

KMPS: A Reinforcement Learning Scheduler for *Kubernetes* Edge-Cloud Systems

Congyue Huang, Wei Tan, Miaohua Ou, Erfu Yang, *Senior Member, IEEE*, and Yun Li, *Fellow, IEEE*

Abstract—Kubernetes (K8s) provides the foundation for integrating distributed edge-cloud resources. However, existing frameworks struggle to address the challenges of cross-cluster coordination and dynamic resource changes, limiting throughput. We propose KMPS, a deep reinforcement learning-based scheduling framework to enhance long-term throughput. KMPS integrates a multi-agent proximal policy optimization algorithm for autonomous edge access point scheduling, combined with gRPC cross-cluster scheduling and invalid target filtering; utilizes graph neural networks to embed system state information, decomposing high-dimensional service orchestration actions through multiple separate policy networks; and constructs a three-time-scale coordination mechanism (0.25s, 2s, 25s) to coordinate scheduling and orchestration, with K8s compatibility. Experiments on real workloads verify that KMPS operates stably under dynamic loads, sudden emergency tasks, and multi-cluster scenarios. Compared to baselines, the proposed framework achieves an over 5.3% increase in long-term throughput and a 60% reduction in cross-cluster scheduling latency.

Index Terms—Edge computing, Kubernetes, reinforcement learning, multi-agent proximal policy optimization

I. INTRODUCTION

A. Background and Problem Statement

THE Internet of Things (IoT) and 5G/6G technologies have triggered an explosive growth in terminal requests, imposing an urgent demand for low-latency responses and effective utilisation of resources. The edge-cloud collaborative architecture (i.e., end-edge-cloud architecture) has been demonstrated to achieve responsive processing and load balancing by deploying services at the edge for proximity or leveraging cloud computing power. Kubernetes (K8s) is widely leveraged in current edge-cloud networks [1]; however, native K8s and its derivatives (e.g., KubeEdge [2], OpenYurt[3]) have inherent limitations, namely the absence of cross-cluster resource coordination capabilities. Consequently,

This work was supported in part by the National Natural Science Foundation of China under Grants W2412112 and W2431044; and in part by the Science and Technology Planning Project of Guangdong Province, China under Grants 2015A010103022, 2014A010103039, and 2014B090901002. (Corresponding author: Wei Tan.)

Congyue Huang, Wei Tan, and Miaohua Ou are with the Dongguan University of Technology, Dongguan 523808, China (email: hcyzlj0713@163.com, tanwphmr@163.com, enmily1347515782@gmail.com).

Erfu Yang is with Department of Design, Manufacturing and Engineering Management, University of Strathclyde, Glasgow G1 1XJ, U.K. (e-mail: erfu.yang@strath.ac.uk).

Yun Li is with the Shenzhen Institute for Advanced Study, University of Electronic Science and Technology of China, Shenzhen 518110, China, and also with i4AI Ltd., London WC1N 3AX, U.K. (email: yun.li@ieec.org).

these systems are incapable of dynamically offloading tasks to neighboring clusters when local resources are inadequate, resulting in tasks being scheduled to the cloud and incurring an additional 100–200 ms of latency [4]. Furthermore, default scheduling policies fail to account for dynamic characteristics such as node performance heterogeneity and load fluctuations. This results in an average CPU utilization below 60% in heterogeneous scheduling scenarios (where low-performance nodes are overloaded while high-performance nodes remain idle), thereby limiting the system's long-term throughput [5].

Existing scheduling solutions rely on the precise modelling of service response time, network latency, or request patterns [6,7], yet they face critical challenges. Edge and cloud resources, in conjunction with request patterns, exhibit substantial randomness, resulting in nonlinear system behaviors. The centralized scheduling approach (e.g., Firmament [8]) is predicated on the aggregation of global state variables. This methodology causes high scheduling latency in large-scale clusters, especially for latency-sensitive services. Moreover, the majority of existing algorithms are designed on the basis of simulated scenarios and lack deployment compatibility with K8s. Consequently, there is an urgent requirement for a learning-based scheduling framework oriented to K8s, capable of adapting to dynamic environments, supporting cross-cluster collaboration, and enabling distributed decision-making.

B. Technical Challenges and Solutions

Learning-based methods can enhance system efficiency by autonomously learning optimal policies. To address this, we propose KMPS, a deep reinforcement learning (DRL)-based scheduling framework for K8s, which autonomously learns from system operation experience without relying on assumptions about environmental dynamics. We address these challenges through three core solutions. First, we design a distributed multi-agent reinforcement learning mechanism for request scheduling. This approach integrates cross-cluster communication and dynamic resource awareness to effectively address the parallel decision-making and action space explosion problems that plague traditional centralized methods (e.g., DQN [9], DDPG [10]) [11]. Second, to address the complexity of service orchestration, Graph neural networks (GNNs) [12] and a separate actor network are introduced to reduce the decision complexity of high-dimensional states (topology, resources, queues) and action spaces (service deployment, replica control). Finally, we propose a time-series adaptive hybrid dequeue strategy that dynamically prioritizes tasks to balance scheduling efficiency and urgency, thereby reducing queuing latency.

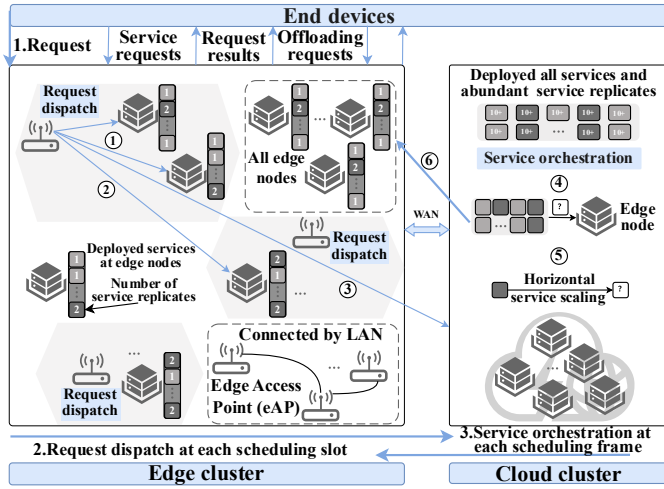


Fig. 1. K8s edge cloud network scheduling. The consists of: ① Dispatch to local edge nodes; ② Dispatch to nearby cluster edge nodes; ③ Dispatch to the cloud; ④ Perform service orchestration; ⑤ Node selection decisions; ⑥ Horizontal service scaling decisions.

C. Main Contributions

Compared to existing studies, the key contributions of KMPS include: it breaks through the limitations of single-cluster scheduling [13,14] or service orchestration [15], thereby enabling dynamic scheduling of global resource pools and service-request collaborative optimization through cross-cluster communication, multi-agent distributed decision-making, and a three-time-scale coordination mechanism. It eliminates the need for preset system states or execution parameters (e.g., single resource constraints [15], preset request computing requirements [16]). It is also able to autonomously learn strategies adapted to real-world scenarios through experience. The core components have been implemented based on the K8s architecture, with the aim of supporting seamless integration with existing edge-cloud infrastructures. The practical value of the components has been verified using real workloads from Alibaba [17]. We summarize our specific contributions as follows:

- The multi-agent proximal policy optimization (MAPPO) algorithm is adopted, integrated with gRPC-based cross-cluster scheduling and a policy context filtering mechanism to adapt to dynamic resource changes. We introduce a dual-buffer asynchronous training mechanism with a view to improving efficiency.
- To address the requirements of real-time performance, task urgency, and system stability, a three-time-scale scheduling framework is designed, encompassing real-time scheduling at short time intervals, emergency re-scheduling in long time slots, and service orchestration in time frames.
- Integrating GNN encoding with a separated Actor-Critic architecture decomposes high-dimensional service orchestration actions into node selection and

service scaling, thereby reducing the complexity of decisions.

- A time-series adaptive hybrid scheduling is used to reduce request failures by dynamically updating the estimated completion time of requests and rescheduling tasks approaching timeout.

Subsequent sections are organized as follows: Sec. II defines the problem and system model; Sec. III elaborate on the algorithm design and implementation; Sec. IV presents experimental validation; Sec. V discusses related work; Sec. VI concludes.

II. SCHEDULING PROBLEM DESCRIPTION

We focus on request scheduling and service orchestration in edge-cloud networks, aiming to improve their long-term throughput, i.e., the ratio of processed requests that meet latency requirements to the total number of requests during long-term system operation. Main notations are listed in TABLE I of Appendix A.

A. Edge-Cloud Network

Edge computing and cloud computing work together to form a three-tier architecture consisting of endpoints, edges, and clouds, enabling efficient request processing: latency-sensitive services are deployed in edge clusters (shortening response distances), and when edge resources are insufficient, the cloud serves as a computing power backup and global management center.

As demonstrated in Fig. 1, edge clusters form resource pools consisting of adjacent Edge Access Points (eAPs) and edge nodes, interconnected via local area networks (LANs). Here, an eAP serves as the entry point of an edge cluster, responsible for receiving, queuing, and scheduling incoming service requests from end devices. Multiple geographically distributed edge clusters connect to the cloud via wide area networks (WANs), forming a collaborative network. Upon arrival at an eAP, requests are able to be processed in the local cluster, in neighboring clusters, or in the cloud. Each edge cluster is responsible for a specific geographic exposes resource status through gRPC to support cross-cluster load balancing. Additionally, each cluster is equipped with an independent service orchestrator to adapt to large-scale geographically distributed scenarios. Specifically, each edge cluster hosts service entities (e.g., Docker containers) and comprises a set of eAPs $B = \{b_1, \dots, b_B\}$. Each eAP b manages its exclusive set of edge nodes $N_b = \{n_1, \dots, n_{N_b}\}$. The total node set of a single cluster is $N = \sum_{b \in B} N_b$, and the total number of nodes across all collaborative edge clusters is denoted as N_{total} . Edge clusters support cross-cluster scheduling through gRPC services (to avoid local resource saturation), while the cloud, with sufficient computing power, processes requests that cannot be handled by the edge and is responsible for global service orchestration.

