Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

A priority-aware hierarchical cooperative reinforcement learning algorithm for LLM training task scheduling in CPN environments

Kangyang Liu

liuky3@chinatelecom.cn

China Telecom (China)

Weihao Yu

China Telecom (China)

Cancan Huang

China Telecom (China)

Wenkuan Qu

China Telecom (China)

Ke Ruan

China Telecom (China)

Research Article

Keywords: Large language model, Computing power network, Task priority scheduling, Self-attention

Posted Date: November 23rd, 2026

DOI: https://doi.org/10.21203/rs.3.rs-11685592/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

A priority-aware hierarchical cooperative reinforcement learning algorithm for LLM training task scheduling in CPN environments

.

Abstract

The development of artificial intelligence has fueled a surge in computing demand that any single facility can no longer accommodate. The computing power network (CPN) addresses this challenge by interconnecting multiple computing nodes and integrating dispersed heterogeneous resources. A core challenge in CPN task scheduling is to jointly determine task priority ordering and node selection under task dynamics and resource heterogeneity. In this paper, we investigate the scheduling of large language model (LLM) training tasks in CPN environments and aim to jointly optimize task satisfaction and load balancing. To this end, we propose PA-HRL, a priority-aware hierarchical cooperative reinforcement learning algorithm that decomposes the scheduling problem into two coordinated subproblems: a high-level agent is responsible for task ordering, while a low-level agent makes node selection decisions. Both agents adopt the advantage actor-critic (A2C) algorithm combined with policy entropy and prioritized experience replay, and employ self-attention to extract the key features of task and node sequences. Simulation results under different scenarios show that PA-HRL achieves better task satisfaction and load balancing than other hierarchical cooperative deep reinforcement learning methods and the first-come-first-served random scheduling scheme (FCFS-Random), and exhibits a shorter decision time than the FCFS-based genetic algorithm scheme (FCFS-GA). In addition, we conduct ablation experiments to verify the effectiveness of the high-level agent and the contribution of each key component to PA-HRL.

Keywords: Large language model; Computing power network; Task priority scheduling; Self-attention

1 Introduction

In recent years, as the development of large language model (LLM) has achieved breakthrough progress in parameter scale and performance (ranging from TinyLlama (Zhang et al. 2024) with only 1.1B parameters to Kimi K3 (Team et al. 2026) with up to 2.8T parameters) and the applications of artificial intelligence are rapidly gaining popularity, the computing demand for artificial intelligence is growing dramatically. A single computing facility can no longer meet the computing requirements, and LLM training tasks often require more computing resources and collaborative work among multiple computing clusters (Narayanan et al. 2021). The computing power network (CPN), as a powerful network architecture for computing, integrates heterogeneous computing resources across multiple computing facilities (Yukun et al. 2024) and provides task scheduling and resource allocation schemes through typical scheduling platforms (Google's Borg (Verma et al. 2015) and Omega (Schwarzkopf

et al. 2013), Apache Mesos (Hindman et al. 2011), and ByteDance's Gödel (Xiang et al. 2023)).

In addition, current research on LLM mainly focuses on the optimization of LLM architecture, training methods (Fedus et al. 2022) and inference deployment (Han et al. 2015; Ruan et al. 2026; Zhang et al. 2025b), whereas the research on LLM training task deployment remains scarce. Current distributed training frameworks still exhibit significant deficiencies in task scheduling under dynamic resource environments and load balancing across heterogeneous devices (Lin et al. 2023). Therefore, it is of practical significance to formulate an efficient task scheduling strategy to guide the deployment and execution of LLM training tasks in CPN.

However, the resource heterogeneity and load dynamics of CPN, together with the characteristics of LLM training tasks, pose the following challenges to task scheduling:

1. Task heterogeneity and dynamic priority determination: LLM training tasks differ significantly in parameter scale, resource requirements, and delay or cost sensitivity. Therefore, the scheduling priority of tasks needs to be determined dynamically.
2. Dynamic preferences of tasks: Different users pay varying degrees of attention to various task factors—some expect to complete tasks with shorter delay, while others prefer lower costs. Therefore, the task scheduling strategy should adapt to the actual demands of each task to maximize task satisfaction.
3. Resource heterogeneity and load fluctuation: The computing nodes in CPN differ in geographical location, computing performance, and resource capacity. Meanwhile, the randomness of task arrivals leads to fluctuations in the load of each node. To promote load balancing, the task scheduling strategy should accurately perceive the load conditions of each node in CPN.

The task scheduling problem is generally NP-hard. Traditional mathematical methods such as integer programming or dynamic programming can hardly obtain exact solutions, while optimization algorithms such as particle swarm optimization (PSO) (Wang et al. 2019) and genetic algorithm (GA) (Devi et al. 2025; Kushala et al. 2025; Pang et al. 2019) suffer from slow convergence and complicated parameter tuning for large-scale problems. Deep reinforcement learning (DRL), which continuously interacts with the environment and learns the optimal decision through deep neural networks, has been widely applied to task scheduling problems owing to its powerful environmental perception and decision-making capabilities. Many studies employ a single DRL agent to execute intelligent task scheduling and optimize multiple objectives including delay, energy consumption, task satisfaction rate, and load balancing (Hao et al. 2024; Nath and Wu 2020; Xie et al. 2022). However, the centralized scheduling of a single agent faces several risks, including scheduling failures caused by damage to the centralized controller and the exponential growth of the state space as the problem scale increases. Therefore, some studies have proposed distributed multi-agent methods: Dong et al. (2023) proposes a MADDPG algorithm based on the CTDE (centralized training with decentralized execution) architecture and combines it with an LSTM, achieving the collaborative optimization of DNN inference accuracy, inference delay, and resource utilization, while Zhang et al. (2025a) introduces a MADDPG algorithm based on the DTDE (decentralized training with decentralized execution) architecture and compensates for the limitations of local observations through a partial communication mechanism. Additionally, several studies have demonstrated that combining DRL with attention can enhance the agent's capability of extracting key features, thereby improving task scheduling performance (He et al. 2025; Ma et al. 2025; Wang 2025).

Nevertheless, for joint decision problems, traditional single-layer DRL methods suffer from the growth of action dimensions and training instability (Li et al. 2023; Sheng et al. 2021). Consequently, recent studies have adopted hierarchical cooperative DRL methods for joint decision problems of task scheduling. For instance, both Cui et al. (2025) and Alsalem and Djemame (2025) employ two levels of agents to realize the cluster-to-node scheduling mapping. Zhu et al. (2026) also decouples the decisions of power control and task scheduling through two agents. For the joint decisions of task priority ordering and node selection, Chen et al. (2025); Liu and Liu (2025); Yue et al.

(2026) all design cooperative dual-layer architectures in which the high-level agent is responsible for task ordering and the low-level agent for node selection.

The above methods mostly adopt fixed weights among multiple task factors, so that the scheduling results cannot fit the dynamic preferences of users. To this end we design a priority-aware hierarchical cooperative reinforcement learning algorithm, PA-HRL, which incorporates task priority ordering to jointly optimize task satisfaction and load balancing. Task ordering methods can be divided into two categories: pre-scheduling ordering (Xie et al. 2025) and local scheduling ordering (Feng et al. 2024). The former is performed by a centralized controller that sorts task sequences before task assignment, while the latter performs ordering on computing nodes after tasks are assigned and is prone to load imbalance due to the lack of global information. Therefore, we adopt the pre-scheduling ordering method in our algorithm.

The main contributions of this paper are as follows:

1. We propose a relative sensitivity coefficient to characterize the dynamic preferences of tasks, and further introduce a metric to quantify task satisfaction, namely task scheduling penalty. Then we formulate the joint optimization problem of task satisfaction and load balancing.
2. We design a CPN architecture composed of multiple layers and propose a priority-aware hierarchical cooperative reinforcement learning algorithm, PA-HRL, which decouples task priority ordering from node selection through the collaboration between two levels of agents. The high-level agent uses self-attention to extract the key features of task sequences and then utilizes the pointer network to obtain task priority weights, while the low-level agent also extracts the key features of node sequences through self-attention and then fuses the task embedding features to make node selection decisions. Both agents adopt advantage actor-critic (A2C) combined with policy entropy and prioritized experience replay mechanisms for policy optimization.
3. Multiple simulation experiments show that PA-HRL achieves higher task satisfaction and load balancing compared with the first-come-first-served random scheduling (FCFS-Random) and other hierarchical cooperative DRL methods, and incurs a shorter decision time than the first-come-first-served scheduling with a genetic algorithm (FCFS-GA). Ablation experiments verify the effectiveness of the high-level agent and the contribution of each key component to PA-HRL.

The remainder of this paper is organized as follows: Section 2 reviews related work. Section 3 presents the system model and gives the mathematical formulation of the optimization problem. Section 4 details the design of PA-HRL. Section 5 presents the simulation experiments and analyzes the results. Section 6 summarizes this paper and outlines future work.

2 Related work

This section provides an overview of recent research on task scheduling from three perspectives: traditional optimization methods, single-layer DRL algorithms, and hierarchical cooperative DRL methods.

2.1 Traditional optimization methods

Traditional optimization methods have attracted extensive attention due to their mature theoretical foundations. Wang et al. (2019) proposes a particle swarm optimization algorithm with weighted adaptive inertia weight (WAIW-PSO) in the multi-edge-cloud collaboration scenario to improve the optimization performance of PSO by dynamically adjusting the inertia weight. Pang et al. (2019) combines the EDA and GA algorithms to achieve task-to-virtual-machine allocation, effectively reducing task completion delay and improving load balancing. Devi et al. (2025) also designs a task scheduling method based on the genetic algorithm, which aims to minimize the makespan and mean flowtime. Kushala et al. (2025) combines the artificial fish swarm algorithm with GA to reduce energy consumption and delay while improving the deadline satisfaction rate. Yang et al. (2020) applies a cooperative game framework to the rate allocation of computing nodes.

Traditional optimization methods are simple and efficient in static scenarios. Nevertheless, their adaptability and real-time scheduling performance will be greatly compromised when these methods face the resource heterogeneity and load dynamics of CPN.

2.2 Single-layer DRL algorithms

To overcome these limitations, many studies have adopted DRL-based algorithms. As a technique that continuously interacts with the environment and learns the optimal policy through deep neural networks, DRL achieves high performance in dynamic environments and has been widely applied to task scheduling problems.

For instance, Nath and Wu (2020) proposes a centralized DDPG algorithm for a multi-user single-server scenario to jointly optimize deployment location, transmission power, and computing resource allocation, so as to balance system delay and energy consumption. Hao et al. (2024) employs the D3QN algorithm combined with prioritized experience replay for task distribution to jointly optimize total task delay, energy consumption, and user rejection rate. Xie et al. (2022) designs a TD3-based centralized controller that utilizes the double Q-network and target policy smoothing techniques to address the training instability of traditional DDPG, thereby shortening task delay and improving load balancing performance.

To avoid scheduling failures caused by damage to the centralized controller and to enhance the robustness of the system, some studies have proposed distributed frameworks.Dong et al. (2023) introduces a MADDPG algorithm based on the CTDE (centralized training with decentralized execution) architecture integrated with an LSTM to collaboratively optimize DNN inference accuracy, delay, and resource utilization. Zhang et al. (2025a) also introduces a MADDPG algorithm based on the DTDE (decentralized training with decentralized execution) architecture and incorporates a partial inter-agent communication mechanism to supplement the limitations of local observations.

In recent years, some studies have combined Transformer or attention mechanisms with DRL to enhance the extraction of key features and temporal prediction capabilities of DRL. Wang (2025) combines DDQN with multi-head attention to extract key features from server CPU, storage, and network resource utilization as well as server queue length states, thereby achieving the comprehensive optimization of task completion delay, energy consumption, and resource utilization. He et al. (2025) integrates the attention mechanism into DRL to compute attention scores based on task levels and deadlines to minimize urgent task delay and energy consumption. Ma et al. (2025) introduces an encoder-decoder architecture based on Transformer for task scheduling in mobile edge computing.

Some studies also adopt single-layer DRL algorithms to handle joint decision-making in the task scheduling problem. Sheng et al. (2021) proposes a policy-based DRL algorithm to jointly decide task scheduling order and task assignment in the edge IoT scenario. Li et al. (2023) develops an improved PPO algorithm, MEPPO, to jointly output task priority and offloading decisions for vehicular edge computing.

Nonetheless, when facing joint decision-making problems, single-layer DRL suffers from an exponential growth of the action space as the number of tasks or nodes increases, and poor policy convergence.

2.3 Hierarchical cooperative DRL methods

For joint decision problems, adopting a hierarchical cooperative approach is a more reasonable scheme. The hierarchical cooperation among agents decouples the joint actions and reduces the action dimension of each agent.

The effectiveness of the hierarchical cooperation mechanism for task scheduling has been verified. For example, Cui et al. (2025) presents a hierarchical reinforcement-learning-based framework, HITS, for large-scale task scheduling in cloud computing environments. HITS breaks down the large-scale decision space into controllable subproblems through two-level decomposition: the first-level agent determines the cluster to which a task is assigned, and then the second-level agent selects the virtual machine within the cluster. Alsalem and Djemame (2025) introduces a hierarchical federated DRL architecture, HFLDRL, in which the global agent FL-DQNc is responsible for assigning tasks to edge clusters and the

local agent FL-DQNn within a cluster is responsible for assigning tasks to specific edge nodes. The local agents in each cluster achieve collaborative training through federated learning. Zhu et al. (2026) proposes a hierarchical reinforcement learning algorithm, HIRL, for multi-dimensional resource orchestration in mobile edge computing environments. HIRL decomposes the hybrid continuous-discrete optimization problem into two collaborative sub-modules: the upper layer adopts TD3 for continuous power control, while the lower layer adopts DDQN for discrete task assignment.

