Edge Cloud Resource Scheduling with Deep Reinforcement Learning

Youling FENG, Mengzhao LI, Jun LI, Yawei YU

School of Management Science and Information Engineering, Jilin University of Finance and Economics, Changchun 130117, China

{limengzhaolihai,cheeseyu}@foxmail.com, {fengyouling, lijun}@jlufe.edu.cn

Submitted November 10, 2024 / Accepted January 16, 2025 / Online first March 18, 2025

Abstract. *Designing optimal scheduling algorithms for task allocation in edge cloud clusters presents significant challenges due to the constantly changing workloads and service requests in edge cloud data center environments. These challenges stem from the need to manage the vast amounts of information transmitted by IoT devices, as well as the necessity of offloading computational tasks to cloud data centers. To tackle this issue, we propose a novel deep reinforcement learning-based resource allocation method called Decima#, which offers an effective resource optimization solution for edge cloud data centers. We utilize a transformer architecture to capture resource states on directed acyclic graphs (DAGs), accelerating the aggregation speed of the Graph Neural Network (GNN). Moreover, we develop innovative reward functions and concurrent processing mechanisms to minimize training time. Furthermore, we enhance the Proximal Policy Optimization (PPO) algorithm to improve adaptability, increase the accuracy of likelihood ratio estimation, identify a more suitable activation function, and impose constraints on gradient updates. In simulation environments, Decima# reduced the average job duration by 19% compared to the Decima algorithm, while also achieving a 56% increase in training convergence speed. Code has been made available at https://github.com/limengzhaolihai/spark-decimasharp-ppog.*

Keywords

Edge computing, deep reinforcement learning, resource scheduling

1. Introduction

In recent years, with the continuous growth in resource demand, an increasing number of enterprises and organizations have opted to migrate applications and data to cloud data centers, making the transition to edge data centers a significant trend. However, traditional cloud computing models rely on centralized computing architectures, necessitating the transfer of data and applications from Internet of Things (IoT) devices to remote central cloud servers for processing and computation, followed by the return of results to the devices. This model presents challenges such as high latency and substantial network load. In response, edge computing has emerged as an extension of core cloud computing technologies, establishing a form of distributed computing on edge devices that facilitates the offloading of computational power to the network's edge. By being positioned closer to the data source, edge computing not only optimizes bandwidth utilization but also enhances service quality, reduces network traffic, lowers latency, and improves user experience. Furthermore, edge cloud computing integrates the advantages of both edge and cloud computing, enabling the decentralization of resources and computational capabilities to the edge network. This approach facilitates smarter and more efficient resource allocation through centralized management and scheduling, effectively meeting low-latency requirements while providing greater flexibility to support complex application scenarios.

Resource management issues have become a key research area in data centers, involving virtual machine management [1], [2], resource scheduling and load balancing [3–6], as well as task allocation [7], [8]. Resource scheduling directly impacts the real-time allocation of resources and the efficiency of task execution, making it a crucial factor in enhancing overall system performance. The efficiency and reliability of task allocation are central to improving the effectiveness of resource scheduling. Currently, one of the most significant challenges in resource scheduling is to efficiently allocate tasks, maximize system resource utilization, and ensure both the effectiveness and reliability of task execution. Traditionally, task allocation is managed through algorithms such as fair scheduling, round-robin scheduling, or heuristic scheduling. However, with the wide application of machine learning techniques in resource scheduling, the research in this area has been diversified. Different machine learning techniques such as bionic solutions [9], LSTM-based solutions [10], Q-learning based solutions [11] and deep RL-based solutions [12], [13], all play a role in task processing. Currently, one of the most critical challenges in

DOI: 10.13164/re.2025.0092

resource scheduling is how to efficiently allocate tasks, maximize system resource utilization, and ensure the effectiveness and reliability of task execution. The efficiency and reliability of task allocation is the key to the utilization of system resources. Deep reinforcement learning's ability to automatically discover and learn optimal resource management strategies through environment interaction is highly beneficial in resource scheduling. Therefore, the main goal of this paper is to further explore and enhance task assignment by applying deep reinforcement learning techniques.

Among the case of resource scheduling solved by applying deep reinforcement learning methods, DeepRM [14] is by far the most representative solution for resource management. DeepRMPlus [15] is an improvement on the network structure and training mode of DeepRM, which reduces the training time and improves the resource utilization. TVW-RL [16] exploits time-varying workloads and creates the technique of creating equivalence classes in a large number of production workloads with a novel reward function, which optimizes the resource utilization. DRAW [17] designs a new feedback control mechanism to enhance resource utilization. All of these innovations in the field have improved the efficiency of the models in handling tasks, but for a short period of time large number of tasks generated, the data processing speed of the above methods is still slow.

Therefore, Decima [18] uses Spark tools to process tasks, which is currently the most representative model for processing substantial task sets quickly. Nevertheless, Decima encounters hurdles such as extended training periods, prolonged data collection durations, and sluggish convergence, while its graph neural network (GNN) aggregation of directed acyclic graphs (DAGs) operates with limited efficiency. Decima++ [19] designs a continuous-time discount return for training based on Decima and a GNN-based message passing method, which greatly reduces the training time and improves the task processing speed. Given that Decima++ still needs to further adapt to the workload and minimize the average task completion time and training time in the worst case, we have improved the model network structure and training algorithms to address these challenges.

In this paper, we design a new model Decima# based on Decima with the following contributions:

- This study introduces a resource scheduling scheme, Decima#, for edge cloud environments, providing a robust solution for processing and allocating large-scale tasks.
- To capture resource state information more accurately, we employ a transformer architecture to extract feature data on the DAG. We propose a new scheduler, termed the Decima#, which is compared against schedulers that incorporate GRU and lightweight ResNet-18 architectures.
- Given that training the Decima series models for large-scale iterations (800 times) takes over 15 hours, we designed a reward function focused on resource utilization to accelerate the iteration speed. This approach is further supported by multithreading collaboration to address concurrency issues.
- To enhance the stability of the training process, we propose an improved Proximal Policy Optimization Gradient (PPOG) algorithm, which boosts adaptability to the system and provides more precise constraints on likelihood ratios. This is validated with respect to gradient constraints and compared against a series of algorithms utilizing the PPO algorithm and its enhancements.

The remainder of this paper is organized as follows: Section 2 discusses the literature survey. Section 3 introduces preliminary information on relevant technologies and describes the system assumptions, while also formally defining two cloud resource scheduling problems. In Sec. 4, we present the details of Decima#. Section 5 provides the experimental environment and evaluation results. Finally, we summarize and conclude the paper in Sec. 6.

2. Related Work

In the research on edge cloud resource management, the work focuses on several aspects: 1) the lightweight deployment of environments at edge nodes [20], which examines methods to optimize resource usage on edge devices to enhance overall system performance; 2) dynamic scheduling and load balancing of edge resources [21], which aim to achieve efficient resource allocation through intelligent scheduling algorithms that can adapt to varying workloads; and 3) task allocation to improve the utilization of edge cloud resources [22], [23], where research in this area concentrates on effectively distributing computational tasks in edge environments to maximize resource utilization. In this paper, we focus on analyzing the task allocation aspect, and the researchers propose various meth ods to optimize the resource utilization in edge cloud. The related work can be broadly categorized into two types: traditional edge-cloud task allocation methods and machine learning-based edge-cloud task allocation methods.

For traditional task allocation methods, the problem of long task waiting times may arise in the case of heavy load. Alameddine et al. [24] proposed a method based on the Logic-Based Benders Decomposition (LBBD) approach, which decomposes multi-tasks and dynamically adjusts task allocation on edge servers according to the importance and real-time nature of tasks. Dynamic task allocation schemes in edge clouds aim to optimize the distribution of tasks among edge nodes by considering real-time load variations and resource availability. These schemes typically employ intelligent algorithms, such as genetic algorithms and particle swarm optimization, to monitor the performance and network status of edge devices in real-time and dynamically adjust the task allocation strategy.

Another category is machine learning methods. It is widely used in resource management. For similar reasons we focus only on the task allocation problem to maximize the use of resources. Maximize the utilization of resources. In edge cloud resource management, research can be categorized based on different machine learning techniques, including traditional machine learning methods and deep learning based solutions. Traditional machine learning methods, such as support vector machines (SVMs) and decision trees, use training data for feature extraction and classification to optimize task allocation and resource scheduling. As technology evolves, deep learning methods are gradually being introduced, which automatically learn complex feature representations through deep neural networks to enable more efficient task scheduling and resource utilization. By combining these different machine learning techniques, edge cloud resource management can better adapt to dynamic environments and improve system performance and responsiveness. Regarding traditional machine learning solutions, Wang et al [25] proposed a method for resource allocation in cloud computing environments using SVMs using a machine learning framework, which optimizes resource allocation strategies by analyzing historical resource usage data. Ebadi et al [26] proposed a fusion machine learning model that can be used to optimize the resource allocation in various scenarios through its dynamic, intelligent resource allocation Optimization solutions have a significant impact on the development process of edge cloud. It also exhibits lower latency, higher throughput and better overall efficiency scores.

In terms of deep learning based solutions, Farahnakian et al [27] proposed a Q-learning based approach for dynamic allocation of incoming requests. The agent is able to make optimal decisions by learning from previous experiences to determine whether a host should remain active or go into hibernation. Thonglek et al. [28] designed and implemented a neural network model based on Long Short-Term Memory (LSTM) to predict more efficient allocation of job resources based on historical data.

As for deep RL-based solutions, Mao et al. proposed a system called DeepRM, which efficiently manages nonpreemptive scheduling of online jobs through deep reinforcement learning. However, DeepRM [14] requires a long training process and the structure of the neural network is not conducive to extracting complex state-action relationships in data centers. Subsequently, based on the spark big data processing framework, Mao et al. proposed a Decima [18] system, which uses graph neural networks to aggregate DAG task information, and assigns corresponding resources to tasks through a policy network. However, Decima is primarily designed for cloud computing environments and demands time-extensive training. Consequently, its average task completion time has not yet been adequately optimized for edge computing environment.

3. Preliminaries

To help better understand the paper and to make the paper self-contained, we briefly review some background techniques. We aim to integrate the Spark environment within the edge architecture to leverage graph neural networks (GNNs) for extracting feature information from DAG structures. Through a deep reinforcement learning approach, this setup will enable the training of an optimal scheduling policy.

3.1 Edge Cloud Structure

Edge cloud architecture is a significant branch of cloud computing, characterized by the deployment of computing resources and data processing capabilities at the network's edge, close to end users and data sources. This architecture is particularly well-suited for applications requiring rapid response times, such as the Internet of Things (IoT), Augmented Reality (AR), Virtual Reality (VR), and real-time data analytics. Integrating the Decima# model into the edge cloud data center offers significant advantages that enhance overall system performance (see Fig. 1). By bringing computation and data processing closer to end users, the model effectively reduces latency, enabling faster response times essential for real-time data analytics and interactive applications in the Internet of Things (IoT), Augmented Reality (AR), and Virtual Reality (VR) environments. Additionally, deploying the Decima# model at the edge optimizes energy consumption by allowing local data processing, which minimizes the need for extensive data transmission to centralized cloud servers. This not only lowers energy usage associated with data transfer but also enhances task allocation and processing efficiency. The Decima# model intelligently distributes workloads across available edge cloud resources, ensuring optimal execution of tasks. This capability improves resource utilization, leading to cost savings and increased scalability. Overall, the integration of the Decima# model results in reduced latency, lower energy consumption, and enhanced efficiency, making it a valuable asset in the edge cloud computing landscape and enabling more effective and responsive applications across various domains.

