Global Microservice Autoscaling Over Heterogeneous Edge Environments for Internet Applications: A Reinforcement Learning Approach

Kai Peng, Jie Rao, Hao Li, Yi Hu, *Graduate Student Member, IEEE*, Bo Jin, Tianyue Zheng, *Member, IEEE*, and Menglan Hu

***Abstract*—The integration of microservice architecture and edge computing offers innovative solutions for highly interactive, low-latency Internet applications. To manage the dynamic nature of requests in edge computing, microservice autoscaling techniques are frequently employed. However, the resource limitation of individual edge servers and the heterogeneity among edge servers present significant challenges for autoscaling in edge computing. Meanwhile, few studies have considered the long-term optimization and the joint optimization of instance adjustment and request routing in edge computing. This article aims to fill these gaps. First, we propose global horizontal pod autoscaler (GHPA), a novel framework that addresses microservice autoscaling from the perspective of edge server clusters. Second, we consider the joint optimization of instance adjustment and request routing, and formulate a long-term optimization problem. Third, we transform the long-term optimization problem into a Markov Decision Problem (MDP) and use reinforcement learning techniques to solve it. Finally, we conduct extensive experiments using both real and synthetic data. The experiment results demonstrate that our algorithm achieves at least a 10% performance improvement in various test environments compared to state-of-the-art algorithms.**

***Index Terms*—autoscaling, edge computing, Microservice architecture, reinforcement learning (RL), service deployment.**

I. Introduction

MICROSERVICE architecture is a modern software development approach that decouples monolithic applications into a collection of loosely coupled, independently deployable microservices to promote modularity, scalability, and resilience [1]. This decoupled architecture has been widely employed for Internet applications by giants, including Microsoft and Alibaba [2], [3]. Internet applications are characterized by scalability, delay sensitivity, and high concurrency. As a result, an increasing number of microservices are deployed at the network edge to reduce latency, enhance real-time responsiveness, and minimize bandwidth consumption [4], [5].

The integration of edge computing with microservice architectures leverages the respective advantages of both paradigms, thereby providing a flexible, scalable, and low-latency framework for data processing and management in distributed environments. However, the highly dynamic workload in edge computing may lead to uneven resource allocation, degraded service quality, and reduced system reliability. Microservice autoscaling, which facilitates the dynamic adjustment of microservice instances in response to real-time demand, has emerged as a critical strategy for addressing workload variability in cloud computing. Nonetheless, the implementation of microservice autoscaling in edge computing for Internet applications presents substantial challenges, and this area remains underexplored in the existing literature.

First, efficient microservice autoscaling needs to consider the heterogeneity across different edge servers and the resource constraints of individual servers. Different edge servers are distributed across various geographic locations and possess different resources, often necessitating distinct autoscaling strategies. For example, frequently accessed microservices should be deployed closer to the user, whereas less frequently accessed microservices can be deployed further away. Additionally, edge servers with limited resources are often unsuitable for hosting larger-scale microservices. However, heterogeneity and resource constraints are usually overlooked in previous research. Numerous studies [6], [7], [8], [9] have implemented autoscaling based on Kubernetes. Since these studies only focus on single edge server, they struggle to effectively address resource heterogeneity and resource constraints in edge computing environments.

Second, microservice autoscaling necessitates the joint optimization of instance adjustment and request routing. These two aspects exhibit strong coupling characteristics: the microservice instance adjustment strategy is influenced by request routing while simultaneously affecting routing design decisions. Current research predominantly fails to account for this intrinsic coupling relationship. Most existing approaches focus solely on microservice instance adjustment without corresponding adaptations to request routing

Received 4 June 2025; accepted 9 July 2025. Date of publication 23 July 2025; date of current version 25 September 2025. This work was supported in part by the National Natural Science Foundation of China under Grant 62171189; and in part by the Key Research and Development Program of Hubei Province, China, under Grant 2024BAB031, Grant 2024BAB016, and Grant 2024BAA011. *(Corresponding author: Menglan Hu.)*

Kai Peng, Jie Rao, Hao Li, Yi Hu, and Menglan Hu are with the Hubei Laboratory of Internet of Intelligence, School of Electronic Information and Communications, Huazhong University of Science and Technology, Wuhan 430074, China (e-mail: pkhust@hust.edu.cn; m202473249@hust.edu.cn; lihao_eic@hust.edu.cn; hust_huyi@hust.edu.cn; humenglan@hust.edu.cn).

Bo Jin is with the Information Center, State Grid Hubei Information and Telecommunication Company, Wuhan 430077, China (e-mail: jinbo724@126.com).

Tianyue Zheng is with the Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen 518055, China (e-mail: zhengty@sustech.edu.cn).

Digital Object Identifier 10.1109/JIOT.2025.3589684

strategies [10], [11], [12]. Such isolated optimization methods inevitably lead to suboptimal autoscaling performance.

Third, efficient microservice autoscaling requires consideration of long-term optimization. Each adjustment of microservice instances incurs certain dynamic costs. For instance, adding or removing an instance requires time, during which computational resources remain unavailable. Consequently, large-scale adjustments are impractical in real-world production environments. Instead, minimal adjustments should be employed to achieve maximum benefits. Some studies addressed workload variability by dividing time into segments [13], [14], [15], thereby transforming dynamic scenarios into a series of static scenarios. Within each time segment, a static service deployment algorithm is executed to handle dynamic requests. However, this approach often results in significant changes to the deployment of microservice instances even in response to minor variations in requests. Although these methods may yield favorable results within a single timeslot, it is ultimately unacceptable from a long-term perspective because of the huge dynamic costs.

Motivated by the aforementioned needs and difficulties, this article proposes a cluster-based microservice autoscaler termed global horizontal pod autoscaler (GHPA) to address resource heterogeneity and resource constraints in edge computing. To accomplish this, several challenges must be addressed. First, the complex intercommunication between microservices needs to be managed. In scenarios integrating microservice architecture and edge computing, the invocation relationships between microservices become even more intricate. Second, joint optimization of instance adjustment and request routing must account for both dynamic deployment strategies and adaptive routing mechanisms to achieve optimal system performance. Finally, long-term optimization poses a significant challenge. Conventional methods struggle to solve long-term optimization problems, necessitating the design of an appropriate approximation algorithm.

In summary, our work makes the following contributions.

1) We propose a global microservice autoscaler GHPA that operates at the level of edge server clusters. Within the cluster, one server is designed as the master server, referred to as the edge cloud, responsible for data collection and decision-making, while the remaining edge servers act as slavers, executing the actions.
2) We consider the joint optimization of instance adjustment and request routing. Then, we employ undirected graphs along with open Jackson queueing networks to model the long-term optimization problem.
3) We transform the long-term optimization problem into a Markov decision process (MDP) and employ a reinforcement learning (RL) algorithm to handle long-term optimization. The RL agent is deployed on the master server, collecting data from the slave servers and learn an autoscaling policy through continuous interaction with the environment.
4) We compare our algorithm with the most popular existing methods using extensive real-world and simulated data. Experimental results demonstrate that our algorithm GHPA achieves at least 10% performance improvement.

The remainder of this article is organized as follows. Section II reviews the related work. In Section III, we introduce the cluster-based autoscaling mechanism and the multiobjective long-term optimization problem. Section IV details the proposed GHPA algorithm. Section V provides the performance evaluation, and we conclude this article in Section VI.

II. Related Work

This section reviews the related work on microservices deployment and autoscaling.

A. Microservice Deployment

In cloud computing, Singh and Peddoju [16] proposed an automated system for deploying and integrating microservices using Docker containers, demonstrating improved performance and reduced effort compared to monolithic designs through a social networking case study. Wan et al. [17] explored optimizing application deployment in cloud data centers using microservice architecture and Docker containers to minimize costs while meeting service delay requirements. Xu et al. [18] and Hu et al. [19] utilized queuing networks for modeling and jointly optimize microservice deployment and request routing to enhance performance and reduce latency in cloud data centers. As for edge computing, Deng et al. [20] proposed a novel approximation algorithm to optimize the deployment of microservice instances in resource-limited MEC. Guo et al. [21] crafted a multiobjective evolutionary approach named MSCMOE to tackle the problem of joint optimization of service latency and deployment cost in MEC. Hu et al. [22] and Peng et al. [23] considered the joint optimization of microservice deployment and request routing in edge computing, significantly reducing communication costs between microservices. Lv et al. [24], Peng et al. [5], Hu et al. [25] employed RL to optimize microservice deployment in edge environment.