B. Improving Long-Term System Throughput

KMPS achieves coordinated optimization of request scheduling and service orchestration through a three-time-scale mecha-

Algorithm 1: Training and Scheduling Process of KMPS

```

1 Initialize the system environment and neural networks
2 for slot  $t = 0.25, 0.5, \dots$  do
3   if frame  $\tau$  begins then
4     Get reward  $u_{\tau-1}$  and store  $[s_{\tau-1}, a_{\tau-1}, u_{\tau-1}]$ 
5     Use GNNs to embed system states as Eq. (4)
6     Select  $H$  high-value edge nodes ( $a_{\tau}^*$ ) and compute their service
7     scaling actions ( $a_{\tau}^*$ ) using policy networks  $\theta_g$  and  $\theta_v$ , respectively
8     Execute orchestration action  $a_{\tau} = (a_{\tau}^*, a_{\tau}^*)$ 
9     Update GNNs, policy networks, and critic networks, respectively
10  end
11 for all eAP agent  $b \in B$  do in parallel
12   if  $Q_b == \emptyset$  then
13     Continue
14   end
15   Update request queue  $Q_b$  and get reward  $u_{b,t-1}$ 
16   Store  $[s_{b,t-1}, a_{b,t-1}, A(s_{b,t-1}, a_{b,t-1}), u_{b,t-1}, F_{b,t-1}]$  for  $\theta_p$  (actor);
17   Dequeue request  $r_{b,t}$  according to  $\Psi$ 
18   Compute the resource context  $F_{b,t}$ 
19   Take dispatch action  $a_{b,t}$  for  $r_{b,t}$  using Eq. (2)
20   if  $t \bmod 2 == 0$  then
21     for task  $t \in Q_b$  do
22        $\bar{T}_{w,t} = \text{Sliding Window Avg}(T_{w,t})$ ;
23        $E_{w,t+1} = \lambda_e \cdot E_{w,t} + (1 - \lambda_e) \cdot T_{w,t}$ 
24       if  $t_{\text{deadline}} - E_{w,t+1} \leq \bar{T}_{w,t}$  then
25         Place the urgent task at the head of the queue.
26       end
27     end
28   end
29   Execute dispatched requests according to  $\Psi'$ 
30   Store  $\{s_{t-1}, V^*(s_t; \theta_v, \pi)\}$  for  $\theta_v$  (critic)
31   Asynchronously update neural networks  $\theta_p$  (actor) and  $\theta_v$  (critic),
32   respectively
33   Synchronize  $\theta_p$  periodically to distributed eAPs
34 end

```

Orchestrate (GPO)

Dispatch (MAPPO)

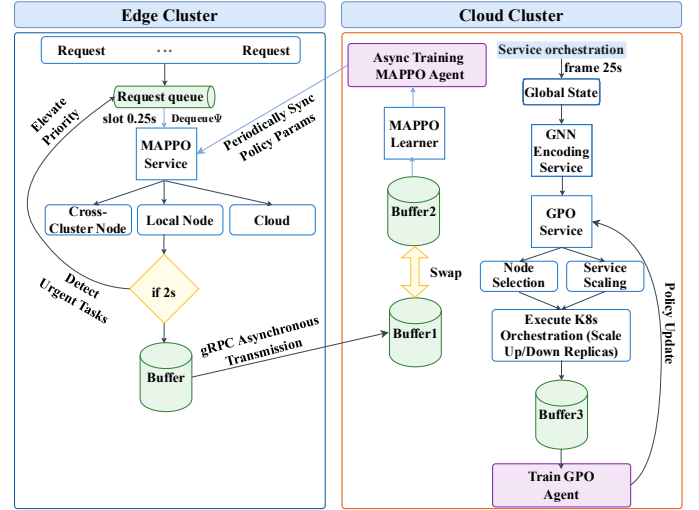


Fig. 2. The overall execution workflow of KMPS.

nism of real-time response, dynamic rescheduling, and global orchestration: short time slots (t) handle real-time request scheduling, long time slots (t_{long}) perform emergency task rescheduling, and time frames (τ) execute global service orchestration. It autonomously learns scheduling strategies from operational experience using deep reinforcement learning (DRL), encoded into neural networks.

1) *Request Scheduling at eAPs:* Each eAP b maintains Q_b and processes requests using a MAPPO-based dynamic scheduling strategy $\pi_{b,t}$ and dequeue strategy Ψ . Latency-sensitive requests arrive randomly in the queue; the request with the highest priority $r_{b,t}$ is dequeued according to Ψ . At time slot t , eAP b schedules it to a local edge node with sufficient resources hosting the required service, a neighboring cluster node via gRPC cross-cluster scheduling, or the cloud with sufficient resources (the cloud incurs additional transmission latency). Edge nodes and the cloud receive requests through Q'_n and Q_c , respectively, and using dequeue strategy Ψ' to process requests in priority order through idle service entities. Scheduling is independently performed by the eAP receiving the request (distributed decision-making) to avoid delays from global state aggregation in centralized decision-making. Requests uncompleted within the timeout are periodically discarded for free resources. A dual-buffer asynchronous training mechanism separates data collection and model updates to reduce conflicts, asynchronously updating policy network parameters and periodically synchronizing them across all eAPs to ensure policy consistency.

2) *Global Service Orchestration:* This addresses the matching of services to nodes and replica quantity control. Due to resource constraints on individual edge nodes, it is infeasible

to deploy all service types $W = \{1, \dots, w, \dots, W\}$ on every node. Single-replica services struggle to handle load surges, requiring multiple replicas for load distribution. Service demands are dynamic, with varying request arrival patterns across time windows, so orchestration strategies must adjust deployments based on real-time loads. However, overly frequent large-scale adjustments consume excessive resources, destabilize the system, and increase costs. Thus, the cloud executes global service orchestration at each frame τ using a dynamic strategy π_τ based on the GNN-based Proximal Policy Optimization (GPO) algorithm, dynamically controlling the number of replicas $d_{w,n} \in \mathbb{N}$ of service w on edge node n via discrete actions (increasing or decreasing replicas), where $d_{w,n} = 0$ indicates that service w is not deployed on node n .

3) *Optimization Objective:* The core goal is to maximize the long-term throughput Φ' . Its formulation is as follows: Define request sets $R_{n,\tau}$ (edge node n) and $R_{c,\tau}$ (cloud) within time frame τ . A binary function $\Gamma(r)$ determines if a request is completed on time ($\Gamma(r) = 1$ if $t'_r \leq \tau_r$, otherwise 0). The number of on-time requests processed by edge nodes and the cloud is $Y_\tau(Q'_n)$ and $Y_\tau(Q_c)$, respectively. The total cumulative on-time requests processed by the system is $\Phi = \sum_{\tau=0}^{\infty} (\sum_{n \in N} Y_\tau(Q'_n) + Y_\tau(Q_c))$. The normalized long-term throughput is $\Phi' = \frac{\Phi}{\sum_{\tau=0}^{\infty} \sum_{b \in B} r_\tau(Q_b)}$, where $r_\tau(Q_b)$ is the total number of requests arriving at eAP b within frame τ . Thus, the request scheduling and service orchestration problem is formulated as:

$$\max_{\pi_{b,t}; b \in B, \pi_\tau} \Phi' = \max_{\pi_{b,t}; b \in B, \pi_\tau} \frac{\sum_{\tau=0}^{\infty} (\sum_{n \in N} Y_\tau(Q'_n) + Y_\tau(Q_c))}{\sum_{\tau=0}^{\infty} \sum_{b \in B} r_\tau(Q_b)} \quad (1)$$

This involves optimizing the dynamic request scheduling strategies $\pi_{b,t}; b \in B$ of each eAP and the cloud service orchestration strategy π_τ .

The problem exhibits exponential growth in state space due to integer scheduling variables, coupling of cross-cluster scheduling, and multi-time-scale decision-making. Such issues are proven to be NP-hard (see [4] for a detailed proof), making

them difficult to solve efficiently with traditional exact algorithms. Existing methods suffer from high latency in centralized scheduling (e.g., Firmament [8]), lack of cross-cluster collaboration in single-cluster scheduling (e.g., KaiS [19]), and fixed service replica strategies that fail to adapt to dynamic loads.

III. ALGORITHM DESIGN

Algorithm 1 presents the training and scheduling workflow of KMPS. To visually illustrate this process, the overall pipeline is summarized in Fig. 2. The framework operates on a three-time-scale mechanism: real-time request dispatch at 0.25s slots using MAPPO; urgent task detection and priority elevation at 2s long slots; and global service orchestration at 25s frames using GPO. A dual-buffer mechanism enables asynchronous training to prevent scheduling blocking, with policy parameters periodically synchronized from the cloud to edge nodes. The following sections elaborate on the technical details of request scheduling and service orchestration, while detailed training configurations and mechanisms are provided in Sec. IV-A. 4).

A. Distributed MADRL Scheduling

The core of request scheduling is to enable each edge access point (eAP) to autonomously determine the target for request allocation (edge node or cloud). By balancing the load of edge nodes and timely offloading requests to neighboring clusters or the cloud, the optimization objective Φ' is maximized.

- 1) *Markov Game Modeling*: To apply multi-agent deep reinforcement learning (MADRL), the autonomous scheduling process of eAPs is modeled as a Markov game $G = (B, S, A, P, U)$ for the set of eAP agents B , defined as follows.
 - *State (S)*: At each time slot, the eAP observes a local state that encapsulates the current request attributes (e.g., service type and latency constraints), the queue lengths of the eAP and its associated edge nodes, the remaining CPU and memory resources, and network latency between the eAP and the cloud. The global state (for centralized critic training) includes the aforementioned information of all eAPs and edge nodes, as well as the queue of the cloud.
 - *Action Space (A)*: The action space A of each eAP contains $N_{total} + 1$ discrete actions, corresponding to scheduling to local edge nodes, the cloud, or edge nodes of neighboring clusters, respectively. Finally, the validity of cross-cluster scheduling is verified in real time through gRPC communication.
 - *Reward Function (U)*: The goal of each agent is to maximize its expected return $E[\sum_{i=0}^{\infty} \gamma^i u_{b,t+i}]$ (where $\gamma \in (0,1]$ is the discount factor), and the immediate reward $u_{b,t}$ is defined as: $u_{b,t} = \omega_1 \cdot \text{throughput}_{b,t} + \omega_2 \cdot \text{fail_penalty}_{b,t} + \omega_3 \cdot \text{res_pen}_{b,t}$. Specifically, (i) $\text{throughput}_{b,t}$ is the percentage of requests completed in the current time slot; (ii) $\text{fail_penalty}_{b,t}$ is the per-

centage of uncompleted requests; (iii) $\text{res_pen}_{b,t}$ is the penalty imposed only on nodes with overloaded CPU/memory usage (triggered when resource utilization exceeds 85%).

