Container Scheduling with Dynamic Computing Resource for Microservice Deployment in Edge Computing

Jingxi Lu[2], Wenhao Li[2], Jianxiong Guo[1, 2, *], Xingjian Ding[3], Zhiqing Tang[1], and Tian Wang[1]
[1] Advanced Institute of Natural Sciences, Beijing Normal University, Zhuhai 519087, China
[2] Department of Computer Science, BNU-HKBU United International College, Zhuhai 519087, China
[3] Faculty of Information Technology, Beijing University of Technology, Beijing 100124, China
{q030026100, q030026075}@mail.uic.edu.cn, {jianxiongguo, zhiqingtang, tianwang}@bnu.edu.cn, dxj@bjut.edu.cn

***Abstract*—With the massive increase of Internet of Things devices and their data, executing applications by using microservice architecture has emerged as the predominant trend. As the container technology emerges, microservices can be lightweightly deployed in resource-constrained edge nodes. However, existing container scheduling algorithms often overlook the allocation of computing resources on edge servers. When multiple containers are assigned to an edge node, it is usually assumed that they share a CPU frequency, which is obviously unrealistic. In this paper, we first formulate an online container-based microservice scheduling problem with dynamic computing power to minimize the total delay and energy consumption, where we need to determine the assignment between microservices and edge nodes and the allocation of computing power to each microservice in an edge node. Then, we propose a Soft Actor-Critic (SAC) based reinforcement learning algorithm to address this problem, where a GRU unit is designed in the policy network to extract the correlation among multiple decisions, and an action selection mechanism is given to speed up the convergence. Finally, a simulated scheduling system is implemented to validate our algorithm, which demonstrates that our algorithm outperforms the commonly used baselines by up to 65% in terms of the total objective on average.**

***Index Terms*—Microservice Deployment, Online Container Scheduling, Dynamic Resource Allocation, Edge Computing, Reinforcement Learning.**

I. Introduction

With the development of Artificial Intelligence (AI), Internet of Things (IoT), and The Fifth Generation Mobile Network (5G), more and more IoT devices and the massive data generated by them flood into the Internet. In traditional cloud computing, these massive computing tasks can be offloaded to the cloud center for processing. To reduce response delay and save bandwidth and energy, Mobile Edge Computing (MEC) has emerged as a promising solution that allows computing power to be deployed at the edge of the network [1], thus it is incredibly beneficial to the popular delay-sensitive and computation-intensive intelligent applications. Microservice architecture [2] [3] is commonly used in cloud-native application design and development but lacks extensive research on microservice scheduling in edge computing.

Container [4], known for its lightweight virtualization, serves as an ideal tool for encapsulating and deploying microservices, which can achieve rapid and scalable deployment by utilizing its lightweight. Some practical container scheduling algorithms have been proposed, which can be used to implement container-based microservice deployment in edge computing, as shown in Fig. 1. However, current container management tools, such as Docker [5] and Kubernetes [6], implement only basic strategies for allocating containers to physical nodes, which only consider physical resource consumption [7]. Soon after, research focuses on optimizing resource scheduling while transmission delay, image reutilization, and load balancing are considered. However, there are some issues in existing resource allocation modeling. For example, when multiple microservices/containers are assigned to an edge node, it is usually assumed that the edge node has sufficient and stable computing power. The CPU frequency the edge node provides to each microservice/container is the same [8]. Because the edge nodes are limited in resources, this is unrealistic. Or, it is assumed that microservices/containers are queued to be addressed in the edge node. However, this mechanism has a certain probability of causing a large delay, which is unacceptable for delay-sensitive tasks.

The first challenge is allocating multiple microservices to multiple edge nodes according to the distribution of container images on edge nodes, especially how to schedule the computing power of an edge node when multiple microservices are allocated to it. Here, we adopt the Round-Robin (RR) strategy [9] to equally distribute the remaining computing power to all new arriving tasks. Once it is determined, it will not change until the task is completed. This is in line with the container management mechanism in Kubernetes. Once a container is created and executed, corresponding computing

* Jianxiong Guo is the corresponding author.

This work was supported in part by the National Natural Science Foundation of China (NSFC) under Grant No. 62202055, the Start-up Fund from Beijing Normal University under Grant No. 312200502510, the Internal Fund from BNU-HKBU United International College under Grant No. UICR0400003-24, the Project of Young Innovative Talents of Guangdong Education Department under Grant No. 2022KQNCX102, and the Interdisciplinary Intelligence SuperComputer Center of Beijing Normal University (Zhuhai).

2994-3523/24/$31.00 ©2024 IEEE
DOI 10.1109/MSN63567.2024.00041

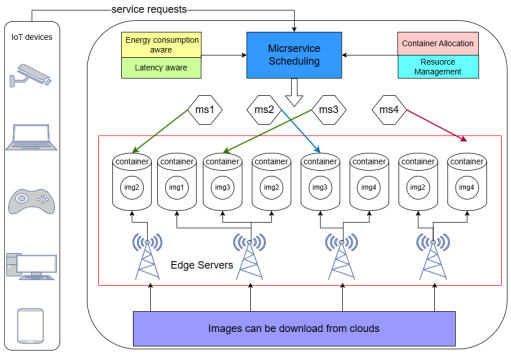


Fig. 1. The microservice management architecture: Demonstrates our microservice deployment strategy using containers for optimized service delay and energy usage across edge nodes, maintaining acceptable performance under variable loads.

resources should be allocated. However, this modeling creates new challenges. We need to assign multiple microservices to the specific edge node at the same time. However, we do not know how many microservices will be assigned to an edge node when making decisions. If the current remaining CPU frequency is significant, we tend to assign it too many tasks, which may cause poor performance.