Microservice deployment and microservice autoscaling share certain similarities. However, microservice deployment addresses a static problem, whereas microservice autoscaling tackles the issue of dynamic requests. Therefore, the methods used for microservice deployment cannot be directly applied to microservice autoscaling.

B. Microservice Autoscaling

In cloud computing, Ding and Huang [26], and Hossen et al. [27] used CPU utilization as the primary metric for scaling decisions. Expanding on the reactive model, Zhang et al. [28] and Kwan et al. [29] incorporated both CPU and memory utilization to determine the appropriate scaling actions for microservices. Moreover, studies by Qiu et al. Abdullah et al. [30], Iqbal et al. [31], and Prachitmutita et al. [32] employed a proactive approach to predeploy microservice instances by predicting request arrival rates using methods, such as elastic net regression, linear regression, and LSTM networks. Xu et al. [33] analyzed the

TABLE I
NOTATION AND TERMINOLOGY

Notation	Definition
E	the set of edge servers (E_n denotes the n-th edge server)
S	the set of microservice (S_i denotes the i-th microservice)
R	the set of application requests (R_k denotes the k-th application requests)
G_l	timeslot l
$V_{m,n}$	the transmission velocity between E_m and E_n
$d_{m,n}$	the distance between E_m and E_n
$data_{i,j}$	communication data between S_i and S_j
$\Lambda_m^k(l)$	the arrival rate of R_k in E_m during G_l
$N_m^i(l)$	the number of instances of S_i deployed in E_m during G_l
$U_m^i(l)$	the utilization ratio of instances of S_i deployed in E_m during G_l
$\lambda_m^i(l)$	the arrival rate of S_i in E_m during G_l
$\mu_m^i(l)$	the service rate of a single instance of S_i in E_m during G_l
$\rho_m^i(l)$	the load of S_i in E_m during G_l
$P_m(l)$	the energy consumption of E_m during G_l
$T_m^k(l)$	the service latency of R_k in E_m during G_l
$T_{latency}(l)$	the average service latency of all requests during G_l
$E_{cost}(l)$	the energy consumption of all servers during G_l

currently prevalent dominant scaling techniques (horizontal scaling, vertical scaling, brownout) and demonstrated that the RL methods can achieve better results. Wang et al. [34] comprehensively considered both resource wastage and service level objective (SLO) assurances, thereby rendering the RL methods more applicable to real-world production environments. As for edge computing. Phuc et al. [7], Kundroo et al. [8], and Ju et al. [9] implemented autoscaling based on Kubernetes by modifying the evaluation metrics within Kubernetes to better meet the requirements of edge computing. Focusing on long-term benefits, Cheng et al. [14] decomposed the entire optimization problem into several per-timeslot subproblems and solved them individually. Cheng et al. [11] designed two components, an autoscaler and a burst handler, to handle predictable workloads and unpredictable request bursts, respectively.

The aforementioned research on microservice autoscaling mostly neglected the heterogeneity among edge servers. Moreover, few studies have taken into account the scenarios of multi-instances and instance multiplexing.

III. PROBLEM FORMULATION

A. System Models

We apply microservice architecture in edge computing. We utilize an undirected graph $H(E, V)$ to represent the edge server distribution and use $E = \{E_1, E_2, \ldots, E_M\}$ to denote the set of edge servers. Each server is subjected to a maximum computational resource constraint and a maximum memory constraint, represented by E_m.cpu and E_m.mem, respectively. In our study, we disregard the specific connections between servers and instead use $V_{m,n}$ to uniformly represent the transmission velocity between servers E_m and E_n.We summarize used notations in Table I.

B. Microservice Models

In a microservice architecture, a complete service is decomposed into a series of independent microservices, which may be deployed on the same or different edge servers. We use $S = \{S_1, S_2, \ldots, S_I\}$ to denote the set of microservices, where S_i.cpu and S_i.mem are the computational and memory resources required to deploy an instance of S_i, respectively. We define the application requests as $R = \{R_1, R_2, \ldots, R_K\}$. An application request is a sequence of microservices arranged in a specific order and we need to finish all the required microservices to fulfill the application request.

Considering the general scenarios, we assume that the edge servers are deployed near the base stations, as shown in Fig. 1. Application requests initially arrive at the base station, which is then forwarded between edge servers until all microservices are completed.

C. Request Routing

Instances of the same microservice may be deployed on different servers. For example, in Fig. 1, there are instances of S_3 on both E_4 and E_6. Therefore, it is essential to establish a request routing mechanism for service requests to select the most appropriate server to provide service. To balance efficiency and load, we adopt the weighted routing approach. The faster the transmission rate of the target server and the more microservice instances are deployed, the higher the routing probability. For instance, we need to invoke S_j after finishing S_i, and assuming that S_i is deployed on E_m, the probability of invoking S_j on E_n can be expressed as follow:

$$P(S_i, S_j | E_m, E_n) = \omega_1 \frac{V_{m,n}}{\sum_{k\in\Theta} V_{m,k}} + \omega_2 \frac{N_n^j}{\sum_{k\in\Theta} N_k^j} \quad (1)$$

$$\Theta = \{k | S_j \text{ in } E_k\} \quad (2)$$

where two adjustable weighting parameters ω_1 and ω_2 are introduced with $\omega_1 + \omega_2 = 1$ and N_n^j is defined as the number of instances of S_j deployed on E_n.

D. Service Response Latency

The service response latency of each request is determined by two components: 1) transmission latency and 2) processing latency.

Transmission Latency: Assuming a microservice S_i deployed on server E_m invokes a microservice S_j deployed on the server E_n with data size $\text{data}_{i,j}$, then the transmission latency is:

$$T_t = \frac{\text{data}_{i,j}}{V_{m,n}}. \quad (3)$$

Processing Latency: In practical applications, when a request arrives at a server, it is not necessarily executed immediately but instead incurs a queuing delay. Currently, to handle high-concurrency service demands, deployment using multiple instances is commonly adopted. Therefore, we model the queuing delay using M/M/C queuing theory. For instances of S_i deployed in E_m. The number of instances is N_m^i. The average service rate for a single instance is μ_m^i which is a

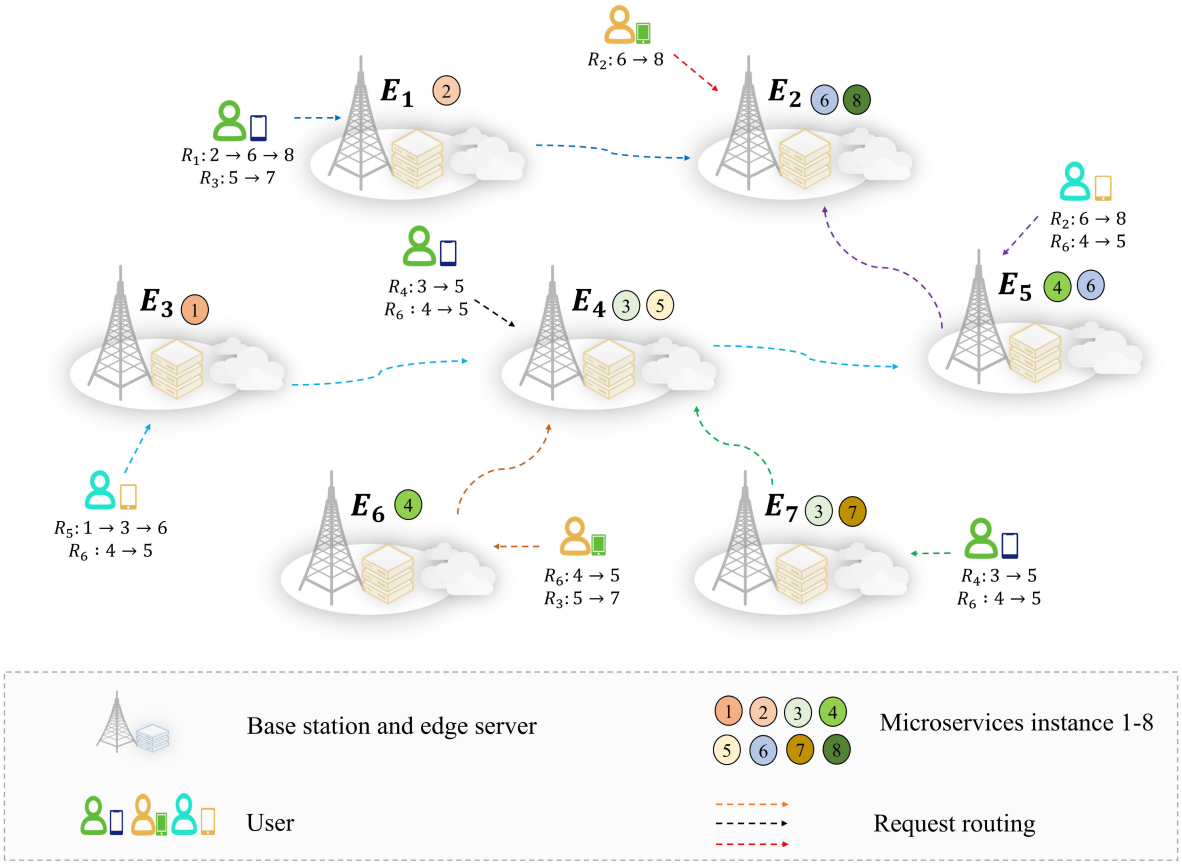


Fig. 1. Microservice architecture in edge environment.

