zhang, and Y. Yu, "A-SARSA: A predictive container auto-scaling algorithm based on reinforcement learning," in *IEEE International Conference on Web Services (ICWS)*, 2020, pp. 489–497.

[11] B. Tan, H. Ma, and Y. Mei, "A hybrid genetic programming hyper-heuristic approach for online two-level resource allocation in container-based clouds," in *IEEE Congress on Evolutionary Computation (CEC)*, 2019, pp. 2681–2688.

[12] Y. Wu, C. Wu, B. Li, L. Zhang *et al.*, "Scaling social media applications into geo-distributed clouds," *IEEE/ACM Transactions On Networking*, vol. 23, pp. 689–702, 2014.

[13] Y. Mansouri, A. N. Toosi, and R. Buyya, "Cost optimization for dynamic replication and migration of data in cloud data centers," *IEEE Transactions on Cloud Computing*, vol. 7, pp. 705–718, 2017.

[14] T. Shi, H. Ma, G. Chen, and S. Hartmann, "Location-aware and budget-constrained service deployment for composite applications in multi-cloud environment," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, pp. 1954–1969, 2020.

[15] Amazon. (2022) AWS fargate pricing. [Online]. Available: https://aws.amazon.com/fargate/pricing/

[16] Azure. (2022) Azure container instances pricing. [Online]. Available: https://azure.microsoft.com/en-us/pricing/details/container-instances/

[17] Y. Aldwyan, R. O. Sinnott, and G. T. Jayaputera, "Elastic deployment of container clusters across geographically distributed cloud data centers for web applications," *Concurrency and Computation: Practice and Experience*, p. e6436.

[18] T. Shi, H. Ma, G. Chen, and S. Hartmann, "Cost-effective web application replication and deployment in multi-cloud environment," *IEEE Transactions on Parallel and Distributed Systems*, 2021.

[19] Amazon. (2022) AWS auto scaling. [Online]. Available: https://aws.amazon.com/autoscaling/

[20] M. Imdoukh, I. Ahmad, and M. G. Alfailakawi, "Machine learning-based auto-scaling for containerized applications," *Neural Computing and Applications*, vol. 32, pp. 9745–9760, 2020.

[21] M. Cheng, J. Li, and S. Nazarian, "DRL-cloud: Deep reinforcement learning-based resource provisioning and task scheduling for cloud service providers," in *Asia and South Pacific Design Automation Conference*. IEEE, 2018, pp. 129–134.

[22] N. Liu, Z. Li, J. Xu, Z. Xu *et al.*, "A hierarchical framework of cloud resource allocation and power management using deep reinforcement learning," in *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2017, pp. 372–382.

[23] D. Yi, X. Zhou, Y. Wen, and R. Tan, "Efficient compute-intensive job allocation in data centers via deep reinforcement learning," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, pp. 1474–1485, 2020.

[24] T. Shi, H. Ma, G. Chen, and S. Hartmann, "Location-aware and budget-constrained service brokering in multi-cloud via deep reinforcement learning," in *International Conference on Service-Oriented Computing*. Springer, 2021, pp. 756–764.

[25] R. Cheng, G. Orosz, R. M. Murray, and J. W. Burdick, "End-to-end safe reinforcement learning through barrier functions for safety-critical continuous control tasks," in *AAAI Conference on Artificial Intelligence*, vol. 33, 2019, pp. 3387–3395.

[26] E.-J. van Baaren, "Wikibench: A distributed, wikipedia based web application benchmark," *Master's thesis, VU Amsterdam*, 2009.

[27] Lawrence Berkeley National Laboratory. (2022) Traces available in the internet traffic archive. [Online]. Available: http://ita.ee.lbl.gov/html/traces.html

[28] Z. Tang, X. Zhou, F. Zhang, W. Jia, and W. Zhao, "Migration modeling and learning algorithms for containers in fog computing," *IEEE Transactions on Services Computing*, vol. 12, pp. 712–725, 2018.