In this paper, we formulate the online container-based microservice scheduling problem with dynamic computing power as a Markov Decision Process (MDP) to minimize the total delay and energy consumption. Its goal is to make informed decisions based on the current state to achieve long-term advantages in reducing latency and energy consumption. A policy gradient-based RL algorithm, Soft Actor-Critic (SAC) [10], achieves our online scheduling because of its good performance and fast convergence. Since all microservices' decisions interact, we add a GRU unit to the policy network to extract the correlation between multiple actions. The hidden state of this GRU is cleared once every time slot because it only focuses on the things in this time slot. Finally, we implement our algorithms in a simulated container scheduling system to validate the effectiveness of our proposed algorithm, which can be found in https://github.com/Blacktower27/CSDCRMDE. Experimental results stand out by substantially decreasing latency and augmenting energy efficiency, enabling quick and consistent convergence and enhancing system performance compared to existing paradigms.

The main contributions of this paper are as follows: (1) To the best of our knowledge, we are the first to mathematically model the online container-based microservice scheduling problem with dynamic computing power to minimize total latency and energy consumption, which improves the reliability and efficiency of microservice applications in resource-constrained edge nodes. (2) We propose a SAC-based microservice scheduling algorithm for online offloading decisions, addressing multi-objective optimization by sequentially executing simultaneous decisions. A GRU unit in the policy network captures correlations among actions in the same time slot, with an action selection mechanism to ensure faster training convergence while reducing the action space. (3) We extensively evaluate against commonly used heuristic and RL algorithms. Results demonstrate superior performance regarding low latency, energy efficiency, and guaranteed convergence. Additional ablation experiments validate our approach, achieving up to a 65% reduction in total objective compared to baselines.

II. Related Work

Containers are widely adopted for application deployment due to their simplified management and lightweight nature compared to traditional Virtual Machines (VMs) [4]. They streamline deployment by sharing the host system's kernel, eliminating the need for separate operating systems per instance. Container-based microservice deployment and scheduling have garnered significant attention in academia and industry [11]. Singh *et al.* [12] automated microservice deployment on Docker containers for a social networking application in cloud computing. Container-based cloud platforms offer low-overhead, secure environments ideal for modular microservices [13] [14]. Wen *et al.* [15] introduced GA-Par, a microservice orchestration framework emphasizing reliability and Quality of Service (QoS) across distributed cloud data centers, focusing on security and consistent network performance.

Reinforcement learning (RL) is utilized in deploying microservices because of its notable strengths in addressing strategic decision-making challenges. Wang *et al.* [16] investigated microservice coordination among edge clouds to enable seamless and real-time responses, where they formulated it as a MDP and solved it by using a RL-based algorithm to learn the optimal strategy. Chen *et al.* [17] developed a multi-buffer deep deterministic policy gradient approach to optimize service deployment strategies to reduce the average response time. Lv *et al.* [18] proposed a multi-objective microservice deployment problem in edge computing by using a Reward Sharing Deep Q Learning (RSDQL), which can minimize the communication overhead and achieve load balance between edge nodes.

III. System Model and Problem Formulation

A. System model

We first model the offloading scenario for computational microservices. As shown in Fig. 1, in the Internet of Things (IoT) environment, computation-intensive microservice requests are generated by IoT devices and offloaded to a group of edge nodes $\mathcal{N}$ for processing. The set of nodes is defined as $\mathcal{N} = \{n_1, n_2, \cdots, n_N\}$, where N denotes the number of nodes in this set. Each edge node n possesses limited resources, primarily including total CPU frequency F_n, memory resource M_n, and storage space D_n, and can concurrently run a finite number of containers. The system time is divided into consecutive time slots of equal length, denoted by $\mathcal{T} = \{t_1, t_2, \cdots\}$ with the same length Δ. At the beginning of each time slot t, a

set of microservice requests $\mathcal{K}_t$ are generated by different IoT devices, and they can be denoted by $\mathcal{K}_t = \{k_1, k_2, \cdots, k_{K_t}\}$.

Each microservice request k in $\mathcal{K}_t$ should be executed within a container and requires a specific image from $\mathcal{I} = \{i_1, i_2, \cdots, i_I\}$ to initialize this container, which can be denoted by $k = \{d_k, c_k, m_k, i_k, l_k\}$, where d_k is the size of the microservice, c_k is the total number of CPU cycles required to accomplish the computation microservice, m_k is the required memory of the microservice, $i_k \in \mathcal{I}$ is the required image of the microservice, and l_k is the maximum delay tolerance of the request. IoT devices can communicate with the edge node via a wireless connection to determine which node the microservice should be offloaded to.

B. Microservice Latency

In the process of a demand for microservice, there will inevitably be some delays, which directly affect the performance of the system and user experience. Major delays include communication latency, image download latency, and computation latency, which are critical in IoT and edge computing environments.

Communication latency. Regarding communication latency, we consider a communication model in which IoT devices share the bandwidth of edge nodes. The uplink wireless transmission rate $\xi_{n,k}(t)$ of the device of microservice require k to edge node n can be calculated using the following formula: $\xi_{n,k}(t) = \frac{B_n}{U_n(t)} \log\left(1 + \frac{p_k h_{n,k}}{\sigma^2}\right)$ where B_n is the bandwidth of edge node n and $U_n(t)$ is the number of microservice requires transmitted to node n at the time slot t. Besides, p_k is the transmission power, $h_{n,k}$ is the channel gain between the IoT device and the node, and σ represents the power of Gaussian white noise. The communication latency of microservice requires k to transmitted to node n can be defined as $T^{comm}_{n,k}(t) = d_k/\xi_{n,k}(t)$. Here, we assume the communication delay $T^{comm}_{n,k}(t)$ will not be larger than the duration of a time slot. That is $T^{comm}_{n,k}(t) \leq \delta$ for any $n \in \mathcal{N}$ and $k \in \mathcal{K}_t$. Furthermore, we overlook the return communication latency of results, as it can be considered relatively small compared to the microservice processing itself.