constant. The average arrival rate of microservice S_i in E_m is λ_m^i. Then the processing latency can be represented as follows:

$$T_p = T_q + \frac{1}{\mu_m^i} \tag{4}$$

$$T_q = \frac{\left(N_m^i \rho_m^i\right)^{N_m^i} \rho_m^i}{\lambda_m^i N_m^i! \left(1-\rho_m^i\right)^2} P_0 \tag{5}$$

$$\rho_m^i = \frac{\lambda_m^i}{N_m^i \mu_m^i} \tag{6}$$

$$P_0 = \left(\sum_{h=0}^{N_m^i} \frac{1}{h!} \left(\frac{\lambda_m^i}{\mu_m^i}\right)^h + \frac{1}{N_m^i! \left(1-\rho_m^i\right)} \left(\frac{\lambda_m^i}{\mu_m^i}\right)^{N_m^i} \right)^{-1}. \tag{7}$$

The service response latency of a request is the sum of all transmission and processing latency in this request

$$T_s = \sum T_t + \sum T_p. \tag{8}$$

We denote the average service response latency of R_k in E_m as $T_{k,m}$. The total service response latency can be expressed as

$$T_{\text{delay}} = \sum_{m=1}^{M} \Lambda_m^k T_m^k. \tag{9}$$

E. Energy Consumption

Servers exhibit idle energy consumption and load energy consumption. Idle energy consumption refers to the energy consumed by a server when it is powered on, regardless of whether any services are running on it. Load energy consumption refers to the energy consumed when services are running on the server. We model the server energy consumption as follows:

$$P_m = P_{\text{startup}} + P_{\text{instance}} \sum_{i=1}^{I} N_m^i \varphi\left(U_m^i\right) \tag{10}$$

where P_{startup} represents the startup energy consumption of the server, and P_{instance} denotes the energy consumption incurred by running one microservice instance on the server. U_m^i denotes the utilization ratio of the S_i in E_m, and φ represents a nonlinear function that describes the relationship between utilization ratio and energy consumption, which is dependent on the computational and storage devices employed. The total energy consumption of all servers can be expressed as

$$E_{\text{cost}} = \sum_{m=1}^{M} P_m. \tag{11}$$

F. Autoscaling

Traditional Kubernetes-based microservice autoscaling, as illustrated in the left part of Fig. 2, involves each server monitoring the current request arrival rate on that server and utilizing its local resources to perform microservice autoscaling. However, due to the resource constraints of individual edge servers, this approach often fails to achieve effective

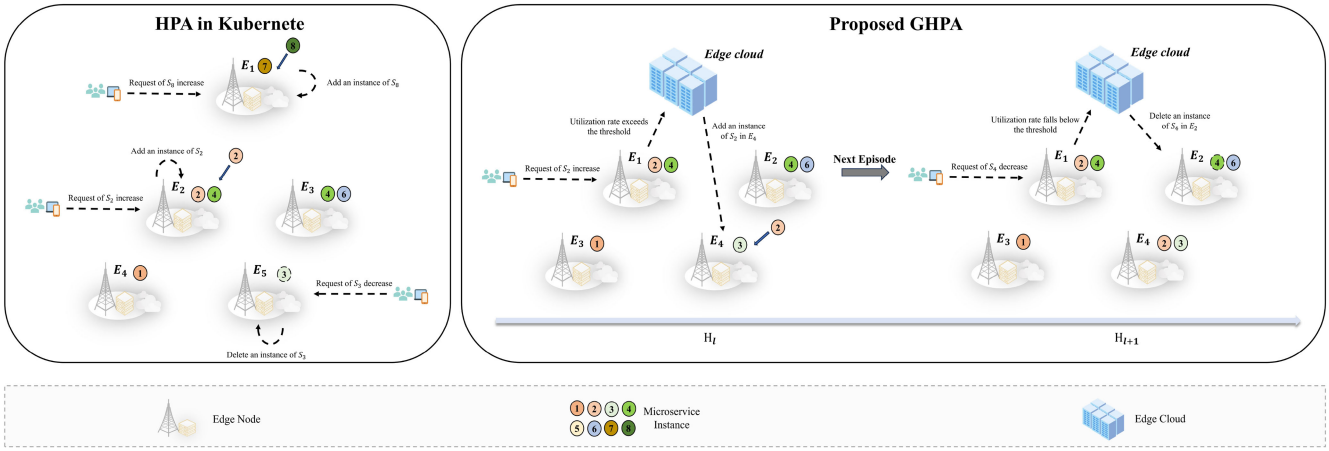


Fig. 2. Autoscaling in edge environment.

microservice autoscaling. Additionally, this method is highly sensitive to the initial deployment [12], where a poor initial deployment can significantly degrade the performance of subsequent microservice autoscaling.

Our proposed GHPA algorithm introduces a central controller on top of the traditional Kubernetes-based microservice autoscaling framework, enabling microservice autoscaling at the level of edge clusters. We adapt a master-slave model, where one edge server in the cluster is selected as the master, referred to as the edge cloud, while the remaining edge servers act as slaves, as shown in the right part of Fig. 2. This master-slave model is widely used in microservice architectures, such as in service registration and discovery (Eureka), as well as fault detection (Prometheus) in edge servers.

When the request arrival rate increases beyond the load capacity of the existing instances, unlike traditional microservice autoscaling, the server will send a request to the edge cloud server. The edge cloud server then executes the microservice autoscaling algorithm, deploys additional microservice instances on suitable target edge servers, and routes part of the requests to the target server based on the routing rules. Conversely, when the request arrival rate decreases and the instance load falls below the minimum threshold, the edge server similarly sends a request to the edge cloud server. The edge cloud server executes the microservice autoscaling algorithm, removes excess microservice instances from the target edge server, and routes the requests from the target server to other edge servers based on the routing rules.

We model the autoscaling problem as follows. The request arrival rate exhibits variability both temporally and spatially, which can be represented by the function $\lambda(s,t)$, where s denotes location and t denotes time. In our model, we discretize both time and space. A day is divided into several timeslots $G = \{G_1, G_2, \ldots, G_L\}$, while the spatial division is based on the coverage areas of edge servers. Therefore, we use $\Lambda_m^k(l)$ to represent the arrival rate of R_k from E_m during timeslot G_l. In different timeslots, to handle the variations in request arrival rates, we need to adjust the instance deployment on the edge server. We use $N_m^i(l)$ to denote the number of instances of S_i deployed on E_m during timeslot G_l.

G. Long-Term Optimization Objective

This article proposes a multiobjective long-term optimization problem that considers both service latency and energy consumption

$$\min_{N_m^i(l)} \lim_{L\to\infty} \frac{1}{L} \sum_{l=1}^{L} \left(\alpha T_{\text{latency}}(l) + \beta E_{\text{cost}}(l)\right) \tag{12}$$

where α, β represent the weights of service latency, energy cost, respectively. The summation and averaging operations are employed to achieve the objective of long-term optimization. In general, reducing the latency tends to increase energy consumption, and vice versa. In real-world applications, we need to balance the weights to achieve the best performance.

