Online Microservice Orchestration for IoT via Multiobjective Deep Reinforcement Learning

Yinbo Yu, *Member, IEEE*, Jiajia Liu, *Senior Member, IEEE*, and Jing Fang

***Abstract*—By providing loosely coupled, lightweight, and independent services, the microservice architecture is promising for large-scale and complex service provision requirements in the Internet of Things (IoT). However, it requires more fine-grained resource management and orchestration for service provision. Most of the existing microservice orchestration solutions are based on those designed for the traditional cloud. They can only provide coarse-grained resource allocation using possibly conflicting weighted objectives. In this article, we present a fine-grained microservice orchestration approach to provide services online for dynamic requests of IoT applications. By using a fine-grained resource model of energy cost and service end-to-end response time of orchestrated microservices, we formulate the microservice orchestration problem as a multiobjective Markov decision process. We then propose a multiobjective optimization solution based on deep reinforcement learning (DRL) to simultaneously reduce energy consumption and response time. Through extensive experiments, our proposed algorithm presents significant performance results than the state of the art. To the best of our knowledge, this is the first work that addresses microservice orchestration using DRL for multiple conflicting objectives.**

***Index Terms*—Deep reinforcement learning (DRL), energy consumption, Internet of Things (IoT), online microservice orchestration, Quality-of-Service (QoS) assurance.**

I. Introduction

RECENTLY, the Internet of Things (IoT) has been developed rapidly and widely applied in many areas. IoT provides various applications for service provision, e.g., smart city, smart home, smart health, etc. Traditionally, all the components of an IoT application are assembled and tightly packaged as a monolith in the cloud [1]. However, with the explosive growth of IoT devices and service requirements, the complexity and scalability of IoT applications increase exponentially, resulting in the monolithic architecture exposing several issues regarding scalability, flexibility, and reliability [2], [3]. Hence, the microservice architecture is gaining broad support. With this architecture, an IoT application is built as a collection of small independent services, i.e., *microservice*. Each microservice is self-contained and can be implemented, updated, and managed independently [2], [4]. For service provision, a set of microservices (MS) communicate over well-defined APIs (e.g., REST) and form as a microservice chain (MSC). Supported by container technologies (e.g., Docker), microservice simplifies deployment and management of IoT applications to improve energy efficiency and guarantee Quality of Services (QoS) to end users [3], [5], [6].

In this article, we focus on the problem of online *microservice orchestration*, which is responsible for mapping service requests of IoT applications and allocating hardware resources to microservice instances. Benefiting from the lightweight orchestration of containers, microservices can achieve efficient resource management via on-demand *vertical* and *horizontal* resource scaling [7], [8]. However, it also comes with potentially increasing energy consumption and decreasing QoS due to service decomposition [2], [4]. Specifically, allocating more hardware resources (e.g., CPU) to microservice instances in a chain can reduce its end-to-end response time to a service request (referred to as *service time*) for QoS assurance but can also introduce high energy consumption. Therefore, it is necessary to find an optimal orchestration policy to allocate resources efficiently. Still, it is not trivial since microservices are being deployed at a large scale, e.g., Uber's application is composed of more than 1000 instances [9].

The problem of microservice orchestration is similar to typical resource allocation in virtual-machine-based (VM) clouds or networking, which has attracted much attention [10]–[14]. But as a new architecture, microservice orchestration faces unique challenges due to differences from the existing architectures. First, the major difference is that a service request is processed through a predefined chain in microservices. Hence, microservices require a chain-oriented allocation decision. Such allocation has also been studied in service function chaining (SFC) [15], [16]. Since SFC is used in the networking scenario (e.g., 5G slicing) to chain virtual network functions (VNFs) for network service provision, the orchestrator aims to find a set of VNFs to route network requests, which has the optimal usage of CPU/memory resources and link bandwidths. But, the microservices are typically used on the application

Manuscript received 18 June 2021; revised 11 February 2022; accepted 26 February 2022. Date of publication 1 March 2022; date of current version 7 September 2022. This work was supported in part by the Basic Research Programs of Taicang under Grant TC2020JC03; in part by the Guangdong Basic and Applied Basic Research Foundation under Grant 2021A1515110279; in part by the Natural Science Basic Research Program of Shaanxi under Grant 2022JQ-611; and in part by the Fundamental Research Funds for the Central Universities under Grant D5000210588. *(Corresponding author: Jiajia Liu.)*

Yinbo Yu is with the Research and Development Institute, Northwestern Polytechnical University in Shenzhen, Shenzhen 518057, Guangdong, China, and also with the School of Cybersecurity, Northwestern Polytechnical University, Xi'an 710072, Shaanxi, China (e-mail: yinboyu@nwpu.edu.cn).

Jiajia Liu is with the National Engineering Laboratory for Integrated Aero-Space-Ground-Ocean Big Data Application Technology, School of Cybersecurity, Northwestern Polytechnical University, Xi'an 710072, Shaanxi, China (e-mail: liujiajia@nwpu.edu.cn).

Jing Fang is with the School of Computer, Wuhan University, Wuhan 430072, Hubei, China (e-mail: jingfang@whu.edu.cn).

Digital Object Identifier 10.1109/JIOT.2022.3155598

level and microservice requests in the form of HTTP requests. Hence, microservice orchestration tasks [3], [4], [6], [7], [17] usually do not consider link bandwidths, but consider the latency of processing these HTTP-based requests inside each instance. These differences require us to consider the heterogeneity of microservice instances, and resource competition between MSCs [17]. Second, microservices are supplied by containers that support dynamic resource allocation at a low cost. Research works on SFC orchestrations [15], [16] focus on the instance-level resource allocation, in which VNF instances are configured with fixed CPU and memory resources, and the allocation is a problem similar to a chain-oriented job scheduling task. As tested in [18], enabling resource allocation on the level of CPU cycles (i.e., configure and adjust CPU cycles dynamically according to workload) can achieve more efficient and accurate resource management for container-based clouds. Hence, both horizontal and vertical scaling on CPU-cycle-level deserves to be supported in microservice orchestrations [4], [7], [8].

Several recent works have studied microservice orchestrations with different approaches, e.g., threshold-based method [19], [20], greedy algorithm [21], and heuristic algorithm [4], [17], [22], [23]. But several problems have not been studied well. First, these existing approaches require expert knowledge and human interventions. Second, most existing contributions focused on resource allocation where different objectives are incorporated into a single objective via a weighted preference. However, these objectives in the complex IoT environment may conflict with each other and make these scalarized solutions invalid [24]. Besides, preferences for dynamic orchestration may be variable as the changing requirements of IoT providers and users. Once the preference changes, these scalarized solutions need to be retained or even redesigned. Multiobjective orchestration problems are studied in [4], [21], and [22]. But while [4] and [22] conduct static orchestrations, the work in [21] focuses on dynamic microservice scheduling, rather than chain-level orchestrations.

Recently, solutions using reinforcement learning (RL) algorithms and deep neural networks (DNNs) are proposed for resource allocations in many areas [1], [11], [25]–[30], which require less knowledge and can provide more efficient decisions. Hence, in this article, we address these above problems with deep RL (DRL). We focus on the containerized microservice system for IoT application provision, in which multiple containers can run an identical type of microservice (we refer to as microservice *instances*). The problem of online microservice orchestration is to make decisions for the fluctuating workload of service requests at runtime with two objectives of reducing energy consumption and service time. This problem involves complicated decisions, including selecting servers, activating and allocating resources to instances, and mapping service requests to instances simultaneously. Besides, various components in the microservice system introduce a large dimension of states. Hence, the problem involves a huge state and action space and can cause RL algorithms to suffer from the curse of dimension and hinder the convergence of network updates. To this end, we design a multiobjective DRL algorithm for online microservice orchestration, called MOTION.

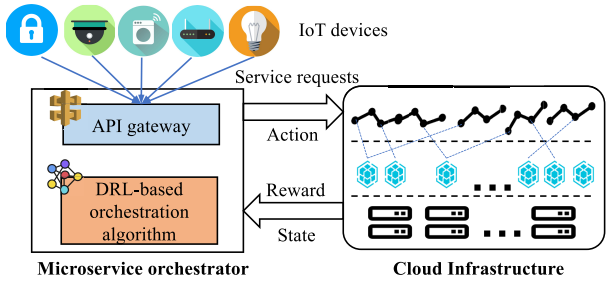


Fig. 1. Illustration of the system model.

To our best, this is the first work that considers DRL for IoT microservice orchestration with multiobjectives. Our main contributions are summarized as follows.

1) We formulate the problem of microservice orchestration for IoT application provision as a biobjective optimization problem. We aim to reduce long-term energy consumption and service time jointly.
2) We model the orchestration process as a multiobjective Markov decision process, in which we decouple the large action space into a set of intermediate decisions and shape returns with smaller intermediate rewards for improving learning efficiency.
3) We design an on-policy actor–critic-based multiobjective DRL algorithm, MOTION, which can learn the set of Pareto-optimal policies within a single model.
4) The evaluation results show that MOTION achieves significant performance for dynamic multiobjective tasks of IoT microservice orchestration.

The remainder of this article is organized as follows. We formulate the microservice system of IoT and the orchestration problem in Section II. Section III gives the detailed design of MOTION. Numerical results of evaluations are presented in Section IV. We finally discuss the related works in Section V and conclude this article in Section VI.

II. System Model and Problem Statement

As illustrated in Fig. 1, we consider the scenario that an IoT system uses a microservice system to provide applications for IoT devices management and service provision. The microservice system is composed of a cloud computing infrastructure and a microservice orchestrator. The orchestrator deploys IoT applications in the form of MSCs and needs to simultaneously reduce their service time and energy consumption for running them. In this section, we formulate the problem of online microservice orchestration in detail and summarize used notations in Table I.