Image download latency. Image download delay refers to the delay in obtaining images related to microservice processing, which can be expressed by the following formula: $T^{down}_{n,k}(t) = x_{n,i_k}(t) \times \left(\frac{s_{i_k}}{B_n} + T^{queue}_n(t)\right)$ where i_k represents the image requested by microservice k and s_{i_k} represents the size of the image required to process microservice k, and $x_{n,i}(t) \in \{0, 1\}$ is a binary variable indicating whether the image is located on node n at the beginning of time slot t. When $x_{n,i}(t) = 0$, it means that the image i has been on node n, otherwise it is not on node n. $T^{queue}_n(t)$ represents the image download queue on node n at the time slot t, so if the image required for the microservice is already available locally on the node, then the download delay is 0.

Computation latency. Different microservices are executed in different containers, and all microservices are executed in parallel. The computation latency can be calculated as $T^{comp}_{n,k}(t) = c_k \cdot U_n(t)/F_n(t)$, where c_k is the number of CPU cycles requested by microservice k and $F_n(t)$ is the remaining CPU frequency of node n at the time slot t. Here, we assume the remaining CPU frequency will be evenly distributed to each arriving microservice.

C. Energy consumption.

Each microservice will be offloaded on a node n. After processing the microservice, the node returns the calculation result. Note that we ignore the transmission energy consumption of returning the calculated results from the node because in most cases the data volume is small and the downlink rate is higher. The overall power consumption related to microservice k can be categorized into two primary components. The first component pertains to the power consumed during the transfer of the microservice to the node. The second component accounts for the power consumed during the actual computation on the node. Based on the time it takes to upload the microservice, the power consumption of the microservice k upload can be defined as $E^{comm}_{n,k}(t) = p^{comm}_n \cdot T^{comm}_{n,k}(t)/U_n(t)$ where p^{comm}_n is the total power of transmission in node n. When the node is processing the uploaded computing microservice, the corresponding energy consumption of the node can be expressed as: $E^{comp}_{n,k}(t) = \frac{p^{comp}_n \cdot T^{comp}_{n,k}(t) \cdot F_n(t)}{U_n(t) \cdot F_n}$ where p^{comp}_n is the total power of computation in node n. It is not difficult to find that we adopt the strategy of equal distribution in transmission and computing power.

D. Problem Formulation and Analysis

For convenience, we define $y_{n,k}(t) \in \{0, 1\}$ as an indicator variable. When $y_{n,k}(t) = 1$, the microservice k is executed on edge node n at the time slot t; Otherwise, we have $y_{n,k}(t) = 0$. For each microservice k, it can be assigned to at most one edge node, then we have $\sum_{n \in \mathcal{N}} y_{n,k}(t) = 1$. For each node n, it has $U_n(t) = \sum_{k \in \mathcal{K}_t} y_{n,k}(t)$. Thus, the total latency $T_k(t)$ and total energy $E_k(t)$ to complete microservice k at the time slot t can be formulated as

$$\begin{aligned} T_k(t) &= \sum\nolimits_{n \in \mathcal{N}} y_{n,k}(t) \left[T^{comm}_{n,k}(t) + T^{down}_{n,k}(t) + T^{comp}_{n,k}(t)\right] \\ E_k(t) &= \sum\nolimits_{n \in \mathcal{N}} y_{n,k}(t) \left[E^{comm}_{n,k}(t) + E^{comp}_{n,k}(t)\right]. \end{aligned} \tag{1}$$

At the beginning of time slot t, we denote the remaining CPU frequency of node n as $F_n(t)$, the remaining memory of node n as $M_n(t)$, and the remaining storage space of node n as $D_n(t)$, respectively. Following this, the constraints in our model can be defined.

- Delay constraint: the total delay to finish the microservice k cannot exceed its maximum tolerance. That is

$$T_k(t) \leq l_k, \forall k \in \mathcal{K}_t. \tag{2}$$

- Memory constraint: The total memory on each node is limited. That is

$$\sum\nolimits_{k \in \mathcal{K}_t} y_{n,k}(t) \cdot m_k \leq M_n(t), \forall n \in \mathcal{N}. \tag{3}$$

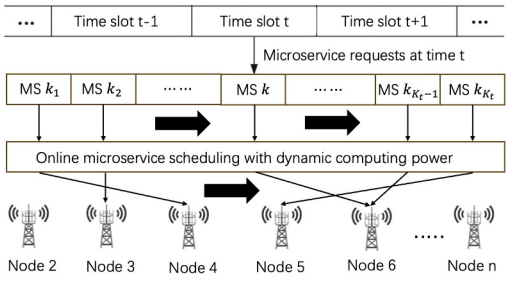


Fig. 2. Overview of the system model, in each time slot t, there will be K_t microservices that need to be offloaded to acceptable edge nodes at the same time. However, in the actual scheduling, it will be processed in a sequential manner, which will be described in Sec. IV-B.

- Storage constraint: The total storage on each node is limited. That is

$$\sum\nolimits_{k\in\mathcal{K}_t} y_{n,k}(t)\cdot x_{n,i_k}(t)\cdot s_{i_k} \le D_n(t), \forall n\in\mathcal{N}. \quad (4)$$

Generally, once a microservice is finished, the memory and CPU frequency it occupies will be released. However, once an image is downloaded to a node, it will always exist on this node and is allowed to be shared by multiple microservices. Thus, it implies $x_{n,i}(t') \le x_{n,i}(t)$ if $t' \ge t$.