3.2 Apache Spark

Apache Spark applications run in a Spark-managed cluster and utilize cluster resources by submitting jobs. Each job is organized into a DAG of stages, each representing a set of tasks that can be executed independently. These tasks run on different data partitions and are distributed across the cluster's worker nodes. Dependencies between stages ensure sequential execution, with one stage completing before the next one begins. Tasks within a stage are executed by executors.

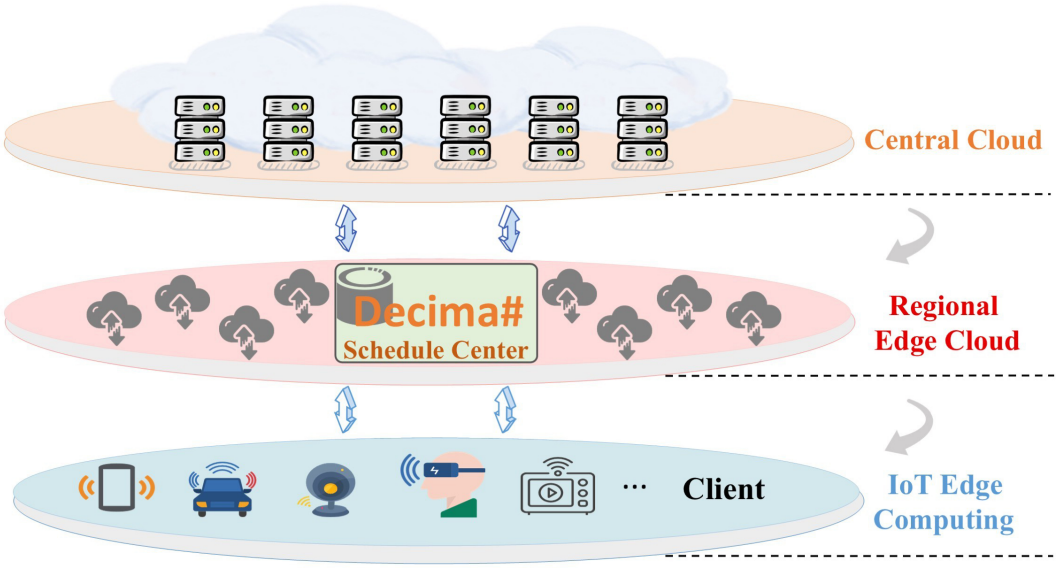


Fig. 1. The Decima# edge cloud architecture consists of three layers: the cloud layer, the edge cloud layer, and the IoT device layer. Each layer processes and transmits data based on its computational capacity, enabling efficient, low-latency task management across the system.

By default, applications are assigned a fixed number of worker nodes, but Spark can be configured to dynamically adjust worker node assignments based on workload and resource availability. Each Spark application manages its own job scheduler, which is responsible for assigning tasks to executors and optimizing task execution based on DAG dependencies. This distributed computing framework provides fault tolerance, scalability, and parallel processing for large-scale data processing tasks.

3.3 Deep Reinforcement Learning

Reinforcement Learning (RL) is a branch of machine learning where agents learn to make decisions by interacting with an environment. The agent takes actions in given states to maximize cumulative rewards over time. This process involves trial and error, where the agent explores different actions and their outcomes to improve its policy, which maps states to actions. The goal is to find an optimal policy that maximizes the expected sum of future rewards, often discounted to prioritize immediate rewards.

We adopt the discounted formulation for the policy optimization problem as described in [29]. At time t, the agent observes a state vector $\mathbf{s}_t$ and selects an action $\mathbf{a}_t$ according to the policy $\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)$, where π_θ represents the policy parameterized by θ, such as neural network weights. Our focus is on on-policy methods, where the optimized policy also determines the exploration distribution. Upon executing the chosen action, the agent transitions to a new state $\mathbf{s}'_t$ and receives a scalar reward r_t. The objective is to identify the parameter vector θ that maximizes the expected sum of future discounted rewards, $\mathbb{E}\left[\sum_{t=0}^{\infty}\gamma^t r_t\right]$, with γ being a discount factor in the interval $[0, 1]$. A lower value of γ biases the learning process towards short-term rewards over long-term gains.

Both PPO and PPOG gather experience tuples $[\mathbf{s}_i, \mathbf{a}_i, r_i, \mathbf{s}'_i]$ by running several episodes during each optimization iteration. Each episode begins with an initial state $\mathbf{s}_0$, which is drawn from a stationary distribution determined by the application. The episode proceeds until it reaches a terminal (absorbing) state or the predefined maximum episode length T is reached. After the total simulation budget N for the iteration is used up, the policy parameters θ are then updated.

3.4 Graph Neural Networks on DAG

We refer to a graph as the tuple $\mathbf{G} = (\mathbf{V}, \mathbf{E}, \mathbf{X})$ [30], [31], where V is the set of nodes, $\mathbf{E} \subseteq \mathbf{V} \times \mathbf{V}$ is the set of edges, and $\mathbf{X} \in \mathbb{R}^{n\times d}$ represents the node features. Each row of $\mathbf{X}$ is a feature vector corresponding to a node, with n denoting the number of nodes and d the feature dimension. The features of a node v are denoted by $\mathbf{x}_v \in \mathbb{R}^d$.

A directed acyclic graph (DAG) is a directed graph without directed cycles. For a DAG G, we can define a unique strong partial order $\preceq$ on the node set $\mathbf{V}$ such that, for any pair of nodes $u, v \in V$, $u \preceq v$ if and only if there is a directed path from u to v. We define the reachability relation $\preceq$ in a DAG based on this partial order: $u \preceq v$ if and only if v is reachable from u. Moreover, if $u \preceq v$, then u is called a predecessor of v, and v is a successor of u. Nodes without predecessors are source nodes, and nodes without successors are sink nodes.

Graph neural networks (GNNs) have proven to be powerful tools for learning representations of graph-structured data. Applying GNNs to DAGs introduces unique opportunities and challenges due to the inherent hierarchical structure and the absence of cycles. The strong partial order in a DAG allows for a natural topological sorting of nodes, which can be leveraged in the message-passing framework of GNNs.

In this context, each node aggregates information from its predecessors, allowing the network to propagate information in a manner consistent with the directional flow of the graph. This property is particularly advantageous for tasks that require an understanding of hierarchical dependencies, such as temporal event prediction, structured sequence modeling, and hierarchical classification.

3.5 Problem Formulation

In this section, we define two resource scheduling problems within the reinforcement learning framework, focusing on minimizing training time and average job duration.

Problem 1: Scheduling for Minimum Training Time. We aim to minimize the overall training time for jobs arriving at the data center. The training time for each job J_i is denoted as $T_{\text{train},i}$. The objective is to minimize the average training time across all jobs, while ensuring that the total allocated resources at any given time do not exceed the available resource capacity.

The formulation can be stated as:

$$\min_{D_i(t)} \lim_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} T_{\text{train},i}$$

subject to:

$$\sum_{i=1}^{n} x_j^i(t) \le c_j, \quad \forall t,\ j = 1, \ldots, d$$

Problem 2: Minimizing Average Job Duration. We also consider the problem of minimizing the average duration of jobs. The duration of a job is defined as the total time taken from the start of the job until its completion, which includes both waiting time and run time.

The objective is to minimize the average duration for all jobs while adhering to resource constraints:

$$\min_{D_i(t)} \lim_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} T_{\text{duration},i}$$

subject to:

$$\sum_{i=1}^{n} x_j^i(t) \le c_j, \quad \forall t,\ j = 1, \ldots, d.$$

In this context, $x_j^i(t) \ge 0$ represents the amount of resource type j allocated to job J_i at time t, and c_j denotes the capacity of resource type j. The constraints ensure that at any time t, the total resources allocated do not exceed the available capacities.

4. System Model

We develop a scheduling system called Decima# (see Fig. 2), specifically designed for edge cloud environments, rather than traditional cloud computing systems. Although we use the same simulation environment as Decima++, our network architecture has been differently designed with a focus on lightweight and responsive features. This ensures that the system can efficiently operate in dynamic, resource-constrained edge cloud environments, where low-latency processing and quick adaptability are critical. The system is characterized by a transformer architecture over Directed Acyclic Graphs (DAGs), efficiently processing edge-specific task dependencies. A policy network coordinates task allocation and executor assignment at the edge, enabling decentralized operation. Furthermore, reinforcement learning (RL) is applied to iteratively optimize system performance, allowing real-time adaptation to varying edge network conditions. Tasks are executed on designated edge executors based on the policy chosen by Decima#, thus optimizing resource usage and enhancing overall operational efficiency.

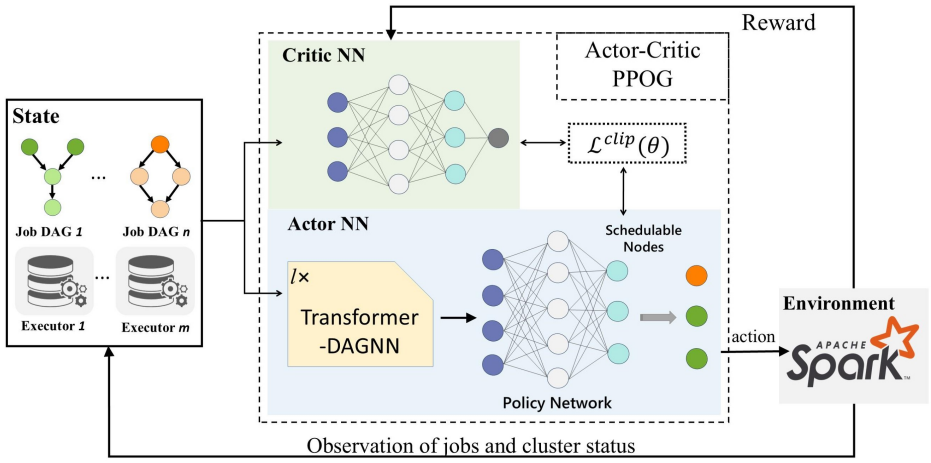


Fig. 2. In the Decima#'s reinforcement learning framework, the scheduling agent determines scheduling actions in the cluster environment by observing the current state of the cluster. The agent receives rewards based on high-level goals and evaluates the selected actions using the actor-critic (AC) network, which helps update the network parameters. The framework employs the Proximal Policy Optimization Gradient (PPOG) algorithm to ensure stable and efficient learning.

In the design of model for edge cloud environments, we incorporate a lightweight convolutional neural network (CNN) within the transformer layers to enhance computational efficiency. Furthermore, in the architecture of the DAG encoder and global encoder, we eliminate redundant fully connected layers and simplify the classification module. To achieve this, we adopt a streamlined structure that utilizes only max pooling, followed by a single fully connected layer, thereby reducing latency. Additionally, we explore the fusion of BatchNorm layers and TransformerConv layers to simplify computations and further optimize inference performance.

4.1 Node Messaging Process

To efficiently compute the stage representation of the DAG, we use a transformer network layer to update the node representation [32]. Algorithm 1 shows the node feature encoding process.

