Fast DRL-based scheduler configuration tuning for reducing tail latency in edge-cloud jobs

Wen, Shilin; Han, Rui; Liu, Chi Harold; Chen, Lydia Y.

DOI
10.1186/s13677-023-00465-z

Publication date
2023

Document Version
Final published version

Published in
Journal of Cloud Computing

Journal of Cloud Computing:
Advances, Systems and Applications

RESEARCH

Open Access

Fast DRL-based scheduler configuration tuning for reducing tail latency in edge-cloud jobs

Shilin Wen[1], Rui Han[1*], Chi Harold Liu[1] and Lydia Y. Chen[2]

Abstract

Edge-cloud applications are rapidly prevailing in recent years and pose the challenge of using both resource-strenuous edge devices and elastic cloud resources under dynamic workloads. Efficient resource allocation on edge-cloud jobs via cluster schedulers (e.g. Kubernetes/Volcano scheduler) is essential to guarantee their performance, e.g. tail latency, and such allocation is sensitive to scheduler configurations such as applied scheduling algorithms and task restart/discard policy. Deep reinforcement learning (DRL) is increasingly applied to optimize scheduling decisions. However, DRL faces the conundrum of achieving high rewards at a dauntingly long training time (e.g. hours or days), making it difficult to tune the scheduler configurations online in accordance to dynamically changing edge-cloud workloads and resources. For such an issue, this paper proposes EdgeTuner, a fast scheduler configuration tuning approach that efficiently leverages DRL to reduce tail latency of edge-cloud jobs. The enabling feature of EdgeTuner is to effectively simulate the execution of edge-cloud jobs under different scheduler configurations and thus quickly estimate these configurations' influence on job performance. The simulation results allow EdgeTuner to timely train a DRL agent in order to properly tune scheduler configurations in dynamic edge-cloud environment. We implement EdgeTuner in both Kubernetes and Volcano schedulers and extensively evaluate it on real workloads driven by Alibaba production traces. Our results show that EdgeTuner outperforms prevailing scheduling algorithms by achieving much lower tail latency while accelerating DRL training speed by an average of 151.63x.

Keywords Edge-cloud jobs, Tail latency, Scheduler configurations, DRL, Kubernetes and Volcano

Introduction

With the fast development of Internet of Things (IoT), traditional cloud-based applications suffer from high transmission latency due to large data volume and limited bandwidth. On the other hand, edge computing provides quick response and protects data privacy via local data processing, but has limited computational resources to execute expensive vision and machine learning tasks [20, 21, 24, 27, 43, 58]. Increasing numbers of IoT applications, therefore, apply a new paradigm that processes a job's tasks on both cloud and edge nodes [28, 69, 80]. This paper studies two typical types of edge-cloud jobs:

- Directed acyclic graph (DAG) jobs, whose tasks have sequential dependence. Examples include distributed data processing [4, 18, 23], face/fingerprint recognition [52, 53], and image classification [36, 55, 56, 62]. In particular, *MapReduce* jobs are a representative type of DAG jobs. For a typical *MapReduce* job, it usually consists of a three-step execution process: (1) a *start-up* task slices the input data set into multiple separate chunks; (2) many parallel *Map* tasks process them; (3) after all *Map* tasks are completed,

*Correspondence:
Rui Han
hanrui@bit.edu.cn
[1] School of Computer Science and Technology, Beijing Institute of Technology, Beijing, China
[2] TU Delft, Delft, Netherlands

the results will be sent to the *Reduce* task for final processing. So we can see that for a *MapReduce* job, subsequent tasks require the completion of previous tasks before they can start (that is, its tasks are executed sequentially).

- Artificial intelligence (AI) jobs, which usually have a large number of concurrent tasks. Examples include smart home applications [42, 82], smart healthcare [46], anomaly detection [81], object recognition [37], autonomous driving services [8], and intelligent photo management [68]. In particular, machine learning jobs that are submitted in *TensorFlow* [1] framework are a representative type of AI jobs. For a *TensorFlow* job, it usually consists of multiple (e.g. 50) tasks that execute concurrently. Since there is no sequential execution dependency between these tasks, they will not affect each other during execution.

When processing the above workloads on edge and cloud nodes [28, 69, 80], proper scheduling of their tasks is critical to their performance. In particular, a small proportion (e.g. 1%) of straggling tasks, called tail latency, decide a job's response time.

Example. Figure 1 shows an example scenario using in a Kubernetes cluster. The Volcano scheduler [10] allocates a list of jobs to edge and cloud nodes under different scheduler configurations. Specifically, the scheduling algorithm (e.g. *GANG_LRP*, *GANG_MRP* and *GANG_BRA*) decides how resources are allocated to tasks of these jobs. In addition, tail latency mitigation policy (task resubmitting policy (*TRP*) or task discard policy (*TDP*)) is another type of scheduler configuration, which is used to control which tasks (that cause the long tail latencies) need to be restarted or discarded. This example shows that for the same workload, when using six different configuration combinations (e.g. c1, c2, c3, c4, c5 and c6), the scheduler results in considerably different tail latencies. Note that, due to the tail latency determines the response time of a job, when this value is larger, the job completion time is correspondingly larger. Hence, in order to make the running workload complete faster, we can achieve this by selecting the one with lower tail latency from different scheduling configuration combinations. For example, in Fig. 1, c1 is a scheduling configuration combination with lower tail latency, thus we can select c1 to obtain lower job completion time.

Configuration sensitive schedulers propose two key challenges in practice. First, real cluster schedulers have massive optional configurations, including dozens of scheduling algorithms (e.g. 11 Kubernetes scheduling algorithms and 13 Volcano scheduling algorithms) and settings of tail latency mitigation policies (e.g. different values of *TRP* and *TDP*). The combination of these configurations constructs a huge search space. Second, at run-time, jobs of different characteristics arrive continuously and most jobs last for a few seconds to minutes. Moreover, in a resource-sharing environment, the available resources in edge and cloud nodes also dynamically change.

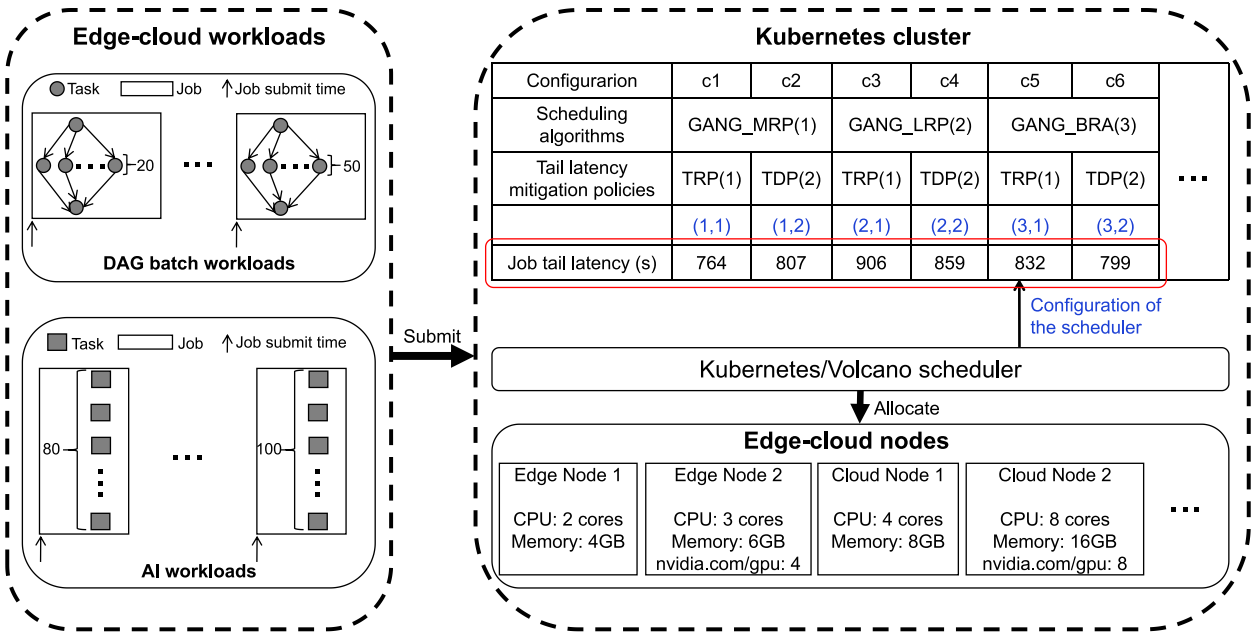


Fig. 1 Edge-cloud workloads scheduling through Kubernetes cluster schedulers

Recently, deep reinforcement learning (DRL), which is an important extension of the traditional reinforcement learning (RL) method, has been applied to various sophisticated online optimization problems with large solution spaces [40, 44, 63]. However, in real cluster scheduling, applying this technique to tune scheduler configurations requires a dauntingly large number of data samples for training DRL agents. Moreover, due to randomly arrival jobs, various resource demands of tasks, and elastic node resources, when constructing time-variant states in a DRL environment, a training sample needs rather *long time* (e.g. 10 seconds) to generate. And the DRL training may also need millions of samples to converge. Therefore, the **time-consuming sampling phase** (may take dozens of hours) is the bottleneck of the whole DRL training process. Note that *at run-time, an outdated DRL agent may lead to significant deviations from the optimal configuration combination and incur job performance degradation*. Some techniques have been proposed to support the DRL training in an offline fashion through developing simulation platforms [54, 75, 76]. But most of them have such limitations: (1)they only consider these *long-running jobs* in high performance computing (HPC) data centers; (2)in their considered scheduling scenarios, these long-running jobs are allocated to proper servers according to **a fixed job scheduling algorithm**; (3)in DRL training, they implicitly assume **fixed available resources** in the cluster.

In this paper, we propose EdgeTuner, an online approach that effectively uses a DRL agent to select scheduler configurations for edge-cloud jobs. To overcome the expensive training overhead, we develop a cluster simulator to emulate the volatile and complex state space of edge-cloud jobs (e.g. their task dependance, tasks' resource demands, and cluster resource utilization). The simulator effectively captures the on-line adaption across different scheduling algorithms as well as the dynamicity of edge-cloud jobs. As such, the training of DRL agent can be effectively and quickly converged via this offline simulation phase. Note that EdgeTuner differs from traditional hyper-heuristic approaches that find an optimal scheduling algorithm for pre-specified cloud workflow [73] or batch jobs [61, 64]. This is because EdgeTuner needs no prior knowledge about the jobs to be scheduled and provides fast scheduling algorithm tuning for continuously arrival jobs in the cluster.

In this paper, our contributions are mainly as follows:

▹ **Complex Edge-Cloud job scheduling modelling**. We formulate the tuning of scheduling algorithms for edge-cloud jobs as a sequential decision making process (MDP) to leverage the DRL technique. To incorporate various scheduling scenarios, we define general *state* representation of complex nodes and workloads, use *action* to reflect optional scheduling algorithms, and define *reward* function to estimate job tail latency.

▹ **DRL training acceleration**. We develop a cluster simulator to emulates a scheduling algorithm's resource allocation mechanism and its influence factors, including available resources, and waiting and running tasks. At each scheduling interval, the simulator takes the agent's state and action as inputs and outputs the reward instantly (this reward production process takes at least a few seconds in real clusters). The training can be performed offline by directly interacting with the simulator in the usually adopted online learning scheme.

▹ **Implementation and evaluation**. We incorporate our controller on the popular Kubernetes and Volcano schedulers and evaluate both schedulers using workloads driven from the Alibaba cluster trace [2, 3]. The extensive comparative evaluations against latest Kubernetes and Volcano scheduling algorithms show: (i) by applying DRL agents in such various scenarios of dynamic workloads and resources, EdgeTuner outperforms baselines by achieving an average of 21.66% reductions in tail latencies; (ii) EdgeTuner accelerates the whole DRL training speeds by an average of 151.63x.

The remainder of this paper is organized as follows: Section "Background" introduces the background, and Section "Problem Formulation" formulates the problem. Section "EdgeTuner" explains our approach, and Section "Evaluation" evaluates it. Section "Related Work" introduces the related work, and finally, Sections "Conclusion" summarizes the work.

Background

Heterogeneous workloads and machines in edge-cloud collaboration scenarios

Heterogeneous workloads

Although **cloud computing** has strong computing processing capabilities, its resources are relatively concentrated and the distribution of computing centers is relatively sparse. This leads to large latencies in real-time communication with users. On the contrary, since the edge nodes are deployed close to the base stations (that is, the communication cost with users is very low), **edge computing** can better handle latency-sensitive tasks compared to cloud computing. Therefore, in practical edge-cloud environment, edge-cloud nodes often need to cooperate to handle diverse workloads. Specially, some edge-cloud applications in traditional DAG-based systems generate a lot of DAG jobs (e.g. distributed computing [4, 18], face/fingerprint recognition [52, 53], and big data classification [36, 56, 62]). The other edge-cloud applications for executing AI tasks generate a large number of AI jobs (e.g. smart service and control applications [8, 37, 42, 46, 68, 81, 82]).

In this paper, we mainly consider two edge-cloud workloads: *DAG batch workloads* and *AI workloads*. And based on the real cluster trace provided by Alibaba [2, 3], we analyze the different characteristics of these two workloads, respectively.

Trace analysis for DAG batch workloads. We study the real cluster-trace-v2018 [2], which mainly records the operation of offline batch workloads in the Alibaba mixed CPU cluster with 4,034 machines running in consecutive 8 days. From the trace, we can find that a batch job usually contains multiple tasks, and most tasks have DAG dependencies. Each task also usually consists of several instances (that is, an instance is the smallest unit of batch job scheduling), these instances will execute the same application code and request the same resources, but their input data is different. In detail, we analyze the trace from four aspects: *job and task allocation*, *instance completion time*, *arrival pattern of jobs and tasks*, and *resource utilization*.

- *Job and task allocation.* In particular, we analyze the distribution of task number for each job in Fig. 2(a), and the results show that most jobs contain less than 150 tasks.
- *Instance completion time.* Figure 2(b) shows the CDF distribution of the completion time for all instances, and the 80%, 90%, and 99% of the instance completion time are 58*s*, 177*s*, and 828*s*, respectively.
- *Arrival pattern of jobs and tasks.* Figure 2(c) shows the arrival frequency of jobs and tasks, and tasks have the highest arrival rate at 4:00am each day (the arrival rate is lower during the daytime). In other words, the task activity in this cluster follows a *Daytime-Night* pattern: the cluster will execute more tasks at night, while executing less tasks during the daytime.
- *Resource utilization.* Figure 2(d) shows the average resource usage of tasks. The results show that the CPU and Memory resources required by tasks are the largest from 24:00 (the day) to 6:00 (the next day), so the resource usage of all tasks is periodic. It verifies the *Daytime-Night* pattern that exists in the cluster.

Trace analysis for AI workloads. We study the real cluster-trace-gpu-v2020 [3], that records the workload information collected from Alibaba PAI (Artificial Intelligence Platform) with over 6,500 GPUs (about 1800 machines) in a month. From the trace, we can observe that user-submitted workloads consist of AI jobs from various machine learning frameworks, such as TensorFlow [1], PyTorch [50], Graph-Learn [74], RLlib [39]. Among these workloads, each job is composed of multiple different tasks running concurrently on many machines, and there is no DAG dependencies among these tasks. Similarly, we also analyze the trace from four aspects: *job and task allocation*, *job completion time*, *arrival pattern of jobs and tasks*, and *resource utilization*.

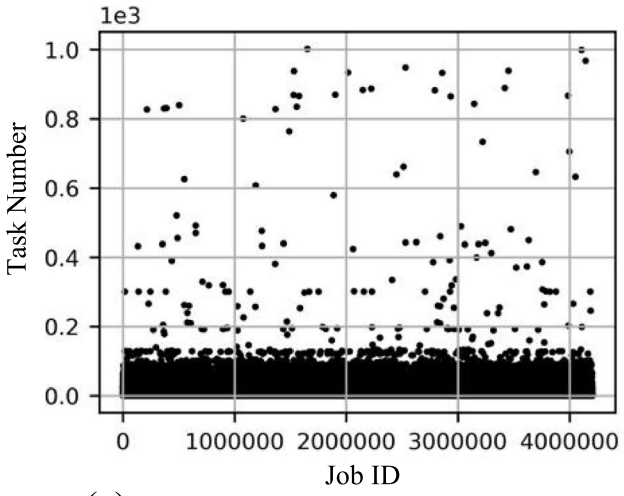
- *Job and task allocation.* In particular, we select the trace data of the first week as the basis for analysis. In Fig. 3(a), we analyze the CDF distribution of the task number concurrently executed by each AI job, and conclude: about 25.16% of jobs have less than 10 of concurrent tasks, about 80.48% of jobs have less than 50 of concurrent tasks, and about 92.76% of jobs have less than 100 of concurrent tasks.
- *Job completion time.* In Fig. 3(b), we analyze the CDF distribution of the completion time for all AI jobs, and conclude: about 48.51% of jobs run less than 10*min*, about 64.83% of jobs run less than 30*min*, about 73.86% of jobs run less than 60*min*, and about 80% jobs run less than 90*min*.
- *Arrival pattern of jobs and tasks.* Figure 3(c) shows the periodic characteristics between jobs and tasks, and most jobs and tasks are submitted from 11:00am to 14:00pm.
- *Resource utilization.* Figure 3(d) shows the average resource requests of tasks in the GPU cluster. The results show that the GPU, CPU and Memory resources required by tasks are the largest from 24:00 (the day) to 8:00 (the next day), so the resource usage of all tasks is also periodic. It also indicates that a *Daytime-Night* pattern also exists in the GPU cluster.

Heterogeneous machines

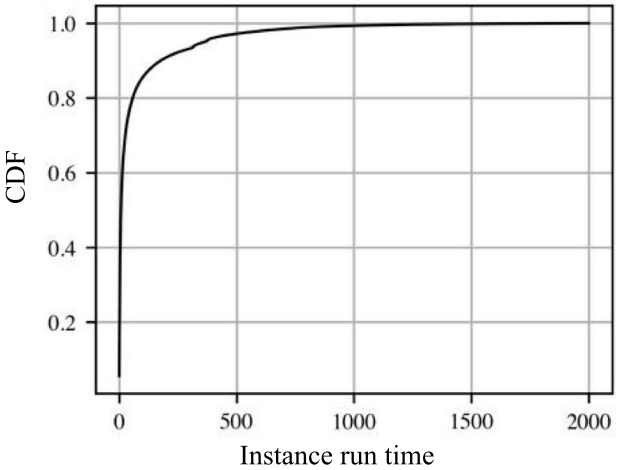
In edge-cloud environment, due to *heterogeneity of hardware architecture* and *resource sharing at run-time* of different machines in the cluster, this leads to these edge-cloud nodes' heterogeneity.

