Learning to Optimize Workflow Scheduling for an Edge–Cloud Computing Environment

Kaige Zhu, Zhenjiang Zhang, Sherali Zeadally, *Senior Member, IEEE*, and Feng Sun

***Abstract*—The widespread deployment of intelligent Internet of Things (IoT) devices brings tighter latency demands on complex workload patterns such as workflows. In such applications, tremendous dataflows are generated and processed in accordance with specific service chains. Edge computing has proven its feasibility in reducing the traffic in the core network and relieving cloud datacenters of fragmented computational demands. However, the efficient scheduling of workflows in hybrid edge–cloud networks is still challenging for the intelligent IoT paradigm. Existing works make dispatching decisions prior to real execution, making it difficult to cope with the dynamicity of the environment. Consequently, the schedulers are affected both by the scheduling strategy and by the mutual impact of dynamic workloads. We design an intelligent workflow scheduler for use in an edge–cloud network where workloads are generated with continuous steady arrivals. We develop new graph neural network (GNN)-based representations for task embedding and we design a proximal policy optimization (PPO)-based online learning scheduler. We further introduce an intrinsic reward to obtain an instantaneous evaluation of the dispatching decision and correct the scheduling policy on-the-fly. Numerical results validate the feasibility of our proposal as it outperforms existing works with an improved quality of service (QoS) level.**

***Index Terms*—Workflow scheduling, edge computing, Internet of Things, reinforcement learning.**

I. Introduction

THE data-driven intelligent Internet of Things (IoT) paradigm is becoming the future trend of digital factories and even the intelligent manufacturing industry. The widespread deployment of intelligent sensors brings with it an increasing complexity of IoT digital services which imposes more urgent demands for timely responses. Although the centrally structured cloud paradigm has proven it usability in previous practices in the industrial IoT, typical cloud centers still face a dilemma when dealing with emerging IoT services [1]. This can be attributed to the increasing requirements in terms of tighter response delays, larger data streams, and more fragmented computational demands.

As a supplement to cloud computing as well as an enabler for ubiquitous computing, edge computing has been widely studied in recent years. Edge computing enables tasks to be processed in a timely manner at the back end of data sources. Since edge computing can effectively reduce the burden of data streams in the core network and mitigate fragmented computational requests in the cloud, it promotes the inherent advantages of cloud centers in terms of their large data storage capacities and superior computational capabilities. However, to support design features such as distributed deployment and easy installation, most edge servers are powered by batteries, resulting in high demands in terms of network utility. The studies in [1], [2] have concluded that under certain network and service conditions, the introduction of edge servers will even degrade the network utility and make the solution even more complex. The situation remains challenging when more complex digital services are implemented in a hybrid edge–cloud network, necessitating further improvement in the current edge computing paradigm.

As an emerging architecture of digital services, the microservice regulates a pluggable structure of applications as a combination of loosely coupled, collaborating and interdependent services [3]. Different from existing works in which workloads are simplified as independent tasks, in this paper, we consider the workload that follows the microservices architecture wherein the IoT applications are divided into atomic microservices that follow certain dependency constraints [4], [5], [6] exists as a workflow. Here, tasks are executed following specific precedence constraints, and we express such dependency constraints in the form of a task dependency graph for ease of notation. Compared with task offloading in the case of independent tasks, workflow scheduling presents three additional challenges: (1) workflow schedulers have a lagging effect, i.e., the makespan of a workflow is the result of mutual impacts among the tasks, making it difficult to evaluate the offloading decision for a single task; (2) different dispatching orders of tasks that belong to different branches can change the critical path of the workflow; and (3) a complex execution order of tasks will degrade the scheduling performance, but it is difficult to perceive the latent structure of the dependency graph if the tasks are treated individually. The

Manuscript received 12 April 2023; revised 27 May 2024; accepted 28 May 2024. Date of publication 31 May 2024; date of current version 6 September 2024. This work was supported in part by the Fundamental Research Funds for the Central Universities under Grant 2022JBZY002, and in part the National Natural Science Foundation of China under Grant 62173026. Recommended for acceptance by W. Zhang. *(Corresponding author: Zhenjiang Zhang)*

Kaige Zhu is with the School of Electronic and Information Engineering, Beijing Jiaotong University, Beijing 100044, China (e-mail: kaigezhu@bjtu.edu.cn).

Zhenjiang Zhang is with the School of Electronic and Information Engineering, Beijing Jiaotong University, Beijing 100044, China, and also with the Key Laboratory of Communication and Information Systems, Beijing Municipal Commission of Education, Beijing 100044, China (e-mail: zhangzhenjiang@bjtu.edu.cn).

Sherali Zeadally is with the College of Communication and Information, University of Kentucky, Lexington, KY 40506 USA (e-mail: szeadally@uky.edu).

Feng Sun is with the School of Electronic and Information Engineering, Beijing Jiaotong University, Beijing 100044, China (e-mail: sunfeng@bjtu.edu.cn).

Digital Object Identifier 10.1109/TCC.2024.3408006

ability to perceive and utilize the structural information of the dependency graph is key to a workflow scheduler.

Since workflow scheduling in a hybrid edge–cloud network is a complex tradeoff involving inter- and intra-workflow interactions, this scheduling problem has been proven to be NP-hard in many recent works [1], [7], [8], [9]. To solve this problem, traditional heuristics-based approaches require specially designed representations of expert experience as well as tremendous efforts in terms of iteration and validation. However, the workload of such approaches grows exponentially when the network scale increases, preventing their further implementation in intelligent IoT scenarios. Recent advances in machine learning have motivated efforts to design workflow schedulers in a more intelligent manner. Studies have already emerged in which workflow scheduling is modeled as a discrete-time control problem [1], [10], [11], [12] in order to develop a workflow scheduler with the help of deep reinforcement learning (DRL). In these works, the task dispatching strategies are usually defined to maximize the designed reward of the current task. However, the lagging effect of a workflow is closely coupled with the structure of the task dependency graph, and manually designed task-specific rewards may be misaligned with the intended goals and can lead to undesirable results [13], [14]. Consequently, how to better perceive and understand the task dependency graph and overcome the lagging effect remains an open problem.

In this paper, we focus on addressing the problem of workflow scheduling in a microservice-oriented hybrid edge–cloud network with continuous steady task arrivals. We formulate a nonconvex optimization problem to minimize the makespan of the workflows while simultaneously optimizing the energy utilization. We model the sequential task scheduling decision-making procedure as a Markov decision process (MDP) and propose an intrinsic-reward-based intelligent scheduler as the solution. Specifically, we develop a new graph neural network (GNN)-based encoder for representing task dependency constraints to better capture the latent structural information of the task dependency graphs. To address the challenge of the lagging effect of workflow schedulers, we introduce an intrinsic reward to help evaluate the intermediate decisions during the scheduling process. Taking these representations as the input, we design an online reinforcement-learning-based scheduler to manage the scheduling decisions.

Research contributions of this work

We summarize our main research contributions as follows:

1) In this paper, we explore the workflow scheduling problem in a hybrid edge–cloud network. We aim to maximize the network utility by balancing the tradeoff between the makespan and energy efficiency. Our work presents an online learning framework that not only makes efficient offloading decisions for workflows but also adjusts the policies based on the network status.
2) We design a scalable neural network that can better handle workflows of different forms. Based on GNNs, we encode the tasks in the workflows as fixed-dimensional representations to utilize the latent structural information of the task dependency graphs. We further use a neural network that takes the task embeddings as its input to determine the scheduling order of these tasks.
3) Using dedicated designed agents, we show that the lagging effect of workflow schedulers can be mitigated if intrinsic rewards are introduced to evaluate the intermediate scheduling decisions. Our experimental results demonstrate that, compared with existing baselines, our proposed algorithm yields superior performance in handling workflows with continuous steady arrivals.

We organize the remainder of this paper as follows. Section II presents an overview of related works. Section III introduces the basic assumptions of the environmental settings and the system model. Section IV presents our proposed workflow scheduler algorithm. In section V, we discuss the numerical results obtained. Finally, section VI concludes the paper.

II. Related Work

Many works have sought a solution to the task offloading problem in a heterogeneous edge computing environment. In [4], Qu et al. investigated multiple microservice deployment policies on an edge computing platform, and verified the realization of the deployment mechanism on the container, strengthening the foundation of a scheduler for microservices in the edge computing environment. Nevertheless, the design of workflow schedulers remains an open problem. The major barriers concern the representation of the task dependency graph and can be divided into two aspects. First, the adjacency matrix of the task dependency graph is sparse due to the nature of workflows. Second, the size of the graph varies from one workflow to another, and the connectivity patterns of the tasks may be highly diverse. Consequently, it remains challenging to perceive the latent information in the task dependency graphs while generating scheduling decisions from high-dimensional features. The features and shortcomings of the current works are summarized in Table I.

Heuristics-based schedulers have been extensively studied [6], [15], [16], [17], [18], [19] in the context of directed acyclic graph (DAG) scheduling. Typical heuristics-based schedulers can be generalized into three major categories, i.e., list-based scheduling, cluster-based scheduling and task-duplication-based scheduling [16]. The key to list-based scheduling lies in maintaining a list of tasks that are ready to be executed [15]. The authors of [19] proposed a list-based scheduler HEFT by employing topological sorting of the workflows. In their work, the average execution time of the tasks and the average communication time are jointly taken into consideration to assist in the scheduling process. Critical path on a processor (CPOP) is another scheduler proposed in the same work that further considers the effect of the critical path. Garg et al. [17] proposed a scheduler based on particle swarm optimization (PSO) to jointly optimize makespan, cost and reliability objectives under deadline constraints. Similarly, Xie et al. [6] focused on improving the PSO-based workflow scheduler by employing inertia weight with selection and directional search process. Task duplication is another typical approach for saving transmission time. The authors of [18] proposed a clustering-based scheduler

TABLE I
SUMMARY OF LITERATURE REVIEW

Ref	Workflow	Edge-cloud	Algorithm	Optimization objective	Shortcomings
[19]	Static	--	List-based scheduling	Makespan	Missing latent structural information of the tasks, and absence of load balancing
[17]	Static	Grid computing	PSO	Makespan, cost, reliability	Complexity associated with network scale (PSO)
[6]	Static	Edge-cloud	Improved PSO	Makespan, price	Only addressed the scheduling of static workflow
[18]	Static	--	Clustering-based scheduling	Makespan	Complexity associated with network scale (extra traffic on control flow)
[8]	Static	Cloud only	Q-Learning	Makespan, energy, budget constraint	Only addressed the scheduling of static workflow
[20]	Static	Distributed compute clusters	GNN+DRL	Makespan	Only addressed the scheduling of static workflow
[21]	Static	--	GNN+ DRL	Makespan	Only addressed the scheduling of static workflow
[23]	Static	--	GIN+PPO	Makespan	Only addressed the scheduling of static workflow
[1]	Dynamic	Edge-cloud	PPO	Makespan, energy	Missing latent structural information of the tasks, and affected by lagging effect
[5]	Dynamic	Edge only	Convex optimization	Network throughput, total network delay	Lagging effect, and missing latent structural information of the tasks
[12]	Dynamic	Edge-cloud	Monte Carlo Tree Search	Energy, Makespan, SLA violation rates	Complexity associated with network scale (Monte Carlo tree), and affected by lagging effect
Proposed	*Dynamic*	*Edge-cloud*	*PPO with intrinsic reward*	*Makespan, energy*	

to duplicate critical parent tasks. Inevitably, heuristics-based approaches can produce satisfactory solutions at a lower cost under certain circumstances. However, even without additional consideration of the control flow, the scheduling problem itself is still NP-hard. In addition, these approaches rely heavily on expert experience as well as human instinct; consequently, scheduler design becomes increasingly difficult as the network scale increases.

