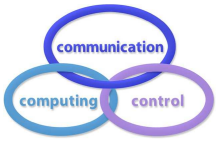
INTERNATIONAL JOURNAL OF COMPUTERS COMMUNICATIONS & CONTROL
Online ISSN 1841-9844, ISSN-L 1841-9836, Volume: 18, Issue: 4, Month: August, Year: 2023
Article Number: 5013, https://doi.org/10.15837/ijccc.2023.4.5013



CCC Publications

A deep reinforcement learning-based optimization method for long-running applications container deployment

L. Deng, Z. Y. Wang, H. Y. Sun, B. Li, X. Yang

Lei Deng
1. School of Computer Science, Northwestern Polytechnical University
Xi'an, 710129, Shaanxi, China
2. Science and Technology on Electro-optic Control Laboratory
Luoyang, 471023, China

Zhaoyang Wang*
Science and Technology on Electro-optic Control Laboratory
Luoyang, 471023, China
*Corresponding author: zdszr2022@163.com

Haoyang Sun
School of Computer Science, Northwestern Polytechnical University
Xi'an, 710129, Shaanxi, China

Bo Li
Science and Technology on Electro-optic Control Laboratory
Luoyang, 471023, China

Xiao Yang
School of Computer Science, Northwestern Polytechnical University
Xi'an, 710129, Shaanxi, China

Abstract

Unlike the short execution cycles of batch jobs, intelligent algorithmic applications typically run in long-cycle containers in the cloud(Long-Running Applications, LRA). Both need to meet strict SLO (service level objective) requirements, consider performance scaling to cope with peak load demands, and face issues such as I/O dependencies and resource contention and interference from coexisting containers. The above greatly complicates container deployment and can easily lead to performance bottlenecks. Therefore, the optimization of LRA-like container deployment is one of the key issues that cannot be avoided and needs to be addressed in the cloud computing model. This research uses deep reinforcement learning (DRL) to optimize the deployment of LRAs class containers. The proposed non-generic model is able to customize a dedicated model for each container group, providing high-quality placement and low training complexity; meanwhile, the proposed batch deployment scheme is able to optimize various scheduling objectives that are not directly supported by existing constraint-based schedulers, such as minimizing SLO violations. The experimental results show that the performance of the DRL deployment algorithm improves by

56.2% compared to the average RPS of the baseline, indicating that the manual deployment scheme can only meet the basic requirements but cannot cover the complex interactions between containers under constraints from a global perspective. This limitation severely limits the performance of the whole pod. Meanwhile, based on previous experience, the time consumption of a single deployment scheme is about 1 hour, while the time consumption of the DRL deployment algorithm may be less than 7.5 minutes.

1 Introduction

Production clusters typically run two types of workloads, long-running applications (LRA) for online cloud services [1–8] and batch jobs for offline data analysis[9, 10]. When deploying, the principle of high priority for service and low priority for batching is usually followed.

Unlike short execution cycles of batch jobs, LRAs run in long-cycle containers, lasting from hours to months [11–13]. Compared with batch jobs, LRAs have to meet strict SLO (Service-Level Objective) requirements and consider performance scaling in response to load peak demands [14]. At the same time, I/O dependencies [15] are common in different LRAs. For example, the instance of mirror image algorithm container may transfer the processed data to Redis for caching which also complicates the deployment of LRAs class containers.

Additionally, LRAs class containers often face interference from co-existing containers that compete for shared resources such as CPU cache, network, I/O, and memory bandwidth. The above problems greatly complicate container deployment and easily lead to performance bottlenecks. Therefore, LRA class container deployment optimization is one of the key issues that cannot be avoided and need ironing out in the cloud computing model.

Existing LRAs schedulers define various location constraints [11, 16, 17], such as affinity and anti-affinity. According to the above principles, I/O-dependent containers should be placed on one machine to avoid communication overhead, while competing containers should be placed on separate machines to avoid resource interference.

Typically, the scheduler deploys containers by simple heuristics [11, 18–24] to satisfy as many constraints as possible. However, this constraint-based scheduling has the following problems: First, it requires cluster deployers to identify complex container interactions based on operational experience, and manually map these interactions to deployment-limited constraints, which costs much time and may still not precise. Second, position constraints only provide a qualitative scheduling criterion, but do not quantify the actual performance impact (e.g., how much throughput will suffer if a certain constraint is violated). Therefore, when the scheduler cannot satisfy all constraints, it may choose incorrectly, thereby violating those constraints that are more influential. Third, complex layout constraints have difficulties in optimizing in large clusters.

This paper uses deep reinforcement learning (DRL)[25–27] to optimize the deployment of LRAs class containers. The specific innovations are as follows:

- Propose a non-generic model: tailoring a dedicated model for each container group enables high-quality placement and reduces training complexity ;
- The batch deployment scheme is also able to optimize various scheduling objectives that are not directly supported by existing constraint-based schedulers, such as minimizing SLO violations ;
- While containers can effectively divide CPU cores and memory capacity among packaged applications, they are limited when they compete for shared resources that are not managed by the OS kernel (such as networking, CPU cache, disk I/O, and memory bandwidth). Therefore, putting many competing containers together can increase resource contention and cause severe interference that hurt the performance of all hosting containers.

Overall, the DRL algorithm improves the deployment efficiency of LRAs containers and the RPS performance of pods after deployment. Additional computing overhead brought by the deployment is moderate.

2 Related work

Lu Shenglin et al. [28] added a weight scheduling module to the algorithm that comes with Docker Swarm. When a container receives a request from the client, the weight scheduling module calculates the weight of the node based on the resource consumption of each one. By convention, higher weights indicate higher total resource occupancy. This strategy indeed optimizes the load-balancing ability of Docker Swarm, but it only considers the overall resource allocation. It does not classify and calculate the resources and cannot guarantee the utilization of various resource allocations.

Dong Zhang et al. [29] designed a new container scheduling algorithm, which considers three aspects: the network transmission overhead between the server container and the client, the network overhead of transferring images from the remote repository to the local, and the energy consumption of nodes. The article defines corresponding functions for each factor, integrates them into a linear equation, and obtains the optimal solution through calculation. The algorithm uses integer programming to find the optimal solution.

Bo Liu et al. [30] proposed a multi-objective algorithm, which chooses the most suitable node to deploy the container, and reduces the maximum TPS and average response time by fully considering 5 factors: the CPU utilization and memory utilization of each node, the transmission overhead of images on the network, the relationship between containers and nodes, and container clusters.

Daniel Guimaraes Lago et al. [31] proposed a container scheduling algorithm based on awareness of resource type. This algorithm includes two strategies: the first part is to find the optimal deployment physical machine for the container, and the other part is to reduce the consumption of network transmission. However, this algorithm fails to take into account the characteristics of container mirroring.

Kaewkasi et al. [32] proposed a container scheduling strategy based on ant colony optimization. The goal of this algorithm has a better performance by load balancing. But the parameters of this algorithm cannot be adjusted according to the actual situation.