A. System Model

We consider that the cloud infrastructure consists of a set $\mathbb{N}$ of homogeneous server nodes that have identical physical resources, including CPU, memory, and disk. The orchestrator is composed by $\mathbb{M}$, $\mathbb{S}$, and $\mathbb{R}$: $\mathbb{M}$ is an MS, each of which m is encapsulated in a container image; $\mathbb{S}$ is a set of MSC (i.e., IoT applications) in the form of a set $\mathcal{V}^{sc}$ of ordered MS instances communicating over REST APIs. An MS instance is

TABLE I
NOTATION

$\mathbb{N}$	set of server nodes
$\mathbb{M}$	set of microservices (MS)
$\mathbb{S}$	set of microservice chains (MSCs)
$\mathbb{R}$	set of service requests
λ	service rate of a service request
ttl	the live duration of a service request
$\mathbb{I}$	set of MS instances
c^e	elastic resource need of an MS
c^r	rigid resource need of an MS
$c_{\langle n,m,i\rangle}$	CPU cycles in the instance $i_{\langle n,m\rangle}$
$\mathcal{C}$	CPU scheduling period
$\mathbb{C}$	total CPU resource in a server node
C_n	used CPU cycles in the server n
ς	required CPU cycles of a thread
η	number of threads in an instance
w	computational parameter for response time
τ^q	scheduling time of CPU cycles for a thread
τ^p	request processing time of an instance
$x^r_{\langle n,m,i\rangle}$	binary variable of service mapping between request r and instance $i_{\langle n,m\rangle}$
T	end-to-end service time of an MSC
$\mathcal{Q}^A$	an acceptance ratio of new coming service requests
$\mathcal{Q}^T$	service time of all live requests
$\mathcal{O}^{SA}$	server activation cost
$\mathcal{O}^{ID}$	instance deployment cost
$\mathcal{O}^{IR}$	instance resizing cost
$\mathcal{O}^D$	objective function of dynamic costs ($\mathcal{O}^{SA}$, $\mathcal{O}^{ID}$, and $\mathcal{O}^{IR}$)
$\mathcal{O}^P$	static cost of overall power rate
α^R_r	binary status of request (r) mapping
α^S_n	binary status of server (n) activation
$\alpha^A_{\langle n,m,i\rangle}$	binary status of instance ($i_{\langle n,m\rangle}$) activation
$\alpha^I_{\langle n,m,i\rangle}$	binary status of instance ($i_{\langle n,m\rangle}$) resizing
φ^s	horizontal scaling cost of activating a server
φ^h	horizontal scaling cost of instantiating a container
φ^v	cost of resizing per CPU cycle
P	power rate of a server node
U	CPU utilization of a server node

the actual container instantiation of an MS image; $\mathbb{R}$ is a set of service requests from IoT users, each of which *r* is denoted by a tuple $\langle sc, \lambda, ttl\rangle$ where $sc \in \mathbb{S}$ is the type of MSC, λ is the service rate, and *ttl* is the live duration. The orchestrator uses an API gateway with a series of service APIs to receive service requests and takes decisions to allocate resources to MSCs and route requests to these activated MSCs.

For simplicity, we focus on CPU cycles as the main physical resource to allocate since CPU capabilities are the most critical part of QoS assurance and energy consumption [31]. We build the model for CPU cycle allocation based on Docker, which uses a CFS scheduler via *cgroup* command to control CPU access [18]. The scheduler has a CPU scheduling period $\mathcal{C}$ and a *n*-core CPU can be sliced into $n * \mathcal{C}$ cycles to schedule; thereby, we have $\mathbb{C} = n * \mathcal{C}$.

We assume that the orchestrator works in a time-slotted fashion, spanning time slots $1, 2, \ldots, \mathbb{T}$. At the start of each slot (referring to a timestamp $t \in \mathbb{T}$), it orchestrates microservices for the leaving or new arriving service requests $\mathbb{R}^{(t)}$ during the last slot. We denote by $\mathbb{I}$ the set of MS instances that can be deployed totally and $\mathbb{I}_{\langle n,m\rangle}$ the instance set of the MS $m \in \mathbb{M}$ that can be deployed in the server $n \in \mathbb{N}$. Let us refer to $i_{\langle n,m\rangle}$ and $c^{(t)}_{\langle n,m,i\rangle}$ as the *i*th instance of $\mathbb{I}_{\langle n,m\rangle}$ and its allocated CPU cycles, respectively. We denote by $C^{(t)}_n \leq \mathbb{C}$ the CPU cycles in the server $n \in \mathbb{N}$ being used by these instances at timestamp *t*.

Different MS $m \in \mathbb{M}$ has unique logic of processing requests and exposes a REST API to interact with others. Therefore, it has unique *CPU cycle requirements c* and corresponding *response time* T_m for processing REST requests coming from its API. The CPU cycle requirements *c* of an instance can be divided into *rigid* (c^r) and *elastic* ($c^{e,(t)}$) [32]: c^r represents the constant fraction for maintaining its container and does not scale with workloads; $c^{e,(t)}$ specifies the load-dependent fraction allocated to threads to achieve the required performance, for which we introduce an unique and constant ς to denote the required cycles of a thread for finishing processing a request [4]. Hence, the CPU cycles $c^{(t)}_{\langle n,m,i\rangle}$ of an MS instance $i_{\langle n,m\rangle}$ are defined by

$$c^{(t)}_{\langle n,m,i\rangle} = c^r_m + c^{e,(t)}_{\langle n,m,i\rangle} = c^r_m + \varsigma_m \eta^{(t)}_{\langle n,m,i\rangle} \tag{1}$$

where $\eta^{(t)}_{\langle n,m,i\rangle} \in \mathbb{Z}_{\geq 0}$ is the number of threads in the instance $i_{\langle n,m\rangle}$. We assume that if $\eta^{(t)}_{\langle n,m,i\rangle} = 0$, the instance is shut down and does not occupy CPU cycles. We use a binary variable $\alpha^{A,(t)}_{\langle n,m,i\rangle} \equiv \eta^{(t)}_{\langle n,m,i\rangle} > 0$ to denote this status and have the following expression:

$$C^{(t)}_n = \sum_{m\in\mathbb{M}} \sum_{i\in\mathbb{I}_{\langle n,m\rangle}} c^{(t)}_{\langle n,m,i\rangle} \alpha^{A,(t)}_{\langle n,m,i\rangle}. \tag{2}$$

We envisage that an MS instance is configured with a set of threads to process requests. Similar to [6] and [26], we adopt a three-parameter model $\langle \lambda^{(t)}_{\langle n,m,i\rangle}, \eta^{(t)}_{\langle n,m,i\rangle}, w_m\rangle$ to calculate the processing time of service requests in an instance $i_{\langle n,m\rangle}$, where λ_i is the overall rate that need it to serve at epoch *t*, and w_m is the computational parameter of MS *m* for processing incoming service requests. Besides, considering that the server needs to schedule CPU cycles for threads used in container instances, it will introduce additional delays to request processing. Typically, to schedule CPU cycles in a set of service nodes, a global resource model (e.g., M/M/c) is necessary, which can provide more accurate results due to its global view. For microservice orchestration, different server nodes may have capabilities for processing service requests varying over time since instances of an MSC are usually distributed in different service nodes. It is hard to accurately model such various and complex behaviors among server nodes into a single model. Hence, we build the resource model individually. Following the previous paper [33], we use a simple but popular model (M/M/1 queue system) to model the behavior of CPU cycle scheduling for a thread as:

$$\tau^q = \frac{1}{\mathcal{C}/_{\varsigma_m} - 1} \tag{3}$$

where $\mathcal{C}/_{\varsigma_m}$ denotes the cycle schedule rate. Finally, the average response time of the instance for processing requests at epoch *t* can be calculated as

$$T_m = \tau^{p,(t)} + \tau^{q,(t)} = \frac{\lambda^{(t)}_{\langle n,m,i\rangle} w_m}{\eta^{(t)}_{\langle n,m,i\rangle}} + \frac{1}{\mathcal{C}/_{\varsigma_m} - 1}. \tag{4}$$

Since a service request $r^{sc,(t)} \in \mathbb{R}^{(t)}$ is processed by an MSC $sc \in \mathbb{S}$, the end-to-end *service time* of an IoT application is the time from when the request arrives at the first MS instance of sc to when it ended after traversing all MSs of sc, including processing time in instances and transmission time among instances. While a lot of advanced techniques have optimized transmission time among instances (e.g., in-memory communications [34]), it has already been widely agreed that processing delay is the most dominant one [35]. Hence, for tractability, we focus on the processing time and calculate the service time as follows:

$$\overline{T}^{sc,(t)} = \sum_{m \in \mathcal{V}^s} \sum_{n \in \mathbb{N}} \sum_{i \in \mathbb{I}_{\langle n,m \rangle}} x^{r,(t)}_{\langle n,m,i \rangle} T^{(t)}_{\langle n,m,i \rangle}. \tag{5}$$

Since an MS instance may be shared among several MSCs, each of which serves different service request streams, we need to coordinate instances to map these MSCs. We call this operation *service mapping* and use a binary variable $x^{r,(t)}_{\langle n,m,i \rangle}$ to represent the status of service mapping that assumes the value of 1 if the MSC serving a request $r^{sc,(t)}$ traverses the instance $i_{\langle n,m \rangle}$ and 0 otherwise.

B. Problem

At each timestamp t, the orchestrator decreases the value of *ttl* of all live service requests by one and removes timeout requests. Then, the orchestrator needs to generate an orchestration policy $\pi^{(t)}$ for new coming service requests with two optimization objectives: QoS and energy cost.

QoS: To improve QoS, the orchestrator needs to improve the *acceptance ratio* of service requests and reduce the *service time* of accepted requests. At each timestamp t, each service request stream will be mapped into an MSC with a set of instances. We set a threshold $\lambda^{\max}$ to limit the maximum request rate that an instance can serve. If a service request is mapped into an instance i, but causes $\lambda^{\max}$ to be exceeded, the orchestrator will reject this request. Hence, at each t, we have an acceptance ratio of new coming service requests as

$$\mathcal{Q}^{A,(t)} = \frac{\sum \alpha^{R,(t)}_r}{|\mathbb{R}^{(t)}|}, r \in \mathbb{R}^{(t)} \tag{6}$$

where $\alpha^{R,(t)}_r \equiv \bigwedge_{m \in \mathcal{V}^s} \{ \bigvee_{n \in \mathbb{N}, i \in \mathbb{I}_{\langle n,m \rangle}} x^{r,(t)}_{\langle n,m,i \rangle} \wedge \lambda^{(t)}_{\langle n,m,i \rangle} < \lambda^{\max}_m \}$ is a binary variable that assumes the value 1 if the request r is mapped into instances successfully.

After removing timeout requests, the orchestrator reduces *service time* of all live requests with the following objective:

$$\mathcal{Q}^{T,(t)} = \frac{\sum_{r^{sc} \in \mathbb{R}^{(t)}} \frac{\overline{T}^{\max,(t)}_{r^{sc}} - \overline{T}^{(t)}_{r^{sc}}}{\overline{T}^{\max,(t)}_{r^{sc}} - \overline{T}^{\min,(t)}_{r^{sc}}}}{|\mathbb{R}^{\text{live},(t)}|} \tag{7}$$

where we first use two variables $\overline{T}^{\max,(t)}_{r^{sc}}$ and $\overline{T}^{\min,(t)}_{r^{sc}}$ to normalize the service time $\overline{T}^{(t)}_{r^{sc}}$, which are both calculated according to (5), but with different parameters: $\overline{T}^{\max,(t)}_{r^{sc}}$ is the upper bound service time of an MSC $sc \in \mathbb{S}$ and is calculated with $\lambda^{\max}_m$ and a single thread; $\overline{T}^{\min,(t)}_{r^{sc}}$ is the lower bound service time and is calculated with the rate λ_r of r^{sc} and a maximum allocatable number of threads $\eta^{\max}_m$. We further normalize the sum of all normalized service times (i.e., $\sum_{r^{sc} \in \mathbb{R}^{(t)}} ([\overline{T}^{\max,(t)}_{r^{sc}} - \overline{T}^{(t)}_{r^{sc}}]/[\overline{T}^{\max,(t)}_{r^{sc}} - \overline{T}^{\min,(t)}_{r^{sc}}])$) with the number of all live accepted requests $|\mathbb{R}^{\text{live},(t)}|$. Finally, we combine these two normalized QoS functions into a single function

$$\mathcal{Q}^{(t)} = \mathcal{Q}^{A,(t)} \mathcal{Q}^{T,(t)}. \tag{8}$$

Energy Cost: The energy consumption of a server $n \in \mathbb{N}$ can be divided into *dynamic* and *static* cost. While the former is costed by allocating resources at the beginning of each timestamp, the latter is the energy consumption over a time period T that can be calculated by $E_n = P_n T$, where P_n is the average power rate of n.