For the joint scheduling problem that incorporates task priority ordering, Liu and Liu (2025) designs a dual-layer DRL architecture, DaDRLA, in which the task priority agent determines task priorities to obtain the currently scheduled task, and the deployment agent makes node selection decisions. Both layers of agents adopt an encoder-decoder structure combined with multi-head attention and the pointer network. Chen et al. (2025) proposes a multi-agent advantage actor-critic algorithm, MA2C, which also adopts the encoder-decoder architecture. The improvement of MA2C over DaDRLA is that its encoder adopts a dual-channel design combining TCN and attention, which improves the model's prediction accuracy for users' dynamic preferences. Yue et al. (2026) introduces a hierarchical deep reinforcement learning approach, CPN-HRL, for the priority-aware task scheduling in CPN enabled by cloud-edge-end environments. In CPN-HRL, the high-level agent (LSTM-PPO) performs temporal prediction on task sequences and outputs global task priorities, while the low-level agent (GAT-PPO) fuses network topology and resource features through graph attention and outputs node selection decisions.

Table 1 summarizes some of the existing methods and our algorithm. An in-depth analysis reveals that existing methods generally adopt fixed weights among multiple task factors (e.g., delay and cost), so that the scheduling results cannot satisfy the actual demands of users. Therefore, we design a priority-aware hierarchical cooperative reinforcement learning algorithm, PA-HRL, to ensure task satisfaction and load balancing in CPN environments.

3 System model and problem formation

In this section, we first present the CPN architecture, then introduce the system model, including the delay model, the resource model, and the cost model. Finally, the mathematical expression of the optimization problem is given.

3.1 The CPN architecture and task scheduling process

Due to the limited hardware capabilities of current personal terminal devices, most of the LLM training tasks need to be offloaded to the CPN. Drawing on the scenario proposed by Xie et al. (2022), we design a CPN architecture consisting of four layers: the access layer, the routing layer, the resource layer, and the scheduling layer. Figure 1 shows the overall architecture of CPN.

Access layer: The access layer contains the task users and the CPN routers. It is assumed that the set of users in the system is denoted by $\mathcal{M} = \{1, 2, \ldots, M\}$ and is distributed over different geographic regions $\mathcal{A}_i = \{1, 2, \ldots, A\}$, and the LLM training task submitted by a user can be characterized as:

$$u_j = (prm_j, t_j, h_j, s_j, b_j, l_j, \tau_j, w_j), \quad j \in \mathcal{M} \quad (1)$$

where prm_j is the parameter size of the LLM (in billions), t_j is the total number of token samples used in model training, h_j and s_j are the hidden layer depth and the hidden layer size of the model, respectively, b_j is the batch size of the LLM, which determines the concurrent processing capability of the LLM, l_j is the average sequence length of a single sample, τ_j is the expected training duration of the LLM, and w_j is the relative sensitivity coefficient of the task, which represents the relative preference of a task for delay versus cost and specifically reflects the task requirements.

After a user generates an LLM training task, the task request is sent to the CPN router in its region through wireless or wired access, and the CPN router then forwards the task request to the scheduling platform for task scheduling decisions.

Routing layer: The routing layer contains ordinary routers located within the operator's backbone network and metropolitan area network,

Table 1 Comparison of our algorithm with existing methods

Paper	Optimization objective	Dynamic demand	Task priority	Load balancing	Attention	Algorithm type
Wang et al. (2019)	Total delay	×	×	×	×	Traditional optimization methods
Pang et al. (2019)	Total delay and load balancing	×	×	✓	×	
Devi et al. (2025)	Completion rate and mean flow time	×	×	×	×	
Kushala et al. (2025)	delay, energy consumption, and satisfaction rate	×	×	×	×	
Yang et al. (2020)	Task reliability and energy consumption	×	×	×	×	
Nath and Wu (2020)	delay and energy consumption	×	×	×	×	Single-layer DRL methods
Dong et al. (2023)	Inference accuracy, delay, and resource utilization	×	×	×	×	
Zhang et al. (2025a)	delay and energy consumption	×	×	×	×	
Hao et al. (2024)	Total delay, energy consumption, and service rejection rate	×	×	×	×	
Xie et al. (2022)	delay and load balancing	×	×	×	×	
Wang (2025)	delay, energy consumption, and resource utilization	×	×	×	✓	
He et al. (2025)	Task delay and energy consumption	×	✓	×	✓	
Ma et al. (2025)	delay and energy consumption	×	×	×	✓	
Sheng et al. (2021)	Task satisfaction rate	×	✓	×	×	
Li et al. (2023)	delay and energy consumption	×	✓	✓	×	
Liu and Liu (2025)	Energy consumption	×	✓	×	✓	Hierarchical cooperative DRL methods
Chen et al. (2025)	Total delay and task success rate	×	✓	×	✓	
Yue et al. (2026)	Task completion rate, average delay, and resource utilization	×	✓	×	✓	
PA-HRL	task satisfaction and load balancing	✓	✓	✓	✓	

and is responsible for forwarding task data to the designated computing nodes.

Resource layer: The resource layer consists of multiple computing nodes $\mathcal{N} = \{1, 2, \ldots, N\}$ which differ in performance, capacity, and cost. Each computing node can be characterized as

$$e_i = (A_i, G_i, F_i, D_i, B_i, p_i^G, L_i), \quad i \in \mathcal{N} \tag{2}$$

where A_i represents the data center region to which node i belongs, G_i represents the total number of GPUs on node i, and F_i is the parameter characterization of the GPU board model on node i:

$$F_i = (f_i, \eta_i, m_i) \tag{3}$$

where f_i, η_i, and m_i denote the peak floating-point performance (TFLOPS), the computational efficiency, and the per-card memory size (GB) of the GPU card, respectively, D_i is the total storage size (TB) of node i, p_i^G is the unit rental cost of a GPU card on the node (per card per day), B_i is the transmission bandwidth (GB/s) of the node, and L_i is the queue length of node i with $L_i \leq L$. When the resources are sufficient, a computing node can execute multiple tasks in parallel; when the idle resources are insufficient, tasks need to wait in the queue.

Scheduling layer: The scheduling layer hosts the scheduling platform, which serves as the core of CPN. The platform can communicate with each CPN router and each computing node to collect task requests and the resource status of computing nodes. There are two task request queues on the platform: the scheduling queue and the cache queue, whose lengths are L_s and L_c, respectively. When a task request arrives, if the scheduling queue length is smaller than L_s, the platform places the task request into the scheduling queue to participate in the current round of scheduling; otherwise, the task request is placed into the cache queue to wait for the next round of scheduling, and the current task is abandoned if the number

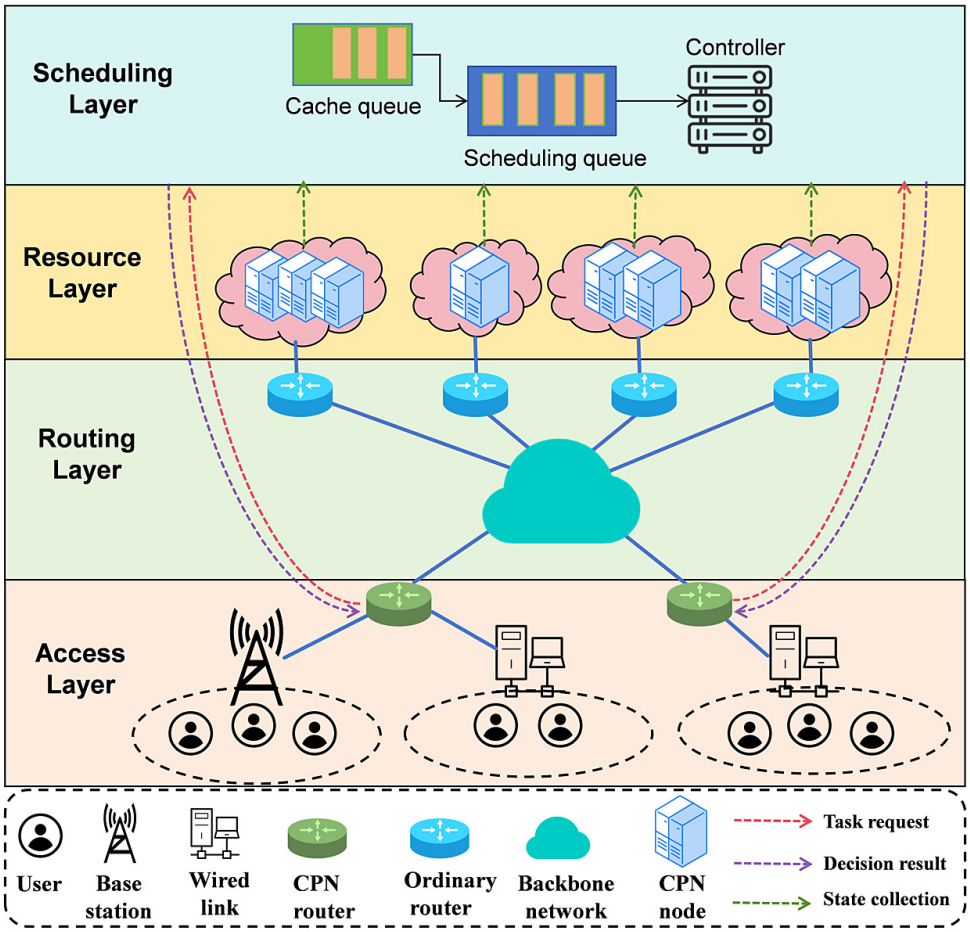


Fig. 1 The CPN architecture

of cached task requests has already exceeded L_c. The platform makes scheduling decisions through its internal controller and sends the decisions to the CPN routers.

Task scheduling process: In the CPN architecture, each task flows through the access layer, the scheduling layer, and the routing layer to the resource layer, and is finally executed on a computing node. Figure 2 shows the complete process of task scheduling, and the specific scheduling steps are as follows:

1. In the access layer, each user submits task requests to their corresponding CPN routers.
2. Each CPN router uploads task requests to the scheduling platform, which then places requests into the scheduling queue or the cache queue.
3. The scheduling platform periodically collects the status information of each computing node.
4. In combination with current task requests and the status information of computing nodes, the scheduling platform utilizes the controller to make task scheduling decisions and issues them to the corresponding CPN routers.
5. CPN routers forward task data to the corresponding computing nodes according to the scheduling decisions through the backbone network or the metropolitan area network.
6. Each computing node receives tasks and allocates computing resources to execute them, or places them into its waiting queue.
7. After tasks are completed, the computing nodes return the task results (model parameters) to users and release the occupied resources.

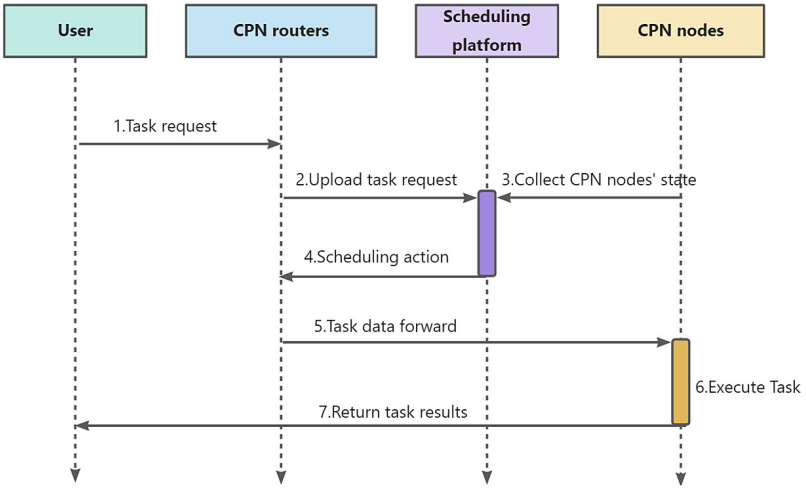


Fig. 2 Task scheduling process in CPN

3.2 Delay model

The completion delay of an LLM training task consists of transmission delay and request delay:

$$T_j = T_j^{\text{Trans}} + T_j^{\text{Req}} \tag{4}$$

For a task $u_j = (prm_j, t_j, h_j, s_j, b_j, l_j, \tau_j)$, the transmitted data mainly includes training samples and initial model parameters. For an LLM, the number of bytes occupied by a single token varies with the tokenizer and text encoding scheme adopted. For simplicity, we assume the widely used BPE tokenizer and English text, where one token occupies 4 bytes on average. Meanwhile, the size of model parameters is $2 \cdot prm_j$ bytes under FP16 precision. Therefore, the total transmission bytes are approximately $d_j \approx 4 \cdot t_j + 2 \cdot prm_j$. Suppose the number of hops from the task source node (the CPN router to which the task is uploaded) to the deployment node i is $k_{i,j}$ and the average forwarding delay per hop is t_{hop}; ignoring propagation delay and the delay of uploading tasks to CPN routers, the transmission delay is:

$$T_j^{\text{Trans}} \approx \frac{d_j}{B_i} + k_{i,j} \cdot t_{\text{hop}} \tag{5}$$

The request delay includes waiting delay and computation delay:

$$T_j^{\text{Req}} = T_j^{\text{Wait}} + T_j^{\text{Comp}} \tag{6}$$

We design the following execution scheme: when a task arrives, if the remaining resources (GPU and storage) on the computing node satisfy the task requirement, a virtual machine is created and the corresponding resources are allocated to execute the task; otherwise the task will be placed into the node's waiting queue, thereby incurring the waiting delay T_j^{Wait}.

The computation delay is related to the model parameters, the total number of sample tokens, and the performance and quantity of GPUs used:

$$T_j^{\text{Comp}} = \frac{6 \cdot prm_j \cdot t_j}{n_{i,j} \cdot f_i \cdot \eta_i} \tag{7}$$

where the numerator represents the computational workload of an LLM training task.