Recent years have witnessed breakthroughs in artificial intelligence, providing new opportunities in the design of workflow schedulers. In [8], Yao et al. focused on minimizing the makespan and energy consumption under a budget constraint. The authors used reinforcement learning based on the Chebyshev scalarization function to solve the problem of weight selection for the two indices. To simplify the scheduling problem in edge–cloud computing environments, Amanda et al. [1] proposed an actor–critic-based scheduling framework with proximal policy optimization (PPO). In their work, they trained two identical neural networks to decouple the task dispatching decisions to separately generate decisions on the kind of server to be chosen for offloading and the exact dispatching decisions.

Previous workflow schedulers have relied on the direct processing of the tasks in a workflow. The dependencies among these tasks have simply been treated as constraints, foregoing any potential improvements based on the topology of these task dependency constraints. To this end, we can use a GNN to better perceive the latent structural information of the tasks in a workflow. The authors of [20] proposed a hierarchical task embedding scheme in which the embedding information is categorized into the job stage, a DAG summary, and a global summary. In [21], the authors proposed a GNN-based deep reinforcement learning structure to address the problem of co-flow scheduling, a typical kind of scheduling problem that coordinates a set of flows that are correlated in a task with multiple phases [21], [22]. In their work, the authors limited the workflows to a certain number of possible topologies and trained the agents accordingly. In contrast, the authors of [23] focused on job shop scheduling, a special kind of scheduling problem in which the tasks must be processed on every processor. Here, the authors adopted a graph isomorphism network to process the task dependency graph.

Most existing works [9], [20], [21], [23] have simply considered the situation in which the workflows are generated at the beginning of the scheduling process. Thus, the proposed scheduling algorithms can iteratively optimize the dispatching decisions in accordance with stationary task dependencies. However, workflow scheduling for dynamic workflows is far more complex. In [5], Samanta et al. proposed a dynamic microservice scheduler in the edge computing environment that jointly optimizes the network throughput and the QoS-level to end users. In [12], the authors focused on workflows with continuous steady arrivals. The authors used a tree-based search strategy to optimize the decision-making process. However, the proposed Monte Carlo tree has difficulty remaining effective, especially as the system scales. In contrast to the previous work in [12], in this paper, we propose a scheduler based on meta-reinforcement learning to address the lagging effect during the task dispatching process. By combining this scheduler with GNN-based task embedding, our work provides a solution that is better suited for dynamic workloads.

III. SYSTEM MODEL AND PROBLEM DEFINITION

In this paper, we consider microservices for IoT applications in edge–cloud computing scenario. Fig. 1 shows an overview of

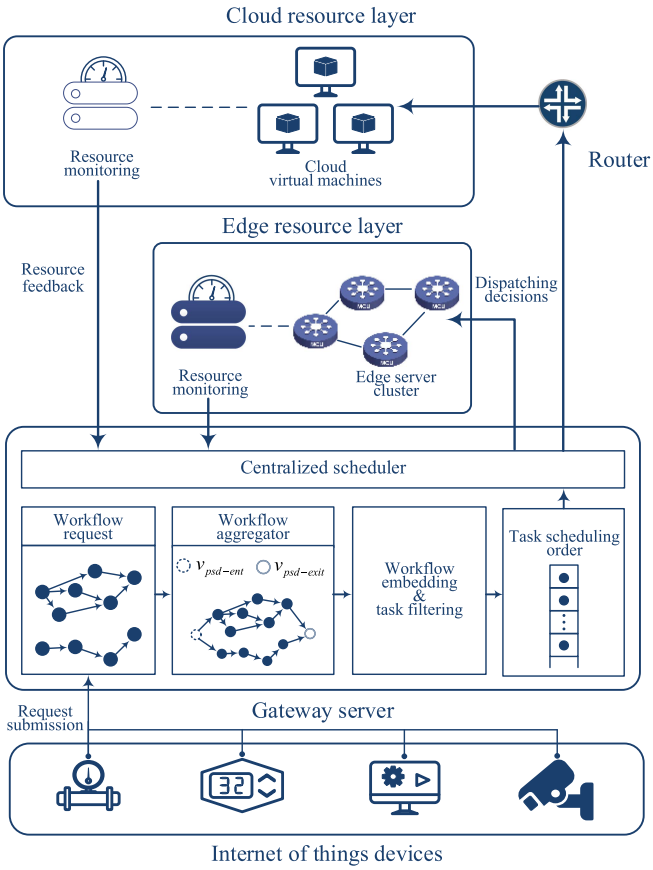

Fig. 1. System model.

the system model. During each timeslot, IoT infrastructures such as sensors and monitoring devices will continuously generate jobs in the form of workflows. Each of these jobs includes a series of tasks with dependency constraints expressed as a DAG. It is assumed that the computational resources required by each task, the data to be transmitted, and the workflow deadline are known in advance. Computationally expensive tasks are to be offloaded to the edge–cloud network for further execution through wired or wireless access points (APs). These APs are further connected with the edge resource layer through fiber links. The edge resource layer includes an edge computing network that consists of heterogeneous edge servers. The cloud resource layer includes cloud machines at a multihop distance from the edge computing layer. The cloud resource layer and the edge resource layer are also connected via optical fiber links. In this paper, assume that the computing capabilities of the edge and cloud servers are known in advance. The whole system complies with a master-slave work mode as suggested in [1]. A gateway server is deployed with a workflow scheduler acting as the master server while the rest of the edge servers are treated as slave servers. The master server is responsible for making scheduling decisions for each new arrival and remaining workloads. The slave servers share their resource availability (CPU, link throughput, workload) with the master server periodically to keep their information updated. Table II summarizes the definitions of the symbols used in this section.

TABLE II
SYMBOL TABLE

Symbol	Definition
I_t , T^{ep}	The current and total number of intervals
Job_t , G_t^{WF}	Tasks generated during interval I_t and the corresponding dependency graph
$\boldsymbol{V}_t$, v_t , K_t	The set of tasks, the exact task, and the number of available tasks in I_t
$\boldsymbol{E}_t$, $e_{i,j}$	The set and exact pair of edges
z_i , $data_{i,j}$	The required number of FLOPs for task v_i and the required data stream transmitted from task v_i to v_j
$\boldsymbol{N}^e$, $\boldsymbol{N}^c$	The set of edge and cloud servers
$\boldsymbol{\phi}$, $\varphi_{l,m}$, $B_{l,m}^t$	The set of links, the link between n_l and n_m , and the corresponding throughput
f_l	The computing capability of n_l
$\mathcal{T}_{\text{exe}}(v_i,n_l)$, $Energy_{exe}(v_i,n_l)$	The execution time and energy cost when the task v_i is executed on n_l
$\mathcal{T}_{trans}(v_i,v_s)$, $Energy_{trans}(v_i,v_s)$	The transmission time and energy cost between v_i and v_s
$AST(v_i,n_l)$, $AFT(v_i,n_l)$	The actual start and finish time of task v_i
$EST(v_i,n_l)$, $EFT(v_i,n_l)$	The earliest start and finish time of task v_i
Q_l , $Q_{\max}$	The number of tasks in the waiting queue of n_l and the maximum queue length of the servers

A. Workflow Model

Workflow scheduling in a heterogeneous network is generally treated as a discrete-time control problem. A workload is modeled as a timeline consisting of multiple equal-duration intervals denoted by I_t. During each interval, each IoT device generates its corresponding task, and these tasks must be processed in a specific order. We model the tasks generated during the same interval as workflow Job_t and we extract the task dependency graph as a DAG $G_t^{WF} = (\boldsymbol{V}_t, \boldsymbol{E}_t)$, where $\boldsymbol{V}_t$ denotes the set of tasks, represented as nodes, and $\boldsymbol{E}_t$ denotes the set of dependencies among the tasks, represented as directed edges. In this paper, we assume that the dependency graphs of the workflows generated in each interval can be varied, but the workload in itself follows a certain number of patterns based on their topology. Each edge $e_{i,j}$ in $\boldsymbol{E}_t$ indicates that task v_j can be processed only after its predecessor task v_i has been completed and the corresponding data have been completely transmitted. We introduce two functions for ease of notation: $succ(v)$ refers to the set of direct successors of task v, and $pred(v)$ refers to the set of direct predecessors of task v [16], [23]. Specifically, the task with no predecessor task is considered as the entry task, and the task with no successor task is noted as the exit task. In dealing with workflows generated by multiple IoT devices in a centralized manner, we can treat the workload

generated in the same interval as a whole by adding a pseudo entry task $v_{psd-ent}$ before the entry tasks, and a pseudo exit task $v_{psd-exit}$ after the exit tasks. The $v_{psd-ent}$ and $v_{psd-exit}$ are tasks with zero execution and transmission cost, and it is worth noting that this change will not influence the actual scheduling process. Moreover, z_i denotes the required number of floating-point operations (FLOPs) for task v_i, and $data_{i,j}$ denotes the necessary data stream transmitted from task v_i to v_j. In general, a successor task can be executed only if: 1) all of the direct predecessor tasks have finished and, 2) the necessary data from all of the direct predecessors have been received.

B. Network Model

Let $\boldsymbol{N}^e$ denote the set of edge servers and let $\boldsymbol{\phi}$ denote the set of links between edge servers. We consider an edge computing network consisting of N edge servers. For each edge server $n_l \in \boldsymbol{N}^e$, f_l is defined as the computing capability of the edge server measured in FLOPs/s, and Q_l is defined as the number of tasks in the waiting queue of n_l. For each link $\varphi_{l,m} \in \boldsymbol{\phi}$, we use $B^t_{l,m}$ (in bits/s) to reflect the throughput between edge servers n_l and n_m in the current interval. The throughput is assumed to remain constant within an interval but may be different between intervals.

The cloud server is abstracted as a cluster of cloud virtual machines (VMs) denoted by $\boldsymbol{N}^c$. In cloud computing, the computing capabilities of the cloud VMs (f_c) are much higher than those of the edge servers (f_l), but the cloud VMs suffer higher round-trip delays. Similarly, we use $B^t_{l,c}$ (in bits/s) to reflect the throughput between edge servers n_l and cloud server $\boldsymbol{N}^c$ in the current interval.

C. Delay Model

The delay of a task in an edge–cloud network is widely modeled as a combination of execution delay, queuing delay and transmission delay. The execution delay of a task v_i mainly depends on the required computational resources (z_i) and the computing resources of the chosen server (f_l) and is defined as follows:

$$\mathcal{T}_{\text{exe}}(v_i, n_l) = \frac{z_i}{f_l}, n_l \in \boldsymbol{N}^e \cup \boldsymbol{N}^c \tag{1}$$

Different from typical task offloading problems that consider only the execution of identical tasks, the execution of the tasks in a workflow follows certain dependency constraints as specified by the task dependency graph G^{WF}_t. This means that a task can be executed only after the results from all of its predecessor tasks have been received. In this paper, we ignored the overhead of the task transmission time from the IoT device to the servers for simplicity. The transmission delay from task v_i to its successor task $v_s \in succ(v_i)$is defined as follows:

$$\mathcal{T}_{trans}(v_i, v_s) = \begin{cases} 0, & if\ n_l = n_q \\ \frac{Data_{i,s}}{B^t_{l,q}}, & otherwise \end{cases} \tag{2}$$

where n_l and n_q ($n_l, n_q \in \boldsymbol{N}^e \cup \boldsymbol{N}^c$) denote the identification numbers of the servers on which v_i and v_s are deployed, respectively. If task v_i is offloaded to the same server as its successor task v_s, the corresponding data transmission time is zero.