- *State Transition Probability (P)*: denotes the probability of the state transitioning from s_t to s_{t+1} under the joint scheduling action a_t , and the action is deterministic.
- 2) *Collaborative Multi-Agent Proximal Policy Optimization*: The challenges of training scheduling agents include: (i) When multiple agents learn simultaneously, their policies dynamically change and interact with the environment, increasing coordination difficulty; (ii) Feasible scheduling options change in real time with system resource states (e.g., edge nodes with exhausted resources must be excluded), which traditional DRL algorithms struggle to adapt to. Therefore, we design collaborative multi-agent proximal policy optimization (MAPPO), as shown in Fig. 3, which adopts a centralized critic and decentralized actor architecture. The critic estimates the global advantage based on the global state by sharing centralized state value functions, while the actors independently generate actions based on the local state. At the same time, through a policy context filtering mechanism, the agents adapt to the dynamic action space and establish explicit coordination. The target value of the centralized state-value function (Critic) integrates the immediate reward of the eAP and the predicted next-state value. The critic trains the shared state-value function by minimizing the Huber loss to enhance robustness against outliers. For local advantage value calculation, Generalized Advantage Estimation (GAE) is employed to balance bias and variance. In the policy optimization of the Actor module, to adapt to dynamic action spaces, MAPPO is designed with a policy context filtering mechanism: a binary vector $F_{b,t} \in \{0,1\}^{N_{total}+1}$ is computed to mark available nodes (1 for valid, 0 for invalid; the cloud is always valid). After the actor policy network outputs raw logits, invalid actions (i.e., edge nodes with insufficient available resources) are filtered out via element-wise multiplication with $F_{b,t}$. The probability of a valid action is:

$$\pi_{\theta_p}(a_{b,t} = j | s_{b,t}) = \frac{[p(s_{b,t}) \odot F_{b,t}]_j}{\|p(s_{b,t}) \odot F_{b,t}\|_1} \quad (2)$$

where θ_p are the actor policy network parameters. Policy updates are implemented by maximizing the clipped objective function, which restricts excessive policy adjustments through a clipping mechanism to ensure stable improvement.

- 3) *Request Dequeue Strategy*: Two issues regarding request queue management: how eAPs select requests for priority dispatch, and how edge nodes/clouds determine the processing order of dispatch requests. As shown in Fig. 4. For Ψ , a time-series adaptive hybrid scheduling dequeue strategy is adopted, with the following workflow: requests are sorted by the estimated completion time $E_{w,t}$ of each service type $w \in W$ (dynamically up-

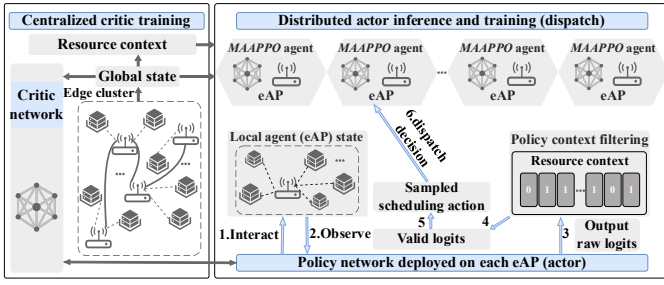


Fig. 3. Collaborative multi-agent proximal policy optimization

dated based on historical data), and requests with the shortest $E_{w,t}$ are dequeued first. To prevent long-duration tasks from being neglected, detection is triggered in long time slots, if the current time of a request equals its deadline minus $E_{w,t}$, whose priority is forcibly elevated for preemptive scheduling. The actual completion time $T_{w,t}$ of type w requests is collected periodically, and the estimated value is updated using $\lambda_e \in (0,1)$:

$$E_{w,t+1} = \lambda_e \cdot E_{w,t} + (1 - \lambda_e) \cdot T_{w,t} \quad (3)$$

As for Ψ' , it follows the same logic as Ψ . However, Ψ' ignores transmission time and only considers local processing latency. Multiple requests can be processed in a single time slot, continuously allocated to matching idle service entities until resources are exhausted or the queue is empty.

B. Service Orchestration

We propose an algorithm (GPO) that integrates graph neural networks and proximal policy optimization: GNN encodes system states, a separated Actor network decomposes high-dimensional actions, and dual Critic networks stabilize value estimation. To improve execution efficiency, the GPO service orchestrator is deployed in the cloud to leverage its stronger computing capability, and it performs system state encoding only once per global service orchestration cycle (25 seconds), with the encoding results reusable for all service adjustment decisions within this time frame.

- 1) *GNN-based System State Encoding*: Given the graph-structured nature of edge-cloud networks (including eAPs, edge nodes, cloud, and service entities), GNN is used to embed states layer by layer to capture topological dependencies and resource correlations, as shown in Fig. 5. Additionally, a lightweight GNN architecture is adopted, which consists of only two hidden layers (with 64 and 32 units respectively) and outputs a 24-dimensional feature vector. In the specific layered embedding design, we first focus on edge node embedding: the state vector $s_{n_b,\tau}$ of each node $n_b \in N_b$ at time frame τ includes: available resources (CPU, memory); network latency; request queue Q_{n_b} ; and service deployment information. Node embedding aggregates only three-hop neighbor information via message passing, and the embedding result is calculated as:

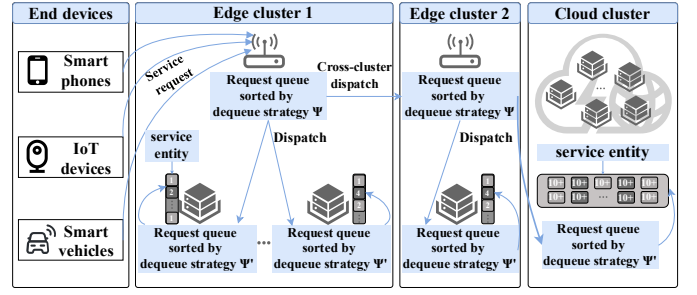


Fig. 4. Dequeue strategies

$$x_{n_b,\tau} = h_1 \left[\sum_{n'_b \in \zeta(n_b)} f_1(x_{n'_b,\tau}) \right] + s_{n_b,\tau} \quad (4)$$

where $h_1(\cdot)$ and $f_1(\cdot)$ are feature transformation neural networks. The computational complexity of a single message passing step is simplified to $O(1 \times E \times 32)$, and the total number of parameters is controlled within 10,000. Building on the edge node embedding, the embedding logic for eAPs and clusters is further extended: the embedding $y_{b,\tau}$ of each eAP b aggregates embeddings of its managed nodes; the global cluster embedding z_τ aggregates all eAP embeddings. Both use GNN structures but with independent neural networks ($h_2(\cdot)$, $f_2(\cdot)$ for eAPs; $h_3(\cdot)$, $f_3(\cdot)$ for clusters) to adapt to hierarchical features and avoids redundant cross-level feature calculations.

- 2) *Separated Actor Network*: To reduce the learning complexity and training instability of high-dimensional service orchestration actions, we employ a two-step decomposition method to split the joint action space into node selection and service adjustment decisions. The first step is node selection (NodeActor), which selects $H(\leq N)$ high-priority nodes from all edge nodes. Taking edge node embeddings $x_{n,\tau}$, eAP embeddings $y_{b,\tau}$, and cluster embeddings z_τ , as inputs, the NodeActor network calculates a scalar score $g(n, b, \tau)$. Subsequently, the selection probabilities are generated via softmax:

$$\sigma_{n,\tau} = \frac{e^{g_{n,b,\tau}}}{\sum_{b' \in B} \sum_{n_b' \in N_{b'}} e^{g_{n_b',b',\tau}}} \quad (5)$$

The second step is service adjustment (ServiceActor), which is service replica adjustment for selected nodes. Its action space is defined as $A \triangleq \{-w, \dots, 0, \dots, w\}$, where $l = w$ denotes increasing replicas of service w , $l = -w$ denotes decreasing, and $l = 0$ denotes no adjustment. The optimal action is selected via softmax. If node resources are insufficient, invalid scaling actions are automatically converted to $l = 0$.

- 3) *Dual Critic Network*: To stabilize PPO training, dual Critic networks are designed to estimate node and service values, respectively, with advantage functions computed via GAE. Specifically, NodeCritic takes as input the global node features generated by GNN and NodeActor action probabilities, outputting node-level state

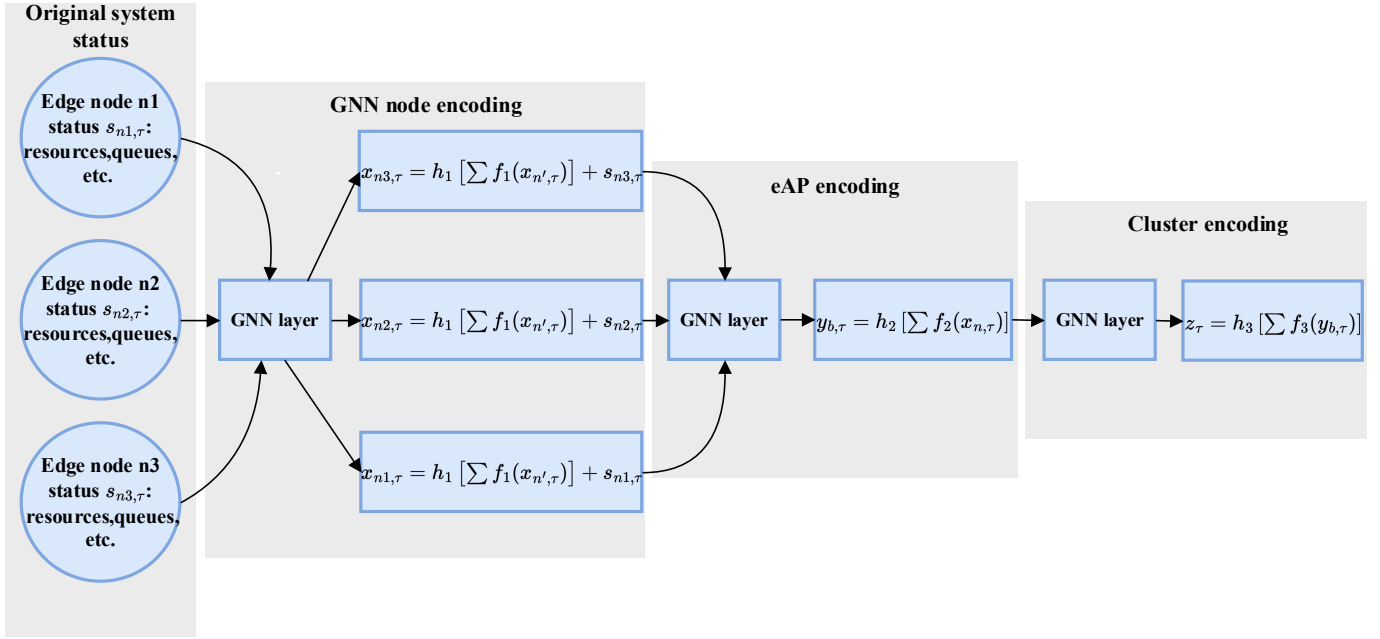


Fig. 5. GNN-based system state encoding for the edge cluster

values $V_{\text{node}}(s)$. ServiceCritic takes as input global cluster features and ServiceActor action probabilities, outputting service-level state values $V_{\text{ser}}(s)$. Accordingly, the critic loss combines losses from these dual Critics, optimizing value estimation via mean squared error (MSE):

$$L_{\text{critic}} = E \left[\left(V_{\text{node}}(s) - \hat{V}_{\text{node}} \right)^2 \right] + \left[\left(V_{\text{ser}}(s) - \hat{V}_{\text{ser}} \right)^2 \right] \quad (6)$$

For the policy optimization, the Actor loss combines the losses of NodeActor and ServiceActor, utilizing the PPO Clip mechanism to stabilize updates and an entropy term $H(\pi)$ to encourage exploration:

$$L_{\text{actor}} = L_{\text{node}} + L_{\text{service}} - \beta \cdot H(\pi) \quad (7)$$

Joint parameters, including GNN encoding parameters ($f_i(\cdot), h_i(\cdot)$ for $i = 1, 2, 3$) and policy network parameters (θ_g, θ_q) are updated via gradient descent over the GPO training cycle, with the learning rate controlling update step size. In addition, gradient blocking is used in the Critic network to embed the system state output by the GNN, which restricts the GNN to focus on state encoding and prevents it from being updated by the gradients of the Critic.