3.3 Resource model

The number of GPUs $n_{i,j}$ required for LLM training is determined by the memory occupied during the training process and the expected computation delay τ_j.

Specifically, the memory occupied during the LLM training process includes parameter memory, gradient memory, optimizer memory, and activation memory. Since a gradient has to be computed once for each model parameter during training, the parameters and gradients each occupy $2 \cdot prm_j$ bytes under FP16 precision. The commonly used

optimizer is the Adam optimizer, which computes a momentum and a variance for each model parameter and needs to retain a copy of the parameters in mixed-precision training. Each state parameter of the optimizer generally occupies 4 bytes, so the memory occupied by the optimizer is $12 \cdot prm_j$. Meanwhile, the activation memory is also a non-negligible factor and its memory estimation is $b_j \cdot l_j \cdot h_j \cdot s_j$. Therefore, the total memory occupied during the LLM training process under FP16 precision is:

$$Mem_j = 16 \cdot prm_j + b_j \cdot l_j \cdot h_j \cdot s_j \tag{8}$$

Then, the number of GPUs required based on the total memory is computed as:

$$n_{i,j}^m = \frac{Mem_j}{m_i} \tag{9}$$

And the number of GPUs required to satisfy the expected computation delay is computed as:

$$n_{i,j}^c = \frac{6 \cdot prm_j \cdot t_j}{\tau_j \cdot f_i \cdot \eta_i} \tag{10}$$

In order to satisfy the total memory requirement during the LLM training process and the expected computation delay, the final number of GPUs required to complete an LLM training task is:

$$n_{i,j} = \max(n_{i,j}^m, n_{i,j}^c) \tag{11}$$

3.4 Cost model

The total cost of an LLM training task consists of transmission cost and computation cost:

$$P_j = p_j^{\text{Trans}} + p_j^{\text{Comp}} \tag{12}$$

Operators currently adopt several billing schemes for data transmission, such as billing by traffic volume and billing by bandwidth duration. In the bandwidth-duration billing scheme, operators usually set tiered pricing for bandwidth, i.e., different bandwidth tiers correspond to different unit-duration prices, and the higher the bandwidth tier, the wider the bandwidth and the higher the unit-duration price. Assuming that the unit-duration price of the corresponding bandwidth tier is $P(B_i)$, the transmission cost of task u_j is:

$$p_j^{\text{Trans}} = P(B_i) \cdot T_j^{\text{Trans}} \tag{13}$$

The computation cost of a task is the rental cost of GPUs. For a task deployed on node i, the rental price per card per day is p_i^G, so the basic cost is $p_i^G \cdot n_{i,j} \cdot T_j^{\text{Comp}}$. In practice, the electricity cost of the region where the node is located affects the pricing of the node's computing resources to a certain extent, because a large amount of electricity is consumed in the process of using GPUs to perform computations. The computing resource providers often raise the pricing of computing resources to cover electricity costs. For example, regions with vast territory and abundant green energy usually have lower electricity costs, whereas regions with strong electricity demand and tight energy supply have higher electricity costs, making the pricing of computing resources in such regions correspondingly higher.

We adopt the city vitality ranking indicator proposed by Xiang et al. (2020) as an indirect reflection of the influence of regional electricity costs on the pricing of computing resources. The city vitality indicator is a comprehensive measure of a city's economy, covering three indicators: the number of enterprises, the economic density, and the industrial structure. The higher the economic level of a city, the stronger its electricity demand tends to be and the higher the corresponding electricity cost. Table 2 presents the city vitality indicators of major cities in China.

Let the city vitality indicator of the region with the lowest value be $E_{\text{base}} = \min_i(E_i)$. Then, the influence of electricity costs of a certain region i on the basic pricing of computing resources can be represented by a cost influence coefficient $V_i = E_i / E_{\text{base}}$, which directly acts on p_i^G to raise the unit price of the computing resources in that region. Accordingly, the computation cost of task u_j is:

$$p_j^{\text{Comp}} = p_i^G \cdot n_{i,j} \cdot T_j^{\text{Comp}} \cdot V_i \tag{14}$$

3.5 Problem formulation

Our goal is to achieve a lower completion delay or cost according to the actual demands of tasks, so as to maximize task satisfaction, while pursuing load balancing in CPN. Task satisfaction can be quantified by the task scheduling penalty ϱ_j:

Table 2 City vitality indicators of major cities in China

City	Enterprises	Economic density	Industrial structure	Economic vitality	Ranking	Vitality difference
Shenzhen	203.1	112627.68	0.58	26751.86	1	–
Shanghai	204.8	48317.02	0.69	11492.63	2	15259.23
Dongguan	53.4	30758.99	0.51	7305.89	3	4186.739
Guangzhou	110.2	28925.41	0.71	6878.61	4	427.2814
Suzhou	63.8	20403.69	0.51	4850.22	5	2028.393
Nanjing	64.6	17758.22	0.60	4222.65	6	627.5662
Beijing	152.1	17071.85	0.81	4071.74	7	150.9093
Wuhan	52.6	15787.25	0.53	3753.33	8	318.4133
Tianjin	62.0	15527.53	0.58	3693.00	9	60.33057
Zhengzhou	53.3	12347.30	0.60	2937.23	10	755.7694
Ningbo	44.4	10026.55	0.45	2385.29	11	551.9361
Qingdao	55.6	9783.09	0.55	2329.09	12	56.20445
Chengdu	85.0	9510.03	0.53	2268.27	13	60.81439
Changsha	36.8	8638.72	0.51	2054.98	14	213.2893
Hangzhou	64.1	7594.22	0.63	1810.90	15	244.0828
Xi'an	51.4	7392.06	0.61	1761.20	16	49.69611
Shenyang	33.4	4467.64	0.06	1064.63	17	696.5781
Kunming	33.2	2424.86	0.57	580.11	18	484.517
Chongqing	97.5	2366.48	0.49	574.94	19	5.166027

$\varrho_j = w_j \cdot \hat{T}_j + (1 - w_j) \cdot \hat{P}_j$, where w_j is the relative sensitivity coefficient of the task. $\hat{T}_j$ is the actual completion delay normalized with respect to the estimated minimum and maximum completion delays under the current load level, i.e., the delay penalty, while $\hat{P}_j$ is the total cost normalized with respect to the estimated minimum and maximum cost, i.e., the cost penalty. Therefore, the smaller the task scheduling penalty ϱ_j, the higher the task satisfaction.

The load balancing level of CPN can be represented by the variance of comprehensive resource utilization across all computing nodes, which is be computed as:

$$\sigma = \frac{\sum_{i=1}^{N}(U_i - \bar{U})^2}{N} \tag{15}$$

where U_i represents the comprehensive resource utilization of node i and is defined as:

$$U_i = \zeta \cdot U_i^G + (1 - \zeta) \cdot U_i^D \tag{16}$$

where U_i^G and U_i^D are the GPU and storage utilization of node i, respectively, and ζ is their relative weight. The smaller the load variance, the more balanced the resource utilization across computing nodes.

Then, the optimization problem of this paper is to minimize the sum of the task utilities of all tasks, while minimizing load variance to ensure load balancing. Let a binary variable $s_{i,j}$ indicate whether task u_j is deployed on node i. If task u_j is deployed on node i, then $s_{i,j} = 1$; otherwise, it equals 0. The mathematical expression of the optimization problem is as follows:

$$\min_{s_{i,j}} \quad \sum_{j=1}^{M} \varrho_j \cdot \sigma \tag{17}$$

s.t.

$$\text{C1: } \sum_{j \in T_i^e} n_{i,j} \le G_i, \qquad \forall i \in \mathcal{N} \tag{17a}$$

$$\text{C2: } n_{i,j} \ge \max(n_{i,j}^m, n_{i,j}^c), \quad \forall i \in \mathcal{N}, j \in \mathcal{M} \tag{17b}$$

$$\text{C3: } \sum_{j \in T_i} d_j \le D_i, \qquad \forall i \in \mathcal{N} \tag{17c}$$

$$\text{C4: } L_i = \sum_{j \in T_i^W} s_{i,j} \le L, \qquad \forall i \in \mathcal{N} \tag{17d}$$

$$\text{C5: } s_{i,j} \in \{0, 1\}, \qquad \forall i \in \mathcal{N}, j \in \mathcal{M} \tag{17e}$$

$$\text{C6: } \sum_{i=1}^{N} s_{i,j} = 1, \qquad \forall j \in \mathcal{M} \tag{17f}$$

C1 is the constraint on the GPU resources occupied by all currently executing tasks on a node, where T_i^e denotes the set of tasks currently being executed on node i. C2 states that the number of allocated GPUs must simultaneously satisfy the memory requirement of LLM training and the expected computation delay. C3 is the storage capacity constraint, stating that the total data volume of all tasks deployed on a node must

not exceed the storage capacity. C4 is the waiting queue length constraint of a node, where T_i^W denotes the set of tasks currently queued on node i. C5 restricts the decision variable $s_{i,j}$ to a binary value. C6 states that a task can only be deployed on one node, i.e., it cannot be deployed across multiple nodes.

4 The design of PA-HRL

To solve the optimization problem, this section introduces a priority-aware hierarchical cooperative reinforcement learning algorithm, PA-HRL, whose framework is shown in Figure 3.

PA-HRL decouples task priority ordering from node selection and constructs two-level DRL agents: the high-level agent employs an encoder-decoder architecture and is responsible for performing priority ordering on task sequences, while the low-level agent adopts self-attention to extract the key features of node sequences and fuse them with task embedding features, and is responsible for making node selection decisions. Both agents adopt A2C combined with policy entropy and prioritized experience replay to achieve policy exploration and stable training.

Subsequent sections detail the design principles, the training method, and the time complexity of PA-HRL.

4.1 High-Level agent

The goal of the high-level agent is to realize dynamic priority ordering of task sequences. Its priority decision process can be modeled as a Markov decision process (MDP).

4.1.1 MDP definition

State space S_h: At time slot t, the state of the high-level agent is $s_t^h = X_{task}^t = (u_1^t, u_2^t, \ldots, u_j^t, \ldots, u_{L_s}^t)$, where X_{task}^t is the feature matrix of the task sequence, and each row u_j^t corresponds to the feature vector of a single task.

Action space A_h: The action of the high-level agent is the task ordering weight $a_t^h = (p_1^{task}, p_2^{task}, \ldots, p_j^{task}, \ldots, p_{L_s}^{task})$, where p_j^{task} denotes the priority weight of the j-th task in the scheduling queue and serves as the basis for task priority ordering.

Reward function r_h: The priority ordering mainly affects the waiting delay of tasks. Tasks placed at the front of the scheduling queue are scheduled first and are more likely to obtain shorter waiting delays under the current workload of the CPN, whereas tasks placed at the back of the scheduling queue are scheduled later, obtaining longer waiting delay due to the occupation of resources caused by tasks already deployed

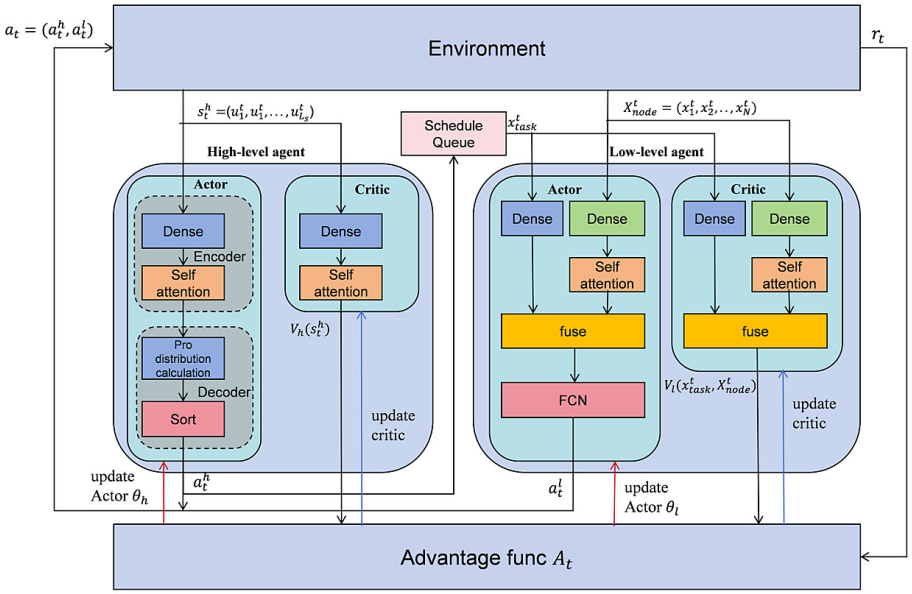


Fig. 3 The framework of PA-HRL

on nodes. In theory, tasks that are more delay-sensitive should be scheduled first, while tasks that are more cost-sensitive can be scheduled later because they can tolerate longer waiting delay. Therefore, the reward function of the high-level agent is designed as follows:

$$r_h = -\left(\frac{\sum_{j=1}^{L} w_j \cdot T_j^{\text{Wait}}}{L}\right) \tag{18}$$

where w_j is the relative sensitivity coefficient of the task, T_j^{Wait} denotes waiting delay, and L is the length of the scheduling queue, i.e., the number of tasks to be scheduled.

4.1.2 Policy network

The policy network of the high-level agent adopts an encoder-decoder structure, where the encoder adopts self-attention to extract the dynamic preferences of tasks and other key features to improve the accuracy of task prediction, and the decoder adopts a pointer network to compute the priority weights and thereby realize task priority ordering.

Encoder

Task sequence encoding: The input to the encoder of the policy network is the task feature matrix X_{task}^t. It is preprocessed through a fully connected layer to obtain the initial embedding vectors $\tilde{X}_{task}^t = (e_1^t, e_2^t, \ldots, e_{L_s}^t)$. The calculation formula for this step can be described as

$$e_j^t = w_t \cdot u_j^t + b_t, \quad j \in \{1, 2, \ldots, L_s\} \tag{19}$$

where w and b are both learnable parameters.