There are two constraints: resource constraints and load constraints. The resource constraints ensure that the computational and memory resources utilized by the server do not exceed the total computational and memory resources available on that server. This can be expressed as follows:

$$\sum_{i=1}^{I} N_m^i(l) S_i.\text{cpu} < E_m.\text{cpu} \quad m = 0, 1, 2, \ldots, M \tag{13}$$

$$\sum_{i=1}^{I} N_n^i(l) S_i.\text{mem} < E_m.\text{mem} \quad m = 0, 1, 2, \ldots, M. \tag{14}$$

The load constraints primarily ensure the normal operation of the server, preventing system crashes. We denote the load by the ratio of the total arrival rate to the total service rate. The load constraints can be expressed as follows:

$$\rho_m^i(l) = \frac{\lambda_{i,m}(l)}{N_m^i(l)\mu_{i,m}(l)} \tag{15}$$

$$\rho_m^i(l) < \rho_{\max} \tag{16}$$

where $\rho_{\max}$ represents the maximum load constraint.

Thus, the overall optimization objective is defined as

$$\min_{N_m^i(l)} \lim_{L\to\infty} \frac{1}{L} \sum_{l=1}^{L} \left(\alpha T_{\text{latency}}(l) + \beta E_{\text{cost}}(l)\right) \tag{17}$$

$$\text{s.t.} \quad (13)(14)(16). \tag{18}$$

IV. Algorithm

Traditional methods struggle to address long-term optimization problems. RL has been extensively applied to solve dynamic problems and has demonstrated promising results. Therefore, we adopt RL to tackle the aforementioned long-term optimization challenge.

A. Preliminary

RL: RL is a subfield of machine learning that focuses on how an agent can learn to make sequential decisions by interacting with an environment to maximize a cumulative reward signal. Unlike supervised learning, which relies on labeled datasets, RL operates through trial and error, where the agent explores the environment, takes actions, and receives feedback in the form of rewards. The core components of RL include the agent, the environment, states, actions, rewards, policies, value functions, and Q-value functions. The agent's goal is to learn a policy, denoted as π, which maps states to actions in a way that maximizes the expected cumulative reward. The state-value function $V^{\pi}(s)$ represents the expected return when starting from state s and following policy π, defined as

$$V^{\pi}(s) = \mathbb{E}_{\pi}[G_t | S_t = s]$$

where G_l is the discounted sum of future rewards

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

and γ is the discount factor ($0 \leq \gamma \leq 1$). Similarly, the action-value function $Q^{\pi}(s, a)$ represents the expected return when taking action a in state s and following policy π, defined as

$$Q^{\pi}(s, a) = \mathbb{E}_{\pi}[G_t | S_t = s, A_t = a].$$

The Bellman equation provides a recursive relationship for the value functions. For the state-value function, it is expressed as

$$V^{\pi}(s) = \sum \pi(a|s) \sum P\big(s'|s, a\big)\big[R\big(s, a, s'\big) + \gamma V^{\pi}\big(s'\big)\big].$$

The optimal value functions $V^*(s)$ and $Q^*(s, a)$ are defined as the maximum value functions over all policies

$$V^*(s) = \max_{\pi} V^{\pi}(s)$$
$$Q^*(s, a) = \max_{\pi} Q^{\pi}(s, a).$$

The optimal policy π^* is the policy that achieves these optimal value functions, defined as

$$\pi^*(a|s) = \arg\max_{a} Q^*(s, a).$$

RL provides a robust framework for decision-making in dynamic environments, enabling agents to learn effective policies through iterative interactions and optimization of cumulative rewards. The foundational concepts and formulas outlined above serve as the basis for understanding and applying RL in various domains, ranging from robotics and game playing to resource management and autonomous systems.

Algorithm 1: MFFD

Input: Request arrival rate Λ; Resource limitation $E_m.cpu$ and $E_m.mem$ $m \in 1, 2, ..., M$; Distance between edge servers $d_{m,n}$ $m, n \in 0, 1, 2, ..., M$.
Output: Initial microservice deployment $N_m^i(0)$

```
Arrange all requests in a certain order and denote them as REQ
Create a new stack and initialize it stack ← ∅
for req in REQ do
    REQ.delete(req)
    Get the start server of this request E_s
    Get the arrival rate of this request λ_s
    Get the next undeployed microservice m_s
    stack.put([E_s,λ_s,ms_s])
    while stack is not empty do
        E,λ,m=stack.pop()
        Calculate the required number of instances K for m according to λ
        Get the next next undeployed microservice m_n
        Sorting all servers in ascending order based on their distance to E and denote them as Ẽ
        for E_n in Ẽ do
            Calculate the number of deployable instances K_E in E_n
            Deploy K_E instances of m in E_n
            Update the microservice deployment N_n^i
            Update resource utilization E_n.cpu and E_n.mem
            Route a portion of request λ_n to E_n based on the instance deployment
            if m_n exists then
                stack.put(E_n,λ_n,m_n)
            if Complete the deployment of K instances required for m then
                Break
```

B. MFFD Algorithm

A day is divided into multiple timeslots. In the initial timeslot G_0, a deployment is required as the starting solution for the GHPA algorithm. For the initial solution, this article introduces the microservice first-fit decreasing (MFFD) algorithm, inspired by the first-fit decreasing (FFD) heuristic used in service deployment [35] and [36]. The core idea is to employ a greedy strategy to assign interconnected microservices to the nearest edge servers, thereby minimizing transmission latency. The pseudocode is presented in Algorithm 1.

Step 1: Arrange the requests in a certain order, which is arbitrary. For example, the order could be based on the servers, starting with all requests on E_1, followed by all requests on E_2, and so on. We use E_nR_k to represent the request R_k in server E_n. So the order could be E_1R_1, E_1R_2, ..., E_1R_K, E_2R_1, ..., E_NR_K.

Step 2: Consider the deployment of microservice instances. For each request, we employ the greedy strategy to deploy the interconnected microservice instances on the nearest server. For example, for a pair of microservices S_i and S_j in an application request. If S_i is deployed on E_m, then we could deploy S_j on the available server closest to E_m to save transmission latency. The specific operation is that we first calculate the number of instances required based on the request arrival rate, and then identify the suitable deployment server. Each request is composed of several pairs of interconnected microservices and for each pair, we apply the same operation. It is noteworthy that there may be situations where no single available server has sufficient resources to deploy all the required microservice instances. In such cases, we deploy a subset of the instances on the nearest server and subsequently deploy the remaining instances on the second nearest server, and so forth.

C. GHPA Algorithm

RL is grounded in the framework of MDP, where each action taken by the agent elicits a reward from the environment. The objective of RL is to maximize cumulative rewards through a series of sequential decisions. Consequently, we can transform the long-term optimization problem into an MDP. As illustrated in Fig. 2, the process of dynamic microservice adjustment is converted into a sequence of decisions. For each decision, the agent deployed on the edge cloud first collects information from all edge servers to form the state. Subsequently, the agent selects an appropriate target server to either increase or decrease microservices. Finally, the overall latency and energy consumption across the edge environment are qualified as the reward for the agent's decision. The notation H_l $l = 1, 2, 3, \ldots,$ is employed to denote each decision made by the agent, specifically referring to each adjustment of the microservice instance. Currently, RL encompasses a variety of algorithms to address MDPs, with the most commonly used ones, including A3C, PPO, and DDPG. Different algorithms are suited to different types of problems. A3C, a policy gradient-based algorithm, although less precise in solution accuracy compared to PPO, significantly outperforms both PPO and DDPG in terms of computational speed. Given the critical importance of computational speed in edge computing, we have chosen A3C as our RL algorithm. The pseudocode is shown in Algorithm 2 and the specifics are as follows.

Framework: The framework of GHPA is illustrated in Fig. 3. Master server is responsible for data monitoring and decision-making, while slave servers are tasked with data collection and the scaling of microservice instances.