For the *dynamic cost*, we consider three potential causes involved in the microservice orchestration at each t.

1) *Server activation cost* is the boot-up horizontal scaling cost φ^s of activating a server to provide physical resources for instance deployment. The overall server activation cost can be expressed as

$$\mathcal{O}^{SA,(t)} = \sum_{m \in \mathbb{M}} \varphi^s \left(\alpha^{S,(t)}_n - \alpha^{S,(t-1)}_n \right) \tag{9}$$

where $\alpha^{S,(t)}_n$ is a binary variable that assumes the value 1 if the server $n \in \mathbb{N}$ is active; otherwise, its value is 0. We assume that if all instances $\mathbb{I}_n$ are not allocated CPU cycles (i.e., $\forall m \in \mathbb{M}, i \in \mathbb{I}_n, \eta^{(t)}_{\langle n,m,i \rangle} = 0$), the server n should be shut down. Hence, $\alpha^{S,(t)}_n \equiv \bigwedge_{m \in \mathbb{M}, i \in \mathbb{I}_{\langle n,m \rangle}} \alpha^{A,(t)}_{\langle n,m,i \rangle}$.

2) *Instance deployment cost* is the boot-up horizontal scaling cost φ^h of instantiating a new container from an MS image and deploying into a server (i.e., *horizontal scaling*). It can be expressed as

$$\mathcal{O}^{ID,(t)} = \sum_{m \in \mathbb{M}} \varphi^h \left[\sum_{n \in \mathbb{N}} \mathbb{I}^{(t)}_{\langle n,m \rangle} - \sum_{n \in \mathbb{N}} \mathbb{I}^{(t-1)}_{\langle n,m \rangle} \right]^+ \tag{10}$$

where $[A-B]^+ = \max\{A-B, 0\}$, indicating the number of newly added instances.

3) *Instance resizing cost* is the cost of adding or removing CPU cycles in an instance (i.e., *vertical scaling*). Docker allows to reallocate CPU cycles to containers on-the-fly. We associate a cost φ^v of resizing per CPU cycle. The overall instance resizing cost can be expressed as

$$\mathcal{O}^{IR,(t)} = \sum_{n \in \mathbb{N}} \sum_{m \in \mathbb{M}} \sum_{i \in \mathbb{I}^{(t)}_{\langle n,m \rangle}} \varphi^v \alpha^{I,(t)}_{\langle n,m,i \rangle} \tag{11}$$

where $\alpha^{I,(t)}_{\langle n,m,i \rangle} \equiv c^{(t-1)}_{\langle n,m,i \rangle} > 0 \wedge c^{(t)}_{\langle n,m,i \rangle} > 0$ is a binary variable which takes a value of 1 if instance $i_{\langle n,m \rangle}$ is resized, and 0 otherwise.

We normalize those above dynamic costs and combine them into a maximum objective function as

$$\mathcal{O}^{D,(t)} = \left(2 - \frac{\mathcal{O}^{SA,(t)}}{\varphi^s |\mathbb{N}|} - \frac{\mathcal{O}^{ID,(t)} + \mathcal{O}^{IR,(t)}}{\varphi^h |\mathbb{I}^{\max}|} \right) / 2. \tag{12}$$

For the *static cost*, following existing literature [31], we formulate the power rate of a server n using a linear model:

$$P^{(t)}_n = P^{\text{idle}} + (P^{\max} - P^{\text{idle}}) * U^{(t)}_n. \tag{13}$$

where P^{idle} is the static power rate for maintaining its host system and Docker daemon; $P^{\max}$ is the maximum power rate when n is fully utilized; and $U_n^{(t)}$ is the CPU utilization of n which is calculated as follows:

$$U_n^{(t)} = \left(\sum_{m \in \mathbb{M}} \sum_{i \in \mathbb{I}_{n,m}} \left(c^{e,(t)}_{\langle n,m,i \rangle} + c^{r}_{m} \right) \right) / \mathbb{C}. \tag{14}$$

Hence, during timestamp t, the normalized static cost of overall power rate of the microservice system can be calculated as follows:

$$\mathcal{O}^{P,(t)} = 1 - \frac{\sum_{n \in \mathbb{N}} \alpha_n^{S,(t)} P_n^{(t)}}{P^{\max} |\mathbb{N}|}. \tag{15}$$

We finally have the overall cost during timestamp t as

$$\mathcal{O}^{(t)} = \left(\mathcal{O}^{D,(t)} + \mathcal{O}^{P,(t)} \right) / 2. \tag{16}$$

To optimize the two objectives ($\mathcal{Q}^{(t)}$ and $\mathcal{O}^{(t)}$) simultaneously at runtime, the most common solution is to incorporate multiple objectives into a single scalarized objective via a weighted preference. However, such a scalarized solution is only valid when objectives do not complete with each other [24]. For these competing objectives, it is typically not clear how to evaluate available tradeoffs among them. In the scenario of service provision for complex IoT requirements, it is better to generate different optimal policies and related results, rather than ask providers to provide a preference. Therefore, we formulate the microservice orchestration for IoT applications as a biobjective problem as follows:

$$\max \sum_{t \in \mathbb{T}} \left(\mathcal{Q}^{(t)}, \mathcal{O}^{(t)} \right) \tag{17}$$

$$\text{s. t. } C_n^{(t)} \leq \mathbb{C} \ \forall n \in \mathbb{N} \tag{18}$$

$$0 \leq \eta^{(t)}_{\langle n,m,i \rangle} \leq \eta_m^{\max} \ \forall i_{\langle n,m \rangle} \in \mathbb{I}_{\langle n,m \rangle} \tag{19}$$

which has two types of decision variables $\eta^{(t)}_{\langle n,m,i \rangle}$ and $x^{r,(t)}_{\langle n,m,i \rangle}$ to determine thread allocation and service rate mapping, respectively. Other nondecision variables can be calculated according to $\eta^{(t)}_{\langle n,m,i \rangle}$ and $x^{r,(t)}_{\langle n,m,i \rangle}$, including $\alpha_r^{R,(t)}, \alpha_n^{S,(t)}, \alpha^{A,(t)}_{\langle n,m,i \rangle}$, and $\alpha^{I,(t)}_{\langle n,m,i \rangle}$. We use constraint (18) to restrict all $\eta^{(t)}_{\langle n,m,i \rangle}$ in n from exceeding $\mathbb{C}$. Similar to $\lambda_m^{\max}$ for service rate mapping, we set an upper threshold $\eta_m^{\max}$ for thread allocation to each MS m. If we only focus on thread allocation for instance deployment, or service mapping, problem (17) can be simplified as a bin-packing problem which is a classic NP-hard problem. Our problem is more complex than bin packing since it involves more staged decisions and multiple tradeoffs between two conflicting objectives. Hence, it is obvious that without any weighted preference, our problem is NP-hard.

III. Multiobjective DRL for Microservice Orchestration

To address the multiple-objective optimization problem, in this section, we present our MOTION.

A. Multiobjective Markov Decision Process

We formulate the orchestration for IoT applications as a multiobjective Markov decision processing (MOMDP) denoted by a $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \boldsymbol{r}, \Omega, f_\Omega \rangle$. In the MOMDP, the infrastructure is considered as the environment and the orchestrator plays the role of an agent which interacts with the environment over time. At each epoch t, the agent observes a state $s^{(t)}$ in the state space $\mathcal{S}$ and selects an action $a^{(t)}$ from the action space $\mathcal{A}$. After the environment performs $a^{(t)}$, it will return a vector reward $\boldsymbol{r}^{(t)}$ to the agent according to the vector reward function $\boldsymbol{r}(s^{(t)}, a^{(t)})$, and steps to the next state $s^{(t+1)}$ according to a transition probability $p(s^{(t+1)}|s^{(t)}, a^{(t)}) \in \mathcal{P}$. The agent follows a stochastic policy $\pi(a^{(t)}|s^{(t)}, \boldsymbol{\omega})$, which represents the probability of selecting action $a^{(t)}$ at state $s^{(t)}$ and a preference $\boldsymbol{\omega} \in \Omega$. Different from the standard MDP, the MOMDP has a preference space Ω and preference functions f_Ω, with which it can produce a scalar utility $f_{\boldsymbol{\omega}}(\boldsymbol{r}^{(t)})$ using preference $\boldsymbol{\omega} \in \Omega$. In this article, we focus on linear preference functions, i.e., $f_{\boldsymbol{\omega}}(\boldsymbol{r}(s,a)) = \boldsymbol{\omega}^\top \boldsymbol{r}(s,a)$.

Action Space: As service request streams $\mathbb{R}^t$ coming or leaving at each timestamp t, the agent takes actions $\mathcal{A}^{(t)}$ to handle dynamics, including actions for releasing timeout resources, configuring activated instances in servers, and mapping service requests into instances. However, generating all these actions in a single decision epoch requires a large action space which could make policy learning very challenging [36]. Hence, we reduce the action space as follows.

The agent first releases timeout resources (e.g., $\lambda^{(t),\text{timeout}}$ and $\eta^{(t),\text{timeout}}$) at the beginning of each timestamp. After that, time is frozen to orchestrate microservice requests for all new coming service requests in turn, until all of them are orchestrated. Specifically, we decouple each timestamp t into several small decision epochs, at each of which the agent only takes two actions according to the two decision variables of problem (17): 1) map an MS of a new coming service request into an instance and 2) adjust the resources allocated in the instance. Therefore, we have the action space as follows:

$$\mathcal{A} = \left\{ (a^x, a^\eta) \middle| \begin{matrix} a^x \in \mathbb{Z}_{\geq 0}, & a^x \leq |\mathbb{I}_m||\mathbb{N}| \\ a^\eta \in \mathbb{Z}_{\geq 0}, & a^\eta \leq \eta_m^{\max} - \eta_m \end{matrix} \right. \tag{20}$$

where $\mathbb{I}_m$ denotes the set of instances for an MS m that can be launched in a server. For the MS m in the MSC of a request r, $a^x = 0$ means that the agent does not map m into instances; otherwise, it decides that the $(a^x \bmod |\mathbb{N}| - 1)$th instance in the $(a^x/|\mathbb{N}| - 1)$th server will serve m and its served rate becomes $\lambda^{(t)} = \lambda^r + \lambda^{(t-1)} - \lambda^{(t),\text{timeout}}$. a^η is used to adjust the thread numbers in an instance: $\eta^{(t)} = a^\eta + \eta^{(t-1)} - \eta^{(t),\text{timeout}}$. For simplicity, we combine these two actions into a single action space

$$\mathcal{A} = \left\{ 0, \ldots, |\mathbb{I}_m||\mathbb{N}|\eta_m^{\max} \right\}. \tag{21}$$

Given the constraints of problem (17), an action $a \in \mathcal{A}$ may lead to violations, i.e., an invalid action. Once the agent generates an invalid action, it rejects the service request, backtracks its state to the state before starting to map the request, and routes the next request.