D. Energy Consumption Model

The energy consumption during the execution of a workflow comprises the task execution cost and the transmission cost. The execution cost of a server is the combined energy cost of the power consumed for task execution and idle power and it is formulated as follows:

$$Energy_{exe}(v_i, n_l) = \mathcal{T}_{exe}(v_i, n_l) \times p_l^{active} + \mathcal{T}_{idle} \times p_l^{idle} \tag{3a}$$

$$\mathcal{T}_{idle} = I_t - \mathcal{T}_{exe}(v_i, n_l) \tag{3b}$$

where p_l^{active} and p_l^{idle} denote the power consumption rates when server n_l is active and idle respectively [1]. Meanwhile, the energy consumed during the transmission of the result is closely coupled with the deployment of the successor tasks and is defined in (4):

$$Energy_{trans}(v_i, v_s) = \mathcal{T}_{trans}(v_i, v_s) \times p_{trans}, v_s \in succ(v_i) \tag{4}$$

where p_{trans}denotes the transmission power.

Consequently, the energy consumption for task v_i and the total energy consumption for the whole workflow can be formulated in (5) and (6) respectively.

$$Energy(v_i, n_l) = Energy_{exe}(v_i, n_l) + \sum_{v_s \in succ(v_i)} Energy_{trans}(v_i, v_s) \tag{5}$$

$$Energy(Job_t) = \sum_{v_i \in \boldsymbol{V}_t} Energy(v_i, n_l) \tag{6}$$

E. Problem Definition

The initial goal of our work is to simultaneously minimize the makespan and energy cost of the workflows in an edge–cloud network. To provide a clear illustration of the problem, we present the formal definitions of the corresponding functions as follows.

The actual finish time of a task, denoted by $AFT(v_i, n_l)$, is defined as the time at which task execution actually ends for task v_i offloaded to server n_l [16], [24].

$$AFT(v_i, n_l) = AST(v_i, n_l) + \mathcal{T}_{exe}(v_i, n_l), n_l \in \boldsymbol{N}^e \cup \boldsymbol{N}^c \tag{7}$$

where $AST(v_i, n_l)$ denotes the actual start time for task v_i executed on server n_l.

The earliest start time of a task, denoted by $EST(v_i, n_l)$, is defined as the earliest time at which task v_i may begin if it is offloaded to server n_l.

$$EST(v_i, n_l)$$

$$= \begin{cases} 0, & if\ pred(v_i) \in \emptyset \\ \max\limits_{v_p \in pred(v_i)} [AFT(v_p, n_q) & \\ +\mathcal{T}_{trans}(v_p, v_i)], & otherwise \end{cases} \tag{8}$$

$$EFT\left(v_i, n_l\right) = EST\left(v_i, n_l\right) + \mathcal{T}_{exe}\left(v_i, n_l\right), n_l \in \boldsymbol{N}^e \cup \boldsymbol{N}^c \tag{9}$$

where n_q is the server on which $v_p \in pred(v_i)$ is deployed.

The makespan represents the completion time of a workflow and is formulated in (10).

$$makespan(Job_t) = \max_{succ(v_i) \in \emptyset} [AFT(v_i, n_l)] \tag{10}$$

With the execution information for all tasks, we can obtain the makespan of the workflow. Our goal is to find a feasible scheduling function $\mathcal{D}$ that can minimize the makespans and energy costs of the workflows generated by the IoT devices during a certain period. With T^{ep} defined as the number of intervals in an episode, we formulate the optimization problem as follows:

$$\begin{aligned} \boldsymbol{P}: \underset{\mathcal{D}}{\text{minimize}} & \sum_{t=0}^{T^{ep}} \alpha \times makespan(Job_t) \\ & + (1-\alpha) \times Energy(Job_t) \\ s.t. & Q_l \leq Q_{\max}, \forall t \in [0, T], \forall n_l \in \boldsymbol{N}^e \cup \boldsymbol{N}^c \end{aligned} \tag{11}$$

where α is a hyperparameter that adjusts the objective of the optimization problem, Q_l is the number of tasks in the waiting queue of n_l, and $Q_{\max}$ denotes the maximum queue length of the servers. The constraint guarantees that the number of tasks allocated to the edge servers will not exceed a certain limit.

Workflow scheduling in an edge–cloud network is defined here as follows: Given the necessary information of the workflows, namely, the 1) dependency graphs of the tasks, 2) the computing resources required for each task, and 3) the size of the data to be transmitted between tasks, as well as the necessary network information, namely, 1) the computing capability of each server and 2) the throughput between servers, workflow scheduling refers to dispatching the tasks in the workflows to edge/cloud servers to minimize the total time and energy consumed in processing the workload during a certain period.

Compared with previous works on workflow scheduling, the challenges we address here are two-fold:

- *Heterogeneous environment and queuing constraint:* the servers in a hybrid edge–cloud network are usually heterogeneous in computing capabilities and transmission throughput [16]. Different scheduling strategies result in different preferences in choosing the target servers, leading to different task completion times. We use a queuing constraint (limiting the number of tasks in the wait queue of edge servers) to avoid introducing extra queuing delay. However, this makes the problem even more complex. A task dispatching decision is treated as invalid if the waiting queue of the target server is full; thus, the task will need to wait for a new dispatching decision in the next turn. Therefore, the scheduler must not only to balance the demands of the tasks but also consider the current workload of the servers.
- *Continuous arrival of jobs:* in this paper, the workload considered is a series of workflows with continuous steady arrivals, i.e., the same workflow is generated following a $Poisson(\lambda)$ distribution during each interval. The impact of the remaining tasks and future incoming workflows are likely to compromise the performance of workflow schedulers designed for static workloads [16]. Together with the impact of the queuing constraint, the critical path among the workflows will no longer be fixed, placing a higher demand on the scheduler design. Consequently, an ideal workflow scheduler should also be sufficiently flexible to cope with a dynamic workload.

IV. The Proposed Framework

To minimize the makespans of the workflows and the energy costs in the edge–cloud network, the problems of the task scheduling order, heterogeneous capabilities of the servers, and offloading decisions for predecessor tasks should be carefully considered. Since new jobs will be generated at the beginning of each interval, an ideal workflow scheduler must cope with the parallelism of the jobs while avoiding a backlog of the workload.

To bridge the gap between the instantaneous reward obtained after each scheduling decision and the eventual makespans of the workflows, in this paper, we propose an intelligent scheduling agent called the intrinsic-reward-based workflow scheduler (IRWS). In this section, we present the design of the proposed IRWS. First, we describe the embedding method used to capture the latent information contained in the dependency graphs of the workflows and determine the task processing order. Then, we introduce the design of a typical PPO scheduler and discuss its inherent drawbacks in dealing with workflows. Finally, we propose a solution to address these drawbacks by employing an intrinsic reward as the target of the policy network and we introduce the structure of the proposed IRWS. Fig. 2 presents an overview of the proposed IRWS. Next, we describe the main components of the IRWS.

A. Workflow Embedding

Workflows with different patterns are subject to dependency constraints among their tasks. Different task dependencies lead to differences in the input dimensional parameters. Such dynamism will inevitably require additional parameters of a learning network, making the learning process even more complex. By employing graph embedding, however, the task representations can be mapped to fixed dimensions, which can effectively reduce the number of input parameters while improving the network training efficiency.

The proposed workflow embedding module consists of two phases, i.e., a task embedding phase and a task filtering phase. In the task embedding phase, the tasks in the current workloads are embedded into node-level vectors, while in the task filtering phase, the task scheduling order is determined. Fig. 3 shows an illustration of the workflow embedding process.

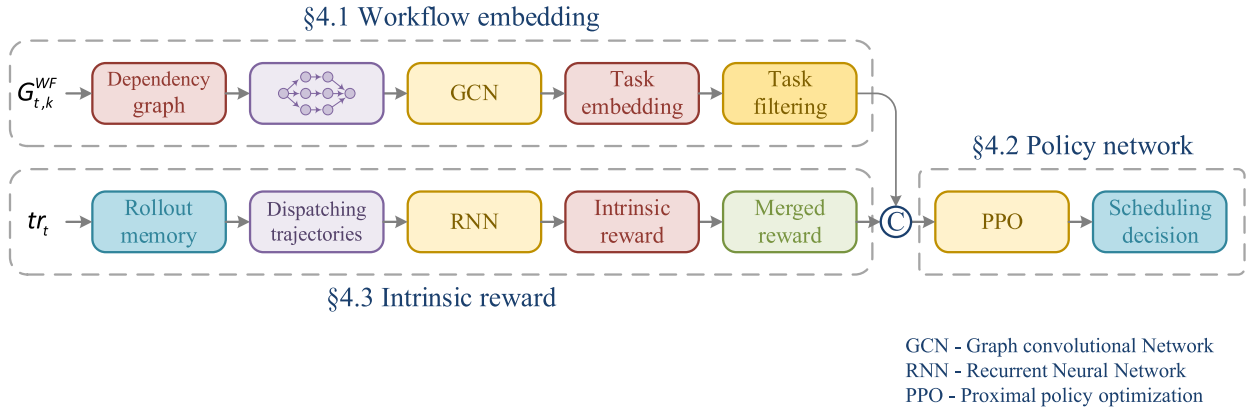

Fig. 2. Overview of the proposed IRWS scheme.

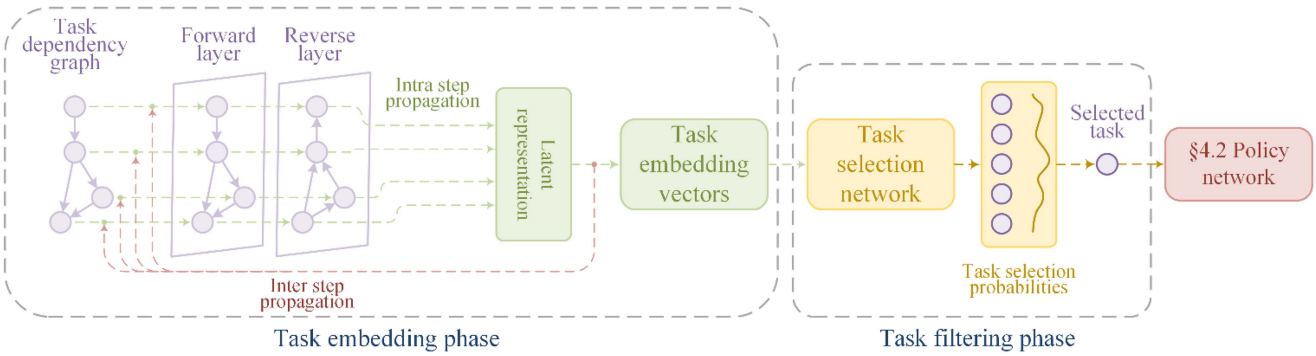

Fig. 3. An illustration of the workflow embedding process.