Heterogeneity of hardware architecture. For these edge-cloud nodes, they have different CPU architectures, such as X86 and ARM. And they have multiple GPU types, such as NVIDIA Jetson series, Raspberry Pi, and other NVIDIA GPUs. Therefore, different nodes usually have large performance variance (e.g. different processing speeds).

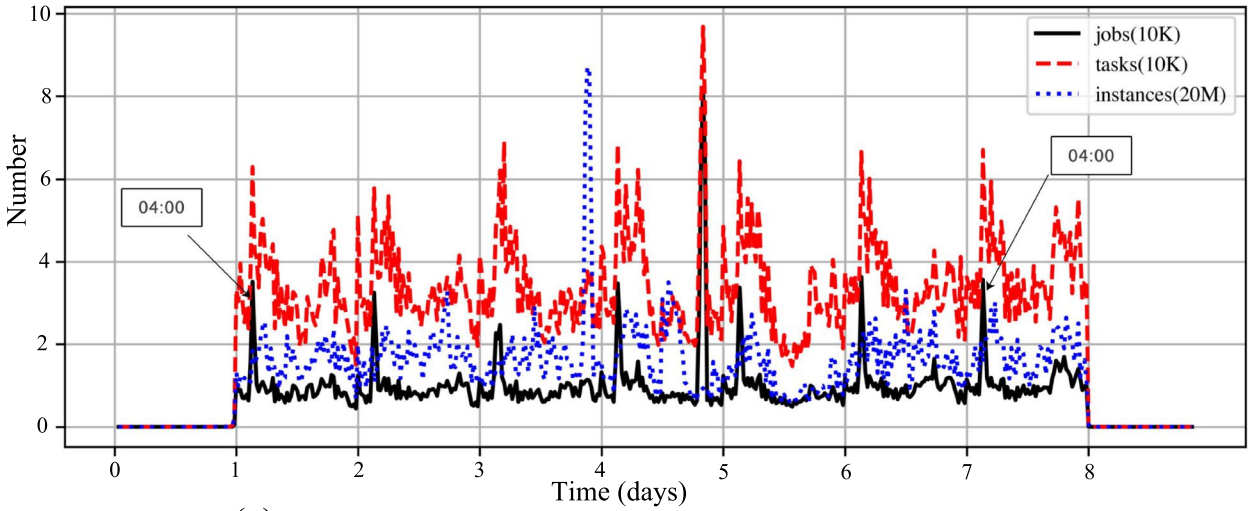
Resource sharing at run-time. When the cluster is at run-time, the resources of edge-cloud nodes are shared by all submitted jobs/tasks, leading to dynamic changes in available resources. At this time, the performance of different nodes will also be very different. For example, the performance of nodes with intense resource competition will be greatly affected.



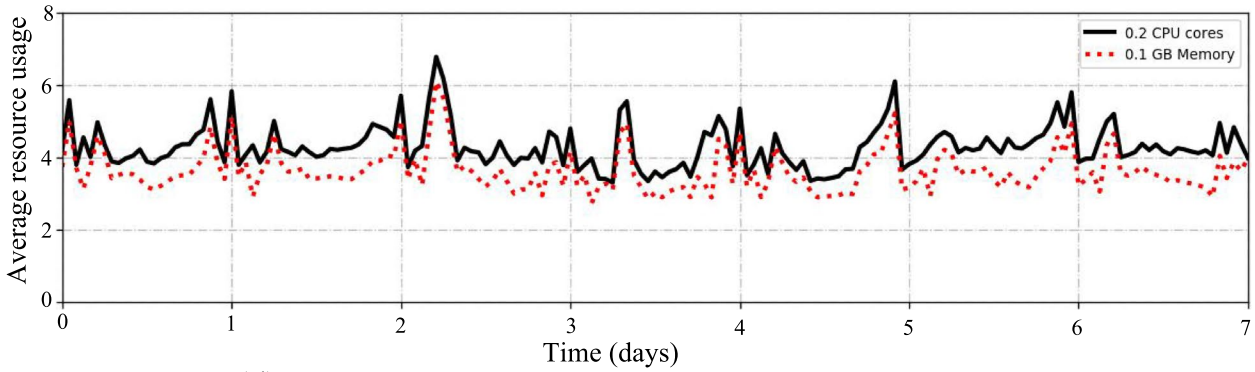
(a) The distribution of task number for each job



(b) CDF of instance completion time

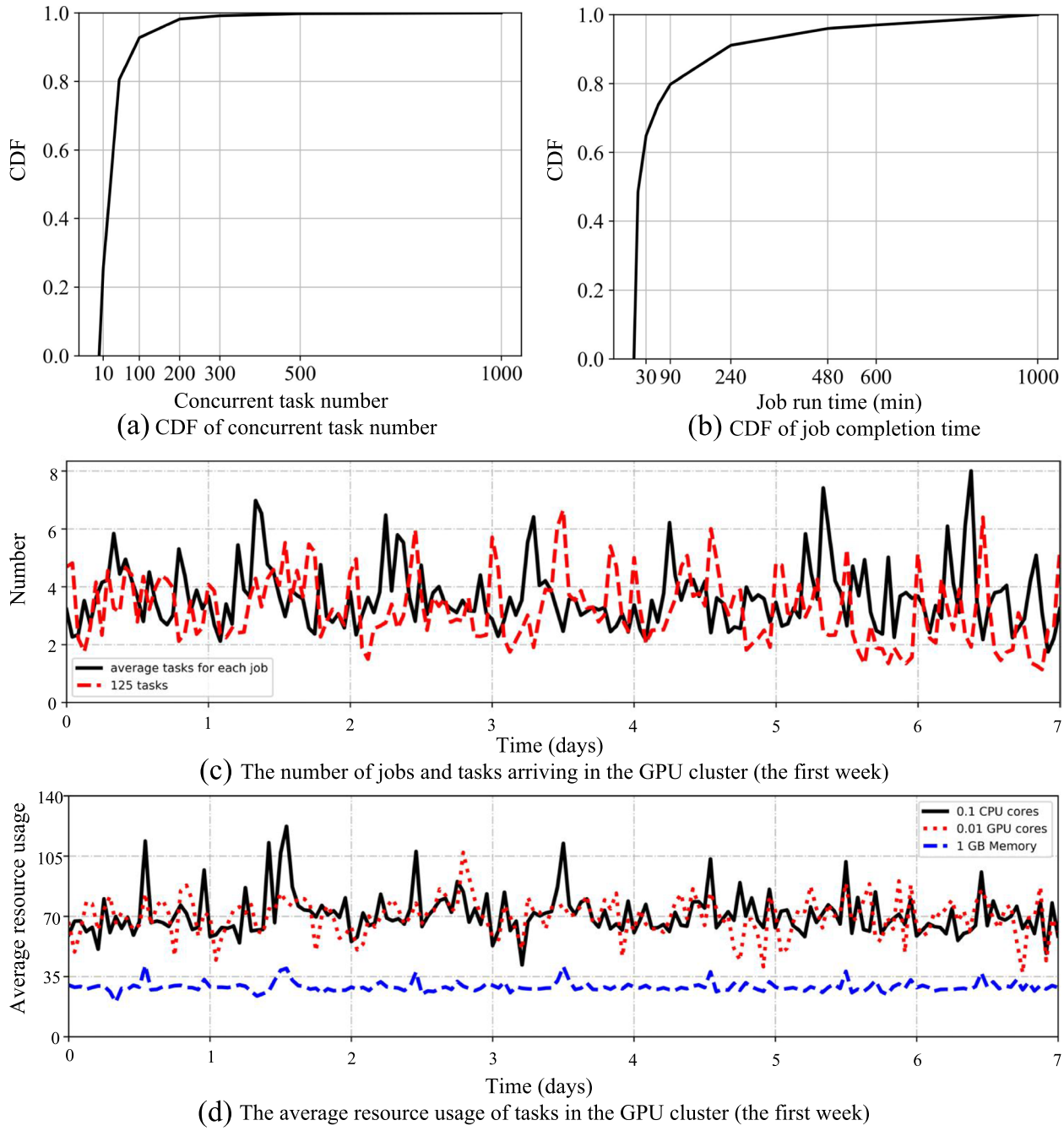


(c) The number of batch jobs and tasks arriving in the CPU cluster



(d) The average resource usage of batch tasks in the CPU cluster

Fig. 2 Alibaba cluster-trace-v2018 analysis for DAG batch workloads



(a) CDF of concurrent task number

(b) CDF of job completion time

(c) The number of jobs and tasks arriving in the GPU cluster (the first week)

(d) The average resource usage of tasks in the GPU cluster (the first week)

Fig. 3 Alibaba cluster-trace-gpu-v2020 analysis for AI workloads

Sensitive scheduler configurations of affecting tail latency

In real Kubernetes cluster, when scheduling edge-cloud workloads (such as DAG batch workloads and AI workloads), the tail latencies of jobs in such workloads are influenced by different scheduler configurations. In this paper, we mainly considered two categories: *scheduling algorithms* and *tail latency mitigation policies*.

Scheduling algorithms

In our proposed cluster simulator, we implement 11 Kuberentes scheduling algorithms, and 13 Volcano scheduling algorithms. In **Kubernetes scheduler** [19], there are three typical scheduling algorithms are: (1)*BRA*: this algorithm balances the utilization of CPU and memory resources in different nodes. (2)*LRP*: this algorithm

calculates the amount of resources and the number of tasks allocated to different nodes, and prefers to allocate tasks to nodes with more available resources. (3)*MRP*: this algorithm prefers to allocate tasks to nodes with less available resources, thus running the same tasks with the least number of nodes. And in **Volcano scheduler** [10], three typical scheduling algorithms are: (1)*GANG_LRP*: this algorithm means that first, only when the cluster resources meet the request of the minimum parallel tasks required by a job, the job can be scheduled (that is, *GANG* [71]); then it prefers to allocate tasks of the job to nodes with more available resources. (2)*GANG_MRP*: this algorithm aims that first, only when the cluster resources meet the request of the minimum parallel tasks required by a job, the job can be scheduled; then it prefers to allocate tasks of the job to nodes with less available resources. (3)*GANG_BRA*: this algorithm indicates that first, only when the cluster resources meet the request of the minimum parallel tasks required by a job, the job can be scheduled; then it prefers to allocate tasks of the job to nodes with more balanced resources. In addition, DRF [17], SLA [48] and BINPACK [7] are also two typical allocation algorithms for how jobs are scheduled in Volcano scheduler.

Tail latency mitigation policies

There have been some research works on reducing the tail latency for online concurrent service components, such as request retransmission techniques that produce accurate results [12, 35, 59] and request partial execution techniques that produce approximate results [12, 14, 29, 30, 35, 78]. In view of the above works, for edge-cloud workloads' scheduling scenarios, we develop two mitigation policies to reduce the tail latency.

- ***Task Restart Policy (TRP):*** this policy will restart a certain number of tasks that meet the conditions by controlling a restart ratio. First, we set a task restart time. Then if a task has not been started for a long time or the execution has not been completed before the restart time, the task will be immediately restarted and redistributed to such a node that processes the task faster in the cluster.
- ***Task Discard Policy (TDP):*** the policy will terminate a certain number of tasks with a long latency by controlling a discarding ratio. First, we set a task completion deadline. Then if a task has not been started for a long time or the execution has not been completed before the completion deadline, the task will be directly discarded.

Example of job scheduling using three Volcano scheduler configurations

Example. Figure 4 illustrates an example of allocating three tasks (of an AI job) to two nodes using three

Task	CPU request (cores)	CPU limit (cores)	Memory limit (GB)	Workload (CPU core × seconds)
V_1	2	4	1	80
V_2	3	4	4	120
V_3	4	6	1	720

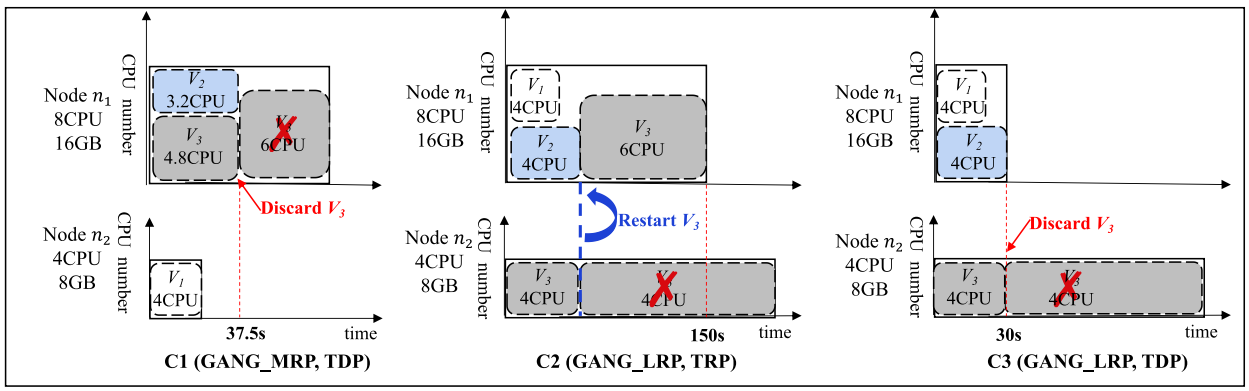
(a) Three tasks

Node	CPU capacity (cores)	Memory Limit (GB)
n_1	8	16
n_2	4	8

(b) Two nodes

Configuration	C1	C2	C3
Scheduling algorithm	GANG _MRP	GANG _LRP	GANG _LRP
Tail latency mitigation policy	TDP(✗)	TRP(↻)	TDP(✗)
	(GANG _MRP, TDP)	(GANG _LRP, TDP)	(GANG _LRP, TDP)

(c) Three scheduler configurations



(d) Tail latencies under three different scheduler configurations

Fig. 4 An example of job scheduling using three Volcano scheduler configurations

Volcano scheduler configurations. Here, the example settings include: (1)three tasks are submitted and executed at the same time; (2)when the tail latency mitigation policy is *TDP*, this AI job is considered for completion once at least two tasks in the job have completed.

Results. We can see that the job performance (that is, job tail latency) is determined by the slowest executing task, and it is influenced by three factors: (1)Different tasks with requiring different amounts of resources in Fig. 4(a). (2)Available resources of the cluster nodes in Fig. 4(b). (3)Scheduler configurations in Fig. 4(c). Among them, scheduling algorithms (such as *GANG_LRP* and *GANG_MRP*) first decide how tasks are allocated to nodes, and then tail latency mitigation policies (such as *TRP* or *TDP*) control that which tasks (that cause the long tail latencies) need to be restarted or discarded.

When using different scheduler configurations (e.g. $C1$, $C2$ and $C3$ in Volcano [10]), the job has considerably different tail latencies. Moreover, Fig. 4(d) shows that compared with the other two configurations, $C3$ achieves the lowest tail latency, because its mechanism can allocate the resources most efficiently for this specific scenario.

Problem formulation

Scenarios. In our considered edge-cloud collaboration scenarios, these edge-cloud jobs usually contain multiple concurrent tasks (e.g. 50 tasks). For the resource demands, we mainly consider CPU, GPU and memory. As shown in Section "Sensitive Scheduler configurations of Affecting Tail Latency", different scheduler configurations (including *scheduling algorithms* and *TDP/TRP*) can decide how tasks can be better executed concurrently in the cluster based on their resource need, so that the tail latencies of jobs are minimized. Here, we specifically consider scenarios that different configuration combinations can be dynamically chosen during the system runtime.

Modeling a RL problem. Recently, deep reinforcement learning (DRL), which is an important extension of the traditional reinforcement learning (RL) method, has been applied to various sophisticated online optimization problems with large solution spaces. Therefore, we model the tuning for scheduler configurations as a RL problem, in which an agent (tuner) learns to act (selecting a configuration combination of one scheduling algorithm and one possible tail latency mitigation policy) in an environment (cluster), in order to maximize a scalar reward signal [60]. At each discrete time-step (episode) $t = 0, 1, 2 \cdots$, the cluster provides the tuner with an observation s_t, the tuner responds by selecting a configuration combination a_t and obtains the feedback of reward $R(s_t, a_t)$, and next state s_{t+1} from the environment. This interaction is formalized within the framework of Markov Decision Process (MDP), which is a controlled stochastic process defined by the state space $\mathcal{S}$, action space $\mathcal{A}$, transition dynamics $0 \leq P(s_{t+1}|s_t, a_t) \leq 1$, and reward function $R(s_t, a_t)$.

State. The modeling of state considers two factors that determine the jobs' tail latencies at run-time: the resource utilizations in nodes and the waiting and running tasks in the cluster. In scheduling, waiting tasks are the targets that a configuration combination of cluster schedulers needs to manage together; running tasks occupy resources and then release them after completion. Formally, $s_t = (N, V^w, V^r)$ denotes the node and task state information during a scheduling interval.

- A node $n \in N$ is denoted as a 9-tuple $(cpu^u, cpu^r, cpu^c, mem^u, mem^r, mem^c, gpu^u, gpu^r, gpu^c)$ (1)cpu^u represents the actual usage of CPU cores; (2)cpu^r represents the *requested* (reserved by tasks) CPU cores; (3)cpu^c represents the capacity of CPU cores in the node; (4)mem^u represents the actual memory usage; (5)mem^r represents the requested memory by tasks; (6)mem^c represents the node's memory capacity; (7)gpu^u represents the actual usage of GPU number; (8)gpu^r represents the *requested* (reserved by tasks) GPU number; (9)gpu^c represents the capacity of GPU number in the node.
- A waiting task $v^w \in V^w$, it is denoted as a 7-tuple $(cpu^r, cpu^l, mem^r, mem^l, gpu^l, work, j^{id})$: (1)$cpu^r$ represents the requested CPU by the task;(2)cpu^l represents the CPU limit of the task; (3)mem^r represents the requested memory by the task; (4)mem^l represents the task's memory limit; (5)gpu^l represents the requested GPU number by the task; (6)$work$ represents the workload of task, e.g. workload 400 means the task needs 100 seconds to complete when running in 4 CPU cores; and (7)j^{id} is the job ID the task belongs to.
- A running task $v^r \in V^r$ is denoted as a 6-tuple $(work, node, cpu^l, gpu^l, et, j^{id})$: (1)$work$ represents the task's workload (that is, $execution\ time \times cpu^l$); (2)$node$ represents the node the task is allocated; (3)cpu^l represents the task's CPU limit; (4)gpu^l represents the task's GPU limit (Note that gpu^l and gpu^r are equal when a task uses GPU); (5)et represents the elapsed time when the task starts running; and (6) j^{id} is its job ID.