State Space: We formulate the state $s \in \mathcal{S}$ for microservice orchestration at each decision epoch t as follows:

$$\mathcal{S} = \left\{ s | s = \left(\bigcup_{\mathbb{I}} \lambda, \bigcup_{\mathbb{I}} \eta, sc^r, \lambda^r, m \right) \right\} \tag{22}$$

where $\bigcup_{\mathbb{I}} \lambda$ and $\bigcup \mathbb{I} \eta$ denote the mapped service rate and the number of threads in each instance, respectively, sc^r, λ^r denotes the request r's service chain and arrival rate, respectively, and m denotes the next microservice in sc^r which needs the orchestrator to take an action.

Reward: The major reward signal in the environment of microservice orchestration is a vector reward $\boldsymbol{r} = [\mathcal{Q}, \mathcal{O}]$. However, since we split each timestamp into several small decision epochs, only when all these epochs are completed, the environment can return $\boldsymbol{r}$. It is a classic *sparse reward issue* [37] that challenges policy learning since there is no effective signal to guide the agent to generate intermediate decisions. We address this issue by adding artificial shaping rewards to the native rewards that at a decision epoch for the ith MS of an MSC sc^r, we feed the agent with an additional reward vector $\boldsymbol{r}(s, a) = [r^q, r^o]$ for intermediate decisions

$$r^q = \begin{cases} -\frac{(i+1)*\lambda^r}{\lambda^{\text{avg},\mathcal{R}^{(t)}}}, & \text{if } a \in \mathcal{A} \text{ is invalid} \\ 1 - \frac{T^{\max} - T}{T^{\max} - T^{\min}}, & \text{otherwise} \end{cases} \tag{23}$$

$$r^o = \begin{cases} -(i+1), & \text{if } a^\eta \text{is invalid} \\ 1 - \frac{a^\eta \varsigma}{\eta^{\max} \varsigma + c^r}, & \text{otherwise} \end{cases} \tag{24}$$

where $\lambda^{\text{avg},\mathcal{R}^{(t)}}$ is the average rate of $\mathcal{R}^{(t)}$. We give minor intermediate rewards for valid actions, but penalties invalid actions. When $\mathbb{R}^{(t)}$ are fully orchestrated, the agent receives both two reward vectors, and we combine them into a single vector as follows:

$$\boldsymbol{r}(s, a) = \left[\beta^r \middle| \mathbb{R}^{(t)} \middle| \mathcal{Q} + r^q, \beta^r \middle| \mathbb{R}^{(t)} \middle| \mathcal{O} + r^o \right] \tag{25}$$

where we use a parameter β^r and the request length of $\mathbb{R}^{(t)}$ to scale up the major rewards. Controlling major rewards greater than intermediate ones can avoid greedy learning [38].

B. MOTION

Given the MOMDP, we envisage the agent with the goal of maximizing the expectation of the long-term rewards. Starting at a state s and a preference $\boldsymbol{\omega}$, the agent would learn a policy $\pi(a|s, \boldsymbol{\omega})$, which maps each state and corresponding preference to an action so that the selected action can maximize the expected return after s. Hence, the *state-value* function of π is expressed as follows:

$$\boldsymbol{V}(s, \boldsymbol{\omega}) = \mathbb{E}_\pi \left[\sum_{t=1}^{\infty} \gamma^{t-1} \boldsymbol{r}\left(s^{(t)}, \pi\left(a^{(t)} | s^{(t)}, \boldsymbol{\omega} \right) \right) | s^{(1)} = s \right] \tag{26}$$

where $\gamma \in [0, 1]$ is the discount factor and specifies the importance between future rewards and the current reward, i.e., $\gamma = 0$ represents an "myopic" orchestrator only concerned with its immediate reward, while $\gamma = 1$ denotes an orchestrator striving for a long-term higher reward.

Since problem (17) consists of two objectives, different policies can be optimal w.r.t. different objectives. We use the Pareto dominance relation as the optimality criterion to learn policies. Specifically, a policy π_1 dominates another policy π_2 (denoted by $\pi_1 \succ \pi_2$) if at least one objective in $\boldsymbol{V}^{\pi_1}$ is greater than the corresponding objective in $\boldsymbol{V}^{\pi_2}$ and other objectives are not less than. Besides, if π_1 and π_2 both have at least one objective that is greater than the other, they are incomparable. If a policy π dominates or is incomparable to others, we say it be Pareto optimal. Through MOTION, we aim to train an agent to learn a set of Pareto-optimal policies Π^*

$$\pi \in \Pi^* \Rightarrow \exists \boldsymbol{\omega} \in \Omega, \text{ s.t. } \forall \pi' \in \Pi, \boldsymbol{\omega}^\top \boldsymbol{V}^{\pi}(s_0) \geq \boldsymbol{\omega}^\top \boldsymbol{V}^{\pi'}(s_0) \tag{27}$$

where s_0 is an initial state. For each $\pi \in \Pi^*$, there exists at least one preference $\boldsymbol{\omega}$ such that under it, no other policy π' dominates π. The returns of Π^* construct the Pareto frontier: $\mathcal{F} = \{\boldsymbol{r} | \nexists \boldsymbol{r}' \succ \boldsymbol{r}\}$.

To learn Π^* for continuous online microservice orchestration, choosing off-policy and on-policy learning is related to the bias–variance tradeoff [39]. While off-policy RL algorithms (e.g., Q-learning [40], DDPG [41]) perform policy learning using samples from previously collected experiences, on-policy RL algorithms (e.g., A3C [42], Sarsa [43]) update policies using data generated by the current policy. These experiences used by off-policy algorithms are generated by a different policy (e.g., ϵ-greedy), allowing for greater exploration. But such learnings introduce a bias for a time-varying microservice system because these learning experiences may be generated under a different operating condition for the system. Such a bias can make the policy suboptimal, or even divergent [39].

Hence, we follow the actor–critic architecture to design an on-policy DRL algorithm, MOTION. It maintains a multiobjective critic network $\boldsymbol{V}$ parameterized by θ_v, which is responsible for evaluating the performance of the actor and guiding the actor's actions in the next stage, and an actor network π parameterized by θ_π, which is responsible for generating actions and interacting with the environment.

To update the critic network for a single-objective MDP, an objective $y = r + \gamma(1 - d)V(s'; \theta_v)$ over multiple steps is maintained. We aim to learn Π^* for the entire preference space Ω within a single model, rather than multiple scalarized models. Hence, to obtain the set of Pareto-optimal policies Π^* for our MOMDP problem, we extend y with a multiobjective envelope operator $\boldsymbol{\mathcal{T}}$ (proposed in [44])

$$\boldsymbol{y} = (\boldsymbol{\mathcal{T}}\boldsymbol{V})(s, \boldsymbol{\omega}) = \boldsymbol{r} + \gamma \mathbb{E}_{s'}(\boldsymbol{\mathcal{H}}\boldsymbol{V})\left(s', \boldsymbol{\omega}\right) \tag{28}$$

where $(\boldsymbol{\mathcal{H}}\boldsymbol{V})(s', \boldsymbol{\omega}) \coloneqq \arg_{\boldsymbol{V}} \sup_{\boldsymbol{\omega}' \in \Omega} \boldsymbol{\omega}^\top \boldsymbol{V}(s', \boldsymbol{\omega}')$ takes the multiobjective values corresponding to the supremum of $\boldsymbol{V}(s', \boldsymbol{\omega}')$ according to the preference set $\boldsymbol{\omega}$. Given a state s and preferences $\boldsymbol{\omega}$, $\boldsymbol{\mathcal{H}}$ can produce the state-value function for optimization by identifying the convex envelope of the current solution frontier. Maintaining the envelope $\sup_{\boldsymbol{\omega}' \in \Omega} \boldsymbol{\omega}^\top \boldsymbol{V}(\cdot, \boldsymbol{\omega}')$ allows the network to quickly update the optimal policy aligned with a new preference which may have not been explored yet.

Based on the objective $\boldsymbol{y}$, by randomly sampling a set of preferences Ω^s with the size of $N_{\boldsymbol{\omega}}$, we update the parameter θ_v of critic using a mean-squared error loss function

$$L^1(\theta_v) = \mathbb{E}_{s,\boldsymbol{\omega}}\Big[\|\boldsymbol{y} - \boldsymbol{V}(s,\boldsymbol{\omega};\theta_v)\|_2^2\Big], \boldsymbol{\omega} \in \Omega^s. \tag{29}$$

Thanks to the multiobjective envelope operator $\mathcal{T}$, minimizing $L^1(\theta_v)$ can guide the network to update in the direction of Pareto optimization. However, since the optimal frontier of problem (17) contains a large number of discrete actions and states, the landscape of L^1 may change sharply. Hence, it is hard to directly minimize L^1. We introduce a flat loss function

$$L^2(\theta_v) = \mathbb{E}_{s,\boldsymbol{\omega}}\Big[\Big|\boldsymbol{\omega}^\top \boldsymbol{y} - \boldsymbol{\omega}^\top \boldsymbol{V}(s,\boldsymbol{\omega};\theta_v)\Big|\Big] \tag{30}$$

which is used to directly minimize the value metric $(\sup_{\substack{s\in\mathcal{S}\\ \boldsymbol{\omega}'\in\Omega^s}} |\boldsymbol{\omega}^\top(\boldsymbol{V}(s,\boldsymbol{\omega}) - \boldsymbol{V}(s',\boldsymbol{\omega}))|)$. As an auxiliary loss, L^2 can pull L^1 along the direction with better utility. Hence, we combine them as a single loss function

$$L(\theta_v) = \Big(1-\beta^l\Big)L^1(\theta) + \beta^l L^2(\theta) \tag{31}$$

where β^l is a weight to balance $L^1(\theta)$ and $L^2(\theta)$. By slowly increasing its value from 0 to 1, we can shift L from L^1 to L^2. This can provide better opportunities to find the global optimal parameters.