1) Task Embedding Phase: Graph-structured data has high data density and contains rich topological information. Simply integrating the features of the tasks (such as resource requirement and data volume) limits the representation ability of the proposed model. In dealing with complex workflows with varying numbers of tasks and varying workloads, it is crucial to know the structure of the dependency graph of each workflow from high-dimensional features. Moreover, to achieve better scalability, the scheduler must handle workflows of various forms [20]. Based on the prior hypotheses that the adjacent tasks in the workflow usually have the same features, we employ graph neural networks to supplement the topological dependencies between tasks. The proposed IRWS can preserve the dependency structure and achieve compatibility with diverse task dependency graphs by employing a GNN. The essence of graph neural networks can be considered as the aggregation process of neighboring nodes. In this paper, the attributes and dependency structure of tasks are combined and transformed into a unified embedding vector with fixed dimensions. Our work is a workflow-scheduling-specific extension of the graph convolutional neural networks proposed in [16], [20], [25].

The main idea of the task embedding phase is that the information propagates sequentially [26], and the proposed embedding process mimics the information flow. For every task $v \in \boldsymbol{V}$, we aggregate latent information from its direct predecessors and combine the aggregated information with its own attributes to update the hidden state.

Well-chosen task attributes should contain adequate information about the workflow while avoiding the introduction of redundant parameters. Metrics such as the upward rank $Rank_{up}$ and the downward rank $Rank_{dn}$ are commonly used to measure the importance of a node in a DAG [19]; these metrics are defined in (12) and (13) respectively. In this paper, we use these metrics as the attributes of the tasks.

$$Rank_{up}(v_i) = \max_{v_j \in succ(v_i)} \left(\frac{Data_{i,j}}{\overline{B}} + Rank_{up}(v_j) \right) + \frac{z_i}{\overline{f}} \tag{12}$$

$$Rank_{dn}(v_j) = \max_{v_i \in pred(v_j)} \left(\frac{Data_{i,j}}{\overline{B}} + Rank_{dn}(v_i) + \frac{z_i}{\overline{f}} \right) \tag{13}$$

where $\bar{f}$ is the average computation capability of the servers and $\bar{B}$ is the average throughput of the links.

Inspired by the previous applications in [26], [27], [28], we use the reversed graph to better perceive the structure of a workflow. Here, the reversed graph $\tilde{G}_t^{WF}$ is defined to have the same structure as G_t^{WF}, but the task execution order is inverted. For a given dependency graph G_t^{WF}, we use x^v and e_l^v to denote the attribute vector and the hidden state of task v in the l^{th}

step respectively. The task embedding process under $\mathcal{L}$ steps of message passing can be formulated as follows:

$$e_v^0 = \text{Tanh}(\omega_1^0 \times x^v + b) \tag{14}$$

$$e_v^\ell = \mathcal{F}^\ell\left(e_v^{\ell-1}, \mathcal{G}^\ell\left(\left\{e_u^\ell | u \in pred(v)\right\}, e_v^{\ell-1}\right)\right), \ell = 1, \ldots, \mathcal{L} \tag{15}$$

$$\begin{aligned} h_v^\ell &= \mathcal{G}^\ell\left(\left\{e_u^\ell | u \in pred(v)\right\}, e_v^{\ell-1}\right) \\ &= \text{Softmax}(\omega_1^\ell e_v^{\ell-1} + \omega_2^\ell e_u^\ell) \end{aligned} \tag{16}$$

$$\mathcal{F}^\ell\left(e_v^{\ell-1}, h_v^\ell\right) = Bi - GRU^\ell\left(e_v^{\ell-1}, h_v^\ell\right) \tag{17}$$

where ω and b denote the weight matrix and bias parameter respectively; $\mathcal{G}^\ell$ and $\mathcal{F}^\ell$ are the aggregation function and combination function respectively; and h_v^ℓ denotes the aggregated information. In our work, $\mathcal{G}^\ell$ aggregates the information from the past state and the states of the direct predecessors of the current task. The combination function $\mathcal{F}^\ell$ is designed to merge the aggregated information h_v^ℓ with the previous hidden state $e_v^{\ell-1}$. To obtain a higher-level view of the task, we further calculate a DAG-level embedding as (18) shows.

$$e_v^{DAG} = FC\left(mean - pooling(e_v^{\mathcal{L}} || e_v^{\mathcal{L}})\right) \tag{18}$$

2) Task Filtering Phase: In the task filtering phase, tasks are rearranged into an ordered queue based on their scheduling order. Let Ξ denote the set of tasks that are ready to be scheduled in the current interval. Following the design of the policy network recommended in [16], the task embedding vectors are processed through a feedforward neural network and a Softmax layer calculates the probability of selecting each task to be scheduled. The probability for each task is defined as:

$$P(v) = \frac{\exp\left(\mathfrak{Q}\left(e_v^{DAG}\right)\right)}{\sum_{u \in \Xi} \exp\left(\mathfrak{Q}\left(e_u^{DAG}\right)\right)} \tag{19}$$

where $\mathfrak{Q}(\centerdot)$ is a nonlinear scoring function that takes embedding vectors as input to calculate the relative importance of tasks.

B. PPO-Based Policy Network

The MDP is a framework designed to model the sequential decision-making procedure. It can be represented as a tuple (state, action, transition probability, and reward). The learning agent interacts with the environment through a sequence of actions under discretized time steps to maximize the long-term reward, while training its estimates of the transition probability according to the experiences. An ideal scheduling agent should balance the tradeoff between computation cost and communication cost [16]. However, finding the optimal solution to the original optimization problem $\mathbf{P}$ is still challenging, especially when the network scales up. Thanks to recent advances in DRL, deep neural networks (DNNs) are now widely employed in building workflow schedulers.

With K_t representing the number of available tasks during interval I_t, during each interval, a typical scheduler solves the task allocation problem through K_t steps of consecutive decisions. During each step, the scheduler assigns a selected task to a server and updates the status of the network. This procedure is iteratively performed until all the tasks have been assigned.

Given the embedding of the currently selected task, a DRL model can be used to train a workflow scheduling agent. In this section, we leverage the PPO algorithm [29] as the base scheme to train such an agent. We formulate the underlying DRL model as follows.

State: At the beginning of each step, all the servers update their respective states and share their information with the scheduler. Inspired by the work in [1], the state s_k in step k is a combination of the following properties:

- The embedding vector of the selected task v_k: $e_v^{\mathcal{L}}$.
- The computational resource requirement of task v_k: z_k.
- The current workload of each server, which comprises the number of tasks in the waiting queue of the server and their estimated execution time.
- The total computing capacity and bandwidth of each server.
- The size of the data to be transferred from each server to the destination server according to the direct predecessors of task v_k. When $pred(v_k) = \emptyset$, no data will be transferred to the destination server.

Action: The action $a_{t,k}$ in step k is the destination server to which the selected task will be offloaded.

Reward: In a typical DRL-based workflow scheduler, the reward is generally modeled as the improvement in the execution time and energy cost. We denote the reward of a single step by $r_{step}^{t,k}$ and define it as follows:

$$\begin{aligned} r_{step}^{t,k} = \alpha \times \frac{\left[EFT(v, \bar{n}) - EFT(v, n_{a_k})\right]}{EFT(v, \bar{n})} \\ + (1-\alpha) \times \frac{\left[Energy(v, \bar{n}) - Energy(v, n_{a_k})\right]}{Energy(v, \bar{n})} \end{aligned} \tag{20}$$

Where $\bar{n}$ indicates that the task will be executed on a pseudo edge server with the average computing capability and the average link throughput $\bar{B}$.

The basic concept of the PPO algorithm is based on the trust region policy optimization (TRPO) algorithm proposed in [30]. In this paper, the workflow scheduling agent is trained using the actor–critic version of the PPO algorithm, as recommended in [31], [32]. To approximate the policy π and the value function V simultaneously, we deploy actor and critic networks parameterized by θ and μ respectively [31]. The advantage function is defined as:

$$A_k = r_{step}^{t,k} + \gamma V^{\pi_\theta}(s_{k+1}; \mu) - V^{\pi_\theta}(s_k; \mu) \tag{21}$$

In contrast to a typical actor–critic network, the PPO algorithm learns to optimize a clipped surrogate objective function to update the actor network [31]. This assures the updated policy will not suffer from sudden change from the previous policy and guarantees a smoother policy update [32] without introducing further external constraints. Let $\xi_k(\theta)$ be the probability ratio and let θ_{old} represent the policy parameters before the update. The policy network is updated according to the following clipped loss function:

$$L^{clip}(\theta) = \mathbb{E}_k\left[\min\left\{\xi_k(\theta) A_k, clip\left(\xi_k(\theta), 1-\epsilon, 1+\epsilon\right) A_k\right\}\right] \tag{22}$$

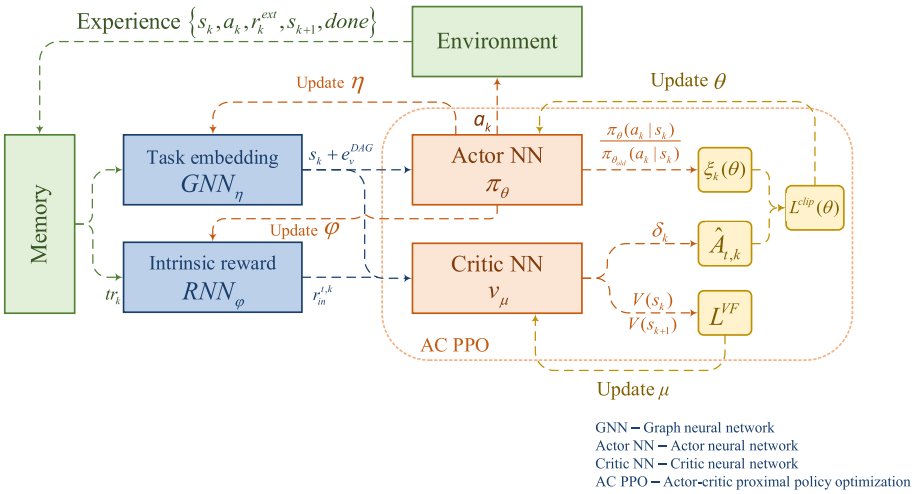

Fig. 4. Workflow of the proposed IRWS scheme.

$$\xi_k(\theta) = \frac{\pi_\theta(a_k|s_k)}{\pi_{\theta_{old}}(a_k|s_k)} \tag{23}$$

where ϵ is a hyperparameter, and we take $\epsilon = 0.2$ as recommended in [29]. The $clip(.)$ function limits the ratio $\xi_k(\theta)$ to within the range $[1-\epsilon, 1+\epsilon]$, which avoids the update step that may exceed the limit. The critic network, on the other hand, is trained to estimate the state-value function by minimizing the mean square error between the target state-value function and its prediction, which is defined as:

$$L^{VF}(\mu) = \mathbb{E}_k\left[\left(r_{step}^{t,k} + \gamma V^{\pi_\theta}(s_{k+1};\mu) - V^{\pi_\theta}(s_k;\mu)\right)^2\right] \tag{24}$$

However, the makespan of a workflow is not determined solely by the scheduling result of a single step but can also be affected by the dependency relations among the tasks. In other words, simply minimizing the execution time of a single task does not necessarily result in the minimum makespan of the workflow. Nevertheless, in most DRL-based workflow schedulers, only the task-level reward is considered. This can be attributed to two distinct characteristics of workflows:

- The task-level reward can be easily estimated whereas the ultimate makespan of a workflow can be obtained only after all tasks have been allocated.
- Since jobs are generated at the beginning of every interval, the makespan is influenced not only by the jobs generated in the same interval but also by the remaining and emerging jobs generated in the neighboring intervals.