- 4) *Reward Functions*: Differentiated reward functions are designed for NodeActor and ServiceActor based on their respective optimization objectives: For NodeActor, the reward function is defined as $u_{\text{node}} = \beta_1 \cdot g - \beta_2 \cdot r$, where g is the total difference in completed requests at edge nodes before and after decision-making, and r is a resource imbalance penalty term based on Jain's fairness index. For ServiceActor, the reward function is formulated as $u_{\text{service}} = \mu_1 \cdot t - \mu_2 \cdot i$, where t is the incremental reward for replicating high-demand services, and i is the penalty for unnecessary expansion of non-high-demand services.

IV. EXPERIMENT AND RESULTS ANALYSIS

A. Experimental Data and Environmental Settings

1) *Dataset*: Real-world workload trace data from Alibaba clusters is used, containing 12 service types covering the distribution of resource demands such as CPU and memory. Task attributes include: task type (1-12 categories, mapped to indices 0-11), arrival time, deadline, CPU demand (0.1-0.5 vCPU), and memory demand. Task features are generated based on the following rules from the dataset: "start_time" is used for the arrival time of each request, "task_type" is used for the type of each request, "plan_cpu" and "plan_mem" correspond to the CPU and memory demands respectively, and "end_time - start_time" is used to determine the delay requirement of each request.

2) *Hardware and Software Environment*: The edge cluster consists of 3 K8s clusters (simulating geographical distribution), each cluster containing 1 eAP primary node and 6 edge nodes, with VM configurations of 4 vCPUs and 16 GB of memory. The cloud cluster consists of 1 K8s primary node and 6 K8s cluster nodes, with VM configurations of 4 vCPU and 16 GB of memory. Network control is simulated using LinuxTC, with latency $< 10\text{ms}$ within the edge cluster (LAN), cross-cluster latency $< 30\text{ms}$ and 100ms between the edge and cloud (WAN). The framework implementation is based on Kubernetes v1.30, Python 3.6, TensorFlow 1.14, and integrates gRPC to enable cross-cluster scheduling. Request generation is modified using Alibaba's real workload trace data to generate 12 service types. Forwarded to eAP via the request generator without the need for real terminal devices.

3) *Main components of the algorithm*: KMPS is decoupled into two main components, as shown in Fig. 6 (Prototype design of KMPS). *Decentralized Request Scheduler*: (i) A state monitor is deployed on the K8s master node to periodically

collect local resource status (CPU, memory, task queues) and network latency (covering local, neighboring clusters, and cloud), supporting dynamic routing decisions (e.g., gRPC cross-cluster). Edge nodes read Docker service status and physical node resource information via Kubelet from the Linux virtual filesystem/proc/*, and push the data to the master node to form a global view. (ii) Each K8s master node deploys a MAPPO service, which generates scheduling actions based on local states at 0.25s (slot) (as shown in Fig. 13) and notifies the K8s scheduler. Long-slot rescheduling is triggered every 8 time slots to adjust task priorities and resource allocation according to load fluctuations (e.g., sudden increases in task urgency), and to preemptively schedule high-priority tasks to resource-sufficient nodes or neighboring clusters. (iii) A gRPC-based cross-cluster three-level routing (local edge \rightarrow neighboring cluster \rightarrow cloud) is adopted. Neighboring cluster services are exposed via Kubernetes Service, and load-balancing strategies such as Round Robin are used to dynamically schedule resource-insufficient tasks, balancing load and latency. *Centralized Service Orchestrator:* (i) Different GNN encoding services are deployed on K8s edge nodes, master nodes, and cloud master nodes. They communicate with each other to compute embeddings for edge nodes, eAPs, and edge clusters, respectively, with the master node's GNN service merging the results. The GPO policy network is deployed on the master node, pulls embedding results from the GNN service at 25-second intervals (100 times the scheduling time slot), generates service scaling actions using the policy network, and interacts with the K8s API via Python-K8sclient to complete operations (services can only be deleted when idle; otherwise, operations are delayed). (ii) Selects optimal nodes for service deployment (e.g., nodes with low load and sufficient resources) based on GNN-encoded global state. Adjusts the number of service replicas based on GNN embeddings and task queue dynamics (e.g., sudden high load). (iii) Dual Critics evaluate the long-term benefits of node selection and service adjustment, respectively, optimizing resource utilization and task response time.

4) *Training settings:* For the decentralized request scheduler, we configured three key aspects. First, the Critic network adopts a four-layer fully connected neural network with 256, 128, 64, and 32 nodes per layer, using ReLU activation. The Adam optimizer is adopted (initial learning rate of 5×10^{-4} , exponentially decaying every 1000 steps). Training is stabilized using a target value network (with fixed parameters θ'_v , update at the end of each cycle). Second, the Actor policy network is a three-layer fully connected neural network with 256, 128, and 32 hidden units per layer. The output layer uses ReLU+1 activation to ensure positive logits. The optimizer and learning rate are the same as those of the Critic. Besides the aforementioned network structure and optimizer configurations, the following hyperparameters are also set: Huber loss threshold $\delta=1.0$; GAE parameters $\lambda = 0.95$ and $\gamma = 0.99$; PPO clipping parameter $\epsilon = 0.2$; exponential smoothing coefficient for task processing time estimation $\lambda_e = 0.8$. The centralized service orchestrator includes a GNN encoding module, which consists of 6 GNNs $f_i(\cdot), h_i(\cdot) (i = 1, 2, 3)$, each with two hid-

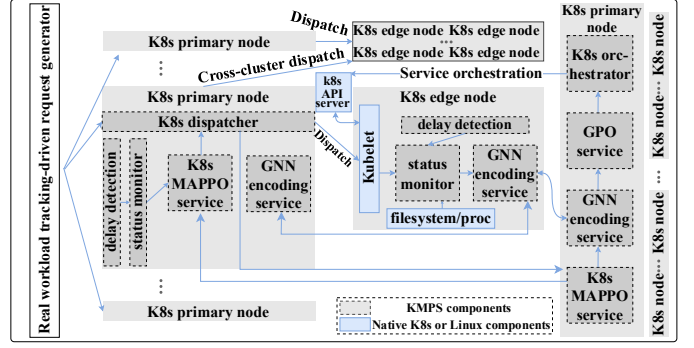


Fig. 6. Implementation and prototype design of KMPS.

den layers (64 and 32 units) for hierarchical encoding of system states. On this basis, both service selection network (θ_g) and service scaling network (θ_q) are three-layer fully connected neural networks (128, 64, 32 nodes) for evaluating node priority and service scaling action priority, respectively. The Adam optimizer is used with an initial learning rate of 10^{-3} (exponentially decaying every 1000 steps). The Critic learning rate is half that of the Actor. To enhance the model's adaptability in dynamic environments, a progressive training strategy is employed. The initial pre-training utilizes simple, short sequences (low-load, low-latency tasks), gradually introducing complex sequences (high-load, sudden emergency tasks) to enhance the model's adaptability to dynamic scenarios. In terms of training mechanisms, we are designed with a dual-buffer asynchronous training architecture. Two circular buffer pools, B_1 and B_2 (each with a capacity of 10^6 samples), are designed. The main thread collects interaction data (states, actions, rewards, etc.) and writes to the currently idle buffer pool (B_1). After completing one episode, an atomic pointer switches the writing buffer to B_2 (with the main thread continuing to write experiences), and wakes up the training thread to asynchronously read batch data from B_1 (the buffer pool that has just completed writing) and update the Actor network parameters θ_p and Critic network parameters θ_v . This producer-consumer pattern avoids inference and training blocking caused by global synchronization. Thread synchronization is achieved via condition variables to reduce data conflicts. The training thread independently updates parameters based on buffer pool samples (using proximal policy optimization and Adam optimizer) and periodically synchronizes optimized parameters to all distributed eAPs to ensure policy consistency and multi-agent collaboration. Finally, the code related to KMPS training has been open-sourced on GitHub^[18].

5) *System Deployment:* To ensure system non-intrusiveness and compatibility across different Kubernetes versions, KMPS is implemented as a set of decoupled microservices rather than modifying the native Kubernetes Scheduling Framework. The MAPPO and GPO agents operate as independent Pods. The MAPPO agent, deployed on the Kubernetes master node, functions as an intelligent agent that periodically collects local system states and executes scheduling decisions via the Kubernetes API. It integrates cluster network rules to dispatch requests to target edge nodes. Similarly, the GPO service runs

IoT-55040-2025.R2

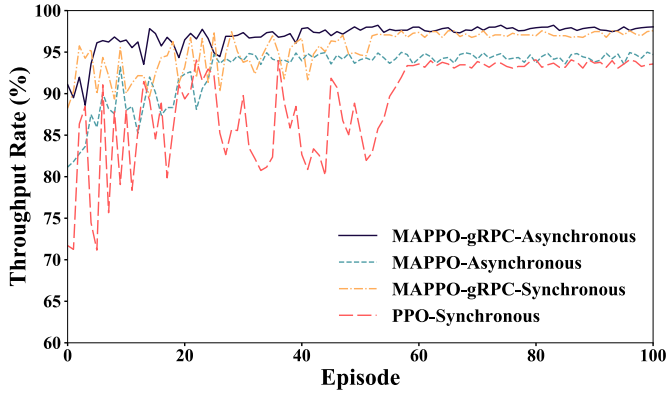


Fig. 7. Multi-agent scheduling experiment results

on the cloud-side master node, invoking the API via python-k8sclient for service orchestration.

For network integration, Cilium is adopted as the default CNI to establish cross-cluster network connectivity via Cluster Mesh. It manages Pod IP allocation and enforces eBPF-based network policies. Furthermore, to provide precise network states for the scheduling agents, a custom latency-probing DaemonSet is deployed. This component actively measures the Round-Trip Time (RTT) between nodes via standard TCP connection probes.

B. Evaluation Metrics

Inference latency (D_{inf}): The time required to execute scheduling operations. *Throughput* (Th_s): $\Phi_f = [\sum_{n \in N} Y_\tau(Q'_n) + Y_\tau(\hat{Q}_c)] / \sum_{b \in B} Y_\tau(Q_b)$, reflecting the short-term characteristics of Φ' .