We aim to minimize the weighted overall microservice processing latency and energy consumption from a long-term perspective. The decision process of the microservice scheduling with dynamic computing power is shown in Fig. 2. At each time slot t, there are totally K_t microservices that should be determined at the same time. We use α to represent the weight assigned to latency at the time slot t. The target is to find the best strategy which can minimize the overall cost while obeying the constraints. Therefore, the objective of our online microservice scheduling problem can be defined as

$$\begin{aligned} &\min \sum\nolimits_{t\in\mathcal{T}}\sum\nolimits_{k\in\mathcal{K}_t}(\alpha\cdot T_k(t)+E_k(t)) \\ &\text{s.t. Constraint (2), (3), and (4);} \\ &\quad x_{n,i}(t)\in\{0,1\}, \forall n\in\mathcal{N}, \forall i\in\mathcal{I}; \\ &\quad x_{n,i}(t+1)\le x_{n,i}(t); \\ &\quad y_{n,k}(t)\in\{0,1\}, \forall n\in\mathcal{N}, \forall k\in\mathcal{K}_t, \forall t\in\mathcal{T}; \\ &\quad \sum\nolimits_{n\in\mathcal{N}} y_{n,k}(t)=1, \forall k\in\mathcal{K}_t. \end{aligned} \quad (5)$$

Obviously, the problem defined in (5) is NP-hard because it is a special case of online integer programming. RL-based methods can get a good solution by continuously interacting with the environment.

IV. Algorithm Design

A. Algorithm Settings

State. The system state s_t^k encompasses both the node resource state and microservice state. In each time slot t, microservices in $\mathcal{K}_t$ are processed sequentially. The node resource state, denoted as s_t^{node}, represents the remaining resources at the beginning of time slot t. Here we denote the remaining memory and CPU frequency of the n-th node at time slot t as $M'_n(t)$ and $F'_n(t)$, respectively. The resource state includes the remaining CPU frequency, remaining memory capacity, storage capacity, communication power, computation power, and bandwidth of the node at time slot t. This state can be defined as:

$$\begin{aligned} s_t^{node} = \{&F'_1(t),\dots,F'_{|N|}(t), M'_1(t),\dots,M'_{|N|}(t), \\ &D_1(t),\dots,D_{|N|}(t), P_1^{comm},\dots,P_{|N|}^{comm}, \\ &P_1^{comp},\dots,P_{|N|}^{comp}, B_1,\dots,B_{|N|}\}. \end{aligned} \quad (6)$$

The microservice state $s_t^{ms,k}$ of microservice $k\in\mathcal{K}_t$ at time slot t encompasses information about the required image on each node, $y_{n,k}(t)$ for each node $n\in\mathcal{N}$ and details about the microservice, including the required image, the required CPU cycles, the microservice size, and the maximum delay tolerance. Thus, the microservice state can be expressed as:

$$s_t^{ms,k} = \{y_{1,k}(t),\dots,y_{|N|,k}(t), d_k, c_k, m_k, i_k, l_k\}. \quad (7)$$

Thus, the system state s_t^k is $s_t^k = s_t^{node}\cup s_t^{ms,k}$.

Action space. The agent needs to determine which node n to assign for the microservice k. Therefore, the action space is the set of all nodes as $a_t^k\in A=\{1,2,\cdots,|N|\}$.

Reward. Defining a proper reward is crucial in the RL algorithm. To minimize the long-term energy consumption of the entire system, we utilize negative energy consumption as the reward. Besides, The reward value should also satisfy the constraint conditions. If Constraint (2) is violated, we set the reward as a punitive negative directly. After making a decision a_t^k, the $T_k(t)$ and $E_k(t)$ can be determined by Eqn. (1). So, the reward for processing microservice k at time slot t can be defined as $r_t^k = \alpha\cdot(l_k - T_k(t)) - E_k(t)$, where α is used to balance the importance of latency optimization and energy consumption. That allows the algorithm to find the optimal strategy that minimizes the energy cost while increasing the microservice completion rate within the specified delay.

B. Policy Network

In each time slot t, the decision a_t^k is made according to its observation s_t^k, however, this s_t^k does not contain any information about other microservices required to be dealt with in this time slot. In other words, there are total $\mathcal{K}_t$ microservices that need to be addressed concurrently. For example, the remaining resources $F'_n(t)$, $M'_n(t)$, and $D_n(t)$ of node n are sufficient at the beginning of time slot t, if there are too many microservices eventually assigned to this node, then it will lead to low efficiency. Thus, we need to design carefully to overcome this drawback and achieve an efficient dynamic scheduling. Since the resources of edge nodes are dynamically allocated, the environment will not change until action decisions of all microservices in $\mathcal{K}_t$ are completed. This means that the microservice scheduling decisions made in a time slot t are made sequentially, forming a time series as shown in Fig. 2. This time series information destroys the Markov nature of the environment, making it impossible for the agent to generate the best action based solely on the currently observed state s_t^k.

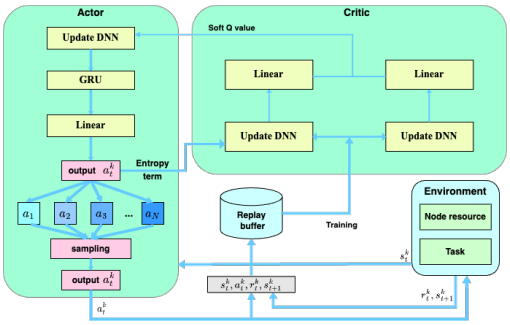


Fig. 3. Overview of the SAC-based framework.