C. Intrinsic Reward-Based Workflow Scheduler (IRWS)

Assigning tasks to the appropriate servers is the major concern of a workflow scheduler. However, the overall impact of the scheduling decisions on the makespan is revealed only after all tasks have been dispatched. In this paper, we seek to bridge this gap by introducing an intrinsic reward to estimate the latent impact of each step. Instead of concentrating on the task-wise reward as most DRL-based schedulers do, we propose to directly train the agents on the optimal policy that minimizes the objective function. Fig. 4 illustrates the proposed IRWS workflow, and we detail the design as follows.

1) Problem Reformulation: The original intent of workflow schedulers is to balance the tradeoff between the makespan and the energy cost. Considering the sparsity of the reward, compared with the original PPO algorithm, we leverage an intrinsic reward to guide the training process. In our proposal, we design our algorithm to learn the intrinsic rewards for the setting in which the underlying reinforcement learning agent is itself a learning agent, specifically, a policy-gradient-based learning agent [33]. Let $\mathcal{T}_v$ denote the wall clock terminal time [16], [20] of task v. Then, within the scope of a single workflow, the scheduler is designed to minimize the expected time-averaged penalty defined below:

$$J_t^{WF} = \mathbb{E}\left[\frac{1}{|\boldsymbol{V}|}\sum_{k=1}^{|\boldsymbol{V}|}[\alpha \times (\mathcal{T}_{v_k} - \mathcal{T}_{v_{k-1}}) + (1-\alpha) \times Energy(v_k, n)]\right] \tag{25}$$

The penalties for an episode and the whole training process can be generalized as follows:

$$J^{ep} = \mathbb{E}\left[\frac{1}{T}\sum_{t=1}^{T^{ep}} J_t^{WF}\right] \tag{26}$$

$$J^{life} = \mathbb{E}\left[\sum_{t=1}^{\infty}\gamma^t J_t^{ep}\right] \tag{27}$$

where γ is a discount factor. The original optimization problem $\boldsymbol{P}$ is thus reformulated to minimize the expected lifetime penalty J^{life}.

2) Intrinsic Reward: To solve the problem of estimating the latent impact of each step to the makespan of the whole workflow, we introduce an intrinsic reward to guide the learning process. By introducing the wall clock terminal time, which

contains the actual queuing time and the perturbation caused by neighboring tasks and can only be obtained after the actual execution of the task,J_t^{WF} is transformed into a direct reflection of the intermediate state of the workflow. We define the intrinsic reward as an estimate of a regularized cost component of J_t^{WF}, as follows:

$$r_{in}^{t,k} = -\left[\alpha \times \frac{t\left(v_k\right) - \left(\mathcal{T}_{v_k} - \mathcal{T}_{v_{k-1}}\right)}{t\left(v_k\right)} + \left(1-\alpha\right) \times \frac{Energy\left(v_k, \bar{n}\right) - Energy\left(v_k, n_k\right)}{Energy\left(v_k, \bar{n}\right)}\right] \tag{28}$$

$$t\left(v_k\right) = \frac{z_k}{\bar{f}} + \frac{Data_{j,k}}{\bar{B}} \tag{29}$$

By its definition, the intrinsic reward is evaluated from the scope of the whole workflow instead of from a single task. However, it is impossible to obtain the wall clock finish time of the task before its real execution. In this paper, we implement intrinsic reward function as a recurrent neural network (RNN) parameterized by φ, which takes the historical task dispatching trajectories $tr_t = (s_0, a_0, r_1, d_1, s_1, a_1, \ldots, r_t, d_t, s_t)$ as input and generates an estimate of the current state. Along with the task-dispatching process, the intrinsic reward is an accumulation of estimations on the intermediate states.

However, the estimation bias between the intrinsic rewards and real intermediate states accumulates along with the scheduling process, and the accumulated bias reaches its maximum at the end of the workflow. In contrast, the stepwise reward $r_{step}^{t,k}$ is an instant reflection of the current dispatching decision, the ultimate makespan can be more accurately evaluated by $r_{step}^{t,k}$ with increasing proximity to the final task. To minimize the accumulated bias, we modify the reward of the scheduling agent as an integration of $r_{in}^{t,k}$ and $r_{step}^{t,k}$ by adaptively changing the weights of these two rewards according to their "distance" to the end of the workflow. The designed reward and the advantage function are defined in (30) and (31) respectively.

$$r^{t,k} = \frac{t\left(v_k\right)}{Rank_{dn}\left(v_k\right)} \times r_{in}^{t,k} + \left(1 - \frac{t\left(v_k\right)}{Rank_{dn}\left(v_k\right)}\right) \times r_{step}^{t,k} \tag{30}$$

$$\hat{A}_{t,k} = r^{t,k} + \gamma V^{\pi_\theta}\left(s_{k+1}; \mu, \eta, \varphi\right) - V^{\pi_\theta}\left(s_k; \mu, \eta, \varphi\right) \tag{31}$$

where η is the hyperparameter of the workflow embedding network.

3) IRWS Learning Scheme: In this paper, we extended the PPO-based policy network, and we propose the IRWS algorithm. The proposed IRWS algorithm employs an intrinsic reward to estimate the intermediate state during the task dispatching phase to further facilitate the training of the scheduling agent.

a) Policy and Value Function Update: Along with the update rules in the vanilla PPO algorithm, by replacing the original advantage function A_k with $\hat{A}_{t,k}$ in (22) and (24), the process of updating the policy and value networks is defined as follows:

$$\theta' \leftarrow \underset{\theta}{\operatorname{argmax}}\, L^{clip}\left(\theta\right) \tag{32}$$

$$\mu' \leftarrow \underset{\mu}{\operatorname{argmin}}\, L^{VF}\left(\mu\right) \tag{33}$$

b) Intrinsic Reward Update: To update the intrinsic reward parameter, we adopt the meta-gradient ascent step proposed in [34], [35] over the objective function J^{life}. The corresponding gradient is derived by applying the chain rule as follows:

$$\nabla_\varphi J^{life} = \nabla_{\theta'} J^{life} \nabla_\varphi \theta' \tag{34}$$

The $\nabla_\varphi J^{life}$ can be regarded as an instance of meta-gradient methods [34], where the φ-gradient on $J^{life}$is defined by the φ-gradient on the parameter θ'. Its components are approximated as follows:

$$\nabla_{\theta'} J^{life} = \mathbb{E}_{\pi_{\theta_{old}}}\left[\sum_{i=t}^{\infty}\sum_{v}\gamma^{i-t} r^{i,k}\left[1\{\left|\xi_k\left(\theta\right)-1\right| < \epsilon or \right.\right.$$
$$\left.\left.\operatorname{sgn}\left(\xi_k\left(\theta\right)-1\right) \neq \operatorname{sgn}\left(\hat{A}_{t,k}\right)\}\right]\xi_k\left(\theta\right)\nabla_\theta \ln \pi_{\theta'}\left(a_{v,t}|tr_t\right)\hat{A}_{t,k}\right] \tag{35}$$

$$\nabla_\varphi \theta' = \nabla_\varphi\left[\theta + \alpha_{in}\sum_{i=t}^{\infty}\sum_{v}\gamma^{i-t} r^{i,k}\nabla_\theta \ln \pi_\theta\left(a_{v,t}|tr_t\right)\right]$$
$$= \alpha_{in}\frac{t\left(v\right)}{Rank_{dn}\left(v\right)}\nabla_\theta \ln \pi_\theta\left(a_{v,t}|tr_t\right)\nabla_\varphi r^{i,k} \tag{36}$$

where α_{in} is the learning rate of the intrinsic reward and tr_t represents the task dispatching trajectory.

c) Workflow Embedding Network Update: For the training of the workflow embedding network, we adopt another meta-gradient method that accommodates the aggregated reward.

$$\nabla_\eta r^{i,k} = \nabla_{\theta'} r^{i,k} \nabla_\eta \theta' \tag{37}$$

where the $\nabla_\eta r^{i,k}$derives similar regulations with the intrinsic reward, and the $\nabla_{\theta'} r^{i,k}$ and $\nabla_\eta \theta'$ are derived as follows:

$$\nabla_{\theta'} r^{i,k} = \mathbb{E}_{\pi_{\theta_{old}}}\left[\left[1\left\{\left|\xi_k\left(\theta\right)-1\right| < \epsilon or \right.\right.\right.$$
$$\left.\left.\left.\operatorname{sgn}\left(\xi_k\left(\theta\right)-1\right) \neq \operatorname{sgn}\left(\hat{A}_{t,k}\right)\right\}\right]\xi_k\left(\theta\right)\nabla_\theta \ln \pi_{\theta'}\left(a_{v,t}|s_t\right)\hat{A}_{t,k}\right] \tag{38}$$

$$\nabla_\eta \theta' = \alpha_{WE}\frac{t\left(v\right)}{Rank_{dn}\left(v\right)}\nabla_\theta \ln \pi_\theta\left(a_{v,t}|s_t\right)\nabla_\eta r^{i,k} \tag{39}$$

where α_{WE} is the learning rate of the workflow embedding network.

Algorithm 1 describes the proposed IRWS scheme. Despite the initialization of the network parameters and workflow generation (Line 1 to Line 4), we perform the workflow embedding to obtain the dispatching order of tasks (Line 5) and store the task dispatching trajectories according to $\pi_{\theta_{old}}$ (Line 7). Line 8 to Line 10 provide a mixed procedure for obtaining the experiences of PPO and the intrinsic reward. Then, Line 15 to Line 16 present

Algorithm 1: IRWS Learning Agent.

Input: Actor network π_θ and behavior actor network $\pi_{\theta_{old}}$ with $\theta_{old} \leftarrow \theta$; Critic network v_μ; Workflow embedding network $\mathcal{N}_{\boldsymbol{\eta}}$; Intrinsic reward network $\mathcal{N}_{\boldsymbol{\varphi}}$; Number of episodes, *EPISODE*; Number of intervals in an episode, T^{ep}; Clipping ratio ϵ; Times of the training rounds S at the end of each episode.