State: The state space is composed of five components: the request arrival rates for all edge servers in the past p timeslots $\vec{\Lambda}$, the resource status(computational and memory resources) of all edge servers RS, the microservice instance deployment of all edge servers N, the identifier of the microservice to be adjusted MS, and an indicator I specifying whether to increase or decrease the microservice. Here, $I = 1$ denotes the need to add an MS microservice instance, while $I = 0$ signifies the need to remove an MS microservice instance

$$s = \{\vec{\Lambda}, \mathrm{RS}, N, \mathrm{MS}, I\}. \tag{19}$$

Action: The action space is the identifiers of the edge servers, indicating the addition or removal of a microservice instance on server E

$$a = E. \tag{20}$$

Reward: The reward function consists of two components. The first is the deployment success reward, which provides a fixed reward c upon successful deployment. The second is the optimization reward for latency and energy consumption,

Algorithm 2: GHPA

Input: Epoch number M; Soft update factor μ; Reward discount factor γ; Request arrival rate Λ; Batch Size; Initial deployment $N_m^i(0)$

Initialize Actor network $\sigma(s|\theta^\sigma)$ and target actor network σ^*
Initialize critic network $Q(s, a|\theta^Q)$ and target critic network Q^*
Let $\theta^{\sigma^*} \leftarrow \theta^\sigma$, $\theta^{Q^*} \leftarrow \theta^Q$
for *epoch* $i = 0, ..., M$ **do**
  Initialize a random process $\mathcal{N}$ to add exploration to the action
  $j \leftarrow 0$
  Accept initial state s_0 according Eq (21)
  while *not done* **do**
    Select action $a_j = \sigma(s_j|\theta^\sigma) + \mathcal{N}_j$
    Execute the action a_j
    if *Action failture* **then**
      Break
    else
      $r = c + \alpha T_{latency} + \beta E_{cost}$
    Update instance deployment N, Resource usage RS
    Update request arrival rate Λ
    Transfer to another state s_{j+1}
    Store the transition $[s_j, a_j, r_j, s_{j+1}]$
    $j \leftarrow j + 1$
    if j *mod BatchSize* $==$ *0* **then**
      Calculate the value of Q, and make $q_j = r_j + \gamma Q^*(s_{j+1}, \sigma^*(s_{j+1}|\theta^{\sigma^*})|\theta^{Q^*})$
      Update the critic network by minimizing the loss function $\mathcal{L} = \frac{1}{N}(\sum_t (q_j - Q(s_j, a_j|\theta^Q)))^2$
      Update the actor network by maximizing the value function through gradient ascent: $\nabla_{\theta^\sigma} \mathcal{J} \approx \frac{1}{N} \sum_t \nabla_a Q(s, a|\theta^Q)|_{s=s_j, a=\sigma(s_j|\theta^\sigma)} \nabla_{\theta^\sigma} \sigma(s|\theta^\sigma)|_{s_j}$
      Update the target network using exponential smoothing: $\theta^{Q^*} = \mu\theta^Q + (1-\mu)\theta^{Q^*}$, $\theta^{\sigma^*} = \mu\theta^\sigma + (1-\mu)\theta^{\sigma^*}$
      Clear the transition list
      $j \leftarrow 0$

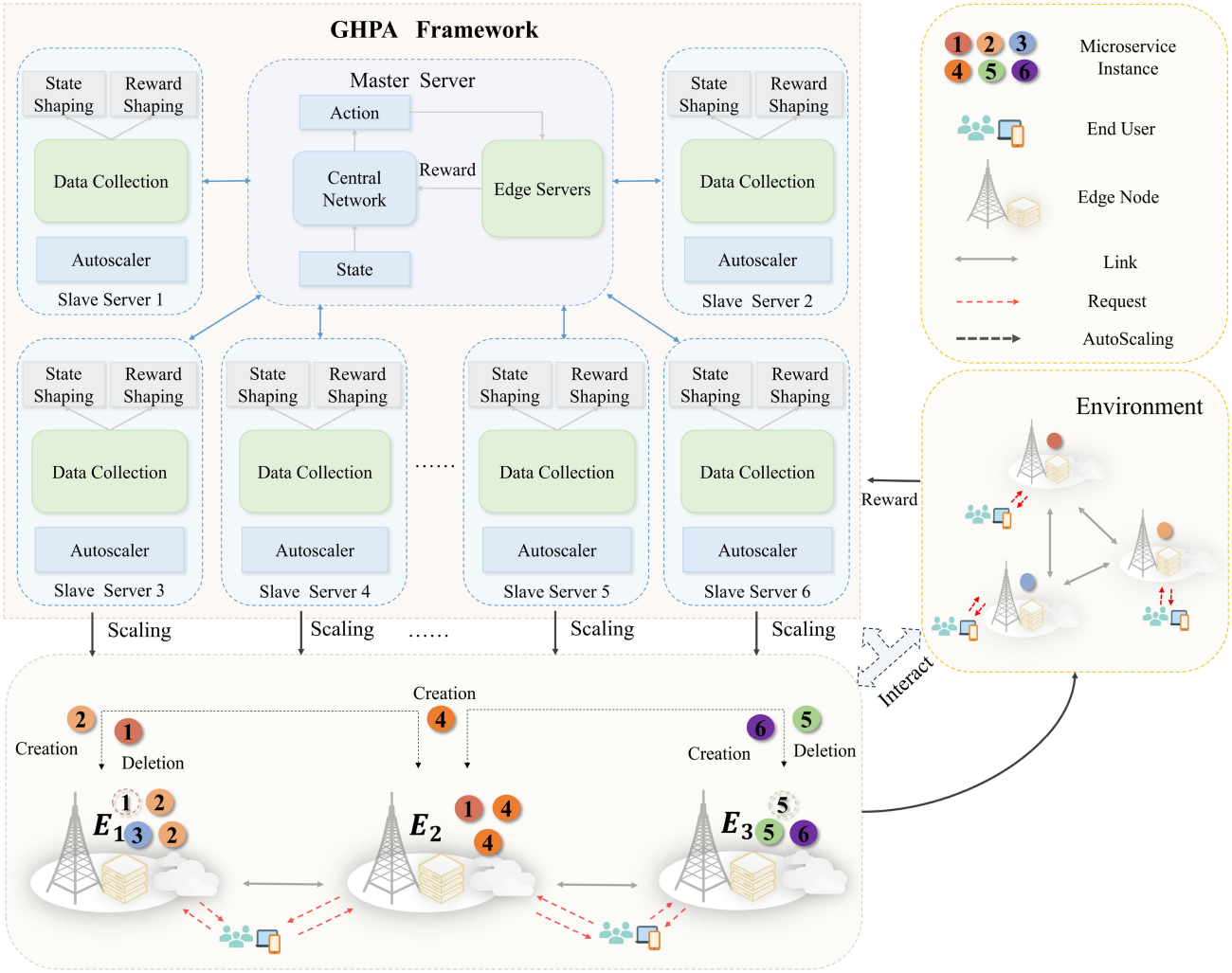


Fig. 3. Framework of GHPA.

which is a weighted reward based on these two factors. Since our objective is to minimize both latency and energy consumption, this component of the reward is negative. The design of the reward function enables the agent to continuously and dynamically adjust the deployment of microservice instances while simultaneously minimizing latency and energy consumption

$$r = c + \alpha T_{\text{latency}} + \beta E_{\text{cost}}. \tag{21}$$

V. Performance Evaluation

In this section, the proposed GHPA is evaluated from multiple perspectives.

A. Experimental Setting

We use both real and synthetic data to perform comprehensive simulations of our algorithm. The hyperparameter settings of Training are summarized in Table II.

Edge Servers and Devices: To simulate the realistic MEC environments, we utilized the geographic coordinates of user devices and edge servers from the Shanghai Telecom dataset [37]. This dataset is open to the public and contains more than 7.2 million Internet access records through 3233 base stations from 9481 mobile phones for six months. The dataset includes the geographical location information and user access information of each base station. We selected base stations with long and uninterrupted records in the dataset as our target base station for simulation. Also, the user traffic during different time period was used as the request arrival rate in the simulation.

Application Requests and Microservices: We choose cluster-trace-v2018 dataset from Alibaba [3] and cache-trace dataset from Twitter [38]. Cluster-trace-v2018 dataset includes information about 4000 machines in a period of 8 days and consists 6 tables. In *machine_meta.csv*, it contains the resource information of each machine. In *batch_task.csv*, it describes the required CPU and memory for each instance and provides a detailed description of the microservice invocation relationship for each service request. Cache-trace dataset comprises user request data collected from 54 servers. User information is documented in the *client_id* file, while the arrival time of each request is recorded in the *timestamp* file.

B. Baselines Algorithms

We select several state-of-the-art algorithms that are most relevant to our study as baseline algorithms for comparison.