Dimensionality of state s_t. We note that in practical job scheduling, $|N|$ is the number of nodes in the cluster, and the numbers of waiting and running tasks continuously change at different time steps. Given that the scheduling interval is short (e.g. 15 seconds), such values do not increase the training complexity. We note that the state

space grows significantly with the number of job/task arrivals and the size of cluster.

Action. Given a scheduler, action a_t represents one possible configuration combination of cluster schedulers. Here, in order to express a_t more intuitively, we denote a 2-tuple (a_t^1, a_t^2): a_t^1 represents one possible scheduling algorithm, a_t^2 represents one possible tail latency mitigation policy (that is, TRP or TDP) that an agent can select.

Transition dynamics. In a MDP, transition dynamics $P(s_{t+1}|s_t, a_t)$ reflects the time-variant dynamics of cluster. Such dynamics are determined by three factors: the tasks $V_t^{allocate}$ that obtain resource allocations; the completed tasks $V_t^{complete}$ at time-step t; and the newly arrival jobs/tasks V_{t+1}^{arrive} at time-step t+1. We note that both $V_t^{allocate}$ and $V_t^{complete}$ are influenced by the scheduler algorithm set by action a_t, and they determine the three elements in state s_{t+1} at time step t+1:

$$V_{t+1}^{w} = V_t^{w} \setminus V_t^{allocate} \cup V_{t+1}^{arrive} \tag{1}$$

$$V_{t+1}^{r} = V_t^{r} \cup V_t^{allocate} \setminus V_t^{complete} \tag{2}$$

Reward. At a time-step t, a reward r_t denotes the job latency as JTL (denoted in [12, 22]) when using a configuration combination (a_t^1, a_t^2). Given that there are only a small number of jobs during a scheduling interval, we consider tail latencies of both jobs and their tasks to accelerate the convergency of RL training. Specifically, let J be the set of jobs completed within period $(t-1, t]$ and JTL_i be the tail latency of a job $j_i \in J$. At time-step t, the reward of job j_i is:

$$r_i^{job} = \alpha_1 * JTL_i|_t + \beta_1 \tag{3}$$

Similarly, let V^{run} be the set of tasks running within period $(t-1, t]$, and $TTL_i^{run}|_{\hat{t}}$ $(\hat{t} \in (t-1, t])$ be the tail latency at time $\hat{t}$ for a task $v_i^{run} \in V^{run}$. At time-step t, the reward of the set V^{run} is:

$$r_t^{run} = \alpha_2 * [\max(TTL_i^{run}|_t) - \max(TTL_i^{run}|_{t-1})] + \beta_2 \tag{4}$$

And let V^{wait} be the set of tasks waiting within period $(t-1, t]$, and $TTL_i^{wait}|_{\hat{t}}$ $(\hat{t} \in (t-1, t])$ be the waiting latency at time $\hat{t}$ for a task $v_i^{wait} \in V^{wait}$. At time-step t, the reward of the set V^{wait} is:

$$r_t^{wait} = \alpha_3 * \max(TTL_i^{wait}|_t) + \beta_3 \tag{5}$$

The reward r_t of time-step t is the summation of jobs, running tasks' and waiting tasks' rewards.

$$r_t = \sum_{i=1}^{|J|} r_i^{job} + r_t^{run} + r_t^{wait} \tag{6}$$

In RL training, we set negative values of α_1, α_2 and α_3, and positive values of β_1, β_2 and β_3 in Equations 3, 4 and 5. These settings ensure the reward is inversely proportional to the tail latencies of running and waiting tasks.

EdgeTuner

Design idea

Our work is proposed with two objectives.

1) Hot swapping scheduler configurations for dynamic workloads and resources. The core component of EdgeTuner, the *DRL-based agent*, is external to the cluster scheduler and just operates on its configuration combinations (consisting of one scheduling algorithm and one tail latency mitigation policy). This design ensures minimum modifications to the scheduler, and more importantly, making it possible to replace any of them at runtime without shutting down the system. Specifically, the agent observes the state (cluster status) periodically (e.g. 10 seconds) and selects a configuration combination for the cluster scheduler.

2) Simulator-based DRL training acceleration. We note that under *diverse workloads*, the whole training process needs a lot of experience (e.g. several million samples) to converge. However, in real job scheduling scenarios, the actor takes at least a few seconds to evaluate the effectiveness of an action (that is, the selection of a configuration combination) and obtains a sample from the environment (the cluster). Even using the latest DRL training techniques (e.g. IQN+Ape-X [63]), the training may take dozens of hours to complete due to the long sampling phase. Moreover, when the cluster resource changes, the training process needs to be re-executed and the long training time makes the DRL agent infeasible for online scheduler configuration tuning. Given this motivation, we develop a cluster simulator and use it as the environment for the actor. We explain how to train DRL agent under dynamic workloads and resources in Section "Simulator-based DRL Training".

Overview

Our cluster simulator is implemented using Golang and it can support different operating systems such as Linux and Max-Os. Figure 5 illustrates the implementation of cluster simulator for DRL training, and it consists of two major parts: ***Simulator-based DRL Training*** and ***Cluster Simulator***.

Simulator-based DRL training

Based on the Kubernetes framework, we incorporated the proposed approach with its two important schedulers (such as Kubernetes scheduler [19] and Volcano scheduler [10]). Similar to other mainstream resource negotiation systems, Kubernetes provides access to various

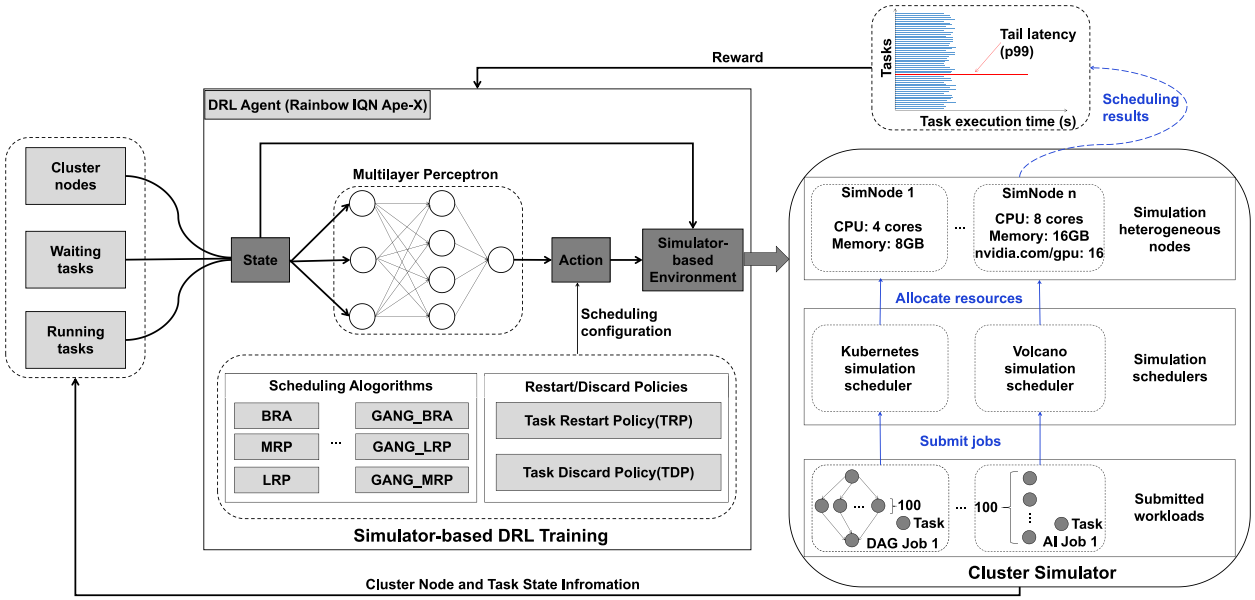


Fig. 5 Implementation of Cluster Simulator for DRL training

information regarding to resources, jobs, and scheduling constraints. When a job is submitted, Kubernetes also provides interfaces to obtain its submission time, resource demand, and task information. When an agent generates an action according to the above state information, it is pushed to the Kubernetes scheduler (or Volcano scheduler) that supports run-time adjustment of its configuration combinations.

Crucially, for the whole simulator-based DRL tarining process, there are four key modules.

State. At each sampling or training time-step t, this module receives the cluster nodes' and tasks' state information from the cluster simulator, and then constructs state s_t in the experience. In particular, this module has two main functions: (1)initializing the simulation environment, including initial node information, tasks waiting to be scheduled, and tasks that have been executed. (2)storing the state information during the training process.

Action. At each sampling or training time-step t, this module is responsible for accepting an action a_t from the agent, converting it into the corresponding configuration combination (a_t^1, a_t^2), and then forwarding it to the simulator-based environment. Note that the configuration combination is composed of one scheduling algorithms (such as *LRP*, *MRP*, *BRA*, *GANG_LRP*, *GANG_MRP* and *GANG_BRA*) and one tail latency mitigation policies (such as *TRP* and *TDP*).

Reward. At each sampling or training time-step t, this module receives the scheduling results (that is, 99th% quantile's tail latency of a edge-cloud job) from the cluster simulator, and then constructs reward r_t in the experience.

Simulator-based Environment. After the DRL agent generates an action according to the state information, this module will provide the action to the cluster simulator. In addition, this module is also responsible for providing these information to the cluster simulator, including simulation heterogeneous nodes' configuration and submitted workloads.

DRL agent for training

The DRL agent is the core of the whole simulator-based DRL tarining process. And the training of a DRL agent has two phases: (1)In the sampling phase, the actor collects experience training samples by interacting with the simulator. (2)The learning phase starts when a prespecified number of samples is collected. Similar to other simulation platforms [76, 77], our simulator is driven by workload traces and it can generate an experience sample instantly, thus considerably accelerating the sampling and training process.

Actor. Algorithm 1 details the steps of the actor. It first initializes the environment by obtaining the latest network parameters (line 1) and getting initial state from environment (line 2). Subsequently, it iteratively obtains samples and adds them to the replay memory (lines 3 to 14). At each iteration, the actor first selects an action a_{t-1} (that is, a configuration combination) and applies it to the environment (line 4). It then triggers environment. SimulateOneStep(s_{t-1} ,a_{t-1}) to obtain state s_t and reword r_t constructed

using information of nodes, jobs, and tasks in the environment (line 5). Finally, it gets episode termination signal (line 6) and adds the sample data to the local buffer (line 7). When the buffer size is larger than the maximal size B, the actor calculates priorities for the current experience and triggers the remote call to add experience to the replay memory (lines 8 to 12). The actor also periodically obtains the latest network parameters (line 13).

Require: B: the maximal size of local buffer;
T_s: the number of sampling steps.
1. $\theta_0 \leftarrow$ learner.Parameters();
2. $s_0 \leftarrow$ environment.InitialState();
3. **for** t = 1 to T_s **do**
4. $a_{t-1} \leftarrow \pi_{\theta_{t-1}}(s_{t-1})$;
5. $s_t, r_t \leftarrow$env.SimulateOneStep(s_{t-1}, a_{t-1});
6. $\gamma_t \leftarrow$ environment.HasDone();
7. localBuffer.Add($(s_{t-1}, a_{t-1}, r_t, \gamma_t)$);
8. **if** localBuffer.Size() $> B$ **then**
9. $\tau \leftarrow$ localBuffer.Get(B);
10. $p \leftarrow$ ComputePriorities(τ);
11. replay.Add(τ, p);
12. **end if**
13. Periodically($\theta_t \leftarrow$ learner.Parameters()).
14. **end for**

Algorithm 1 Actor

Require: N: the experience number to start training;
L: the maximal size of local buffer;
T_s: the maximum number of training.
1. $\theta \leftarrow$ environment.InitialState();
2. **while** not replayMemory.HasEnoughExperience(N) **do**
3. WaitAmoment();
4. **end while**
5. **for** t = 1 to T_s **do**
6. id, $\tau \leftarrow$ replay.SampleExperience();
7. $l \leftarrow$ GetLoss(τ, θ);
8. UpdateParameters(l, θ);
9. $p \leftarrow$ GetPriorities();
10. replayMemory.SetPriority(id, p);
11. **if** replayMemory.len() $> L$ **then**
12. replayMemory.RemoveOldExperience();
13. **end if**
14. **end for**

Algorithm 2 Learner**Learner.** Similarly, Algorithm 2 details the steps of the learner. The learner starts when the replay memory has N samples (lines 1 to 4) and trains the model using T_s iterations (line 5). At each iteration, the learner first samples a prioritized batch of experience (training samples), applies the learning rule, and updates the model parameters (lines 6 to 8). Subsequently, this function calculates and updates the priorities for experience (lines 9 to 10). And when there is too much data in replay memory, the function will remove old experience from replay memory (lines 11-13).

Task and resource allocation of simulator-based environment The function env.SimulateOneStep(s_{t-1},a_{t-1}) is explained in Algorithm 3. This function first gets the set N of nodes, and the waiting tasks V^w and running tasks V^r from state s_{t-1} (line 1). It then simulates resource allocations using a list of iterations (lines 3 to 7). At each iteration, the function sequentially allocates resources to waiting tasks using a scheduling configuration combination a_{t-1} (line 4) and checks the completion of running tasks at the current simulation time t^s (line 5). The status of nodes, and waiting, running and completed tasks are updated before moving to the next iteration. The iterations end when the simulation time exceeds the scheduling interval. Finally, the function converts the information of nodes and tasks into state s_t and reward r_t of time-step t, and returns them (lines 8 to 11).

Require: t^s: the simulation time;
Δt: the duration of one iteration in simulation;
V^w,V^r,V^c: the sets of waiting, running, and completed tasks;
N: the set of nodes.
1. Obtain N, V^w, V^r from state s_{t-1};
2. t^s=0;
3. **while** t^s is smaller than the scheduling interval **do**
4. $N, V^w, V^r \leftarrow$ environment.AllocateResource $(N, Task^w, a_{t-1})$;
5. $N, V^r, V^c \leftarrow$ environment.CheckCompletion $(N, Task^r)$;
6. t^s=$t^s + \Delta t$;
7. **end while**
8. $s_t \leftarrow$ stateBuilder.Build(N, V^w, V^r);
9. $V_r^t \leftarrow$ GetRunningTasks(V^r);
10. $V_w^t \leftarrow$ GetWaitingTasks(V^w);
11. $r_t \leftarrow$ rewardBuilder.Build(V_r^t, V_w^t);
12. **return** s_t, r_t.

Algorithm 3 env.SimulateOneStep(s_{t-1}, a_{t-1})Implementation of cluster simulator

The cluster simulator is mainly responsible for simulating jobs' scheduling according to the simulation nodes, jobs and actions' information provided from the simulation environment. Crucially, it implements two popular simulation schedulers: *Kubernetes simulation scheduler* and *Volcano simulation scheduler*, which are used to schedule different edge-cloud jobs. For example, DAG jobs will be scheduled by Kubernetes simulation scheduler, and AI jobs will be scheduled by Volcano simulation scheduler. After the scheduler completes scheduling, the simulator will get the simulation scheduling results (that is, tail latencies of edge-cloud jobs), so as to provide a basis for the reward of DRL training.

Note that in the cluster simulator, each sampling/training time-step t (e.g. an episode) corresponds to multiple iterations, and an iteration process is concretely shown in Fig. 6. At each iteration, the simulator

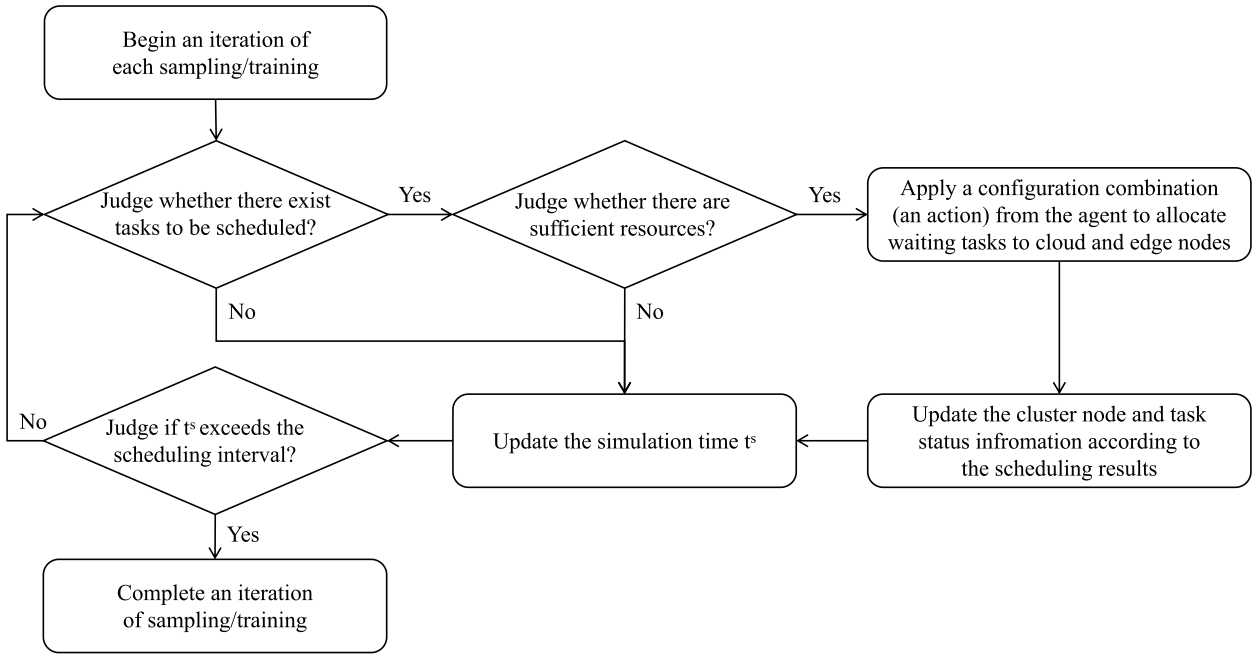


Fig. 6 An iteration process of sampling and training in cluster simulator

first judges if there exists tasks to be scheduled in the waiting queue and if there are sufficient resources. If the available resources exceed the requested resources by the waiting tasks, the simulator applies a configuration combination to assign the tasks to these cluster nodes, and updates the node and task statuses; otherwise it updates the simulation time t^s. The simulation completes if t^s is longer than the scheduling interval; otherwise it continues scheduling the remaining tasks at the next iteration.

