Cluster Computing (2025)28:179
https://doi.org/10.1007/s10586-024-04828-2

Reinforcement learning-based task scheduling for heterogeneous computing in end-edge-cloud environment

Wangbo Shen[1] · Weiwei Lin[1,2] · Wentai Wu[3] · Haijie Wu[1] · Keqin Li[4]

Received: 12 June 2024 / Revised: 28 September 2024 / Accepted: 16 October 2024
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2024

Abstract

The End-Edge-Cloud (EEC) computing framework can offer low-latency, high-quality services to users of diverse demands by leveraging pervasive resources. However, the inherent disparities in task requirements and the strong heterogeneity of computational resources in these systems make it non-trivial for scheduler design, particularly in high load scenarios (e.g. burst of tasks). This also complicates the adaptation of traditional cloud-oriented schedulers considering their limited support of heterogeneous processors and accelerators (e.g., CPUs, GPUs and NPUs). In light of this, we first present a system framework for task scheduling in the EEC architecture. In the framework we adopt a reinforcement learning (RL)-based scheduler tailored for reducing task completion time and waiting time. Our method integrates task characteristics and environmental constraints within matrices, based on which an adapted Q-Learning agent is employed for decision making. We then introduce the implementation of our framework that features Kubernetes and Rancher-based coordination with extended support for heterogeneous processing units. Experimentally we built a real-world EEC testbed comprising PC, Atlas 200 DK, and Raspberry PI devices. Evaluation results of our algorithm demonstrate a 271% enhancement in performance compared to existing algorithms in the context of burst-arrival task queues, which underscores the efficacy of our solution in realistic scenarios.

Keywords End-Edge-Cloud computing framework · Heterogeneous computing · Machine learning · Reinforcement learning

1 Introduction

THE advent of the End-Edge-Cloud (EEC) computing framework, propelled by advancements in the Industrial Internet of Things (IIoT) and 5G technologies, holds the promise of transforming computing services and elevating Artificial Intelligence (AI) applications in terms of efficiency and user experience. Nonetheless, the inherent complexity of EEC systems poses significant challenges, particularly in task scheduling-a critical aspect for maximizing system performance amidst diverse hardware architectures and computational capacities [1–3].

The heterogeneity of computational resources presents a primary challenge in the EEC computing framework. The diversity of hardware architectures, including general-purpose processors and specialized AI accelerators, alongside varying task requirements and computational capacities, complicates unified resource management essential for efficient system collaboration. Task scheduling, aiming to match tasks with appropriate resources to enhance system performance, becomes critically complex in this context [4].

Despite this complexity, current scheduling methods frequently fail to account for the distinct demands of tasks, particularly those in AI applications with strong device affinity. For example, Zhou et al. [5] addressed task scheduling within a space-air-ground integrated network

✉ Weiwei Lin
linww@scut.edu.cn

1 School of Computer Science and Engineering, South China University of Technology, Guangzhou 510006, China

2 Pengcheng Laboratory, Shenzhen 518000, China

3 Department of Computer Science, College of Information Science and Technology, Jinan University, Guangzhou 510632, China

4 Department of Computer Science, State University of New York, New York 12561, USA

Published online: 21 January 2025

for IoT services, developing a policy to reduce task offloading and computation delays while considering energy constraints of UAVs. Furthermore, the application of metaheuristic approaches, like the whale optimization algorithm to cloud task scheduling [6], underscores efforts to adapt to specific computing environments. Nevertheless, such methods often do not fully embrace the unique intricacies of AI tasks, a gap highlighted in comprehensive reviews of scheduling techniques [7].

Mainstream research has not adequately addressed the heterogeneity of computing resources, a gap that is particularly apparent when considering the disparate computational strengths of various computing chips such as CPUs, GPUs, and NPUs. Innovative attempts have been made, such as the multi-queue scheduling method for managing temporal disparities in hybrid clouds with heterogeneous tasks [8], and efforts to enhance security quality in heterogeneous multiprocessor systems within IoT frameworks [9]. These endeavors are discussed in a comprehensive review [10], which highlights a critical need for advanced scheduling research in this area.

This urgency is compounded by the practical challenges in designing and implementing a scheduling system that can manage a myriad of server clusters, each with its own set of heterogeneous computing devices for EEC applications. Current state-of-the-art cloud platforms, like Kubernetes, are tailored predominantly for single-server cluster management and do not support the intricate task distribution and resource management needed for multi-cluster EEC scenarios. Furthermore, the ability to oversee a diverse range of computing resources beyond CPUs-such as GPUs, NPUs, and FPGAs-is conspicuously absent, posing a significant barrier to the effective allocation and management of computational tasks within EEC infrastructures. This deficiency calls for a substantial enhancement in cloud computing management platforms and a reimagined approach to heterogeneous resource control in research.

Considering the importance of real-time task scheduling for EEC systems that prioritize service response time, another main challenge for scheduler design is how to make online decisions in a vast decision space. To address this, we propose a novel reinforcement learning-based strategy tailored for real-time EEC scheduling capable of handling the heterogeneity of computing resources. Our method employs an adaptive Q-learning algorithm for immediate resource allocation, prioritizing urgent tasks via an intelligent queuing system. It takes into account varied compute capabilities and network conditions, aiming to optimize task distribution and minimize delays.

We conducted experiments in a testbed where we implemented our EEC framework and the scheduler over commercial devices including PC/Atlas 200 DK/Raspberry PI. The also system was build upon Kubernetes and Rancher. The results show that our method outperforms mainstream task scheduling algorithms especially in the case of dense task arrivals (performance improvement by 271%), which manifests the capability of our solution in handling bursts of tasks in EEC systems.

In summary, we make the following key contributions:

1. We propose an easy-to-implement, RL-based scheduling algorithm specifically designed to effectively utilize heterogeneous resources (e.g., AI accelerators) within EEC systems. Our algorithm is optimized for minimizing task completion time and waiting time.
2. We developed an EEC system framework, namely EECRL, based on extended Kubernetes and Rancher platforms. The proposed algorithm serves as the scheduler in our framework to empower resource-efficient coordination within the EEC architecture.
3. Through experiments on a real-world testbed we demonstrate that our RL-based scheduling algorithm outperforms existing approaches under different settings of task arrival rate.

The remainder of this paper is structured as follows: Sect. 2 provides background information, introducing the task scheduling framework, heterogeneous computing task scheduling problem, and research on reinforcement learning in related fields. The proposed model is detailed in Sects. 3, 4, and 5. Section 6 conducts experiments and analyzes performance. Finally, Sect. 7 concludes the paper.

2 Background and related work

2.1 Orchestration frameworks

Kubernetes, like other mainstream cloud computing technologies, is a leading orchestration platform widely used for resource monitoring and task scheduling due to its high-performance, fairness-based schedulers. However, it has two critical limitations in EEC scenarios [11]:

1. **Single-Cluster Scheduling:** Kubernetes is designed for single-cluster scheduling, which is suitable for cloud server or edge server cluster scenarios. However, EEC environments involve multiple clusters (end, edge, cloud), requiring cross-cluster task scheduling and monitoring. This limitation makes Kubernetes less effective for EEC scenarios.
2. **Limited Resource Support:** The default Kubernetes schedulers only support monitoring and scheduling of CPU resources. Even with third-party support, it currently only handles CPU and GPU resources. EEC architectures include various heterogeneous computing

resources such as CPUs, GPUs, NPUs, TPUs, and FPGAs, which Kubernetes' default scheduling algorithms cannot efficiently manage and coordinate.

To address these challenges, this paper employs the Rancher platform [12], which supports cross-cluster task scheduling and resource management through APIs. Additionally, it uses direct socket communication with nodes and driver commands like `npu-smi` and `nvidia-smi` to monitor various heterogeneous computing resources, compensating for Kubernetes' limitations in managing heterogeneous resources.

2.2 Heterogeneous task scheduling in EEC systems

In the EEC scenario, beyond establishing a framework capable of cross-cluster task scheduling and monitoring heterogeneous computing resources, designing an efficient scheduling algorithm is a crucial component for effectively completing computational tasks. Currently, several excellent scheduling algorithms have been developed for edge computing, cloud computing, and similar scenarios. For instance, [13] introduced Gavel, a heterogeneity-aware scheduler that generalizes existing scheduling policies into optimization problems and transforms them into heterogeneity-aware versions. Additionally, [14] studied a two-month workload trace from a production MLaaS cluster at Alibaba with over 6000 GPUs, highlighting the challenges in cluster scheduling. [15] proposed a framework for heterogeneous computation and resource allocation to effectively implement FL, aiming to minimize energy consumption and maximize energy harvesting for smart devices by considering multiple controls. Similarly, [16] proposed a strategy for cost-efficient container orchestration through heterogeneous task allocation, focusing on resource utilization optimization and elastic instance pricing. Furthermore, [17] introduced a KNN-based scheduler that starts with speculative prefetching, performs KNN clustering on the intermediate map output, and then directs it to the reducer for final processing.

These studies have addressed the heterogeneity of devices and tasks, using advanced algorithms and machine learning models to predict resource usage patterns and facilitate real-time scheduling decisions. However, EEC scenarios differ from edge computing in that heterogeneity is not only present in computing chips and tasks but also in the computing capabilities of devices across the end, edge, and cloud layers. This aspect is often overlooked in current research, leading to significant computational power waste in the EEC framework, as shown in Fig. 1. Therefore, this paper comprehensively considers the heterogeneity in devices, chips, and computational power within the EEC architecture to design Reinforcement Learning-based Scheduling algorithms that achieve unified scheduling of computational power and efficient completion of heterogeneous tasks.

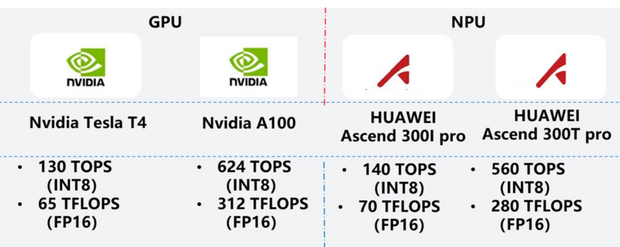


Fig. 1 EEC scenarios differ from edge computing in that heterogeneity is not only present in computing chips and tasks but also in the computing capabilities of devices across the end, edge, and cloud layers

2.3 Reinforcement learning-based scheduling

Over the past few years, reinforcement learning (RL), a subset of machine learning, has proven to be effective for enhancing scheduling algorithms [18]. RL enables an agent to interact with its environment, receive rewards for actions, and learn from these interactions. This approach has been successfully applied in various fields, including gaming, robotics, and resource management [19].