The actor network (θ_π) for the policy $\pi(a|s,\boldsymbol{\omega};\theta_\pi)$ can be directly updated with an estimator of the gradient, known as the REINFORCE rule [45], since the action space of our MOMDP problem is discrete. The estimator updates θ_π in the direction of $\nabla_{\theta_\pi}\log\pi(a \mid s,\boldsymbol{\omega};\theta_\pi)(R - b(s))$, where $b(s)$ is a baseline for reducing the variance of gradient estimate. For Pareto optimization, MOTION uses $\boldsymbol{V}(s,\boldsymbol{\omega})$ as the baseline and has the following advantage function:

$$\begin{aligned}\boldsymbol{A}_{ij} &= \boldsymbol{\omega}_i^\top\big((\mathcal{T}\boldsymbol{V})_{ij} - \boldsymbol{V}\big(s_j,\boldsymbol{\omega}_i;\theta_v'\big)\big)\\ &\quad \forall i \in [1,N_{\boldsymbol{\omega}}], j \in [1,N_\tau]\end{aligned} \tag{32}$$

where N_τ is the length of selected actions. We then update the actor network with respect to parameter θ_π

$$\begin{aligned}d\theta_\pi = &\frac{1}{N_{\boldsymbol{\omega}}N_\tau}\sum_{i,j}\nabla_{\theta_\pi'}\log\pi\big(a_j \mid s_j,\boldsymbol{\omega}_i;\theta_\pi'\big)\boldsymbol{A}_{ij}\\ &+\ \beta^e\nabla_{\theta_\pi'}H\big(\pi\big(s;\theta_\pi'\big)\big).\end{aligned} \tag{33}$$

To discourage premature convergence to suboptimal policies, we improve state explorations by adding the entropy H of the policy π to the above gradient and using a hyperparameter β^e to control its strength.

Based on the above update strategies, we design our MOTION following the asynchronous advantage actor–critic [42]. First, MOTION sets up a set of actor-learner workers for network updates, each is equipped with an independent environment (θ_π and θ_v) to interact with the agent to learn policies following Algorithm 1. Such a setting can improve the variety of transitions under different preferences. Second, MOTION maintains a global shared parameter vector $\langle\theta_\pi^g,\theta_v^g\rangle$. As shown in Algorithm 1, at the beginning of each learning loop, each worker synchronizes its θ_π and θ_v to θ_π^g and θ_π^g, respectively, and then explores and accumulates its own gradients. At the end of each loop, θ_π^g and θ_v^g are asynchronously updated according to θ_π and θ_v, respectively. Third, to address the dilemma of the tradeoff between exploitation and exploration, we use a softmax method with a temperature parameter τ^s to ensure the exploration of actions. Specifically, MOTION selects an action using a Boltzmann distribution as follows:

$$p_i(t+1) = \frac{e^{\hat{\mu}_i(t)/\tau^s}}{\sum_{j=1}^k e^{\hat{\mu}_j(t)/\tau^s}}, i = 1\ldots n \tag{34}$$

where $\hat{\mu}_i(t)$ is the mean of probabilities for each action; τ^s is used to control the randomness of the choice. When $\tau^s = 0$, the exploration acts like pure greedy. As τ^s tends to infinity, it picks actions uniformly at random. Therefore, MOTION starts with a big τ^s for randomness and then slowly decrease the value of τ^s to 0.

Algorithm 1: MOTION-Pseudocode for Each Worker

Input:
- a preference sampling distribution $\mathcal{D}_{\boldsymbol{\omega}}$;
- minibatch sizes for preferences $N_{\boldsymbol{\omega}}$;
- a multi-objective critic network $\boldsymbol{V}_{\theta_v}$ parameterized by θ_v;
- an actor network π parameterized by θ_π;

for *t=1 to* $\mathbb{T}$ **do**
- Synchronize worker-specific parameters $\theta_v = \theta_v^g$ and $\theta_\pi = \theta_\pi^g$; //$\theta_v^g$ and θ_π^g are two global shared parameters for critic and actor, respectively
- Sample a linear preference $\boldsymbol{\omega} \sim \mathcal{D}_{\boldsymbol{\omega}}$;
- A temporal memory $\mathcal{B}$;
- **for** *epoch* = 1 *to* N_τ **do**
  - Observe state s;
  - Select an action a from the $\pi_{\theta_\pi}(s,\boldsymbol{\omega})$ with probability (34);
  - Receive a vector reward $\boldsymbol{r}$ and observe s' and d;
  - Store transition $(s,a,\boldsymbol{r},s',d)$ in $\mathcal{B}$;
  - **if** *d is terminal* **then**
    - reset environment state and resample $\boldsymbol{\omega} \sim \mathcal{D}_{\boldsymbol{\omega}}$;
- Extract all transitions $(s,a,\boldsymbol{r},s',d)$ in $\mathcal{B}$;
- Sample $N_{\boldsymbol{\omega}}$ preferences $\Omega^s = \{\boldsymbol{\omega}_i \sim \mathcal{D}_{\boldsymbol{\omega}}\}$;
- Compute targets y_{ij} using Equ. (28);
- Asynchronous update θ_v^g using the loss function (31);
- Asynchronous update θ_π^g using the gradient (33);
- Increase β^l and decrease τ^s;

C. Architecture

MOTION has three types of DNN architectures, in each, inputs are processed by multiple linear layers (F:$\{|\mathcal{S}| \rightarrow 1024 \rightarrow 1024 \rightarrow 1024\}$, A:$\{1026 \rightarrow 512 \rightarrow |\mathcal{A}|\}$, and C:$\{1027 \rightarrow 64 \rightarrow 2\}$). The F DNN takes the state as the input and is shared for feature extraction before the critic and actor network. The actor network (F+A) gathers the F and A DNNs together by concatenating the extracted feature and the preference and feeding them into the A DNN and outputs the normalized probability distribution over the discrete action space ($|\mathcal{A}|$) by a softmax activation function. The critic network (F+C) gathers the extracted feature, preference, and

TABLE II
MICROSERVICE RESOURCE REQUIREMENTS (CYCLES)

	0	1	2	3	4	5
c^r	25	36	34	24	25	15
ς	92	52	99	68	78	112

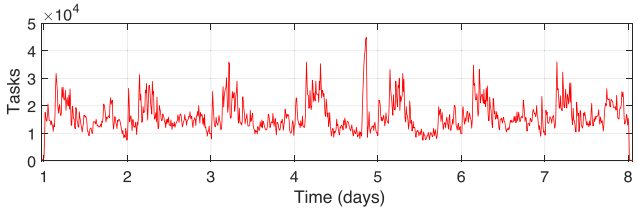


Fig. 2. Number of tasks in the datacenter traces released by Alibaba in 2018.

action, and its output is 2-D returns prediction. Each linear layer is followed by a rectified linear unit (ReLU), and its weight is initialized using Xavier initialization with a standard deviation of 10^{-2}. We use an ADAM optimizer for both the actor and the critic, with a learning rate l^r.

IV. EVALUATION

To evaluate the performance of our proposed multiobjective DRL framework on addressing the online microservice orchestration problem, in this section, we conduct extensive trace-driven simulations and present the results.

A. Experiment Setup

We assume that the IoT system contains $\mathbb{N} = 10$ homogeneous servers (each of which has the scheduling period $\mathcal{C} = 8000$ and totally has $\mathbb{C} = 24\,000$ cycles to schedule), $\mathbb{M} = 6$ microservices with features shown in Table II for deploying instances of IoT applications. Following the CPU cycle allocation range used in [18], we use a lognormal distribution to generate these parameters in Table II. Besides, the orchestrator provides $\mathbb{S} = 8$ MSC ($\{1 \to 2\}$, $\{1 \to 2 \to 5\}$, $\{0 \to 2 \to 4 \to 3 \to 1\}$, $\{0 \to 4 \to 1 \to 5\}$, $\{0 \to 1 \to 3 \to 2 \to 5\}$, $\{3 \to 2 \to 0\}$, $\{4 \to 2 \to 3 \to 0\}$, and $\{2 \to 3 \to 4 \to 5\}$) for serving service requests of IoT applications and we assume each chains has 10 possible arrival rates following a lognormal distribution $N(\mu, \sigma^2)$, where the means $\mu = 400$ and standard deviation $\sigma = 800$. To simulate the workload pattern of IoT application requests, we use the cluster traces released by Alibaba in 2018,[1] as shown in Fig. 2. This dateset provides seven days of traces. We combine 10 min of tasks to replay in one timestamp. Then, we transform these tasks into service requests with unique arrival rate, MSC type and duration to construct the workload data as an experiment. According to the workload, we set $\eta_m^{\max} = 30$ and $\lambda_m^{\max} = \eta_m^{\max}([\mathcal{C}/\varsigma_m] - \eta_m^{\max}) - 0.5$. Note that naturally, the value of these above parameters in practice depends highly on the implementation of microservice systems. Different parameter values may lead to different results. But our problem formulation is general, and our MOTION can reach similar observations with a large variety of parameter values considered.

All the experiments are conducted on a desktop equipped with a 3.6-GHz 8-Core Intel i9 9900k CPU and an RTX 2080Ti GPU. For training the network, we initialize MOTION with parameters including, $\beta^r = 0.8$ for controlling major rewards, $\beta^e = 0.02$ for controlling the strength of entropy, $\tau^s = 10$ for controlling the randomness of action choice, $\gamma = 0.99$ for discounting rewards, $\delta = 10^{-3}$ for increasing β^l from 0 to 1 during training, and $l^r = 10^{-3}$ learning rate.

B. Benchmarks and Evaluation Metrics

In order to assess the quality of our MOTION, we compare it with other both off-policy and on-policy algorithms.

1) *Greedy:* We design two greedy algorithms G1 and G2. G1 is designed to greedily achieve the maximum of objective $\mathcal{Q}$: to map service, G1 always searches the best action at each epoch, with which the used instance can be allocated the most threads and have the minimal service load. G1 can achieve enough small service time under the current condition but at the cost of energy consumption. Instead, G2 aims to maximize objective $\mathcal{O}$ by greedily searching an action that can use the minimum computing resources. Note that both G1 and G2 will not reject any requests.
2) *MNDQN-Greedy:* DQN [40] is an off-policy RL algorithm equipped with a replay buffer to store experiences, which will be sampled for policy updates. To enable DQN for multiobjective, we adopt a method (MNDQN-Greedy) proposed in [46], which trains multiple Q-networks on different weight vectors via a linear scalarization function to cover multiple regions of the weight-space and uses knowledge in previously trained networks to speed up learning. MNDQN-Greedy uses the ϵ-greedy policy to explore actions.
3) *MNDQN-Chev:* Using ϵ-greedy with a linear scalarization function can only find policies that lie in the convex Pareto frontier. Hence, we adopt another scalarization function, Chebyshew [47] in MNDQN for different exploration and implement MNDQN-Chev for comparison.
4) *MO-DDPG:* DDPG [41] is improved based on DQN with deterministic policy gradient and learns with an actor–critic-based network for continuous actions. Based on a discrete version of DDPG,[2] we introduce the multiobjective envelope operator (28) to implement a multiobjective DDPG (i.e., MO-DDPG) for comparison.
5) *MO-SARSA:* Sarsa [43] is a slight variation of the Q-learning algorithm, but learns the value function according to the current action (i.e., on-policy). Similarly, we introduce the multiobjective envelope operator to Sarsa and implement MO-SARSA.