Mohamed K et al. [33] proposed a new architecture based on CAAS. Aiming at the previous research focusing on scheduling containers to physical machines, a two-level structure of the container-virtual machine-physical machine is proposed. Meanwhile, the ACO-BF algorithm is proposed to improve the resource utilization rate on consideration of both the resource utilization of physical machines and the CPU utilization and memory utilization of virtual machines.

D. Kang et al. [34] built a heterogeneous cluster energy efficiency model that classifies applications by using the K-medoid algorithm. The model was demonstrated that can effectively reduce operating costs and energy consumption and ensure application performance through experiments on various application types.

Xiaolong Xu et al. [35] proposed a container scheduling method for the problem of application performance degradation and energy consumption during container migration. This method does not distribute the container image to each resource node in the cluster in advance so that the container needs to increase the distribution time of the image during the scheduling process, which leads to an increase in scheduling time. Meanwhile, the algorithm will cause problems such as increased load in the progress of container migration.

For the virtual machine deployment problem, You Qinggen et al. [36] proposed a virtual machine deployment decision analysis method combined with the bilateral matching model. In the condition of mathematically modeling the bilateral matching problem of virtual machine deployment to maximize the satisfaction of both the virtual machine and the physical machine, a multi-objective optimization algorithm model was constructed, and the optimal matching result was obtained through Lingo software.

For the container deployment problem, Shi Chao et al. [37] combined with the bilateral matching model, introduced several similarity calculation methods in machine learning into the calculation of container preference rules and continued to add virtual machines that have been simulated allocating to containers to preference list, thus addressing the initial deployment issue of integrating containers onto virtual machines.

Kubernetes [38] Community provides device service providers with a unified plug-in solution, Device Manager, which allows users to customize the management and control logic of certain device resources on Kubernetes through the Extended Resources and Device Plugin modules. However, it is

limited to resource discovery, health and allocation, and is not responsible for topology maintenance or monitoring data collection of heterogeneous nodes at the level of management and control.

NVIDIA provides a Device Plugin that can run on a Kubernetes cluster in the form of a DaemonSet. The NVIDIA Device Plugin exposes the number of GPUs on each node, tracks the health of GPUs, and helps enable GPU containers. The vGPU device plugin is an open-source pod resource scheduler built on Device Plugin. Based on retaining the official functions, it divides the physical GPU and limits the video memory and computing units, thereby simulating multiple small vGPU cards. In the Kubernetes cluster, scheduling is based on these split vGPUs, so that different containers can safely share the same physical GPU and improve GPU utilization. In addition, the plug-in can also virtualize the video memory (the used video memory can exceed the physical video memory), run some tasks with large video memory requirements, or increase the number of shared tasks.

Different from normal container deployment, Kubernetes deployment requires GPU resources to be imaged in units of pods, thus isolating GPU resources. The pod resource scheduler used in this article is the k8s-device-plugin that can segment GPU resources.

3 Technical background

When deploying container images, various open-source schedulers can be used to collect cluster node information and deploy them in pod units [39]. The intelligent data analysis algorithm library currently mainly accepts two forms of deployment requests, and the cluster nodes used in the two deployment schemes do not interfere with each other.

The first is a real-time single deployment. Specifically, when deploying a target detection algorithm image, check whether the node has enough resources firstly. If so, it can be deployed directly. If not, an algorithm is designed to close the algorithm image that has not been called for the longest time, choose the pod which can deploy the current target detection algorithm image after releasing resources, and then deployed it. This deployment mode just requires the deployment to be very simple, fast, efficient ,and stable and does not require further performance improvement.

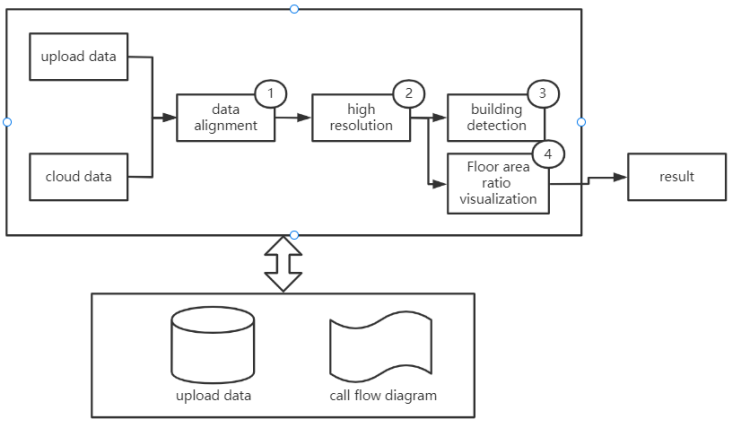


Figure 1: Integrated call example

The second is the integrated deployment as shown in Figure 1. For the caller, multiple requests are still sent for algorithm request processing, but related algorithms may have strong closeness in a certain field and need to run for a long time after one-time deployment. This type of deployment task is the typical LRAs deployment described above. When the intelligent data algorithm library accepts this type of task, it is necessary to make a reservation with the platform maintainer one or two days in advance, give some test samples, and use the test samples to test the stability of the algorithm image before the scheduled time of deployment. Generally, after deployment, the operating cycle is one quarter. One or two algorithm images may be offline for part of the time. The algorithm source code and executable files are not modified, only the mount files and models are replaced, and repeated calls are made.

In the algorithm deployed in Figure 1, the demand for running resources such as CPU and GPU is different, and the demand for the simultaneously deployed object storage service and Redis image service is different too. When only using resource constraints, affinity, and anti-affinity to qualitatively describe deployment constraints, if multiple copies of each algorithm are required to be deployed at the same time, relying on manpower to give appropriate performance deployment locations is impossible.

The platform provides two modes for container deployment, one is to reserve and deploy a batch of containers in advance according to deployment constraints, and the other is to deploy online based on resource configuration. The container images reserved for deployment are mainly long-running applications (LRAs). If only relying on deployment constraints, scheduler performance is mediocre. Because qualitative descriptions do not lead to optimal performance. The platform is based on the open-source virtual GPU scheduler k8s-vGPU-scheduler, which utilizes deep reinforcement learning (DRL) to optimize the deployment of LRAs containers. Cluster resources used for batch pod deployment and single pod deployment requests are completely isolated and non-interfering.

For the request of single pod deployment, an LFU queue is maintained in DRL-Scheduler, and the memory usage and invocation of all deployments are recorded. When a new single deployment request is sent, it will first compare the Gmemory occupied by the video memory of the single pod deployment request with the remaining video memory occupancy, that is, the total resource space minus the already deployed algorithm video memory occupancy (Tmemory - Umemory). If Gmemory $<$ Tmemory - Umemory, it will directly use the scheduler for node deployment. If Gmemory $>$ Tmemory, it will select the least recently called pod from the LFU queue to go offline. Actual pod deployment will not take place until Gmemory $<$ Tmemory - Umemory.