For each node v and neighboring node u in the DAG, compute the set of reachable nodes $N(v)$. This approach limits the receptive field of each node to those that are reachable, ensuring a more focused and relevant context for processing.

$$N(v) = \{(u, v) \in \leq\} \cup \{(v, u) \in \leq\} \tag{1}$$

The positional embedding is then computed considering only the node depth, add the depth-based positional encoding (DAGPE) to the node features before attention calculation, denoted as

$$\begin{aligned} \mathbf{PE}(v, 2i) &= \sin\left(\frac{\text{depth}(v)}{10000^{2i/d}}\right), \\ \mathbf{PE}(v, 2i+1) &= \cos\left(\frac{\text{depth}(v)}{10000^{2i/d}}\right) \end{aligned} \tag{2}$$

where d is the feature dimension and i is the dimension index.

The formula for calculating the attention weight of node v, can be defined as

$$\alpha_{\mathbf{x}_v} = \sum_{u \in N_v} \frac{\kappa_{\exp}(\mathbf{x}_v + \mathbf{PE}_v, \mathbf{x}_u + \mathbf{PE}_u)}{\sum_{w \in N_v} \kappa_{\exp}(\mathbf{x}_v + \mathbf{PE}_v, \mathbf{x}_w + \mathbf{PE}_w)} f(\mathbf{x}_u) \tag{3}$$

where $\mathbf{x}$ is the feature of node v, where $f(\mathbf{x}) = \mathbf{W}_V(\mathbf{x})$ is a linear function and $\kappa_{\exp}$ is the kernel parametrized by $\mathbf{W}_Q$ and $\mathbf{W}_K$ in $R^d \times R^d$ space, which is expressed as

$$\kappa_{\exp}(\mathbf{x}_v, \mathbf{x}_u) = \frac{\mathbf{x}_v \mathbf{W}_Q, \mathbf{x}_u \mathbf{W}_K}{\sqrt{d}}. \tag{4}$$

We have incorporated multiple transformer layers as node encoders into the graph neural network, along with a lightweight neural network to enhance computational efficiency. This approach captures the complex structure and information of the input graph more effectively (see Fig. 3).

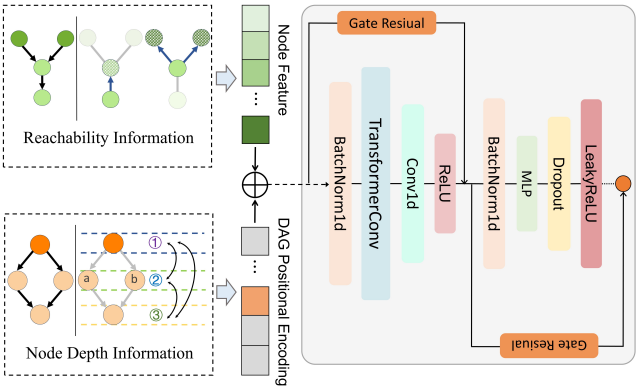


Fig. 3. The reachability-based graph neural network transforms the raw information of each DAG node into a vector representation, employing a convolutional layer in the encoder's initial layer as a replacement for the traditional MLP.

Algorithm 1 Reachability-based messaging approach.

Require: DAG $(\mathbf{V}, \mathbf{E})$ with node features $\mathbf{X} \in \mathbb{R}^{|\mathbf{V}| \times c}$, weight matrices $\mathbf{W}^{\mathbf{Q}}, \mathbf{W}^K, \mathbf{W}^{\mathbf{V}}, \mathbf{W}^R \in \mathbb{R}^{c \times d}$
Ensure: Node representations $\mathbf{H} \in \mathbb{R}^{|\mathbf{V}| \times d}$
Initialize $\mathbf{Q} \leftarrow \mathbf{X} \cdot \mathbf{W}^{\mathbf{Q}}$, $K \leftarrow \mathbf{X} \cdot \mathbf{W}^K$, $\mathbf{V} \leftarrow \mathbf{X} \cdot \mathbf{W}^{\mathbf{V}}$
Initialize $\mathbf{H} \leftarrow 0 \in \mathbb{R}^{|\mathbf{V}| \times d}$
for each node $v \in \mathbf{V}$ **do**
  $N_k(v) \leftarrow$ Compute reachable nodes from v
  $d_v \leftarrow$ Calculate depth of node v
  $\alpha_v \leftarrow$ Compute attention scores from $\mathbf{Q}_{N_k(v)}$ and K_v
  $\mathbf{H}_v \leftarrow$ Aggregate messages from reachable nodes $N_k(v)$ using α_v
  $\mathbf{H}_v \leftarrow$ Apply non-linear transformation (e.g., ReLU) to $\mathbf{H}_v$
  $g_v \leftarrow$ Compute gating mechanism for node v
  $\mathbf{H}_v \leftarrow g_v \cdot \mathbf{H}_v + (1 - g_v) \cdot (\mathbf{X}_v \cdot \mathbf{W}^R)$
  $\mathbf{H}_v \leftarrow$ Apply final non-linear transformation to $\mathbf{H}_v$
end for
$\mathbf{H} \leftarrow$ Concatenate all updated representations $\mathbf{H}_v$ for $v \in \mathbf{V}$
return $\mathbf{H}$

This process effectively captures the structural bias of DAGs and leads to improved performance in various tasks. By using transformer layers, the model leverages self-attention mechanisms that allow it to focus on key relational features and dependencies among nodes, particularly beneficial for directed acyclic graphs (DAGs). This configuration not only captures the structural biases of DAGs but also enables the network to process complex relationships more accurately, leading to improved task performance across a range of applications. Furthermore, the transformer layers' ability to dynamically weigh node interactions enhances the model's generalization, making it more effective in diverse graph-related tasks.

4.2 Overall Messaging Process

The graph neural network (GNN)-based transformer network incorporates two additional encoders: the DAG encoder and the global encoder. The overall framework is shown in Fig. 4. The DAG encoder is responsible for aggregating the features of each updated node into a DAG representation, denoted as $\mathbf{y}$. The global encoder $\mathbf{z}$ aggregates each DAG into a global embedding value. In contrast to Decima,

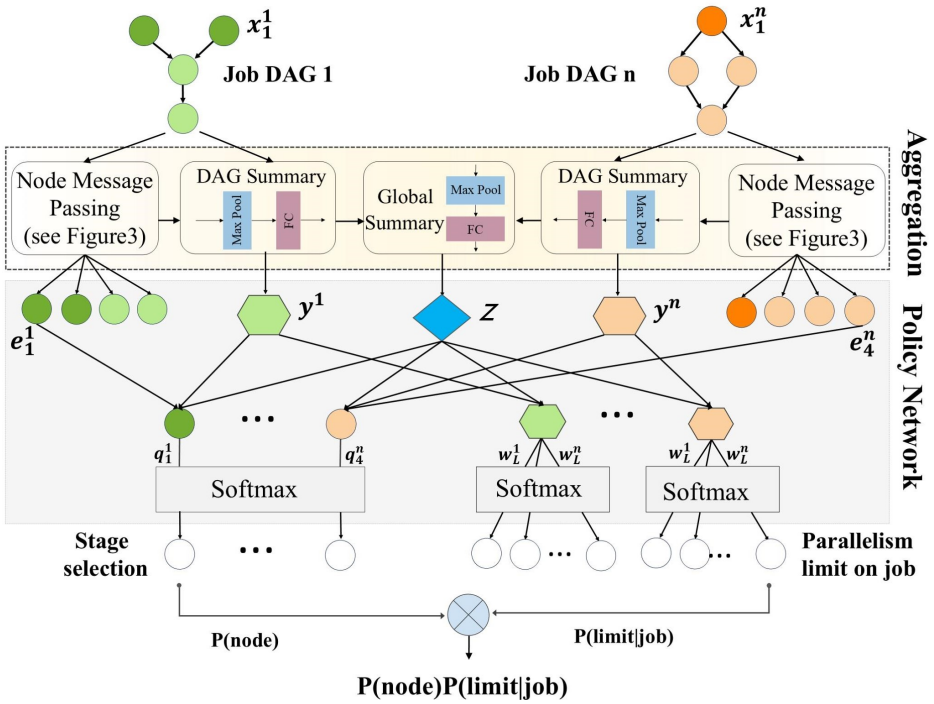


Fig. 4. In this framework, a transformer integrates with a GNN for node feature extraction $\mathbf{e}_v^i$, DAG embedding $\mathbf{y}^i$, and global embedding $\mathbf{z}$, unlike Decima, which directly uses GNN for aggregation. For all these embeddings, a simple classification layer is employed, consisting of a max pooling operation followed by a fully connected layer. This approach is used to calculate two key scores: $\mathbf{q}_v^i$, which represents the score for selecting a node to schedule, and $\mathbf{w}_l^i$, which determines the parallelism limit for the job associated with that node. Additionally, in this framework, we reduce parameters by optimizing layer ratios and integrating batchnorm with convolution layers for better efficiency.

both encoders use max-pooling as the aggregation method, and the aggregation process is represented as follows:

$$\left\{\mathbf{x}_v^1, \mathbf{x}_v^2, \ldots\right\} \longmapsto \left\{\mathbf{y}^1, \mathbf{y}^2, \ldots\right\} \longmapsto \mathbf{z} \tag{5}$$

these values obtained from the encoder provide the conditions for the policy network to calculate the score q_i of the i_{th} sampled nodes to be scheduled, and the score w_i of the sampled node jobs with parallelism constraints.

The node selection network takes as input the embedding vectors of all runnable nodes and outputs their corresponding scores. It is implemented as a fully connected neural network with two hidden layers, each comprising 64 units, and an output layer containing 5 units. For each runnable node n_i, the network computes a score s_i, reflecting its scheduling priority. The computation of s_i is governed by (6). Decima's policy network utilizes these scores to determine the selection probability of each runnable node, as defined by (7). Subsequently, a softmax layer is applied to select the scheduling node v by evaluating the probabilities across all runnable nodes. The node selection network is trained using reinforcement learning techniques.

$$q_v^i \triangleq q\left(\mathbf{x}_v^i, \mathbf{y}^i, \mathbf{z}\right), \tag{6}$$

$$P(\text{node} = v) = \frac{\exp\left(q_v^i\right)}{\sum_{u \in \mathcal{A}_t} \exp\left(q_u^{j(u)}\right)} \tag{7}$$

where $j(u)$ represents the job assigned to node u, and $\mathcal{A}_t$, the set of nodes available for scheduling at time t, defines a smaller action space. For each job i, the policy network of Decima# also employs another scoring function to calculate the score $w_l = w(\mathbf{y}^i, \mathbf{z}, l)$, which assigns the parallelism limit l to job i. The job parallelism limit computation process is similar to the process of obtaining node v.

4.3 Design Reward Functions

Since our algorithm utilizes neural networks to schedule information processing and select appropriate nodes, these neural networks must be trained. When training a scheduling algorithm, it is difficult to determine which scheduling action is optimal; therefore, supervised learning methods are not applicable in this case. On the contrary, reinforcement learning can dynamically update the scheduling algorithm based on its results, thereby gradually improving performance. Therefore, we employ a reinforcement learning algorithm to train the neural network by adjusting its hyperparameters.

Firstly, we incorporate the use of time penalties, to calculate the waiting time in the queue, which can be selected as

$$R_K = -(t_k - t_{k-1}) * \text{num}_{\text{dag}} \tag{8}$$

where t_k is the time of the k_{th} scheduling event and num_{dag} represents the number of DAG applications in the system between t_{k-1} and t_k.