1: Initialize π_θ, v_μ, $\mathcal{N}_{\boldsymbol{\eta}}$, and $\mathcal{N}_{\boldsymbol{\varphi}}$;
2: **for** $episode = 1$ to $EPISODE$ **do**:
3: **for** $t = 1$ *to* T^{ep} **do**:
4: Generate workflow Job_t and update the task dispatching sequence;
5: Perform workflow embedding and generate the task dispatching sequence Ω_t in accordance with (19);
6: **for** v_k **in** enumerate (Ω_t):
7: Sample $a_{v,t}$ in accordance with $\pi_{\theta_{old}}(a_{v_k,t} \mid s_t)$ and obtain the rollout trajectories;
8: Receive the reward $r_{step}^{t,k}$, proceed to the next state s_{t+1}, and generate the intrinsic reward $r_{in}^{t,k}$;
9: Calculate the extrinsic reward $r^{t,k}$, $\xi_k(\theta)$ and the advantage estimate $\hat{A}_{t,k}$ in accordance with (30), (23) and (31), respectively;
10: Store the transition $\left(s_t, a_{v_k,t}, r^{t,k}, s_{t+1}\right)$ in memory $\boldsymbol{D}$;
11: **end for**
12: **for** $i = 1$ *to* S **do**:
13: Randomly extract a minibatch of samples with size Sze from memory $\boldsymbol{D}$;
14: **for** $j = 1$ *to* Sze **do**:
15: **Update** critic network v_μ : $\mu' \leftarrow \operatorname{argmin}_\mu L^{VF}(\mu)$;
16: **Update** actor network π_θ : $\theta' \leftarrow \operatorname{argmax}_\mu L^{clip}(\theta)$;
17: **Update** intrinsic reward network $\mathcal{N}_{\boldsymbol{\varphi}}^{\boldsymbol{\theta}}$ in accordance with (34);
18: **Update** workflow embedding network $\mathcal{N}_{\boldsymbol{\eta}}$ in accordance with (37);
19: **end for**
20: **end for**
21: $\theta_{old} \leftarrow \theta'$;
22: **Clear** memory $\boldsymbol{D}$;
23: **end for**
24: **end for**

the training process of the actor-network and the critic-network while the value function is replaced with the objective of the PPO. Finally, Line 17 and Line 18 are the training process of the intrinsic reward network and workflow embedding network.

V. Performance Evaluation

We conducted a series of simulations to evaluate the proposed IRWS and compare it with existing workflow scheduling algorithms.

A. Simulation Setup

In this paper, we present simulations conducted based on the COSCO framework [12], [36] to help assess the proposed scheduler. The COSCO framework is an integrated testbed designed to accelerate the development of edge–cloud network applications. In this study, we implemented the emulation with a trace-driven simulation model, in which no actual workloads were executed but the execution cost could be derived in accordance with the scheduling decisions. We considered a hybrid edge–cloud network with 30 edge servers and 20 cloud servers, with simulation testbed settings similar to those considered in [12]. In the simulation, workflows were continuously generated and scheduled for 1000 time slots. Table III summarizes the basic settings for the various parameters used. All of the simulations were performed on a server using an Intel Xeon Gold 5118 at a frequency of 2.3GHz, 16GB RAM, Tesla V100 GPU, and Ubuntu 18 OS.

TABLE III
Simulation Parameters

Environmental parameter	Value
Number of intervals per episode (T^{ep})	500
Hyperparameter of objective function (α)	0.5
Link throughput (B)	200 MB/s-250 MB/s
Response delay of edge/cloud	1 ms / 10 ms
Computational resource of edge server (f_e)	50 MIPS – 100 MIPS
Computational resource of cloud server (f_c)	1000 MIPS
Power of edge server (p_e^{idle}, p_e^{active})	Idle 10 W / Active 50 W
Power of cloud server (p_c^{idle}, p_c^{active})	Idle 50 W / Active 500 W
Network parameter	**Value**
Discount factor (γ)	0.2
Mini-batch size (Sze)	64
Learning rate ($\alpha_a, \alpha_c, \alpha_{in}, \alpha_{WE}$)	1e-5/1e-4/1e-5/1e-6
Optimizer	Adam
No. of embedding dimension	5
Trajectory length	16
No. of neurons in each hidden layer of Actor network	128 / 64
No. of neurons in each hidden layer of Critic network	64 / 64 / 32

B. Dataset

In our simulation for ablation analysis, we used the execution patterns of 3 typical applications, each containing 5 subtasks, as the benchmark workload. The resource requirements for each subtask, including CPU and bandwidth utilization, were extracted from traces collected in experiments on 10 servers [1]. At the beginning of each step, we generated workflows following a $Poisson(\lambda)$ distribution with $\lambda = 5$ to emulate real-world deployment, as recommended in [12]. These workloads are randomly chosen from the above mentioned 3 patterns of workflows.

Then, for the comparative analysis, we chose five typical types of workflows proposed in Pegasus [37], [38] and the GitHub project [39] to further evaluate the feasibility of the IRWS in different circumstances.

C. Performance Evaluation Metrics

To better assess the performance of the different schedulers, we used average response time, average migration time, and average energy cost of all executed workflows as the evaluation

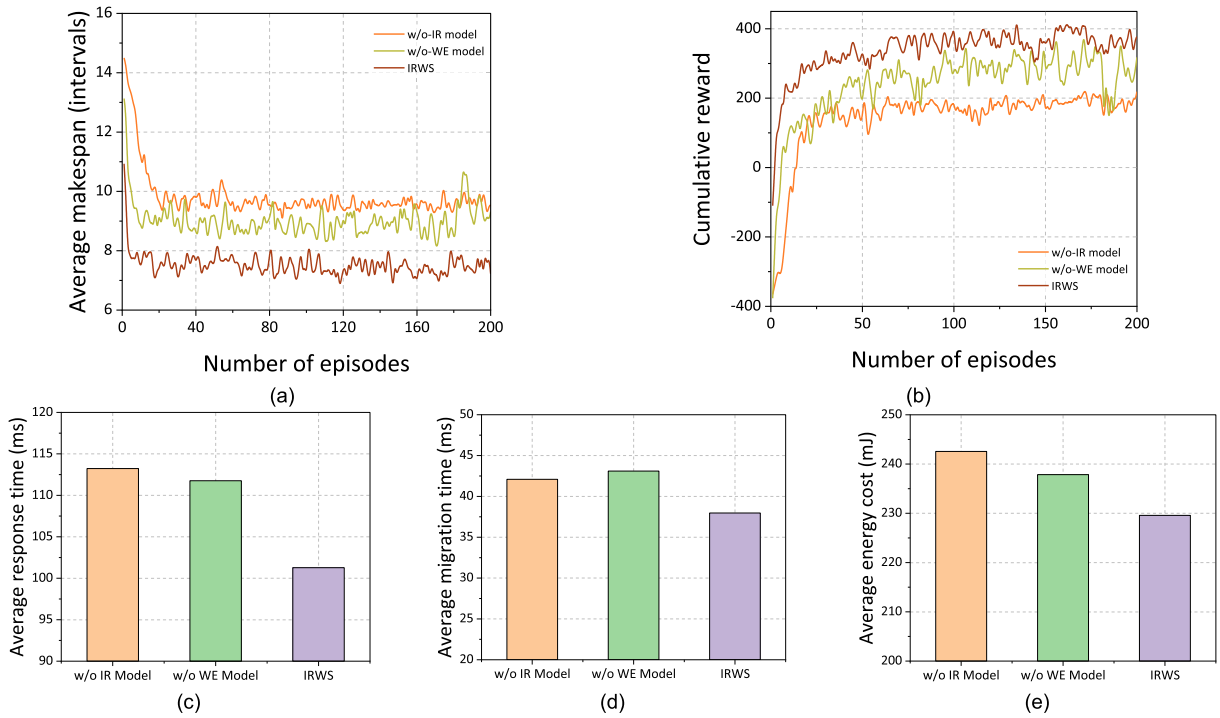

Fig. 5. Comparison of various aspects of performance (a) learning curve for the average makespan; (b) learning curve for the cumulative reward; (c) average response time; (d) average migration time; and (e) average energy cost.

metrics. The average response time is the average wall clock time consumed from generation to the termination of a workflow. The average migration time refers to the time consumed during data transmission. The average energy cost measures the energy cost by the cloud and edge servers. These metrics can demonstrate the performance of a scheduler in terms of different aspects of the objective function.

D. Ablation Analysis

To verify the impact of each component of the proposed IRWS algorithm on its performance, we removed every major component of the proposed model in turn and evaluated the performance of the resulting schedulers. We present the results in Fig. 5. First, we removed the intrinsic reward from the IRWS, meaning that only the pre-trained PPO model was used to obtain the output (w/o-IR model). Then, we executed the IRWS without the embedding of tasks, i.e., only the intrinsic reward was implemented in the proposed scheduler (w/o-WE model). We trained the IRWS, the w/o-IR model and the w/o-WE model for 200 episodes each, and all agents were trained using cosine annealing. For a better illustration of the training process, we further reveal the learning curve of the average makespan (total intervals consumed by the workflow) and cumulative reward (the accumulation of the merged reward $r^{t,k}$).

From Fig. 5(a) and (b), we observe that the IRWS, w/o-IR model and w/o-WE model converge at different rates. However, the IRWS achieves better performance in terms of both the average makespan and the cumulative reward. This result confirms that the proposed solution can better capture the structural information of the dependency graphs and train the agent at a higher efficiency. Notably, regarding the w/o-IR and w/o-WE models, although both agents converge with an increasing number of iterations, the w/o-WE model still achieves better performance than the w/o-IR model. This implies that the intrinsic reward makes a greater contribution to improving the scheduler performance. Meanwhile, the w/o-WE model suffers the highest disturbance, which can be interpreted to mean that the algorithm that relies on the intrinsic reward alone must pay a heavier price to perceive the current workload and adjust the policies. Fig. 5(c)–(e) further demonstrate the effectiveness of the IRWS in terms of response time, migration time, and energy cost. The IRWS achieves a better performance for all three metrics than the w/o-IR model or the w/o-WE model. A comparison between the results achieved with the workflow embedding strategy (w/o-IR model) and the intrinsic reward (w/o-WE model) further supports our findings regarding the performance of the intrinsic reward.

E. Experimental Results

To verify the effectiveness of the proposed IRWS scheme, we conducted three experiments concerning the scalability, generality and elasticity of the scheduler. The simulations were implemented on the same hardware configurations with that of the ablation analysis. We compare the results obtained based on five real-world workflows with those of state-of-the-art scheduling algorithms such as HAC-PPO [1], MCDS [12], GOBI [36] and DRL. Furthermore, we have compared the performance of the schedulers with some typical scheduling

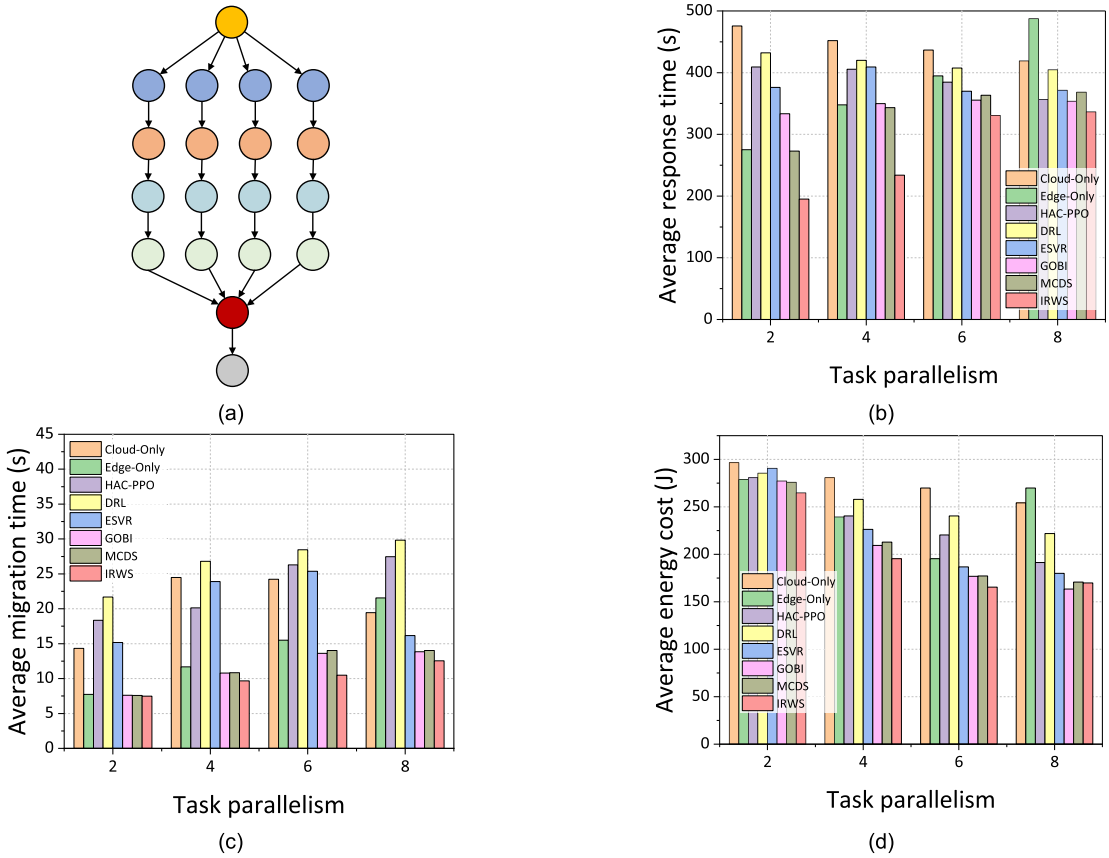

Fig. 6. Comparison of performance under different levels of task parallelism (a) an illustration of EpiGenomics; (b) average response time; (c) average migration time; and (d) average energy cost.

strategies such as ESVR [40], Cloud-Only (IMPSO [41]) and Edge-Only.