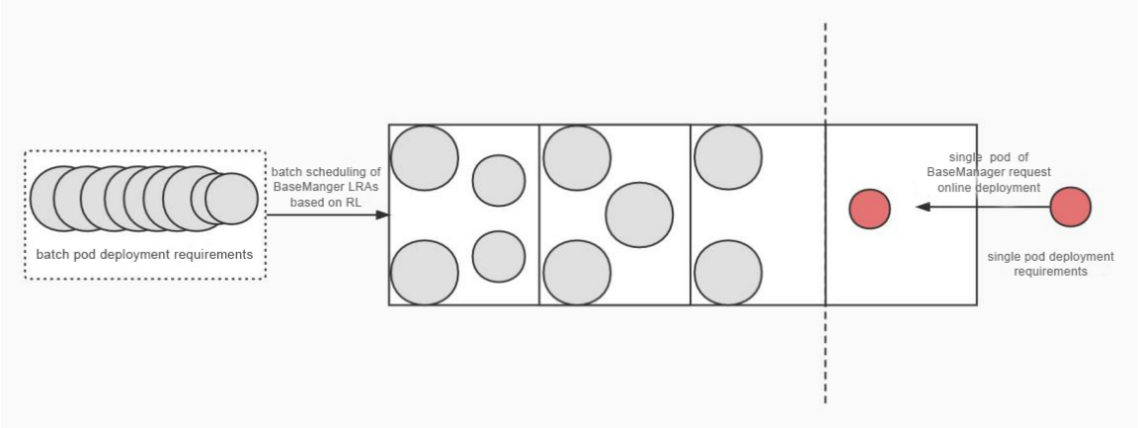


Figure 2: Batch deployment and single pod online deployment

The optimal location of LRAs can be automatically learned using deep reinforcement learning (DRL) techniques without specifying location constraints. In the platform's invocation deployment system, batch deployment of pods uses an intelligent, all-purpose LRAs scheduler (DRL-Scheduler), which can optimize various scheduling objectives, such as throughput, SLO satisfaction, and cluster utilization.

4 Method for LRAs class container deployment optimization based on deep reinforcement learning

DRL-Scheduler understands the complex performance between LRAs containers and between operations through past workload logs or lightweight offline analysis. According to this information, DRL-Scheduler encodes the scheduling strategy into a neural network and trains the containers through a large number of simulated experiments, in which it deploys LRAs containers, predicts their performance, and iteratively improves the strategy.

The key to the DRL-Scheduler is to build a DRL model that can scale to large clusters with thousands of LRAs containers running on thousands of machines. However, directly training a DRL agent at this scale is computationally infeasible because it requires a high-dimensional state representation

of the data, i.e., deployments of containers on all machines. Recently developed DRL techniques for other scheduling problems provided little help in addressing the scalability challenges of container deployments. Most of these works learn to find the optimal scheduling order of batch jobs or network flows, while LRAs scheduling focuses on the interaction between containers, which is essentially a combinatorial layout optimization problem.

Instead of training a generic scheduling strategy offline for all possible input workloads, a dedicated DRL model is trained each time a set of containers arrives. Tailoring a dedicated model for each container group enables high-quality placement and reduces the complexity of training. Although it takes more time to make decisions, LRAs are insensitive to scheduling delays due to their long-running nature [9].

The platform implements batch deployment of pods as a pluggable scheduling service in Kubernetes, and this paper evaluates its performance in clusters using real applications. The batch deployment scheme is also able to optimize various scheduling objectives not directly supported by existing constraint-based schedulers, such as minimizing SLO violations. DRL-Scheduler can easily scale to large clusters running thousands of containers on multiple machines. While containers can effectively divide CPU cores and memory capacity among packaged applications, they are limited when they compete for shared resources that are not managed by the OS kernel (such as networking, CPU cache, disk I/O, and memory bandwidth). Therefore, putting many competing containers together can increase resource contention and cause severe interference that can hurt the performance of all managed containers. On the other hand, sometimes it is beneficial to deploy LRAs containers concurrently. In production clusters, many online services are structured as graphs that rely on LRAs containers, where the output of upstream containers is forwarded to downstream instances for further processing. Putting two dependent LRAs containers on a single machine avoids transferring large amounts of data over the network, resulting in faster responses to query processing. Deployment constraints cannot quantify the significance of potential performance gains (or losses). Take the affinity constraint as an example. Other than stating that it is advisable to have two coexisting containers, it provides no clue as to how much performance gain coexistence can provide. Hence, given conflicting constraints, the scheduler may unwisely choose to violate those constraints that have a greater impact. LRAs scheduling naturally creates a DRL problem. Given the location (state) of existing containers in the cluster, the scheduler (DRL agent) learns to deploy new LRAs containers (actions) based on its interaction with the cluster (environment). The scheduling strategy is encoded using a neural network and trained through extensive simulation experiments: the scheduler deploys containers, observes performance results (rewards), and iteratively improves the strategy.

The scheduler schedules T containers in groups when they arrive. Consider each group schedule as a set of T steps in which only one container is deployed to the machine. More specifically, consider an N-node cluster running M applications. Assume that in step t, the DRL agent has deployed $t-1$ containers in the group and will schedule the next container ct. Embed the container ct into the one-hot encoded vector $e =< e_1, e_2, ..., e_M >$, if e_i is 1, it means the corresponding ct belongs to application i. The node state is further defined as a vector $Vn =< v_{n1}, v_{n2}, ..., v_{nM} >$, where v_{ni} counts the number of containers it is running for application i. Concatenate the container ct with the state of all nodes in step t, defining the cluster state $st =< e, v_1, ..., v_N >$, which is observed by the DRL agent.

Given a state st, the DRL agent performs an action $at = n$, schedules the container ct to node n, and transfers the system to a new state $st + 1$ in the next step. After all T containers have been scheduled, the agent will evaluate the performance of the group deployment in a final step T. More specifically, the agent receives no reward rt in an intermediate step $t < T$, while the final reward rT can be any performance metric such as normalized average throughput of planned container groups, SLO satisfaction rate, cluster utilization, or Their combination is independent of the actual container scheduling order within the group. The scheduling policy is encoded as a neural network with parameters θ, namely the policy network $\pi\theta$. It takes the cluster state as input and outputs a distribution over all possible operations. The policy network is trained using a reinforcement learning algorithm that uses the rewards observed during training to perform gradient ascent on the parameter θ.