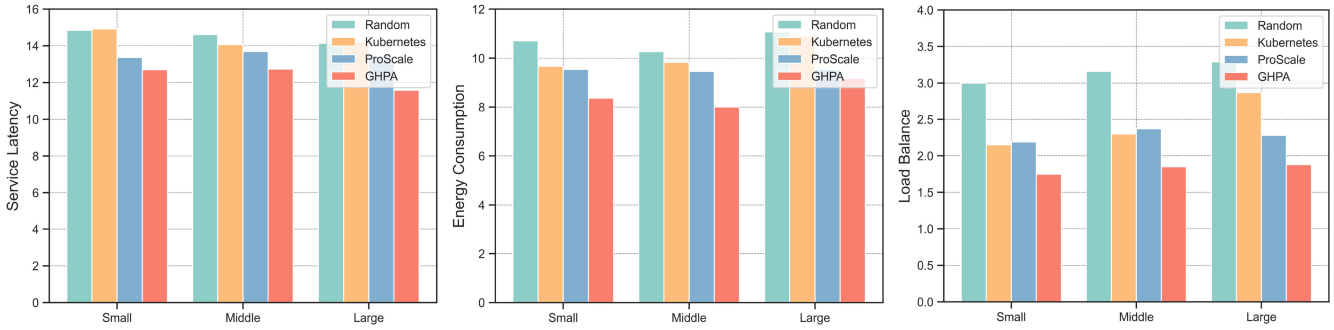


Fig. 4. Service latency, energy consumption, and load balance of four algorithms under different scales of edge environment.

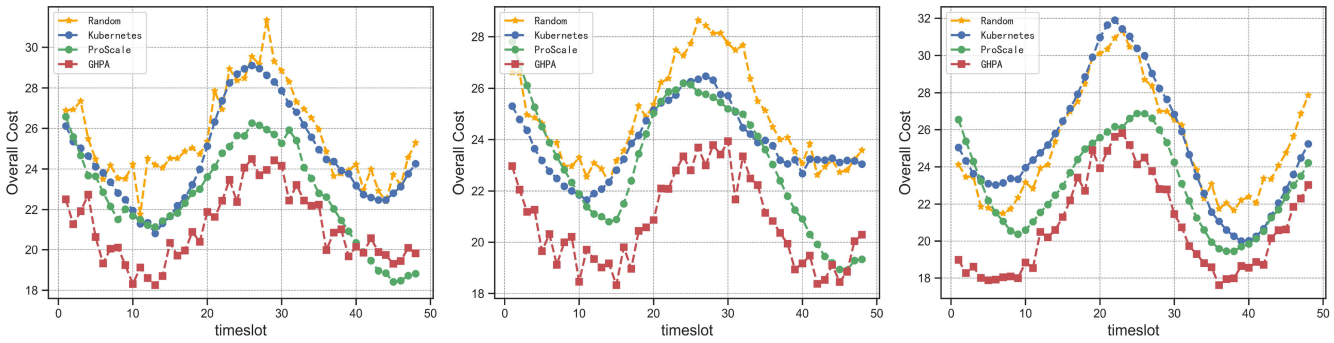


Fig. 5. Overall cost of the four algorithms in each timeslot under different scales of edge environment.

Random: Microservice instances are randomly deployed in each timeslot while meeting the constraints. This approach can only meet the basic requirements and does not incorporate any optimization.

Kubernetes: Kubernetes is deployed on each server to dynamically adjust the number of microservice instances based on real-time load. The monitoring metrics selected are CPU and memory utilization. While Kubernetes performs well on cloud servers in handling request fluctuations, its effectiveness on edge servers is limited due to resource constraints.

ProScale [11]: In ProScale, the autoscaler employs a greedy strategy, iterating through all servers during each timeslot and selecting the server that minimizes latency for deploying microservice instances. However, long-term optimization is highly complex, and a purely greedy strategy often fails to yield a good solution.

C. Metrics

We carefully select four metrics to comprehensively assess the performance of our algorithm.

Service Latency: Service latency is directly related to the user experience, and we aim for the shortest possible service latency.

Energy Consumption: Energy consumption is linked to costs, and we aim to reduce energy consumption.

Load Balance: We use the variance of instance utilization to measure the load balancing status among instances. A smaller variance indicates a smaller gap in utilization across instances, suggesting a more balanced load distribution.

TABLE II
HYPERPARAMETER SETTINGS OF TRAINING

Parameter	Value	Description
lr_actor	0.005	the learning rate of Actor network
lr_critic	0.005	the learning rate of Critic network
μ	0.01	soft update factor
γ	0.95	reward discount factor
REG	0.01	entropy coefficient
$optimizer$	Adam	the optimizer for training the network
$batch\ size$	32	the number of data samples
$hidden\ layer$	3	the hidden layer of neural network
p	8	input sequence length for prediction

Overall Cost: Cost is defined as the weighted average of service latency and energy consumption. Since these two metrics are minimization objectives, a lower cost indicates better performance.

D. Experimental Results

The Impact of Edge Scales: We established three scales of edge environment: 1) small scale (6 types of microservices, 4 types of service requests, and 6 edge servers); 2) middle scale (8 types of microservices, 6 types of service requests, and 10 edge servers); and 3) large scale (12 types of microservices, 8 types of service requests, and 14 edge servers). The results are illustrated in Fig. 4. In this figure, the horizontal axis represents the edge scales, while the vertical axis represents the service latency, energy consumption, and

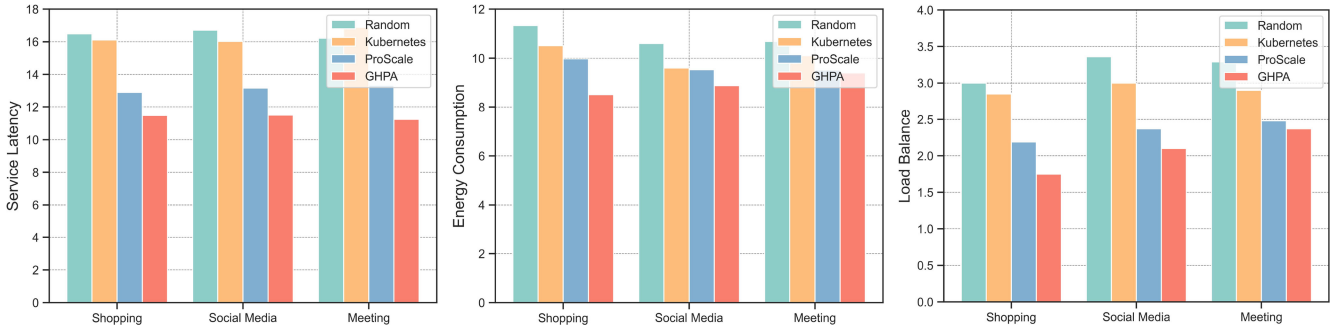


Fig. 6. Service latency, energy consumption, and load balance of four algorithms under different application requests.

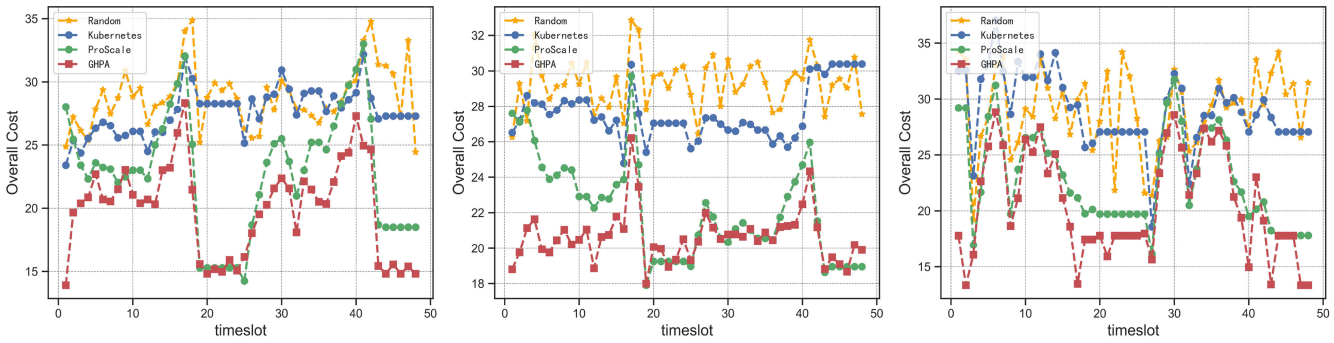
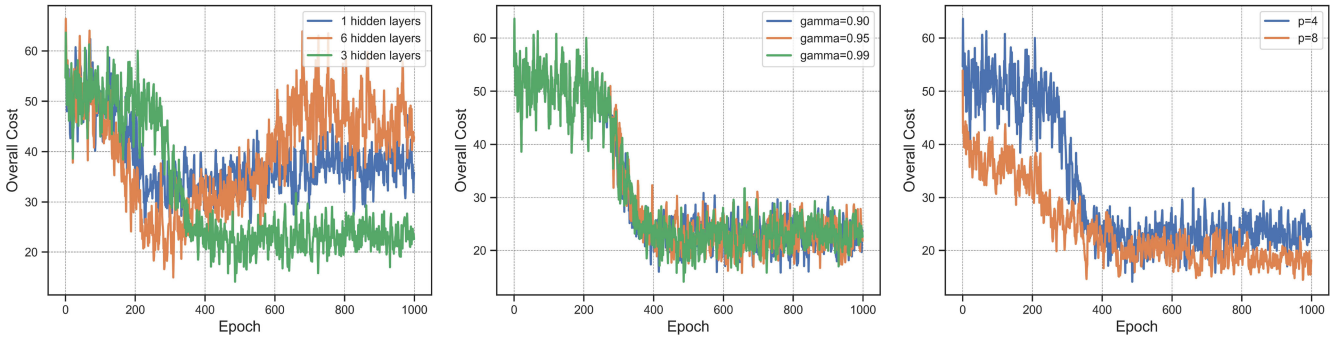