[29] RANCHER. (2021) Hybrid cloud and multi cloud. [Online]. Available: https://rancher.com

[30] R. Hat. (2021) Red hat openshift container platform. [Online]. Available: https://www.openshift.com/products/container-platform

[31] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, "Autonomic vertical elasticity of docker containers with elasticdocker," in *IEEE International Conference on Cloud Computing (CLOUD)*, 2017, pp. 472–479.

[32] C. de Alfonso, A. Calatrava, and G. Moltó, "Container-based virtual elastic clusters," *Journal of Systems and Software*, vol. 127, pp. 1–11, 2017.

[33] S. N. Srirama, M. Adhikari, and S. Paul, "Application deployment using containers with auto-scaling for microservices in cloud environment," *Journal of Network and Computer Applications*, vol. 160, p. 102629, 2020.

[34] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych *et al.*, "Autopilot: workload autoscaling at google," in *Fifteenth European Conference on Computer Systems*, 2020, pp. 1–16.

[35] V. Podolskiy, M. Mayo, A. Koay, M. Gerndt, and P. Patros, "Maintaining slos of cloud-native applications via self-adaptive resource sharing," in *IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, 2019, pp. 72–81.

[36] A. Bauer, N. Herbst, S. Spinner, A. Ali-Eldin, and S. Kounev, "Chameleon: A hybrid, proactive auto-scaling mechanism on a level-playing field," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, pp. 800–813, 2018.

[37] Y. M. Ramirez, V. Podolskiy, and M. Gerndt, "Capacity-driven scaling schedules derivation for coordinated elasticity of containers and virtual machines," in *IEEE International Conference on Autonomic Computing (ICAC)*, 2019, pp. 177–186.

[38] A. J. Fahs, G. Pierre, and E. Elmroth, "Voilà: Tail-latency-aware fog application replicas autoscaler," in *International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2020, pp. 1–8.

[39] A. J. Fahs and G. Pierre, "Tail-latency-aware fog application replica placement," in *International Conference on Service-Oriented Computing (ICSOC)*. Springer, 2020, pp. 508–524.

[40] H. Arabnejad, C. Pahl, P. Jamshidi, and G. Estrada, "A comparison of reinforcement learning techniques for fuzzy cloud auto-scaling," in *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2017, pp. 64–73.

[41] J. B. Benifa and D. Dejey, "RLPAS: Reinforcement learning-based proactive auto-scaler for resource provisioning in cloud environment," *Mobile Networks and Applications*, vol. 24, pp. 1348–1363, 2019.

[42] P. Jamshidi, A. Sharifloo, C. Pahl, H. Arabnejad *et al.*, "Fuzzy self-learning controllers for elasticity management in dynamic cloud architectures," in *International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA)*, 2016, pp. 70–79.

[43] R. N. Calheiros, E. Masoumi, R. Ranjan, and R. Buyya, "Workload prediction using ARIMA model and its impact on cloud applications' QoS," *IEEE Transactions on Cloud Computing*, vol. 3, pp. 449–458, 2014.

[44] S. Levine, C. Finn, T. Darrell, and P. Abbeel, "End-to-end training of deep visuomotor policies," *The Journal of Machine Learning Research*, vol. 17, pp. 1334–1373, 2016.

[45] V. Mnih, A. P. Badia, M. Mirza, A. Graves *et al.*, "Asynchronous methods for deep reinforcement learning," in *International Conference on Machine Learning*, 2016, pp. 1928–1937.

[46] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, p. 529, 2015.

[47] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *International Conference on Machine Learning*, 2015, pp. 1889–1897.

[48] D. Silver, A. Huang, C. J. Maddison, A. Guez *et al.*, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, p. 484, 2016.

[49] J. Garcia and F. Fernández, "Safe exploration of state and action spaces in reinforcement learning," *Journal of Artificial Intelligence Research*, vol. 45, pp. 515–564, 2012.

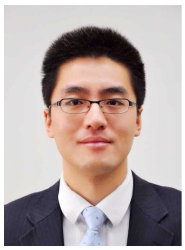
[50] J. Achiam, D. Held, A. Tamar, and P. Abbeel, "Constrained policy optimization," in *International Conference on Machine Learning*. PMLR, 2017, pp. 22–31.