Algorithm 1 Flow of policy gradient training algorithm

Input: Cluster of N node $n \in \{1, ..., N\}$, the group of size T is assigned to the Container of the cluster $\{c_1, c_2, ..., c_T\}$

Output: Allocate $a_1, a_2, ..., a_T$ to the group of size T, $a_t \in \{1, ..., N\}$ means that container ct is allocated on node $n = a_t$

1: Initialize the environment and performance evaluation **R**, leave the replay buffer **B** and top performance **R*** empty
2: **for** episode=1,2,...,E **do**
3: State of initialization $s = \{e, v_1, ..., v_N\}$
4: **for** t=1,2,...T **do**
5: choose an action a_t from strategy $\pi(a_t|s_t)$
6: perform the action and get a new state s_{t+1}
7: **end for**
8: Collect all performance indicators as a reward $\sum_t R(c_t)$
9: **if** $r \geq \eta R*$ **then**
10: store experience $\{s_1, a_1, ..., s_T, a_T, r\} R* \leftarrow (r, R*)$ to **B**
11: **end if**
12: use reinforcement learning algorithm to update $\pi(a_t|s_t)$ in every group **C** and the latest experience of **C** and batch experience selected from **B** is used
13: **end for**
14: Return the action $a_1, a_2, ...a_T$ made with experience according to the highest reward $r = R*$ of playback buffer **B**

$$\theta \leftarrow \theta + \alpha \sum_t \nabla_\theta \log \pi_\theta \left(\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} - b \right) \tag{1}$$

where α is the learning rate, $\gamma \in (0, 1]$ is the factor used to discount future rewards, and b is the baseline used to reduce the variance of the policy gradient. A common choice of the baseline is the average reward observed during training. These hyperparameters can be tuned for faster convergence. Intuitively, this gives the agent a high chance of choosing an action with an above-average reward, thereby speeding up the training process. Additionally, the DRL-Scheduler uses experience replay to further speed up training. The main idea is to store execution actions that lead to high rewards in a replay buffer. The agent will periodically sample these high-performance actions for policy updates. By repeating good experiences encountered in previous training, the agent can learn faster.

Alorithm 1 shows the pseudocode for DRL agent training running in normal and enhanced versions. Specifically, the input is a set of containers, and the output is their corresponding assignment. Line 3 shows the state input, which is the LRAs vector e of the current container to be scheduled and all nodes v1,...,vN. Lines 4-6 are the allocation process for all T containers in the batch, and the reward is obtained after all tasks have been scheduled. Lines 9-10 describe the replay technique, which stores the experience with a high reward in the replay buffer. In line 10, the agent updates its policy with recent experience and experience sampled from the replay buffer, using a reinforcement learning algorithm. Finally, the DRL agent selects the highest rewarding action sequence in the replay buffer and returns it as the final deployment decision (Line 11). Training neural networks with policy gradients requires DRL agents to frequently interact with the environment, which is time-consuming for LRAs scheduling. In a real system, it takes at least a few minutes to deploy a container and measure its performance. Since DRL training typically takes tens of thousands of iterations to complete, having the batch pod scheduler interact directly with the actual cluster is too slow and impractical. Similar to previous work, the platform develops a high-fidelity simulator of the cluster environment that can predict container performance at a given location. This enables learning through simulated experiments without deploying containers in real clusters.

Instead of modeling low-level resource interference based on contention on CPU cache or memory bandwidth (which may not be available in production traces), this paper moves to high-level perfor-

mance metrics such as container throughput and request latency. Then it predicts how these metrics will change under different container deployment scenarios. In particular, the co-location vectors of machine-level containers, and the observed RPS/latency of each resident container are recorded and used as training samples for the simulator.

The simulator uses multivariate random forest (RF) as the main regressor to characterize the container interactions. RF methods use combinatorial decision trees to perform regression tasks that can make accurate predictions with a small amount of training data. It also accommodates overfitting when a large number of replicates are provided. These two characteristics make RF regressors ideal for simulators. The previous scheduler based on DRL trains a unified DRL model offline and uses it online for scheduling decisions. When it comes to large clusters, training scheduling strategies offline to deal with extremely varied workloads inevitably leads to poor performance. Instead, DRL-Scheduler trains a dedicated DRL model on the spot when a new batch of containers arrives. While training a dedicated model takes time, the long-running nature of LRAs containers allows them to tolerate relatively long scheduling delays (e.g., tens of minutes) in exchange for better locations. In the training set, the DRL agent deploys a set of containers by taking a series of actions that together constitute a scheduling decision. Since the operational space deployed per container is proportional to the number of machines in the cluster (i.e., one machine is chosen from all machines), the space of scheduling decisions also grows exponentially with the size of the cluster.

5 Design and Implementation of Deployment System

The algorithm library platform realizes the storage, deployment, and invocation through web interfaces such as RESTful API/Socket provided by DRL-Scheduler. As shown in Figure 3, the management of the core image is based on the interaction between StorageShell and the storage center. StorageShell has done a layer of encapsulation based on Docker Daemon to realize basic work such as image information parsing and operation. The daemon thread Docker Daemon can directly manage the images downloaded by the docker Registry. This paper implements the model assembly function based on it, makes RESTful API calls to the daemon thread Docker Daemon through StorageShell, and performs image recovery operations through *OveDRLaps.json. OveDRLaps.json* is specially used to record model information and implement multi-version model switching and management. Implement the management and download of file sets. The storage center uses the object storage node to manage OveDRLaps Lib, which realizes the management and distribution of related data such as models, files, and datasets. The docker Registry of the storage center is unified in format through encapsulation of the storage center, the calling interface and OveDRLaps Lib in the code. Containers use the interfaces and methods provided by ContainerManager while deploying. The process of interacting with StorageShell is mainly based on the commands provided by docker itself and *container_integration.json.*

5.1 Deploying Pods System Design

The platform's deployment scheme mainly uses the DRL-Scheduler component to deploy and call pods, uses the open source pod call component k8s-vgpu-scheduler, and embeds the DRL deployment algorithm to complete batch pod deployment requests and single pod deployment requests, which is shown in Figure 4. At the same time, DRL-Scheduler completes large files and the rapid switching of models through *container_integration.json* and *package.json.* It also realizes version control and automatic assembly. Automatic removal is possible for idle nodes and nodes that are no longer needed. For the resource request of a single pod, DRL-Scheduler directly calls *ContainerManager.CreateFleet()* to expand the required instance by evaluating the requested resource size, node selector, affinity, tolerance, etc.