Applied to scheduling, RL can dynamically adjust scheduling policies based on system state changes, leading to more efficient resource use and improved system performance.

Many studies have explored RL in scheduling algorithms. For example, researchers have developed RL-based scheduling for cloud computing to reduce makespan or balance loads across servers. RL has also been used for task scheduling in heterogeneous computing systems to maximize resource utilization or minimize energy consumption. Notable contributions include [20], which developed an RL model for task scheduling in a cloud-based Spark cluster considering SLA objectives, [21], which proposed an ML job feature-based scheduling system (MLFS) for ML clusters, and [22], which introduced RLScheduler, an automated HPC batch job scheduler using RL.

However, current RL models for scheduling are mostly focused on cloud and edge computing, with few studies addressing EEC scenarios. Additionally, there is a lack of training environment design for EEC. This paper introduces a reinforcement learning interactive environment and model specifically designed for EEC scenarios.

3 Problem formulation

In our EEC architecture, the hierarchical system is organized in clusters and each cluster typically encompasses a variety of heterogeneous compute resources that differ in compute power, architecture, and chip models. Formally, we represent the features of each node in each cluster as $E = [C_k, C_m, G_k, G_m, N_k, N_m, \ldots, Disk, Net]$ where $C_k, G_k, \ldots$ denote the number of available cores for heterogeneous processors/accelerators such as CPUs, GPUs, etc. $C_m, G_m, \ldots$ represent the available memory for the corresponding modules. *Disk* refers to the available storage space, and *Net* is the device's network communication capacity.

Based on resource requirements, tasks in the system are manually grouped into six categories during the preprocessing stage: AI-intensive, memory-intensive, network-intensive, storage-intensive, I/O-intensive, and CPU-intensive. This classification is done ahead of time by system administrators or developers based on the specific characteristics and resource needs of each task. For instance, AI-intensive tasks typically require specialized accelerators (e.g., GPUs and NPUs), and are modeled in a unified way to ensure efficient scheduling. By manually categorizing tasks during preprocessing, the system can optimize resource allocation and ensure that each task is assigned to the most appropriate resources within the heterogeneous environment.

In heterogeneous computing, the main complexity lies in the fact that an AI-intensive task may be able to run on different types of computational units and the resulting performance can differ a lot. For instance, a CNN model training and inference task for image recognition can be run solely on CPU, CPU combined with high-performance chips like GPU, NPU and TPU, or using FPGA for optimized parallel computation. The required computational resources and execution efficiency for each option can vary significantly due to the difference in compute power and their compatibility with the task. To cover these factors, we represent the set of tasks as $T = < T_1, T_2, \ldots, T_k >$, where $T_i = [C^i_{rk}, C^i_{rm}, G^i_{rk}, G^i_{rm}, N^i_{rk}, N^i_{rm}, .., Disk^i_r, Runtime_i]$. In this context, k represents the type of heterogeneous resource that not only depends on processor or accelerator class (e.g., NPU, GPU, TPU) but also on their models such as Nvidia 1080ti, 3090, among others. And $C^i_{rk}, G^i_{rk}, \ldots$ represent the estimated core requirements on heterogeneous chips like GPUs, NPUs, and FPGAs. For GPUs, cores are managed using CUDA's grids and blocks; NPUs use tensors to handle data flow and parallelism; and FPGAs utilize LUTs and DSP units to set parallelism and resource usage. $C^i_{rm}, G^i_{rm}, \ldots$ represent the estimated memory required for these tasks on these types of chips. $Disk^i_r$ refers to the estimated storage space needed for the computational task, and $Runtime_i$ denotes the estimated duration for the task to run on the current node, given the current execution method. This matrix provides a detailed description of the computational requirements of task T and its potential execution methods.

Assume the EEC system consists of a total of M clusters, each comprising N nodes. We use a decision variable x to denote the allocation of task T to node D. This variable is constrained to be an integer within the range $0 \leq x < k$, where k represents the number of heterogeneous computational units present within the cluster, with each value corresponding to a specific execution strategy on the node. A task allocation plan across all clusters can thus be encapsulated within the following matrix representation:

$$T^D = \left\{ \begin{array}{cccc} \text{Cluster}_1 & \text{Cluster}_2 & \ldots & \text{Cluster}_M \\ x^1_1 & x^2_1 & \ldots & x^M_1 \\ x^1_2 & x^2_2 & \ldots & x^M_2 \\ \vdots & \vdots & \ddots & \vdots \\ x^1_N & x^2_N & \ldots & x^M_N \end{array} \right\}. \quad (1)$$

If clusters are of different sizes, the matrix can be augmented by filling the positions of unavailable chip types with zeros. Using this representation, the assignment of task T to node D can be expressed as T^D_x. This indicates that the final output of our scheduling algorithm is the identification of the appropriate cluster and node for the current task, along with the designated execution strategy x.

For every task assignment to a node, the embeddings of both the task and the node are updated. Let E_i and T_i denote the i-th element of the node and task embeddings tuples respectively. The updated embeddings after the assignment can be represented as E'_i and T'_i. The update rules can then be defined as follows:

$$E'_i = E_i - \text{Net}_i \quad (2)$$

$$T'_i = T_i - \text{Runtime}_i \quad (3)$$

With the constraint that for each i, the updated task embedding must be less than or equal to the updated node embedding:

$$T'_i \leq E'_i \quad (4)$$

For the average waiting and completion times, we retain the summation across all tasks, but now with explicit indexing to denote the summation is over individual tasks:

$$\text{Avg}_{\text{wait}} = \frac{1}{c} \sum_{i=1}^{c} T_{\text{wait,i}} \quad (5)$$

$$\text{Avg}_{\text{complete}} = \frac{1}{c}\sum_{i=1}^{c}(T_{\text{Runtime,i}} + T_{\text{wait,i}}) \tag{6}$$

Here, c still denotes the count of tasks. The optimization objective becomes:

$$\min\left(\alpha \times \text{Avg}_{\text{complete}} + (1-\alpha) \times \text{Avg}_{\text{wait}}\right) \tag{7}$$

where α is a parameter between 0 and 1 that balances the completion time and waiting time in the objective function. Our study prioritizes both average waiting time and completion time as key metrics, recognizing their importance from the perspectives of both users and system managers. While users are primarily concerned with completion time, when the task is finished and results are available, waiting time plays a critical role from the system manager's viewpoint. Efficient management of waiting time is essential for optimizing task allocation and maintaining load balancing across heterogeneous devices. By carefully minimizing waiting times, the system can maximize resource utilization, ensuring smooth operation and distributing workloads evenly across the available hardware. Therefore, we emphasize both metrics to strike a balance between user experience and overall system performance and efficiency.

Recognizing that the optimization of task scheduling is classified as Mixed Integer Linear Programming (MILP) and is inherently a non-convex NP-hard problem, as identified in the literature [20], precise estimation of the actual completion time for each computational task becomes a critical prerequisite for effective scheduling. The challenges are amplified in heterogeneous computing environments, where the scheduling demands consideration of additional data dimensions to accommodate diverse resource types, thereby increasing the scheduler's complexity. While heuristic algorithms and reinforcement learning offer viable solutions, their application to such multi-dimensional and intricate problems requires strategic tailoring to navigate these complexities successfully.

4 Architecting the heterogeneous EEC scheduling system

As illustrated in Fig. 2, the proposed framework is designed specifically for task scheduling in EEC systems, aiming to optimize the management of heterogeneous computing resources in server clusters. We base our framework on Kubernetes to provide the central orchestration capabilities and integrate lightweight Kubernetes distributions, such as K3S and KubeEdge, to enable more efficient resource management at the network edge.

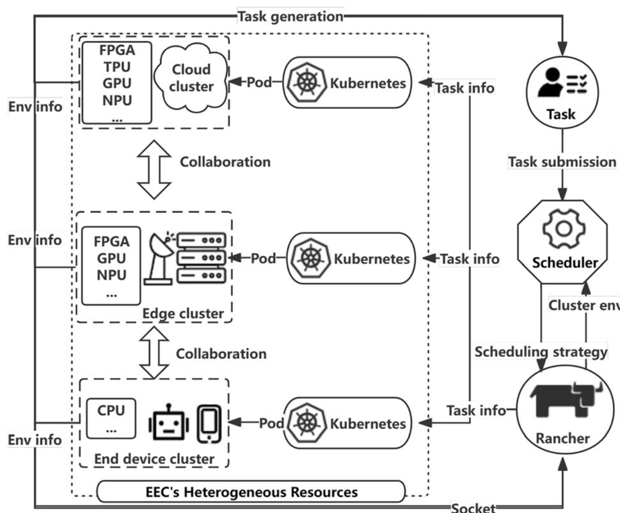


Fig. 2 Schematic of our scheduling framework for Heterogeneous EEC System Architecture. The first layer encapsulates heterogeneous computing resources, the second layer realizes cluster node management based on Kubernetes, and the third layer implements cross-cluster management and task distribution based on Rancher

In the top layer of our framework we utilize Rancher, a higher-level container management platform similar to Kubernetes Federation (KubeFed). While Kubernetes handles the deployment and management of containers within each cluster, Rancher provides the user interface, access control, and additional services that simplify cluster administration. This hierarchical approach allows for cohesive and coordinated management of multiple Kubernetes clusters across the end-edge-cloud continuum.

The native modules within Kubernetes and Rancher can only monitor basic metrics such as the utilization of CPU, memory, and disk. However, extra support of data acquisition is needed for the EEC environment to manage heterogeneous compute resources including GPUs, TPUs, NPUs, and FPGAs, which are pivotal for AI workloads. To address this gap, we integrated monitoring modules on each node to collect real-time data using low-level libraries like PyCUDA and GPUtil, as well as device-specific toolkits (e.g., `npu-smi` for the Huawei Ascend-310 NPU to monitor NPU utilization and map memory metrics N_k and N_m in E). In our framework the collection of these metrics are then aggregated to the scheduler using a Socket connection.

Finally, based on Docker we built environment images for each type of node (according to their processor/accelerator class) in the system. This allows the scheduler to pull and launch the corresponding environment images (such as GPU, NPU, etc.) directly via the Kubernetes API to execute the distributed tasks. The scheduler distributes tasks in JSON format through the Rancher API, guiding the deployment of pods on each node of every cluster for the

specified tasks. This implementation achieves high-performance and resource-aware scheduling within the EEC framework.