HAC-PPO [1]: a PPO-based workflow scheduling algorithm that focuses on optimizing the response time and energy cost of subtasks. The scheduling agent integrates the Actor Critic framework with PPO update rules and trains two independent action networks to generate hierarchical scheduling decisions. Among them, the top-level network determines whether the task is uninstalled to the edge server or cloud server, while the low-level network selects the exact server to which the task is assigned.

MCDS [12]: a metaheuristic-based workflow scheduling algorithm, which employs a decision tree-based search strategy and a deep neural network-based surrogate model to evaluate the long-term impact on QoS of immediate actions of scheduling decisions.

GOBI [36]: a gradient based workflow scheduler that aggregates the output of graph embedding, scheduling decision and time series window networks and update the respective neural network according to the back propagation of the gradients.

DRL: a deep reinforcement learning-based workflow scheduler with the vinalla Actor-Critic network.

ESVR [40]: a heterogeneous earliest finish time (HEFT) based workflow scheduling algorithm where the scheduling order of the tasks is determined based on topological sorting. The dispatching decisions are optimized according to the traces and the features of the hosts.

Cloud-Only: all of the workflows are offloaded to the cloud server for further execution. The immune-based particle swarm optimization (IMPSO) [41] is employed to make the scheduling decision in the cloud and use candidate affinity to refine the learning process.

Edge-Only: uses the same scheduling strategy as IRWS but the subtasks in the workflows will only be offloaded to the edge server. This means that no cloud server is utilized even though the wait queue will increase.

a) Scalability analysis

To evaluate the scalability of the workflow scheduling algorithms, the scientific workflow EpiGenomics [38], shown as Fig. 6(a), was employed as the benchmark workload. In the simulation, we compared the performance of the schedulers under different levels of task parallelism, that is, by deliberately varying the number of branches of the original workload while keeping the total number of tasks.

As Fig. 6(b) shows, the average response time of the schedulers increases with an increasing level of task parallelism. However, although the schedulers designed for task-level returns (HAC-PPO, DRL, ESVR and Cloud-only) suffer less performance degradation as the task parallelism grows, those designed for workflow-level rewards

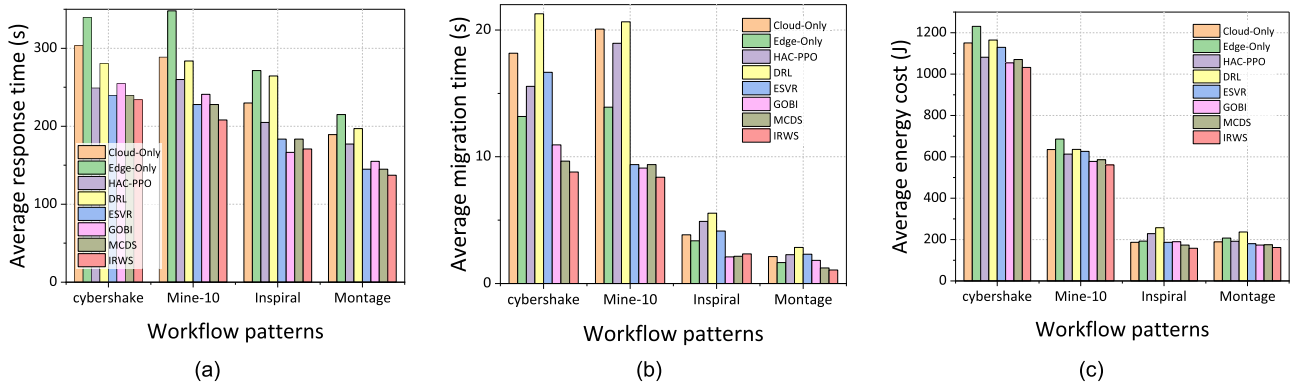

Fig. 7. Comparison of performance under different workflow patterns (a) average response time; (b) average migration time; and (c) average energy cost.

(GOBI, MCDS, Edge-Only and IRWS) still outperform the former in terms of average response time. The proposed IRWS achieves 28% and 37.7% improvement in terms of average response time and average migration time respectively. As the task parallelism grows, we note a sharp rise in the average response time when the task parallelism increases from 4 to 6. The IRWS reaches its highest difference in average response time of 96.7 seconds, which can be interpreted as the introduction of a higher queuing delay when more tasks are executed concurrently. Furthermore, when these findings are combined with those on the average migration time shown in Fig. 6(c), we note that the queuing constraint is more likely to be reached with more tasks in execution; thus, a task is less likely to be offloaded to the same server as its predecessors. Regarding the energy consumption shown in Fig. 6(d), the IRWS achieves an average of 14.6% improvement. However, when the task parallelism level is 8, the GOBI scheduler consumes 163.4 Joule compared with IRWS scheduler which consumes 169.8 Joule. This is assumed to be a result of the tradeoff between the response time and energy cost. Concessions in energy cost must be made when computational resources are scarce. We can further verify these findings by comparing with the offloading strategies of Cloud-only and Edge-Only. With the growth of the task parallelism, the performance of the Edge-Only strategy suffers obvious degradation due to more tasks being involved in the contention on edge servers concurrently. In contrast, the Cloud-only strategy benefits from the increasing parallelism, which can be explained by the relatively abundant resources at the cloud server side.

b) Generality analysis

Next, in Fig. 7, we further verify the generality of the IRWS algorithm. We compare the response time, migration time, and energy cost of the workflows against the four typical scientific workflows namely, CyberShake [38], Mine-10 [39], Inspiral [38], and Montage [38].

Fig. 7(a) and (b) show the average response times and migration times respectively for the different workflows. Among the baselines, GOBI obtains the lowest response (166.4 seconds) and migration times (2.1 seconds) for Inspiral whereas the IRWS yields an average improvement of 17.9% and 36.3% for the average response time and the average migration time respectively. Compared with the task-level schedulers (DRL, HAC-PPO, ESVR and Cloud-Only), most workflow-level schedulers (GOBI, MCDS and IRWS) except for the Edge-Only strategy achieve significant time reductions for all four workflows. This is because the objective of the latter is to optimize the long-term reward of the workflows while the former schedulers focus more on the consequences of the dispatching of single tasks. Therefore, the workflow-level schedulers are better at managing the balance between inter- and intra-workload interferences. Regarding the energy costs illustrated in Fig. 7(c), we observe that the three workflow-level schedulers achieve similar energy costs for all of the four workflow patterns, but the IRWS shows an average improvement of 2.8%, 3.6%, 12.8% and 7.2% for Cybershake, Mine-10, Inspiral, and Montage respectively. This further demonstrates that the intrinsic reward enables the proposed scheduler to better perceive the current workload status and reduce the energy cost in the long run. It is worth noting that, even with fewer available resources, the average migration time of the Edge-Only strategy is still less than half of the baselines, indicating that the IRWS-based strategy can effectively reduce the data transmission among edge servers. However, the limited resources result in a higher queueing delay, and the performance of the Edge-Only strategy is less effective in terms of response time and energy cost.

c) Elasticity analysis

To further evaluate the elasticity of the workflow scheduling algorithms, we chose the same scientific workflow EpiGenomics [38] as in the scalability analysis, and we kept the task parallelism to 4 as the benchmark workload. In the simulation, we compared the performance of the schedulers as the rate of workload arrival varies.

Fig. 8(a) and (b) show the average response times and migration times respectively for the different workload arrival rates. The response time and migration time increase when the workload grows heavier. In most cases, the workflow-level schedulers (GOBI, MCDS, and IRWS, except for Edge-Only strategy) obtain better performance than those designed for task-level returns (DRL, HAC-PPO, ESVR, and Cloud-Only). Among these schedulers, the proposed IRWS scheduler achieves

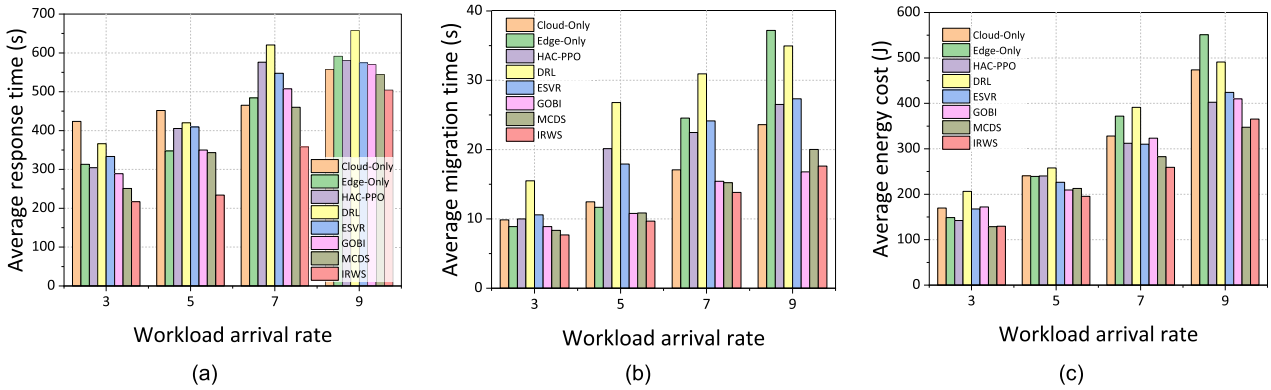

Fig. 8. Comparison of performance under different workload arrival rates (a) average response time; (b) average migration time; and (c) average energy cost.