[51] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *ACM Symposium on Cloud Computing*, 2012, pp. 1–13.

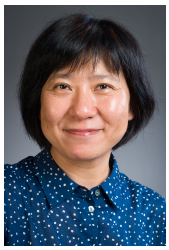
[52] K.-C. Huang and B.-J. Shen, "Service deployment strategies for efficient execution of composite saas applications on cloud platform," *Journal of Systems and Software*, vol. 107, pp. 127–141, 2015.

[53] D. Weyns, B. Schmerl, V. Grassi, S. Malek *et al.*, "On patterns for decentralized control in self-adaptive systems," in *Software Engineering for Self-Adaptive Systems II*. Springer, 2013, pp. 76–107.

[54] Z. Han, H. Tan, G. Chen, R. Wang *et al.*, "Dynamic virtual machine management via approximate markov decision process," in *IEEE International Conference on Computer Communications (INFOCOM)*, 2016, pp. 1–9.
[55] R. Buyya, A. Beloglazov, and J. Abawajy, "Energy-efficient management of data center resources for cloud computing: A vision, architectural elements, and open challenges," *Eprint Arxiv*, vol. 12, pp. 6–17, 2010.
[56] J. Vilaplana, F. Solsona, I. Teixidó, J. Mateo *et al.*, "A queuing theory model for cloud computing," *The Journal of Supercomputing*, vol. 69, pp. 492–507, 2014.
[57] J. D. Little and S. C. Graves, "Little's law," in *Building intuition*. Springer, 2008, pp. 81–100.
[58] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, pp. 1735–1780, 1997.
[59] V. Huang, G. Chen, P. Zhang, H. Li *et al.*, "A scalable approach to sdn control plane management: High utilization comes with low latency," *IEEE Transactions on Network and Service Management*, vol. 17, pp. 682–695, 2020.
[60] I. Saidu, S. Subramaniam, A. Jaafar, and Z. A. Zukarnain, "A load-aware weighted round-robin algorithm for IEEE 802.16 networks," *EURASIP Journal on Wireless Communications and Networking*, vol. 2014, pp. 1–12, 2014.
[61] S. Kardani-Moghaddam, R. Buyya, and K. Ramamohanarao, "ADRL: A hybrid anomaly-aware deep reinforcement learning-based resource scaling in clouds," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 3, pp. 514–526, 2020.
[62] Amazon. (2023) Configure connection draining for your classic load balancer. [Online]. Available: https://docs.aws.amazon.com/elasticloadbalancing/latest/classic/config-conn-drain.html
[63] A. N. Toosi, C. Qu, M. D. de Assunção, and R. Buyya, "Renewable-aware geographical load balancing of web applications for sustainable data centers," *Journal of Network and Computer Applications*, vol. 83, pp. 155–168, 2017.
[64] B. Wickremasinghe, R. N. Calheiros, and R. Buyya, "Cloudanalyst: A cloudsim-based visual modeller for analysing cloud computing environments and applications," in *IEEE International Conference on Advanced Information Networking and Applications (AINA)*, 2010, pp. 446–452.
[65] T. Shi, H. Ma, G. Chen, and S. Hartmann, "Location-aware and budget-constrained application replication and deployment in multi-cloud environment," in *IEEE International Conference on Web Services (ICWS)*, 2020, pp. 110–117.
[66] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *ACM SIGCOMM Conference on Internet Measurement*, 2010, pp. 267–280.
[67] A. Paszke, S. Gross, F. Massa, A. Lerer *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, pp. 8026–8037, 2019.
[68] G. Brockman, V. Cheung, L. Pettersson, J. Schneider *et al.*, "OpenAI Gym," *arXiv preprint arXiv:1606.01540*, 2016.
[69] A. V. Papadopoulos, L. Versluis, A. Bauer, N. Herbst *et al.*, "Methodological principles for reproducible performance evaluation in cloud computing," *IEEE Transactions on Software Engineering*, vol. 47, pp. 1528–1543, 2019.
[70] S. Siami-Namini, N. Tavakoli, and A. S. Namin, "A comparison of arima and lstm in forecasting time series," in *IEEE International Conference on Machine Learning and Applications (ICMLA)*, 2018, pp. 1394–1401.
[71] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.
[72] R. Kuschnig, I. Kofler, and H. Hellwagner, "Improving internet video streaming performance by parallel tcp-based request-response streams," in *IEEE Consumer Communications and Networking Conference*, 2010, pp. 1–5.
[73] S. M. R. Nouri, H. Li, S. Venugopal, W. Guo *et al.*, "Autonomic decentralized elasticity based on a reinforcement learning controller for cloud applications," *Future Generation Computer Systems*, vol. 94, pp. 765–780, 2019.
[74] L. M. Vaquero, L. Rodero-Merino, and R. Buyya, "Dynamically scaling applications in the cloud," *ACM SIGCOMM Computer Communication Review*, vol. 41, pp. 45–52, 2011.
[75] N. S. Gill. (2023) Stateful and stateless applications and its best practices. [Online]. Available: https://www.xenonstack.com/insights/stateful-and-stateless-applications
[76] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi, "Cloud container technologies: a state-of-the-art review," *IEEE Transactions on Cloud Computing*, vol. 7, pp. 677–692, 2017.
[77] X. Wan, X. Guan, T. Wang, G. Bai, and B.-Y. Choi, "Application deployment using microservice and docker containers: Framework and optimization," *Journal of Network and Computer Applications*, vol. 119, pp. 97–109, 2018.
[78] Kubernetes. (2023) Pods. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/pods/
[79] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, pp. 74–80, 2013.
[80] Amazon. (2023) The state of online retail performance. [Online]. Available: https://s3.amazonaws.com/sofist-marketing/State+of+Online+Retail+Performance+Spring+2017+-+Akamai+and+SOASTA+2017.pdf
[81] Enterspeed. (2023) How fast should your website be in 2022? [Online]. Available: https://www.enterspeed.com/blog/how-fast-should-your-website-be
[82] A. Samir and C. Pahl, "Detecting and predicting anomalies for edge cluster environments using hidden markov models," in *International Conference on Fog and Mobile Edge Computing (FMEC)*. IEEE, 2019, pp. 21–28.
[83] N. Tsikoudis, A. Papadogiannakis, and E. P. Markatos, "LEoNIDS: A low-latency and energy-efficient network-level intrusion detection system," *IEEE Transactions on Emerging Topics in Computing*, vol. 4, pp. 142–155, 2014.