5.2 BaseMananger class

DRL-Scheduler class is the core class of the algorithm library platform to deploy pods, and it manages all pods on the master node of the cluster. The Batch deployment and single pod online

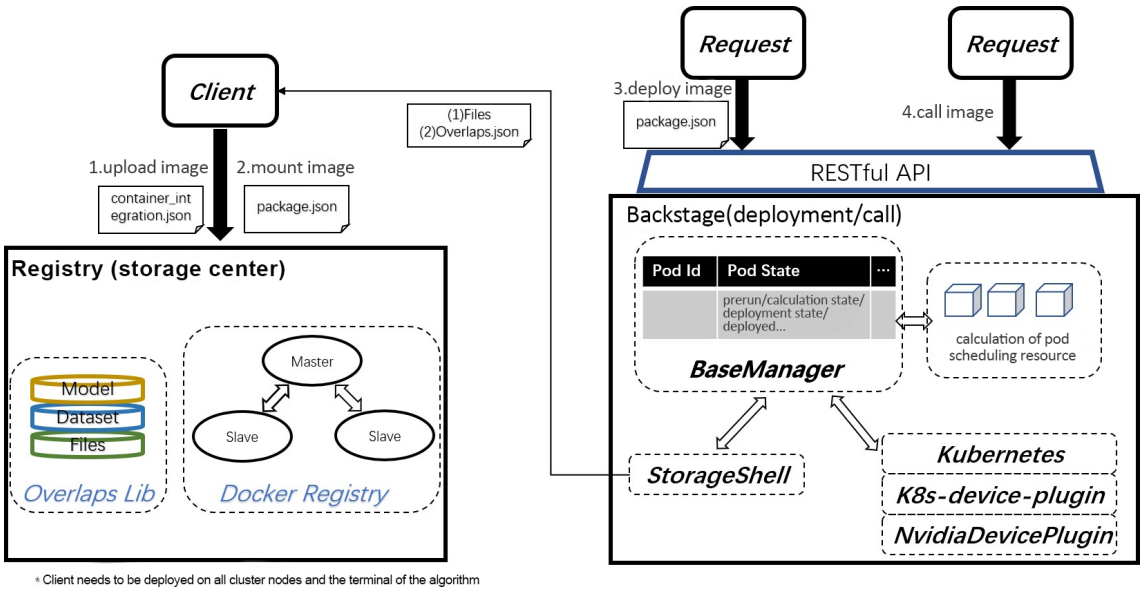


Figure 3: Deployment System Architecture

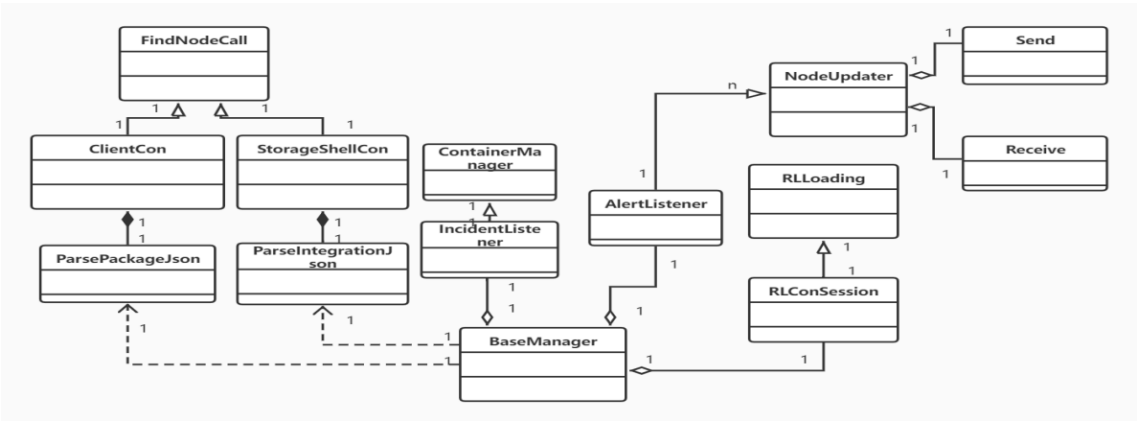


Figure 4: DRL-Scheduler java class design diagram

deployment are given in Figure 5. Use *package.json* and *container_integration.json* to configure StorageShell and Container-Manager respectively to implement fine-grained management and deployment of container images. ContainerManager calculates the best batch deployment scheme through DRLLoading, returns the result to BaseManager, and finally uses the pod scheduler to achieve batch deployment. DRL-Scheduler summarizes the resource occupancy of all pods into Redis, providing an open resource status query interface. The core methods include sendPodsMapper and podArrange, which respectively use the DRL algorithm to calculate the current optimal deployment and complete single-node or multi-node pod deployment.

During the entire deployment process of DRL-Scheduler, pod deployment requests will trigger different types of events. For example, after first receiving the deployment request, if it is PodsArrange (batch deployment), it will first enter the pre-running state. At this time, DRL Loading will not be called immediately to calculate the optimal deployment plan, but a collection function will be triggered first, which will take resource usage of all nodes from Redis. Only in the step of obtaining the resource status of the node, multiple event listeners are encapsulated to prevent huge errors in the resource status of the node. After the pre-running state, it enters the calculation state of batch deployment scheme calculation. At this state, the current thread will be blocked until DRLLoading returns the result of LocationMapper. The deployment event will be triggered after obtaining the deployment plan. If it is OnePodArrange (single-node online deployment), skip the pre-running and computing state and directly enter the deployment state.

Listener components include event sources, event objects, and event listeners. When the pod status

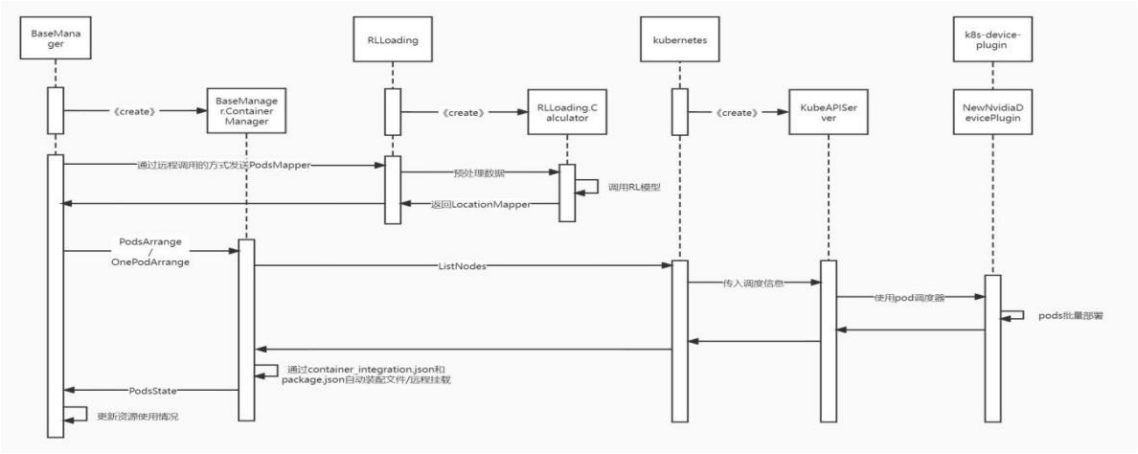


Figure 5: Batch deployment and single pod online deployment

is queried or modified, the method of event listener will be triggered, and the event object will be passed in when the method of event listener is called. In the event listener of the pod, you can operate by the object source acquired from the event object.

Take deployment events in DRL-Scheduler as an example, using Kubernetes and k8s-device-plugin to arrange the actual deployment nodes for each pod, it will freely deploy each pod to the specified physical machine. The deployment process is parallel, that is, each physical machine node is free to use the pod scheduler, and any single deployment failure does not affect the deployment of other nodes. Significantly, on the node which adopts the online deployment pod scheme, the scheduler will close the algorithm which has the minimum probability to be called when the pod resources are insufficient through the calling situation, and the obtained by closing the algorithm will be called, and the corresponding resource status of the pod in Redis will be modified.