Given these state-of-the-art (SOTA) approaches, we use C-metric ($C(A, B)$) to evaluate the agent's ability to recover

[1] https://github.com/alibaba/clusterdata

[2] https://github.com/LxzGordon/Deep-Reinforcement-Learning-with-pytorch

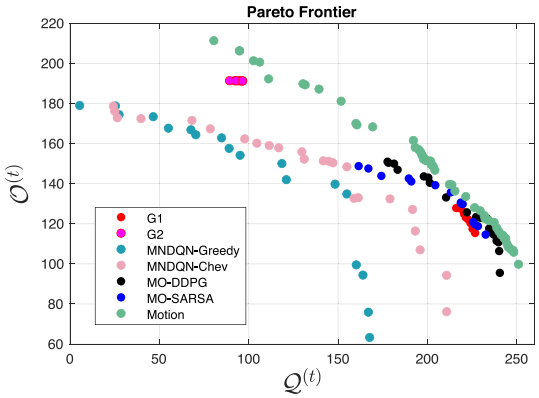


Fig. 3. Solution frontier generated by SOTA algorithms.

TABLE III
QUALITY OF ACHIEVED APPROXIMATED PARETO FRONTIER

C(**MOTION**,G1): C(G1,**MOTION**)=1:0
C(**MOTION**,G2): C(G2,**MOTION**)=1:0
C(**MOTION**,MNDQN-Greedy): C(MNDQN-Greedy,**MOTION**)=1:0
C(**MOTION**,MNDQN-Chev): C(MNDQN-Chev,**MOTION**)=1:0
C(**MOTION**,MO-DDPG): C(MO-DDPG,**MOTION**)=0.9130:0.0172
C(MO-SARSA,**MO-DDPG**): C(**MO-DDPG**,MO-SARSA)=0.0435:0.6429
C(**MOTION**,MO-SARSA): C(MO-SARSA,**MOTION**)=1:0

optimal solutions. $C(A, B)$ estimates the percentage of results generated by the approach B that are dominated by at least one solution in A. The higher the value of $C(A, B)$, the better quality of the frontier achieved by A, e.g., $C(A, B) = 1$ means that all solutions generated by A are better than those by B.

C. Evaluation Results

For a quick comparison, we use the workload data in the first six timestamps to train the network. For MNDQN-Greedy and MNDQN-Chev, we slice the preference space into a weight set containing ten weight vectors (i.e., $\langle 0.09, 0.91\rangle$, $\langle 0.18, 0.82\rangle$, ..., $\langle 0.91, 0.09\rangle$) and train policy for each weight vector for 1000 steps. (In each step, the environment goes through all service requests and policies are updated 50 times with current trajectories or sampled experiences.) For MOTION, we set up 16 workers to train the network with different sampled preferences in parallel for 1000 steps. To be fair, all other RL algorithms are performed for more than 10 000 steps.

Fig. 3 and Table III illustrate the quality of solution frontier generated by MOTION and other benchmark algorithms. We can find that two greedy algorithms (G1 and G2) can achieve fairly good results but only for some fixed preferences. Two DQN-based algorithms (MNDQN-Greedy and MNDQN-Chev) can obtain an extensive frontier of solutions, but all of them are suboptimal. These three envelope-operator-based RL algorithms obtain better solutions than greedy and DQN-based algorithms, in which our MOTION achieves the most comprehensive and best solution. The fundamental reason lies in that MOTION trains the network with different preferences in parallel, controls a temperature τ_s for the randomness of action selected and introduces additional entropy for state exploration.

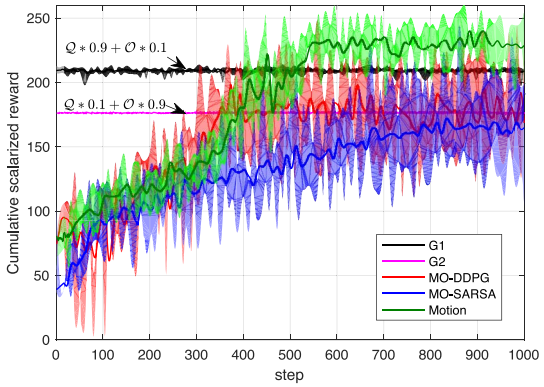


Fig. 4. Average cumulative scalarized rewards for microservice orchestration. While the rewards achieved by G1 and G2 are scalarized by $\langle 0.9, 0.1\rangle$ and $\langle 0.1, 0.9\rangle$, respectively, the other three envelope-operator-based algorithms use randomly sampled weights to scalarize their reward vector.

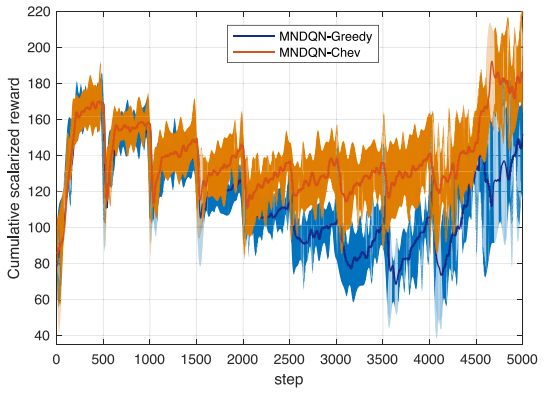


Fig. 5. Average cumulative scalarized rewards for microservice orchestration generated by DQN-based solutions. Note that for each selected weight, the network is trained for 1000 steps and the results of two adjacent steps are combined in this figure.

Figs. 4 and 5 show the cumulative scalarized rewards achieved by different algorithms. We can find that G1 and G2 can obtain comparable results for some fixed preferences but will fail in other preferences. MO-DDPG, MNDQN-Greedy, MNDQN-Chev, and MO-SARSA can achieve convergence, but the results they produce are volatile. The major reason is that they are stuck in suboptimal policies. Besides, since we randomize the order of service requests in the request stream of each timestamp, these off-policy RL algorithms, which learn policies from previously collected experiences, suffer from huge fluctuations. We can find that MOTION achieves more stable learning for such varying requests. Compared to MO-SARSA, our MOTION uses an actor–critic network and combines the advantages both of value-based and policy-based learning. Hence, MOTION can quickly converge to a more optimal policy than other algorithms and produce better rewards for each explored preference.

Furthermore, we compare the efficiency of MOTION and other algorithms under different preferences. Considering their performance compared previously, we only use two greedy algorithms and MO-DDPG as the benchmark algorithms. We train policies with the traffic data on the first day (144 timestamps) and carry out the comparison under the same

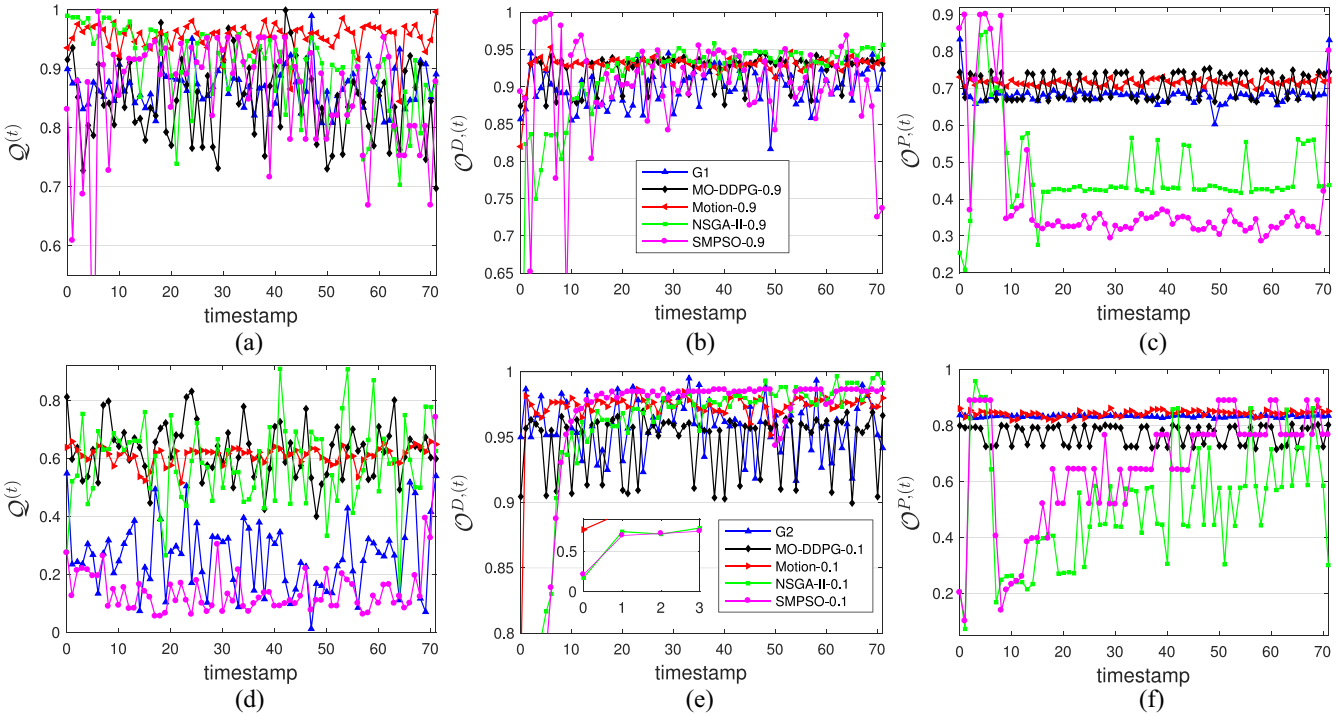


Fig. 6. Service time and operation costs of microservice orchestration. Note that Algorithm-0.9 (MOTION, MO-DDPG, NSGAII, and SMPSO) denotes results achieved under the preference $\boldsymbol{\omega}_9$:$\langle 0.9, 0.1\rangle$, and Algorithm-0.1 denotes results achieved under $\boldsymbol{\omega}_1$:$\langle 0.1, 0.9\rangle$. (a) Service time. (b) Dynamic cost. (c) Static cost. (d) Service time. (e) Dynamic cost. (f) Static cost.

workload pattern (77 timestamps on the second day). Besides, we further take two metaheuristic algorithms: 1) NSGA-II [48] and 2) SMPSO [49] (implemented by the MOEA framework [50]), for comparison, which are two popular and efficient multiobjective optimization algorithms based on iterative decision searches. At each timestamp, different from previous compared DRL approaches, these two algorithms process the state to search all actions in the form of a Pareto-optimal frontier within 20 000 iterations. We mainly compare these algorithms under two extreme preferences $\boldsymbol{\omega}_9 : \langle 0.9, 0.1\rangle$ and $\boldsymbol{\omega}_1 : \langle 0.1, 0.9\rangle$. According to the design of G1 and G2, we compare G1 with other algorithms under $\boldsymbol{\omega}_9$ and G2 under $\boldsymbol{\omega}_1$. Hence, after each timestamp, NSGA-II and SMPSO take the best decision from its Pareto frontier according to these two preferences to update the environment, respectively. At the next timestamp, these two algorithms process the updated state to generate new Pareto frontiers. Fig. 6 shows how all these algorithms decide to allocate resources and map services to instances for incoming service requests under different preferences. We discuss their efficiency as follows.