Self-attention computation: Then, the task embedding vectors $\tilde{X}_{task}^t$ are processed by self-attention. The self-attention first generates a query vector $q_j^t = w_q \cdot e_j^t$, a key vector $k_j^t = w_k \cdot e_j^t$, and a value vector $v_j^t = w_v \cdot e_j^t$ for each data point in the embedding matrix, and then computes the dot product of each query vector with all key vectors, followed by softmax normalization to obtain a set of attention scores, which reflect the similarity between the query vector and each key vector:

$$\xi_{j,i}^t = \text{softmax}\left(\frac{q_j^t \cdot (k_i^t)^{\mathrm{T}}}{d_k}\right) \tag{20}$$

where $(k_i^t)^{\mathrm{T}}$ denotes the transpose of the key vector k_i^t, and d_k is the dimension of k_i^t.

The attention scores are then used to weight and sum the value vectors v_j^t, and the attention vector of each data point is obtained:

$$o_j^t = \sum_{i=1}^{L_s} \xi_{j,i}^t \cdot v_i^t, \ j \in \{1, 2, \ldots, L_s\} \tag{21}$$

Encoder output: Finally, the attention vectors are connected with the embedding vectors of the original data points via residual connections, and the output of the encoder is obtained through layer-normalization:

$$E_{task}^t = (e_1^{t'}, e_2^{t'}, \ldots, e_{L_s}^{t'}) \tag{22}$$

$$e_j^{t'} = \text{Layernorm}(e_j^t + o_j^t), \quad j \in \{1, 2, \ldots, L_s\} \tag{23}$$

Decoder

In the task priority ordering problem, the elements of the output sequence are usually selected from the input sequence, and the order of elements in the output sequence has a direct correspondence with that of the input sequence. Inspired by this, we use the pointer network (Vinyals et al. 2015) to construct the decoder.

Computation of the pointer vector and matching vectors: In the decoder of the policy network, the output of the encoder is used as the input of the decoder. First, the encoder output is mean-pooled: $\bar{E}_{task}^t = \frac{1}{L_s}(e_1^{t'} + e_2^{t'} + \cdots + e_{L_s}^{t'})$. Then, the pointer vector $Pt_t = w_z \cdot \bar{E}_{task}^t$ and the matching vectors $Ma_j^t = w_m \cdot e_j^{t'}$ are computed separately.

Task priority weight computation: The pointer vector Pt_t is then used to compute the attention weight of each matching vector Ma_j^t. Finally, the task priority weights are obtained through the softmax function. Here, we adopt the weight computation method used by Chen et al. (2025):

$$p_j^{task} = \text{softmax}(C \cdot \tanh(u_j^t)), \ j \in \{1, 2, \ldots, L_s\} \tag{24}$$

$$u_j^t = \begin{cases} \dfrac{Ma_j^t \cdot (Pt_t)^{\mathrm{T}}}{d_{hid}}, & \text{if the task is valid} \\ -\infty, & \text{otherwise} \end{cases} \tag{25}$$

In the computation of p_j^{task}, C is a constant, and $\tanh(\cdot)$ is adopted as the activation function to avoid the influence of excessively large or small individual inputs, thereby improving numerical stability. d_{hid} is the dimension of the matching vector.

The task priority weights are then used as the selection probabilities of tasks, and the task sequence is ordered according to these probabilities, thereby obtaining the scheduling order of the current task sequence.

4.2 Low-Level agent

The low-level agent focuses on dynamically mapping task priority sequences to nodes. Its node selection decision-making process can also be modeled as a Markov decision process.

4.2.1 MDP definition

State space S_l: At time slot t, the state of the low-level agent is $s_t^l = (x_{task}^t, X_{node}^t)$, where x_{task}^t is the feature of the currently scheduled task selected by the high-level agent, and $X_{node}^t = (x_1^t, x_2^t, \ldots, x_N^t)$ is the feature matrix of the current computing nodes, and each row x_i^t corresponds to the feature vector of a single node:

$$x_i^t = (F_i, B_i, S_i^G, S_i^D, U_i^G, U_i^D, L_i, p_i^G, V_i, W_i) \tag{26}$$

where F_i is the performance characterization of the GPU boards on node i, i.e., $F_i = (f_i, \eta_i, m_i)$, B_i is the node bandwidth, S_i^G, S_i^D are the current remaining amounts of GPU and storage on the node, respectively, U_i^G, U_i^D are the current utilization rates of the node's GPU and storage, respectively, L_i is the waiting queue length on the node, i.e., the current number of waiting tasks, p_i^G, V_i are the unit rental cost of a GPU card on the node and the cost influence coefficient of the region where the node is located, respectively, and W_i is the estimated waiting delay of the task on node i.

Action space A_l: The action of the low-level agent is the node selection probability vector $a_t^l = (p_1^{node}, p_2^{node}, \ldots, p_i^{node}, \ldots, p_N^{node})$, and $\sum_{i=1}^N p_i^{node} = 1$, where p_i^{node} is the probability of selecting node i as the task deployment node. During the training phase, the corresponding deployment node is obtained by sampling according to the probabilities, whereas during the testing phase, the node with the highest probability is selected as the task deployment node $i^* = \text{argmax}_i\, p_i$.

Reward function r_l: The low-level agent focuses on task scheduling penalty and load balancing, so its reward function is defined as:

$$r_l = -p_{l,task} \cdot p_{l,D} \tag{27}$$

where $p_{l,task}$ represents the task deployment penalty, $p_{l,task} = w_j \cdot \hat{T}_j + (1 - w_j) \cdot \hat{P}_j$, and $p_{l,D}$ represents the load balancing penalty, $p_{l,D} = \sigma_t$, i.e., the variance of comprehensive resource utilization across computing nodes at the current time.

4.2.2 Policy network

The policy network of the low-level agent consists of an encoder and fully connected layers, where the encoder adopts self-attention to extract the key features of the node sequences, which are fused with task embedding features and then mapped through multiple fully connected layers to output node selection probabilities.

Task feature encoding: The features of the currently scheduled task are embedded through a fully connected layer:

$$\tilde{x}_{task}^t = w_t \cdot x_{task}^t + b_t \tag{28}$$

Node sequence encoding: Each row of the node feature matrix X_{node}^t is also passed through a fully connected layer to obtain the corresponding embedding matrix: $\tilde{X}_{node}^t = (m_1^t, m_2^t, \ldots, m_N^t)$, $m_i^t = w_n \cdot x_i^t + b_n,\ i \in \{1, 2, \ldots, N\}$, where both w_n and b_n are learnable parameters.

Self-attention computation: The node sequence embedding matrix $\tilde{X}_{node}^t$ is passed through self-attention to compute the attention scores among nodes. Similarly, a query vector $q_i^t = w_q \cdot m_i^t$, a key vector $k_i^t = w_k \cdot m_i^t$, and a value vector $v_i^t = w_v \cdot m_i^t$ are generated for each data point in the embedding matrix. Then, the attention vector o_i^t is computed and the attention output is obtained through residual connection and layer normalization:

$$M_{node}^t = (m_1^{t'}, m_2^{t'}, \ldots, m_N^{t'}) \tag{29}$$

$$m_i^{t'} = \text{Layernorm}(m_i^t + o_i^t), \quad i \in \{1, 2, \ldots, N\} \tag{30}$$

Feature fusion and output: Then, the attention output is fused with task embedding features to obtain the fused feature vector:

$$U^t = [\tilde{x}_{task}^t \,\|\, M_{node}^t] \tag{31}$$

The fused feature vector is then mapped through multiple fully connected layers, and the node selection probabilities p_i^{node} $(i = 1, 2, \ldots, N)$ are finally obtained in the output layer through the softmax function.

4.3 Training method

To ensure the effective coordination between the high-level and low-level agents in PA-HRL, we adopt the two-stage training method proposed by Yue et al. (2026) and combine it with the gradient update method of A2C.

The system policy is the combination of the high-level policy $\pi_h(a_t^h \mid s_t^h; \theta_h)$ and the low-level policy $\pi_l(a_t^l \mid s_t^l; \theta_l)$, and the overall policy gradient is expressed as:

$$\begin{aligned}\nabla_\theta J(\theta) = \mathbb{E}\Big[\sum_t & A_t^h \cdot \nabla_{\theta_h} \log \pi_h(a_t^h \mid s_t^h; \theta_h) \\ & + A_t^l \cdot \nabla_{\theta_l} \log \pi_l(a_t^l \mid s_t^l; \theta_l) \Big]\end{aligned} \tag{32}$$

where A_t^h and A_t^l are the advantage functions of the high-level and low-level agents, respectively. The advantage function is computed through the generalized advantage estimation (GAE) (Schulman et al. 2015):

$$A_t = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}, \quad \delta_t = r_t + \gamma V(S_{t+1}) - V(S_t) \tag{33}$$

where γ is the discount factor, λ is the GAE smoothing parameter, and δ_t denotes the temporal-difference (TD) error at time t.

In PA-HRL, we set the smoothing parameter to 0, and then A_t is simplified to depend only on the TD error at the current moment: $A_t = r_t + \gamma V(S_{t+1}) - V(S_t)$. The critic network updates its parameters by minimizing the TD error, and its loss function is:

$$L_{critic} = \frac{1}{2} \mathbb{E}_{s_t \sim \rho_\pi} \left[(r_t + \gamma V(s_{t+1}) - V(s_t))^2 \right] \tag{34}$$

To prevent the policy from falling into a local optimum, we adopt the policy entropy method during the DRL training process to increase exploration and employ an entropy coefficient decay mechanism in the form of cosine annealing to avoid policy oscillation. The loss function of the policy network is expressed as:

$$\begin{aligned}L_{actor} = & - \mathbb{E}_{s_t \sim \rho_\pi} \left[A_t \cdot \log \pi_\theta(a_t \mid s_t) \right] \\ & - \alpha_H \cdot H(\pi_\theta(a_t \mid s_t))\end{aligned} \tag{35}$$

where $H(\pi_\theta(a_t \mid s_t))$ represents the policy entropy:

$$\begin{aligned}H(\pi_\theta(a_t \mid s_t)) = -\mathbb{E}_{s_t \sim \rho_\pi} \Big[& \sum_a \pi_\theta(a_t \mid s_t) \\ & \cdot \log \pi_\theta(a_t \mid s_t) \Big]\end{aligned} \tag{36}$$

where α_H is the entropy coefficient, which determines the trade-off between exploitation and exploration. Under cosine annealing, α_H is updated according to the following formula:

$$\alpha_H^t = \alpha_H^{end} + \frac{1}{2} \left(\alpha_H^{init} - \alpha_H^{end} \right) \left(1 + \cos \left(\frac{T_t}{T_{\alpha_H}} \pi \right) \right) \tag{37}$$

When the number of parameter updates T_t reaches the predetermined number T_{α_H}, α_H^t becomes equal to α_H^{end} and then remains constant at α_H^{end} to achieve stable training convergence.

Algorithm 1 presents the complete training steps of PA-HRL. The entire training process is divided into two stages: pre-training and joint training. Before training begins, the network parameters of both agents and training hyperparameters are initialized.

In the pre-training stage, the network parameters of the high-level agent are kept fixed, and the high-level agent generates task priority sequences using a random shuffle strategy (Yue et al. 2026) each time. Then, the low-level agent sequentially makes node selection decisions for the tasks in the task priority sequence, and stores its trajectory data in the experience pool D_l. When the number of samples exceeds the memory threshold

C_{th}, the low-level agent draws samples from D_l to update its policy and value networks. After each round is finished, the high-level agent receives the reward feedback and stores its trajectory data in the experience pool D_h.

If the reward of the low-level agent remains stable for 20 consecutive iterations, then the joint training stage starts. At this stage, the high-level agent uses its policy network to obtain the task priority sequence, and then the low-level agent makes node selection decisions. Both the high-level and low-level agents collect trajectories, and they separately draw samples to update their network parameters. Meanwhile, during parameter updates, the gradient norms of both agents are clipped to 10 to avoid gradient explosion.

After training, the policy networks of the high-level and low-level agents can be deployed to the controller for runtime task scheduling.

4.4 Complexity analysis

In this subsection, we analyze the convergence stability of PA-HRL as well as the time complexity of the training and inference phases.

(1) Convergence stability

Traditional single-layer agent methods jointly determine the combined action of task ordering and node selection. Within a single time step, for at most L_s scheduling tasks and N nodes, the action combination space dimension is $L_s \cdot N$. In contrast, PA-HRL decouples the task ordering and node selection actions through hierarchical decomposition, where the action dimension of the high-level agent is L_s, and that of the low-level agent for node selection is N. This decomposition reduces each agent's action space dimension. Therefore, compared with single-layer DRL methods, PA-HRL achieves a controllable action space dimension and superior stability.

(2) Training time complexity

During the training process of PA-HRL, the policy networks and value networks of both agents need to be updated separately.

For the high-level agent, the self-attention of its encoder needs to compute the query, key, and value vectors as well as the attention scores for each task. Within a time step, for at most L_s scheduling tasks, the resulting time complexity is $O(L_s^2 d_h)$, where d_h is the dimension of the hidden layer of the encoder. The decoder obtains action probabilities through the pointer network, and the time complexity is $O(L_s^2 d_h)$. The high-level agent uses only the policy network during data collection and its time complexity within a time step is $O(2L_s^2 d_h)$.

For the low-level agent, its self-attention computes attention for N nodes with a time complexity of $O(N^2 d_l)$, where d_l is the node embedding dimension. The output is then forwarded through fully connected layers, with a time complexity of $O\left((d_z + Nd_l)H_1 + H_1H_2 + NH_2\right)$, where H_1 and H_2 are the dimensions of the intermediate hidden layers and $(d_z + Nd_l)$ represents the dimension of the fused features. The low-level agent also uses only the policy network during data collection, and its time complexity within a time step is $O\big(L_s \cdot (N^2 d_l + (d_z + Nd_l)H_1 + H_1H_2 + NH_2)\big)$.