In addition, We also define the machine usage rate as in (9), which evaluates the percentage of utilization of the

currently used machines, and the addition of this incentive mechanism allows the agent to allocate more unused machine resources to the workload, thereby increasing machine utilization.

$$R_{\mathrm{U}} = K_u * - \sum_{d} \sum_{m \in M_u} |U_m(t,d)| \tag{9}$$

where R_{U} is under-utilization penalty, M_u denotes the set of machines currently in use, and $U_m(t,d)$ denotes the unused resource for machine m at time t across resource dimension d. The calibration of weight K_u in the penalty function also helps in teaching the scheduler to switch between highly optimum placement mode (when less incoming scheduling requests are expected) to a less optimum placement mode (when burst of scheduling requests are expected to hit the cluster). Decima# can have different penalty coefficients for different resource dimensions.

Decima# utilizes the sequential continuous-time discounted reward presented in Decima++. The reward function is defined as

$$\begin{aligned} R(s,a,s')_{\mathrm{total}} &= -\int_{t_s}^{t_s'} \mathrm{e}^{-\beta(t-t_s)}(R_k + R_{\mathrm{U}})\mathrm{d}t \\ &= -\frac{1}{\beta} \sum_{j \in J_s \cup J_{s'}} [\exp\left(-\beta \times (\max\{t_{j,s}^{\mathrm{a}}, t_s\} - t_s)\right) \\ &\quad - \exp\left(-\beta \times (\min\{t_{j,s'}^{\mathrm{c}}, t_{s'}\} - t_s)\right)](R_k + R_{\mathrm{U}}). \end{aligned} \tag{10}$$

The reward generated by taking action a in state s to state s', where β is a continuous discount factor, $\beta \geqq 0$, can be easily decomposed into the cumulative form of the reward function. t represents the current time, indicating the real-time progress within the simulation. J denotes the subset of currently active jobs and N_j represents the maximum number of events (J),$\{j \in [N_j] : t \in [t_j^{\mathrm{a}}, t_j^{\mathrm{c}}]\}$, It signifies the subset of active tasks within the simulation, where each job j with an index from 1 to N_j is considered active if its execution time falls between its arrival time (t_j^{a}) and completion time (t_j^{c}). If job j is still active in the simulation, its completion time is indicated as infinite. The reinforcement learning algorithm attempts to maximise the reward for each action, thus reducing scheduling as training proceeds, the interval and number of DAG applications running in the system decreases. As a result, the average training time can be effectively reduced.

4.4 Adaptive Resource Scheduling Algorithm

Proximal Policy Optimization (PPO) [33] is a policy gradient method, whose loss function formulated as follows

$$\begin{aligned} L^{\mathrm{CLIP}}(\pi) = \mathbb{E}\big[&\min(r_{s,a}(\pi)A^{\pi_{\mathrm{old}}}(s,a)), \\ &\mathcal{F}(r_{s,a}(\pi),\epsilon)A^{\pi_{\mathrm{old}}}(s,a))\big] \end{aligned} \tag{11}$$

where $r_{s,a}(\pi) = \frac{\pi_\theta(a|s)}{\pi_{\mathrm{old}}(a|s)}$ is an importance sampling ratio, $A^{\pi_{\mathrm{old}}}$ is an advantage estimate (e.g., $= R - \hat{V}^{\pi}(s_k)$) and $\epsilon \in [0,1]$ is a hyperparameter, which V calculated by baseline. To reduce variance, Decima# inputs the relevant baseline for Monte Carlo dominance estimation as in the following (12).

$$V^{\pi}(s,\xi) = \mathbb{E}_{\tau \sim \pi|\xi}\left[\sum_{k=1}^{\infty} \mathrm{e}^{-\beta t_k} R \,\middle|\, s_1 = s\right] \tag{12}$$

This is the expected cumulative time penalty and unused resource penalty spent by the job during execution and waiting, after discounting, the input-related baseline becomes

Moreover, PPO adopts the actor-critic [34] method, where the critic's estimate optimizes the policy $\pi(a|s;\theta)$, also referred to as the actor parameterized by θ. Following this, the policy parameters can be updated using the gradient of the agent objective as $\theta_{t+1} = \theta_t + \beta \nabla L^{\mathrm{CLIP}}(\theta_t)$, where β is the step size. The advantage of defining such a loss function is that, through the importance sampling rate, data reuse can be achieved so that multiple policy updates can be made in each training iteration, similar to mini-batch stochastic gradient ascent over multiple iterations. In addition, clipping ensures that θ is reasonably close to θ_{old}, thus ensuring that updates are restrained. Therefore, we delve into the clip mechanism in detail. In PPO, $\mathcal{F}$ is defined as

$$\mathcal{F}^{\mathrm{PPO}}\left(r_{s,a}(\pi),\epsilon\right) = \begin{cases} 1-\epsilon & r_{s,a}(\pi) \leq 1-\epsilon \\ 1+\epsilon & r_{s,a}(\pi) \geq 1+\epsilon \\ r_{s,a}(\pi) & \text{otherwise} \end{cases}. \tag{13}$$

Algorithm 2 PPOG algorithm for training Decima#.

```
Initialize policy parameters θ and best state
for each training iteration do
    for j = 1, ..., M (number of training jobs) do
        Sample time limit for job j: t_j^max ~ exp(1/t̄_max)
        Generate job sequence ξ^j with job arrivals before t_j^max
        for i = 1, ..., N (number of workers) do
            Collect trajectory τ^ij by running policy π_θ on job sequence
            ξ^j until all jobs are completed or time limit t_j^max is reached
        end for
    end for
    Compute discounted returns for each trajectory R_total^ij using decay
    factor β
    Fit return estimates R^ij(t) to time steps t_k^ij within each trajectory
    Calculate baseline b_k^ij for each job sequence ξ^j
    Estimate advantage A_k^ij = R_total^ij − b_k^ij
    Reset best state if necessary:
    if R_total^ij > best_state then
        best_state ← R_total^ij
        Store current policy as best policy: θ_best ← θ
    end if
    Store current policy as old policy: θ_old ← θ
    for E epochs do
        Randomly partition collected trajectories into mini-batches B
        for each mini-batch B do
            Compute importance sampling ratios r_k^ij(θ) for each sample
            in B
            Update policy parameters θ by minimizing the clipped surro-
            gate loss:
                [min(r_k^ij(θ)A_k^ij, clip(r_k^ij(θ), 1 − ϵ, 1 + ϵ)A_k^ij)]
        end for
    end for
end for
```

PPG [35] offers the benefits of training both policy network and value function network using a common network, allowing the features learned for each goal to enhance the optimization of the other goal. GePPO [36] seeks to enhance the stability and efficiency of reinforcement learning algorithms in practical decision-making scenarios by enhancing assurances on sample reuse and providing theoretical backing. PPORB [37] illustrates, based on some theoretical proofs, that its ability to prevent out-of-range ratios from going further out of range can be improved. The PPOS smoothing mechanism, proposed in [38], improves the clipping method and enhances convergence. Based on PPORB and PPOS, we find a better family of curves to limit the magnitude of the variation in the likelihood ratio.

We define this function in (14) with a modified structure to enhance the stability of constrained likelihood ratios and gradient updates, incorporating the softsign activation function as a key component of our approach.

$$\mathcal{F}^{\mathrm{PPOG}}\left(r_{s,a}(\pi),\epsilon,\alpha\right)= \begin{cases} -\alpha\mathrm{softsign}(r_{s,a}(\pi)-1)+1+\varepsilon+\alpha\mathrm{softsign}(\varepsilon) \\ \qquad\qquad r_{s,a}(\pi)\le 1-\varepsilon \\ -\alpha\mathrm{softsign}(r_{s,a}(\pi)-1)+1-\varepsilon-\alpha\mathrm{softsign}(\varepsilon) \\ \qquad\qquad r_{s,a}(\pi)\ge 1+\varepsilon \\ r_{s,a}(\pi) \qquad\qquad \text{otherwise} \end{cases} \tag{14}$$

where $\alpha > 0$, the hyperparameter α governs the scale of the functional clipping, with larger values leading to a more pronounced decrease in slope. Algorithm 2 shows in detail the process of training decima# using this algorithm.

We demonstrate how the PPOG can restrict the likelihood ratio and impose constraints on the stability of the gradient. Let $L_t^{\mathrm{PPOG}}(\theta)$ denote the corresponding objective function for sample (s_t, a_t); and let $\hat{L}^{\mathrm{PPOG}}(\theta)$ denote the overall empirical objective. The PPOG function $\mathcal{F}^{\mathrm{PPOG}}\left(r_t(\theta),\epsilon,\alpha\right)$ generates a negative incentive when $r_t(\theta)$ is outside of the clipping range. Thus, it could somewhat neutralize the incentive deriving from the overall objective $\hat{L}^{\mathrm{PPOG}}(\theta)$. For convenience, we introduce the following notation. Given parameter θ_0, let $\theta_1^{\mathrm{CLIP}} = \theta_0+\beta\nabla\hat{L}^{\mathrm{CLIP}}(\theta_0)$, $\theta_1^{\mathrm{PPOG}} = \theta_0+\beta\nabla\hat{L}^{\mathrm{PPOG}}(\theta_0)$. The study in this paper is conditioned on exceeding the likelihood ratio ($r_{s,a} > 1$). The set of indexes of the samples which satisfy the clipping condition is denoted as $\Omega = \{t|1 \le t \le T, |r_t(\theta_0)-1| \ge \epsilon \text{ and } r_t(\theta_0)A_t \ge r_t(\theta_{\mathrm{old}})A_t\}$.

Theorem 1 *Suppose that $t \in \Omega$ and $r_t(\theta_0)$ satisfies $\sum_{t'\in\Omega}\langle\nabla r_t(\theta_0), \frac{\nabla r_{t'}(\theta_0)}{[r_{t'}(\theta_0)]^2}\rangle A_t A_{t'} > 0$, then there exists some $\bar{\beta} > 0$ such that for any $\beta \in (0,\bar{\beta})$, we have*

$$\left|r_t(\theta_1^{\mathrm{PPOG}})-1\right| < \left|r_t(\theta_1^{\mathrm{PPO}})-1\right|. \tag{15}$$

Theorem 2 *Suppose that $t \in \Omega$, we have two conclusions:*

1. *If $r_t(\theta_0)$ satisfies $\sum_{t'\in\Omega}\langle\nabla r_t(\theta_0), \frac{\nabla r_{t'}(\theta_0)}{[r_{t'}(\theta_0)]^2} - \nabla r_{t'}(\theta_0)\rangle A_t A_{t'} < 0$, then there exists some $\bar{\beta} > 0$ such that for any $\beta \in (0,\bar{\beta})$, we have*

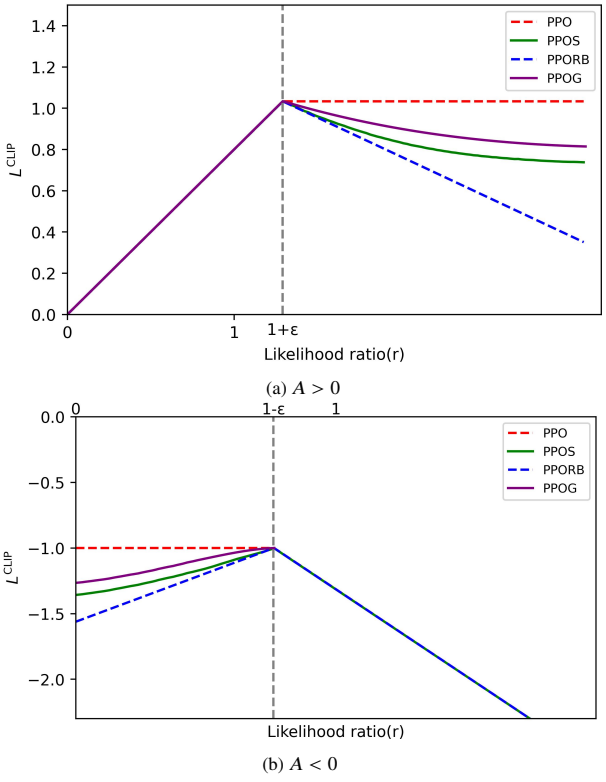
$$\left|r_t(\theta_1^{\mathrm{PPOG}})-1\right| > \left|r_t(\theta_1^{\mathrm{PPORB}})-1\right|. \tag{16}$$

2. *If $r_t(\theta_0)$ satisfies $\sum_{t'\in\Omega}\langle\nabla r_t(\theta_0), \frac{\nabla r_{t'}(\theta_0)}{[r_{t'}(\theta_0)]^2} - \sec^2(r_{t'}(\theta_0)-1)\nabla r_{t'}(\theta_0)\rangle A_t A_{t'} < 0$, then there exists some $\bar{\beta} > 0$ such that for any $\beta \in (0,\bar{\beta})$, we have*

$$\left|r_t(\theta_1^{\mathrm{PPOG}})-1\right| > \left|r_t(\theta_1^{\mathrm{PPOS}})-1\right|. \tag{17}$$

The proofs are in Appendix A and Appendix B.