5 Constructing RL-based scheduling strategy

We employ a RL agent as the core of our scheduler. The agent interacts with the environment that encapsulates the server conditions and task details at each time point. The assignment of tasks, denoted as a_t, is considered an action following a policy $\pi(a_t|s_t)$ to learn. The learning process targets at the maximization of the total reward $R = \Sigma r$ in the form of environment feedbacks.

The proposed RL-based scheduler makes scheduling decisions subject to the resource requirements of all scheduled tasks and the capacity constraints of the available resources. The RL agent constantly observes the computational capabilities of nodes, inter-device communication costs, and the demands of incoming tasks.

As depicted in Fig. 3, the task queue, the clock (which also serves as a timer recording time steps), and the EEC's heterogeneous computing resources collectively constitute the environment for our task scheduler to observe. The task queue primarily provides information (Task info) that includes the computational resources required by different computational methods of the task and an approximate computation time. The clock primarily provides the current system time point t, recording the time each task arrives, is queued, assigned for execution, and finally completed. Lastly, the EEC's heterogeneous computing resources primarily provide information (Env info), including the basic information of each cluster, such as available computational resources, network environment, and the number of completed tasks.

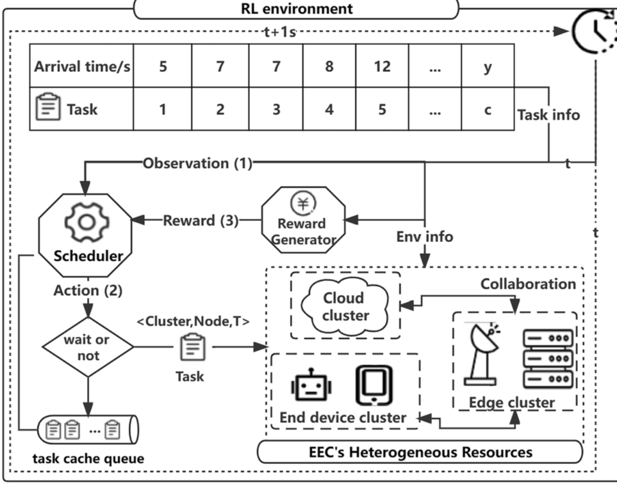


Fig. 3 Conceptual diagram of the reinforcement learning model proposed in this research

The scheduler takes two types of actions based on the current environment after observing it. One action directly outputs the cluster and node number where the current task is assigned, along with T execution method $< Cluster, Node, T >$. The other action allows the current task to wait for a while P, in addition to the $< Cluster, Node, T >$ information.

Upon the agent's action, the reward generator in the figure evaluates the computational reward resulting from the action at the current time. It is important to note that in the RL environment, the rewards given to the agent are always external. However, RL algorithms can have their internal reward (or parameter) calculations, which they continuously update to discover superior strategies.

5.1 Fundamental design

In the following we introduce the key elements in the design of our RL-based scheduler.

State space: The state space is one of the fundamental concepts in the RL model, typically representing a set of all possibilities that the agent can observe in the environment. In the system designed in this paper, it is mainly divided into three parts:

- **Env info:** This primarily includes $E = [C_k, C_m, G_k, G_m, N_k, N_m, \ldots, Disk, Net]$ mentioned in Sect. 3. The only difference is that the Env_{info} mentioned here is a set of E corresponding to all nodes in all clusters in the system, which can be represented as $Env_{info} = \{E_1, E_2, \ldots, E_l\}$, where l represents the number of all nodes.
- **t:** This represents the built-in time step of the system. It records the step distance the system has walked from the start of scheduling the first task to the present. It is worth noting that when there is no task to be scheduled in the system, or all computing resources are idle, this t value will be reset to zero until the next task arrives and the timing restarts.
- **Task info:** This mainly includes $T = < T_1, T_2, \ldots, T_k >$, where $T_i = [C^i_{rk}, C^i_{rm}, G^i_{rk}, G^i_{rm}, N^i_{rk}, N^i_{rm}, .., Disk^i_r, Runtime_i]$ mentioned in Sect. 3. The only difference is that the $Task_{info}$ mentioned here is the computing task being scheduled in the system; the tasks currently waiting for scheduling and their waiting time can be represented as $Task_{info} = \{T'_1, T'_2, \ldots, T'_x\}$, where x represents the number of tasks, $T'_i = T_i + P$, and P is the waiting time of the task.

Consequently, in the scheduling problem addressed in this paper, the state space expands solely with the increase in cluster size (total number of virtual machines) and the length of the task waiting queue rather than depending on the total number of jobs. This implies that the complexity of the optimization problem is directly proportional to the scale of heterogeneous computing resources in the system and fundamentally unrelated to the size and quantity of tasks.

Action Space: The scheduler's decision-making process, encapsulated as actions within the system, influences the state of the environment by determining task allocation. The action space consists of two principal decision types that the scheduling agent can opt for. The first type is an immediate scheduling action, where the scheduler outputs a direct strategy for task allocation and execution on a specified cluster and node. The second type involves a delayed scheduling action, where the scheduler decides to wait for P seconds before task execution. These decisions can be formally represented by the tuple $Strategy = <Cluster, Node, T, P>$.

To elaborate:

- The choice between these two scheduling actions is determined by the output of a reinforcement learning-based scheduling agent. Depending on the state of the system and the task requirements, the agent may choose to immediately allocate the task to a node (the first type) or delay the task by P seconds before scheduling (the second type).
- The decision-making process is an online, single-task scheduling method, reflecting real-time task assignments. The execution of this process aligns with Equation (1), which enumerates all potential scheduling scenarios for a single task within the system.
- The parameter P, representing the wait time before scheduling, is also determined by the scheduling agent based on the reinforcement learning model. The value of P is within a range from 0 to *max_time*, allowing for dynamic adjustment based on immediate system conditions and task priorities.

Given a system with l nodes and the inclusion of k possible execution methods outlined in Sect. 3, the action space is quantified as $l + k + 1$ discrete actions. Each action corresponds to a unique scheduling decision, either instantaneously placing a task on a node or imposing a calculated delay, enhancing the adaptability of the scheduling system.

Reward: The reward represents the quality of the feedback the environment gives for each action the scheduler takes with respect to the optimization objective. In this paper, the reward is designed with two primary cases. One is -1, which indicates that the current action has not changed the environment (i.e., the task is assigned to wait rather than execute immediately). The other is a positive number greater than zero, representing the degree to which the action optimizes the objective. This positive reward is proportional to the quality of the task assignment in terms of minimizing both task completion time and waiting time.

Since reinforcement learning accumulates rewards over time, each individual reward contributes to the overall long-term optimization. The reward for each action serves as a learning signal in the RL model, guiding future task allocations.

Overall, the reinforcement learning process is illustrated in Fig. 4. The paper concludes with an explanation of the reward calculation for each action. The reward for each action, with respect to a given task, is formalized as:

$$\text{Reward}(t) = \begin{cases} -1 & \text{if wait} \\ R(t) & \text{if distribute} \end{cases} \tag{8}$$

where the function $R(t)$ is defined as:

$$\begin{aligned} CTC &= \alpha\left(\frac{\sum T_{\text{Runtime}}}{c} + T_{\text{Runtime}}(t)\right), \\ WTC &= (1-\alpha)\left(\frac{\sum T_P}{c} + T_P(t)\right), \\ R(t) &= c + \frac{1}{CTC + WTC}. \end{aligned} \tag{9}$$

where $\sum T_{\text{Runtime}}$ and $T_{\text{Runtime}}(t)$ represent the total runtime of all tasks completed up to the current time and the estimated runtime for task t, respectively. Similarly, $\sum T_P$ and $T_P(t)$ denote the total waiting time for all completed tasks and the estimated waiting time for task t, respectively. The variable c denotes the count of tasks that have been completed.

In practical reinforcement learning scenarios, the objective function is typically maximized. However, in our scheduling context, the objective is to minimize both task completion and waiting times. Consequently, the reciprocals of these times are incorporated into the reward function to align with the minimization goal.

The success of RL-based scheduling heavily relies on the reward function. In our design, α controls the trade-off between completion time and waiting time, allowing the scheduler to adapt to different system needs. We acknowledge that mis-specification or suboptimal task allocation may occur in complex heterogeneous environments. However, by adjusting α, the system can balance short-term and long-term objectives, optimizing task distribution.

By design the `Runtime` attribute represents the estimated running time required for each task when executed on a specific node. A naïve way is to maintain a lookup table for any specific device, but this is very time-consuming and hard to extendable. For instance, in [23], the

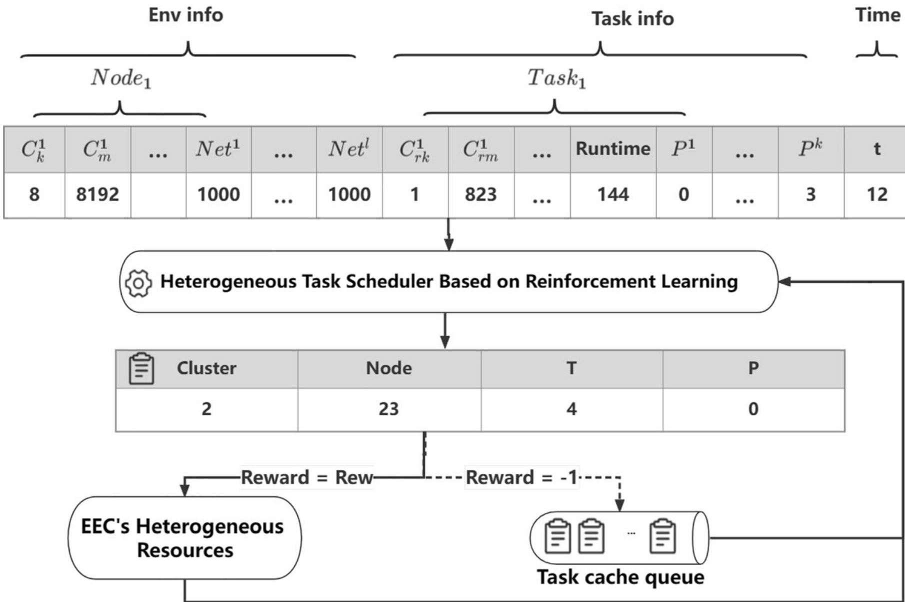


Fig. 4 The internal working of the proposed RL-based scheduler. With our reward design, the scheduler (agent) learns to minimize both the task completion time and the waiting time

authors measured the average inference latency of 5000 sample DNNs and constructed a latency lookup table for that specific device. However, Lu et al. [24] found that building a latency predictor for each operator's single device could take over 27 h, assuming an ideal case of 20 s per measurement and uninterrupted measurements. To this end, they further proposed an efficient proxy adaptation technique that significantly improved the monotonicity of latency, enabling the discovery of Pareto-optimal architectures almost identical to those of existing single-device NAS while only using a proxy device, thus avoiding the high cost of building predictors for each device. However, considering the current EEC scenario, with the rapid updates of devices and computing architectures on the market, the wide variety of heterogeneous computing chips, and the absence of a unified protocol to manage them [25], building such predictors still consumes a lot of time and resources.