C. Experimental Results and Comparative Analysis

1) *Dynamic adaptive multi-agent scheduling*: To verify the synergy between dual-buffer asynchronous training and gRPC cross-cluster routing, four comparison schemes are designed. The experimental group uses dual-buffer asynchronous training combined with gRPC cross-cluster three-level routing. Baseline 1 retains dual-buffer asynchronous training but restricts scheduling to local edges and the cloud, without cross-cluster collaboration. Baseline 2 uses single-buffer synchronous training with gRPC cross-cluster routing. Baseline 3 is the traditional PPO algorithm with single-buffer synchronous training and no cross-cluster communication, supporting only local edge and cloud scheduling.

As shown in Fig. 7, the experimental group achieves significantly higher throughput than baselines, with a stable peak above 95%. It converges rapidly after 25 episodes with minimal fluctuations, demonstrating the synergy between dual-buffer asynchronous training and gRPC cross-cluster communication. Baseline 1 performs slightly worse due to the lack of cross-cluster communication. Baseline 2 converges slowly (requiring over 40 episodes) with poor stability due to synchronous training bottlenecks. Baseline 3 (traditional single-agent) exhibits severe throughput fluctuations and low early efficiency, failing to adapt to multi-agent dynamic scheduling. These results indicate that the dual-buffer mechanism reduces

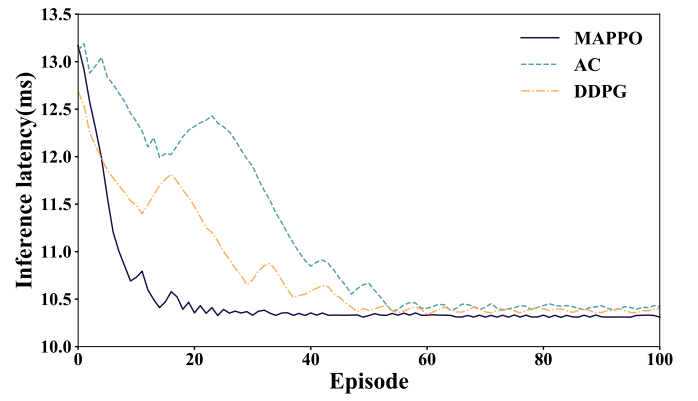


Fig. 8. Comparison of Inference Latency for Request Scheduling (Cumulative latency of 20 consecutive inference tasks)

conflicts and improves sample efficiency by separating data reading/writing, while gRPC three-level routing enables resource collaboration, significantly enhancing scheduling efficiency.

To further validate multi-agent scheduling algorithms' inference latency performance, we evaluate MAPPO against two other DRL benchmarks (AC and DDPG). Fig. 8 defines inference latency (y-axis) as the total delay of 20 consecutive tasks within an episode. The x-axis represents the number of rounds in the episode. Fig. 8 shows that MAPPO exhibits lower inference latency than both AC and DDPG, stabilizing at 10.1–10.2 ms. It converges rapidly after approximately 25 rounds with minimal fluctuation. In contrast, AC, lacking multi-agent parallel exploration and asynchronous update capabilities, demonstrates slightly inferior performance (final latency around 10.4 ms, requiring over 50 rounds to converge with noticeable fluctuations). DDPG, constrained by its single-agent update mechanism, converged more slowly (requiring over 45 rounds) with moderate stability and fluctuating mid-term latency. This demonstrates that distributed actor parallel exploration, combined with policy context filtering to reduce invalid action selections and asynchronous updates, enhances performance and convergence stability.

We have added experiments on end-to-end (E2E) latency. We decomposed the E2E latency into decision latency, transmission latency, and execution latency. In Fig. 9, under low load (100–200 req/s), the performance of each algorithm is similar. However, under high load (500–600 req/s), the Average E2E Delay of MAPPO is lower than that of AC and DDPG, ranging from 24.5 to 30.8 ms. This is because the single-agent AC and DDPG, due to the lack of a policy context filtering mechanism (excessively large state space), often assign tasks to saturated nodes or offload tasks to the cloud, leading to serious queuing problems and load imbalance. Additionally, the deterministic policy of DDPG is prone to oscillations in dynamic environments, further exacerbating the instability of scheduling. In contrast, MAPPO can effectively filter invalid actions and utilize cross-cluster scheduling to balance loads, thereby reducing queuing latency and optimizing E2E latency.

Shown in Fig. 10 and Fig. 11, both the CPU utilization and

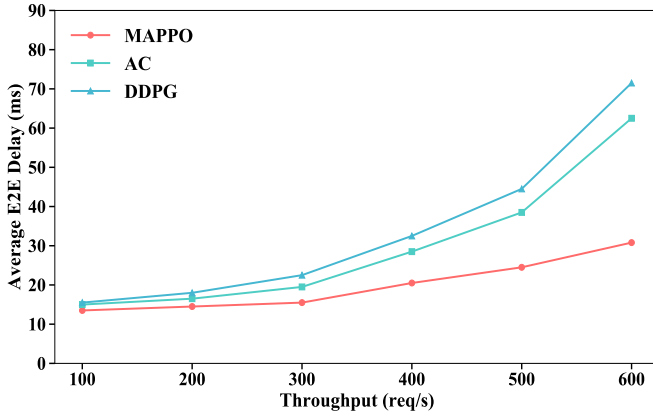


Fig. 9. Comparison of E2E Latency in Request Scheduling

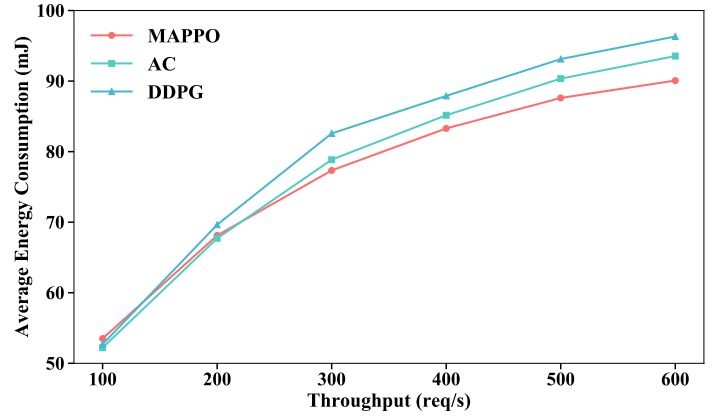


Fig. 10. Comparison of average energy consumption per inference in Request Scheduling

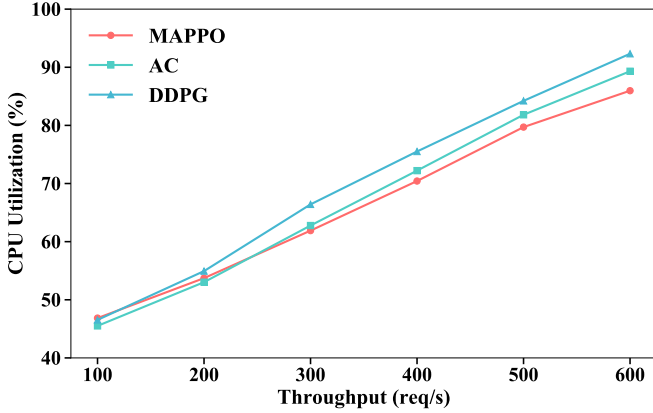


Fig. 11. Comparison of CPU utilization in eAP

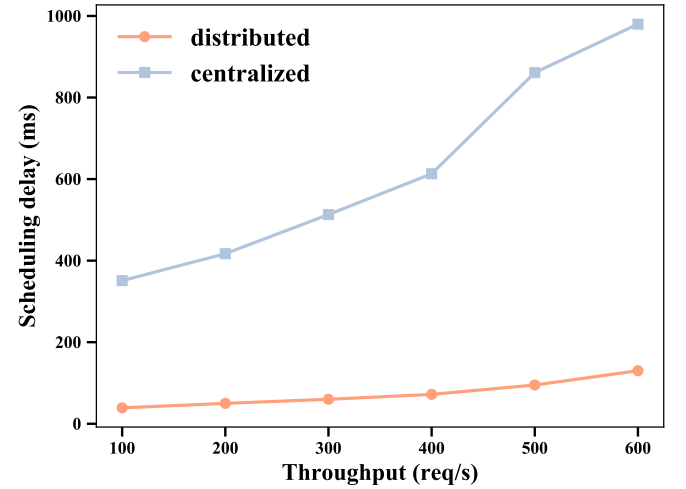


Fig. 12. Performance of distributed and centralized scheduling

average energy consumption for all methods exhibit an upward trend as the number of continuously processed requests increases. Because a continuous influx of requests extends active working hours, leading to a synchronous increase in both dynamic computing and transmission energy consumption. At low loads (e.g., 100 req/s), MAPPO does not show a significant difference in CPU utilization or energy consumption compared to the baseline methods, despite the introduction of the GNN. At high loads, centralized scheduling strategies (AC and DDPG) are restricted to local cluster resources and lack cross-cluster collaboration mechanisms. Leads to the rapid saturation of local computing resources and tasks queuing for long periods on overloaded nodes, resulting in inefficient execution and additional latency-related energy consumption. In contrast, MAPPO, based on its distributed multi-agent architecture combined with the gRPC-based cross-cluster scheduling mechanism, can rapidly offload local overflow tasks to idle nodes in neighboring clusters. By effectively distributing the load, MAPPO demonstrates a lower growth rate in CPU utilization and lower average energy consumption under the same throughput levels.

Distributed or centralized scheduling? As shown in Fig. 12, validate the significant advantages of distributed multi-agent scheduling in edge cloud environments. At low throughput (100–200 req/s), the latency of distributed scheduling remains stable at 40–50 ms, far below the 350–400 ms of cen-

tralized scheduling. At high throughput (500–600 req/s), the latency of distributed scheduling remains around 130 ms, while centralized scheduling approaches 1000 ms. This indicates that distributed scheduling achieves higher throughput through a distributed decision-making mechanism; centralized scheduling, however, experiences a sharp decline in performance as load increases due to global coordination overhead and communication latency.

2) *Three-time-scale coordinated scheduling mechanism*: Design a three-time-scale mechanism to match the real-time, task urgency, and system stability requirements of edge cloud networks: The experimental group (three-time-scale strategy) adopts a three-level cooperative mechanism of short-slot real-time response → long-slot emergency rescheduling → long-frame global orchestration. Baseline 1 (two-time-scale strategy) uses short-slot real-time response and long-frame orchestration, lacking mid-scale emergency rescheduling.