The network parameter updating involves forward and backward propagation, where the backward propagation process requires computing the loss and gradients of each layer. During the pre-training stage only the low-level agent is updated, and the time complexity is:

$$\begin{aligned} O_1 = O\Big(E_p \cdot T \cdot \Big((L_s + 3\mathcal{B}) \\ \cdot \big(N^2 d_l + (d_z + Nd_l)H_1 \\ + H_1H_2 + NH_2\big) + 6\mathcal{B}N^2 d_l\Big)\Big) \end{aligned} \tag{38}$$

During the joint training stage, both the high-level and low-level agents need to be updated at the end of each time step, and the time complexity is

$$\begin{aligned} O_2 = O\Big(E \cdot T \cdot \Big((L_s + 3\mathcal{B}) \\ \cdot \big(N^2 d_l + (d_z + Nd_l)H_1 \\ + H_1H_2 + NH_2\big) \\ + 6\mathcal{B}N^2 d_l\Big) + L_s^2 d_h \cdot (2 + 9\mathcal{B})\Big) \end{aligned} \tag{39}$$

where E_p and E denote the numbers of pre-training and joint-training epochs, T is the number of time steps, and $\mathcal{B}$ is the batch size.

Therefore, the total training time complexity of PA-HRL is

$$\begin{aligned} O_{train} &= O_1 + O_2 \\ &= O\Big(T \cdot \big((E_p + E) \cdot ((L_s + 3\mathcal{B}) \\ &\quad \cdot (N^2 d_l + (d_z + Nd_l)H_1 \\ &\quad + H_1H_2 + NH_2) + 6\mathcal{B}N^2 d_l) \\ &\quad + E \cdot L_s^2 d_h \cdot (2 + 9\mathcal{B}))\Big) \end{aligned} \tag{40}$$

Algorithm 1 PA-HRL Training Method

Require: Initial policy network parameters θ_h, θ_l; critic network parameters ϕ_h, ϕ_l; pre-training episodes E_p; joint training episodes E; time steps per episode T; memory capacity C; memory threshold C_{th}; batch size $\mathcal{B}$

Ensure: Optimized policy parameters θ_h^*, θ_l^*

Phase 1: Pre-training Stage
for $e = 1$ to E_p **do**
  Initialize the environment, obtain initial states s_0^h and s_0^l
  for $t = 1$ to T **do**
    Generate a_t^h through the random shuffle strategy and reorder the scheduling queue
    for each task in the scheduling queue **do**
      Construct low-level state s_t^l, sample action a_t^l, select the optimal node j^*
      Assign task based on resource availability, obtain reward r_t^l
      Store $(s_t^l, a_t^l, r_t^l, s_{t+1}^l)$ in replay buffer D_l
    end for
    Sample a batch from D_l and compute advantage estimates A_t using GAE
    Update θ_l and ϕ_l for the low-level agent
    Store $(s_t^h, a_t^h, r_t^h, s_{t+1}^h)$ in replay buffer D_h
  end for
end for
Phase 2: Joint Training Stage
for $e = 1$ to E **do**
  Initialize the environment, obtain initial states s_0^h and s_0^l
  for $t = 1$ to T **do**
    Generate a_t^h via the policy network of the high-level agent and reorder the scheduling queue
    for each task in the scheduling queue **do**
      Construct low-level state s_t^l, sample action a_t^l, select the optimal node j^*
      Assign task based on resource availability, obtain reward r_t^l
      Store $(s_t^l, a_t^l, r_t^l, s_{t+1}^l)$ in replay buffer D_l
    end for
    Environment steps forward, obtain new state s_{t+1}^h and high-level reward r_t^h
    Store $(s_t^h, a_t^h, r_t^h, s_{t+1}^h)$ in replay buffer D_h
    Sample batches from D_h and D_l, respectively
    Update θ_h and ϕ_h for the high-level agent, and θ_l and ϕ_l for the low-level agent
  end for
end for

(3) Inference time complexity

When using PA-HRL for online inference after training, only the policy networks need to be deployed, without the participation of the value networks. Therefore, within a time step, the inference time complexity of the high-level agent is $O(2L_s^2 d_h)$. Since at most L_s tasks can be consecutively scheduled within a time step, the inference time complexity of the low-level agent in the worst case within a time step is

$$O\big(L_s \cdot ((d_z + Nd_l)H_1 + H_1H_2 + NH_2)\big)$$

Therefore, the total inference time complexity of PA-HRL within a time step is at most

$$\begin{aligned} O_{infer} = O\big(&2L_s^2 d_h \\ &+L_s \cdot \big((d_z + Nd_l)H_1 \\ &+H_1H_2 + NH_2\big)\big) \end{aligned} \tag{41}$$

5 Simulation results and analysis

We conduct simulation experiments under different scenarios and compare PA-HRL with several

Table 3 Resource and pricing parameters for each type of computing node

Index	GPU type	GPU memory (GB)	Perf. (TFLOPS)	Storage (TB)	Band. (Gbps)	Price ($/d)	GPU capacity
1	RTX5090	32	318	30	10	50	2800
2	A100	40	312	50	10	68	2500
3	L40S	48	181	50	8	45	3000
4	H100	80	756	80	15	110	2000
5	H100 SXM	80	989	80	20	150	1800

baseline methods, and then verify the effectiveness of the high-level agent and the contribution of each key component to PA-HRL through ablation experiments.

5.1 Experiment setup

The simulation experiments are conducted on the Ubuntu 22.04 LTS operating system and EdgeSimPy framework (Souza et al. 2023), and the algorithm is implemented in Python 3.12 with PyTorch 2.14. To simplify the CPN topology, we fix the number of routing hops from each regional CPN router to the computing nodes in other regions to a constant and ignore network congestion.

In the simulation scenario, multiple computing nodes are distributed in the same or different city regions, and they differ in GPU type, GPU memory, computational performance, resource capacity, transmission bandwidth, and rental price. We design five types of computing nodes, and Table 3 lists the resource and pricing parameters of each type.

Meanwhile, the unit-time transmission costs under different bandwidth tiers are set as follows: 250 $/h for 8 Gbps, 400 $/h for 10 Gbps, 800 $/h for 15 Gbps, and 1200 $/h for 20 Gbps.

Based on the city vitality indicators in Table 2, we select three city regions in China: Chengdu, Qingdao, and Ningbo. Among them, Chengdu has the lowest city vitality indicator and thus has the smallest cost influence coefficient. Then, 10 computing nodes are randomly distributed across these regions and configured with the parameters in Table 3, while the computational efficiency η of each node is drawn uniformly at random from $[0.5, 0.55]$.

In the simulation, we consider three categories of language models, distinguished by the number of model parameters: small-scale ($(1, 10]$B), medium-scale ($(10, 70]$B), and large-scale (> 70B). The training duration for each LLM training task is generally estimated in advance. Table 4 lists the parameters and typical examples of each category. We also classify LLM training tasks according to the relative sensitivity coefficient w into delay-sensitive ($w = 0.7$ to 0.9), balanced ($w = 0.4$ to 0.6), and cost-sensitive ($w = 0.1$ to 0.3) tasks.

The specific settings of training and network hyperparameters are detailed in Table 5. In the decoder of the high-level agent, the attention weights between the pointer vector and matching vectors are processed by the tanh activation function, multiplied by a constant $C = 10$, and finally normalized by the softmax function. In the low-level agent, the attention output of self-attention is fused with the task embedding features and further passed through two fully connected layers of size 128 and 64 to output node selection probabilities. Agents at both levels use the Adam optimizer. The discount factor of the high-level agent is $\gamma = 0.99$, which emphasizes long-term

Table 4 Parameters and typical examples for each category of language model

Scale	Params (B)	Tokens (B)	Number of hidden layers	Data vol. (GB)	Training duration (days)	Example
Small	1–10	10–100	24–40	40–400	2–8	TinyLlama(1.1B), Gemma(2B), Phi-4-mini(3.8B)
Medium	10–70	100–2000	40–80	400–8000	8–20	GLM 4.7Flash (30B), Llama 2 (70B)
Large	> 70	≥ 2000	80–128	≥ 8000	20–60	GLM 4.5Air (106B), DeepSeek R1 (671B), GPT-4

Table 5 Training and network hyperparameters of PA-HRL

Parameter	High-level	Low-level
Actor learning rate lr_{actor}	1e-3	1e-4
Value network learning rate lr_{critic}	2e-3	2e-4
Discount factor γ	0.99	0.96
Initial entropy coefficient $\varepsilon_{\text{init}}$	0.1	0.1
Final entropy coefficient ε_{end}	0.01	0.01
Memory capacity C	20000	20000
Memory threshold C_{th}	256	256
Batch size $\mathcal{B}$	128	128
Embedding dimension d_{hid}	64	32
The number of attention heads n_{head}	8	8

returns, while that of the low-level agent is $\gamma = 0.96$, which emphasizes immediate feedback.

The training process consists of 200 iterations in total. Before each training iteration, 200 tasks are first randomly generated and randomly deployed to computing nodes as the initial load of the CPN, and subsequent tasks are then generated according to a Poisson distribution with $\lambda = 8$ until 100 additional tasks have been generated in total. After training, the algorithm is tested under different scenarios. Before each test, 200 tasks are still randomly generated and randomly deployed as the initial load, and after the subsequent tasks have been scheduled we collect related metrics to evaluate the performance of the algorithm. To ensure reproducibility, the random seed is fixed at $seed = 4321$ in each test.

For an LLM training task, its transmission delay, measured in hours, is much shorter than its computation delay and waiting delay, which are measured in days. Moreover, the computation delay is usually predetermined by task requests ($T_j^{\text{Comp}} = \tau_j$). Therefore, we neglect the transmission delay and computation delay when evaluating the completion delay of tasks and mainly focus on the waiting delay. The transmission delay is mainly used to calculate the transmission cost (see Eq. (13)), whereas the computation delay is used to determine the number of GPUs required by a training task (see Eq. (10)). Accordingly, we collect four metrics in tests: total scheduling penalty, load variance, total waiting delay, and total cost.

5.2 Comparison baseline

To evaluate the performance of PA-HRL, we compare it with the following baseline methods, all of which use the same state representation and experimental settings.

FCFS-Random: Tasks are scheduled according to the FCFS principle, and nodes are randomly selected during node assignment.

FCFS-GA: Tasks are also scheduled in FCFS order, and a genetic algorithm (GA) heuristic selects the deployment node for each task.

Priority-A2C: The high-level agent is first used to compute task priority weights to sort task sequences, and A2C is then used for node selection.

CPN-HRL: The hierarchical deep reinforcement learning approach proposed by Yue et al. (2026) for priority-aware task scheduling in CPN. In this method, the high-level agent uses an LSTM to model the temporal dependencies of task sequences, while the low-level agent uses a graph attention network (GAT) to extract the adjacency relations among nodes. Both agents adopt proximal policy optimization (PPO).

PA-HRL: The priority-aware hierarchical cooperative reinforcement learning algorithm proposed in this paper. Both agents use self-attention to extract the key features of task sequences or node sequences, and adopt A2C combined with decaying policy entropy and a prioritized experience replay mechanism.

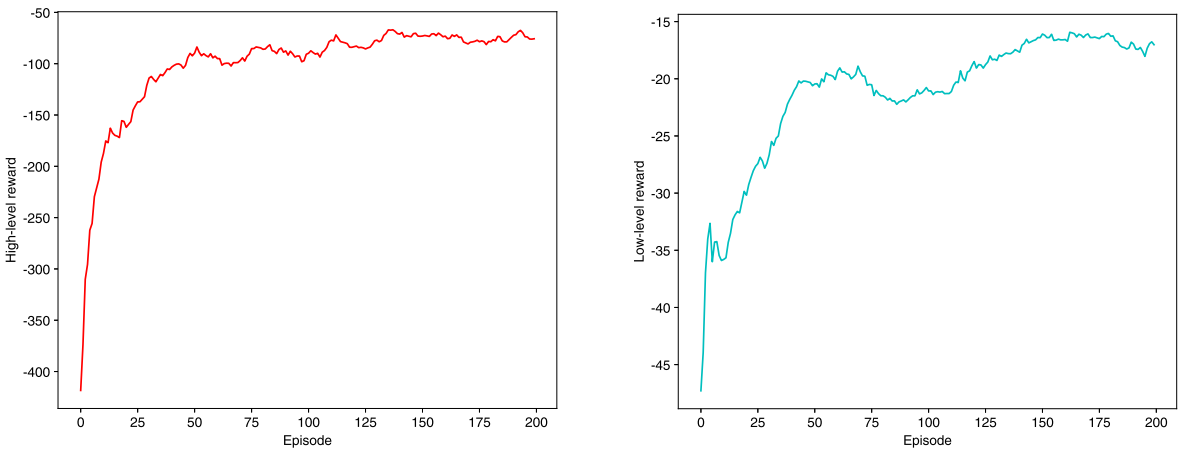
5.3 Results analysis

5.3.1 Convergence analysis

Figure 4 illustrates the training convergence performance of PA-HRL. During the pre-training stage, although the policy of the high-level agent is not updated, its reward still increases because the low-level agent is being optimized. The joint training stage starts after about 50 iterations; at this stage, the reward of the low-level agent first drops slightly due to the update of the high-level agent. However, the rewards of the high-level and low-level agents then gradually increase and converge after about 150 iterations owing to the cross-layer cooperative optimization between the two agents.