Tao Shi received his Ph.D degree from Victoria University of Wellington, New Zealand. He is currently a Lecturer with the Science and Information College, Qingdao Agricultural University, China. His main research interests include cloud computing and distributed system. His research focuses on resource management and combinatorial optimization in clouds.



Hui Ma received her B.E. degree from Tongji University and her B.S. (Hons.), M.S. and Ph.D degrees from Massey University. She is currently an Associate Professor at Victoria University of Wellington. Her research interests include service computing, database systems, and resource allocation in clouds. She has served as a PC member for more than 80 international conferences, including seven times as a PC chair for conferences and twice of a local co-chair.

Gang Chen received the Ph.D degree from Nanyang Technological University (NTU), Singapore. He is currently a Senior Lecturer with the School of Engineering and Computer Science, Victoria University of Wellington, New Zealand. His research interests include reinforcement learning, evolutionary computation and their application to optimization and scheduling problems, resource management and load balancing in networked systems.

Sven Hartmann received his Ph.D. in 1996 and his D.Sc. in 2001, both from the University of Rostock (Germany). From 2002 to 2007 he worked first as an associate professor, then full professor for information systems at Massey University (New Zealand). Since 2008 he is a full professor of computer science and chair for databases and information systems at Clausthal University of Technology (Germany). There he is also serving as academic dean at the Faculty of Mathematics, Informatics and Mechanical Engineering. Sven has more than 150 publications. He served as a PC member for more than 80 conferences, including 10 times as PC chair. His research interests include database systems, big data management, conceptual modelling, and combinatorial optimization.