As shown in Fig. 13, in the early training stage, the two-time-scale strategy converges rapidly to 90% throughput due to simpler coordination. The three-time-scale strategy converges slightly slower due to the initialization of the long-slot rescheduling module. In the mid-to-late stages, the two-time-scale strategy fails to exceed 95% throughput under dynamic

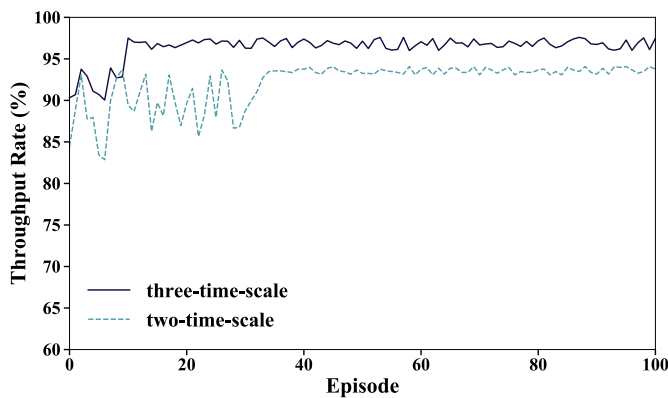


Fig. 13. Performance of scheduling at different time scales

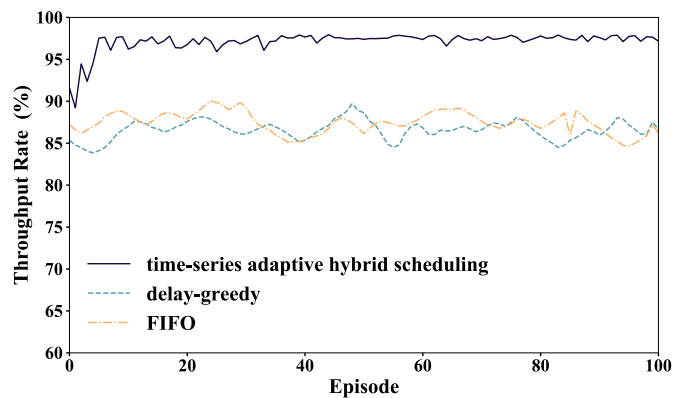


Fig. 15. Performance of different dispatch strategies

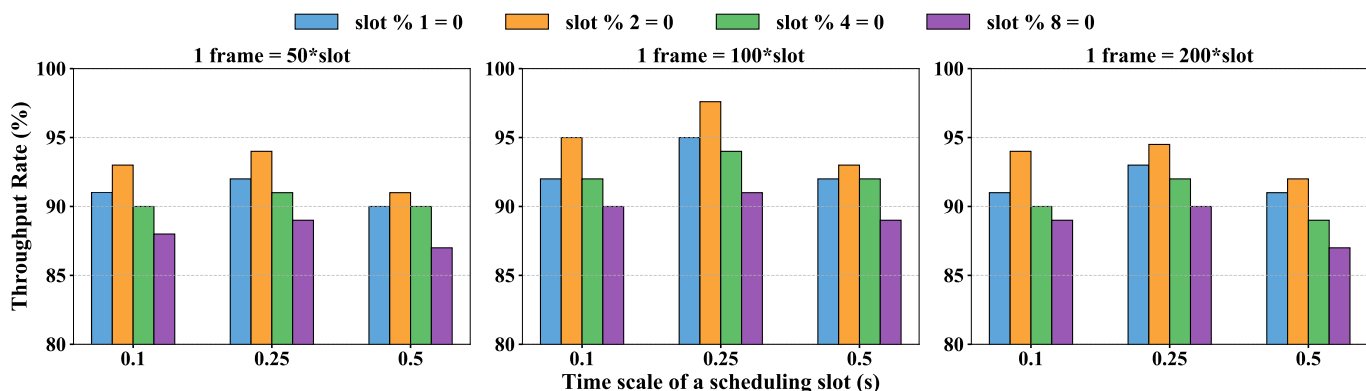


Fig. 14. KMPs scheduling performance under various settings

task urgency fluctuations (e.g., sudden high-priority tasks) due to the lack of long-slot rescheduling. In contrast, the three-time-scale strategy achieves stable high throughput through long-slot proactive intervention (task priority reordering, preemptive resource allocation), highlighting the value of long-slot rescheduling for adapting to task urgency.

As shown in Fig. 14, frequent adjustments to scheduling and orchestration do not necessarily improve performance. If the scheduling slot is too short (0.1s) or the rescheduling trigger occurs too frequently (e.g., slot %1 = 0), it may lead to “repeated state dependency” due to repeated system states, thereby weakening learning capabilities; Scheduling intervals that are too long (0.5s) or rescheduling triggers that are too slow (e.g., slot %4 or 8 = 0) may result in delayed responses, preventing timely intervention for urgent tasks, leading to a decrease in throughput as the orchestration cycle increases; Overly frequent service orchestration increases costs and interferes with learning, while overly slow orchestration may cause task blocking due to delayed resource optimization. Therefore, configuring a 0.25s (slot), a 25s (frame), and a rescheduling trigger when slot%2=0 can balance task urgency response with global resource optimization. However, note that parameters must be adjusted through experimentation based on application characteristics, load, and other scenarios, and are not universal values.

3) *Effectiveness of dispatch and rescheduling strategies.* Three departure strategies were designed for comparison: The experimental group (time-series adaptive hybrid scheduling)

sorts tasks by duration and dynamically adjusts emergency task priorities to balance scheduling efficiency and emergency response. Baseline 1 (delay-greedy) sorts requests by timeout proximity, prioritizing those closest to timeout. Baseline 2 (FIFO) schedules tasks in enqueue order without differentiation.

As shown in Fig. 15, FIFO causes long tasks to block short ones and emergency tasks to timeout, wasting resources and reducing throughput. The delay-greedy strategy prioritizes near-expiry requests but still suffers from timeouts due to insufficient remaining time, wasting resources. The experimental group improves overall turnover efficiency via shortest-job-first and ensures emergency response via dynamic rescheduling, addressing FIFO's indiscriminate blocking and delay-greedy's inefficient resource turnover.

4) *Experiment on GPO-Based Decoupled Service Orchestration:* A separated Actor network and dual Critic network architecture decouple the high-dimensional service orchestration action space into node selection and service adjustment dimensions, combined with GNN's system state encoding for refined policy optimization. The experimental group (dual Actor-dual Critic, GNN) uses dual Actors to handle node selection and service adjustment, dual Critics to evaluate long-term benefits of both actions, and GNN to perform graph embedding of system states to provide global correlation features. Baseline 1 (dual Actor-dual Critic, no GNN) decouples the action space with dual Actors and evaluates benefits with dual Critics but lacks GNN for deep encoding of system topology

IoT-55040-2025.R2

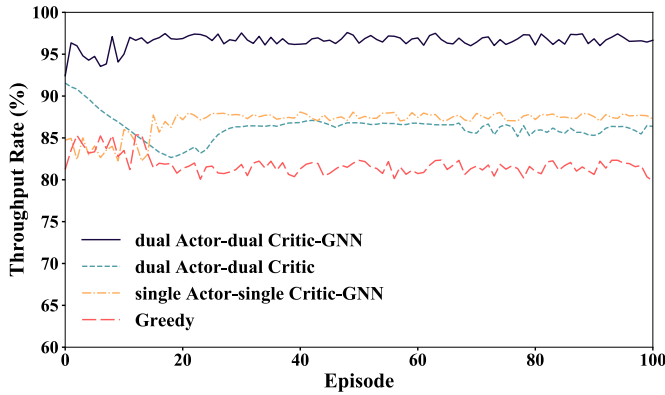


Fig. 16. Performance of different service orchestration strategies

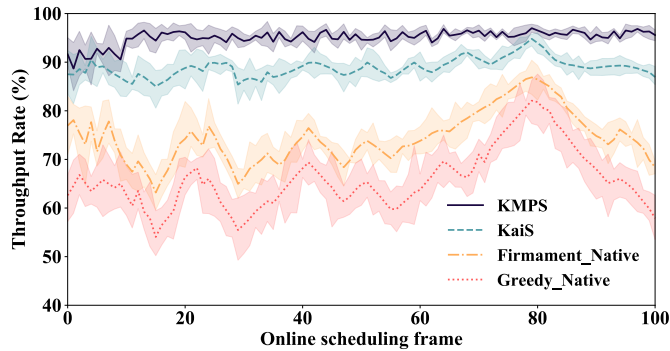


Fig. 18. Baseline comparison test results

and states. Baseline 2 (single Actor-single Critic-GNN) is a single decision-evaluation architecture with GNN-based state encoding. Baseline 3 (Greedy) scales up the service type with the longest request queue, and prioritizes deploying the newly added replicas to the node with the lowest CPU utilization.

As shown in Fig. 16, performance differences are significant. Baseline 2 degrades in complex service orchestration scenarios due to policy coupling (interference between node selection and service adjustment). Baseline 1 alleviates action dimension coupling but maintains throughput fluctuations between 83%-86% due to the lack of GNN-based global state representation. Baseline 3 exhibits significant performance degradation due to the absence of a service orchestration mechanism to adaptively release and capture global system resources. The experimental group achieves steadily increasing throughput through GNN's deep system state encoding and decoupled architecture for refined decision-making, highlighting the synergy of dual-Actor action decoupling, dual-Critic accurate evaluation, and GNN's global correlation capture.

Furthermore, we evaluated the performance comparison between the GPO algorithm and the AC algorithm, the PPO algorithm, and the Greedy algorithm, as shown in Fig. 17. The inference latency on the y-axis is defined as the total sum of latency for 50 consecutive inference tasks within an episode. The experimental results reveal that Greedy relies on fixed heuristic rules without neural network computations, resulting in extremely low latency (approximately 20ms). PPO and AC require forward inference of reinforcement learning policy networks and both involve neural network computations, but

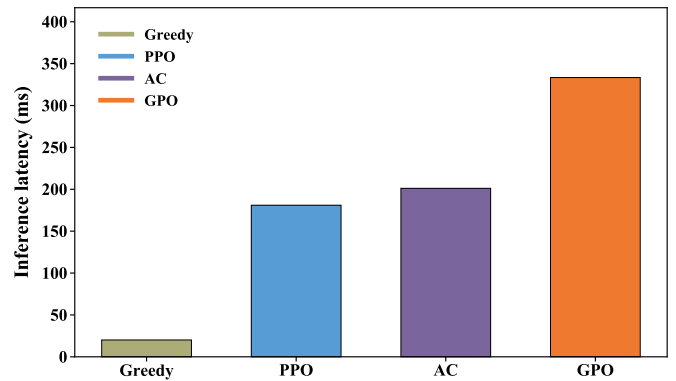


Fig. 17. Comparison of Service Orchestration Inference Latency (Cumulative latency of 50 consecutive inference tasks)

without GNN's graph structure encoding, their delays are moderate (approximately 180–200ms); GPO additionally introduces GNN for feature encoding of the system state graph, then combines with PPO to complete scheduling decisions. It has the most computational steps and thus the highest delay (approximately 330ms), but it provides support for enhancing system throughput.