5.3.2 Performance comparison under different task arrival rates

To examine the sensitivity of PA-HRL to the task arrival rate, we keep the numbers of training tasks of the three scale categories equal and set w to



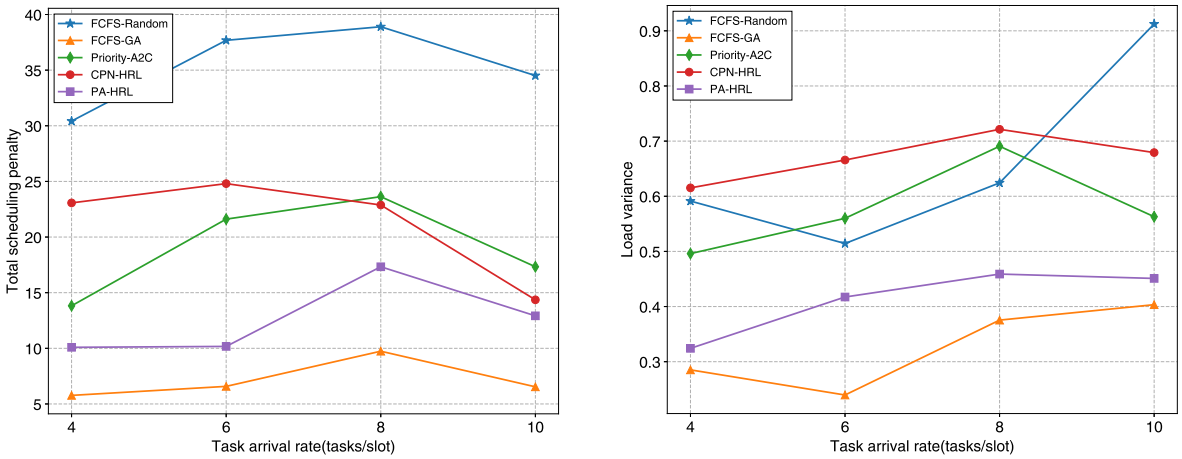
(a) (b)

Fig. 4 Training convergence performance of PA-HRL: (a) high-level reward, (b) low-level reward

a uniformly distributed random value, and then vary the task arrival rate λ and observe how the core metrics change.

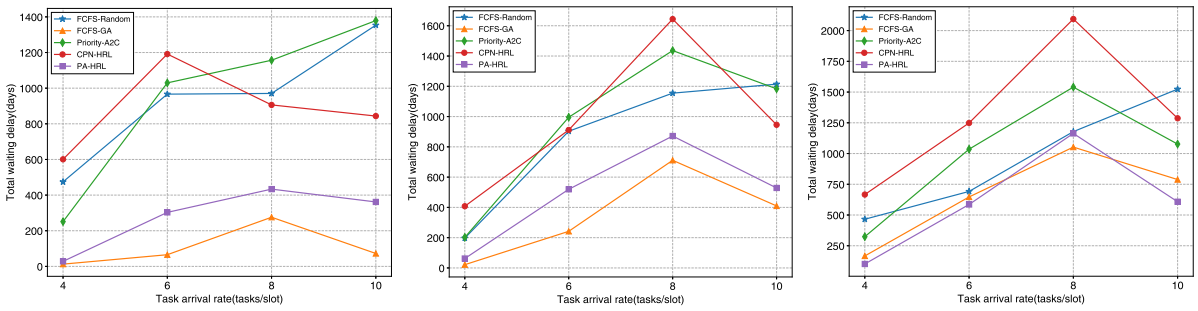
Figure 5(a) and Figure 5(b) show that, as the task arrival rate increases, both the total scheduling penalty and load variance of PA-HRL are always lower than those of other methods except for FCFS-GA, which indicates that PA-HRL can always achieve high task satisfaction and keep the balance of resource utilization among nodes under different task arrival rates.

Figure 6(a), (b), and (c) respectively show the total waiting delay of each sensitivity type under different task arrival rates. The total waiting delay of all sensitivity types generally increases as λ grows from 4 to 8; at $\lambda = 10$, however, it decreases for all methods except for FCFS-Random and Priority-A2C. This is because at most 10 tasks can be scheduled within a single time step ($L_s = 10$). When $\lambda = 10$, the number of task requests within a time step exceeds the scheduling queue capacity. In this case, the excess tasks remain in the cache queue. Before the next scheduling round, some of the tasks being executed on the computing nodes have finished and released their resources, so the newly admitted tasks are likely to be placed on nodes with more abundant resources, which reduces their waiting delay. FCFS-GA and PA-HRL consistently achieve shorter total waiting



(a) (b)

Fig. 5 Total scheduling penalty and load variance under different arrival rates: (a) total scheduling penalty, (b) load variance



(a) (b) (c)

Fig. 6 Total waiting delay for each sensitivity type under different arrival rates: (a) delay-sensitive, (b) balanced, (c) cost-sensitive

delays under different task arrival rates, and PA-HRL exhibits the shortest waiting delay for the cost-sensitive tasks at the arrival rates of 4, 6, and 10, with total waiting delays of about 102, 586, and 608 days, respectively.

Figure 7(a), (b), and (c) respectively present the total cost of each sensitivity type under different task arrival rates. The experimental results show that as λ increases, Priority-A2C and CPN-HRL obtain the lowest cost at most cases; whereas, as Figure 6 illustrates, their delay performances are poor. This indicates that they overemphasize cost and fail to focus on task priority and node load, leading to task queueing congestion. Figure 7(c) demonstrates that PA-HRL yields a cost similar to those of the other non-random methods at the arrival rates of 4, 6, and 8.

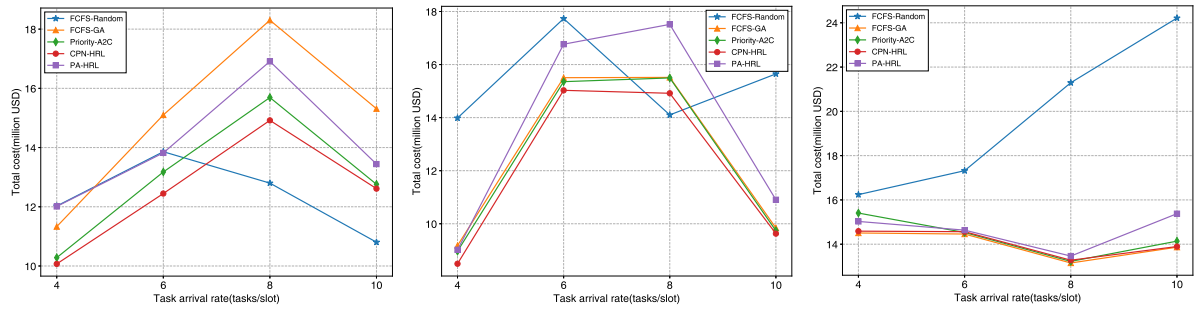
Table 6 reports the total decision time of each method under different task arrival rates. As λ grows, the total decision time of each method also increases. This is because the number of task requests within a time step rises as λ increases, so that each method needs to make decisions for more tasks. FCFS-Random always takes the shortest decision time, while FCFS-GA, which requires multiple rounds of iterative search for the optimal solution, always incurs the longest. The total decision time of DRL-based methods increases slowly and is always less than 200 ms.

Table 6 Total decision time (ms) under different task arrival rates

Method	$\lambda = 4$	$\lambda = 6$	$\lambda = 8$	$\lambda = 10$
FCFS-Random	81	130	130	137
FCFS-GA	497	855	1005	1215
Priority-A2C	98	146	153	163
CPN-HRL	103	153	157	164
PA-HRL	104	148	155	164

5.3.3 Performance comparison under different categories of language model

To validate the applicability of PA-HRL across language models of varying scales, we conduct experiments under three scenarios of different categories of language model: small-, medium-, and



(a) (b) (c)

Fig. 7 Total cost for each sensitivity type under different arrival rates: (a) delay-sensitive, (b) balanced, (c) cost-sensitive

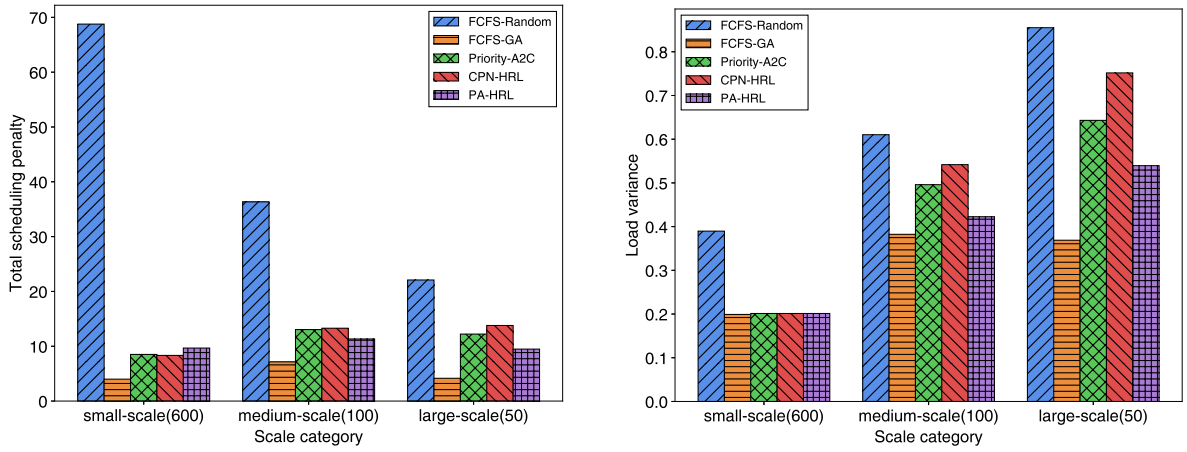


Fig. 8 Total scheduling penalty and load variance for each category of language model: (a) total scheduling penalty, (b) total load variance

large-scale language models. In the experiments, 600 small-scale, 100 medium-scale, and 50 large-scale language model training tasks are generated, with the task arrival rate fixed at $\lambda = 8$ and w drawn uniformly at random.

Figure 8(a) and Figure 8(b) respectively illustrate the total scheduling penalty and the load variance of each method under different scales. It can be observed from the experimental results that the total scheduling penalty and load variance of FCFS-GA and all the DRL-based methods are always lower than those of FCFS-Random. In addition, PA-HRL has a slightly higher task scheduling penalty than Priority-A2C and CPN-HRL in the small-scale scenario, while it obtains lower task scheduling penalty and load variance than other methods except for FCFS-GA in the medium- and large-scale scenarios.

As shown in Figure 9(a), (b), and (c), CPN-HRL and PA-HRL always achieve shorter total waiting delays in the scenario with only small-scale tasks. However, CPN-HRL has a longer total waiting delay in the medium- and large-scale scenarios, and it incurs the longest waiting delay for the delay-sensitive and cost-sensitive tasks in the large-scale scenario. Moreover, Priority-A2C always suffers a longer waiting delay than FCFS-GA and PA-HRL. In contrast, PA-HRL always achieves the shortest total waiting delay in the small-scale scenario, and it also exhibits a shorter waiting delay than all methods except FCFS-GA in the medium- and large-scale scenarios. Compared with FCFS-Random, Priority-A2C, and CPN-HRL, the total waiting delay of PA-HRL is at most reduced by 98.7%, 84.5%, and 8.3% (aggregated across different sensitivity types) in the small-scale scenario, by 66.7%,

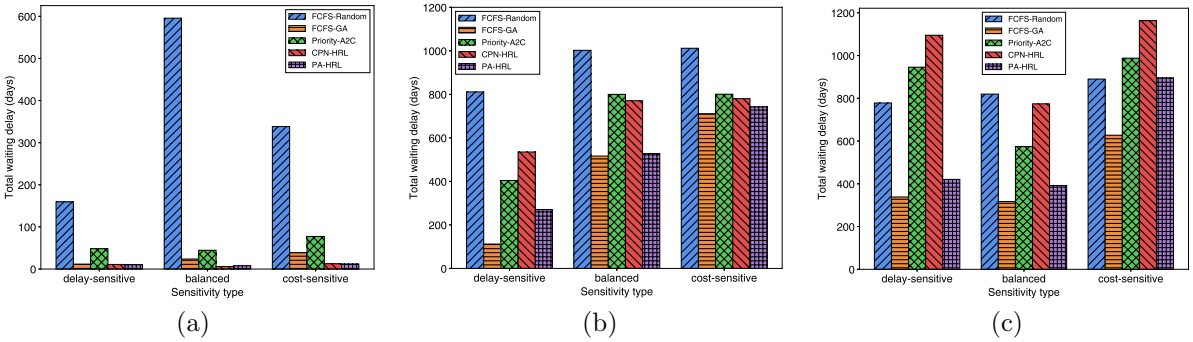
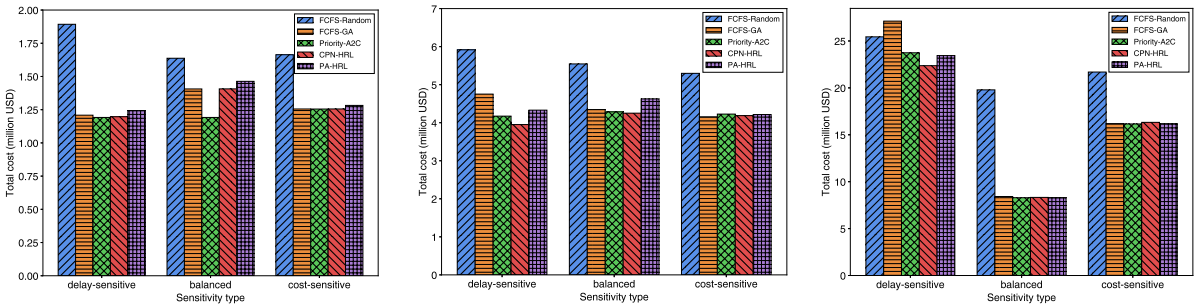


Fig. 9 Total waiting delay for each category of language model: (a) small-scale (600 tasks), (b) medium-scale (100 tasks), (c) large-scale (50 tasks)



(a) (b) (c)

Fig. 10 Total cost for each category of language model: (a) small-scale (600 tasks), (b) medium-scale (100 tasks), (c) large-scale (50 tasks)

34.0%, and 55.5% in the medium-scale scenario, and by 52.2%, 55.5%, and 61.6% in the large-scale scenario, respectively.