Service Time: Fig. 6(a) and (d) shows the service time for live service requests in the form of the expression (8) $\mathcal{Q}^{(t)}$. We can find that under both $\boldsymbol{\omega}_9$ and $\boldsymbol{\omega}_1$, our MOTION can generate more stable decisions for QoS guarantee compared to other SOTA algorithms. Under $\boldsymbol{\omega}_9$, the policy achieved by MOTION results in a lower service time (the higher $\mathcal{Q}^{(t)}$, the lower service time). At the first few timestamps (0-15), NSGA-II-9 achieves $\mathcal{Q}^{(t)}$ higher than MOTION, but after that, its value begins to drop and oscillate. This is because NSGA-II focuses on searching the immediate optimal of each timestamp, rather than the long-term optimal. Under $\boldsymbol{\omega}_1$, NSGA-II, MO-DDPG, and MOTION can achieve similar $\mathcal{Q}^{(t)}$ on average, but MOTION is more stable. G1 and G2 greedily search decisions for the intermediate best objective and results in suboptimal results. $\mathcal{Q}^{(t)}$ of SMPSO under $\boldsymbol{\omega}_1$ is the lowest one since it obtains solutions that have better cost at some time [see Fig. 6(e) and (f)]. The distribution of service time achieved by MO-DDPG shows great fluctuation. The fundamental reason is that MO-DDPG obtains a suboptimal policy which leads to varying acceptance ratios.

Dynamic Cost: The dynamic costs of resource allocation for the incoming service requests are shown in Fig. 6(b) and (e). Higher $\mathcal{O}^{D,(t)}$, lower the dynamic cost. We can find that MOTION, NSGA-II, and SMPSO-1 achieve $\mathcal{O}^{D,(t)}$ which has a lower value at the beginning and increases to higher values. $\mathcal{O}^{D,(t)}$ of MOTION quickly increases to a high and stable value after the first timestamp. This is because MOTION tends to reduce long-term energy consumption rather than intermediate ones, such as greedy algorithms and generate decisions for the long run at the beginning, which results in more complex resource allocation. $\mathcal{O}^{D,(t)}$ achieved by NSGA-II-9, NSGA-II-1, and especially SMPSO-1 increases slowly, but can be greater than the one of MOTION at many timestamps. But once we combine them with $\mathcal{O}^{P,(t)}$ as shown in Fig. 6(c) and (f), we can find that $\mathcal{O}^{(t)}$ achieved by MOTION is better than other SOTA approaches since MOTION can find solutions nearer the practical Pareto frontier. $\mathcal{O}^{D,(t)}$ of SMPSO-9 shows great fluctuation. G2 greedily deploys instances on a small number of servers, which can result in a higher $\mathcal{O}^{D,(t)}$ than G1.

Static Cost: We compare the rewards of power rate $\mathcal{O}^{P,(t)}$ at each timestamp achieved by these algorithms in Fig. 6(c) and (f). Since G2 centrally deploys instances on a small number of servers, the distribution of static cost achieved by G2 is the most stable. The service requests in terms of multiple MS instances, MSCs, and service rates highly vary over time. Hence, the off-policy MO-DDPG may generate suboptimal or even divergent policy. We can find in Fig. 6(c) and (f), the reward distribution of operation costs achieved by MO-DDPG floats more steadily than MOTION. NSGA-II and SMPSO achieve similar results that $\mathcal{O}^{P,(t)}$ starts with a very low value, then reaches several values higher than other benchmarks, and goes back to low values again. Under $\boldsymbol{\omega}_1$, after around 13 timestamps, $\mathcal{O}^{P,(t)}$ achieved by NSGA-II and SMPSO starts to grow slowly. At some timestamp, they can reach the highest values. The fundamental reason for such results is that NSGA-II and SMPSO focus on the immediate optimal of each timestamp rather than the long-term optimal over all timestamps, which will result in great high fluctuation over time. Under both $\boldsymbol{\omega}_9$ and $\boldsymbol{\omega}_1$, MOTION can achieve a better policy which can adjust decisions following fluctuating service requests.

In summary, MOTION outperforms other compared algorithms in most cases that it can obtain decision making with better and stable service time, dynamic and static costs.

V. Related Work

By combining loosely coupled services and container technology, microservices bring new discussions on resource allocation problems. We briefly review existing approaches for microservice orchestration in this section.

Bhamare *et al.* [51] studied the scheduling microservices across multiple clouds and reduced overall service time of microservices and the total traffic via a weighted affinity-based scheduling heuristic. Sampaio *et al.* [52] proposed an adaptation mechanism based on SMT and first-fit algorithms to optimize the placement of microservices at runtime with objectives of improving application performance and resource utilization. Niu *et al.* [17] designed a load balancing scheme based on Nash bargaining, which balances the number of containers assigned to different microservices to reduce the service time of MSC. Yu *et al.* [3] focused on the load balancing problem of microservice-based IoT applications in the form of a directed acyclic graph with the goal similar to [17] and proposed a fully polynomial-time approximation scheme to achieve the goal. Samanta *et al.* [23] proposed an online auction-based mechanism to address the resource sharing incentive problem in microservice-based edge clouds via balancing resources in overprovisioning and underprovisioning microservices. Samanta and Tang [6] proposed a dynamic microservice scheduling scheme for mobile-edge computing (MEC) enabled IoT with the objectives of maximizing the energy efficiency and providing fair QoS.

With the explosion of machine learning techniques, a lot of solutions using DRL algorithms are proposed to address resource allocation for microservice architectures. Rossi *et al.* [53] studied the problem of controlling the horizontal and vertical scaling of container-based applications according to varying workloads and designed a full backup model-based RL method. Similarly, Roig *et al.* [29] studied the management and orchestration of container-based VNFs and proposed a parameterized actor–critic method to control the scaling of container instances. Wang *et al.* [26] focused on addressing the problem of coordinating microservices in MEC via DRL to reduce the overall service delay with low costs. Chen *et al.* [1] studied the problem of deploying services based on microservice to reduce the waiting time of IoT devices by proposing a DRL-based approach.

These DRL-based approaches have shown superior performance to traditional approaches. However, most of these approaches focused on single-objective and coarse-grained (i.e., instances are allocated with fixed resources) resource allocations. In our previous work [4], we formulated microservice orchestration with a CPU-cycle-level resource model and designed a PSO-based multiobjective optimization algorithm to address the problem of microservice orchestration. However, this focused on static microservice orchestration. Fard *et al.* [21] studied the problem of dynamic multiobjective microservice scheduling via a greedy algorithm. However, they do not address the chain-level organization with long-term optimal policies. Hence, in this article, based on existing advanced approaches, we study the chain-level multiobjective microservice orchestration problem for IoT application provisions.

VI. Conclusion and Future

This article presented an on-policy-based DRL algorithm to implement online microservice orchestration for IoT application provision. Based on features of the microservice architecture, we designed the microservice orchestration problem as a biobjective optimization problem that is aware of QoS assurance and energy consumption. We then formulated the problem as a MOMDP and proposed a multiobjective DRL algorithm (MOTION) for autonomous online microservice orchestration. Experiments showed that MOTION outperforms benchmark heuristic greedy algorithms as well as other DRL algorithms. To the best of our knowledge, this was the first work that considers DRL for objective microservice orchestration.

As future research directions, we consider improving MOTION from two aspects. First, MOTION mainly focused on computing resources. The allocations of other resources, such as memory and I/O operations, will be considered, which are essential for energy-saving and QoS guarantee. Second, in this work, we only considered a simple scenario that the interconnections between microservice instances have the same delay. But in fact, such delay varies with different types of interconnections, e.g., connections on the same kernel, in the same subnetwork, or across multiple network domains. Besides, we follow the existing literature to use an M/M/1 queueing model for CPU cycle scheduling, rather than a global

model which can provide more accurate allocation. In M/M/1, the arrivals are assumed to follow a Passion distribution which may be not realistic. Hence, a more accurate and realistic model of network environment and resource management is worth building for accurate and practical microservice orchestration.

REFERENCES

[1] 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.

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

[3] R. Yu, V. T. Kilari, G. Xue, and D. Yang, "Load balancing for interdependent IoT microservices," in *Proc. IEEE INFOCOM*, 2019, pp. 298–306.

[4] Y. Yu, J. Yang, C. Guo, H. Zheng, and J. He, "Joint optimization of service request routing and instance placement in the microservice system," *J. Netw. Comput. Appl.*, vol. 147, Dec. 2019, Art. no. 102441.

[5] M. Alam, J. Rufino, J. Ferreira, S. H. Ahmed, N. Shah, and Y. Chen, "Orchestration of microservices for IoT using docker and edge computing," *IEEE Commum. Mag.*, vol. 56, no. 9, pp. 118–123, Sep. 2018.

[6] 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.

[7] N. C. Coulson, S. Sotiriadis, and N. Bessis, "Adaptive microservice scaling for elastic applications," *IEEE Internet Things J.*, vol. 7, no. 5, pp. 4195–4202, May 2020.

[8] 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 ICDCS*, 2019, pp. 80–90.

[9] A. Panda, M. Sagiv, and S. Shenker, "Verification in the age of microservices," in *Proc. ACM HotOS*, 2017, pp. 30–36.

[10] D. Wu, Z. Yang, H. Wang, B. Yang, and R. Wang, "UCRA: A user-centric context-aware resource allocation for network slicing," in *Proc. IEEE ICNC*, 2020, pp. 808–814.

[11] W. Guo, W. Tian, Y. Ye, L. Xu, and K. Wu, "Cloud resource scheduling with deep reinforcement learning and imitation learning," *IEEE Internet Things J.*, vol. 8, no. 5, pp. 3576–3586, Mar. 2021.

[12] T. Chen and R. Bahsoon, "Self-adaptive trade-off decision making for autoscaling cloud-based services," *IEEE Trans. Service Comput.*, vol. 10, no. 4, pp. 618–632, Jul./Aug. 2017.

[13] G. Qu, H. Wu, R. Li, and P. Jiao, "DMRO: A deep meta reinforcement learning-based task offloading framework for edge-cloud computing," *IEEE Trans. Netw. Service Manag.*, vol. 18, no. 3, pp. 3448–3459, Sep. 2021.

[14] C. Zhan, H. Hu, X. Sui, Z. Liu, J. Wang, and H. Wang, "Joint resource allocation and 3D aerial trajectory design for video streaming in UAV communication systems," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 31, no. 8, pp. 3227–3241, Aug. 2021.

[15] S. Guo, Y. Dai, S. Xu, X. Qiu, and F. Qi, "Trusted cloud-edge network resource management: DRL-driven service function chain orchestration for IoT," *IEEE Internet Things J.*, vol. 7, no. 7, pp. 6010–6022, Jul. 2020.

[16] Y. Liu, H. Lu, X. Li, Y. Zhang, L. Xi, and D. Zhao, "Dynamic service function chain orchestration for NFV/MEC-enabled IoT networks: A deep reinforcement learning approach," *IEEE Internet Things J.*, vol. 8, no. 9, pp. 7450–7465, May 2021.

[17] Y. Niu, F. Liu, and Z. Li, "Load balancing across microservices," in *Proc. INFOCOM*, 2018, pp. 1–9.