In summary, GPO's latency increment represents a reasonable trade-off for throughput efficiency. Its high latency stems from deep modeling of system complexity, yielding significant throughput gains in exchange for this trade-off.

5) *Performance Comparison with Baselines*: KMPS implements request scheduling using the Multi-Agent Proximal Policy Optimization (MAPPO) algorithm for distributed decision-making, combined with gRPC cross-cluster scheduling and dynamic invalid target filtering, paired with a time-series adaptive hybrid scheduling dequeuing strategy; service orchestration uses a Graph Neural Network (GNN) to encode system state, combined with a GNN-based Proximal Policy Optimization (GPO) algorithm, coordinating request scheduling, emergency rescheduling, and service orchestration through a three-time-scale mechanism. KaiS [19] uses the coordinated multi-agent actor-observer (cMMAC) algorithm for distributed request scheduling, combined with a discount-based dequeuing strategy; service orchestration uses GNN to encode system state, combined with a GNN-based policy gradient (GPG) algorithm, coordinating request scheduling and service orchestration through a dual-timescale mechanism. Firmament_Native adopts centralized scheduling using the minimum cost maximum flow (MCMF) algorithm for task allocation, first summarizing system state observations into graph structure data and solving the request scheduling problem based on this; service orchestration relies on K8s' native Horizontal Pod Autoscaler [20], which periodically observes resource metrics, compares them to predefined thresholds, and makes decisions based on the differences. Greedy_Native assigns each request to the edge node with the lowest resource utilization in request scheduling, and uses Kubernetes' native horizontal Pod autoscaler in service orchestration.

As shown in Fig. 18, KMPS achieves 5.3% higher throughput than the closest competing baseline. KaiS performs close

to KMPS under moderate loads but exhibits significant throughput drops and fluctuations under sudden load increases due to the lack of cross-cluster collaboration (no gRPC) and only two-time-scale scheduling. Firmament_Native initializes with 350ms latency and only 75% throughput due to centralized global state aggregation. Greedy_Native maintains throughput below 80% with severe fluctuations due to fixed replica strategies unable to adapt to dynamic loads. These results highlight KMPS's superiority through cross-cluster collaboration, multi-time-scale scheduling, and service orchestration optimization.

V. RELATED WORK

Despite the fact that prior studies on optimization have investigated the upper limits of scheduling performance, these studies are not applicable to practical deployment environments (e.g. Kubernetes) due to the various assumptions made on the model. Moreover, there is a paucity of research in the systematic design of edge-edge-cloud collaborative scheduling.

A. Theoretical Analysis

Current research primarily focuses on optimizing task scheduling or service orchestration algorithms. In terms of request scheduling, for edge computing scenarios, Tang et al. [21], Nguyen et al. [22], and Shan et al. [23] propose corresponding scheduling strategies for different optimization objectives. In service orchestration, Liang et al. [24], Kayal et al. [25], and Badri et al. [26] offer solutions from diverse algorithmic perspectives. However, scheduling and orchestration strategies based on traditional models lack the ability to autonomously learn from system interactions, making it difficult to address the inherent dynamism and uncertainty of edge cloud environments.

Lin et al. [27] and Hwang et al. [28] have proposed scheduling schemes for stochastic computing or service request off-loading, which complement this research. In the field of joint optimization, Farhadi et al. [6], Poularakis et al. [7], and Ma et al. [29] provide theoretical support. However, the one-time scheduling optimization schemes in Poularakis et al. [7] and Ma et al. [29] struggle to handle continuously arriving service requests, as they do not account for the long-term impact of scheduling. Farhadi et al. [6] proposes optimizing at two different time scales to maximize the number of completed requests per scheduling. However, its long-term optimization relies on accurate predictions of future service requests, which is often impractical in real-world scenarios. Importantly, these studies [6,7,29] have limitations in practical applications: they assume that computing resources, network requirements, or processing times of specific requests can be accurately modeled or predicted. Additionally, Kaur et al. [30] proposed a Kubernetes-based Energy and Interference-Driven Scheduler (KEIDS), which is designed to optimize container management on edge-cloud nodes in the IIoT. It enhances system efficiency by minimizing energy consumption and carbon footprint, yet adopts a centralized optimization framework. Jiang et al. [31] developed the F3WDS scheduling framework, which integrates federated learning with three-way decision

theory. Targeting the resource heterogeneity challenge in multi-cloud environments, this framework partitions decision regions to achieve privacy-utility trade-off, but fails to involve proximity-aware scheduling optimization for edge nodes. Overall, while these studies demonstrate promising performance under idealized assumptions, their inability to adapt to environmental dynamics, disregard for long-term scheduling effects, and inherent bottlenecks of centralized architectures make them difficult to deploy effectively in real-world edge-cloud environments.

B. Kubernetes-Native Scheduler and System Design

At the system design level, the Kubernetes ecosystem itself provides end-to-end orchestration support for complex scheduling scenarios. These primarily fall into two categories: standalone batch schedulers (such as Volcano) and the native scheduler plugin framework. Volcano[32] is a Kubernetes-native batch scheduling system designed specifically for high-performance computing scenarios. By introducing a series of advanced scheduling primitives such as pod groups, minimum replica counts, and queue management, it effectively resolves head-of-line blocking issues while providing mechanisms like task topological sorting and fair scheduling. However, Volcano's design limitations include a lack of online learning and adaptive capabilities for handling dynamic factors in edge cloud environments, such as fluctuating network latency and burst requests. On the other hand, the Scheduler Framework mechanism introduced in Kubernetes version 1.15 [33] enables developers to customize scheduling logic through plugins. For example, the LoadAwareScheduling plugin provided by Koordinator[34] filters or prioritizes candidate nodes based on dynamic metrics such as real-time CPU utilization and memory pressure. It enables overselling of idle resources for low-priority tasks while ensuring the quality of service for high-priority tasks. Nevertheless, most plugins optimize for single scenarios (e.g., resource topology, edge dynamic filtering) and lack joint decision-making between request scheduling and service orchestration.

Existing Kubernetes edge scheduling systems exhibit a cloud-centric design tendency, making them ill-suited to the distributed nature of edge environments. Although the hybrid workflow scheduling in Sui et al. [35] and the OpenStack-Kubernetes architecture in Kristiani et al. [36] achieve edge-cloud resource integration, scheduling decisions still rely on a cloud-based central controller, leading to high response latency at edge nodes. The scheduler proposed in Haja et al. [37] periodically measures latency between edge nodes to evaluate whether the expected processing latency of service requests meets requirements and performs service orchestration accordingly. Closely related to this research, Rossi et al. [38] employs model-based reinforcement learning for service orchestration, compatible with geographically dispersed edge clusters. However, neither Haja et al. [37] nor Rossi et al. [38] addresses request scheduling in edge clusters. Current research on Kubernetes system design focuses solely on service orchestration while neglecting request scheduling, resulting in limitations in the joint optimization of requests and services. Overall,

most current research on Kubernetes-based system design focuses solely on either service orchestration or request scheduling, neglecting the synergistic optimization of both. This results in inherent limitations in enhancing overall system efficiency.

VI . CONCLUSION

Efficient collaboration between distributed edge and cloud resources using Kubernetes is a critical direction for addressing the explosive growth of terminal requests. We propose KMPS, a deep reinforcement learning-based scheduling framework tailored for Kubernetes-oriented edge-cloud networks. By dynamically learning request scheduling and service orchestration strategies, KMPS aims to enhance the long-term throughput of the system. To achieve this, KMPS integrates a series of core algorithms: a multi-agent proximal policy optimization algorithm for decentralized request scheduling (combined with a time-series adaptive hybrid dequeue strategy), and a graph neural network (GNN)-based service orchestration algorithm (which decomposes high-dimensional actions via a separated Actor network). Additionally, a three-time-scale mechanism is constructed to coordinate scheduling and orchestration. Experiments show KMPS operates stably under dynamic loads, sudden emergency tasks, and multi-cluster scenarios. Compared to baseline methods, it improves the average long-term throughput by over 5.3% and reduces cross-cluster scheduling latency by 60%. By adjusting the action space and reward function, KMPS can be adapted to other optimization objectives (e.g., minimizing resource imbalance). To adapt to the on-policy training characteristics of MAPPO request scheduling in KMPS, we explore a limited replay window mechanism under the PPO framework, which effectively improves training efficiency. Future work will conduct in-depth research on how to further optimize sample utilization efficiency while ensuring data freshness. For example, exploring strategies that combine exploration with experience replay buffers and more refined replay window designs, as well as researching the optimal replay window size across different task scenarios. Additionally, explore encapsulating the GPO service as a plugin within the Kubernetes Scheduling Framework.

APPENDIX A

DETAILED VARIABLES AND DEFINITIONS

TABLE I
MAIN NOTATIONS

Symbol	Core Description
$a_{b,t}$	eAP b request scheduling action at time slot t
b	Index of eAP
B	Set of eAPs
c	Cloud cluster
$d_{w,n}$	Number of replicas of service w on edge node n

$E_{w,t}$	Estimated completion time of requests of type w
n	Index of edge node
N	Set of edge nodes in a single edge cluster
N_{total}	Total number of nodes in all collaborative edge clusters
Q_b	Request queue of eAP b
Q_c	Request queue of the cloud cluster
Q'_n	Scheduled request queue of edge node n
$r_{b,t}$	Request scheduled by eAP b at time slot t
t	Index of time slot
t_{long}	Index of long time slot
τ	Index of time frame
w	Index of service
W	Set of all services
Φ'	Long-term throughput rate
Ψ	Dequeue strategy of eAP
Ψ'	Dequeue strategy of edge nodes and cloud