For feature extraction of sequence information in microservice scheduling, we consider adding a unit of RNN-based structure in the policy network, which can learn the time dependence between states of previous microservices. By utilizing GRU, we can comprehensively integrate information from previous microservice scheduling decisions in this time slot into the current state. Thus, we can avoid making bad decisions because of the conflict with previous decisions. Specifically, a typical GRU includes two control gates: an update gate (z_t^k) and a reset gate (re_t^k). The update gate (z_t^k) is used to control the degree to which the state information $h_t^{k-1}(cu_t^{k-1})$ at the previous $k-1$ microservice will be brought into the current microservice k. The larger the value of the update gate, the more status information from the previous microservice is carried in. Reset gate (re_t^k) controls how much information from previous states is written to the current candidate set $\hat{cu}_t^k$. The smaller the reset gate (re_t^k), the less information is written to the previous state. The updated equation is calculated as follows: $re_t^k = \sigma(W_{xr} \cdot x_t^k + W_{hr} \cdot h_t^{k-1} + b_r)$, $z_t^k = \sigma(W_{xz} \cdot x_t^k + W_{hz} \cdot h_t^{k-1} + b_z)$, $\hat{cu}_t^k = \tanh(W_{xc} \cdot x_t^k + W_{hc} \cdot (re_t^k \otimes h_t^{k-1}) + b_c)$, $cu_t^k = (1 - z_t^k) \otimes cu_t^{k-1} + z_t^k \otimes \hat{cu}_t^k$, and $h_t^k = cu_t^k$, where w_{xr}, w_{hr}, w_{xz}, and w_{hz} is the network weight matrix, b_r and b_z are bias vectors, and re_t^k and z_t^k are vectors for updating and resetting activation values of gates.

It is worth noticing that the GRU module only needs to focus on the time series feature in a single time slot. Therefore, at the beginning of each time slot, the hidden state of the GRU module h_t^0 would be reset into the same initial value. As shown in Fig. 3, the feature extraction layer extracts the input state features and updates its own hidden state. In addition, state features are input into the hidden linear layer to obtain the final output of the policy network.

C. Soft Actor-Critic (SAC)

The SAC algorithm operates within a Markov Decision Process (MDP) framework, where an agent interacts with an environment over discrete time steps. Each microservice k at time slot t involves observing state s_t^k, taking action a_t^k, receiving reward r_t^k, and transitioning to state s_{t+1}^k. SAC aims to maximize long-term rewards by adjusting policy entropy to balance exploration and exploitation. It uses a maximum entropy reinforcement learning framework to find the optimal policy π, mapping states to action distributions probabilistically to encourage exploration. The Bellman equation is crucial in MDP, relating soft action values to state value functions. Consequently, the relationship between the state s_t^k, action a_t^k, and time slot t for the considered microservice k can be described based on the Bellman equation:

$$Q^\pi(s_t^k, a_t^k) = r(s_t^k, a_t^k) + \gamma \cdot \mathbb{E}_{s_{t+1}^k \sim p}[V^\pi(s_{t+1}^k)] \quad (8)$$

and the value function as

$$V^\pi(s_t^k) = \mathbb{E}_{a_t^k \sim \pi(\cdot|s_t^k)}[Q^\pi(s_t^k, a_t^k) - \log \pi(a_t^k|s_t^k)] \quad (9)$$

where p denotes the trajectory distribution made by π, $r(s_t^k, a_t^k)$ is the reward gained by the state and action space at time slot t for microservice k, and γ is the discount factor in the equation. We consider $Q^\pi(s_t^k, a_t^k) = Q_\omega(s_t^k, a_t^k)$ in the DNN where ω denotes the network parameters. The Q-function parameters are not constant, and the actor and critic are further updated according to the replay buffer regarding action and immediate reward in the replay buffer. Thus, the parameters can be trained by minimizing the loss function $J_Q(\omega) =$

$$\mathbb{E}_{(s_t^k, a_t^k, r_t^k, s_{t+1}^k) \sim rb}\left[\frac{1}{2}(Q_\omega(s_t^k, a_t^k) - Q_{\omega'}(s_t^k, a_t^k))^2\right] \quad (10)$$

where $Q_\omega(s_t^k, a_t^k)$ denotes the soft Q-value, rb denotes replay buffer, and

$$Q_{\omega'}(s_t^k, a_t^k) = R(s_t^k, a_t^k) + \gamma \cdot V_{\theta'}(s_{t+1}^k). \quad (11)$$

In order to stabilize the iteration of the action value function, the Eqn. (11) defines a target action-value function, in which ω' is obtained through an exponential moving average of ω. The performance of DRL algorithms is heavily dependent on the policy. If the policy is optimal, the assigned microservice will be completed within the specified time, and the overall energy consumption will be reduced through reasonable microservice scheduling. Conversely, if the strategy is not optimal, microservice timeouts will become common, and overall energy consumption will be higher. Therefore, in order to improve the strategy, it is updated according to the Kullback-Leibler divergence:

$$\pi_t = \arg\min_{\pi' \in \Pi} D_{KL}\left(\pi'(\cdot|s_t^k)||\frac{\exp((\frac{1}{\Delta})Q^\pi(s_t^k, \cdot)}{Z^\pi(s_t^k)}\right). \quad (12)$$

In the provided text, Π represents a collection of policies that correspond to Gaussian parameters, and the D_{KL} term quantifies the information loss during the approximation process. The Kullback–Leibler divergence can be reduced further by adjusting the policy parameters through the following policy parameter updates: $J_\pi(\theta) =$

$$\mathbb{E}_{s_t^k \sim rb}[\mathbb{E}_{a_t^k \sim \pi_\theta}[\Delta \log(\pi_\theta(a_t^k|s_t^k)) - Q_\omega(s_t^k, a_t^k)]]. \quad (13)$$

The policy iteration is continued until it reaches the optimal value and converges with the maximum entropy. The interactions between the environment, the value function, and the policy network are shown in Fig. 3. The detailed SAC-based microservices offloading algorithm is given in Algorithm 2.