Discussion of system uncertainties. We note that in real clusters, a job's performance is also influenced by random background activities such as system maintenance or garbage collection of operating systems. These activities are not incorporated in our simulator for two reasons. First, although background activities can create considerable CPU or network load (in particular when resource are saturated), this work focuses on comparing job performances across different scheduler configurations and implicitly assumes that the performances are estimated under the same factors (that is, different algorithms suffer from the same performance interferences). Second, in many practical scenarios (when systems have available resources for allocation), the performance impact of background activities is much smaller (e.g. 100 times smaller) than that caused by applying different scheduler configurations.

Evaluation

In this section, we evaluate the proposed approach with two major criteria: (1) its robust performance under diverse scheduling scenarios of dynamic edge-cloud workloads (Section "Evaluation of EdgeTuner's Effectiveness Under Dynamic Edge-Cloud Workloads") and dynamic available cluster resources (Section "Discussion of Reducing Tail Latency Under Dynamic Available Cluster Resources"); and (2) its effectiveness in significantly accelerate DRL training (Section "Evaluation of EdgeTuner's Acceleration Effect in DRL training") and how it is influenced by DRL training settings (Section "Discussion of DRL Training Settings").

Experimental settings

Experimental Platform. For evaluation experiments, we built a Kubernetes cluster, and the specific configuration is as follows:

- ▹ 1 Master Node + 4 Edge Devices + 5 Cloud Nodes
  - 16 Intel(R) Xeon(R) CPU E5-2660 v4 @2.00GHz processor cores and 32GB memory/Master Node, 4-core 1.5GHz Cortex-A72 ARMV8.0 and 4 GB memory (Raspberry Pi 4B)/Two Edge

devices, 384-core 1100MHz NVIDIA VOLTA GPU, 6-core 1.4Ghz NVIDIA Carmel ARMV8.2, and 8GB memory (Jetson Xavier NX)/Two Edge devices, 4 Intel(R) Xeon(R) CPU E5-2680 v4 @2.40GHz processor cores and 8GB memory/ Four Cloud Nodes, and Seven Intel(R) Xeon(R) Gold 6238 CPU @ 2.10GHz processors, 24GB NVIDIA TITAN RTX Graphics Card and 512GB memory/A Cloud Node
* Linux Ubuntu 18.04 LTS
* Python 3.8.5, Go 1.17.6, Docker 20.10.14, Volcano v1.0, and Kubernetes v1.19.0

▹ Two Intel(R) Xeon(R) Silver 4216 processors, 48GB Quadro RTX 8000 Graphics Card, and 256 GB memory (A GPU Node for conducting the training of DRL agents)

Edge-Cloud Workloads. In evaluation, driven by two types of real-world cluster traces provided by Alibaba [2, 3], we generate two edge-cloud workloads: *DAG batch workloads* and *AI workloads*. From the traces, we derive some crucial information of two generated edge-cloud workloads, such as job arrival pattern (here, job is submitted exactly at the job submission interval of the real trace), the number of tasks in a job, the resource (CPU, GPU and memory) request and resource limit of each task, and the workload (that is, the running time of a task).

▹ For *DAG batch workloads*, the basis is as follows:

* Driven by Alibaba cluter-trace-v2018 [2] that mainly records the information of offline batch tasks in the mixed CPU cluster with 4034 nodes running in 8 days
* Two typical application scenarios: *Daytime* (6:00 to 24:00) and *Night* (0:00 to 6:00)
* 136,500 jobs, 8.30 million tasks submitted *in the Daytime*, and 198,100 jobs, 8.40 million tasks submitted *at Night* (in the trace)

▹ For *AI workloads*, the basis is as follows:

* Driven by Alibaba cluter-trace-gpu-v2020 [3] that records the information collected from Alibaba PAI (Artificial Intelligence Platform) with over 6,500 GPUs (about 1800 machines) in a month
* Two typical application scenarios: *Daytime* (8:00 to 24:00) and *Night* (0:00 to 8:00)
* 1.76 million jobs, 12.54 million tasks submitted *in the Daytime*, and 2.46 million jobs, 17.55 million tasks submitted *at Night* (in the trace)

Considered Scheduler Configurations. In evaluation, we mainly consider two scheduler configurations: *scheduling algorithms* and *tail latency mitigation policies*. For **scheduling algorithms**, we compare against 11 representative Kubernetes scheduling algorithms [19]: LeastRequestedPriority (LRP), MostRequestedPriority (MRP), BalancedResourceAllocation (BRA), EqualPriority (EP), Resource Limits Priority (RLP), Taint Toleration Priority (TTP), Node Affinity Priority (NAP), Image Locality Priority (ILP), Node Prefer Avoid Pods Priority (NPAPP), Node Label Priority (NLP), and Inter Pod Affinity Priority (IPAP). And we also compare against 13 representative Volcano scheduling algorithms [10]: GANG_BRA, GANG_MRP, GANG_LRP, DRF_BRA, DRF_MRP, DRF_LRP, GANG_DRF_BINPACK, GANG_DRF_BRA, GANG_DRF_MRP, GANG_DRF_LRP, SLA_BRA, SLA_MRP and SLA_LRP. Moreover, our settings of tail latency mitigation policies are: (1)for DAG batch workloads, we only set 11 different ratios of TRP: *70%*, *72%*, *74%*, *76%*, *78%*, *80%*, *82%*, *84%*, *86%*, *88%*, and *90%*. (2)for AI workloads, we set 11 different ratios of TDP and TRP: *70%*, *72%*, *74%*, *76%*, *78%*, *80%*, *82%*, *84%*, *86%*, *88%*, and *90%*, respectively.

DRL Training Setting. We implemented the proposed approach based on Google DeepMind's RainBow tool [31]. In DQN training, we use the latest technique [63] that combines Implicit Quantile Networks (IQN) [11] and Ape-X [33]. The training settings of three DRL elements are as follows:

▹ *State*. The interval between two time-steps is 10 seconds, hence the number of waiting and running tasks in a state is set to 10. In Equations 3, 4 and 5, the values of α_1, β_1, α_2, β_2, α_3 and β_3 are set to -0.02, 10, -0.1, 5, -0.002, 1, respectively.
▹ *Actor*. The maximal number of time-steps is 60 millions and the experience replay memory capacity is set to 12 million. In sampling, the number of actors is 8, the history length (the number of consecutive states processed) is set to 8, and the frequency of sampling from memory is 5.
▹ *Learner*. The training phase starts after 10K time-steps of the sampling phase. In training, the network hidden node size is set to 64, the batch size is 32, the network is updated every 1000 steps, and the importance sampling weight in prioritised experience replay is 0.5.

Table 1 Description of important notations in problem formulation and proposed approach

Notations	Introduction
$\mathcal{S}$	the state space
$\mathcal{A}$	the action space
$R(s_t, a_t)$	the reward function
$P(s_{t+1}\|s_t, a_t)$	transition dynamics (reflects the time-variant dynamics of cluster, $0 \leq P(s_{t+1}\|s_t, a_t) \leq 1$)
s_t	the node and task state information during a scheduling interval
v^w	a waiting task
v^r	a running task
a_t	an action that is one possible configuration combination of cluster schedulers
$V^{allocate}$	the tasks that obtain resource allocations
$V^{complete}$	the completed tasks
V^{arrive}	the newly arrival tasks
JTL	denoted the job tail latency as JTL
J	the set of jobs completed within period $(t-1, t]$
TTL	denoted the tail latency of a task as TTL
V^{run}	the set of tasks running within period $(t-1, t]$
V^{wait}	the set of tasks waiting within period $(t-1, t]$
r^{job}	the reward of job
r^{run}	the reward of the set V^{run}
r^{wait}	the reward of the set V^{wait}
r_t	the reward of time-step t
$\alpha_1, \alpha_2, \alpha_3$	the negative values
$\beta_1, \beta_2, \beta_3$	the positive values
$B(Actor)$	the maximal size of local buffer in Actor
$T_s(Actor)$	the number of sampling steps in Actor
$N(Learner)$	the experience number to start training in Learner
$L(Learner)$	the maximal size of local buffer in Learner
$T_s(Learner)$	the maximum number of training in Learner
t^s	the simulation time
Δt	the duration of one iteration in simulation
$\|N\|$	the number of cluster nodes

In particular, as shown in Table 1, we summarize some important notations in problem formulation and our proposed approach.

Evaluation Metrics. For *evaluation metrics*, we consider job performance and training efficiency.

▷ *Job performance*: measured by the average tail latency
▷ *Simulation acceleration*: measured by the sampling time and the training time in DRL training

Evaluation of EdgeTuner's effectiveness under dynamic edge-cloud workloads

In this section, we evaluate the effectiveness of EdgeTuner in reducing tail latencies by adaptively selecting its scheduler configurations under dynamic edge-cloud workloads.

Evaluation settings

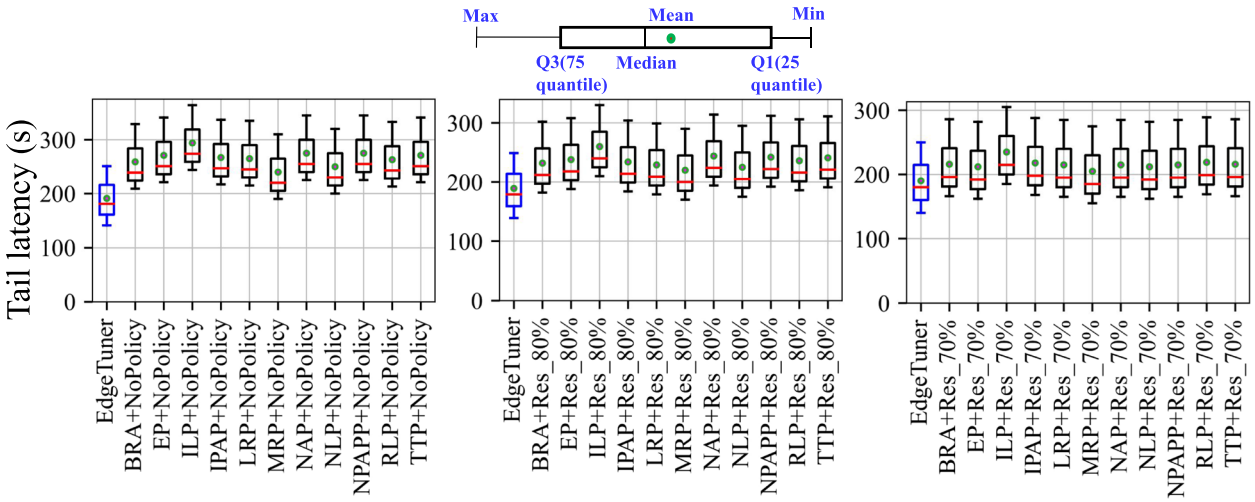
Workloads. This evaluation tests 4 different workloads, consisting of 2 generated edge-cloud workload patterns (*DAG batch workloads* and *AI workloads*) and 2 periods (*Daytime* and *Night*).

Compared Configuration Settings. In particular, for DAG batch workloads, we evaluate them by selecting 11 baseline Kubernetes scheduling algorithms and two ratios for TRP (namely Res_80% and Res_70%). Note that we don't select TDP in scheduling DAG batch workloads, because when using TDP in such workloads' scheduling scenarios, the DAG dependencies between their tasks will be lost, thereby causing the failure of subsequent tasks to run normally, so TDP can not be

selected. However, for AI workloads, due to no DAG dependencies between their tasks, we evaluate them by selecting 13 baseline Volcano scheduling algorithms and a total of four ratios for TRP and TDP (namely Res_80%, Res_70%, Dis_80%, and Dis_70%). *In particular*, for TRP, Res_80% indicates that when 80% of the tasks in a job are completed, the remaining 20% of the tasks are restarted; Res_70% indicates that when 70% of the tasks in a job are completed, the remaining 30% of the tasks are restarted. For TDP, Dis_80% indicates that for TDP, when 80% of tasks in a job are completed, the remaining 20% are discarded; Dis_70% indicates that for TDP, when 70% of tasks in a job are completed, the remaining 30% are discarded.

Comparison of tail latency reduction under dynamic edge-cloud workloads

Comparison Using DAG Batch Workloads. Figure 7 uses box plots to illustrate each DAG batch workload's distribution of tail latencies, including their minimum and maximum values, the first quartile, median, and third quartile. We can observe that in considered 6 scenarios (Scenarios 1 to 6), EdgeTuner achieves lower tail latencies than other scheduler configurations, indicating the DRL agent selects proper scheduler configurations for different DAG batch workloads in the cluster. *In detail*, most of jobs in DAG workloads complete within dozens of minutes. This means the waiting and running tasks continuously change at different scheduling time-steps, and the agent selects the optimal scheduler configuration that brings the largest reward. Particularly, Table 2 summarizes the percentages of reduced tail latency under two DAG batch workloads (Daytime and Night), when comparing EdgeTuner against the 11 baseline Kubernetes scheduler configurations. We can see that these reductions vary across different workloads depending on a variety of factors, such as different types of workloads and multiple different scheduler configurations.



(a) Daytime (DAG batch workloads): different Kubernetes scheduler configurations

(b) Night (DAG batch workloads): different Kubernetes scheduler configurations

Fig. 7 Comparison of tail latency under DAG batch workloads driven by Alibaba trace 2018

Table 2 Percentages of reduced tail latency under DAG batch workloads driven by Alibaba trace 2018

Scenario	Job scheduling scenarios			TRP		BRA	EP	ILP	IPAP	LRP	MRP	NAP	NLP	NPAPP	RLP	TTP
1	DAG batch workloads	Daytime	Original resources	No policy	0%	26.25	29.52	35.03	28.46	27.92	20.42	30.55	23.60	30.55	27.38	29.52
2				Restart rate	80%	18.10	20.17	26.92	18.80	17.03	13.64	22.13	15.56	21.49	19.49	21.16
3				Restart rate	70%	12.50	10.85	19.57	13.30	12.09	7.80	12.09	10.85	12.09	13.70	12.50
4		Night	Original resources	No policy	0%	23.38	28.14	27.94	27.54	30.53	24.82	27.74	26.41	27.44	27.13	27.54
5				Restart rate	80%	17.09	22.88	21.72	21.00	21.84	19.03	21.60	20.64	21.95	21.12	22.19
6				Restart rate	70%	9.91	14.10	11.75	11.75	13.67	11.30	12.35	13.23	14.24	13.38	15.36

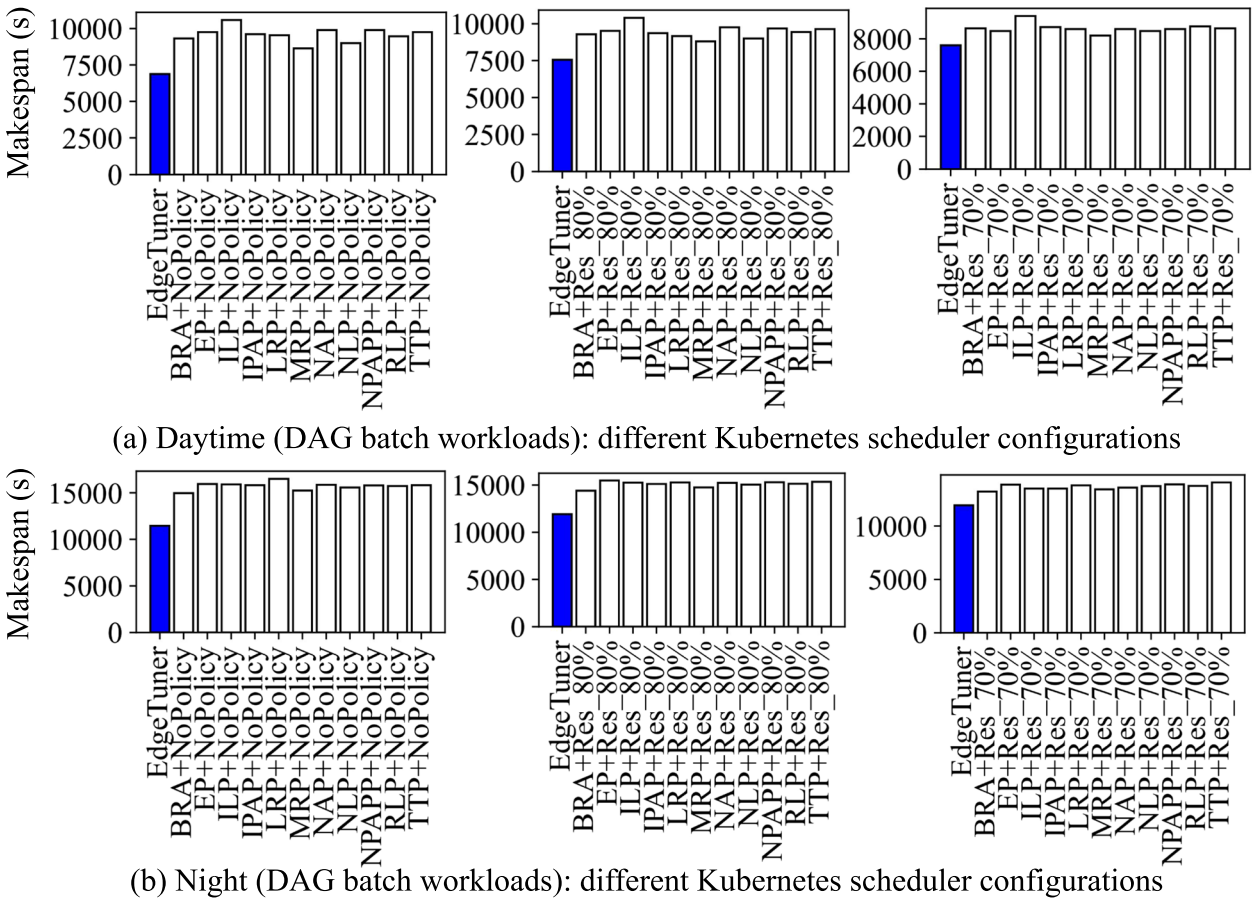


Fig. 8 Comparison of Makespan under DAG batch workloads driven by Alibaba trace 2018

Figure 8 shows the comparison of Makespan (a common performance evaluation metric which represents the total time spent from the start of the first job to the end of all jobs in the workload) between EdgeTuner and other 11 Kubernetes scheduler configurations. We can observe that in considered 6 scenarios (Scenarios 1 to 6), EdgeTuner can also achieve lower Makespan than other scheduler configurations. This indicates the DRL agent of EdgeTuner chooses the optimal scheduling configuration combination for each DAG batch workload scheduling scenario, thereby allowing workload scheduling to be completed faster. In addition, Fig. 9 shows the comparison of average CPU usage of the cluster during a certain execution period (between EdgeTuner and other 11 Kubernetes scheduler configurations). We can see that in considered 6 scenarios (Scenarios 1 to 6), EdgeTuner can also achieve higher cluster CPU usage than other scheduler configurations. This indicates that during cluster execution, EdgeTuner can always choose the optimal scheduling configuration combination for the continuously arriving DAG batch workloads, thus improving the CPU resource utilization of the cluster.