[18] S. G. Kulkarni *et al.*, "NFVnice: Dynamic backpressure and scheduling for NFV service chains," in *Proc. ACM SIGCOMM*, 2017, pp. 71–84.

[19] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, "Autonomic vertical elasticity of Docker containers with ELASTICDOCKER," in *Proc. IEEE Cloud*, 2017, pp. 472–479.

[20] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," *J. Grid Comput.*, vol. 12, no. 4, pp. 559–592, 2014.

[21] H. M. Fard, R. Prodan, and F. Wolf, "Dynamic multi-objective scheduling of microservices in the cloud," in *Proc. IEEE/ACM UCC*, 2020, pp. 386–393.

[22] M. Lin, J. Xi, W. Bai, and J. Wu, "Ant colony algorithm for multi-objective optimization of container-based microservice scheduling in cloud," *IEEE Access*, vol. 7, pp. 83088–83100, 2019.

[23] A. Samanta, L. Jiao, M. Mühlhäuser, and L. Wang, "Incentivizing microservices for online resource sharing in edge clouds," in *Proc. IEEE ICDCS*, 2019, pp. 420–430.

[24] O. Sener and V. Koltun, "Multi-task learning as multi-objective optimization," in *Proc. NeurIPS*, 2018, pp. 527–538.

[25] M. Dai, Z. Su, R. Li, and S. Yu, "A Software-defined-networking-enabled approach for edge-cloud computing in the Internet of Things," *IEEE Netw.*, vol. 35, no. 5, pp. 66–73, Sep./Oct. 2021.

[26] S. Wang, Y. Guo, N. Zhang, P. Yang, A. Zhou, and X. S. Shen, "Delay-aware microservice coordination in mobile edge computing: A reinforcement learning approach," *IEEE Trans. Mobile Comput.*, vol. 20, no. 3, pp. 939–951, Mar. 2021.

[27] Y. Cui, L. Du, H. Wang, D. Wu, and R. Wang, "Reinforcement learning for joint optimization of communication and computation in vehicular networks," *IEEE Trans. Veh. Technol.*, vol. 70, no. 12, pp. 13062–13072, Dec. 2021.

[28] H. A. Shah and L. Zhao, "Multiagent deep-reinforcement-learning-based virtual resource allocation through network function virtualization in Internet of Things," *IEEE Internet Things J.*, vol. 8, no. 5, pp. 3410–3421, Mar. 2021.

[29] J. S. P. Roig, D. M. Gutierrezestevez, and D. Gunduz, "Management and orchestration of virtual network functions via deep reinforcement learning," *IEEE J. Sel. Areas Commun.*, vol. 38, no. 2, pp. 304–317, Feb. 2020.

[30] J. Tao, T. Han, and R. Li, "Deep-reinforcement-learning-based intrusion detection in aerial computing networks," *IEEE Netw.*, vol. 35, no. 4, pp. 66–72, Jul./Aug. 2021.

[31] J. A. Aroca, A. Chatzipapas, A. F. Anta, and V. Mancuso, "A measurement-based characterization of the energy consumption in data center servers," *IEEE J. Sel. Areas Commun.*, vol. 33, no. 12, pp. 2863–2877, Dec. 2015.

[32] M. Stillwell, D. Schanzenbach, F. Vivien, and H. Casanova, "Resource allocation algorithms for virtualized service hosting platforms," *J. Parallel Distrib. Comput.*, vol. 70, no. 9, pp. 962–974, 2010.

[33] W. Dawoud, I. Takouna, and C. Meinel, "Elastic virtual machine for fine-grained cloud resource provisioning," in *Global Trends in Computing and Communication Systems*. Heidelberg, Germany: Springer, 2012, pp. 11–25.

[34] J. Hwang, K. Ramakrishnan, and T. Wood, "NetVM: High performance and flexible networking using virtualization on commodity platforms," in *Proc. USENIX NSDI*, 2014, pp. 445–458.

[35] B. Ruan, H. Huang, S. Wu, and H. Jin, "A performance study of containers in cloud environment," in *Proc. Asia-Pacific Services Comput. Conf.*, 2016, pp. 343–356.

[36] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "Deep reinforcement learning: A brief survey," *IEEE Signal Process. Mag.*, vol. 34, no. 6, pp. 26–38, Nov. 2017.

[37] M. Andrychowicz *et al.*, "Hindsight experience replay," in *Proc. NeurIPS*, 2017, pp. 5055–5065.

[38] Y. Li, "Deep reinforcement learning: An overview," 2017, *arXiv:1701.07274*.

[39] O. Sigaud and F. Stulp, "Policy search in continuous action domains: An overview," *Neural Netw.*, vol. 113, pp. 28–40, May 2019.

[40] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[41] T. P. Lillicrap *et al.*, "Continuous control with deep reinforcement learning," in *Proc. ICLR*, 2016, pp. 1–14.

[42] V. Mnih *et al.*, "Asynchronous methods for deep reinforcement learning," in *Proc. PMLR*, vol. 48, Jun. 2016, pp. 1928–1937.

[43] G. A. Rummery and M. Niranjan, "On-line Q-learning using connectionist systems," Dept. Eng., Univ. Cambridge, Cambridge, U.K., Rep. TR 166, 1994.

[44] R. Yang, X. Sun, and K. Narasimhan, "A generalized algorithm for multi-objective reinforcement learning and policy adaptation," in *Proc. NeurIPS*, 2019, pp. 14636–14647.

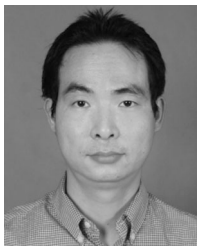
[45] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Mach. Learn.*, vol. 8, nos. 3–4, pp. 229–256, 1992.

[46] A. Abels, D. Roijers, T. Lenaerts, A. Nowé, and D. Steckelmacher, "Dynamic weights in multi-objective deep reinforcement learning," in *Proc. PMLR*, 2019, pp. 11–20.

[47] K. Van Moffaert, M. M. Drugan, and A. Nowé, "Scalarized multi-objective reinforcement learning: Novel design techniques," in *Proc. IEEE Symp. Adapt. Dyn. Program. Reinforcement Learn. (ADPRL)*, 2013, pp. 191–199.
[48] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Trans. Evol. Comput.*, vol. 6, no. 2, pp. 182–197, Apr. 2002.
[49] A. J. Nebro, J. J. Durillo, J. Garcia-Nieto, C. C. Coello, F. Luna, and E. Alba, "SMPSO: A new PSO-based metaheuristic for multi-objective optimization," in *Proc. IEEE Symp. Comput. Intell. Multi-Criteria Decis. Making*, 2009, pp. 66–73.
[50] "MOEA Framework: A Free and Open Source Java Framework for Multiobjective Optimization." [Online]. Available: http://moeaframework.org/ (accessed Feb. 1, 2022).
[51] D. Bhamare, M. Samaka, A. Erbad, R. Jain, L. Gupta, and H. A. Chan, "Multi-objective scheduling of micro-services for optimal service function chains," in *Proc. IEEE ICC*, 2017, pp. 1–6.
[52] A. R. Sampaio, J. Rubin, I. Beschastnikh, and N. S. Rosa, "Improving microservice-based applications with runtime placement adaptation," *J. Internet Services Appl.*, vol. 10, no. 1, pp. 1–30, 2019.
[53] F. Rossi, M. Nardelli, and V. Cardellini, "Horizontal and vertical scaling of container-based applications using reinforcement learning," in *Proc. IEEE Cloud*, 2019, pp. 329–338.

Yinbo Yu (Member, IEEE) received the B.E. and Ph.D. degrees in electronic information engineering from Wuhan University, Wuhan, China, in 2014 and 2020, respectively.

He was a visiting Ph.D. student with Northwestern University, Evanston, IL, USA, from 2017 to 2019. He is currently an Associate Professor with the Research & Development Institute of Northwestern Polytechnical University, Shenzhen, China, and also with the School of Cybersecurity, Northwestern Polytechnical University, Xi'an, China. His research interests span on the area of networking, IoT security, formal verification, and AI security.



Jiajia Liu (Senior Member, IEEE) received the Ph.D. degree in applied information science from Tohoku University, Sendai, Japan, in 2012.

He is a Full Professor (Vice Dean) with the School of Cybersecurity, Northwestern Polytechnical University, Xi'an, China, where he has been the Director of Shaanxi Provincial Engineering Laboratory of Cyber Security since 2021, and the Director of Xi'an Unmanned System Security and Intelligent Communications ISTC Center since 2020. He was a Full Professor with the School of Cyber Engineering, Xidian University, Xi'an, from 2013 to 2018. He has published more than 200 peer-reviewed papers in many high-quality publications, including prestigious IEEE journals and conferences. His research interests cover a wide range of areas including intelligent and connected vehicles, mobile/edge/cloud computing and storage, Internet of Things security, wireless and mobile ad hoc networks, and space–air–ground-integrated networks.

Prof. Liu received the IEEE ComSoc Best YP (Young Professional) Award in Academia in 2020, the IEEE VTS Early Career Award in 2019, the IEEE ComSoc Asia–Pacific Outstanding Young Researcher Award in 2017, the IEEE ComSoc Asia–Pacific Outstanding Paper Award in 2019, the Niwa Yasujiro Outstanding Paper Award in 2012, the Best Paper Awards from many international conferences, including IEEE flagship events, such as IEEE GLOBECOM in 2016 and 2019, IEEE WCNC in 2012 and 2014, IEEE WiMob in 2019, IEEE IC-NIDC in 2018, and AICON in 2019. He was also a recipient of the Tohoku University President Award in 2013. He has been actively joining the society activities, such as serving as an Associate Editor for IEEE TRANSACTIONS ON WIRELESS COMMUNICATIONS since May 2018, IEEE TRANSACTIONS ON COMMUNICATIONS since September 2020, IEEE TRANSACTIONS ON COMPUTERS from October 2015 to June 2017, and IEEE TRANSACTIONS ON VEHICULAR TECHNOLOGY from January 2016 to January 2021, an Editor for IEEE NETWORK since July 2015, and IEEE TRANSACTIONS ON COGNITIVE COMMUNICATIONS AND NETWORKING since January 2019, a Guest Editor of top ranking international journals, such as IEEE TRANSACTIONS ON EMERGING TOPICS IN COMPUTING, *IEEE Network Magazine*, and IEEE INTERNET OF THINGS JOURNAL, and serving for technical program committees of numerous international conferences, such as the Leading Symposium Co-Chair for AHSN symposium for GLOBECOM 2017, CRN symposium for ICC 2018, and AHSN symposium for ICC 2019. He is the Chair of IEEE IOT-AHSN TC, and a Distinguished Lecturer of IEEE Communications Society and Vehicular Technology Society.

Jing Fang received the B.S. degree in communication engineering from Jianghan University, Wuhan, China, in 2014, and the M.S. degree in electronic information engineering from Wuhan University, Wuhan, in 2016, where she is currently pursuing the Ph.D. degree with the National Engineering Research Center for Multimedia Software, School of Computer.

Her research interests include deep learning, deep reinforcement learning, and image superresolution.