Algorithm 1: Action Selection

Get action distribution $P^{\pi}(s_t^k)$ from actor network;
for $n \in \mathcal{N}$ **do**
  get u_n and $u_t^k[n] \leftarrow u_n$;
$P_t^k \leftarrow P^{\pi}(s_t^k) \odot u_t^k$ and regularize P_t^k;
Sample a_t^k from P_t^k;
return Return a_t^k;

D. Action Selection

When selecting actions, the agent samples according to the probability output of the policy network, and it does not judge whether the action is reasonable. Some constraints should be added to the action selection process to avoid some of those unacceptable actions. This problem was solved by defining an Action Mask u_t^k to fit the constraints. The mask is a one-dimensional Boolean vector with the same length as the action space A. This Action Mack is updated before each action selection, its value depends on the feasibility of a specific action in the whole actions space. An example of an action mask $u_t^k = [1, 0, 1, 0, 0, 1, \cdots, u_{n-1}, u_n]$. By employing a mask to set a portion of the action probabilities generated by the policy network to zero, we ensure that these nodes will not be chosen as actions. The calculation of each Boolean value u_n in the Action Mask u_t^k will be shown in the following.

First, for the node n, if there are microservices already processing on it and it has no idle CPU cycle, then the node is considered a busy node, which can be obtained as $u_n^c = \mathbb{I}\{F_n(t) > 0\}$, where $\mathbb{I}(\cdot)$ is an indicator function and $u_n^c = 1$ means that the node is acceptable, otherwise it could not be selected as an action. Second, when the node is experiencing a shortage of memory and is not available for the microservices k, which can be expressed as $u_n^m = \mathbb{I}\{M_n(t) - m_k > 0\}$. The storage considerations for images are nearly the same, but it's crucial to assess whether the node already contains the microservices k required image i_k. Thus, we have $u_n^i = \mathbb{I}\{\{M_n(t) - m_k > 0\} \wedge \{x_{n,i} == 1\}\}$. If u_n^c, u_n^m, and u_n^i are all equal to 1, the action is acceptable. Otherwise, it is not a good action. To sum up, each Boolean value u_n in u_t^k can be summarized as $u_n = u_n^c \cdot u_n^m \cdot u_n^i$. The action selection is shown in Algorithm 1.

V. Numerical Simulations

A. Experimental Settings

Parameter Settings. All IoT devices are heterogeneous and randomly distributed, and the default number of nodes is 15. The node's CPU frequency is set between $[3, 6.5]$ GHz and memory is set between $[80, 180]$ GB. The network bandwidth is limited to between $[2, 6]$ Gbps. Each image requires a maximum of 10 MB, the maximum microservice size is set to 100 MB, and the maximum transfer power is set to 0.0001995.

The hyperparameters used in the SAC-based framework include a replay memory size of 30,000, an Actor network with a GRU hidden dimension of (128, 64) and fully connected layers of (64, 64, 16), and an Actor learning rate of 1e-4. The Critic network has fully connected layers of (64, 64, 16) with a learning rate of 3e-4. The Adam optimizer is used, and the discount factor is set to 0.99.

Algorithm 2: SAC-Based Microservices Offloading

Initialize: $Q_{\omega_1}(s,a)$, $Q_{\omega_2}(s,a)$, $Q_{\omega_1'}$, and $Q_{\omega_2'}$ with weights $\omega_1' = \omega_1$ and $\omega_2' = \omega_2$;
Initialize: policy $\pi_\theta(a|s)$ with weight θ;
for *each epoch* **do**
  Retrieve current state s_1^{node};
  for $t \in \{t_1, t_2, \cdots\}$ **do**
    Obtain state of node s_t^{node} form environment;
    Initialize: hidden state of GRU h_t^0;
    for $k \in \{k_1, k_2, \cdots, \mathcal{K}_t\}$ **do**
      Examine the required image i_k, get $s_t^{ms,k}$;
      Get offloading action a_t^k by Algorithm 1;
    for $k \in \{k_1, k_2, \cdots, \mathcal{K}_t\}$ **do**
      Compute reward r_t^k and estimate state s_{t+1}^k;
      Save $(s_t^k, a_t^k, r_t^k, s_{t+1}^k)$ in replay memory;
  Update $J_Q(\omega)$ and θ by using $\theta \leftarrow \theta + \eta_a \cdot \nabla_\theta J_\pi(\theta)$ and $\omega \leftarrow \omega + \eta_c \cdot \nabla_\omega J_Q(\omega)$;
  Update soft action value function ω';

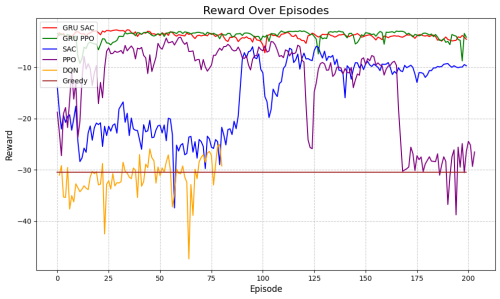
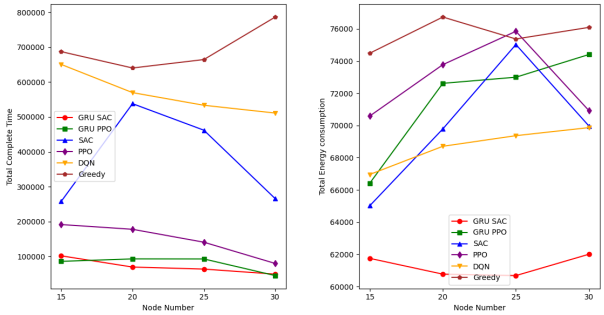


Fig. 4. Reward comparison by using different baselines.

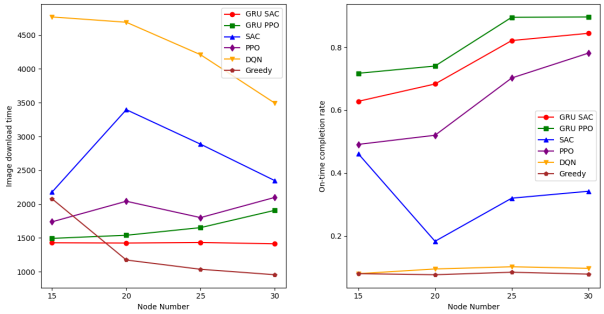
Baselines. To compare performance, several baselines were performed. The details are as follows.