As shown in Figure 10(a), (b), and (c), FCFS-GA and DRL-based methods exhibit comparable total costs for the cost-sensitive tasks across all categories of scale, and their costs are significantly lower than that of FCFS-Random. PA-HRL achieves the lowest total cost for the balanced tasks in the large-scale scenario; specifically, its total cost is 58.2% lower than that of FCFS-Random and up to 1.7% lower than those of the other non-random methods. For the cost-sensitive tasks, although PA-HRL does not obtain the lowest cost, its cost differs from that of the least-costly method by only 2.2%, 1.5%, and 0.1% in the small-, medium-, and large-scale scenarios, respectively. This demonstrates that PA-HRL also exhibits effective cost optimization for the cost-sensitive tasks.

Table 7 presents the total decision time of each method under different scales. In the scenario with 600 small-scale tasks, FCFS-Random has a slightly longer total decision time than the DRL-based methods; by contrast, it achieves the shortest decision time in the medium- and large-scale scenarios. The total decision time of PA-HRL is marginally longer than those of Priority-A2C and CPN-HRL, while being reduced by up to 86.2% compared with FCFS-GA across all scenarios.

5.3.4 Performance comparison under different proportions of delay-sensitive tasks

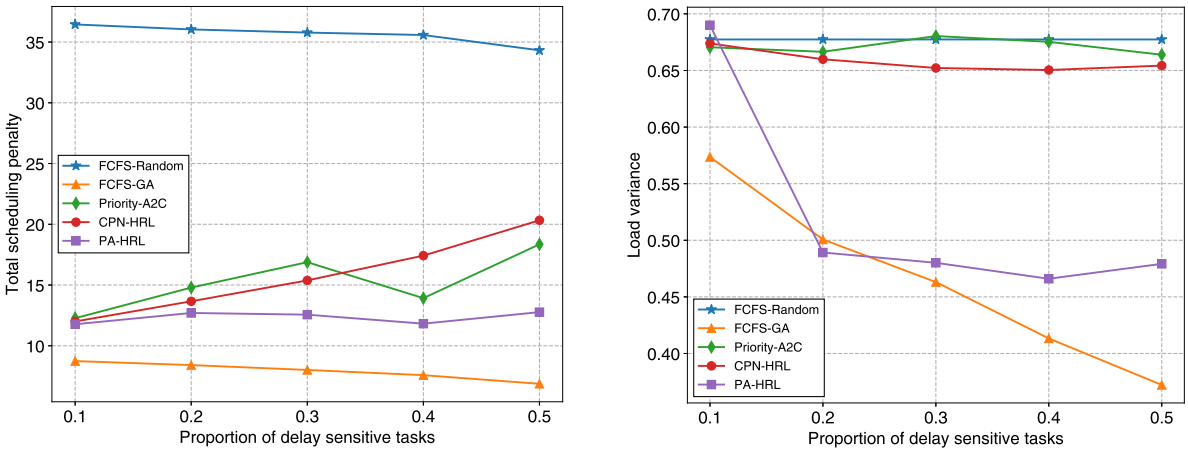
To test the robustness of PA-HRL under varying proportions of delay-sensitive tasks, we restrict task generation to only delay-sensitive and cost-sensitive tasks, and vary the proportion of delay-sensitive tasks, with the task arrival rate fixed at $\lambda = 8$ and the numbers of tasks of the three scale categories kept equal.

Table 7 Total decision time (ms) for each category of language models

Method	Small-scale (600 tasks)	Medium-scale (100 tasks)	Large-scale (50 tasks)
FCFS-Random	6.30	1.32	0.59
FCFS-GA	26.49	11.45	4.29
Priority-A2C	5.16	1.44	0.74
CPN-HRL	5.35	1.56	0.80
PA-HRL	5.37	1.58	0.81

Figure 11(a) illustrates that, as the proportion of delay-sensitive tasks increases, the total scheduling penalty of PA-HRL remains stable and is always lower than those of other methods except FCFS-GA. Figure 11(b) demonstrates that the load variance of FCFS-Random remains constant due to its fixed strategy, while those of Priority-A2C and CPN-HRL always fluctuates between 0.65 and 0.70; only the load variances of FCFS-GA and PA-HRL gradually decrease. This indicates that both FCFS-GA and PA-HRL can effectively balance the resource load across computing nodes.

As shown in Figure 12(a) and (b), the total waiting delays and total costs of the delay-sensitive tasks across all methods increase as the proportion rises. Priority-A2C and CPN-HRL always achieve the lowest total cost, whereas their total waiting delays are even longer than that of FCFS-Random. In contrast, PA-HRL obtains a total cost second only to those of Priority-A2C and CPN-HRL, and its total waiting delay is at most reduced by 80.8% compared with FCFS-Random,



(a) (b)

Fig. 11 Total scheduling penalty and load variance under different proportions of delay-sensitive tasks: (a) total scheduling penalty, (b) load variance

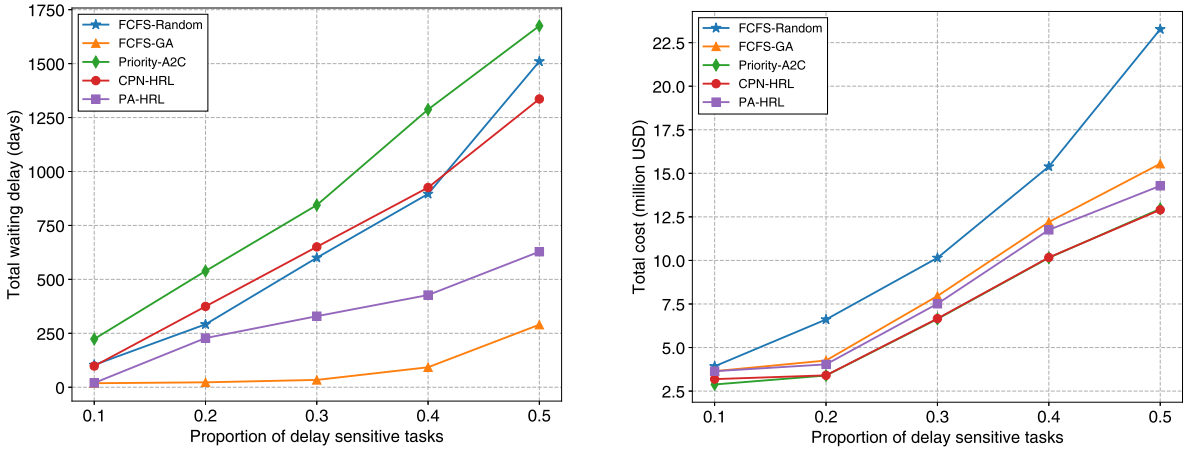
by 91.1% compared with Priority-A2C, and by 79.8% compared with CPN-HRL.

As presented in Figure 13(a) and (b), the total waiting delays and total costs of the cost-sensitive tasks across all methods decrease as the proportion of delay-sensitive tasks rises. Priority-A2C and CPN-HRL still achieve the lowest total cost but incur a longer waiting delay than PA-HRL. The total waiting delay of PA-HRL is still reduced at most by 27.5% compared with FCFS-Random, by 31.1% compared with Priority-A2C, and by 36.1% compared with CPN-HRL, respectively, while remaining close to that of FCFS-GA. This demonstrates that PA-HRL also exhibits excellent delay optimization performance for the cost-sensitive tasks.

Table 8 Total decision time of each method

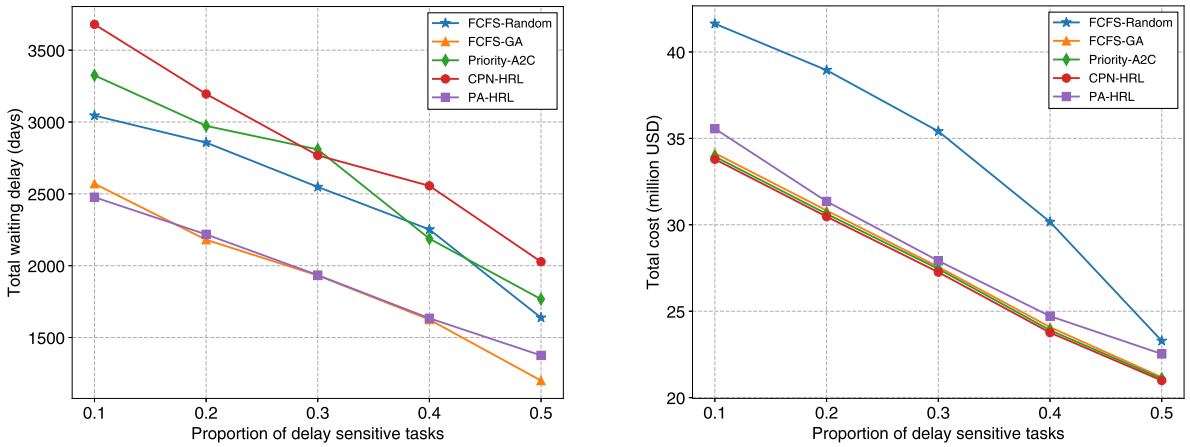
Method	Total decision time(ms)
FCFS-Random	1.28
FCFS-GA	10.16
Priority-A2C	1.54
CPN-HRL	1.54
PA-HRL	1.55

Table 8 shows the total decision time of each method. Among all the methods, FCFS-Random



(a) (b)

Fig. 12 Total waiting delay and total cost of the delay-sensitive tasks under different proportions: (a) total waiting delay, (b) total cost



(a) (b)

Fig. 13 Total waiting delay and total cost of the cost-sensitive tasks under different proportions: (a) total waiting delay, (b) total cost

always takes the shortest total decision time, followed by the DRL-based methods, while FCFS-GA still incurs the longest decision time with a pronounced gap relative to the others.

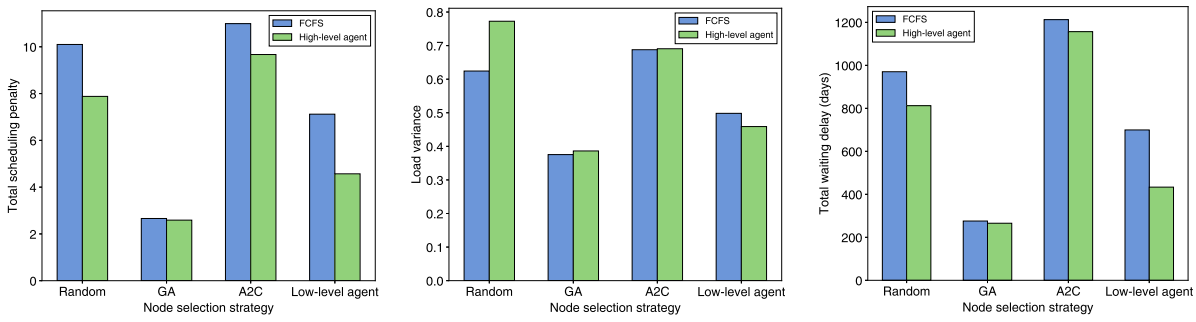
5.3.5 Ablation experiments

To verify the effectiveness of the high-level agent and the contribution of each key component to PA-HRL, we design ablation experiments in this subsection. In the experiments, the task arrival rate is kept at $\lambda = 8$, tasks of the three scale categories are generated in equal proportions, and w is drawn at random, while all other hyperparameters are fixed.

Effectiveness of the high-level agent

First, we conduct the experiment to verify the effectiveness of the high-level agent. We extract the high-level agent of PA-HRL and combine it with different node selection strategies.

Figure 14(a) and (c) demonstrate that the high-level agent reduces the total scheduling penalty and the total waiting delay of the delay-sensitive tasks by 22.0%, 2.6%, 12.0%, and 35.8% in scheduling penalty and 16.3%, 3.7%, 4.6%, and 38.0% in delay, when combined with random, GA, A2C, and the low-level agent of PA-HRL, respectively. This indicates the effectiveness of the high-level agent when combined with different node selection strategies. However, it can be observed from Figure 14(b) that when the high-level agent of PA-HRL is combined with other node selection strategies except for the low-level agent, their load variances do not improve markedly, which indicates that the load balancing is mainly determined



(a) (b) (c)

Fig. 14 Effect of the high-level agent on scheduling performance: (a) total scheduling penalty of the delay-sensitive tasks, (b) load variance, (c) total waiting delay of the delay-sensitive tasks

by the node selection strategy and only weakly related to the task priority ordering method.

Contribution of each key component to PA-HRL

To verify the contribution of each key component to PA-HRL, we implement several variants by removing or replacing individual key components of PA-HRL. Specifically, the following three variants are implemented:

PA-HRL-w/o-High-level: Only the low-level agent of PA-HRL is retained, without the high-level agent for task priority ordering. This variant aims to verify the contribution of the high-level agent.

PA-HRL-w/o-Self-Attention: the self-attention is removed from both the high-level and low-level agents. This variant aims to verify the contribution of the self-attention.

PA-HRL-w/o-Pointer: The pointer network in the high-level agent is replaced with a fully connected network. This variant aims to verify the advantage of the pointer network for sequence ordering.