For task `Runtime` estimation, we have observed a pattern as we continue to run different AI-intensive tasks on various heterogeneous computing resources. A specific multiplicative relationship exists in computing power between different versions of the same type of heterogeneous processing units (Fig. 5).

Figure 5 shows the time consumed by 100 CNN models automatically trained through a Neural architecture search (NAS) [26] model to infer 10,000 MNIST handwritten images (batch = 1) on different heterogeneous chips. The vertical axis in the figure, from top to bottom, represents chips including the Tesla T4 GPU (16 G), Nvidia Geforce GTX 1060 GPU (8 G), Davinci AI core NPU (8 G), ARM cortex-A72@1.5GHz with Broadcom VideoCore VI@500MHz CPU(4 G), Intel(R) Core(TM) i7-6700HQ CPU @2.60GHz(8 G), and A55 Arm core@1.6GHz(8 G)

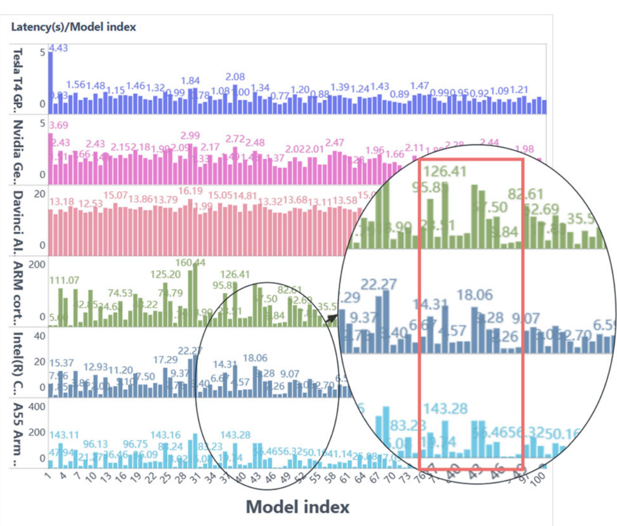


Fig. 5 Inference latency of the 100 CNN models run on different devices

CPU. The figure shows that a particular multiplicative relationship exists between the same type of chips, such as GPUs and CPUs, though different types of computing chips produce significant differences in computation time when running the same AI computation task. Therefore, in the calculation of the `Runtime` attribute of the tasks in this paper, we establish a computation latency lookup table based on one of the chip model and then calculate the power multiples between different versions of the same type of chip to expand the lookup table, based on a large amount of historical data from heterogeneous computing devices, instead of conducting experiments to establish a lookup table on each device. Empirical evidence shows

that this method significantly reduces the workload and provides relatively accurate predictions of the actual running time of tasks.

5.2 Strategy optimization

In the context of an EEC system, the scheduler needs to perceive the state of heterogeneous computing resources on each node in the cluster, take scheduling actions of task allocation, and learn from the experience of these iterative interactions given the environment feedbacks (rewards).

Q-learning, a form of dynamic programming, traditionally uses a two-dimensional matrix to represent the Q-function, which contains values for each state-action pair. However, this tabular approach becomes impractical in high-dimensional spaces due to the exponential growth of state-action pairs. To address this, Deep Q-Learning (DQN) is widely adopted where a neural network with parameters u is used to approximate the Q-values, denoted as $Q(s,a,u) \approx Q^{*}(s,a)$. We employ the DQN algorithm as an optimization method within the reinforcement learning framework. Empirical studies have shown that DQN can learn the availability of heterogeneous computing resources and the demand constraints of heterogeneous computing tasks pertinent to the scheduling problem discussed in this study [20]. Consequently, it can complete the corresponding reinforcement learning episodes for accumulated reward maximization.

In our implement of DQN, the neural network is trained to minimize the following loss at each step i:

$$L_i(u_i) = \mathbb{E}_{s,a,r,s'}(r + \gamma \max_{a'} Q(s',a',u_{i-1}) - Q(s,a,u_i))^2 \tag{10}$$

In this equation, r represents the distribution over transitions $\{s,a,r,s'\}$ sampled from the environment. The term $r + \gamma \max_{a'} Q(s',a',u_{i-1})$ is known as the Temporal Difference (TD) target, and the difference between the TD target and the Q-value is the TD error.

In the proposed reinforcement learning based scheduler, the agent employs an ϵ-greedy policy, which balances the trade-off between exploration and exploitation. The agent's memory, a vital component of the Experience Replay mechanism, stores past transitions and uses them in batches to update the Q-values. This batch update approach enhances the stability and efficiency of the learning process. The agent's Q-values are approximated using a neural network trained to minimize the Mean Squared Error (MSE) loss, as defined in the equation above. The agent uses a separate target network to ensure stable learning, providing more consistent update targets. The entire algorithm is shown in Algorithm 1.

Algorithm 1 Training Algorithm

```
Result: Agent.Qnet
for episode in num_episodes do
    Input: Agent, test as a new Schedule
     env with 'test' as name, x tasks, and
     max_time of y, time to 0;
    Initialize total_reward to 0;
    while test.time ≤ test.max_time do
        for task in test.tasks do
            if task.arrival_time ==
             test.time then
                Agent chooses an action for
                 the task and gets the
                 server and state.;
                Get the next state and
                 reward by stepping the
                 task, server in the test.;
                if reward ≠ -1 then
                    Remember the state,
                     task.task_id, server,
                     reward, next_state, and
                     test.dones;
                    total_reward ←
                     total_reward + reward;
                    Replay (Agent);
                    if Agent.epsilon >
                     Agent.epsilon_min
                     then
                        Agent.epsilon ←
                         Agent.epsilon ×
                         Agent.epsilon_decay;
                    end
                end
            end
            else if task.arrival_time <
             test.time then
                if task.schedule_to is None
                 then
                    Continue;
                end
                Run the test on the server
                 and task;
            end
        end
        if test.tasks is empty then
            Break;
        end
        test.time ← test.time + 1;
    end
end
```

6 Evaluation

In this section, we detail the setup of our experimental environment, which is a heterogeneous EEC computing testbed consisting of clusters at the end, edge, and cloud layers. We then evaluate the performance of the proposed scheduler by comparing it with several baseline scheduling algorithms.

6.1 Experimental setup

Cluster resources. In this study, an real-world testbed was built as a prototype EEC system following our framework. The hardware configurations for the cloud, edge, and end devices are as follows:

- **Cloud server cluster**: One server equipped with 8 Intel (R) Xeon (R) E5-2620 v4@2.10GHz CPU (64 G) and 2 Nvidia Tesla T4 GPU (16 G); 1 TB disk capacity.
- **Edge server cluster**: Three Atlas 200 DK equipped with 2 A55 Arm core@1.6GHz CPU (8 G) and 2 Davinci AI core NPU (8 G); 50GB disk capacity.
- **End device cluster**: Four Raspberry Pi equipped with 4 ARM cortex-A72@1.5GHz CPU (8 G) and Broadcom VideoCore VI@500MHz (4 G); 59GB disk capacity.

In this context, K3S is responsible for managing the cloud server and end-side device clusters, while KubeEdge handles the management of the edge server cluster. Ultimately, all three clusters are centrally managed by the Rancher tool. It is worth mentioning that, for both the cloud server and Atlas 200 DK devices, we have created corresponding Docker images containing the necessary drivers and library functions for their AI chips (GPU, NPU). This allows us to execute tasks based on scheduling and execution policies tailored to different AI chips.

Task information. Next, we focus on task selection and the generation of task queues. As mentioned in Sect. 3, to better simulate the industrial EEC environment, we selected several common computing tasks from practical applications: AI-intensive tasks (inference of 10,000 MNIST images using 10 CNN models with a batch size of 1), memory-intensive tasks (continuously adding data to a list in the program to increase runtime memory), and storage-intensive tasks (repeatedly copying a program that consumes a large amount of storage memory). The task information matrix T for each type of task is shown in Table 1.

In generating task queues to better simulate task distribution in a real EEC scenario, this study introduces two approaches. The first approach is the standard task queue generation mode, where 100 random tasks arrive at arbitrary times within the initial 1000s, termed the "Stable-Arrival Task Queue". For example, in a power grid scenario, an end-to-end camera captures photos during routine inspections. The second approach involves the arrival of 1000 random tasks at random times within a 1000-second interval, termed the "Burst-Arrival Task Queue", similar to computing tasks centrally updated by AI models in a power grid scenario.

Reinforcement learning parameters. Finally, the basic settings of the reinforcement learning parameters utilized in the experiments are displayed in Table 2. It is worth noting that during the training process of reinforcement learning, we adopted an early stopping strategy to prevent model overfitting and save a significant amount of training time. Specifically, the training is halted if the maximum reward does not increase for 80 consecutive iterations.

Table 1 Resource Consumption Metrics for AI-intensive, Memory-intensive, and Storage-intensive Tasks Across Devices and Computational Modes

Task Type	Device	Mode	C_{rk}	C_{rm}/MB	G_{rk}	G_{rm}/MB	N_{rk}	N_{rm}/MB	Disk/MB	Runtime/s
AI-intensive	Cloud	CPU	0.49	1770	0	0	0	0	32	74
		GPU	0.10	1017	0.30	2048	0.00	0	12	14
	Edge	CPU	0.98	900	0.00	0	0.00	0	32	534
		NPU	0.10	1024	0	0	0.30	1024	10	114
	End	CPU	0.98	900	0.00	0	0.00	0	32	534
Memory-intensive	Cloud	CPU	0.11	1942	0.00	0	0.00	0	23	12
	Edge	CPU	0.17	1955	0.00	0	0.00	0	20	37
	End	CPU	0.41	1966	0.00	0	0.00	0	20	26
Storage-intensive	Cloud	CPU	0.11	143	0.00	0	0.00	0	1987	12
	Edge	CPU	0.14	40	0.00	0	0.00	0	1964	134
	End	CPU	0.34	87	0.00	0	0.00	0	1969	47

Table 2 Reinforcement Learning Parameters

Parameter	Value
Gamma (γ)	0.91
Initial Epsilon ($\epsilon_{\text{initial}}$)	1.0
Minimum Epsilon ($\epsilon_{\min}$)	0.01
Epsilon decay	0.995
Batch size	64
Learning rate (lr)	0.001

Baseline scheduling algorithms. In this study, we conducted performance comparison experiments using four scheduling algorithms and the reinforcement learning method proposed in this paper, which is hereafter referred to as EECRL.