- Greedy Approach: A greedy algorithm will select the node n as action a_t^k and minimize the time that microservice k waits for the image $x_{n,i}$ to download.
- DQN [19]: The deep Q-network contains two fully connected layers, which are (128, 64) and (64, 16).
- PPO [20]: The policy network and value network both contain two fully connected layers, which are (128, 64) and (64, 16).
- SAC [10]: The policy network and value network both contain two fully connected layers, which are (128, 64) and (64, 16).
- GRU-PPO: An algorithm based on PPO, which implements the policy network mentioned in this paper.



(a) Total Complete Time

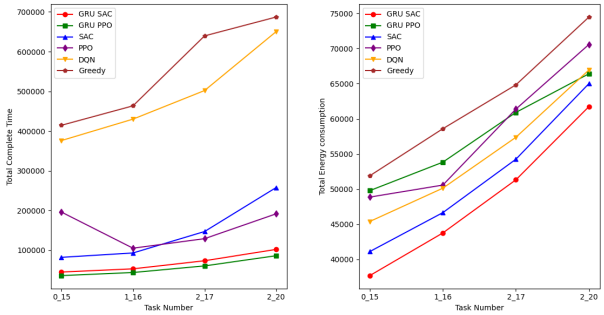
(b) Total Energy Consumption



(c) Image Download Time

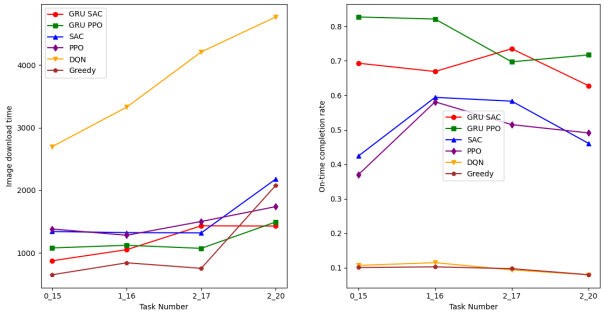
(d) On-time Completion Rate

Fig. 5. Results with the varying number of nodes in different algorithms.



(a) Total Complete Time

(b) Total Energy Consumption



(c) Image Download Time

(d) On-time Completion Rate

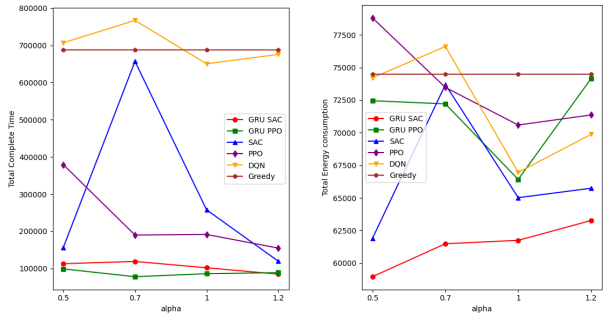
Fig. 6. Results with the varying number of tasks in different algorithms.

B. Experimental results

We present the convergence results for GRU-SAC, GRU-PPO, SAC, PPO, DQN, and Greedy as shown in Fig. 4 when the number of nodes is 15, the number of microservices in each time slot is between 2 and 20, and $\alpha = 1$. First, we observe that GRU-SAC becomes stable after 40 episodes, showing the GRU-SAC's convergence. It can be seen that the reward of GRU-PPO also converges fast, but the average reward is lower than that of our GRU-SAC. PPO and SAC are implemented with a fully connected network, which converges in a lower reward after 75 and 110 episodes, respectively. It is worth noting that PPO overfits after 160 episodes and reward plummets, indicating some instability in this approach. Since Greedy implements a fixed strategy, its reward in each episode will not change. After all, through training with multiple rounds for all policies, the reward of GRU-SAC is better than other baselines.

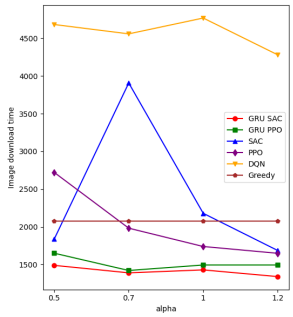
Performance with the different number of nodes. We show the impact of the number of nodes on total completion time, total energy consumption, total image download time, and on-time completion ratio in Fig. 5. As shown in Fig. 5(a), GRU-SAC and GRU-PPO perform better as the GRU module prevents efficiency degradation caused by multiple microservices being offloaded to the same node in each time slot. The efficiency improvement becomes more pronounced with an increased number of edge nodes. Although the delay performance of GRU-SAC and GRU-PPO is similar, GRU-SAC significantly reduces energy consumption, as shown in Fig. 5(b), and consistently outperforms other baseline algorithms. For image download latency, Greedy algorithm performs best when the number of nodes exceeds 20. However, with fewer nodes (e.g., 15), images tend to concentrate on certain nodes. Due to memory limitations, a node cannot accept new microservices without sufficient memory, leading to time spent re-downloading images on other nodes. As the number of nodes increases, the on-time completion rate improves, as shown in Fig. 5(d). Although GRU-PPO slightly outperforms GRU-SAC in this regard, we believe GRU-SAC is superior due to its better control over power consumption.

Performance with the different number of services. Figure 6 depicts the impact of varying service numbers per time slot, with a maximum microservice size set to 100 MB. The horizontal axis ranges from 2 to 20 service requests per slot. In Fig. 6(a), total completion time increases with more requests, with GRU-SAC outperforming SAC by nearly 75% when 2 to 20 requests are processed, indicating the beneficial role of GRU in scheduling. GRU-SAC also shows efficient energy use (Fig. 6(b)), outperforming the greedy algorithm for image download times (Fig. 6(c)) with higher request volumes. Fig. 6(d) illustrates a decrease in on-time completions as service requests increase. It is worth noting that when the number of services is small, the on-time complete ratio of GRU-PPO is higher than GRU-SAC. Fewer service requests imply a relatively stable environment under which the more stable policies generated by GRU-PPO can perform better. This is a great discovery of this experiment.