Comparison Using AI Workloads. Similarly, Fig. 10 also uses box plots to illustrate each AI workload's distribution of tail latencies. We can also observe that in considered 10 scenarios (Scenarios 7 to 16), EdgeTuner still achieves lower tail latencies than other scheduler configurations, which also indicates the DRL agent selects proper scheduler configurations for different AI workloads in the cluster. *Note that*, compared to DAG workloads, most of jobs in AI workloads take longer time to complete (e.g. more than 10min). Despite this, the agent can still select the optimal scheduler configuration that brings the largest reward (that is, achieving the lowest tail latency). Particularly, Table 3 also summarizes the percentages of reduced tail latency under two AI workloads (Daytime and Night), when comparing EdgeTuner against the 13 baseline Volcano scheduler configurations. We can also see that these reductions vary across different workloads depending on a variety of factors, such as types of workloads and multiple different scheduler configurations.

Figure 11 shows the comparison of Makespan between EdgeTuner and other 13 Volcano scheduler

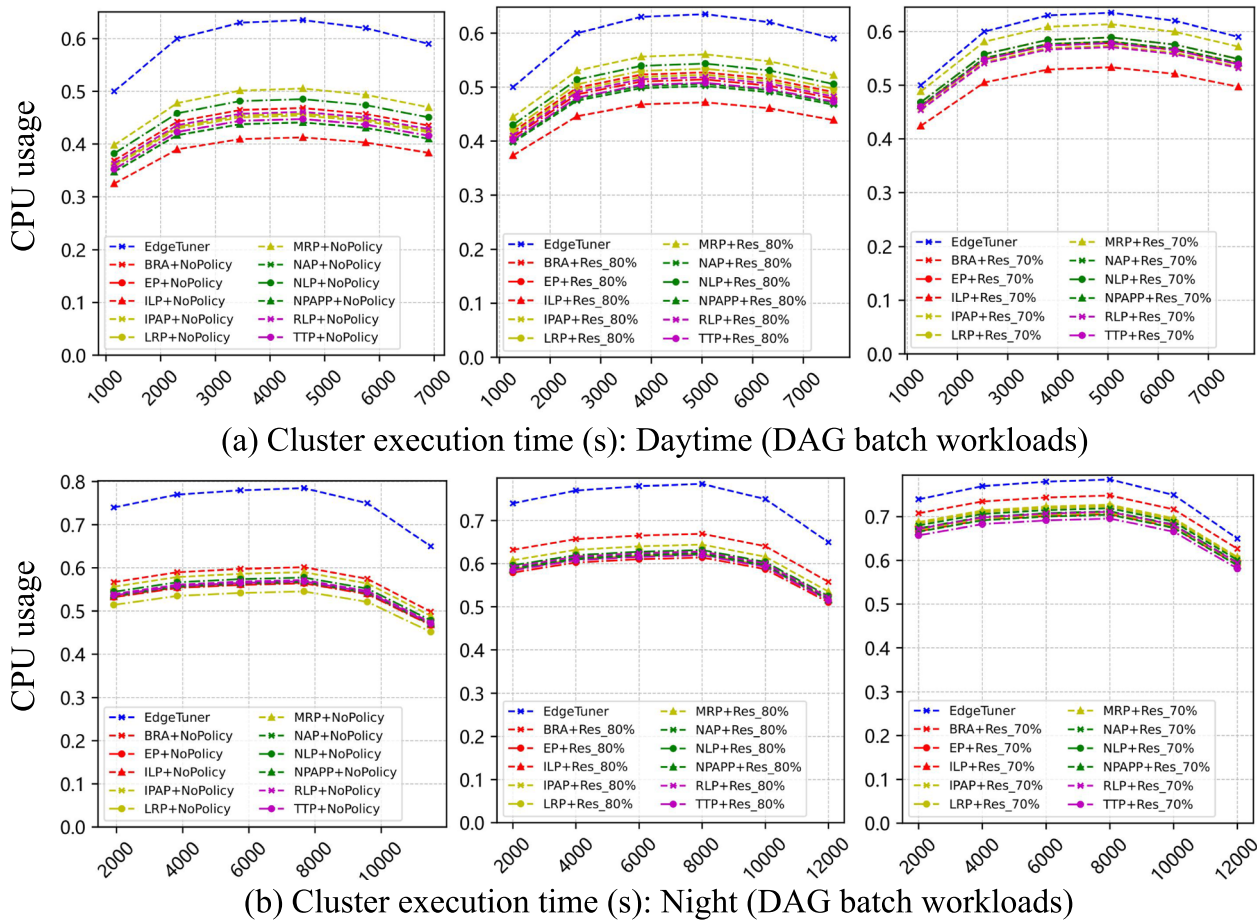


Fig. 9 Average CPU usage of the cluster during a certain execution period under DAG batch workloads

configurations. We can observe that in considered 10 scenarios (Scenarios 7 to 16), EdgeTuner can also achieve lower Makespan than other scheduler configurations, indicating the DRL agent of EdgeTuner chooses the optimal scheduling configuration combination for each AI workload scheduling scenario. In this way, compared to other scheduler configurations, EdgeTuner allows workload scheduling to be completed faster. In addition, Fig. 12 shows the comparison of average CPU usage of the cluster during a certain execution period (between EdgeTuner and other 13 Volcano scheduler configurations). We can see that in considered 10 scenarios (Scenarios 7 to 16), EdgeTuner can also achieve higher cluster CPU usage than other scheduler configurations, which indicates that during cluster execution, EdgeTuner can always choose the optimal scheduling configuration combination for the continuously arriving AI workloads. As a result, EdgeTuner improves the CPU resource utilization of the cluster.

Evaluation Results. In general, when considering all 16 evaluation scenarios, our approach achieves an average of 21.66% reductions in tail latencies.

Discussion of reducing job completion time (under dynamic Edge-Cloud workloads). When scheduling diverse workloads in the cluster, Job completion time (JCT) is also an important performance metric. Note that, for a DAG batch job, the job's tail latency is the most significant factor in its completion time. For an AI job, the maximum job's tail latency represents its completion time. Therefore, we can find that the job's tail latency determines its final completion time. In this paper, in our considered 16 scheduling scenarios (Scenarios 1 to 16), EdgeTuner can achieve lower tail latencies than other scheduler configurations due to its dynamic tuning mechanism, which indicates the DRL agent selects proper scheduler configurations for different DAG batch and AI workloads in the cluster. Correspondingly, EdgeTuner can also achieve lower

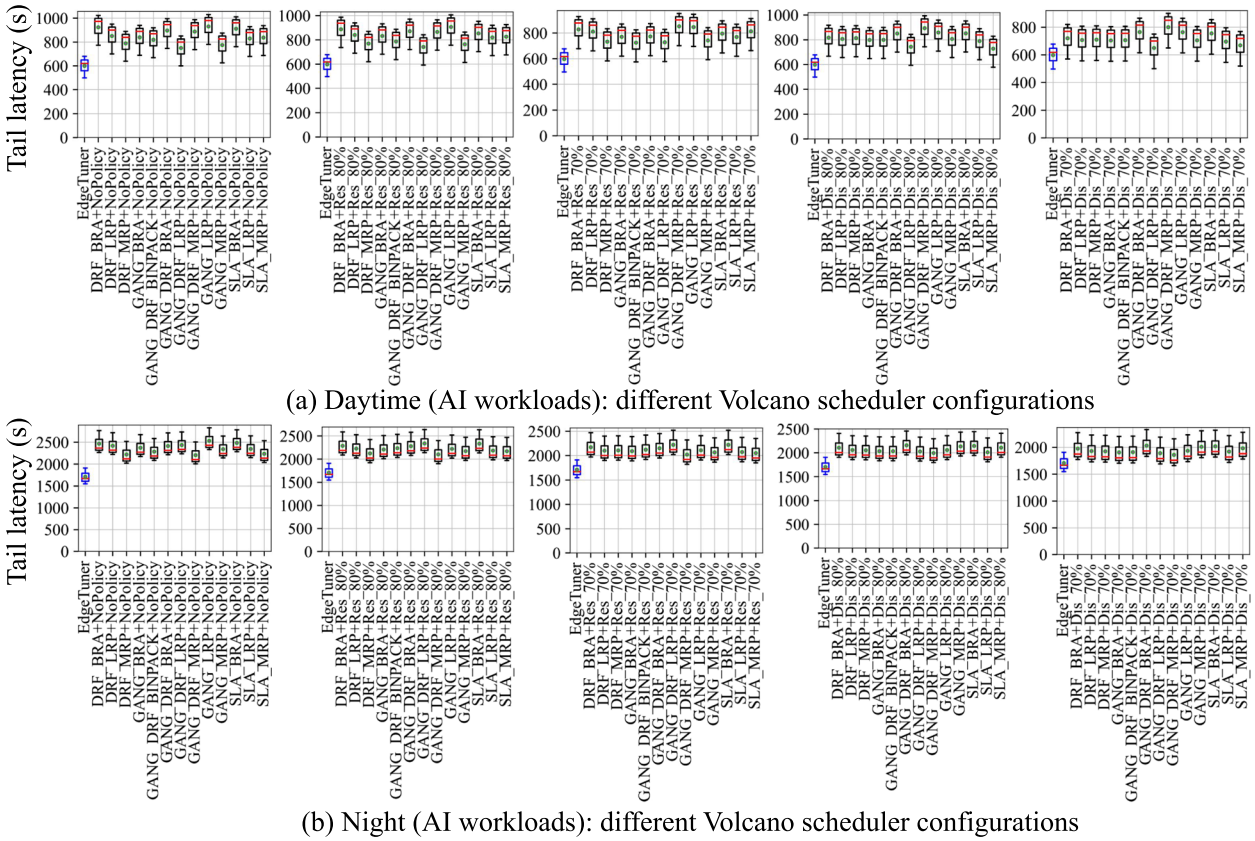


Fig. 10 Comparison of tail latency under AI workloads driven by Alibaba trace 2020

job completion time for each workload scheduling scenario.

Evaluation of EdgeTuner's acceleration effect in DRL training

Note that, the effectiveness of tuning scheduler configurations relies on efficiently training DRL agents. And an agent's training time consists of two parts: (1)the major training time comes from collecting samples in the actor (Algorithm 1). Each training needs several million samples to converge and each sample needs at least a few seconds to obtain in real clusters; (2)using the collected samples in the replay memory, the learner trains the DQN model. Based on this, we evaluate the acceleration effect of EdgeTuner.

Evaluation settings

According to the experimental settings of the previous section, under our considered 16 scenarios, we compare the DRL training time of EdgeTuner (collecting samples with the Kubernetes simulator in Algorithm 3) and the real Kubernetes cluster.

Comparison of DRL training time in EdgeTuner and real Kubernetes cluster

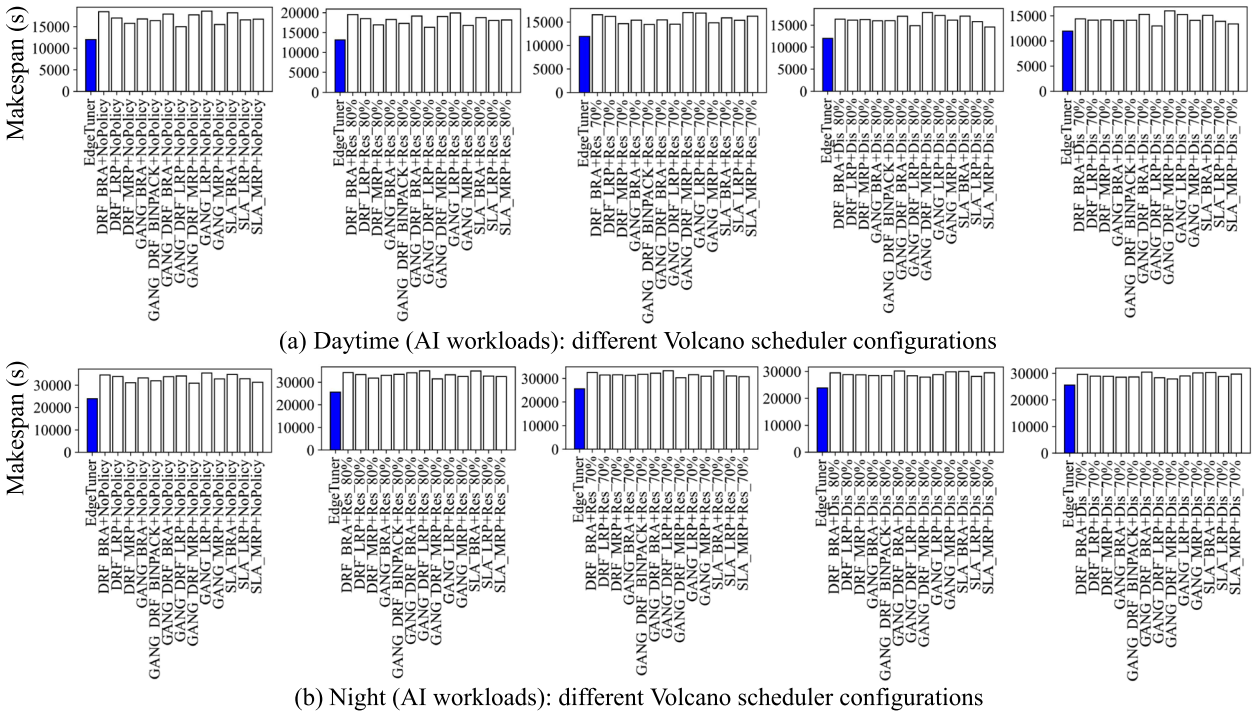
Table 4 lists the sampling times and training times of EdgeTuner and real Kubernetes cluster, and Fig. 13 shows reductions of sampling time and training time.

Evaluation Results. For all considered 16 scenarios, *the sampling phase* takes a long time (more than 92.27 hours) in the real Kubernetes cluster, and EdgeTuner considerably reduce this time to a few minutes (acceleration by up to 3134.17x). Similarly, in real Kubernetes cluster, *the training phase* also completes in dozens of hours (e.g. even several days) due to the time-consuming sampling process. In contrast, EdgeTuner completes the training phase within a few hours. The training time also varies across different states (i.e. different jobs, tasks) and our approach can reduce the training time by an average of 97.15x.

Results. In general, when considering *both sampling and training phases*, our approach accelerates DRL training by an average of 151.63x. This is because it can perform the DRL training to in an offline way, and can provide the adapted DRL agent timely.

Table 3 Percentages of reduced tail latency under AI workloads driven by Alibaba trace 2020

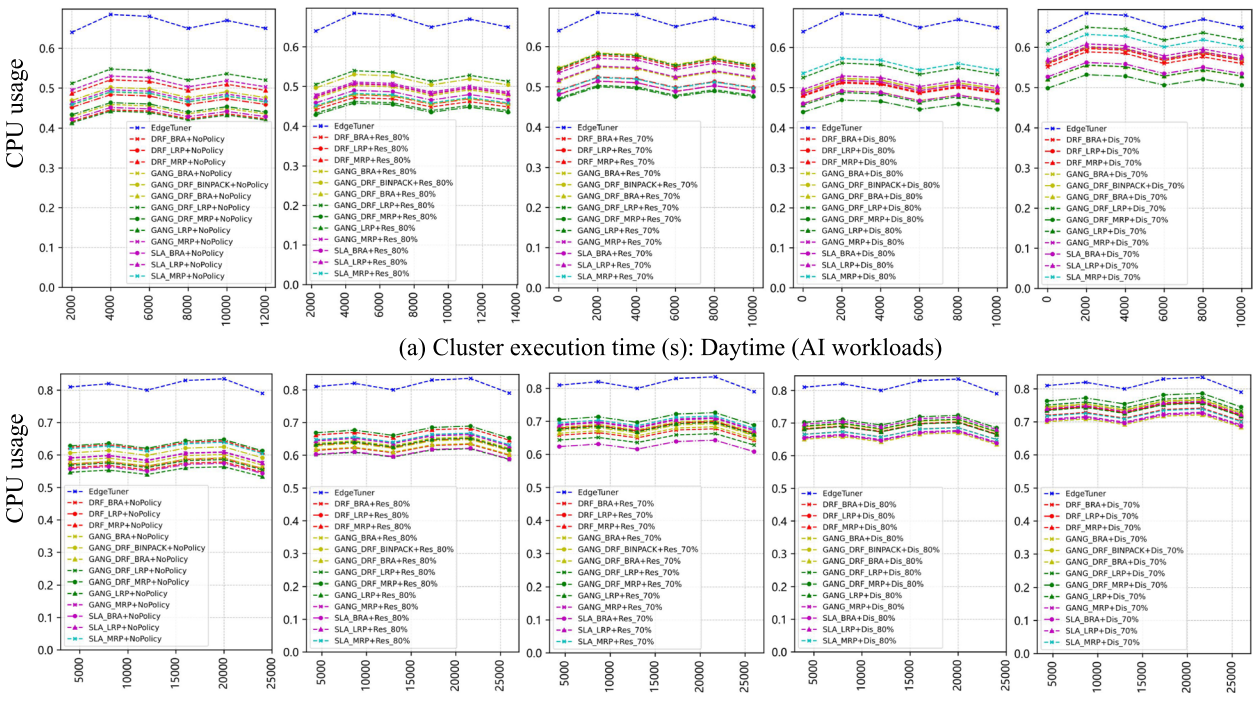
Scenario	Job scheduling scenarios			TDP/TRP		DRF _BRA	DRF _LRP	DRF _MRP	GANG _BRA	GANG _DRF _ BINPACK	GANG _DRF _BRA	GANG _DRF _LRP	GANG _DRF _MRP	GANG _LRP	GANG _MRP	SLA _BRA	SLA _LRP	SLA _MRP
7	AI workloads	Daytime	Original resources	No policy	0%	34.99	29.41	23.95	28.49	26.74	33.04	20.00	32.36	35.48	22.58	34.14	27.62	28.32
8				Restart rate	80%	32.58	31.50	27.52	28.13	24.02	31.34	22.64	34.64	34.00	26.81	29.89	27.34	31.03
9				Restart rate	70%	27.90	26.39	18.55	22.47	17.66	22.77	17.99	29.93	29.43	19.54	24.91	22.37	26.57
10				Discard rate	80%	26.77	25.77	26.41	25.03	25.13	29.61	19.49	33.00	30.27	25.77	29.69	24.18	17.83
11				Discard rate	70%	16.94	15.42	15.77	15.06	15.30	21.83	8.00	25.25	21.73	15.18	20.79	14.08	10.61
12		Night	Original resources	No policy	0%	30.69	29.34	22.97	27.97	25.16	29.19	29.80	22.45	32.46	26.99	31.19	27.14	23.49
13				Restart rate	80%	25.32	23.27	19.51	22.61	23.58	25.05	26.95	18.67	23.06	21.33	26.79	21.76	21.25
14				Restart rate	70%	21.24	18.62	18.81	18.23	19.39	20.51	23.02	15.40	18.93	17.20	23.02	17.52	16.63
15				Discard rate	80%	19.04	17.15	16.95	16.18	16.18	20.96	15.93	14.54	17.15	20.10	20.48	15.22	19.12
16				Discard rate	70%	13.57	11.55	11.32	10.39	10.63	15.99	9.73	8.23	11.78	15.07	15.50	11.09	13.87