Fig. 7. Overall cost of the four algorithms in each timeslot under different application requests.

load balance. As illustrated in the figure, in terms of load balancing, Kubernetes, ProScale, and GHPA exhibit excellent performance at small scales. However, as the scale increases, the load balancing effectiveness of Kubernetes declines sharply due to its inability to handle resource constraints on individual servers, while ProScale and GHPA maintain robust load balance performance. Regarding energy consumption, there is a notable correlation with load balance. A more balanced load enables efficient utilization of resources, resulting in relatively lower energy consumption. In terms of latency, GHPA demonstrates significant improvement over Kubernetes across all scales and achieves approximately a 10% reduction in latency compared to the ProScale, which employs a greedy strategy. In Fig. 5, the horizontal axis represents the time period, while the vertical axis denotes the overall cost. We illustrate the variations in overall cost across different timeslots. As observed in the figure, as the scale gradually increases, the total cost of Kubernetes progressively approaches that of the random strategy. Although GHPA occasionally underperforms compared to Proscale in certain timeslots, it consistently demonstrates superior overall performance among all the compared algorithms.

The Impact of Different Types of Requests: We analyzed the dataset and extracted the variations in requests for different types of applications (shopping, social, and meeting) throughout the day. The edge environment was fixed at a large scale, and the performance of our algorithm was tested using distinct applications. The results are illustrated in Fig. 6. As observed, at the large scale, Kubernetes demonstrates performance comparable to a random strategy, showing no significant advantage. ProScale incorporates optimizations for latency, significantly reducing service latency. GHPA leverages a RL algorithm, further enhancing performance (by approximately 13%) compared to the greedy strategy employed by ProScale. Fig. 7 provides a clearer depiction of the performance of each algorithm across different timeslots. Both the random and Kubernetes struggle to adapt effectively to request fluctuations, whereas ProScale and GHPA can dynamically adjust in response to these variations. Additionally, although ProScale occasionally outperforms GHPA in some timeslots, GHPA focuses on long-term optimization, resulting in superior overall performance compared to ProScale.

The Impact of Hyperparameters: The configuration of hyperparameters has a substantial impact on the performance of RL algorithms. We examined the effect of various hyperparameters on the GHPA. Initially, we assessed the influence of different hidden layer configurations on the algorithm, and the results are illustrated in the Fig. 8(a). The choice of hidden layers should be determined by the specific environment. When there are significant fluctuations in service requests, it is advisable to increase the number of hidden layers. Conversely, the number of hidden layers should be reduced to prevent overfitting. From Fig. 8(a) we can see that GHPA demonstrates the best convergence when the number of hidden layers is set to three. Second, we tested the impact of different discounted factors on the convergence of the algorithm, as shown in Fig. 8(b). A larger discounted factor leads the algorithm to prioritize long-term optimization. Commonly used values are 0.9, 0.95, and 0.99. In our tests, the performance with these three values was similar. Finally, we examine the impact



(a) (b) (c)

Fig. 8. Impact of different hyperparameters. (a) Impact of the number of hidden layers. (b) Impact of discounted factor. (c) Impact of input sequence length.

of input sequence length on algorithm performance. We use p to denote the input sequence length. The comparison result of $p = 4$ and $p = 8$ is illustrated in Fig. 8(c). Longer input sequences contribute to more accurate prediction results but simultaneously incur higher data collection overhead. As illustrated in the figure, the model achieves faster convergence when $p = 8$.

VI. Conclusion

In this article, we have proposed a cluster-based autoscaler GHPA to address resource limitations and resource heterogeneity. Then, we have considered the joint optimization of instance adjustment and request routing to achieve better autoscaling performance. Next, we have conducted a comprehensive analysis of microservice intercommunication in edge computing, employing undirected graphs and open Jackson queueing networks to formulate a long-term optimization problem. Finally, we have transformed the long-term optimization problem into a MDP and used RL techniques to solve it. Through extensive experiments, our proposed approach can achieve at least 10% performance improvement under various test conditions compared to three state-of-the-art alternatives.

References

[1] A. Sill, "The design and architecture of microservices," *IEEE Cloud Comput.*, vol. 3, no. 5, pp. 76–80, Sep./Oct. 2016.

[2] M. Dapr. "APIs for building secure and reliable microservices." 2025. [Online]. Available: https://dapr.io/

[3] "cluster-trace-v2018." 2018. [Online]. Available: https://\penalty\z@{}github.\penalty\z@{}com/\penalty\z@{}alibaba/\penalty\z@{}clusterdata/\penalty\z@{}blob/\penalty\z@{}v2018/\penalty\z@{}cluster-\penalty\z@{}trace-\penalty\z@{}v2018/\penalty\z@{}trace_2018.\penalty\z@{}md

[4] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, "A survey on mobile edge computing: The communication perspective," *IEEE Commun. Surveys Tuts.*, vol. 19, no. 4, pp. 2322–2358, 4th Quart., 2017.

[5] K. Peng et al., "Delay-aware optimization of fine-grained microservice deployment and routing in edge via reinforcement learning," *IEEE Trans. Netw. Sci. Eng.*, vol. 11, no. 6, pp. 6024–6037, Nov./Dec. 2024.

[6] Z. Peng et al., "Microservice auto-scaling algorithm based on workload prediction in cloud-edge collaboration environment," in *Proc. IEEE Int. Conf. Internet Things (iThings) IEEE Green Comput. Commun. (GreenCom) IEEE Cyber, Phys. Soc. Comput. (CPSCom) IEEE Smart Data (SmartData) IEEE Congr. Cybermat. (Cybermatics)*, 2023, pp. 608–615.

[7] L. H. Phuc, L.-A. Phan, and T. Kim, "Traffic-aware horizontal pod autoscaler in Kubernetes-based edge computing infrastructure," *IEEE Access*, vol. 10, pp. 18966–18977, 2022.

[8] L. H. Phuc, M. Kundroo, D.-H. Park, S. Kim, and T. Kim, "Node-based horizontal pod autoscaler in KubeEdge-based edge computing infrastructure," *IEEE Access*, vol. 10, pp. 134417–134426, 2022.

[9] L. Ju, P. Singh, and S. Toor, "Proactive autoscaling for edge computing systems with kubernetes," in *Proc. 14th IEEE/ACM Int. Conf. Utility Cloud Comput. Compan.*, 2021, pp. 1–8.

[10] G. Tong, C. Meng, S. Song, M. Pan, and Y. Yu, "GMA: Graph multi-agent microservice autoscaling algorithm in edge-cloud environment," in *Proc. IEEE Int. Conf. Web Services (ICWS)*, 2023, pp. 393–404.

[11] K. Cheng et al., "Proscale: Proactive autoscaling for microservice with time-varying workload at the edge," *IEEE Trans. Parallel Distrib. Syst.*, vol. 34, no. 4, pp. 1294–1312, Apr. 2023.

[12] Y. Li, H. Zhang, W. Tian, and H. Ma, "Joint optimization of auto-scaling and adaptive service placement in edge computing," in *Proc. IEEE 27th Int. Conf. Parallel Distrib. Syst. (ICPADS)*, 2021, pp. 923–930.