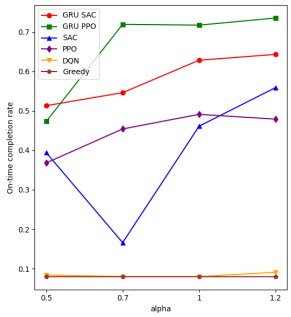


(a) Total Complete Time

(b) Total Energy Consumption



(c) Image Download Time



(d) On-time Completion Rate

Fig. 7. Results with varying α in reward function in different algorithms.

Performance with the different α in the reward function. We show the impact of different α in Fig. 7. The α works well in our model as a parameter that balances the latency and energy consumption. From Fig. 7(a) and Fig. 7(b), it could be found that with the increase of the weight of delay, the total complete time decreases while the energy consumption increases simultaneously. By observing the iteration, we found that When the delay weight is substantial, GRU-PPO will explore the action probability distributions with higher on-time completion rates in the early stages of training. However, it stabilizes the policy there prematurely, without adequately exploring action probability distributions that lead to lower energy consumption. In contrast, GRU-SAC explores the action space more comprehensively by maximizing entropy in its policy, thereby finding a globally optimal solution. Taken together, the GRU-SAC always has the best performance.

VI. Conclusion

In this paper, we consider an online container-based microservice scheduling problem with dynamic computing power, which aims to minimize the total delay and energy consumption. First, we model this problem comprehensively, considering edge nodes' dynamic resource allocation, latency, and energy consumption. Second, a GRU-based feature extraction method is proposed to extract the time-series information and dependency among decisions of microservice placements in the time slot. Third, a SAC-based RL algorithm combined with our designed GRU unit in the policy network is proposed to make online decisions for our problem. Finally, extensive numerical experiments have been conducted to prove that our proposed RL algorithm can efficiently learn the optimal offload scheduling strategy without prior knowledge of the dynamic environment. Our algorithm reduces the total objective by 65% compared with other baselines, and it has outperformed in total delay and energy consumption.

References

[1] W. Z. Khan, E. Ahmed, S. Hakak, I. Yaqoob, and A. Ahmed, "Edge computing: A survey," *Future Generation Computer Systems*, vol. 97, pp. 219–235, 2019.

[2] E. Wolff, *Microservices: flexible software architecture*. Addison-Wesley Professional, 2016.

[3] I. Nadareishvili, R. Mitra, M. McLarty, and M. Amundsen, *Microservice architecture: aligning principles, practices, and culture*. O'Reilly Media, Inc., 2016.

[4] R. Zhou, Z. Li, and C. Wu, "Scheduling frameworks for cloud container services," *IEEE/acm transactions on networking*, vol. 26, no. 1, pp. 436–450, 2018.

[5] [Online]. Available: https://www.docker.com/

[6] [Online]. Available: https://kubernetes.io/

[7] M. Fazio, A. Celesti, R. Ranjan, C. Liu, L. Chen, and M. Villari, "Open issues in scheduling microservices in the cloud," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 81–88, 2016.

[8] H. Cui, Z. Tang, J. Lou, and W. Jia, "Online container scheduling for low-latency iot services in edge cluster upgrade: A reinforcement learning approach," Jul 2023.

[9] R. V. Rasmussen and M. A. Trick, "Round robin scheduling–a survey," *European Journal of Operational Research*, vol. 188, no. 3, pp. 617–636, 2008.

[10] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," in *International conference on machine learning*. PMLR, 2018, pp. 1861–1870.

[11] M. Abdullah, W. Iqbal, and A. Erradi, "Unsupervised learning approach for web application auto-decomposition into microservices," *Journal of Systems and Software*, vol. 151, pp. 243–257, 2019.

[12] V. Singh and S. K. Peddoju, "Container-based microservice architecture for cloud applications," in *2017 International Conference on Computing, Communication and Automation (ICCCA)*. IEEE, 2017, pp. 847–852.

[13] X. Wan, X. Guan, T. Wang, G. Bai, and B.-Y. Choi, "Application deployment using microservice and docker containers: Framework and optimization," *Journal of Network and Computer Applications*, vol. 119, pp. 97–109, 2018.

[14] B. Tan, H. Ma, and Y. Mei, "A nsga-ii-based approach for multi-objective micro-service allocation in container-based clouds," in *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. IEEE, 2020, pp. 282–289.

[15] Z. Wen, T. Lin, R. Yang, S. Ji, R. Ranjan, A. Romanovsky, C. Lin, and J. Xu, "Ga-par: Dependable microservice orchestration framework for geo-distributed clouds," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 1, pp. 129–143, 2019.

[16] S. Wang, Y. Guo, N. Zhang, P. Yang, A. Zhou, and X. Shen, "Delay-aware microservice coordination in mobile edge computing: A reinforcement learning approach," *IEEE Transactions on Mobile Computing*, vol. 20, no. 3, pp. 939–951, 2019.

[17] L. Chen, Y. Xu, Z. Lu, J. Wu, K. Gai, P. C. Hung, and M. Qiu, "Iot microservice deployment in edge-cloud hybrid environment using reinforcement learning," *IEEE Internet of Things Journal*, vol. 8, no. 16, pp. 12 610–12 622, 2020.

[18] W. Lv, Q. Wang, P. Yang, Y. Ding, B. Yi, Z. Wang, and C. Lin, "Microservice deployment in edge computing based on deep q learning," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 11, pp. 2968–2978, 2022.

[19] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[20] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.