DRL- Based algorithm [20]: This reinforcement learning method was proposed by Muhammed Tawfiqul Islam et al. in 2022 for the Spark cloud computing environment. It can reduce task completion time and decrease server wear by nearly 30%. However, since its algorithm and reward function design mainly target server wear and homogeneous computing resources and tasks, we set the reward function consistent with our study during the replication process. Meanwhile, [20] uses Q-learning and REINFORCE as agents. Since we adopted Q-learning in our study, we replaced it with the PPO method. Finally, other reinforcement learning parameters are consistent with the settings in [20].

S-P-GWO [27]: A hybrid optimization algorithm that combines a support vector machine (SVM) to classify virtual machines with a particle swarm optimization (PSO) algorithm to find the optimal virtual machine. Finally, the Grey wolf optimizer (GWO) is applied to determine the scheduling strategy with the shortest execution time.

GA-GWO [28]: A hybrid optimization algorithm that integrates the GWO and Genetic Algorithm (GA), enhancing the optimization capability of GWO to improve global search efficiency in solving scheduling problems.

Sigmoid-PSO [29]: This is a particle swarm algorithm that uses a sigmoid function for inertia weight optimization.

First Fit (FF): This is a classic scheduling strategy that executes tasks on a first-come, first-served basis. It is a greedy algorithm that always considers the maximum reward that can be obtained during task arrival. During the initial phase of training in the RL-based scheduler, the FF strategy serve as a baseline policy to explore the action space efficiently.

It is worth noting that when conducting experiments with the Burst-Arrival Task Queue, the scheduling process of the aforementioned heuristic scheduling models is significantly delayed due to the requirement for a global solution. This substantially impacts their overall performance. Therefore, in this part of the experiment, we adjusted the scheduling interval of the heuristic learning models, setting it to perform scheduling once every 100 tasks.

Since the built-in scheduling algorithm of Kubernetes only supports the scheduling of computing resources and tasks within the cluster, it cannot be directly compared in the EEC scenario. Ultimately, we chose the three most commonly used algorithms in the task retrieval scenario for comparison, namely reinforcement learning, heuristic learning, and the greedy algorithm, to analyze the performance advantages of the method proposed in this paper as comprehensively as possible.

6.2 Parameter Study and Convergence Evaluation of EECRL

We train the EECRL model for both the Stable-Arrival Task Queue and the Burst-Arrival Task Queue and set α to 1.0, 0.75, 0.5, 0.25, and 0.0, respectively, to observe the impact of α on the training process, as shown in Fig. 6.

The value of α affects whether the EECRL model is more inclined to optimize the average completion time of tasks or the average waiting time of tasks. Therefore, several obvious phenomena can be found in Fig. 6:

1. The optimization effect of EECRL on the Burst-Arrival Task Queue is much higher than that on the Stable-Arrival Task Queue.
2. EECRL is better and faster at optimizing the waiting time of tasks in the queue than optimizing the completion time of tasks in the queue.
3. Compared with the FF algorithm, EECRL has a significant performance improvement.

Firstly, regarding the first phenomenon, it is evident from Fig. 6 that the convergence curves of the first row are significantly inferior to those of the second row. Mainly, when performing multi-objective optimization on the Stable-Arrival Task Queue, overfitting is prone to occur. However, for the Burst-Arrival Task Queue, when α is 0.25, 0.5, 0.75, and 1.0, the performance is improved by 10.77%, 14.44%, 15.45%, and 13.63%, respectively, compared to the FF algorithm. The reason is relatively straightforward. The task release of the Stable-Arrival Task Queue is more dispersed on the timeline. Under such circumstances, apart from the randomly assigned task scheduling strategy, most tasks will experience minimal waiting time and will be directly assigned to the most suitable node at the current time point. Therefore, no matter how many times reinforcement learning is

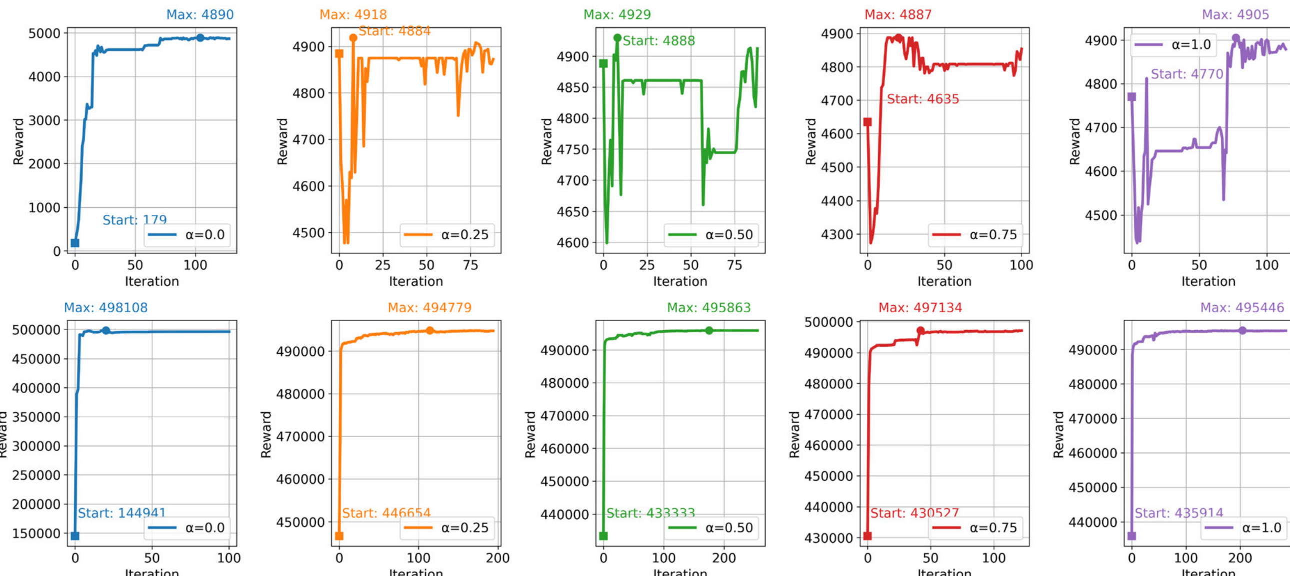


Fig. 6 Evaluation of the Training Process and Convergence Performance of EECRL for both Stable-Arrival Task Queue (first row) and Burst-Arrival Task Queue (second row)

performed, it cannot learn better knowledge. Moreover, the training data and sample space in this situation are limited, which often leads to overfitting.

Secondly, the second phenomenon can be seen from the data in Fig. 6. When EECRL focuses only on optimizing the task waiting time ($\alpha = 0.0$), it improves by 2632.96% and 243.64% for the Stable-Arrival Task Queue and Burst-Arrival Task Queue, respectively. Compared to other situations, the performance improvement of EECRL over the FF algorithm is at least 20 times higher. The reasons for this phenomenon are analyzed as follows: (1) The task waiting time is a single indicator; at least only one time period needs to be recorded in the calculation process, so it is a single-objective optimization problem in the optimization process. (2) The task completion time is influenced by many factors, such as task execution time, task waiting time, model inference time, network latency, etc. Therefore, even if we only consider the task completion time, we consider data from multiple dimensions. Therefore, compared to optimizing task waiting time, it is more challenging to take task completion time as the optimization target, which leads to better optimization performance in that dimension.

Regarding the third phenomenon, it is a feature of reinforcement learning. Reinforcement learning will improve performance through continuous attempts based on a fixed algorithm. Therefore, the reinforcement learning model built based on the FF algorithm is expected to improve specific performance after training.

6.3 Evaluating average task completion time

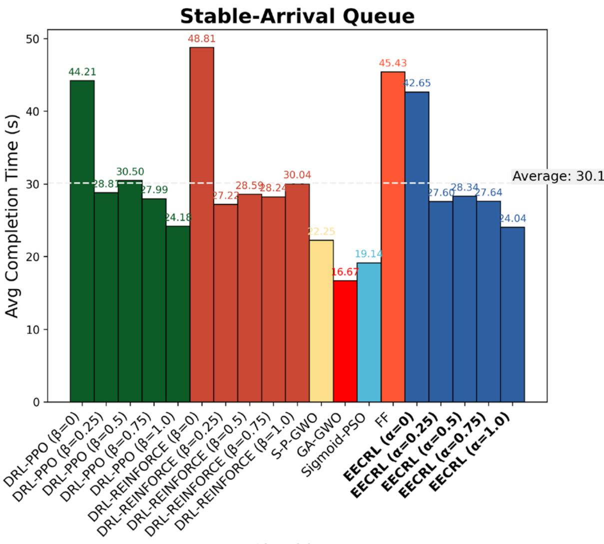
In this section, we mainly evaluate the optimization ability of EECRL on the average completion time of the task queue. By analyzing the comparative experiments of the EECRL algorithm and the four benchmark scheduling algorithms proposed in this chapter, we draw the corresponding conclusions.

In the comparative experiment design of this chapter, experiments were also conducted for the Burst-Arrival Task Queue and Stable-Arrival Task Queue environments, as shown in Fig. 7. From the figures, we can also find several phenomena:

1. For the Stable-Arrival Task Queue, the heuristic learning training method is better than the reinforcement learning effect. However, when it comes to the Burst-Arrival Task Queue, reinforcement learning shows excellent performance.
2. In the EEC reinforcement learning environment designed in this paper, the differences brought about by different reinforcement learning iteration algorithms are not particularly obvious.