Figure 5 illustrates the CLIP function L_{CLIP} of the likelihood ratio r for PPO and PPOG, focusing on positive advantages (top) and negative advantages (bottom). The black circle on each plot indicates the starting point for optimization, where $r = 1$. When r exceeds the clipping range, the slope of L_{CLIP} for PPOG follows the tanh function, where as the slope of L_{CLIP} for PPO becomes flattened. Theorem 1 demonstrates that the PPOG algorithm improves the capability to prevent out-of-range ratios from going further out of range, because if α is sufficiently large, the new strategy is guaranteed to be confined to the specified range. Theorem 2 shows that under certain conditions (similar to Lipschitz continuous), the PPOG method with softsign converges more rapidly than PPOS and PPORB.



(a) $A > 0$

(b) $A < 0$

Fig. 5. Plots showing the clip function L_{CLIP} of the likelihood ratio $r \triangleq r_{s,a}$ of PPO and PPOG, for positive advantages (top) and negative advantages (bottom).

5. Experiments and Analysis

5.1 Experimental Settings

The proposed work is experimented on Pytorch. The simulation scenarios are written in Python. The experiment is launched on a desktop computer with the following hardware configuration:

- **CPU:** Intel(R) Xeon(R) CPU E5-2686 v4 @ 2.30GHz
- **SSD:** Samsung SSD 870 EVO 500GB
- **GPU:** V100, 32GB HBM2
- **Operating System:** Ubuntu 18.04 LTS

We initially train the model using a configuration of 50 executors, a job arrival cap of 200, and a job arrival rate of 4×10^{-5}. Multiple optimized models are trained using various reinforcement learning techniques and different DAG-GNN aggregation methods.

These models are then tested with 10 executors and a job arrival cap of 50, as well as other parameter combinations. Their performance is compared against the baseline algorithms listed in the Tab. 2.

5.2 Design Concurrent Workload

Batched arrivals. In this study, we adopt a batch arrival strategy by randomly sampling jobs from six distinct input sizes (2, 5, 10, 20, 50, and 100 GB) alongside all 22 TPC-H [39] queries. This approach results in a heavy-tailed distribution. A batch random jobs, which were not included in the training phase, is introduced simultaneously, and we measure their average job completion time (JCT). This methodology enables a thorough analysis of the system's performance when handling such workloads and facilitates optimization efforts.

Continuous arrivals. We sample TPC-H jobs of varying sizes and model their arrival times as Poisson processes. This methodology facilitates a comparative analysis of the job arrival rate under cluster load in relation to different heuristic-based scheduling strategies.

Multiprocessing training. The multi-process scheme used for training Decima employs multiple actor processes to collect experience and compute gradients in parallel, while a central process updates the model parameters. When the number of GPUs is fewer than the number of actor processes, this setup results in reduced GPU throughput due to limited batch processing capabilities and competition for GPU resources. To enhance throughput, they adopt an "actor-learner" scheme, wherein the actors are solely responsible for collecting experience, while the learner manages the learning process. This approach not only improves GPU utilization but also facilitates the integration of experiences from multiple actors, making it easier for PPO to adapt to more effective learning. However, since Decima++ estimates the value function rather than learning it, synchronization between the actors and the learner is necessary. In comparison, Decima# employs a complex, multi-layer transformer to process DAG information, which requires a longer experience-gathering process and more intricate computational logic. Consequently, Decima# has high CPU core requirements, leading to the introduction of a concurrent processing scheme known as "cooperative".

5.3 Performance Comparison

In this section, we will conduct two sets of experiments. The first set aims to evaluate the training time of Decima# under different objectives. The second set focuses on assessing the average work duration of Decima# across various cluster configurations, comparing its performance against baseline algorithms in a simulated environment.

5.3.1 Minimum Training Time (Problem 1)

Reinforcement learning (RL) agents must encounter continuous job arrivals during training. However, providing long sequences of jobs often leads to inefficiencies and significant wastage of training time. To address this issue, it is essential to implement earlier reset times (see Fig. 6).

Algorithm	Description
Random	Random scheduling of jobs without priority
Heuristic algorithms	FCFS (first come first service), SJF (Shortest Job First), HRRN (Highest Response Ratio Next)
Tetris	A cluster scheduling algorithm proposed by Microsoft, which matches the multi-resource task requirements to the resource availability of the machine
Fair	Simple fair scheduling, which gives each job an equal fair share of the executors and round-robins over tasks from runnable stages to drain all branches concurrently
Decima	A representative deep RL-based resource scheduling method
Decima++	Decima-based replacement of training methods and DAG handling

Tab. 2. Comparison of scheduling algorithms.

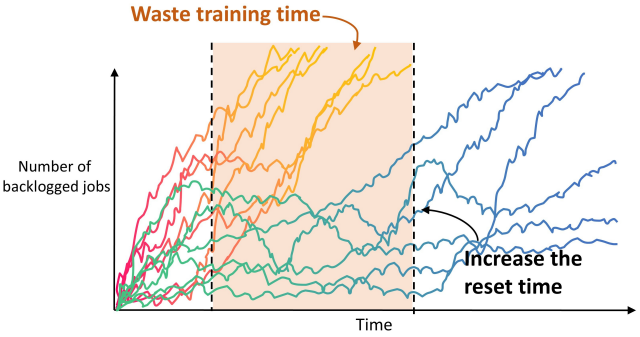
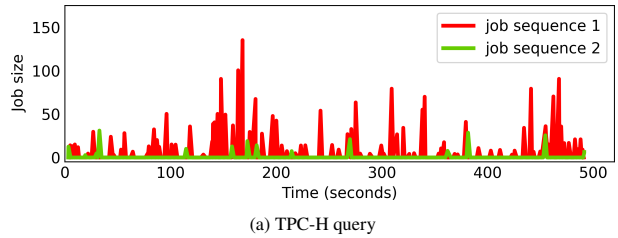


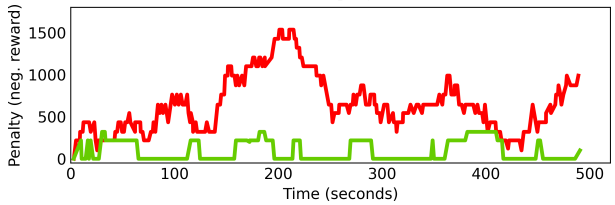
Fig. 6. Continuous input of large work sequences causes inefficiencies, which can be minimized by periodically resetting the model to its optimal state; this reduces queue buildup and optimizes training times based on the chosen reset intervals.

Scheduler	Backbone	Transformer layer	FC type	FC layer dims	RL-methods	Convergence time
Decima	GNN	N/A	MLP	256→256	PG	8.5h ± 25min
Decima# (ours)	GNN	N/A	MLP	256→256→128	PPOG	5.3h ± 36min
Decima# (ours)	GNN-Transformer (Resnet18)	4 × 8	MLP	256→256→128	PPORB	4.6h ± 47min
Decima# (ours)	GNN-Transformer (Resnet50)	3 × 4	LSTM	384→256→256	PPOS	7.5h ± 37min
Decima# (ours)	GNN-Transformer (Gated)	6 × 8	MLP	512→256→128	PPOG	3.0h ± 18min
Decima# (ours)	GNN-Transformer (Gated)	3 × 4	GRU	512→256→128	PPO	3.5h ± 27min
Decima++	GNN	N/A	MLP	512→256→128	PPO	4.2h ± 43min
Decima++	GNN-ATT	3 × 4	GRU	512→256→128	VPG	3.8h ± 28min

Tab. 1. Scheduler convergence time comparison. In this experiment, different models were trained, including LSTM (Long Short-Term Memory networks) and GRU (Gated Recurrent Units), which represent two types of recurrent neural network architectures. MLP (Multilayer Perceptron) refers to a standard fully connected layer model. It is important to note that N/A indicates "not available" data. In the Spark simulation environment, models were trained according to the experimental parameters mentioned in the previous section until convergence, and the approximate training times were recorded for comparison.



(a) TPC-H query



(b) Negative reward

Fig. 7. After time t, we sample two job arrival sequences from a Poisson arrival process (with an average inter-arrival time of 10 seconds). These sequences include randomly sampled TPC-H queries.

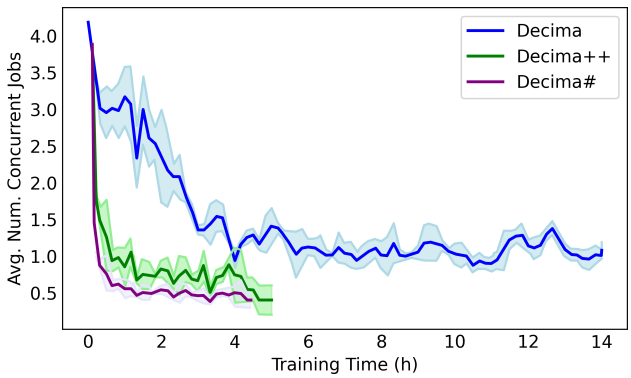


Fig. 8. Smooth training curve with horizontal axis representing training time in hours and vertical axis representing the number of concurrent jobs.

As training progresses, agents develop stronger policies that effectively stabilize the job queue. Therefore, this paper emphasizes the importance of earlier reset times to enhance training efficiency while maintaining queue stability through more robust policies.