After the pod is deployed, StorageShell will be automatically assembled or remotely mounted according to *package.json* and *container_integration.json* to complete the entire deployment process. At this point, the state of the pod is rewritten to the deployed state, the state recording event is triggered, and the resource occupancy status is written to Redis.

In the event of resources mutex, pod deployment failure, etc., the corresponding watchdogs will summarize and report the error information through the above LogAggregate. All pod deployment failures will directly form an emergency-level error report, which will be quickly captured and notified to the relevant team. The deployment process does not support either rollback transactions or processing multiple batches of pod batch deployment requests at the same time, but it supports multiple online single pod deployment requests. The reason is that the node resources for batch deployment of pods and the node resources for online pod deployment are separated (resource requests are limited by a simple formula). Although two pods may be deployed on the same node at the same time, there will be explicit resource calculation formulas to ensure that the respective reserved resources not be preempted.

During the pod deployment process, if DRL-Scheduler finds that the current node has not pulled the corresponding image data from the storage center, it will use api opened by the client to arrange the corresponding node to perform the algorithm pulling operation. The batch or single pod deployment process can be significantly accelerated if the corresponding physical machine has pre-pulled images.

6 Experiment

6.1 Cluster Deployment Dataset and Simulation Components

6.1.1 Architecture of cluster deployment simulation components

The cluster deployment simulation component is an independent module specially developed to further analyze the performance of the deployment algorithm model for batch deployment of long-

running applications. Using policy gradient training of neural networks requires DRL agents to interact with the environment frequently, which is time-consuming for pod scheduling. In a real system, it takes at least a few minutes to deploy a container and measure its performance. Since DRL training typically requires tens of thousands of iterations to complete, having the scheduler interact directly with the actual cluster is too slow and impractical. Similar to previous work, the platform develops a high-fidelity cluster environment simulator based on the core code of DRL-Scheduler, which can predict container performance at a given location. This enables learning through simulated experiments instead of deploying containers in real clusters.

The cluster deployment simulation component does not model low-level resource disturbances based on contention on CPU cache or memory bandwidth, because it is not realistic to obtain relevant information in real production. Instead, it moves to high-level performance metrics such as pod throughput, request latency, and GPU memory usage. Then it predicts how these metrics change under different container deployment scenarios. In particular, the co-location vectors of machine-level containers and the observed RPS/latency of each resident pod are recorded and used as training samples for the simulator. The simulation simulator uses the multivariate random forest as the main regressor to characterize the container interactions. RF methods use combinatorial decision trees to perform regression tasks that can make accurate predictions with a small amount of training data. It also accommodates overfitting when a large number of replicates are provided.

Differing from the interactive model of the DRL-Scheduler in actual deployment, the simulation experiment adopts the cluster deployment simulation component for model testing. In the real environment, request is directly sent to BaseManager and DRL-Scheduler calls the deployment algorithm. Then the actual deployment is carried out through StorageShell. The load information such as node GPU is maintained by BaseManager, as shown in Figure 3. In the simulation environment, DRL-Scheduler and StorageShell are not used for actual deployment, but the simulation components are deployed in clusters. This component simulates actual deployment and interacts with the RF environment simulation simulator.

DRL-Scheduler and cluster deployment simulation components are completely consistent, thus ensuring the complete unification of related data formats. In addition, the loading class DRLLoading of the management model is completely consistent with the actual production environment. That is, DRL-Scheduler and Kubernetes in Figure 3 are completely replaced by the cluster deployment simulation component and RF environment simulation simulator. The cluster deployment simulation component will send the corresponding PodsArrange request to the environment simulator, then receive and record the corresponding PodsState information.

It follows that the deployment algorithm DRL model relies heavily on the feedback and prediction of the cluster deployment simulation components. This means that it needs to guarantee that the environment simulation simulator which deploys simulation components in a cluster under given conditions can accurately give the current state of the cluster after given actions.

6.1.2 Training and accuracy of RF environment simulation simulator

The simulator is essentially a multivariate random forest regression model (RF) consisting of an ensemble of 100 decision trees with a maximum depth of 20. To analyze the accuracy of the simulator, a set of training samples is set specially. A total of 924 different positioning combinations of the LRAs task set of size 6 are collected on a machine corresponding to pod requests per second (Request Per-Second, RPS) information, and the mean square error (Mean-Square Error, MSE) information is used to calculate its accuracy. For evaluating the accuracy of the decision tree model more comprehensively, 1%-90% of the training dataset was randomly sampled, and the rest of the data was generated into the test subset.

The specific service subjects include a Redis service image, an object storage service image developed in Java, a text recognition algorithm image, an inference expression algorithm image that introduces external knowledge, a target detection algorithm image, and a human posture key point detection algorithm image. Among them, all algorithms need to interact with the pods corresponding to the object storage service and the Redis service, so when the scheduling player is used to return the request algorithm pod, Redis and the object storage service will generate additional requests

processing.

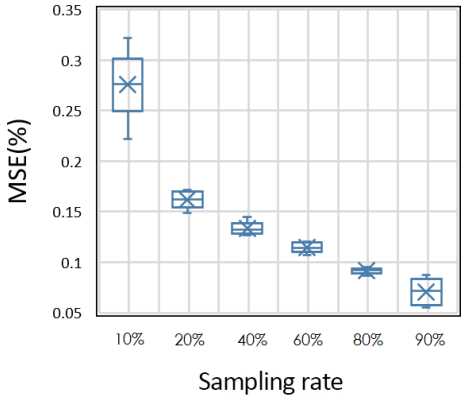


Figure 6: Prediction accuracy

Figure 6-9 shows the prediction accuracy of the test samples with different sampling ratios. The sampling ratio of more than 40% can guarantee MSE $<$ 0.145%, which basically guarantees the accuracy requirements of the RF environment simulation simulator. With the current batch scheduling task of the order of magnitude 6, it takes about 1-2 days to generate a sufficient localization combined dataset for training an environmental simulator. The order of magnitude of this platform in actual deployment is generally 4-5, and the corresponding acquisition time is mostly maintained within 10 hours.