Regarding the first phenomenon, as shown in Fig. 7, the GA-GWO heuristic learning algorithm demonstrates the best performance when handling the Stable-Arrival Task Queue. The average task completion time is only 16.67 s, which is 13.45 s better than the average value of all algorithms. This includes both the S-P-GWO and Sigmoid-PSO algorithms, which also outperform the reinforcement

Comparison of Algorithm Performance on Task Queues



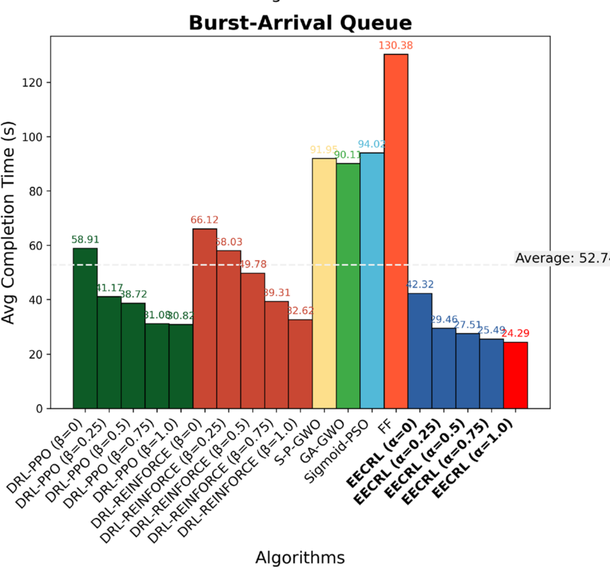


Fig. 7 EECRL Optimization Performance Evaluation for Average Completion Time of Task Queue for both Stable-Arrival Task Queue and Burst-Arrival Task Queue

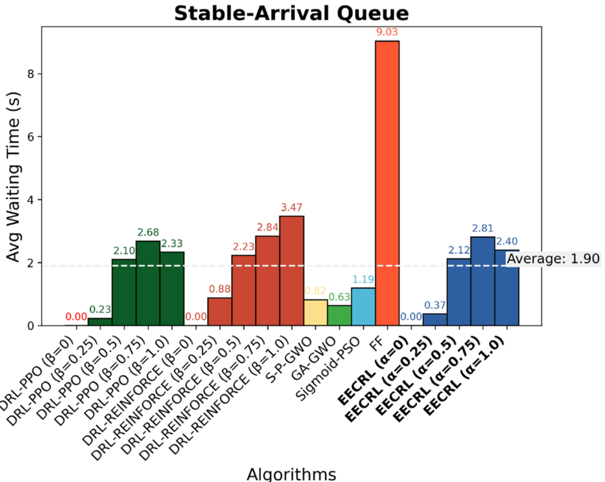
learning algorithms. The primary reason for this lies in the dispersion of task distribution. Although not explicitly shown in the paper, it is worth mentioning that heuristic learning algorithms, particularly GA-GWO, exhibit their greatest advantage when the number of tasks ranges between 20 and 40. In such an environment, tasks do not require long waiting times, and the search space for the algorithm is limited and well-defined. As a result, the optimization upper bound is fixed and easy to identify.

At the same time, with only 100 tasks, the knowledge learned by reinforcement learning is insufficient, increasing the likelihood of overfitting. In contrast, heuristic learning is more likely to find the optimal solution in this scenario. However, when dealing with the Burst-Arrival Task Queue, the presence of 1000 tasks provides a much larger search space, enabling reinforcement learning algorithms to perform better. EECRL achieves the shortest average task completion time of 24.29 s when $\alpha = 1.0$, ranking the best among all algorithms. In this more complex task queue scenario, the heuristic learning algorithms fall short, with the lowest task completion time of 90.13 s, which is 271.06% higher than the optimization achieved by reinforcement learning. Additionally, the portability and online scheduling capabilities of heuristic learning are far inferior to those of models trained via reinforcement learning, which is largely attributable to the significantly longer runtime of heuristic scheduling algorithms.

Regarding the second phenomenon, it can also be seen from Fig. 7 that the EECRL using Q-Learning and the DRL-based algorithm using PPO and REINFORCE strategies do not have a significant gap in optimization results; the biggest is only on the Burst-Arrival Task Queue, with a difference of 8.33s. However, such a gap is also a comparison of the best one chosen from many training model results, which is also one of the reasons why this paper chose Q-Learning as the optimization algorithm. In the results of many training models at regular times, the results shown by the models trained by these three algorithms are similar. Generally speaking, the choice among these three algorithms depends on the characteristics of the problem at hand. Q-Learning might be the preferred choice for problems with discrete state and action spaces. For problems with continuous action spaces, PPO could be more suitable. Lastly, in situations with a large amount of sample data and high variance, the REINFORCE algorithm might be the most appropriate choice. However, these choices are only suggestions based on the rules and extensive data analysis learned by AI researchers and can only be used as a reference. There is no mathematical formula proof for this, so in actual application, it often needs to be tried in practice to determine which algorithm is more suitable. It cannot be said that in the case of a discrete state and action space, Q-learning can show better performance.

6.4 Evaluating Average Task Waiting Time

Similarly, this section evaluates the optimization ability of EECRL on the average task waiting time of the task queue. By analyzing the comparative experiments of the EECRL algorithm and the four benchmark scheduling algorithms proposed in this chapter, as shown in Fig. 8, the corresponding conclusions are drawn.



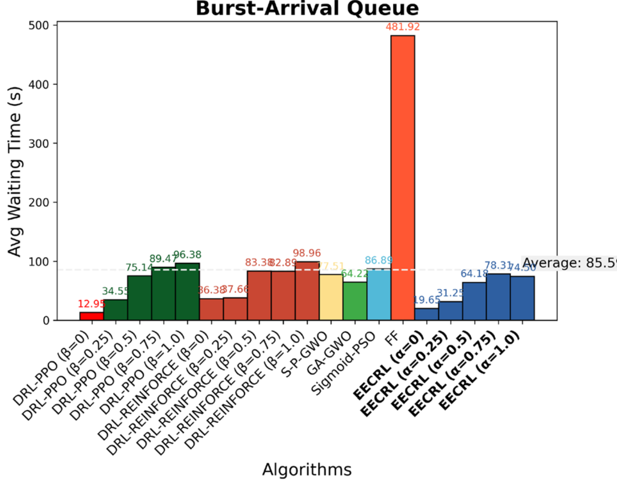
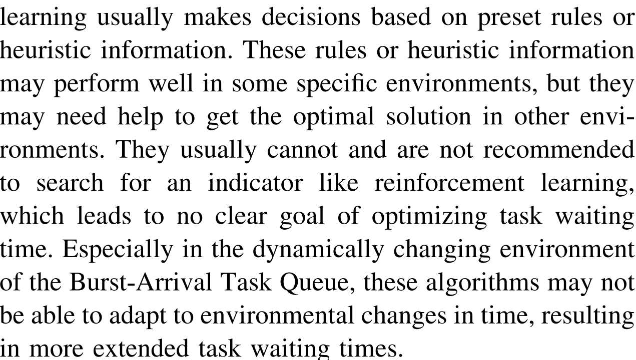


Fig. 8 EECRL Optimization Performance Evaluation for Average wait Time of Task Queue for both Stable-Arrival Task Queue and Burst-Arrival Task Queue

1. Whether in the Stable-Arrival Task Queue or the Burst-Arrival Task Queue, when only considering the task waiting time dimension, the reinforcement learning algorithm that learns only for task waiting time performs better than heuristic learning and greedy algorithms.
2. The average waiting time of the FF greedy algorithm is particularly high.

Regarding the first phenomenon, the main reason is that the reinforcement learning algorithm learns the optimal strategy through interaction with the environment and can adapt to environmental changes. Whether it is a Stable-Arrival Task Queue or a Burst-Arrival Task Queue, the reinforcement learning algorithm can adapt to the characteristics of the task queue by only setting the optimization goal to reduce the average waiting time of tasks, thereby optimizing the task waiting time. Secondly, heuristic



learning usually makes decisions based on preset rules or heuristic information. These rules or heuristic information may perform well in some specific environments, but they may need help to get the optimal solution in other environments. They usually cannot and are not recommended to search for an indicator like reinforcement learning, which leads to no clear goal of optimizing task waiting time. Especially in the dynamically changing environment of the Burst-Arrival Task Queue, these algorithms may not be able to adapt to environmental changes in time, resulting in more extended task waiting times.

Regarding the second phenomenon, the FF greedy algorithm only considers the optimal value at the current time point and does not consider the global gain. Therefore, it only cares about how the current task can be completed the fastest and usually waits for several seconds for the best server, even though there are other slightly worse nodes idle. At the same time, it will not consider the subsequent task selection to go to a slightly worse server. In this way, as more and more subsequent tasks are, the waiting time will become longer and longer. The later complex computing tasks that need excellent servers have to be assigned to inferior nodes to run due to the reason of first-come-first-served because the waiting time is too long, making the load of the entire cluster more unbalanced. This is also why the FF algorithm leads to the most average task completion time.

6.5 Evaluating energy consumption

Finally, to comprehensively evaluate the EECRL model, this section selects three baseline methods for comparison: FF, S-P-GWO (chosen because it executes faster, while the performance difference between the two GWO algorithms is negligible), and Sigmoid-PSO. The experiment focuses on a proportionally generated task sequence, consisting of 1000 tasks over a 1000-second period, with tasks being generated at equal probabilities. The task distribution resulting from the scheduling strategies of the four models (EECRL [$\alpha = 0.75$], FF, GWO, and Sigmoid-PSO) across different devices was recorded and analyzed, as shown in Fig. 9. Additionally, the total execution time (in seconds) of the RL, FF, GWO, and Sigmoid-PSO models across different devices was recorded and analyzed, as presented in Fig. 10.

The power consumption for the experimental devices is as follows: the Cloud server operates at 425 W, the Atlas devices at 25 W, and the Raspberry Pi devices at 5 W.