[13] L. Chen et al., "IoT microservice deployment in edge-cloud hybrid environment using reinforcement learning," *IEEE Internet Things J.*, vol. 8, no. 16, pp. 12610–12622, Aug. 2021.

[14] K. Cheng et al., "GeoScale: Microservice autoscaling with cost budget in geo-distributed edge clouds," *IEEE Trans. Parallel Distrib. Syst.*, vol. 35, no. 4, pp. 646–662, Apr. 2024.

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

[16] V. Singh and S. K. Peddoju, "Container-based microservice architecture for cloud applications," in *Proc. Int. Conf. Comput., Commun. Autom. (ICCCA)*, 2017, pp. 847–852.

[17] X. Wan, X. Guan, T. Wang, G. Bai, and B.-Y. Choi, "Application deployment using microservice and docker containers: Framework and optimization," *J. Netw. Comput. Appl.*, vol. 119, pp. 97–109, Oct. 2018.

[18] B. Xu, J. Guo, F. Ma, M. Hu, W. Liu, and K. Peng, "On the joint design of microservice deployment and routing in cloud data Centers," *J. Grid Comput.*, vol. 22, no. 2, p. 42, 2024.

[19] Y. Hu, H. Wang, L. Wang, M. Hu, K. Peng, and B. Veeravalli, "Joint deployment and request routing for microservice call graphs in data centers," *IEEE Trans. Parallel Distrib. Syst.*, vol. 34, no. 11, pp. 2994–3011, Nov. 2023.

[20] S. Deng et al., "Optimal application deployment in resource constrained distributed edges," *IEEE Trans. mobile Comput.*, vol. 20, no. 5, pp. 1907–1923, May 2021.

[21] F. Guo, B. Tang, and M. Tang, "Joint optimization of delay and cost for microservice composition in mobile edge computing," *World Wide Web*, vol. 25, no. 5, pp. 2019–2047, 2022.

[22] M. Hu et al., "Collaborative deployment and routing of industrial microservices in smart factories," *IEEE Trans. Ind. Informat.*, vol. 20, no. 11, pp. 12758–12770, Nov. 2024.

[23] K. Peng, L. Wang, J. He, C. Cai, and M. Hu, "Joint optimization of service deployment and request routing for microservices in mobile edge computing," *IEEE Trans. Services Comput.*, vol. 17, no. 3, pp. 1016–1028, May/Jun. 2024.

[24] W. Lv et al., "Graph-reinforcement-learning-based dependency-aware microservice deployment in edge computing," *IEEE Internet Things J.*, vol. 11, no. 1, pp. 1604–1615, Jan. 2024.

[25] M. Hu et al., "Joint optimization of microservice deployment and routing in edge via multi-objective deep reinforcement learning," *IEEE Trans. Netw. Service Manag.*, vol. 21, no. 6, pp. 6364–6381, Dec. 2024.
[26] Z. Ding and Q. Huang, "COPA: A combined autoscaling method for kubernetes," in *Proc. IEEE Int. Conf. Web Services (ICWS)*, 2021, pp. 416–425.
[27] M. R. Hossen, M. A. Islam, and K. Ahmed, "Practical efficient microservice autoscaling with QoS assurance," in *Proc. 31st Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2022, pp. 240–252.
[28] F. Zhang, X. Tang, X. Li, S. U. Khan, and Z. Li, "Quantifying cloud elasticity with container-based autoscaling," *Future Gener. Comput. Syst.*, vol. 98, pp. 672–681, Sep. 2019.
[29] A. Kwan, J. Wong, H.-A. Jacobsen, and V. Muthusamy, "Hyscale: Hybrid and network scaling of dockerized microservices in cloud data centres," in *Proc. IEEE 39th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, 2019, pp. 80–90.
[30] M. Abdullah, W. Iqbal, A. Mahmood, F. Bukhari, and A. Erradi, "Predictive autoscaling of microservices hosted in fog microdata center," *IEEE Syst. J.*, vol. 15, no. 1, pp. 1275–1286, Mar. 2021.
[31] W. Iqbal, A. Erradi, M. Abdullah, and A. Mahmood, "Predictive auto-scaling of multi-tier applications using performance varying cloud resources," *IEEE Trans. Cloud Comput.*, vol. 10, no. 1, pp. 595–607, Jan.–Mar. 2019.
[32] I. Prachitmutita, W. Aittinonmongkol, N. Pojjanasuksakul, M. Supattatham, and P. Padungweang, "Auto-scaling microservices on IaaS under SLA with cost-effective framework," in *Proc. 10th Int. Conf. Adv. Comput. Intell. (ICACI)*, 2018, pp. 583–588.
[33] M. Xu et al., "Coscal: Multifaceted scaling of microservices with reinforcement learning," *IEEE Trans. Netw. Service Manag.*, vol. 19, no. 4, pp. 3995–4009, Dec. 2022.
[34] Z. Wang et al., "DeepScaling: Microservices autoscaling for stable CPU utilization in large scale cloud systems," in *Proc. 13th Symp. Cloud Comput.*, 2022, pp. 16–30.
[35] G. Dósa, "The tight bound of first fit decreasing bin-packing algorithm is FFD(i)≤11/9OPT(i)+ 6/9," in *Proc. Int. Symp. Combinator., Algorithms, Probab. Exp. Methodol.*, 2007, pp. 1–11.
[36] A. Alahmadi, A. Alnowiser, M. M. Zhu, D. Che, and P. Ghodous, "Enhanced first-fit decreasing algorithm for energy-aware job scheduling in cloud," in *Proc. Int. Conf. Comput. Sci. Comput. Intell.*, 2014, pp. 69–74.
[37] Y. Guo, S. Wang, A. Zhou, J. Xu, J. Yuan, and C.-H. Hsu, "User allocation-aware edge cloud placement in mobile edge computing," *Softw., Pract. Exp.*, vol. 50, no. 5, pp. 489–502, 2020.
[38] J. Yang, Y. Yue, and K. Rashmi, "A large-scale analysis of hundreds of in-memory key-value cache clusters at twitter," *ACM Trans. Storage*, vol. 17, no. 3, pp. 1–35, 2021.

Kai Peng received the B.E., M.S., and Ph.D. degrees from Huazhong University of Science and Technology, Wuhan, China, in 1999, 2002, and 2006, respectively.

He is currently a Professor with the School of Electronic Information and Communications, Huazhong University of Science and Technology. His research interests include cloud computing and computer networks.

Jie Rao is currently pursuing the M.E. degree with the School of Electronic Information and Communications, Huazhong University of Science and Technology, Wuhan, China.

His research interests include the Internet of Things, cloud computing, and mobile computing.

Hao Li received the B.E. degree from Wuhan University, Wuhan, China, in 2023. He is currently pursuing the master's degree with the School of Electronic Information and Communication, Huazhong University of Science and Technology, Wuhan.

His research interests include reinforcement learning, Internet of Vehicles, microservice deployment, and migration.

Yi Hu (Graduate Student Member, IEEE) is currently pursuing the Ph.D. degree with the School of Electronic Information and Communications, Huazhong University of Science and Technology, Wuhan, China.

His research interests includes cloud computing and distributed systems.

Bo Jin received the B.E. degree in computer science and applications from the Hohai University, Nanjing, China, in 1994.

He is currently the Director of the Information Center, State Grid Hubei Information and Communication Company, Wuhan, China. His research interests include cloud computing distributed systems, and mobile computing.

Tianyue Zheng (Member, IEEE) received the B.Eng. degree from Harbin Institute of Technology, Harbin, China, in 2017, the M.Eng. degree from the University of Toronto, Toronto, ON, Canada, in 2019, and the Ph.D. degree from Nanyang Technological University, Singapore, 2023.

He is currently an Assistant Professor with the Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, China. His research interests include mobile and pervasive computing, the Internet of Things, and machine learning. More information can be found at https://tianyuez.github.io.

Menglan Hu received the B.E. degree in electronic and information engineering from Huazhong University of Science and Technology, Wuhan, China, in 2007, and the Ph.D. degree in electrical and computer engineering from the National University of Singapore, Singapore, in 2012.

He is currently an Associate Professor with the School of Electronic Information and Communications, Huazhong University of Science and Technology. His research interests include cloud computing, computer networks, distributed systems, and mobile computing.