6.1.3 LRAs dataset of cluster deployment

To train DRL model to deploy LRAs, we used the deployment records of different pod scales in the LRAs task presented in the previous subsection submitted in 8 months as samples. Some pod applications of LRAs in the sample are closely related. In addition to recording deployment information, it also includes invocation information within a specified time limit. Finally, the recorded deployment co-location situation is redeployed in the specified environment, and compared with the actual RPS situation after the DRL model is deployed to more accurately analyze the performance improvement. In the actual test of LRAs, in addition to four mirroring algorithms using GPU resources, mirroring of Redis and object storage services is also added. This part of the image can still be used as a pod with 0 GPU consumption and deployed using open-source components. The pod deployment of all recorded LRAs is mounted with large files, which are uniformly mapped according to *package.json*. If there is no file decoupling, directly mark the corresponding field as empty. For test comparison, the test environment is set to 9 nodes to build the test environment. After the related algorithm image is deployed according to the pod, the client that sends the simulated request will send the request according to the recorded request method, and then generate the RPS that uses the RPS of the pod running on a single node for normalization processing. Sending simulated requests per pod client was tested using a rate of 4 requests per second. The four algorithm images will be deployed with multiple copies at the same time in the recorded actual deployment, and will be deployed according to the actual situation in the actual test. If the distribution record of the message queue is used, the complexity of the test will be greatly increased, so the distribution simulation of the message queue is not carried out in the cluster deployment test. The average upper limit of GPU resource usage for the images of the four mirroring algorithms is 1460MiB, and the average call delay is 0.7s. The call delay distribution is shown in Figure 7.

6.2 Experimental verification

The actual number of nodes deployed on the platform is generally 3-9. According to the statistics of batch deployment scheduling task records, the situations of manual deployment and DRL model deployment are compared in 9 nodes. For further analysis, the 9-node deployment records in 8 months are randomly divided into 10 groups of batch deployment tasks, and the pod size of each deployment

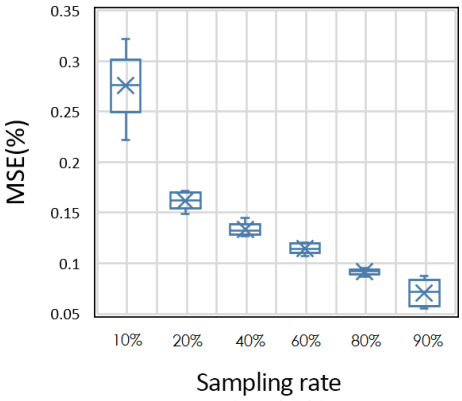


Figure 7: Call Delay CDF

task is 15-48. Each deployment task will generate the corresponding test results as the baseline according to the records of manual deployment under the original constraints. During the training process, the *package.json* file corresponding to the algorithm image needs to be loaded, because information such as the upper limit of the video memory of the algorithm image is required. Taking the GPU memory usage as an example, the size of the video memory will be directly judged in the RF environment emulator, and then a reward will be given. The video memory cap is a very important explicit constraint during pod deployment calls to avoid bad memory allocations. The training curve for reinforcement learning is shown in Figure 8.

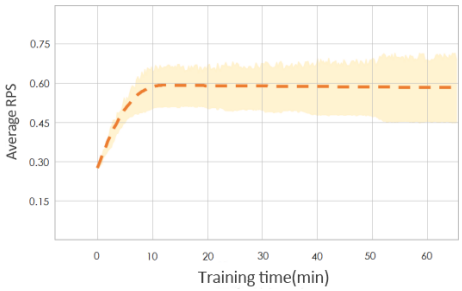


Figure 8: DRL Model 9 Node Learning Curve

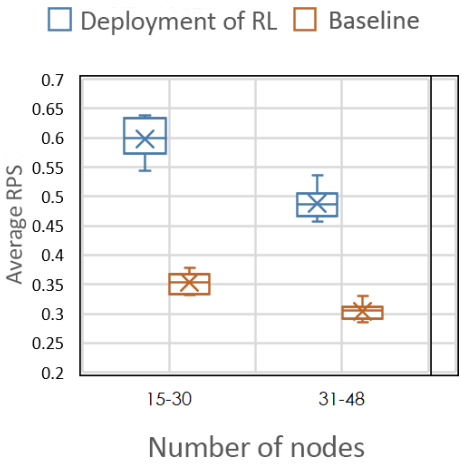


Figure 9: Average RPS for different deployment levels

Since the order of pods deployed in batches in the data is between 15 and 48, in order to better analyze the RPS impact of pods of different orders of magnitude, the interval is divided into two. Figure 9 compares the average RPS of different pod levels. Compared with the average RPS of the baseline, the DRL deployment algorithm is significantly improved, and the overall performance is

improved by 56.2%. This shows that the manual deployment scheme cannot consider the complex interaction between containers from a global perspective under constraints, and only meets the basic configuration requirements, which affects the overall pod performance.

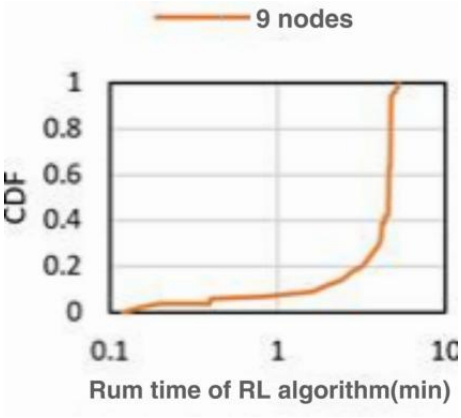


Figure 10: DRL deployment time distribution

The data set does not record the time to manually generate the deployment plan using only constrained dependencies and greedy strategies. According to experience, the time consumed by a single deployment plan is about 1 hour. Figure 10 shows the time-consuming of the DRL deployment algorithm, which is basically less than 7.5 minutes. Overall, the DRL algorithm improves the overall deployment efficiency, and the additional computing overhead brought by its deployment is moderate.

7 Data Availability

No data were used in the study.

8 Conflicts of Interest

The authors declare that they have no conflicts of interest.

9 Acknowledgments

This work was supported by the Aero Science Foundation of China under Grant Number 202051053002(corresponding author: Lei Deng). Lei Deng is with the School of Computer Science, Northwestern Polytechnical University, Xi'an 710129, China, and Lei Deng is also with the Science and Technology on Electro-optic Control Laboratory(e-mail: denglei@ nwpu.edu.cn).

References

[1] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, 2013, pp. 423–438.

[2] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: a system for large-scale machine learning." in *Osdi*, vol. 16, no. 2016. Savannah, GA, USA, 2016, pp. 265–283.

[3] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen *et al.*, "Mllib: Machine learning in apache spark," *The Journal of Machine Learning Research*, vol. 17, no. 1, pp. 1235–1241, 2016.

[4] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A low-latency online prediction serving system." in *NSDI*, vol. 17, 2017, pp. 613–627.

[5] H. Tian, M. Yu, and W. Wang, "Continuum: A platform for cost-aware, low-latency continual learning," in *Proceedings of the ACM Symposium on Cloud Computing*, 2018, pp. 26–40.

[6] W. Park and K. K. Seo, "A study on cloud-based software marketing strategies using cloud marketplace," *Journal of Logistics, Informatics and Service Science*, vol. 7, no. 2, pp. 1–13, 2020.

[7] A. Gujarati, S. Elnikety, Y. He, K. S. McKinley, and B. B. Brandenburg, "Swayam: distributed autoscaling to meet slas of machine learning inference services with resource efficiency," in *Proceedings of the 18th ACM/IFIP/USENIX middleware conference*, 2017, pp. 109–120.