The final energy consumption of the scheduling algorithms for the four models is computed as follows:

- RL model**:**

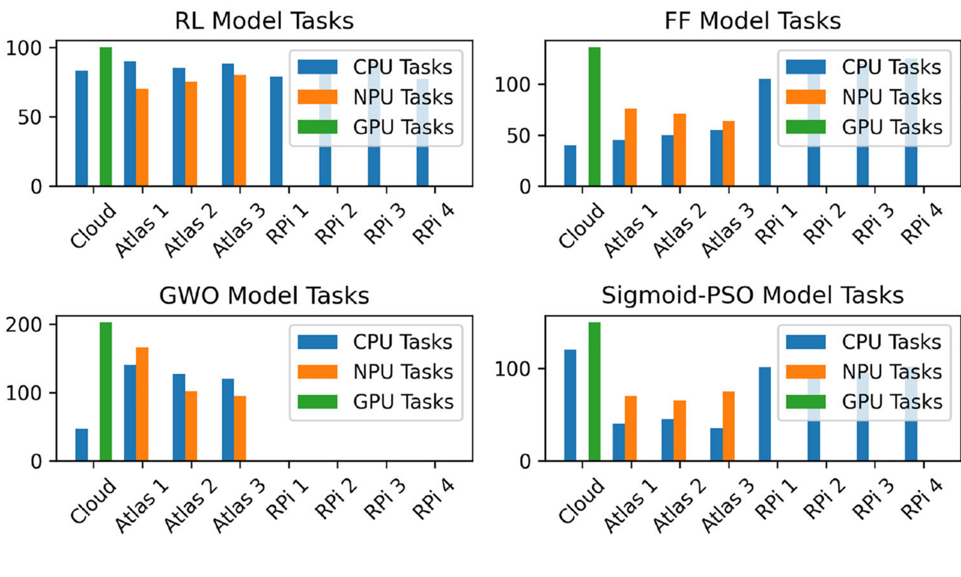


Fig. 9 Task distribution of RL, FF, GWO, and Sigmoid-PSO models across different devices

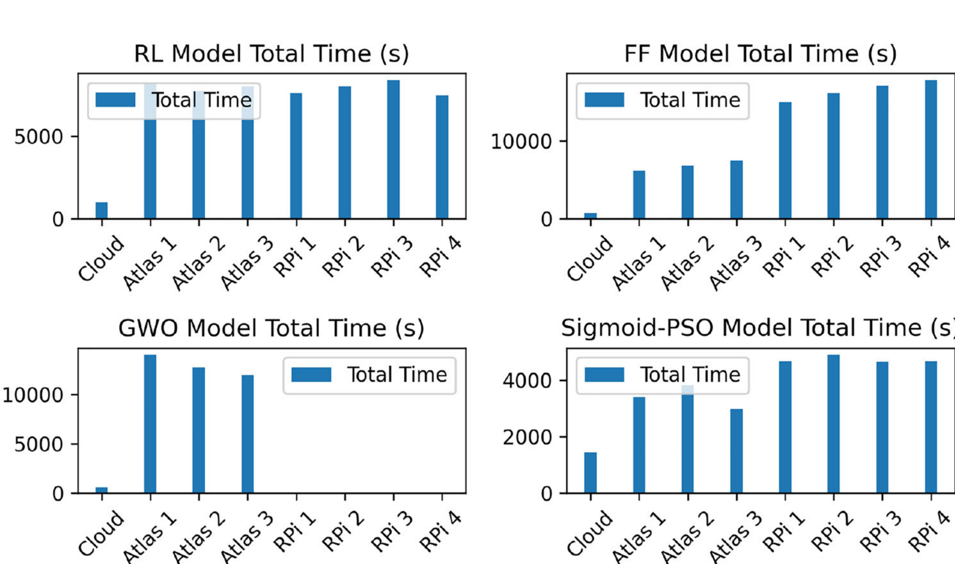


Fig. 10 Total execution time (in seconds) of RL, FF, GWO, and Sigmoid-PSO models across different devices

- Cloud Server: Energy = (996 + 1400) × 425 = 2396 × 425 = 1,018,300 J
- Atlas Devices: Energy = (23,933 + 25,650) × 25 = 49,583 × 25 = 1,239,575 J
- Raspberry Pi Devices: Energy = 31,441 × 5 = 157,205 J
- Total Energy: 1,018,300 + 1,239,575 + 157,205 = 2,415,080 J

- FF model:
  - Cloud Server: Energy = (720 + 2856) × 425 = 3576 × 425 = 1,519,800 J
  - Atlas Devices: Energy = (20,475 + 36,201) × 25 = 56,676 × 25 = 1,416,900 J
  - Raspberry Pi Devices: Energy = 65,977.5 × 5 = 329,887.5 J
  - Total Energy: 1,519,800 + 1,416,900 + 329,887.5 = 3,266,587.5 J
- GWO model:
  - Cloud Server: Energy = (564 + 2842) × 425 = 3406 × 425 = 1,447,550 J
  - Atlas Devices: Energy = (38,700 + 43,560) × 25 = 82,260 × 25 = 2,056,500 J
  - Raspberry Pi Devices: N/A (No tasks were scheduled)
  - Total Energy: 1,447,550 + 2,056,500 = 3,504,050 J
- Sigmoid-PSO Model:
  - Cloud Server: Energy = (1440 + 2100) × 425 = 3540 × 425 = 1,504,500 J
  - Atlas Devices: Energy = (10,200 + 23,940) × 25 = 34,140 × 25 = 853,500 J

– Raspberry Pi Devices: Energy = 18,863 × 5 = 94,315 J
– Total Energy: 1,504,500 + 853,500 + 94,315 = 2,452,315 J

As shown, the energy consumption for each model is calculated by multiplying the total execution time on each device by its corresponding power rating. The total energy consumption for the RL, FF, GWO, and Sigmoid-PSO models is the sum of the energy used by the Cloud, Atlas, and Raspberry Pi devices. From the results, several key observations can be made:

1. The FF and GWO models exhibit the highest energy consumption, while the Sigmoid-PSO model consumes relatively less. The RL model strikes a balance between performance and energy efficiency across all devices, making it a viable option for scenarios requiring both optimization and energy savings.
2. The RL scheduling strategy distributes tasks evenly across all devices, whereas algorithms like GWO tend to favor high-performance computing devices, concentrating tasks on them.

Firstly, it is quite unexpected that the RL algorithm achieves the lowest energy consumption. RL is designed primarily to optimize performance, so it would not be expected to have an advantage in terms of energy efficiency. However, the experimental results show that both RL and Sigmoid-PSO algorithms performed similarly, with RL reaching the lowest energy consumption, even lower by 37, 235 J. The primary reason for this phenomenon lies in the tendency of task allocation. As seen from Fig. 10, the GWO model tends to concentrate tasks on high-performance Atlas devices, while almost no tasks are assigned to the more energy-efficient Raspberry Pi devices. This heavy load on Atlas devices during task execution results in higher energy consumption. The algorithm prioritizes completing tasks as quickly as possible in terms of processing time, which leads to substantial time being wasted on high-performance devices waiting for tasks to complete, thus increasing both the total execution time and energy consumption.

At the same time, the FF model's "first-come, first-served" strategy leads to substantial time wasted in continuous waiting, as shown in Fig. 8, resulting in the longest total execution time and the highest energy consumption. Another factor is the degree of task distribution. The RL and Sigmoid-PSO models have relatively shorter total execution times across all devices, and their more balanced task distribution helps avoid the accumulation of energy consumption in certain devices, leading to lower overall energy usage. The RL model, in particular, achieves a balance between energy efficiency and performance through an evenly distributed task allocation. This advantage arises from the inherent strength of reinforcement learning in handling long-term dependencies and optimizing task allocation over extended periods, ensuring superior energy efficiency and task performance.

It is important to acknowledge the limitations of the experimental setup. The cloud service is supported by only a single server, and the heterogeneous devices are limited to seven. Therefore, the findings should be seen as providing guidance rather than definitive conclusions. As the cloud infrastructure scales and the variety of computational tasks increases, energy consumption outcomes may differ.

For instance, with more cloud servers and Atlas devices available or less densely packed tasks, the RL scheduling strategy could lead to higher energy consumption, similar to the GWO algorithm, which concentrates tasks on high-performance servers, causing inefficiencies. Nonetheless, the experimental results show that EECRL effectively balances energy consumption in resource-constrained environments by factoring in task queueing times, enabling efficient management of limited resources while optimizing both performance and energy efficiency.

Although the current experiments involve only one type of AI-intensive task, EECRL's methodology and architecture leverage the multiplicative relationship (as shown in Fig. 5) to quickly generate lookup tables for various tasks. By adjusting the reward function, EECRL can scale to larger systems and accommodate diverse task types, demonstrating potential for future research in more complex and heterogeneous environments.

Finally, the experimental results indicate that EECRL's energy consumption performance was not outstanding. However, there are clear directions for improvement. A key enhancement would be to incorporate the power consumption data of all devices in the EEC scenario as a variable in the optimization objective. This would allow control parameters to be adjusted according to user requirements, leading to a more dynamic balance between energy consumption and task execution efficiency. Additionally, factoring the power ratings of devices into the RL model's reward function could help optimize the trade-off between energy usage and performance, offering a promising direction for future research.

However, despite these potential trade-offs, the current results demonstrate the feasibility and potential research directions for applying reinforcement learning in EEC scenarios. The findings provide a strong foundation for further exploration into the balance between performance and energy efficiency in such complex environments. To address these challenges, future work could focus on developing more dynamic scheduling strategies that better account for task complexity, resource demand, and energy

efficiency, ensuring a more balanced and fair distribution of tasks across heterogeneous devices.

7 Conclusion and future work

The emergence of heterogeneous computing architectures and the burst of AI-oriented tasks pose a great challenge to the coordination in End-Edge-Cloud systems. At the same time, how to extend traditional cloud computing frameworks to smoothly fit the EEC environments requires much practical thinking. In this paper, we present an EEC system framework to fully utilize the power of heterogeneous computing across the cloud and the network edge. We first formulate the task scheduling problem over heterogeneous computing resources and correspondingly design a scheduling framework based on extended K8s and Rancher for EEC environment. Then we propose a reinforcement learning-based algorithm to solve the optimization problem and employ it as the core scheduler. In a real-world testbed we experimentally demonstrate that our scheduler effectively learns to maximize the rewards under various task settings and, as a result, effectively shortens the completion time of tasks especially in the case of the Burst-Arrival Task Queue.

Our method can be extended to multiple optimization objectives, such as server cost, energy consumption, etc. How to balance their priorities is a problem that we need to consider in our future work. In future work, we will expand the scale of experiments by incorporating a broader range of devices and task types. This will allow us to study how different hardware and task complexities affect scheduling efficiency. Additionally, we will explore the design of mathematical models and fine-tuning of hyperparameters to comprehensively optimize the EEC scheduling system across various dimensions, including task distribution, energy consumption, and system performance.

Author contributions All authors contribute to paper through either code, experiments or writing.

Funding This work is supported by National Natural Science Foundation of China (62072187), Guangdong Major Project of Basic and Applied Basic Research (2019B030302002), Guangzhou Development Zone Science and Technology Project (2023GH02) and the Major Key Project of PCL, China under Grant PCL2023A09.

Data availability Data available on request from the authors.

Declarations

Conflict of interest The authors have no relevant financial or non-financial interests to disclose.