REFERENCES

- [1] Foundation T L. Kubernetes. Accessed: Dec. 13, 2025. [Online]. Available: <https://kubernetes.io>.
- [2] Foundation T L. Kubeedge. Accessed: Dec. 13, 2025. [Online]. Available: <https://kubeedge.io/zh>.
- [3] Foundation T L. Openyurt. Accessed: Dec. 13, 2025. [Online]. Available: <https://openyurt.io/zh/>.
- [4] Z. Chen, W. Hu, J. Wang, S. Zhao, B. Amos, G. Wu, K. Ha, K. Elgazzar, P. Pillai, R. Klatzky, D. Siewiorek, M. Satyanarayanan, "An empirical study of latency in an emerging class of edge computing applications for wearable cognitive assistance." SEC '17: Article No 14 pp 1–14.
- [5] S. Yang, Y. Ren, J. Zhang, J. Guan, B. Li, "KubeHICE: Performance-aware container orchestration on heterogeneous-ISA architectures in cloud-edge platforms." ISPA/BDCLOUD/SocialCom/SustainCom '21: pp 81–91.
- [6] V. Farhadi, F. Mehmeti, T. He, T. F. La Porta, H. Khamfroush, S. Wang, K. S. Chan, and K. Poularakis, "Service placement and request scheduling for data-intensive applications in edge clouds," *IEEE/ACM Trans. Netw.*, vol. 29, no. 2, pp. 779–792, Apr. 2021, doi: 10.1109/TNET.2020.3048613.
- [7] K. Poularakis, J. Llorca, A. M. Tulino, I. Taylor, and L. Tassiulas, "Joint service placement and request routing in multi-cell mobile edge computing networks," in *Proc. IEEE INFOCOM Conf. Comput. Commun.*, Apr. 2019, pp. 10–18.
- [8] I. Gog et al., "Firmament: Fast, centralized cluster scheduling at scale," in *Proc. USENIX OSDI*, 2016, pp. 99–115.
- [9] V. Mnih et al., "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.
- [10] T. P. Lillicrap et al., "Continuous control with deep reinforcement learning," 2015, arXiv:1509.02971.
- [11] F. Wang, F. Wang, J. Liu, R. Shea, and L. Sun, "Intelligent video caching at network edge: A multi-agent deep reinforcement learning approach," in *Proc. IEEE INFOCOM Conf. Comput. Commun.*, Jul. 2020, pp. 2499–2508.
- [12] Z. Zhang, P. Cui, and W. Zhu, "Deep learning on graphs: A survey," *IEEE Trans. Knowl. Data Eng.*, vol. 34, no. 1, pp. 249–270, Jan. 2022.
- [13] J. Chen, Y. Yang, C. Wang, H. Zhang, C. Qiu, and X. Wang, "Multitask offloading strategy optimization based on directed acyclic graphs for edge computing," *IEEE Internet Things J.*, vol. 9, no. 12, pp. 9367–9378, Jun. 2022.
- [14] Z. Liu et al., "Hastening Stream Offloading of Inference via Multi-Exit

- DNNs in Mobile Edge Computing," in *IEEE Transactions on Mobile Computing*, vol. 23, no. 1, pp. 535-548, Jan. 2024, doi: 10.1109/TMC.2022.3218724.
- [15] W. Lv et al., "Microservice deployment in edge computing based on deep Q learning," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 1, pp. 2968-2978, Nov. 2022.
- [16] X. Wang, Z. Ning, and S. Guo, "Multi-agent imitation learning for pervasive edge computing: A decentralized computation offloading algorithm," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 2, pp. 411-425, Feb. 2021.
- [17] *Alibaba-Clusterdata*. Accessed: Dec. 13, 2025. [Online]. Available: <https://github.com/alibaba/clusterdata>
- [18] *KMPS*. Accessed: Dec. 13, 2025. [Online]. Available: <https://github.com/KMPS-2025/KMPS>
- [19] Y. Han, S. Shen, X. Wang, S. Wang, and V. C. M. Leung, "Tailored learning-based scheduling for Kubernetes-oriented edge-cloud system," in *Proc. IEEE INFOCOM Conf. Comput. Commun.*, May 2021, pp. 1-10.
- [20] *K8s Documentation:Horizontal Pod Autoscaler*. Accessed: Dec. 13, 2025. [Online]. Available: <https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/>
- [21] B. Tang, J. Luo, M. S. Obaidat, et al. "Container-based task scheduling in cloud-edge collaborative environment using priority-aware greedy strategy". *Cluster Computing*, 2023, vol. 26, no. 6: 3689-3705.
- [22] N. H. Nguyen, V. -D. Nguyen, A. T. Nguyen, N. V. Thieu, H. N. Nguyen and S. Chatzinotas, "Deadline-Aware Joint Task Scheduling and Offloading in Mobile-Edge Computing Systems," in *IEEE Internet of Things J*, vol. 11, no. 20, pp. 33282-33295, 15 Oct.15, 2024, doi: 10.1109/IJOT.2024.3425854.
- [23] C. Shan et al., "KCES: A Workflow Containerization Scheduling Scheme Under Cloud-Edge Collaboration Framework," in *IEEE Internet of Things J*, vol. 12, no. 2, pp. 2026-2042, 15 Jan.15, 2025, doi: 10.1109/IJOT.2024.3466231.
- [24] Y. Liang et al., "Interaction-oriented service entity placement in edge computing," *IEEE Trans. Mobile Comput.*, vol. 20, no. 3, pp. 1064-1075, Mar. 2021.
- [25] P. Kayal and J. Liebeherr, "Distributed service placement in fog computing: An iterative combinatorial auction approach," in *Proc. IEEE 39th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jul. 2019, pp. 2145 - 2156.
- [26] H. Badri, T. Bahreini, D. Grosu, and K. Yang, "Energy-aware application placement in mobile edge computing: A stochastic optimization approach," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 4, pp. 909-922, Apr. 2020.
- [27] L. Lin, X. Liao, H. Jin and P. Li, "Computation Offloading Toward Edge Computing," in *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1584-1607, Aug. 2019, doi: 10.1109/JPROC.2019.2922285.
- [28] J. Hwang, L. Nkenyereye, N. Sung, J. Kim and J. Song, "IoT Service Slicing and Task Offloading for Edge Computing," in *IEEE Internet Things J*, vol. 8, no. 14, pp. 11526-11547, 15 July15, 2021.
- [29] X. Ma, A. Zhou, S. Zhang, and S. Wang, "Cooperative service caching and workload scheduling in mobile edge computing," in *Proc. IEEE INFOCOM Conf. Comput. Commun.*, Jul. 2020, pp. 2076-2085.
- [30] K. Kaur, S. Garg, G. Kaddoum, S. Ahmed and M. Atiquzzaman. "KEIDS: Kubernetes-Based Energy and Interference Driven Scheduler for Industrial IoT in Edge-Cloud Ecosystem," *IEEE Internet of Things J*, 2020, vol. 7, no. 5, pp. 4228-4237
- [31] C. Jiang and L. Su, "Federated learning with three-way decisions for privacy-preserving multicloud resource scheduling," *Appl. Soft Comput.*, vol. 183, Art. no. 113634, Nov. 2025, doi: 10.1016/j.asoc.2025.113634.
- [32] *CNCF*. Accessed: Dec. 13, 2025. [Online]. Available: <https://github.com/volcano-sh/volcano>
- [33] *Kubernetes. Scheduler Framework*. Accessed: Dec. 13, 2025. [Online]. <https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/>
- [34] *Koordinator*. Accessed: Dec. 13, 2025. [Online]. Available: <https://github.com/koordinator-sh/koordinator>
- [35] W. Sui, Y. Zhou, S. Zhu, Y. Xu and S. Wang. "5G Edge Network of Collaborative Computing Task-Scheduling Algorithm with Cloud Edge," *Mobile Inf. Syst.*, 2022, 2022
- [36] E. Kristiani, C.-T. Yang, C.-Y. Huang, Y.-T. Wang and P.-C. Ko, "The Implementation of a Cloud-Edge Computing Architecture Using Open-Stack and Kubernetes for Air Quality", *Mobile Netw. Appl.*, 2021, vol. 26, no. 3:1070-1092
- [37] D. Haja, M. Szalay, B. Sonkoly, G. Pongracz, and L. Toka, "Sharpening

Kubernetes for the edge," in *Proc. ACM SIGCOMM Conf. Posters Demos*, Aug. 2019, pp. 136-137.

- [38] F. Rossi, V. Cardellini, F. Lo Presti, and M. Nardelli, "Geo-distributed efficient deployment of containers with kubernetes," *Comput. Commun.*, vol. 159, pp. 161-174, Jun. 2020.



Congyue Huang received the B.S. degree in computer science and technology from Hanshan Normal University, Chaozhou, China, in 2024, and he is currently pursuing the M.S. degree in computer science and technology from Dongguan University of Technology, Dongguan, China.

His current research interests include edge computing and its applications.



Wei Tan received the B.S. degree from the China University of Geosciences, Wuhan, China, in 1999, the M.E. degree from Wuhan University, Wuhan, China, in 2002, and the Ph.D. degree from the South China University of Technology, Guangzhou, China, in 2013. He was a Visiting Scholar with the University of

Strathclyde, Glasgow, U.K., in 2018.

He is currently a Faculty Member and Master's Supervisor with the School of Computer Science and Technology, Dongguan University of Technology, Dongguan, China. His professional service includes roles as a member of the Service Computing Association of the China Computer Federation (CCF) and an expert for the Guangdong Province Bid Evaluation Expert Database. He has authored or co-authored over 100 journal papers and two book chapters.

His research interests include edge computing, service computing, cloud manufacturing, and computer vision.



Miaohua Ou received the B.S. degree in computer science and technology from Guangzhou University of Chinese Medicine, Guangzhou, China, in 2021, and the M.S. degree in computer science and technology from Dongguan University of Technology, Dongguan, China, in 2025.

Her current research interests include service composition optimization and intelligent optimization algorithms.



Erfu Yang (Senior Member, IEEE) received the Ph.D. degree in robotics from the School of Computer Science and Electronic Engineering, University of Essex, Colchester, U.K., in 2008.

He is currently a Reader within the Department of Design, Manufacturing and Engineering Management, University of Strathclyde, Glasgow, U.K. He has more than 200 publications in his research areas, including more than 100 journal papers and 10 book chapters. His main research interests in-

clude robotics, autonomous systems, mechatronics, manufacturing automation, signal and image processing, computer vision, and applications of machine learning and artificial intelligence.



Yun Li (Fellow, IEEE) received the B.S. degree from Sichuan University, Chengdu, China, in 1984, the M.E. degree from University of Electronic Science and Technology of China (UESTC), Chengdu, China, in 1987, and the Ph.D. degree from University of Strathclyde, Glasgow, U.K., in 1990.

In 1989, he was an Intelligent Control Engineer with the U.K. National Engineering Laboratory, Glasgow. In 1990, he was a Postdoctoral Research Engineer with Industrial Systems and Control Ltd, Glasgow. From 1991 to 2018, he was an Intelligent Systems Lecturer, Senior Lecturer, and Professor with University of Glasgow, Scotland, and served as Founding Director of University of Glasgow Singapore, Singapore. He later served as the Founding Director of Dongguan Industry 4.0 Artificial Intelligence Laboratory, Dongguan, China, and of i4AI Ltd, London, U.K. He is currently a Changjiang Chair Professor at Shenzhen Institute for Advanced Study, UESTC. He has published over 350 papers, and one of them since publication in the IEEE TRANSACTIONS ON CONTROL SYSTEM TECHNOLOGY in 2005 has been the most popular article in control engineering almost every month. He is interested in the next generation, explainable artificial intelligence (AI) and has supervised over 30 PhD students focusing on AI for Engineering since 1991, his research interests have been focused on computational artificial intelligence and its applications.

Prof. Li is a U.K. Chartered Engineer and is currently an Associate Editor of IEEE TRANSACTIONS ON EMERGING TOPICS IN COMPUTATIONAL INTELLIGENCE.