28.8% and 22.4% improvement for the average response time and the average migration time respectively. However, when the workload arrival rate is 9, the average migration time for IRWS yields 17.6 seconds compared with GOBI's average migration time which is 16.8 seconds. Limited by the scale of the designed neural networks, the proposed IRWS scheduler may be inferior to the heuristics-based workflow scheduling methods (GOBI and MCDS) when the workload is heavy. This can be further verified for the energy costs illustrated in Fig. 8(c). Among the baselines, the MCDS scheduler obtains the lowest energy cost when the workload arrival rate is 9 (347.4 Joule for MCDS versus 365.5 Joule for IRWS), while the IRWS still performs an average improvement of 17.7% for different workloads. Furthermore, by comparing the performance between the IRWS and the Edge-Only strategy, we observe an increased difference both for the response time and the energy cost when the workload arrival increases, which further demonstrates that the scheduling algorithm is less effective if the scale of edge servers is inadequate.

VI. Conclusion

In this paper, we focus on workflow scheduling in a hybrid edge–cloud network with continuous steady task arrivals. Our goal is to simultaneously minimize the makespan of the workflows and optimize the energy efficiency. Here, we leverage workflow embedding to better perceive and understand the task dependency graphs and determine the task scheduling order. To cope with the lagging effect of the workflows, we designed an intelligent workflow scheduler, IRWS, which can be trained directly toward the intended design. In our proposal, we introduce the intrinsic reward as the major learning objective, which functions as the major indicator for evaluating the task dispatching decisions in a single scheduling step. Numerical simulation results obtained confirm that, with a well-trained agent, the proposed scheduler can intelligently dispatch a dynamic workload and can provide a high-performance solution.

However, we also note that the IRWS is a centralized scheduler, thus requiring a global view of the network structure and link status. These strict constraints constitute a major barrier hindering its further extension in practice, especially in large-scale networks. A feasible alternative would be to develop a distributed scheduler under a partially observable environment, and we will explore this area in our future work.

Acknowledgments

The authors thank the reviewers for their valuable comments which helped us to improve the content, organization, and presentation of this paper.

References

[1] A. Jayanetti, S. Halgamuge, and R. Buyya, "Deep reinforcement learning for energy and time optimized scheduling of precedence-constrained tasks in edge–cloud computing environments," *Future Gener. Comput. Syst.*, vol. 137, pp. 14–30, 2022.

[2] F. Jalali, K. Hinton, R. Ayre, T. Alpcan, and R. S. Tucker, "Fog computing may help to save energy in cloud computing," *IEEE J. Sel. Areas Commun.*, vol. 34, no. 5, pp. 1728–1739, May 2016.

[3] C. Richardson, "Pattern: Microservice architecture [EB/OL]," Microservice Architecture," 2017, Accessed: 2023. [Online]. Available: http://microservices.io/patterns/microservices.html

[4] Q. Qu, R. Xu, S. Y. Nikouei, and Y. Chen, "An experimental study on microservices based edge computing platforms," in *Proc. IEEE Conf. Comput. Commun. Workshops*, 2020, pp. 836–841.

[5] A. Samanta and J. Tang, "Dyme: Dynamic microservice scheduling in edge computing enabled IoT," *IEEE Internet Things J.*, vol. 7, no. 7, pp. 6164–6174, Jul. 2020.

[6] Y. Xie et al., "A novel directional and non-local-convergent particle swarm optimization based workflow scheduling in cloud–edge environment," *Future Gener. Comput. Syst.*, vol. 97, pp. 361–378, 2019.

[7] T. Dhand, "A unified framework for service availability and workflow scheduling in edge computing environment," M.S. thesis, Dept. Comput. Sci. Eng., Thapar Univ., Patiala, India, 2017.

[8] Y. Qin et al., "An energy-aware scheduling algorithm for budget-constrained scientific workflows based on multi-objective reinforcement learning," *J. Supercomputing*, vol. 76, no. 1, pp. 455–480, 2020.

[9] J. Sun et al., "Makespan-minimization workflow scheduling for complex networks with social groups in edge computing," *J. Syst. Archit.*, vol. 108, 2020, Art. no. 101799.

[10] H. Cai, V. W. Zheng, and K. C. Chang, "A comprehensive survey of graph embedding: Problems, techniques, and applications," *IEEE Trans. Knowl. Data Eng.*, vol. 30, no. 9, pp. 1616–1637, Sep. 2018.

[11] Y. S. Chang et al., "An agent-based workflow scheduling mechanism with deadline constraint on hybrid cloud environment," *Int. J. Commun. Syst.*, vol. 31, no. 1, 2018, Art. no. e3401.

[12] S. Tuli, G. Casale, and N. R. Jennings, "MCDS: AI augmented workflow scheduling in mobile edge cloud computing systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 11, pp. 2794–2807, Nov. 2022.

[13] J. Clark and D. Amodei, "Faulty reward functions in the wild [EB/OL]," 2016. [Online]. Available: https://blog.openai.com/faulty-reward-functions

[14] Z. Zheng et al., "What can learned intrinsic rewards capture?," in *Proc. Int. Conf. Mach. Learn.*, 2020, pp. 1–11.

[15] T. C. Mowry, "CS745 optimizing compilers - instruction scheduling: List Scheduling," 2003.
[16] Y. Zhou, X. Li, J. Luo, M. Yuan, J. Zeng, and J. Yao, "Learning to optimize DAG scheduling in heterogeneous environment," in *Proc. 23rd IEEE Int. Conf. Mobile Data Manage.*, Paphos, Cyprus, 2022, pp. 137–146.
[17] R. Garg and A. K. Singh, "Multi-objective workflow grid scheduling using ε-fuzzy dominance sort based discrete particle swarm optimization," *J. Supercomputing*, vol. 68, no. 2, pp. 709–732, 2014.
[18] S. Darbha and D. P. Agrawal, "Optimal scheduling algorithm for distributed-memory machines," *IEEE Trans. Parallel Distrib. Syst.*, vol. 9, no. 1, pp. 87–95, Jan. 1998.
[19] H. Topcuoglu, S. Hariri, and M. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 3, pp. 260–274, Mar. 2002.
[20] H. Mao et al., "Learning scheduling algorithms for data processing clusters," in *Proc. ACM Special Int. Group Data Commun.*, pp. 270–288, 2019.
[21] P. Sun et al., "Deepweave: Accelerating job completion time with deep reinforcement learning-based coflow scheduling," in *Proc. 29th Int. Conf. Int. Joint Conferences Artif. Intell.*, 2021, pp. 3314–3320.
[22] M. Chowdhury and I. Stoica, "Coflow: A networking abstraction for cluster applications," in *Proc. ACM Workshop Hot Topics Netw.*, 2012, pp. 31–36.
[23] C. Zhang et al., "Learning to dispatch for job shop scheduling via deep reinforcement learning," in *Proc. Adv. Neural Inf. Process. Syst.*, 2020, pp. 1621–1632.
[24] "What does actual finish date mean project management dictionary of terms [EB/OL]," Aug. 19, 2023. [Online]. Available: https://www.stakeholdermap.com/project-dictionary/actual-finish-date-meaning.html
[25] V. Thost and J. Chen, "Directed acyclic graph neural networks," in *Proc. Int. Conf. Learn. Representations*, 2021, pp. 1–16.
[26] S. Amizadeh, S. Matusevych, and M. Weimer, "Learning to solve circuit-SAT: An unsupervised differentiable approach," in *Proc. Int. Conf. Learn. Representations*, 2018, pp. 1–14.
[27] M. C. D. P. Kaluza, S. Amizadeh, and R. Yu, "A neural framework for learning DAG to DAG translation," in *Proc. Int. Conf. Neural Inf. Process. Syst. Workshop*, 2018, pp. 1–7.
[28] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2014, pp. 3104–3112.
[29] J. Schulman et al., "Proximal policy optimization algorithms," 2017, *arXiv:1707.06347*.
[30] J. Schulman et al., "Trust region policy optimization," in *Proc. Int. Conf. Mach. Learn.*, 2015, pp. 1889–1897.
[31] H. Wu, A. Nasehzadeh, and P. Wang, "A deep reinforcement learning-based caching strategy for IoT networks with transient data," *IEEE Trans. Veh. Technol.*, vol. 71, no. 12, pp. 13310–13319, Dec. 2022.
[32] H. Lim et al., "Federated reinforcement learning for training control policies on multiple IoT devices," *Sensors*, vol. 20, no. 5, 2020, Art. no. 1359.
[33] Z. Zheng, J. Oh, and S. Singh, "On learning intrinsic rewards for policy gradient methods," in *Proc. Adv. Neural Inf. Process. Syst.*, 2018, pp. 4649–4659.
[34] B. Fang et al., "Meta proximal policy optimization for cooperative multi-agent continuous control," in *Proc. Int. Joint Conf. Neural Netw.*, 2022, pp. 1–8.
[35] C. C. Hsu, C. Mendler-Dünner, and M. Hardt, "Revisiting design choices in proximal policy optimization," 2020, *arXiv:2009.10897*.
[36] S. Tuli, S. R. Poojara, S. N. Srirama, G. Casale, and N. R. Jennings, "COSCO: Container orchestration using co-simulation and gradient based optimization for fog computing environments," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 1, pp. 101–116, Jan. 2022.
[37] N. Anwar and H. Deng, "A hybrid metaheuristic for multi-objective scientific workflow scheduling in a cloud environment," *Appl. Sci.*, vol. 8, no. 4, 2018, Art. no. 538.
[38] E. Deelman et al., "Pegasus, a workflow management system for science automation," *Future Gener. Comput. Syst.*, vol. 46, pp. 17–35, 2015.
[39] R. A. Rezaeian, "Multi-workflow scheduler [EB/OL]," 2018. [Online]. Available: https://github.com/ralthor/Scheduler
[40] T.-P. Pham and T. Fahringer, "Evolutionary multi-objective workflow scheduling for volatile resources in the cloud," *IEEE Trans. Cloud Comput.*, vol. 10, no. 3, pp. 1780–1791, Third Quarter 2020.
[41] P. Wang, Y. Lei, P. R. Agbedanu, and Z. Zhang, "Makespan-driven workflow scheduling in clouds using immune-based PSO algorithm," *IEEE Access*, vol. 8:pp. 29281–29290, 2020.

Kaige Zhu received the MS degree in electrical & communication engineering from Beijing Jiaotong University (BJTU), Beijing, China, in 2018. He is currently working toward the PhD degree with the School of Electronic and Information Engineering, Beijing Jiaotong University. His research interests include edge computing and machine learning techniques.

Zhenjiang Zhang received the PhD degree in communication and information systems from Beijing Jiaotong University. From 2008 to 2013, he was an associate professor with the Beijing Jiaotong University. He has been a professor in BJTU since 2013. He is currently served as the director of the Institute of Intelligent Network and Information Security in BJTU. His research interests include cognitive radio, wireless sensor networks, and edge computing. He has been the guest editor of a number of journals, including IET communications, sensors, and international journal of distributed sensor networks, etc.

Sherali Zeadally (Senior Member, IEEE) received the bachelor's degree in computer science from the University of Cambridge, U.K., and the PhD degree in computer science from the University of Buckingham, U.K. He completed his postdoctoral research with the University of Southern California, Los Angeles, CA, USA. He is currently a professor with the College of Communication and Information, University of Kentucky. His research interests include cybersecurity, privacy, the Internet of Things, computer networks, and energy-efficient networking. He is a fellow of the British Computer Society and the Institution of Engineering Technology, U.K. He has received numerous awards and honors for his research and teaching.

Feng Sun received the bachelor's degree from the School of Electronic and Information Engineering, Beijing Jiaotong University, in 2017, where he is currently working toward the PhD degree in communication engineering. His research interests include edge computing and vehicular network.