References

1. Duan, S., Wang, D., Ren, J., Lyu, F., Zhang, Y., Wu, H., Shen, X.: Distributed artificial intelligence empowered by end-edge-cloud computing: a survey. IEEE Commun. Surv. Tutor. **25**(1), 591–624 (2023). https://doi.org/10.1109/COMST.2022.3218527
2. Jiang, M., Wu, T., Wang, Z., Gong, Y., Zhang, L., Liu, R.P.: A multi-intersection vehicular cooperative control based on end-edge-cloud computing. IEEE Trans. Vehicular Technol. **71**(3), 2459–2471 (2022). https://doi.org/10.1109/TVT.2022.3143828
3. Ren, J., Jiang, H., Shen, X., et al.: Editorial of ccf transactions on networking: special issue on intelligence-enabled end-edge-cloud orchestrated computing. CCF Trans. Netw. **3**, 155–157 (2020). https://doi.org/10.1007/s42045-020-00048-5
4. Ren, J., Zhang, D., He, S., Zhang, Y., Li, T.: A survey on end-edge-cloud orchestrated network computing paradigms: Transparent computing, mobile edge computing, fog computing, and cloudlet. ACM Comput. Surv. **52**, 1–36 (2019).
5. Zhou, C., Wu, W., He, H., Yang, P., Lyu, F., Cheng, N., Shen, X.: Deep reinforcement learning for delay-oriented iot task scheduling in sagin. IEEE Trans. Wirel. Commun. **20**(2), 911–925 (2021). https://doi.org/10.1109/TWC.2020.3029143
6. Chen, X., Cheng, L., Liu, C., Liu, Q., Liu, J., Mao, Y., Murphy, J.: A woa-based optimization approach for task scheduling in cloud computing systems. IEEE Syst. J. **14**(3), 3117–3128 (2020). https://doi.org/10.1109/JSYST.2019.2960088
7. Houssein, E.H., Gad, A.G., Wazery, Y.M., Suganthan, P.N.: Task scheduling in cloud computing based on meta-heuristics: review taxonomy open challenges and future trends. Swarm Evol. Comput. **62**, 100841 (2021)
8. Yuan, H., Bi, J., Zhou, M.: Multiqueue scheduling of heterogeneous tasks with bounded response time in hybrid green iaas clouds. IEEE Trans. Ind. Inform. **15**(10), 5404–5412 (2019). https://doi.org/10.1109/TII.2019.2901518
9. Zhou, J., Sun, J., Cong, P., Liu, Z., Zhou, X., Wei, T., Hu, S.: Security-critical energy-aware task scheduling for heterogeneous real-time mpsocs in iot. IEEE Trans. Serv. Comput. **13**(4), 745–758 (2020). https://doi.org/10.1109/TSC.2019.2963301
10. Hosseinzadeh, M., Azhir, E., Lansky, J., Mildeova, S., Ahmed, O.H., Malik, M.H., Khan, F.: Task scheduling mechanisms for fog computing: a systematic survey. IEEE Access (2023). https://doi.org/10.1109/ACCESS.2023.3277826
11. Carrión, C.: Kubernetes scheduling: taxonomy, ongoing issues and challenges. ACM Comput. Surv. (2022). https://doi.org/10.1145/3539606
12. Hardikar, S., Ahirwar, P., Rajan, S.: Containerization: Cloud computing based inspiration technology for adoption through docker and kubernetes. In: 2021 Second International Conference on Electronics and Sustainable Communication Systems (ICESC), pp. 1996–2003 (2021). https://doi.org/10.1109/ICESC51422.2021.9532917
13. Narayanan, D., Santhanam, K., Kazhamiaka, F., Phanishayee, A., Zaharia, M.: Heterogeneity-aware cluster scheduling policies for deep learning workloads. In: 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), pp. 481–498. USENIX Association, (2020). https://www.usenix.org/conference/osdi20/presentation/narayanan-deepak
14. Weng, Q., Xiao, W., Yu, Y., Wang, W., Wang, C., He, J., Li, Y., Zhang, L., Lin, W., Ding, Y.: MLaaS in the wild: Workload analysis and scheduling in Large-Scale heterogeneous GPU clusters. In: 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22), pp. 945–960. USENIX Association, Renton, WA (2022). https://www.usenix.org/conference/nsdi22/presentation/weng
15. Feng, J., Zhang, W., Pei, Q., Wu, J., Lin, X.: Heterogeneous computation and resource allocation for wireless powered

federated edge learning systems. IEEE Trans. Commun. **70**(5), 3220–3233 (2022). https://doi.org/10.1109/TCOMM.2022.3163439
16. Zhong, Z., Buyya, R.: A cost-efficient container orchestration strategy in kubernetes-based cloud computing infrastructures with heterogeneous resources. ACM Trans. Internet Technol. (2020). https://doi.org/10.1145/3378447
17. Kalia, K., Dixit, S., Kumar, K., Gera, R., Epifantsev, K., John, V., Taskaeva, N.: Improving mapreduce heterogeneous performance using knn fair share scheduling. Robot. Auton. Syst. **157**, 104228 (2022)
18. Abdulazeez, D.H., Askar, S.K.: Offloading mechanisms based on reinforcement learning and deep learning algorithms in the fog computing environment. IEEE Access **11**, 12555–12586 (2023). https://doi.org/10.1109/ACCESS.2023.3241881
19. Prudencio, R.F., Maximo, M.R.O.A., Colombini, E.L.: A survey on offline reinforcement learning: taxonomy, review, and open problems. IEEE Trans. Neural Netw. Learn. Syst. (2023). https://doi.org/10.1109/TNNLS.2023.3250269
20. Islam, M.T., Karunasekera, S., Buyya, R.: Performance and cost-efficient spark job scheduling based on deep reinforcement learning in cloud computing environments. IEEE Trans. Parallel Distrib. Syst. **33**(7), 1695–1710 (2022). https://doi.org/10.1109/TPDS.2021.3124670
21. Wang, H., Liu, Z., Shen, H.: Job scheduling for large-scale machine learning clusters. In: Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies. CoNEXT '20, pp. 108–120. Association for Computing Machinery, New York, NY, USA (2020).
22. Zhang, D., Dai, D., He, Y., Bao, F.S., Xie, B.: Rlscheduler: An automated hpc batch job scheduler using reinforcement learning. In: SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–15 (2020).
23. Cai, H., Zhu, L., Han, S.: ProxylessNAS: Direct neural architecture search on trget task and Hardware. Preprint at http://arxiv.org/abs/1812.00332 (2018)
24. Lu, B., Yang, J., Jiang, W., Shi, Y., Ren, S.: One proxy device is enough for hardware-aware neural architecture search. Proc. ACM Meas. Anal. Comput. Syst. **5**(3), 1–34 (2021). https://doi.org/10.1145/3491046
25. Wu, C.-J., Brooks, D., Chen, K., Chen, D., Choudhury, S., Dukhan, M., Hazelwood, K., Isaac, E., Jia, Y., Jia, B., Leyvand, T., Lu, H., Lu, Y., Qiao, L., Reagen, B., Spisak, J., Sun, F., Tulloch, A., Vajda, P., Wang, X., Wang, Y., Wasti, B., Wu, Y., Xian, R., Yoo, S., Zhang, P.: Machine learning at facebook: understanding inference at the edge. IEEE Int. Symp. High Perform. Comput. Archit. (2019). https://doi.org/10.1109/HPCA.2019.00048
26. Liu, Y., Sun, Y., Xue, B., Zhang, M., Yen, G.G., Tan, K.C.: A survey on evolutionary neural architecture search. IEEE Trans. Neural Netw. Learn. Syst. **34**(2), 550–570 (2023). https://doi.org/10.1109/TNNLS.2021.3100554
27. Khan, M.S.A., Santhosh, R.: Task scheduling in cloud computing using hybrid optimization algorithm. Soft Comput. **26**(23), 13069–13079 (2022). https://doi.org/10.1007/s00500-021-06488-5
28. Behera, I., Sobhanayak, S.: Task scheduling optimization in heterogeneous cloud computing environments: a hybrid ga-gwo approach. J. Parallel Distrib. Comput. **183**, 104766 (2024)
29. Huang, X., Li, C., Chen, H., An, D.: Task scheduling in cloud computing using particle swarm optimization with time varying inertia weight strategies. Clust. Comput. **23**(2), 1137–1147 (2020). https://doi.org/10.1007/s10586-019-02983-5

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.

Wangbo Shen received the BS from Changsha University, Changsha, China in 2012 and 2016, and MS degrees from Central South University, Changsha, China in 2016 and 2019 respectively. Currently, he is working toward the PhD degree in the School of Computer Science and Engineering, from the South China University of Technology, Guangzhou, China supervised by Dr. Weiwei Lin. His research interests mainly include Kernel learning, AutoML and edge computing.

Weiwei Lin (Senior Member, IEEE) received his B.S. and M.S. degrees from Nanchang University in 2001 and 2004, respectively, and his Ph.D. in Computer Application from South China University of Technology in 2007. Currently, he is a professor in the School of Computer Science and Engineering at South China University of Technology. His research interests include distributed systems, cloud computing, and AI application technologies. He has published more than 150 papers in refereed journals and conference proceedings. He has been a reviewer for many international journals, including IEEE TPDS, TSC, TCC, TC, TCYB, etc. He is a distinguished member of CCF and a senior member of the IEEE.

Wentai Wu (Member, IEEE) received his Bachelor and Master degrees in Computer Science from South China University of Technology in 2015 and 2018, respectively. Sponsored by CSC, he in 2022 received the degree of Ph.D. in Computer Science from the University of Warwick, United Kingdom. His research interests mainly include distributed systems, federated learning, machine learning, and sustainable computing.

Haijie Wu is currently working toward the MS degree in the School of Computer Science and Engineering, South China University of Technology, Guangzhou, China supervised by Dr. Weiwei Lin. His research interests mainly include cloud edge collaboration, edge computing, and AI algorithms.

Keqin Li (Fellow, IEEE) is a SUNY Distinguished Professor of Computer Science with the State University of New York. He is also a National Distinguished Professor with Hunan University, China. His current research interests include, fog computing and mobile edge computing, energy-efficient computing and communication, embedded systems and cyber-physical systems, heterogeneous computing systems, big data computing, high performance computing, computer architectures and systems, computer networking, ML, intelligent and soft computing. He is currently an associate editor of the ACM Computing Surveys and the CCF Transactions on High Performance Computing. He has served on the editorial boards of the IEEE Transactions on Parallel and Distributed Systems, the IEEE Transactions on Computers, the IEEE Transactions on Cloud Computing, the IEEE Transactions on Services Computing, and the IEEE Transactions on Sustainable Computing. He is an IEEE Fellow and an AAIA Fellow. He is also a Member of Academia Europaea (Academician of the Academy of Europe).