(a) Daytime (AI workloads): different Volcano scheduler configurations

(b) Night (AI workloads): different Volcano scheduler configurations

Fig. 11 Comparison of Makespan under AI workloads driven by Alibaba trace 2020



(a) Cluster execution time (s): Daytime (AI workloads)

(b) Cluster execution time (s): Night (AI workloads)

Fig. 12 Average CPU usage of the cluster during a certain execution period under AI workloads

Table 4 Sampling time and training time of EdgeTuner and real Kubernetes cluster under different workloads

Scenario	Sampling time (seconds)		Training time (seconds)	
	EdgeTuner	Kubernetes	EdgeTuner	Kubernetes
1	110	332160	6224	742708
2	118	342578	7002	774568
3	112	351027	7016	780246
4	155	341463	8189	896439
5	164	349873	8964	914782
6	159	356047	8902	913026
7	583	673240	9336	882708
8	595	688452	10402	943258
9	585	680064	9408	866802
10	587	682247	9386	856349
11	596	684065	9502	876543
12	740	1114389	16134	1446439
13	780	1156000	17225	1456020
14	764	1175438	16548	1470865
15	796	1118634	16208	1450862
16	802	1116065	17106	1465684

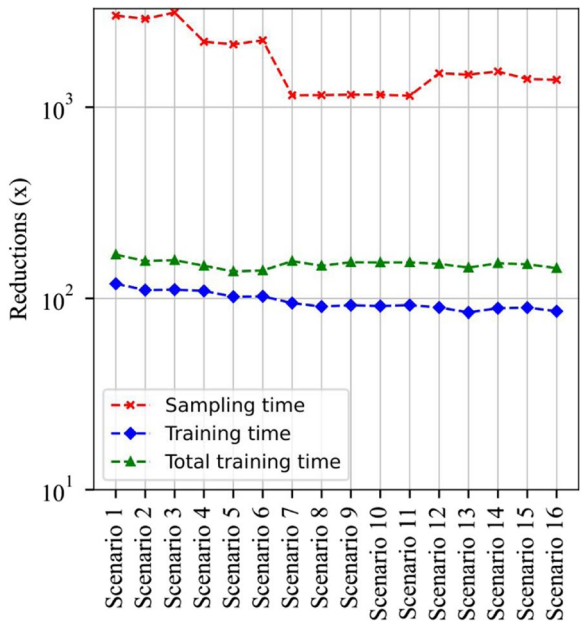


Fig. 13 Reductions of sampling time and training time under considered 16 scenarios

Discussion of DRL training settings

Evaluation settings

We take a DAG batch workload (Daytime) and an AI workload (Night) as two examples, and design experiments to discuss the *three major factors* that influence DRL training efficiency. In addition, we use five metrics to evaluate: *sampling time*, *training time*, *the total number of samples*, *the total number of sampling and training iterations*, and *the total time of sampling and training phases*.

Comparison of different DRL training techniques

In DRL training, our work adpots the latest Rainbow tool [31] combined with two model training techniques: IQN for distributional reinforcement learning [11], and Ape-X for distributed sampling and prioritized experience replay [33].

Evaluation Results. We can see that under two different edge-cloud workload scheduling scenarios: (i) Fig. 14(a) shows that IQN incurs the longest sampling time because it only uses one actor. In contrast, Ape-X supports multiple actors and considerably reduces the sampling time when collecting the same number of samples. (ii) Fig. 14(b) shows that Ape-X has the longest training time, while the IQN technique accelerates the convergence speed. (iii) Fig. 14(c) and (d) show that Ape-X needs the largest samples and training iterations, and thus takes the longest time to complete the whole training process (Fig. 14(e)).

Comparison of actors with different numbers

This evaluation considers three different numbers of actors: 8 (used in EdgeTuner), 4, and 16.

Evaluation Results. We can see that under two different edge-cloud workload scheduling scenarios: (i) Fig. 15(a) shows that more actors indeed reduce sampling time. (ii) However, when the actor number is 16, the sampling speed exceeds the training speed. This means the 16 actors need to wait for the learner while occupying resources, thus delaying the training process (Fig. 15(b)). This claim is also verified in Fig. 15(c), (d) and (e)'s results.

Comparison of different history length

In DRL training, history length decides the number of time-steps to construct a state in the environment. Note that, the longer the history length, the more information the agent can learn from a state. This evaluation considers three history lengths: 8 (used in EdgeTuner), 1, and 4.

Evaluation Results. We can see that under two different edge-cloud workload scheduling scenarios, (i) Fig. 16 display that when the history length is 1 (that is, the states in different iterations are independent of each other), the training needs the largest numbers of samples and the longest time to converge. (ii) However, when the history length is 8, the training needs the smallest number of samples (Fig. 16(c)), but its sampling time is still longer than that of history length 4 (Fig. 16(a)). This is because the state of history length 8 is two times larger

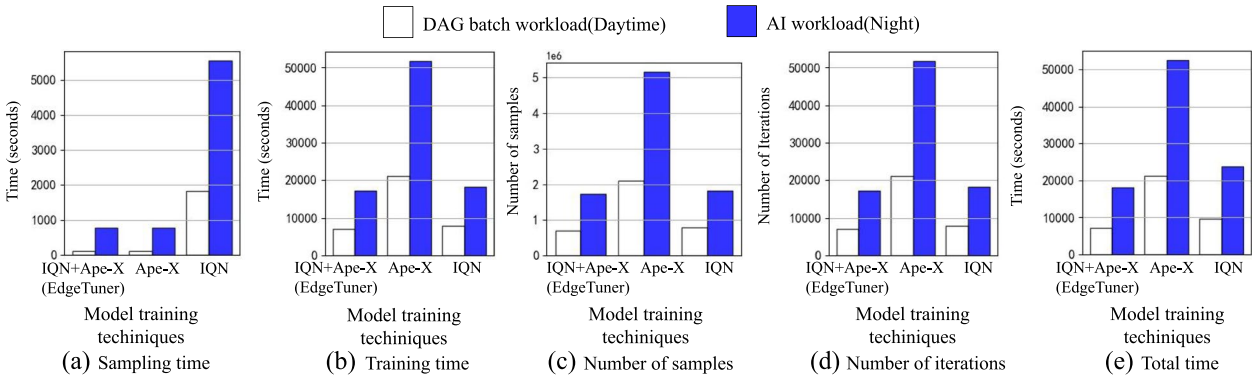


Fig. 14 Comparison of DRL training overheads under different model training techniques

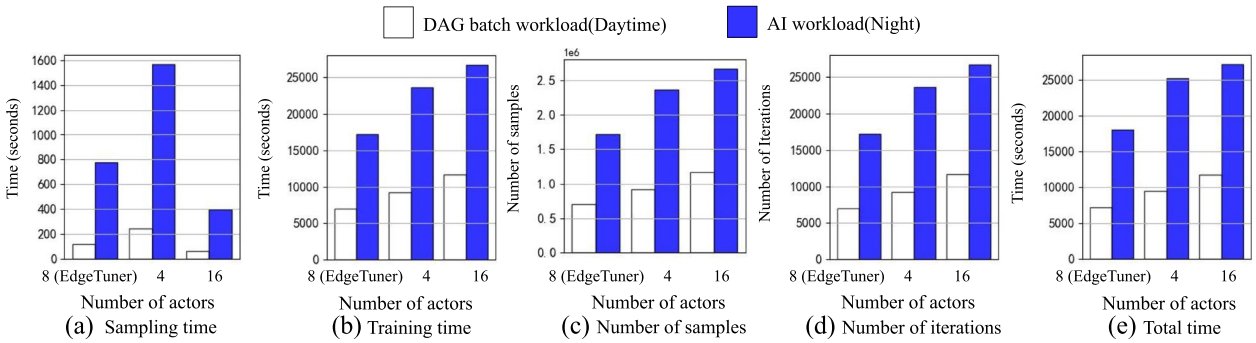


Fig. 15 Comparison of DRL training overheads under different actors

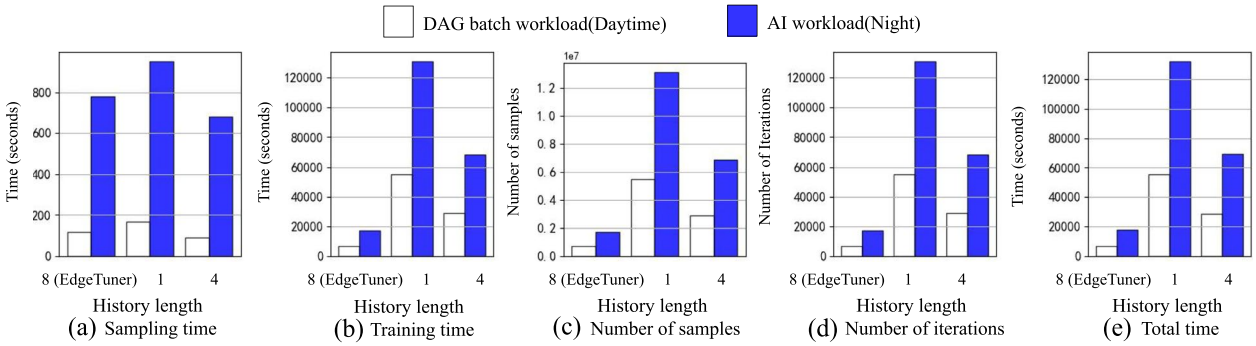


Fig. 16 Comparison of DRL training overheads under different history lengths

than that of history length 4 and hence each sample's collection time is longer in the former setting.

Discussion of reducing tail latency under dynamic available cluster resources

Evaluation settings

Workloads and Available Resources. In evaluation, we test 4 different workloads, covering 2 generated edge-cloud workload patterns (*DAG batch workloads* and *AI workloads*) and 2 periods (*Daytime* and *Night*). Moreover, we further test 4 dynamic resource changes: 50% decrease, 25% decrease, 25% increase, and 50% increase in cluster resources, respectively.

Compared Configuration Settings. In particular, for DAG batch workloads, we evaluate them by selecting 11 baseline Kubernetes scheduling algorithms and one median ratio 80% of TRP (namely Res_80%). And for AI workloads, we evaluate them by selecting 13 baseline

Table 5 Percentages of reduced tail latency of DAG batch workloads driven by Alibaba trace 2018 under dynamic resources

Scenario	Job scheduling scenarios			TRP		BRA	EP	ILP	IPAP	LRP	MRP	NAP	NLP	NPAPP	RLP	TTP
17	DAG batch workloads	Daytime	Original resources	Restart rate	80%	18.10	20.17	26.92	18.80	17.03	13.64	22.13	15.56	21.49	19.49	21.16
18			50% decrease	Restart rate	80%	15.56	15.02	17.65	17.39	18.15	2.92	20.83	18.15	21.76	16.61	20.12
19			25% decrease	Restart rate	80%	19.08	24.29	17.19	25.35	21.48	8.23	26.39	27.89	20.30	28.14	22.63
20			25% increase	Restart rate	80%	16.99	26.92	25.00	26.29	28.45	13.64	28.15	32.68	16.59	34.48	25.33
21			50% increase	Restart rate	80%	18.78	31.03	28.89	29.20	30.13	14.89	25.93	34.96	27.93	33.88	29.52
22		Night	Original resources	Restart rate	80%	17.09	22.88	21.72	21.00	21.84	19.03	21.60	20.64	21.95	21.12	22.19
23			50% decrease	Restart rate	80%	16.28	10.05	10.48	9.17	15.52	0.85	9.26	11.92	10.05	7.64	9.96
24			25% decrease	Restart rate	80%	19.60	19.25	17.72	19.60	19.94	16.62	18.79	17.11	19.25	22.91	22.59
25			25% increase	Restart rate	80%	10.92	10.44	13.65	10.92	21.55	9.48	14.53	9.80	14.96	14.53	12.00
26			50% increase	Restart rate	80%	11.68	13.42	14.79	10.87	22.56	10.20	13.26	14.03	14.94	12.79	14.03

Table 6 Percentages of reduced tail latency of AI workloads driven by Alibaba trace 2020 under dynamic resources

Scenario	Job scheduling scenarios			TDP/TRP		DRF _BRA	DRF _LRP	DRF _MRP	GANG _BRA	GANG _DRF _ BINPACK	GANG _DRF _BRA	GANG _DRF _LRP	GANG _DRF _MRP	GANG _LRP	GANG _MRP	SLA _BRA	SLA _LRP	SLA _MRP
27	AI workloads	Daytime	Original resources	Discard rate	80%	26.77	25.77	26.41	25.03	25.13	29.61	19.49	33.00	30.27	25.77	29.69	24.18	17.83
28			50% decrease	Discard rate	80%	21.38	24.41	21.29	21.29	21.38	21.38	24.41	21.38	24.41	21.38	21.38	24.59	21.20
29			25% decrease	Discard rate	80%	22.75	24.03	15.64	22.54	15.25	22.64	24.14	15.64	24.03	15.64	22.54	24.03	15.51
30			25% increase	Discard rate	80%	19.28	12.91	31.80	19.28	31.92	19.28	12.72	31.68	12.91	31.92	19.28	12.91	31.80
31			50% increase	Discard rate	80%	20.03	18.75	37.43	20.39	37.43	20.03	18.57	37.32	18.75	37.32	20.39	18.57	37.21
32		Night	Original resources	Discard rate	80%	19.04	17.15	16.95	16.18	16.18	20.96	15.93	14.54	17.15	20.10	20.48	15.22	19.12
33			50% decrease	Discard rate	80%	19.25	22.36	19.15	19.15	19.25	19.25	22.36	19.25	22.36	19.25	19.25	22.53	19.05
34			25% decrease	Discard rate	80%	13.78	15.21	5.85	13.54	5.41	13.66	15.33	5.85	15.21	5.85	13.54	15.21	5.70
35			25% increase	Discard rate	80%	13.93	7.15	27.29	13.93	27.42	13.93	6.94	27.16	7.15	27.42	13.93	7.15	27.29
36			50% increase	Discard rate	80%	16.17	14.84	34.41	16.55	34.41	16.17	14.64	34.29	14.84	34.29	16.55	14.64	34.18

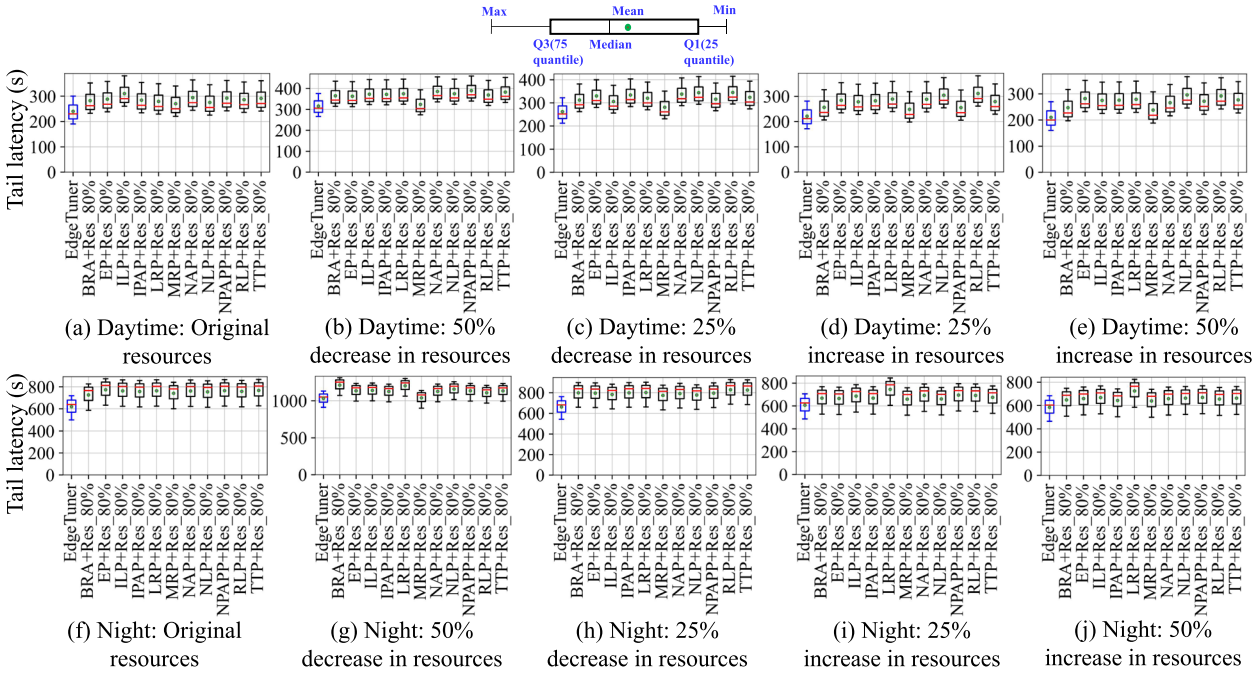


Fig. 17 Comparison of tail latency of DAG batch workloads driven by Alibaba trace 2018 under dynamic resources

Volcano scheduling algorithms and one median ratio 80% of TDP (namely Dis_80%).

Comparison of tail latency reduction under dynamic available cluster resources

Table 5 and Table 6 summarizes the percentages of reduced tail latency, when comparing EdgeTuner against the baseline scheduler configurations. Figures 17 and 18 also use box plots to display the comparison results.