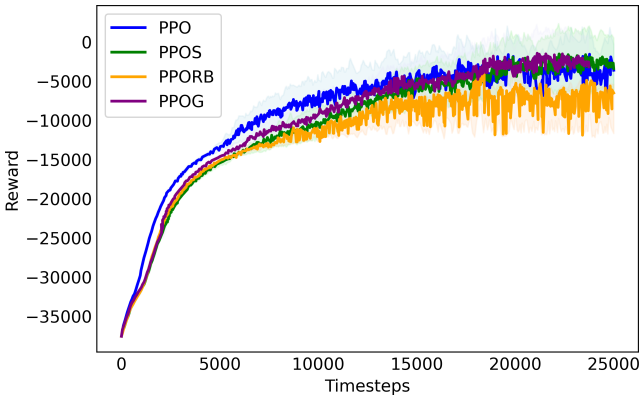


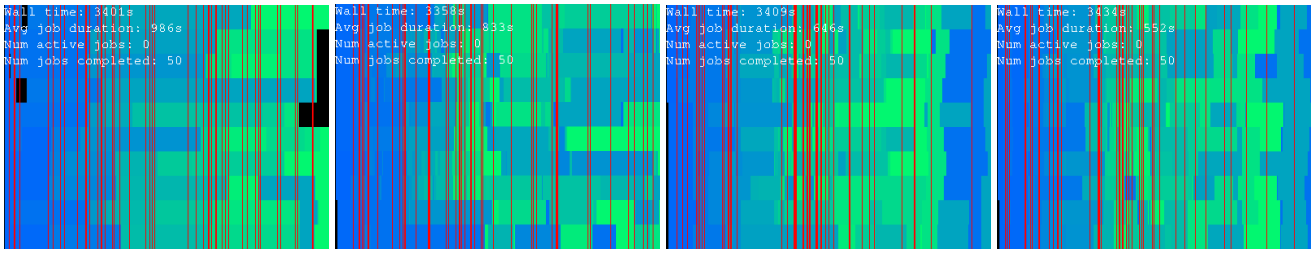
Fig. 9. Spark-v $|\mathbf{O}| = 25$, where $|\mathbf{O}|$ represents the dimension of observation. Decima# reward changes under different training algorithms.

In this case, Decima# uses R_{total} as the reward function. Figure 7 illustrates how different job sizes result in varying reward values. It can be observed that smaller job sizes lead to smaller rewards, while larger job sizes correspond to larger rewards.

Since heuristic baseline algorithms do not require training, no comparisons are needed in this section. To understand why Decima# outperforms Decima++, we extracted concurrent job data over time for comparison with training duration, as illustrated in Fig. 8. The results clearly indicate that Decima# exhibits superior learning speed.

We can observe from Tab. 1 that Decima# significantly reduces training time through various neural network structure configurations. Furthermore, the main goal in reinforcement learning is to maximize the cumulative reward of a given trajectory. As shown in Fig. 9, Decima# cumulative rewards using different algorithms all increase with training time.

If the decisions of some samples are much better than the average then there may still be room for further improvement in the current strategy. Otherwise, the model has stabilized as the gap becomes smaller.



(a) Fair scheduling. (b) Decima++(GNN-ATT) scheduling. (c) Decima scheduling. (d) Decima# scheduling.

Fig. 10. Decima# reduces the average job duration for 50 random TPC-H queries by 44% compared to the fair scheduler, 33% over Decima++, and 19% over Decima in a cluster with 10 task slots (executors). The visualization uses different colors to represent various queries, with vertical red lines marking job completions and purple segments indicating idle periods.

5.3.2 Minimum Average Job Duration (Problem 2)

This section focuses on the average job duration for Decima#, examining how its resource scheduling optimizations contribute to faster task completion times and improved system efficiency. Decima++ demonstrates strong performance in training time and convergence speed, though it falls short of Decima in minimizing average job duration. Figure 10 illustrates the scheduling strategies used: (a) a fairer, more realistic scheduler that dynamically distributes task slots among jobs; (b) Decima++ scheduler using topology messaging; (c) Decima's learning-based scheduling approach; and (d) Decima#'s scheduling method.

We evaluated the average job duration over 50 tasks. By leveraging graph structures, the average job duration was reduced by 44% compared to the fair scheduler, achieved through faster short-task completion and increased parallel efficiency. Decima# operates near its optimal parallelism point, resetting to maintain its best state, achieving a 19% reduction in average job duration compared to Decima and 33% compared to Decima++.

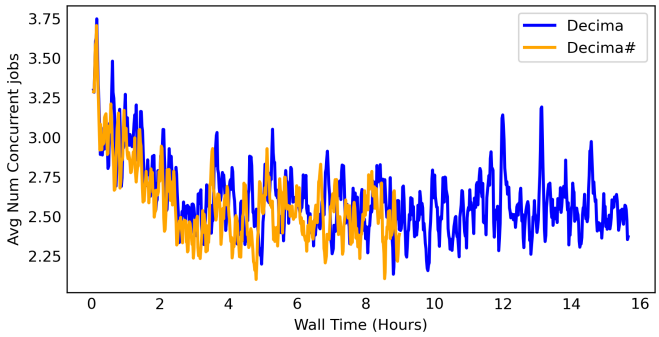
Moreover, we benchmark the model's performance using the TPC-H dataset [39]. To validate the model's effectiveness, we conducted experiments in scenarios closely resembling real-world scheduling conditions, testing with various ratios of executors to jobs. Specifically, for Decima#(a), an initial ratio of 1:20 was adopted. Additionally, we observed that under excessively high load conditions, the average job duration significantly increases, highlighting potential performance bottlenecks. Subsequently, the ratio was reduced to 1:10 for Decima#(b). Finally, for Decima#(c) and Decima#(d), the ratios were further adjusted to 1:9 and 1:10, respectively, to simulate higher load levels. We also observed that, at the same ratio, for example, comparing Decima#(b) with Decima#(d), an increase in the number of executors leads to a reduction in the average job duration. The experimental results demonstrate that Decima# consistently outperforms other algorithms across different load conditions, showcasing superior adaptability and performance. Detailed results are presented in Table 3.

Figure 11(a) demonstrates that Decima# sustains a lower number of concurrent jobs and a shorter average job duration compared to Decima (Fig. 11(b)), especially under high load. Performance in high-load scenarios is critical for batch clusters, as they often face extended job queues. In such cases, optimized scheduling decisions can significantly reduce the need for excessive resource allocation during workload spikes, maximizing both efficiency and resource utilization. The data presented in Fig. 11 illustrates how Decima#'s approach to managing job concurrency and duration optimizes system resource utilization. In comparison to Decima, Decima# demonstrates a more effective mechanism for reducing job contention, resulting in a more efficient distribution of resources and improved workload balancing. This capability allows Decima# to accommodate a higher volume of concurrent jobs without risking system overload, leading to enhanced overall throughput and a reduction in job turnaround times. Such improvements contribute to the system's ability to maintain high performance under varying workload conditions, reinforcing the efficiency and scalability of Decima# in large-scale job processing environments.

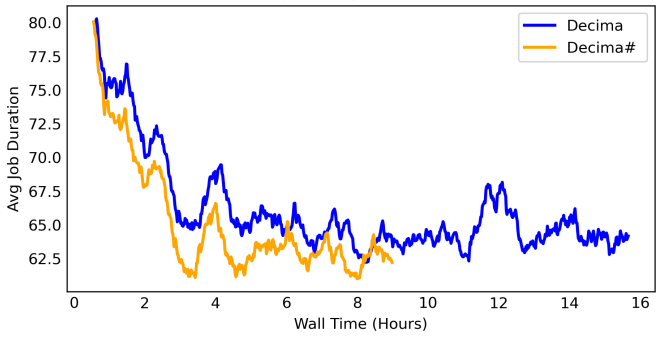
Moreover, Decima#'s adaptive scheduling dynamically adjusts to load variations, optimizing resource allocation across job phases. By minimizing queuing and maintaining efficient resource use, it prevents bottlenecks, enhances stability, and ensures responsive performance, making the system more robust and scalable for fluctuating job arrival rates.

Model	Executors	Jobs	AJD(s)
Fair	10	200	3483.3
Decima++	10	200	3136.7
Decima	10	200	2891.5
Decima#(a)	**10**	200	**2620.2**
Fair	10	100	1716.3
Decima++	10	100	1557.5
Decima	10	100	1320.9
Decima#(b)	**10**	100	**1062.8**
Fair	100	900	37.8
Decima++	100	900	35.2
Decima	100	900	33.8
Decima#(c)	**100**	900	**31.5**
Fair	200	2000	30.35
Decima++	200	2000	28.52
Decima	200	2000	26.6
Decima#(d)	**200**	2000	**24.75**

Tab. 3. Performance of the model in terms of the number of jobs completed with varying numbers of executors.



(a) The number of jobs completed varies over time



(b) Time as a function of average job duration

Fig. 11. The data files recorded in TensorBoard captured a time-series analysis of arrival times for consecutive TPC-H jobs within the Spark cluster (as illustrated in figures a and b). The analysis reveals that, in terms of job duration, this approach outperforms Decima by completing jobs more quickly.

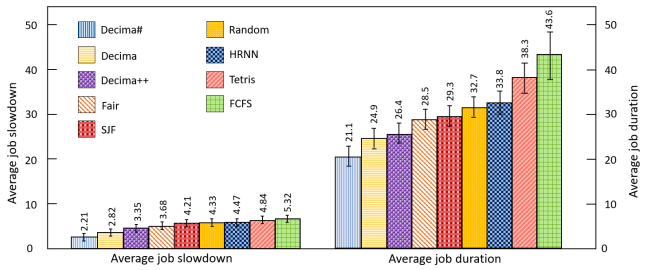


Fig. 12. Performance for different objectives.

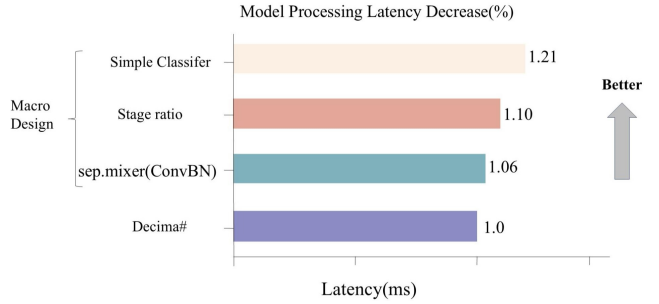


Fig. 13. Added the latency reduction ratio for each structural improvement step based on Decima#.

Figure 12 illustrates the performance of two objectives (average job slowdown and average job duration) when a cluster with a large number of executors handles workloads.

It is important to note that Decima# uses different reward functions for each objective (see Sec 4.3 for details). Consistent with previous results, we find that Decima# outperforms other machine learning-based algorithms, while fair scheduling is superior to other heuristic methods. However, when Decima# is specifically trained to optimize each objective using the appropriate reward function, it achieves the best performance for each goal. Thus, Decima# can be effectively customized to target different objectives.

5.3.3 Measurement of Processing Latency

The inference speed of a model can certainly be optimized using metrics such as floating point operations (FLOP) or model size. However, these metrics have a weak correlation with the actual latency in edge cloud environments or on edge devices [40]. To better address the needs of edge applications, we have shifted the focus of our model evaluation to latency-related metrics, emphasizing that our design is specifically tailored for edge environments.

Given the stringent responsiveness requirements of edge cloud environments, we prioritized designing models with high responsiveness and conducted a detailed latency analysis at every stage of model operation. Figure 13 illustrates the entire testing process, highlighting the impact of network design details at each stage on latency. All models were trained and validated using the TPC-H dataset. By systematically analyzing the impact of each design decision on latency and optimizing them step by step, we minimized processing latency and ensured optimal performance, making our models well-suited for deployment in latency-sensitive edge cloud applications.