Figure 15(a), (b), (c) and (d) present the corresponding core metrics of different variants. Based on the experimental results, we can observe that each component of PA-HRL has an influence on the performance of our algorithm. Removing the high-level agent (PA-HRL w/o High-level) results in the highest task scheduling penalty, i.e., the lowest task satisfaction, and the total waiting delays of PA-HRL w/o High-level for the delay-sensitive and cost-sensitive tasks are significantly

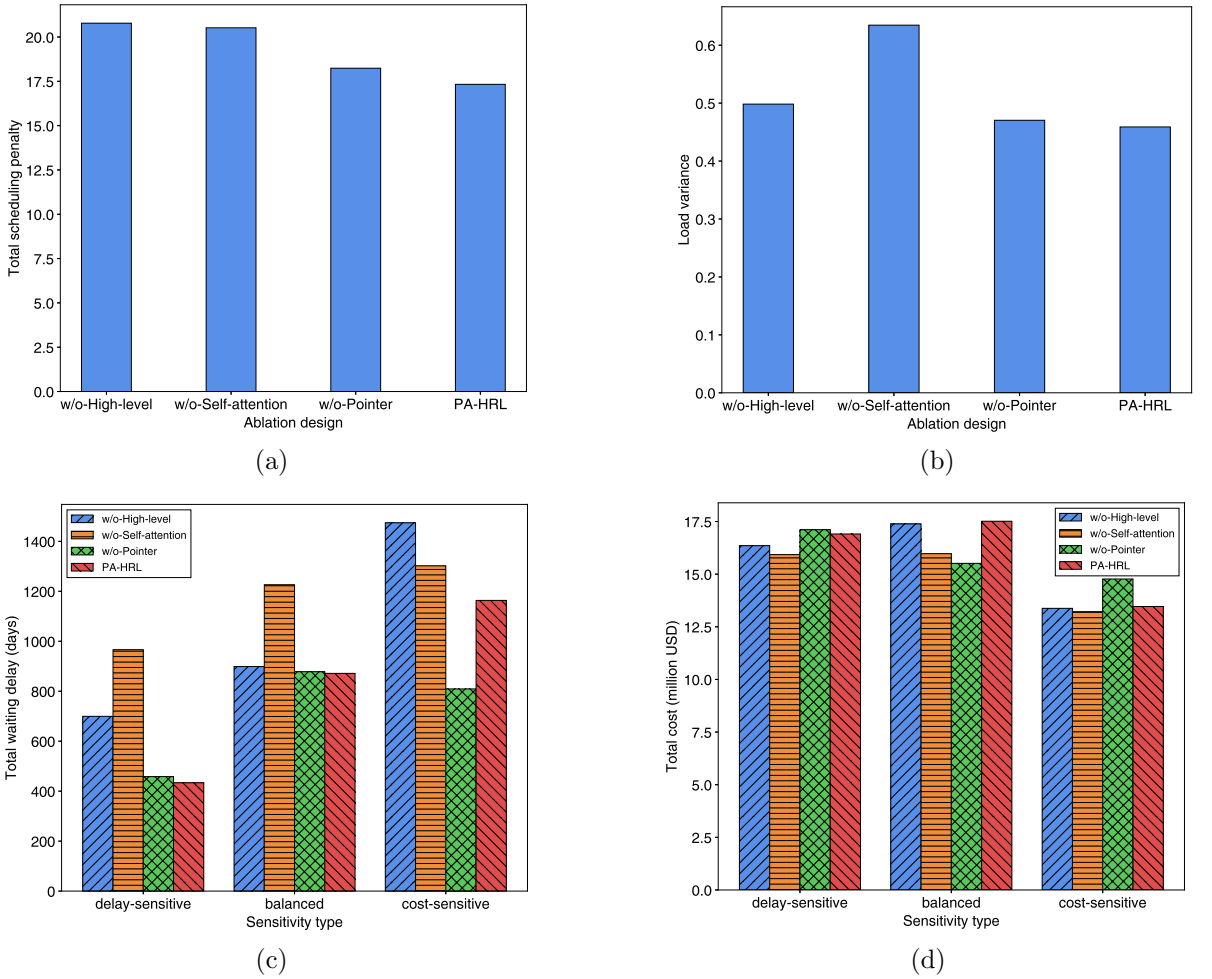


Fig. 15 The performances of different variants: (a) total scheduling penalty, (b) load variance, (c) total waiting delay, (d) total cost

longer than those of PA-HRL, which indicates that the high-level agent of PA-HRL plays an important role in task priority ordering. The absence of self-attention (PA-HRL w/o Self-Attention) leads to a higher task scheduling penalty, the largest load variance, and longer waiting delays, which proves the ability of self-attention to extract the key features of task workloads and node load, thereby improving the load balancing ability of our algorithm. The replacement of the pointer network (PA-HRL w/o Pointer) results in slightly higher task scheduling penalty and load variance than those of PA-HRL. It incurs slightly longer total waiting delays for the delay-sensitive and balanced tasks compared with PA-HRL, and yields higher costs than other variants for the delay-sensitive and cost-sensitive tasks. These demonstrate that the pointer network also has a certain optimization effect on task satisfaction and load balancing.

6 Conclusion

This paper proposes a priority-aware hierarchical cooperative reinforcement learning algorithm, PA-HRL, which decouples task priority ordering from node selection through multi-agent hierarchical cooperation to jointly optimize task satisfaction and load balancing in CPN environments. The simulation results show that PA-HRL achieves better task satisfaction and load balancing than other hierarchical cooperative DRL methods and FCFS-Random under different scenarios. Specifically, it obtains shorter waiting delays for delay-sensitive and balanced tasks, while for cost-sensitive tasks it achieves a lower load variance and a near-minimal cost. Furthermore, its decision time is always shorter than that of FCFS-GA. Finally, the effectiveness of the high-level agent and the contribution of each key component to PA-HRL are verified through ablation experiments.

Future research can extend this work in several directions:

(1) Evolution toward a distributed control architecture: Extending our algorithm to operate in a distributed architecture to avoid scheduling failures caused by damage to the centralized controller.

(2) Consideration of network topology: Incorporating network topology into the system model to enhance the topology awareness of our algorithm.

(3) Real-world deployment validation: Deploying our algorithm in real CPN environments and conducting experiments under actual business scenarios.

References

Alsalem L, Djemame K (2025) Task Scheduling in Edge Computing Environments: a Hierarchical Cluster-based Federated Deep Reinforcement Learning Approach. In: Proceedings of the 18th IEEE/ACM International Conference on Utility and Cloud Computing, pp 1–8

Chen J, Zhong J, Wu Z, et al (2025) Priority-aware task offloading for LEO satellite edge computing network: a multi-agent deep reinforcement learning-based approach. Journal of King Saud University Computer and Information Sciences 37(7):168

Cui D, Peng Z, Li K, et al (2025) A novel cloud task scheduling framework using hierarchical deep reinforcement learning for cloud computing. PLoS One 20(8):e0329669

Devi RR, Fallah M, Aich M, et al (2025) Genetic Algorithm based Task Scheduling in High-Performance Cluster Computing. In: 2025 4th International Conference on Distributed Computing and Electrical Circuits and Electronics (ICDCECE), IEEE, pp 1–6

Dong C, Shafiq M, Al Dabel MM, et al (2023) DNN inference acceleration for smart devices in industry 5.0 by decentralized deep reinforcement learning. IEEE Transactions on Consumer Electronics 70(1):1519–1530

Fedus W, Zoph B, Shazeer N (2022) Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. Journal of Machine Learning Research 23(120):1–39

Feng L, Xie R, Tang Q, et al (2024) Delay-prioritized task scheduling with load balancing in computing power networks. In: 2024 IEEE

Wireless Communications and Networking Conference (WCNC), IEEE, pp 1–6

Han S, Pool J, Tran J, et al (2015) Learning both weights and connections for efficient neural network. Advances in neural information processing systems 28

Hao J, Jin M, Bai H, et al (2024) Heterogeneous resource scheduling in computing power networks: A method based on improved D3QN. In: 2024 IEEE 6th Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC), IEEE, pp 1561–1567

He Q, Wei W, Zheng K, et al (2025) Attention Mechanism-Based Knowledge-Defined Networking for Emergency Task Scheduling in Mobile Edge Computing. In: 2025 IEEE 25th International Conference on Communication Technology (ICCT), IEEE, pp 1817–1822

Hindman B, Konwinski A, Zaharia M, et al (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)

Kushala R, Talasila L, Meghana M, et al (2025) A Novel Task Offloading and Scheduling Framework for Fog Computing. In: 2025 International Conference on Wireless Communications Signal Processing and Networking (WiSPNET), IEEE, pp 1–6

Li P, Xiao Z, Wang X, et al (2023) EPtask: Deep reinforcement learning based energy-efficient and priority-aware task scheduling for dynamic vehicular edge computing. IEEE Transactions on Intelligent Vehicles 9(1):1830–1846

Lin H, Yan M, Ye X, et al (2023) A comprehensive survey on distributed training of graph neural networks. Proceedings of the IEEE 111(12):1572–1606

Liu C, Liu W (2025) A Dual-Agent Deep Reinforcement Learning-Based Algorithm for Cloud-Edge Collaborative Task Offloading in Industrial IoT (IIoT). In: 2025 7th International Conference on Artificial Intelligence Technologies and Applications (ICAITA), IEEE, pp 370–375

Ma Z, Pan C, Zhang Y (2025) Transformer-Enhanced Resource Allocation for Delay–Energy Tradeoff in Cell-Free Mobile Edge Computing. In: 2025 11th International Conference on Computer and Communications (ICCC), IEEE, pp 1663–1667

Narayanan D, Shoeybi M, Casper J, et al (2021) Efficient large-scale language model training on GPU clusters using Megatron-LM. In: Proceedings of the international conference for high performance computing, networking, storage and analysis, pp 1–15

Nath S, Wu J (2020) Dynamic computation offloading and resource allocation for multi-user mobile edge computing. In: GLOBECOM 2020-2020 IEEE global communications conference, IEEE, pp 1–6

Pang S, Li W, He H, et al (2019) An EDA-GA hybrid algorithm for multi-objective task scheduling in cloud computing. IEEE Access 7:146379–146389

Ruan C, Chen Y, Tian D, et al (2026) Libra: Flexible Request Partitioning and Scheduling for Serving Unbalanced and Dynamic LLM Workloads. In: 23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26), pp 1243–1258

Schulman J, Moritz P, Levine S, et al (2015) High-dimensional continuous control using generalized advantage estimation. arXiv preprint arXiv:1506.02438

Schwarzkopf M, Konwinski A, Abd-El-Malek M, et al (2013) Omega: flexible, scalable schedulers for large compute clusters. In: Proceedings of the 8th ACM European Conference on Computer Systems, pp 351–364

Sheng S, Chen P, Chen Z, et al (2021) Deep reinforcement learning-based task scheduling in IoT edge computing. Sensors 21(5):1666

Souza PS, Ferreto T, Calheiros RN (2023) Edgesimpy: Python-based modeling and simulation of edge computing resource management policies.

Future Gener Comput Syst 148(C):446–459. https://doi.org/10.1016/j.future.2023.06.013

Team K, Bai T, Bai Y, et al (2026) Kimi K3: Open frontier intelligence. arXiv preprint arXiv:2607.24653

Verma A, Pedrosa L, Korupolu M, et al (2015) Large-scale cluster management at Google with Borg. In: Proceedings of the tenth european conference on computer systems, pp 1–17

Vinyals O, Fortunato M, Jaitly N (2015) Pointer networks. Advances in neural information processing systems 28

Wang M (2025) AutoHPCNet: A Deep Reinforcement Learning Framework for Adaptive Resource Scheduling in High-Performance Computing. In: 2025 5th International Conference on Artificial Intelligence, Automation and High Performance Computing (AIAHPC), IEEE, pp 842–845

Wang Q, Mao Y, Wang Y, et al (2019) Computation tasks offloading scheme based on multi-cloudlet collaboration for edge computing. In: 2019 Seventh International Conference on Advanced Cloud and Big Data (CBD), IEEE, pp 339–344

Xiang N, Huang Y, Xu X, et al (2020) Regional Economic Vitality Evaluation Based on Open Statistic Data. In: 2020 6th International Conference on Big Data and Information Analytics (BigDIA), IEEE, pp 146–153

Xiang W, Li Y, Ren Y, et al (2023) Gödel: Unified large-scale resource management and scheduling at ByteDance. In: Proceedings of the 2023 ACM Symposium on Cloud Computing, pp 308–323

Xie R, Zhu H, Tang Q, et al (2022) Joint task scheduling and load balancing in computing power network-enabled edge computing systems. In: 2022 IEEE 8th International Conference on Computer and Communications (ICCC), IEEE, pp 563–568

Xie R, Feng L, Tang Q, et al (2025) Priority-aware task scheduling in computing power network-enabled edge computing systems. IEEE Transactions on Network Science and Engineering 12(4):3191–3205

Yang J, Jiang B, Lv Z, et al (2020) A task scheduling algorithm considering game theory designed for energy management in cloud computing. Future Generation computer systems 105:985–992

Yue Q, Tian L, Feng X, et al (2026) CPN-HRL: A hierarchical deep reinforcement learning approach for priority–aware task scheduling in CPN enabled by cloud–edge–end environments. J King Saud Univ Comput Inf Sci pp 1–20

Yukun S, Bo L, Junlin L, et al (2024) Computing power network: A survey. China Communications 21(9):109–145

Zhang H, Tian Z, Zeng L, et al (2025a) Distributed multiagent reinforcement learning approach for multiserver multiuser task offloading. IEEE Internet of Things Journal 12(18):37836–37852

Zhang P, Zeng G, Wang T, et al (2024) Tinyllama: An open-source small language model. arXiv preprint arXiv:2401.02385

Zhang Y, Chen Y, Mo X, et al (2025b) A predictive and synergistic two-layer scheduling framework for LLM serving. arXiv preprint arXiv:2509.23384

Zhu J, Chen H, Zhang J, et al (2026) HiRL: Hierarchical Reinforcement Learning for Coordinated Resource Management in Heterogeneous Edge Computing. arXiv preprint arXiv:2605.10443

Statements and Declarations

Funding
Competing Interests
Author Contributions
Data Availability