Experimental Results. We can observe that in our considered 20 different scheduling scenarios (Scenarios 17 to 36): (i) compared to other baseline scheduler configurations, EdgeTuner consistently brings the lowest tail latencies due to its dynamic tuning mechanism. (ii) in particular, for the same workload, less available resources result in higher tail latencies.

Results. In general, when the cluster resource changes, the DRL agent needs to be re-trained because the node information changes in its state. In EdgeTuner, this training can be performed offline by setting different resources in its simulator for the same workload, thus avoiding the time-consuming online learning process. Moreover, as shown in Figs. 19 and 20, we also compare Makespan of EdgeTuner with other scheduler configurations in 20 different workload scheduling scenarios (Scenarios 17 to 36). We can see that under these scenarios that the cluster resource changes, EdgeTuner can also achieve lower Makespan than other scheduler configurations (that is, enabling workload scheduling to be completed more quickly), which indicates the DRL agent of EdgeTuner chooses the optimal scheduling configuration combination for each scenario. Thus, EdgeTuner can well adapt to such situations where cluster resources change dynamically.

Discussion of reducing job completion time (under dynamic available cluster resources). First, the job's tail latency determines its final completion time. In detail, for a DAG batch job, the job's tail latency is the most significant factor in its completion time. For an AI job, the maximum job's tail latency represents its completion time. Secondly, when the available cluster resources change dynamically (Scenarios 17 to 36), we can also find that EdgeTuner can achieve lower tail latencies than other scheduler configurations due to its dynamic tuning mechanism, which indicates the DRL agent selects proper scheduler configurations and how well EdgeTuner can adapt to dynamically changing resource situations. Therefore, similar to dynamically changing Edge-Cloud workload scenarios, EdgeTuner can also achieve lower job completion time for such scheduling scenarios with dynamic available cluster resources.

Related work

In modern cloud data centers, the scheduling of diverse workloads has aroused the pursuit of many scholars and industries. Table 7 summarizes existing scheduling techniques from the following three perspectives.

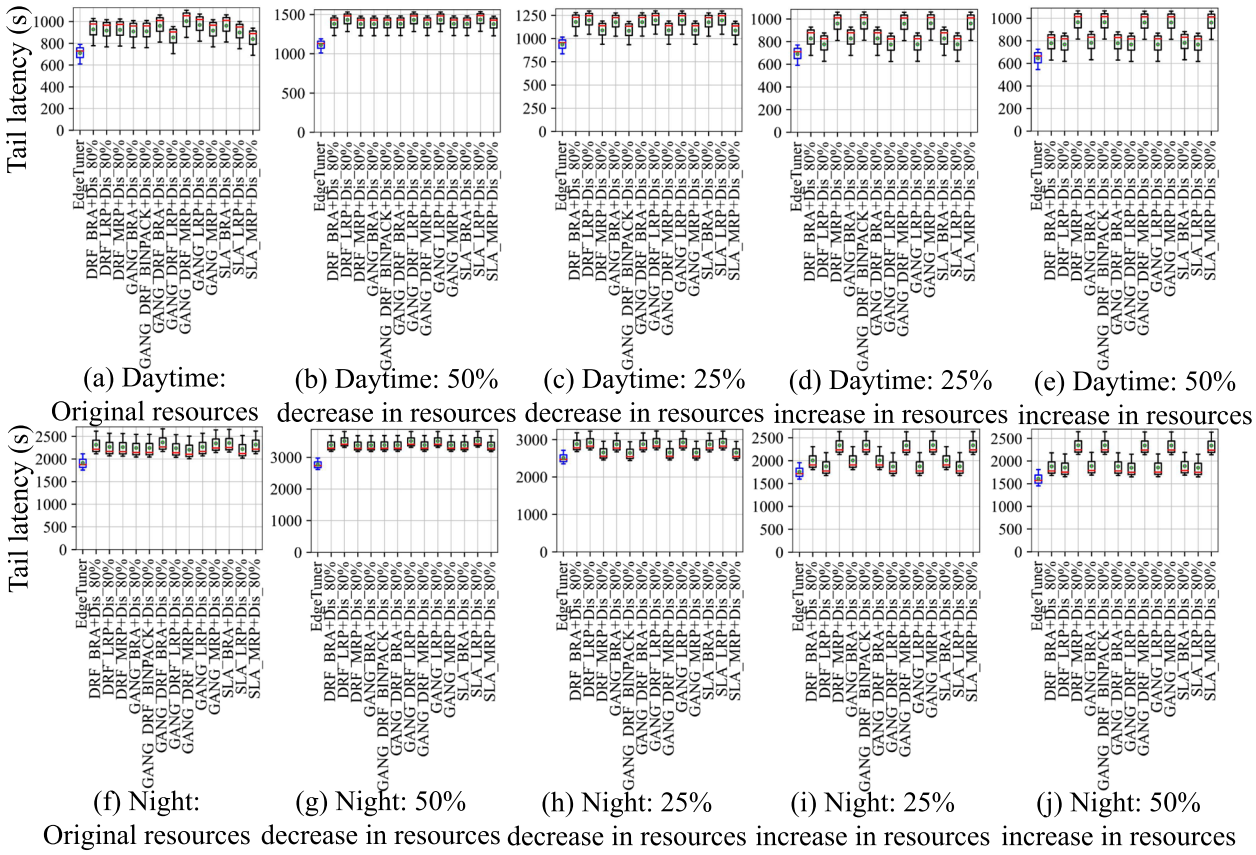


Fig. 18 Comparison of tail latency of AI workloads driven by Alibaba trace 2020 under dynamic resources

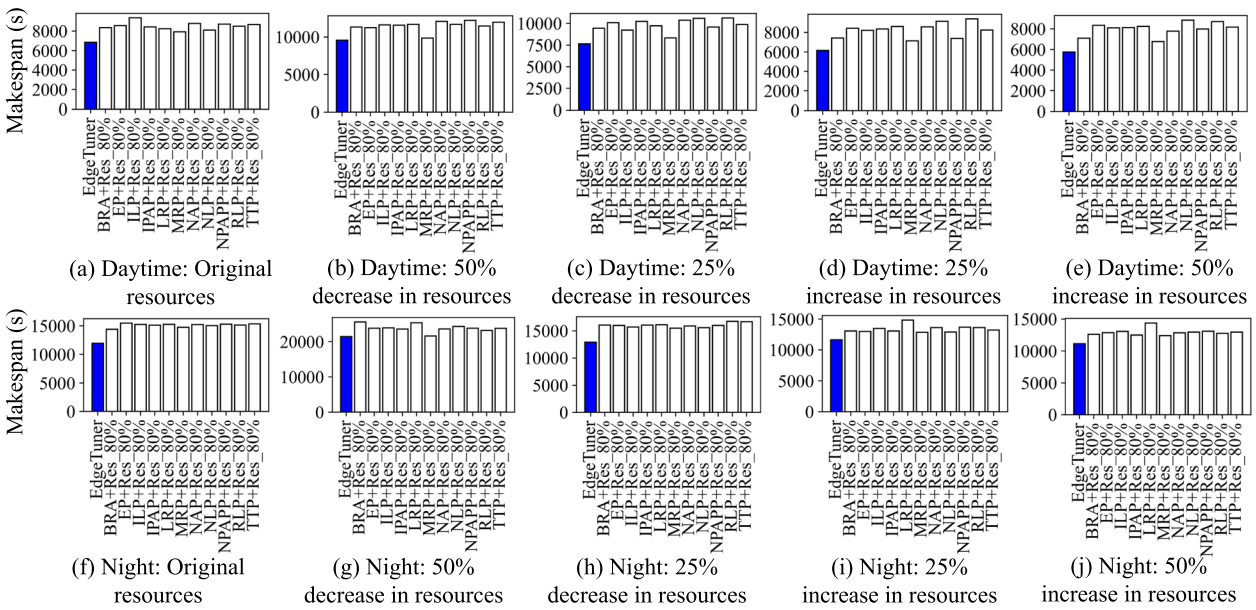


Fig. 19 Comparison of Makespan of DAG batch workloads driven by Alibaba trace 2018 under dynamic resources

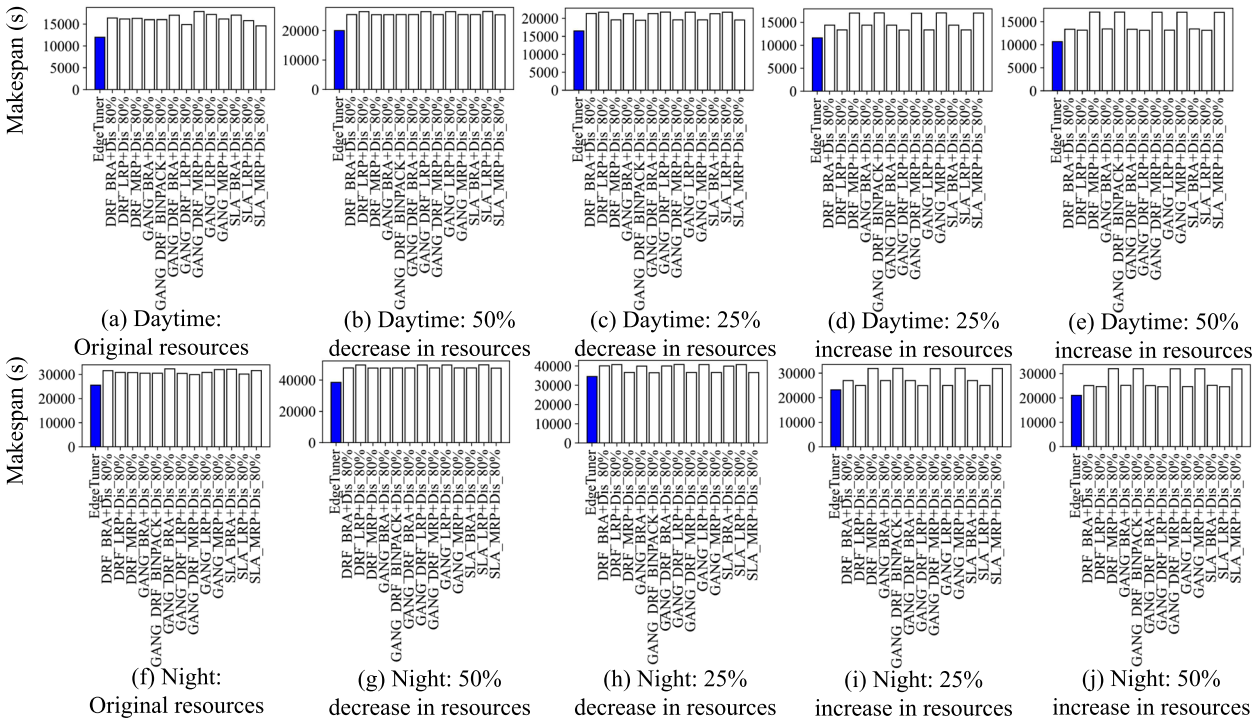


Fig. 20 Comparison of Makespan of AI workloads driven by Alibaba trace 2020 under dynamic resources

Table 7 Scheduling techniques for traditional and AI workloads

Category	Introduction
Cluster resource management systems	Provide scheduling configuration parameters (e.g. Google Borg [66],YARN [65], Mesos [32], Kubernetes [19])
Scheduling optimization for traditional workloads	Apply DRL under a simplified state space (e.g. DeepRM [44])
	Adopt event-driven decision framework to reduce the booming action space (e.g. [9, 13, 41, 70])
	Apply DRL to tackle DAG-based job scheduling problems (e.g. Decima [45], Spear [34])
Scheduling optimization for AI workloads	Take the feature of iterativeness into consideration (e.g. SLAQ [79], Optium [51], OASiS [6])
	Adopt DRL techniques into job scheduling optimization (e.g. Harmony [5], SIREN [67], DSS [57])
	Apply DRL to optimize the task scheduling (e.g. [15, 38, 72])

Cluster resource management systems

Many cluster resource management systems have been developed to allocate available resources to their jobs [16, 25, 26, 45, 49, 83]. For example, Mesos [32] is the first cluster resource management system released by UC Berkeley. Mesos increases the cluster resource utilization using a two-level scheduler, which shares resources among multiple computing frameworks (e.g. Hadoop, Spark and Storm) as well as jobs within each framework. YARN [65] is the next generation of Hadoop compute platform. By separating resource management functions from the programming model, it delegates many scheduling-related functions to per-job components. Google Borg [66] is a pioneer system that divides cloud jobs into high-priority service jobs and low-priority batch jobs, and schedule these jobs with consideration of multiple resource dimensionalities including CPU, memory, disk, and network. In recent years, Google launches a new system (Kubernetes [19]) for the new generation container technology (Docker [47]).

More importantly, they can provide configuration parameters to control the resource allocation mechanism of their schedulers.

Scheduling optimization for traditional workloads

There are lots of research work on traditional workloads' scheduling optimization based on ML/DRL techniques.

For example, DeepRM [44] applies DRL in cluster job scheduling *under a simplified state space*: it assumes a synthetic job arrival process and describes both cluster and job as slots of homogeneous CPU and memory. Hence it can only handle a limited action space: admitting jobs or not. However, a real deployed scheduler usually has a sophisticated state space and a huge action space that decides the allocation of heterogeneous resources to a large number of jobs of different workload characteristics and priorities.

On the other hand, both [70] and [41] adopts *event-driven decision framework* to reduce the booming action space in practice. Here, [41] is a hierachical framework that consists of *global tier* for VM resource allocation and *local tier* for power management of local servers. While [70] mainly targets the constraints of QoS requirements (e.g., average response time). DRL-Cloud [9] improves the energy (e.g. electricity) efficiency of data centers with the consideration of task/data dependencies in task scheduling stage. Du et al. [13] designed a DRL agent that enables simultanueously deciding discrete actions (VMs placement) and continuous actions (dynamic resource pricing).

Besides, both [45] and [34] *apply DRL to tackle DAG-based job scheduling problems*. Here, Decima [45] uses a policy gradient agent and has a similar objective as DeepRM, which is designed to tackle the DAG scheduling problems within each job in Spark, while considering interdependent tasks. Spear [34] works to minimize the makespan of complex DAG-based jobs while considering both task dependencies and heterogeneous resource demands at the same time.

These studies are mostly relied on specific contexts such as power saving or resource pricing, along with diverse constraints defined by users.

Scheduling optimization for AI workloads

Natural characteristics of AI workloads make them different from traditional workloads, so recent years has witnessed and injected such features into job scheduling optimization. For example, SLAQ [79], Optium [51] and OASiS [6] *take the feature of iterativeness into consideration*, and separately propose their online prediction method for predicting the overheads of coming iterations in each iterative step. Difference among them is that SLAQ mainly studies the connection between job latency and model quality, and advocates placing more resources to jobs with a great potential of quality improvement; Optium is designed for DL workloads, and it further enables saving communication cost while improving training efficiency; OASiS is aiming at dynamically controlling the number of concurrent workers and parameter servers for each job to get a higher resource utilization and training expedition.

Some other approaches *adopt DRL techniques into job scheduling optimization*. For example, Harmony [5] implicitly encodes interferences among co-located ML jobs as one of inputs of neural network that maps raw cluster and job states to job placement decisions (workers and PSs allocation). SIREN [67] abstracts a ML job as a set of serverless functions (e.g. AWS Lambda functions) and leverages DRL techniques to adjust the number and memory of such functions. DSS [57] is an automated big-data task scheduling approach in cloud computing environments, which combines DRL and LSTM to automatically predict the VMs to which each incoming big data job should be scheduled to improve the performance of big data analytics while reducing the resource execution cost. They mostly consider the job placement in a manner of Job-to-VMs, whereas we target changing the scheduler configurations in a run-time mode.

Besides, DRL is also applied to *optimize the task scheduling.* For example, Fang et al. [15] propose an advanced deep reinforcement learning (RL) approach (that learns to schedule from experience) to better schedule DNN inference queries, and Li et al. [38] develop a novel and highly effective DRL-based control framework for distributed streaming data processing. Wu et al. [72] present an optimal task allocation scheme with a virtual network mapping algorithm based on deep CNN and value-function based Q-learning. Note that they mostly consider task scheduling scenarios with fixed workloads and resources, whereas we consider such scenarios with dynamically changing workloads and resources.

Our Work. In this paper, we focuses on edge-cloud jobs with diverse workload characteristics (such as stochastic arrival rate, different resource demands and durations). Based on popular Kubernetes scheduler and Volcano scheduler in real Kubernetes cluster, our approach is built upon these two existing configurable schedulers. Simultaneously, we employ DRL to tune their configuration combinations online. On the one hand, early work in this area adopts reinforcement learning (RL) techniques to schedule jobs at particular time slots, so as to minimize their latencies [44]. On the other hand, in order to accelerate the training speed, later techniques in this area applies state-of-the-art DRL techniques [40, 63]. However, when the cluster schedules edge-cloud jobs by using DRL to tune scheduler configurations at runtime, it still suffers from the **time-consuming challenge of the sampling phases**, which is mainly because: (1)in order to achieve convergence, the DRL training needs a large number of samples (e.g. over 1 million); (2)in order to obtain a training sample, it also takes a rather long time (at least a few seconds).

To address the above challenges, some recent approaches train DRL agents in an offline manner [75, 76]. Note that the DRL training is driven by a neural network based computational model, which predicts system states and generates training samples based on history traces. Similarly, Ran et.al [54] develop a simulation platform called DeepEE, which is used to emulate dynamic IT workloads and cooling systems. More crucially, these techniques target *long-running and compute-intensive jobs* in HPC data centers. And compared with our work's scenario, there are two key differences: (i)the long-running jobs follow an arrival queue, and are dispatched to proper servers through **a fixed job scheduling algorithm**. (ii)for these compute-intensive jobs in their scenarios, latency is not a key concern. (iii)in DRL training, these techniques implicitly assume **pre-defined available resources** in the cluster.

Conclusion

In this paper, we propose EdgeTuner, a fast scheduler configuration tuning approach that efficiently leverages DRL to reduce tail latency of edge-cloud jobs. It can timely train a DRL agent in order to properly tune scheduler configurations in dynamic edge-cloud environment. We implement EdgeTuner on both Kubernetes and Volcano schedulers and extensively evaluate it on real workloads driven by Alibaba production traces. The experimental results show that EdgeTuner outperforms prevailing scheduling algorithms by achieving much lower tail latency while accelerating DRL training speed.

Acknowledgements
The authors would like to thank all anonymous reviewers for their invaluable comments.