5.3.4 Overall Summary

Through a series of experiments conducted in a simulated environment, we concluded that Decima# outperforms all baseline methods in terms of average training convergence time and average working duration. Specifically, Decima# demonstrates a significant improvement in training convergence speed, achieving a 56% increase compared to state-of-the-art algorithms. Additionally, it reduces the average working time by 19%, highlighting its efficiency in resource scheduling. These findings indicate that Decima# not only accelerates the training process but also optimizes the operational efficiency of the tasks at hand. The reduced working time implies that resources are utilized more effectively, allowing for quicker turnaround in data-intensive applications. This efficiency can be crucial in edge computing scenarios, where rapid responses and resource optimization are essential for meeting the demands of real-time data processing. In designing Decima#, a series of deliberate steps were taken to create a lightweight network and reduce latency. The model architecture was optimized to minimize computational overhead by adopting efficient network structures with fewer pa-

rameters, simplified layers, and faster activation functions. These design choices not only enhanced the model's computational efficiency but also preserved its accuracy. Looking forward, we plan to further streamline the model using techniques such as model pruning, quantization, and knowledge distillation. These optimizations will accelerate inference speeds and reduce memory consumption, making the model even more suitable for real-time edge computing tasks.

6. Conclusion

In this paper, we present Decima#, an advanced resource scheduling framework designed to enhance the performance and user experience of cloud and edge data centers. By incorporating constraint-gradient-based DAG optimization, multithreaded scheduling, transformer techniques, and lightweight network structures, Decima# improves scheduling efficiency and adapts to diverse workloads. Extensive simulations conducted in Spark demonstrate its robustness in real-world scenarios, while curriculum learning enhances the model's generalization, enabling it to handle complex scheduling tasks in dynamic environments. Looking ahead, future work will focus on refining Decima# to achieve more precise scheduling and exploring its integration within edge-cloud environments. Adapting Decima# to support AI-driven workloads and emerging technologies like 5G and edge AI will ensure its continued relevance, enabling it to meet the evolving demands of next-generation cloud and edge infrastructures. Furthermore, expanding Decima#'s scalability will be critical to handling the growing volume of data and increasing complexity in future applications. The framework's adaptability will also be crucial in supporting a wide range of industries, from autonomous systems to real-time data analytics, ensuring its versatility and long-term impact.

Acknowledgments

The authors would like to thank the anonymous reviewers and the Associate Editor. This work was supported by the Science and Technology Development Project of Jilin Province (20240701127FG), the Research Project of Jilin University of Finance and Economics (2023YB035 and 2024LH011), and the National Natural Science Foundation of China (12271201). We also thank all participants for their contributions.

References

[1] BELOGLAZOV, A., BUYYA, R. Managing overloaded hosts for dynamic consolidation of virtual machines in cloud data centers under quality of service constraints. *IEEE Transactions on Parallel and Distributed Systems*, 2012, vol. 24, no. 7, p. 1366–1379. DOI: 10.1109/TPDS.2012.240

[2] GOUDARZI, H., GHASEMAZAR, M., PEDRAM, M. SLA-based optimization of power and migration cost in cloud computing. In *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. Ottawa (Canada), 2012, p. 172–179. DOI: 10.1109/CCGrid.2012.112

[3] ASSI, C., AYOUBI, S., SEBBAH, S., et al. Towards scalable traffic management in cloud data centers. *IEEE Transactions on Communications*, 2014, vol. 62, no. 3, p. 1033–1045. DOI: 10.1109/TCOMM.2014.012614.130747

[4] HU, J., GU, J., SUN, G., et al. A scheduling strategy on load balancing of virtual machine resources in cloud computing environment. In *2010 3rd International Symposium on Parallel Architectures, Algorithms and Programming*. Liaoning (China), 2010, p. 89–96. DOI: 10.1109/PAAP.2010.65

[5] LEINBERGER, W., KARYPIS, G., KUMAR, V., et al. Load balancing across near-homogeneous multi-resource servers. In *Proceedings 9th Heterogeneous Computing Workshop (HCW 2000)*. Cancun (Mexico), 2000, p. 60–71. DOI: 10.1109/HCW.2000.843733

[6] SHAW, S. B., SINGH, A. K. A survey on scheduling and load balancing techniques in cloud computing environment. In *2014 International Conference on Computer and Communication Technology (ICCCT)*. Allahabad (India), 2014, p. 87–95. DOI: 10.1109/ICCCT.2014.7001474

[7] YANHAO, Z., ABHISHEK, N. V., GURUSAMY, M. RAVEN: Resource allocation using reinforcement learning for vehicular edge computing networks. *IEEE Communications Letters*, 2022, vol. 26, no. 11, p. 2636–2640. DOI: 10.1109/LCOMM.2022.3196711

[8] ZHOU, Z., LIU, P., FENG, J., et al. Computation resource allocation and task assignment optimization in vehicular fog computing: A contract-matching approach. *IEEE Transactions on Vehicular Technology*, 2019, vol. 68, no. 4, p. 3113–3125. DOI: 10.1109/TVT.2019.2894851

[9] DOMANAL, S. G., GUDDETI, R. M. R., BUYYA, R. A hybrid bio-inspired algorithm for scheduling and resource management in cloud environment. *IEEE Transactions on Services Computing*, 2017, vol. 13, no. 1, p. 3–15. DOI: 10.1109/TSC.2017.2679738

[10] LI, R., WANG, C., ZHAO, Z., et al. The LSTM-based advantage actor-critic learning for resource management in network slicing with user mobility. *IEEE Communications Letters*, 2020, vol. 24, no. 9, p. 2005–2009. DOI: 10.1109/LCOMM.2020.3001227

[11] MISHRA, K., RAJAREDDY, G. N. V., GHUGAR, U., et al. A collaborative computation and offloading for compute-intensive and latency-sensitive dependency-aware tasks in dew-enabled vehicular fog computing: A federated deep Q-learning approach. *IEEE Transactions on Network and Service Management*, 2023, vol. 20, no. 4, p. 4600–4614. DOI: 10.1109/TNSM.2023.3282795

[12] LIU, W. X., LU, J., CAI, J., et al. DRL-PLink: Deep reinforcement learning with private link approach for mix-flow scheduling in software-defined data-center networks. *IEEE Transactions on Network and Service Management*, 2021, vol. 19, no. 2, p. 1049–1064. DOI: 10.1109/TNSM.2021.3128267

[13] SHI, J., DU, J., WANG, J., et al. Priority-aware task offloading in vehicular fog computing based on deep reinforcement learning. *IEEE Transactions on Vehicular Technology*, 2020, vol. 69, no. 12, p. 16067–16081. DOI: 10.1109/TVT.2020.3041929

[14] MAO, H., ALIZADEH, M., MENACHE, I., et al. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*. Atlanta (USA), 2016, p. 50–56. DOI: 10.1145/3005745.3005750

[15] GUO, W., TIAN, W., YE, Y., et al. Cloud resource scheduling with deep reinforcement learning and imitation learning. *IEEE Internet of Things Journal*, 2020, vol. 8, no. 5, p. 3576–3586. DOI: 10.1109/JIOT.2020.3025015

[16] MONDAL, S. S., SHEORAN, N., MITRA, S. Scheduling of time-varying workloads using reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*. Virtual Event, 2021, vol. 35, no. 10, p. 9000–9008. DOI: 10.1609/aaai.v35i10.17088

[17] CHEN, X., YANG, L., CHEN, Z., et al. Resource allocation with workload-time windows for cloud-based software services: A deep reinforcement learning approach. *IEEE Transactions on Cloud Computing*, 2022, vol. 11, no. 2, p. 1871–1885. DOI: 10.1109/TCC.2022.3169157

[18] MAO, H., SCHWARZKOPF, M., VENKATAKRISHNAN, S. B., et al. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*. New York (USA), 2019, p. 270–288. DOI: 10.1145/3341302.3342080

[19] GERTSMAN, A. A faster reinforcement learning approach to efficient job scheduling in Apache Spark. *Master's Thesis, University of Illinois at Urbana-Champaign*, 2023, p. 1–48. [Online]. Available at: https://hdl.handle.net/2142/121563

[20] MORABITO, R., COZZOLINO, V., DING, A. Y., et al. Consolidate IoT edge computing with lightweight virtualization. *IEEE Network*, 2018, vol. 32, no. 1, p. 102–111. DOI: 10.1145/3341302.3342080

[21] ZHANG, J., GUO, H., LIU, J., et al. Task offloading in vehicular edge computing networks: A load-balancing solution. *IEEE Transactions on Vehicular Technology*, 2019, vol. 69, no. 2, p. 2092–2104. DOI: 10.1109/TVT.2019.2959410

[22] XIONG, X., ZHENG, K., LEI, L., et al. Resource allocation based on deep reinforcement learning in IoT edge computing. *IEEE Journal on Selected Areas in Communications*, 2020, vol. 38, no. 6, p. 1133–1146. DOI: 10.1109/JSAC.2020.2986615

[23] ALFAKIH, T., HASSAN, M. M., GUMAEI, A., et al. Task offloading and resource allocation for mobile edge computing by deep reinforcement learning based on SARSA. *IEEE Access*, 2020, vol. 8, p. 54074–54084. DOI: 10.1109/ACCESS.2020.2981434

[24] ALAMEDDINE, H. A., SHARAFEDDINE, S., SEBBAH, S., et al. Dynamic task offloading and scheduling for low-latency IoT services in multi-access edge computing. *IEEE Journal on Selected Areas in Communications*, 2019, vol. 37, no. 3, p. 668–682. DOI: 10.1109/JSAC.2019.2894306

[25] WANG, J. B., WANG, J., WU, Y., et al. A machine learning framework for resource allocation assisted by cloud computing. *IEEE Network*, 2018, vol. 32, no. 2, p. 144–151. DOI: 10.1109/MNET.2018.1700293

[26] CHEN, T., ZHANG, X., YOU, M., et al. A GNN-based supervised learning framework for resource allocation in wireless IoT networks. *IEEE Internet of Things Journal*, 2021, vol. 9, no. 3, p. 1712–1724. DOI: 10.1109/JIOT.2021.3091551

[27] THONGLEK, K., ICHIKAWA, K., TAKAHASHI, K., et al. Improving resource utilization in data centers using an LSTM-based prediction model. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. Albuquerque (USA), 2019, p. 1–8. DOI: 10.1109/CLUSTER.2019.8891022

[28] TULI, S., MAHMUD, R., TULI, S., et al. Fogbus: A blockchain-based lightweight framework for edge and fog computing. *Journal of Systems and Software*, 2019, vol. 154, p. 22–36. DOI: 10.1016/j.jss.2019.04.050

[29] HU, B., ZHANG, K., LI, N., et al. Toward a theoretical foundation of policy optimization for learning control policies. *Annual Review of Control, Robotics, and Autonomous Systems*, 2023, vol. 6, no. 1, p. 123–158. DOI: 10.1146/annurev-control-042920-020021

[30] BIANCHINI, M., GORI, M., SCARSELLI, F. Processing directed acyclic graphs with recursive neural networks. *IEEE Transactions on Neural Networks*, 2001, vol. 12, no. 6, p. 1464–1470. DOI: 10.1109/72.963781

[31] SUTTORP, M. M., SIEGERINK, B., JAGER, K. J., et al. Graphical presentation of confounding in directed acyclic graphs. *Nephrology Dialysis Transplantation*, 2015, vol. 30, no. 9, p. 1418–1423. DOI: 10.1093/ndt/gfu325

[32] LUO, Y., THOST, V., SHI, L. Transformers over directed acyclic graphs. In *Advances in Neural Information Processing Systems*. Vancouver (Canada), 2024, p. 47764–47782. DOI: 10.48550/arXiv.2210.13148