[8] W. Shi, P. Zou *et al.*, "Research on cloud enterprise resource integration and scheduling technology based on mixed set programming," *Tehnički vjesnik*, vol. 28, no. 6, pp. 2027–2035, 2021.

[9] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi *et al.*, "Spark sql: Relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, 2015, pp. 1383–1394.

[10] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, 2012, pp. 15–28.

[11] P. Garefalakis, K. Karanasos, P. Pietzuch, A. Suresh, and S. Rao, "Medea: scheduling of long running applications in shared production clusters," in *Proceedings of the thirteenth EuroSys conference*, 2018, pp. 1–13.

[12] J. Guo, Z. Chang, S. Wang, H. Ding, Y. Feng, L. Mao, and Y. Bao, "Who limits the resource efficiency of my datacenter: An analysis of alibaba datacenter traces," in *Proceedings of the International Symposium on Quality of Service*, 2019, pp. 1–10.

[13] Q. Liu and Z. Yu, "The elasticity and plasticity in semi-containerized co-locating cloud workload: a view from alibaba trace," in *Proceedings of the ACM Symposium on Cloud Computing*, 2018, pp. 347–360.

[14] H. Wu, W. Zhang, Y. Xu, H. Xiang, T. Huang, H. Ding, and Z. Zhang, "Aladdin: Optimized maximum flow management for shared production clusters," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2019, pp. 696–707.

[15] Y.-J. Kim and B.-C. Ha, "Logistics service supply chain model," *Journal of Logistics, Informatics and Service Science*, vol. 9, no. 3, pp. 284–300, 2022.

[16] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*, 2013, pp. 1–16.

[17] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proceedings of the Tenth European Conference on Computer Systems*, 2015, pp. 1–17.

[18] C. Delimitrou and C. Kozyrakis, "Paragon: Qos-aware scheduling for heterogeneous datacenters," *ACM SIGPLAN Notices*, vol. 48, no. 4, pp. 77–88, 2013.

[19] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and qos-aware cluster management," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 127–144, 2014.

[20] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3, pp. 607–618, 2013.

[21] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015, pp. 450–462.

[22] Y. Yuan and H. Xu, "Multiobjective flexible job shop scheduling using memetic algorithms," *IEEE Transactions on Automation Science and Engineering*, vol. 12, no. 1, pp. 336–353, 2013.

[23] D. Novaković, N. Vasić, S. Novaković, D. Kostić, and R. Bianchini, "Deepdive: Transparently identifying and managing performance interference in virtualized environments," in *2013 {USENIX} Annual Technical Conference ({USENIX}{ATC} 13)*, 2013, pp. 219–230.

[24] A. Tumanov, T. Zhu, J. W. Park, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger, "Tetrisched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters," in *Proceedings of the Eleventh European Conference on Computer Systems*, 2016, pp. 1–16.

[25] J. Li, H. Shi, and K.-S. Hwang, "Using fuzzy logic to learn abstract policies in large-scale multiagent reinforcement learning," *IEEE Transactions on Fuzzy Systems*, vol. 30, no. 12, pp. 5211–5224, 2022.

[26] H. Shi, J. Li, J. Mao, and K.-S. Hwang, "Lateral transfer learning for multiagent reinforcement learning," *IEEE Transactions on Cybernetics*, 2021.

[27] Y. Kwak, W. J. Yun, S. Jung, J.-K. Kim, and J. Kim, "Introduction to quantum reinforcement learning: Theory and pennylane-based implementation," in *2021 International Conference on Information and Communication Technology Convergence (ICTC)*. IEEE, 2021, pp. 416–420.

[28] L. Li, J. Chen, and W. Yan, "A particle swarm optimization-based container scheduling algorithm of docker platform," in *Proceedings of the 4th International Conference on Communication and Information Processing*, 2018, pp. 12–17.

[29] D. Zhang, B.-H. Yan, Z. Feng, C. Zhang, and Y.-X. Wang, "Container oriented job scheduling using linear programming model," in *2017 3rd International Conference on Information Management (ICIM)*. IEEE, 2017, pp. 174–180.

[30] B. Liu, P. Li, W. Lin, N. Shu, Y. Li, and V. Chang, "A new container scheduling algorithm based on multi-objective optimization," *Soft Computing*, vol. 22, pp. 7741–7752, 2018.

[31] D. G. Lago, E. R. Madeira, and D. Medhi, "Energy-aware virtual machine scheduling on data centers with heterogeneous bandwidths," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 1, pp. 83–98, 2017.

[32] C. Kaewkasi and K. Chuenmuneewong, "Improvement of container scheduling for docker using ant colony optimization," in *2017 9th international conference on knowledge and smart technology (KST)*. IEEE, 2017, pp. 254–259.

[33] M. K. Hussein, M. H. Mousa, and M. A. Alqarni, "A placement architecture for a container as a service (caas) in a cloud environment," *Journal of Cloud Computing*, vol. 8, pp. 1–15, 2019.

[34] D.-K. Kang, G.-B. Choi, S.-H. Kim, I.-S. Hwang, and C.-H. Youn, "Workload-aware resource management for energy efficient heterogeneous docker containers," in *2016 IEEE Region 10 Conference (TENCON)*. IEEE, 2016, pp. 2428–2431.

[35] X. Xu, W. Wang, T. Wu, W. Dou, and S. Yu, "A virtual machine scheduling method for tradeoffs between energy and performance in cloud environment," in *2016 International Conference on Advanced Cloud and Big Data (CBD)*. IEEE, 2016, pp. 246–251.

[36] B. Shojaiemehr, A. M. Rahmani, and N. N. Qader, "Cloud computing service negotiation: A systematic review," *Computer Standards & Interfaces*, vol. 55, pp. 196–206, 2018.

[37] F. Chen, X. Zhou, and C. Shi, "The container deployment strategy based on stable matching," in *2019 IEEE 4th International Conference on Cloud Computing and Big Data Analysis (ICCCBDA)*. IEEE, 2019, pp. 215–221.

[38] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE cloud computing*, vol. 1, no. 3, pp. 81–84, 2014.

[39] H. V. Netto, A. F. Luiz, M. Correia, L. de Oliveira Rech, and C. P. Oliveira, "Koordinator: A service approach for replicating docker containers in kubernetes," in *2018 IEEE Symposium on Computers and Communications (ISCC)*. IEEE, 2018, pp. 00 058–00 063.

This journal is a member of, and subscribes to the principles of,
the Committee on Publication Ethics (COPE).
https://publicationethics.org/members/international-journal-computers-communications-and-control

Cite this paper as:

Deng, L.; Wang, Z. Y.; Sun, H.Y.; Li, B.; Yang, X. (2023). A deep reinforcement learning-based optimization method for long-running applications container deployment, *International Journal of Computers Communications & Control*, 18(4), 5013, 2023.
https://doi.org/10.15837/ijccc.2023.4.5013