Authors' contributions
Shilin Wen and Rui Han wrote the main manuscript text. Chi Harold Liu and Lydia Y. Chen prepared the main figures and tables. All authors reviewed the manuscript. The author(s) read and approved the final manuscript.

Authors' information
Shilin Wen is a PhD student at the School of Computer Science and Technology, Beijing Institute of Technology. His work focuses on optimization of machine learning, big data system for edge computing systems.
Rui Han is a Full Professor at the School of Computer Science and Technology, Beijing Institute of Technology, China.
Chi Harold Liu receives the Ph.D. degree from Imperial College, UK in 2010, and the B.Eng. degree from Tsinghua University, China in 2006. He is currently a Full Professor and Vice Dean at the School of Computer Science and Technology, Beijing Institute of Technology, China.
Lydia Y. Chen received the BA degree from National Taiwan University, and the PhD from Pennsylvania State University. She is currently an associate professor with the Department of Computer Science, Technology University Delft. Prior to joining TU Delft, she was a research staff member with the IBM Zurich Research Lab from 2007 to 2018.

Funding
This work is supported by the National Natural Science Foundation of China (Grant No. 62272046, 62132019, 61872337) and Shandong Provincial Natural Science Foundation (Grant No. ZR2020MF034).

Availability of data and materials
The data and materials used to support the findings of this study are available from the corresponding author upon request.

Declarations

Ethics approval and consent to participate
This declaration is "not applicable".

Competing interests
The authors declare no competing interests.

Received: 31 October 2022 Accepted: 26 May 2023
Published online: 17 June 2023

References

1. Abadi M, Barham P, Chen J, Chen Z, Davis A, Dean J, Devin M, Ghemawat S, Irving G, Isard M et al (2016) {TensorFlow}: a system for {Large-Scale} machine learning. In: 12th USENIX symposium on operating systems design and implementation (OSDI 16). USENIX Association, pp 265–283
2. Alibaba (2018) Alitrace. https://github.com/alibaba/clusterdata/tree/master/cluster-trace-v2018. Accessed 10 Oct 2022
3. Alibaba (2020) Alitrace. https://github.com/alibaba/clusterdata/tree/master/cluster-trace-gpu-v2020. Accessed 10 Oct 2022
4. Arias J, Gamez JA, Puerta JM (2017) Learning distributed discrete bayesian network classifiers under MapReduce with apache spark. Knowl-Based Syst 117:16–26
5. Bao Y, Peng Y, Wu C (2019) Deep learning-based job placement in distributed machine learning clusters. In: IEEE INFOCOM 2019-IEEE Conference on Computer Communications. IEEE, pp 505–513
6. Bao Y, Peng Y, Wu C, Li Z (2018) Online job scheduling in distributed machine learning clusters. In: IEEE INFOCOM 2018-IEEE Conference on Computer Communications. IEEE, pp 495–503
7. Carrión C (2022) Kubernetes scheduling: Taxonomy, ongoing issues and challenges. ACM Comput Surv 55(7):138:1–138:37
8. Chekired DA, Togou MA, Khoukhi L, Ksentini A (2019) 5g-slicing-enabled scalable sdn core network: Toward an ultra-low latency of autonomous driving service. IEEE J Sel Areas Commun 37(8):1769–1782
9. Cheng M, Li J, Nazarian S (2018) Drl-cloud: Deep reinforcement learning-based resource provisioning and task scheduling for cloud service providers. In: Proceedings of the 23rd Asia and South Pacific Design Automation Conference. IEEE Press, pp 129–134
10. Cloud Native Computing Foundation (CNCF). Volcano: Kubernetes native batch system. https://volcano.sh/en/. Accessed 10 Oct 2022
11. Dabney W, Ostrovski G, Silver D, Munos R (2018) Implicit quantile networks for distributional reinforcement learning. In: International conference on machine learning. PMLR, pp 1096–1105
12. Dean J, Barroso LA (2013) The tail at scale. Commun ACM 56(2):74–80
13. Du B, Wu C, Huang Z (2019) Learning resource allocation and pricing for cloud profit maximization. In: The Thirty-Third AAAI Conference on Artificial Intelligence (AAAI-19). AAAI Press, pp 7570–7577
14. Du Z, Sun H, He Y, He Y, Bader DA, Zhang H (2013) Energy-efficient scheduling for best-effort interactive services to achieve high response quality. In: 2013 IEEE 27th International Symposium on Parallel and Distributed Processing. IEEE, pp 637–648
15. Fang Z, Yu T, Mengshoel OJ, Gupta RK (2017) Qos-aware scheduling of heterogeneous servers for inference in deep neural networks. In: CIKM'17. ACM, pp 2067–2070
16. Garefalakis P, Karanasos K, Pietzuch P, Suresh A, Rao S (2018) Medea: scheduling of long running applications in shared production clusters. In: Proceedings of the thirteenth EuroSys conference. ACM, pp 1–13
17. Ghodsi A, Zaharia M, Hindman B, Konwinski A, Shenker S, Stoica I (2011) Dominant resource fairness: Fair allocation of multiple resource types. In: 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11). USENIX Association, pp 24–24

18. Gianniti E, Rizzi AM, Barbierato E, Gribaudo M, Ardagna D (2017) Fluid petri nets for the performance evaluation of MapReduce and spark applications. ACM SIGMETRICS Perform Eval Rev 44(4):23–36
19. Google. Google kubernetes. https://kubernetes.io. Accessed 10 Oct 2022
20. Han R, Ghanem M M, Guo L, et al (2014) Enabling cost-aware and adaptive elasticity of multi-tier cloud applications. Future Gen Comput Syst 32:82–98
21. Han R, Guo L, Ghanem M M, et al (2012) Lightweight resource scaling for cloud applications. In: 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012). IEEE, pp 644–651
22. Han R, Huang S, Wang Z, Zhan J (2017) Clap: Component-level approximate processing for low tail latency and high result accuracy in cloud online services. IEEE Trans Parallel Distrib Syst 28(8):2190–2203
23. Han R, John L K, Zhan J (2017) Benchmarking big data systems: A review. IEEE Trans Serv Comput 11(3):580–597
24. Han R, Liu CH, Li S, Wen S, Liu X (2020) Accelerating deep learning systems via critical set identification and model compression. IEEE Trans Comput 69(7):1059–1070
25. Han R, Liu CH, Zong Z, Chen LY, Liu W, Wang S, Zhan J (2019) Workload-adaptive configuration tuning for hierarchical cloud schedulers. IEEE Trans Parallel and Distrib Syst 30(12):2879–2895
26. Han R, Wen S, Liu C H, et al (2022) EdgeTuner: Fast Scheduling Algorithm Tuning for Dynamic Edge-Cloud Workloads and Resources. In: IEEE INFOCOM 2022-IEEE Conference on Computer Communications. IEEE, pp 880–889
27. Han R, Zhang Q, Liu CH, Wang G, Tang J, Chen LY (2021) Legodnn: block-grained scaling of deep neural networks for mobile vision. In: MobiCom'21. ACM, pp 406–419
28. Hao Z, Yi S, Li Q (2019) Nomad: An efficient consensus approach for latency-sensitive edge-cloud applications. In: INFOCOM'19. IEEE, pp 2539–2547
29. He Y, Elnikety S, Larus J, Yan C (2012) Zeta: Scheduling interactive services with partial execution. In: Proceedings of the Third ACM Symposium on Cloud Computing. ACM, pp 1–14
30. He Y, Elnikety S, Sun H (2011) Tians scheduling: Using partial processing in best-effort applications. In: 2011 31st International Conference on Distributed Computing Systems. IEEE, pp 434–445
31. Hessel M, Modayil J, Van Hasselt H, Schaul T, Ostrovski G, Dabney W, Horgan D, Piot B, Azar M, Silver D (2018) Rainbow: Combining improvements in deep reinforcement learning. In: Thirty-second AAAI conference on artificial intelligence. AAAI Press, pp 3215–3222
32. Hindman B, Konwinski A, Zaharia M, Ghodsi A, Joseph AD, Katz R, Shenker S, Stoica I (2011) Mesos: A platform for {Fine-Grained} resource sharing in the data center. In: 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11). USENIX Association, pp 22–22
33. Horgan D, Quan J, Budden D, Barth-Maron G, Hessel M, Van Hasselt H, Silver D (2018) Distributed prioritized experience replay. arXiv preprint arXiv:1803.00933
34. Hu Z, Tu J, Li B (2019) Spear: Optimized dependency-aware task scheduling with deep reinforcement learning. In: 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS). IEEE, pp 2037–2046
35. Jalaparti V, Bodik P, Kandula S, Menache I, Rybalkin M, Yan C (2013) Speeding up distributed request-response workflows. ACM SIGCOMM Comput Commun Rev 43(4):219–230
36. Kadkhodaei H, Moghadam AME, Dehghan M (2021) Big data classification using heterogeneous ensemble classifiers in apache spark based on MapReduce paradigm. Expert Syst Appl 183:115369
37. Kulshrestha T, Saxena D, Niyogi R, Cao J (2019) Real-time crowd monitoring using seamless indoor-outdoor localization. IEEE Trans Mob Comput 19(3):664–679
38. Li T, Xu Z, Tang J, Wang Y (2018) Model-free control for distributed stream data processing using deep reinforcement learning. Proc VLDB Endowment 11(6):705–718
39. Liang E, Liaw R, Nishihara R, Moritz P, Fox R, Gonzalez J, Goldberg K, Stoica I (2017) Ray rllib: A composable and scalable reinforcement learning library, vol 85. arXiv preprint arXiv:1712.09381
40. Liu N, Li Z, Xu J, Xu Z, Lin S, Qiu Q, Tang J, Wang Y (2017) A hierarchical framework of cloud resource allocation and power management using deep reinforcement learning. In: ICDCS'17. IEEE Computer Society, pp 372–382
41. Liu N, Li Z, Xu J, Xu Z, Lin S, Qiu Q, Tang J, Wang Y (2017) A hierarchical framework of cloud resource allocation and power management using deep reinforcement learning. In: 2017 IEEE 37th international conference on distributed computing systems (ICDCS). IEEE, pp 372–382
42. Liu Y, Zhou Y, Hu S (2017) Combating coordinated pricing cyberattack and energy theft in smart home cyber-physical systems. IEEE Trans Comput-Aided Des Integr Circ Syst 37(3):573–586
43. Ma X, Zhou A, Zhang S, Wang S (2020) Cooperative service caching and workload scheduling in mobile edge computing. In: INFOCOM'20. IEEE, pp 2076–2085
44. Mao H, Alizadeh M, Menache I, Kandula S (2016) Resource management with deep reinforcement learning. In: Proceedings of the 15th ACM workshop on hot topics in networks. ACM, pp 50–56
45. Mao H, Schwarzkopf M, Venkatakrishnan SB, Meng Z, Alizadeh M (2019) Learning scheduling algorithms for data processing clusters. In: Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM 2019. ACM, pp 270–288
46. Mehnaz S, Bertino E (2020) Privacy-preserving real-time anomaly detection using edge computing. In: ICDE'20. IEEE, pp 469–480
47. Merkel D (2014) Docker: lightweight linux containers for consistent development and deployment. Linux J 2014(239):2
48. Mirobi GJ, Arockiam L (2015) Service level agreement in cloud computing: An overview. In: 2015 International Conference on Control, Instrumentation, Communication and Computational Technologies (ICCICCT). IEEE, pp 753–758
49. Park JW, Tumanov A, Jiang A, Kozuch MA, Ganger GR (2018) 3sigma: distribution-based cluster scheduling for runtime uncertainty. In: Proceedings of the Thirteenth EuroSys Conference. ACM, pp 1–17
50. Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, Killeen T, Lin Z, Gimelshein N, Antiga L et al (2019) Pytorch: An imperative style, high-performance deep learning library. In: Annual Conference onNeural Information Processing Systems 2019 (NeurIPS 2019). NIPS, pp 8024–8035
51. Peng Y, Bao Y, Chen Y, Wu C, Guo C (2018) Optimus: an efficient dynamic resource scheduler for deep learning clusters. In: Proceedings of the Thirteenth EuroSys Conference. ACM, pp 1–14
52. Phan A-C, Cao H-P, Tran H-D, Phan T-C (2019) Face recognition using gabor wavelet in mapreduce and spark. In: World Congress on Global Optimization. Springer, pp 769–778
53. Phan A-C, Tran H-D, Phan T-C (2018) Fingerprint recognition using gabor wavelet in mapreduce and spark. In: Proceedings of the Ninth International Symposium on Information and Communication Technology. ACM, pp 54–60
54. Ran Y, Hu H, Zhou X, Wen Y (2019) Deepee: Joint optimization of job scheduling and cooling control for data center energy efficiency using deep reinforcement learning. In: 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS). IEEE, pp 645–655
55. Rattanaopas K (2017) A performance comparison of apache tez and mapreduce with data compression on hadoop cluster. In: 2017 14th International Joint Conference on Computer Science and Software Engineering (JCSSE). IEEE, pp 1–5
56. Requeno JI, Gascón I, Merseguer J (2018) Towards the performance analysis of apache tez applications. In: Companion of the 2018 ACM/SPEC International Conference on Performance Engineering. ACM, pp 147–152
57. Rjoub G, Bentahar J, Wahab OA, Bataineh A (2019) Deep smart scheduling: A deep learning approach for automated big data scheduling over the cloud. In: 2019 7th International Conference on Future Internet of Things and Cloud (FiCloud). IEEE, pp 189–196
58. Shi W, Cao J, Zhang Q, Li Y, Xu L (2016) Edge computing: Vision and challenges. IEEE Internet Things J 3(5):637–646
59. Suresh L, Canini M, Schmid S, Feldmann A (2015) C3: Cutting tail latency in cloud data stores via adaptive replica selection. In: 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15). USENIX Association, pp 513–527
60. Sutton RS, Barto AG (2018) Reinforcement learning: An introduction. MIT Press
61. Tan B, Ma H, Mei Y, Zhang M (2020) A cooperative coevolution genetic programming hyper-heuristic approach for on-line resource allocation in container-based clouds. IEEE Trans Cloud Comput 10(3):1500–1514
62. Tekdogan T, Cakmak A (2021) Benchmarking apache spark and hadoop mapreduce on big data classification. In: 2021 5th International Conference on Cloud and Big Data Computing (ICCBDC). ACM, pp 15–20

63. Toromanoff M, Wirbel E, Moutarde F (2019) Is deep reinforcement learning really superhuman on atari? leveling the playing field. arXiv preprint arXiv:1908.04683
64. Tsai C-W, Huang W-C, Chiang M-H, Chiang M-C, Yang C-S (2014) A hyper-heuristic scheduling algorithm for cloud. IEEE Trans Cloud Comput 2(2):236–250
65. Vavilapalli VK, Murthy AC, Douglas C, Agarwal S, Konar M, Evans R, Graves T, Lowe J, Shah H, Seth S et al (2013) Apache hadoop yarn: Yet another resource negotiator. In: Proceedings of the 4th annual Symposium on Cloud Computing. ACM, pp 1–16
66. Verma A, Pedrosa L, Korupolu M, Oppenheimer D, Tune E, Wilkes J (2015) Large-scale cluster management at google with borg. In: EuroSys'15. ACM, p 18
67. Wang H, Niu D, Li B (2019) Distributed machine learning with a serverless architecture. In: IEEE INFOCOM 2019-IEEE Conference on Computer Communications. IEEE, pp 1288–1296
68. Wang J, Zhang J, Bao W, Zhu X, Cao B, Yu PS (2018) Not just privacy: Improving performance of private deep learning in mobile cloud. In: SIGKDD'18. ACM, pp 2407–2416
69. Wang S, Yang S, Zhao C (2020) Surveiledge: Real-time video query based on collaborative cloud-edge deep learning. INFOCOM 2020:2519–2528
70. Wei Y, Pan L, Liu S, Wu L, Meng X (2018) DRL-scheduling: An intelligent Qos-aware job scheduling framework for applications in clouds. IEEE Access 6:55112–55125
71. Wiseman Y, Feitelson DG (2003) Paired gang scheduling. IEEE Trans Parallel Dist Syst 14(6):581–592
72. Wu C, Xu G, Ding Y, Zhao J (2019) Explore deep neural network and reinforcement learning to large-scale tasks processing in big data. Int J Pattern Recog Artif Intell 33(13):1951010
73. Xiao Q-z, Zhong J, Feng L, Luo L, Lv J (2019) A cooperative coevolution hyper-heuristic framework for workflow scheduling problem. IEEE Trans Serv Comput 15(1):150–163
74. Yang (2019) Aligraph: A comprehensive graph neural network platform. In: Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining. ACM, pp 3165–3166
75. Yi D, Zhou X, Wen Y, Tan R (2019) Toward efficient compute-intensive job allocation for green data centers: A deep reinforcement learning approach. In: ICDCS'19. IEEE, pp 634–644
76. Yi D, Zhou X, Wen Y, Tan R (2020) Efficient compute-intensive job allocation in data centers via deep reinforcement learning. IEEE Trans Parallel Distrib Syst 31(6):1474–1485
77. Yi D, Zhou X, Wen Y, Tan R (2019) Toward efficient compute-intensive job allocation for green data centers: A deep reinforcement learning approach. In: 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS). IEEE, pp 634–644
78. Yun J-M, He Y, Elnikety S, Ren S (2015) Optimal aggregation policy for reducing tail latency of web search. In: Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval. ACM, pp 63–72
79. Zhang H, Stafman L, Or A, Freedman MJ (2017) Slaq: quality-driven scheduling for distributed machine learning. In: Proceedings of the 2017 Symposium on Cloud Computing. ACM, pp 390–404
80. Zhang Q, Zhang Q, Shi W, Zhong H (2018) Firework: Data processing and sharing for hybrid cloud-edge analytics. IEEE Trans Parallel Distrib Syst 29(9):2004–2017
81. Zhang Y, Sheng VS (2019) Fog-enabled event processing based on IoT resource models. IEEE Trans Knowl Data Eng 31(9):1707–1721
82. Zhao Z, Barijough KM, Gerstlauer A (2018) Deepthings: Distributed adaptive deep learning inference on resource-constrained IoT edge clusters. IEEE Trans Comput-Aided Des Integr Circ Syst 37(11):2348–2359
83. Zong Z, Wen L, Hu X, Han R, Qian C, Lin L (2021) Mespaconfig: Memory-sparing configuration auto-tuning for co-located in-memory cluster computing jobs. IEEE Trans Serv Comput 15(5):2883–2896

Publisher's Note