[33] SCHULMAN, J., WOLSKI, F., DHARIWAL, P., et al. Proximal policy optimization algorithms. *arXiv*, 2017, p. 1–12. DOI: 10.48550/arXiv.1707.06347

[34] GRONDMAN, I., BUSONIU, L., LOPES, G. A. D., et al. A survey of actor-critic reinforcement learning: Standard and natural policy gradients. *IEEE Transactions on Systems*, 2012, vol. 42, no. 6, p. 1291–1307. DOI: 10.1109/TSMCC.2012.2218595

[35] COBBE, K. W., HILTON, J., KLIMOV, O., et al. Phasic policy gradient. In *International Conference on Machine Learning*. Virtual Event, 2021, p. 2020–2027. DOI: 10.48550/arXiv.2009

[36] QUEENEY, J., PASCHALIDIS, Y., CASSANDRAS, C. G. Generalized proximal policy optimization with sample reuse. In *Advances in Neural Information Processing Systems*. Virtual Event, 2021, p. 11909–11919. DOI: 10.48550/arXiv.2111.00072

[37] WANG, Y., HE, H., TAN, X. Truly proximal policy optimization. In *Uncertainty in Artificial Intelligence, PMLR*. Virtual Event, 2020, p. 113–122. DOI: 10.48550/arXiv.1903.07940

[38] ZHU, W., ROSENDO, A. A functional clipping approach for policy optimization algorithms. *IEEE Access*, 2021, vol. 9, p. 96056–96063. DOI: 10.1109/ACCESS.2021.3094566

[39] BONCZ, P., NEUMANN, T., ERLING, O. TPC-H analyzed: Hidden messages and lessons learned from an influential benchmark. In *Technology Conference on Performance Evaluation and Benchmarking*. Trento (Italy), 2013, p. 61–76. DOI: 10.1007/978-3-319-04936-6

[40] KAUR, K., GARG, S., AUJLA, G. S., et al. Edge computing in the industrial internet of things environment: Software-defined-networks-based edge-cloud interplay. *IEEE Communications Magazine*, 2018, vol. 56, no. 2, p. 44–51. DOI: 10.1109/MCOM.2018.1700622

About the Authors . . .

Youling FENG received her Doctor of Science degree from the School of Mathematics, Jilin University. Her research focuses on big data, artificial intelligence and operator theory.

Mengzhao LI (corresponding author) is a master's student at the Jilin University of Finance and Economics, with a primary research focus on reinforcement learning and edge computing.

Jun LI received his Doctor of Science degree from the School of Computer Science, Jilin University. His research interests include artificial intelligence and computer vision.

Yawei YU is a master's student at the Jilin University of Finance and Economics, with a primary research focus on multimodal sentiment analysis and graph neural network.

Appendix A: Proof of Theorem1

Proof 1 *Consider* $\phi(\beta) = r_t(\theta_0 + \beta\nabla\hat{L}^{\text{PPOG}}(\theta_0)) - r_t(\theta_0 + \beta\nabla\hat{L}^{\text{CLIP}}(\theta_0))$*, by chain rule, we have*

$$\begin{aligned}\phi'(0) &= \nabla r_t^{\top}(\theta_0)(\nabla\hat{L}^{\text{PPOG}}(\theta_0) - \nabla\hat{L}^{\text{CLIP}}(\theta_0)) \\ &= -\alpha\sum_{t'\in\Omega}\langle\nabla r_t(\theta_0), \nabla r_{t'}(\theta_0)/[r_{t'}(\theta_0)]^2\rangle A_{t'}.\end{aligned} \tag{A1}$$

In $\hat{L}^{\text{CLIP}}$*, when* CLIP = PPO*, for these case where* $r_t(\theta_0) \geq 1+\epsilon$ *and* $A_t > 0$*, we have* $\phi'(0) < 0$*. Hence, there exists* $\bar{\beta} > 0$ *such that for any* $\beta \in (0, \bar{\beta})$

$$\phi(\beta) < \phi(0).$$

Thus, we have $r_t(\theta_1^{\text{PPOG}}) < r_t(\theta_1^{\text{CLIP}})$*. We obtain*

$$\left|r_t(\theta_1^{\text{PPOG}})\right| < \left|r_t(\theta_1^{\text{CLIP}})\right|.$$

Similarly, for the case where $r_t(\theta_0) \leq 1-\epsilon$ *and* $A_t < 0$*, we also have* $\left|r_t(\theta_1^{\text{PPOG}}) - 1\right| < \left|r_t(\theta_1^{\text{CLIP}}) - 1\right|$.

Appendix B: Proof of Theorem2

Proof 2 *For* $\phi(\beta)$*, as Theorem 1, when* CLIP = PPORB*, we have*

$$\phi'(0) = -\alpha\sum_{t'\in\Omega}\langle\nabla r_t(\theta_0), (\nabla r_{t'}(\theta_0)/([r_{t'}(\theta_0)]^2) - \nabla r_{t'}\theta_0)\rangle A_{t'} \tag{B1}$$

when CLIP = PPOS*, we have*

$$\begin{aligned}\phi'(0) = -\alpha\sum_{t'\in\Omega}\langle\nabla r_t(\theta_0), \frac{\nabla r_{t'}(\theta_0)}{[r_{t'}(\theta_0)]^2} \\ -\sec^2(r_{t'}(\theta_0) - 1)\nabla r_{t'}(\theta_0)\rangle A_{t'}.\end{aligned} \tag{B2}$$

In addition, when CLIP = PPORB *or* CLIP = PPOS*, for these case where* $r_t(\theta_0) \geq 1+\epsilon$ *and* $A_t > 0$*, we have* $\phi'(0) > 0$*. Hence, there exists* $\bar{\beta} > 0$ *such that for any* $\beta \in (0, \bar{\beta})$

$$\phi(\beta) > \phi(0).$$

Thus, we have

$$r_t(\theta_1^{\text{PPOG}}) > r_t(\theta_1^{\text{CLIP}}).$$

We obtain

$$\left|r_t(\theta_1^{\text{PPOG}}) - 1\right| > \left|r_t(\theta_1^{\text{CLIP}}) - 1\right|.$$

Similarly, for the case where $r_t(\theta_0) \leq 1-\epsilon$ *and* $A_t < 0$*, we also have* $\left|r_t(\theta_1^{\text{PPOG}}) - 1\right| > \left|r_t(\theta_1^{\text{CLIP}}) - 1\right|$.

Appendix C: Detailed Description

Figure 14 illustrates the fitted trend of the average job duration (AJD) as a function of the number of jobs and the number of executors. The plot highlights how the job duration varies with changes in these two key variables, offering insights into the relationship between workload (represented by the number of jobs) and computational resources (indicated by the number of executors). This trend is derived from a model fitted to the data, which allows us to observe the impact of scaling the number of jobs and executors on the average job duration. The figure provides a clear visual representation of how performance is influenced by these factors, helping to inform decisions regarding optimal resource allocation and system efficiency.

To evaluate the effectiveness of our proposed method, we use the widely recognized TPC-H benchmark dataset, which is designed to simulate the workload of a retail product supplier's decision support system. We conduct the evaluation in a Spark environment, enhancing performance with Decima#. The dataset includes 22 complex SQL queries that represent various business analysis tasks. The TPC-H dataset features a comprehensive database schema that models various entities, such as parts, suppliers, and orders, along with their attributes. In this edge-cloud scenario, by applying TPC-H queries within the Spark environment using Decima#, we can effectively assess the impact of task-oriented Directed Acyclic Graphs (DAGs) on query performance in a distributed setting. This allows us to identify potential improvements in efficiency, reduce resource contention, and optimize scheduling strategies for large-scale data processing tasks (see Fig. 15).

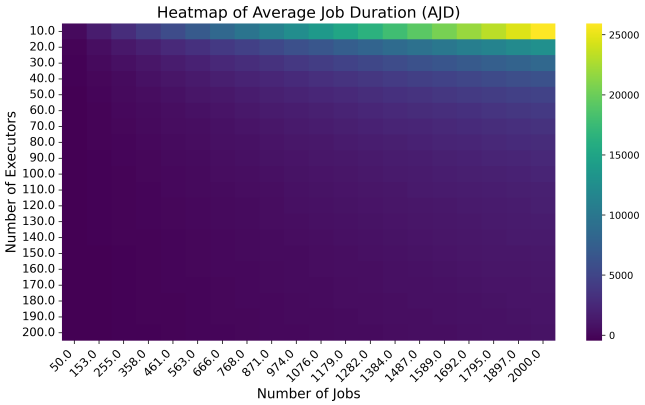


Fig. 14. Performance optimization of average job duration with varying numbers of jobs and executors.

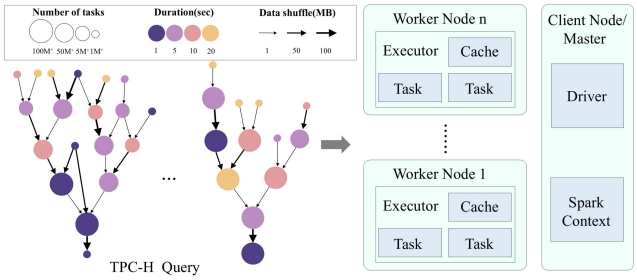


Fig. 15. Spark's TPC-H queries, where each node differs in the number of tasks, task durations, and the sizes of input and output data.

Decima# training scenario	Average JCD (seconds)
Trained with 50× fewer jobs	340 ± 230
Trained with 100× more executors	36 ± 18

Tab. 4. In this test scenario, Decima#'s scheduler was used to test the performance of the scheduler by controlling the number of executors(10) and the number of jobs(900).

This study investigates the comparative dynamics between edge cloud and traditional cloud environments within the context of Spark simulations, highlighting both the similarities and inherent differences. While edge cloud can be regarded as a form of cloud computing, its limited computational resources, such as reduced CPU cores and memory, render the Spark simulation environment at the edge analogous to traditional cloud-based environments during certain stages, particularly when interfacing with edge devices. However, these similarities are overshadowed by significant differences in terms of resource availability, computing power, and latency, which are pivotal factors. In particular, the constraints imposed by edge cloud environments manifest more acutely in areas such as task scheduling and resource allocation, posing unique challenges due to the scarcity of resources. To address these challenges, this research introduces optimized resource scheduling algorithms aimed at enhancing computational resource efficiency in edge cloud settings, a problem that is less pronounced in traditional cloud infrastructures.

Furthermore, this study utilizes the PyTorch framework, differing notably from prior research like Decima, which relies on TensorFlow. This choice not only shifts the toolset but also impacts implementation and performance optimization strategies. PyTorch's dynamic computation graph provides crucial flexibility for designing and debugging complex models, particularly in resource-constrained edge cloud environments. In such latency-sensitive scenarios, we focus on lightweighting and optimizing models to ensure efficient task execution. While sharing some experimental similarities with earlier studies, this work emphasizes the efficient execution of Spark simulations in edge clouds, addressing challenges like limited resources, unstable networks, and strict latency requirements. To overcome these, we introduce strategies such as dynamic resource allocation, model parameter compression, and computational flow optimization, enhancing Spark simulations' efficiency and robustness. By leveraging PyTorch's strengths, this study not only achieves methodological innovation but also offers valuable insights for edge cloud research. Table 4 shows how the performance of this agent compares to the performance of the agent trained on the workload and cluster size in this test scenario.