DRPC: Distributed Reinforcement Learning Approach for Scalable Resource Provisioning in Container-based Clusters

Haoyu Bai, Minxian Xu, *Senior Member, IEEE,* Kejiang Ye, *Senior Member, IEEE,* Rajkumar Buyya, *Fellow, IEEE,* Chengzhong Xu, *Fellow, IEEE*

Abstract—Microservices have transformed monolithic applications into lightweight, self-contained, and isolated application components, establishing themselves as a dominant paradigm for application development and deployment in public clouds such as Google and Alibaba. Autoscaling emerges as an efficient strategy for managing resources allocated to microservices' replicas. However, the dynamic and intricate dependencies within microservice chains present challenges to the effective management of scaled microservices. Additionally, the centralized autoscaling approach can encounter scalability issues, especially in the management of large-scale microservice-based clusters. To address these challenges and enhance scalability, we propose an innovative distributed resource provisioning approach for microservices based on the Twin Delayed Deep Deterministic Policy Gradient algorithm. This approach enables effective autoscaling decisions and decentralizes responsibilities from a central node to distributed nodes. Comparative results with state-of-the-art approaches, obtained from a realistic testbed and traces, indicate that our approach reduces the average response time by 15% and the number of failed requests by 24%, validating improved scalability as the number of requests increases.

Index Terms—Cloud computing, Kubernetes, microservice, reinforcement learning, distributed resources management

I. INTRODUCTION

Microservice architecture has emerged as a transformative approach in the field of software design and development. It is characterized by its modular and decentralized structure, where complex applications are broken down into smaller, independently deployable services [1] [2]. Each microservice focuses on a specific business capability, allowing teams to work on distinct components simultaneously and enabling rapid development, deployment, and scalability. This architecture promotes flexibility, resilience, and maintainability by minimizing the impact of changes within one service on the overall system [3]. As microservices communicate through well-defined APIs, they facilitate seamless integration and support heterogeneous technology stacks. The cloud service providers, such as Amazon, Google, Microsoft, and Alibaba, have embraced microservice to develop and deploy their applications in cloud computing environment [4].

The rapid growth of the microservices paradigm has introduced challenges in resource management [5]. As the number of microservices increases within a system, ensuring optimal resource allocation and utilization becomes complex. Microservices autoscaling has emerged as a viable solution to address these challenges [6]. Autoscaling involves dynamically adjusting the number of instances or resources allocated to microservices based on real-time demand, ensuring efficient resource utilization while maintaining desired performance levels [7]. This technique encompasses both horizontal scaling, which involves adding or removing replicas of a service, and vertical scaling, which involves adjusting the resources allocated to each instance. Autoscaling mechanisms utilize various metrics and triggers, such as CPU usage, memory consumption, and request rates, to make decisions about scaling actions [8]. Implementing effective autoscaling strategies enhances system responsiveness, minimizes operational costs, and optimizes resource allocation in dynamic and unpredictable environments.

Moreover, the intricate interdependencies among microservices often give rise to critical paths and key nodes that significantly impact performance. Efficiently implementing autoscaling for microservices is confronted with its own set of challenges and intricacies. The dynamic nature of these dependencies necessitates a profound understanding of the application's behavior, workload patterns, and their implications for system performance. Identifying the optimal scaling strategy, striking a balance in resource allocation across interconnected services while maintaining overall system stability, proves to be a non-trivial undertaking [9]. Additionally, the real-time nature of scaling decisions and the imperative to minimize disruptions to service quality and user experience further contribute to the complexity.

Reinforcement learning (RL) [10] has the potential to optimize the scaling of microservices, taking into account their intricate interdependencies. Nevertheless, the majority of RL solutions adopt a centralized decision-making approach, which poses scalability challenges. This centralized structure encounters difficulties with the increasing complexity and interdependencies of microservices, resulting in adverse effects

on scalability, response times, and reliability. Centralized management approaches such as Borg [11] permit users to over-provision resources while assigning jobs to machines, leading to resource wastage and consequently diminishing overall performance in resource-limited environments. In addition, centralized approaches can compromise system performance as microservices scale up, given the continuous communication between huge amount of services and the central node may introduce bottlenecks and latency.

In this paper, we explore novel approaches to address these limitations and propose a distributed reinforcement learning framework for microservices automatic scaling. This framework aims to enhance scalability, improve decision-making efficiency, and ensure the adaptability of microservices scaling strategies to dynamic and evolving environments. By decentralizing the decision-making process, we aim to mitigate the challenges associated with centralized approaches and unlock the full potential of RL in optimizing microservices resource allocation and scalability.

The main **contributions** of this paper can be summarized as follows:

- We design a distributed reinforcement learning framework for resource provisioning for container-based autoscaling (*DRPC*). This framework facilitates precise modelling of system resources and their scalable allocation to address the dynamic demands of microservices.
- We propose a method for selecting optimization strategies based on reinforcement learning and a distributed algorithm that incorporates domain knowledge through deep imitation learning. This approach facilitates efficient and adaptive decision-making, optimizing the choice of scaling strategies across clusters of microservices.
- We perform comprehensive testing and validation of our proposed model and policies, utilizing real-world traces and a dedicated testing platform. These experiments are conducted to thoroughly assess the effectiveness and performance of our distributed reinforcement learning framework for microservices autoscaling.

II. RELATED WORK

In this section, we discuss the current autoscaling methods for microservices, categorizing them into three groups according to their key mechanisms: threshold-based and heuristic approaches, machine learning (ML) based approaches, and deep learning (DL) based approaches.

A. Threshold-based and Heuristic Autoscaling

The threshold-based heuristic approach for microservice resource allocation relies on predefined rules, scaling resources up when utilization surpasses a predetermined threshold (e.g., 75%). This method proves efficient in scenarios with abundant resources and stable request patterns. He et al. [12] leveraged both genetic and heuristic algorithms to determine the optimized microservice deployment location within an edge-cloud environment. Horizontal pod auto-scaler (HPA) [13] is a scaling technique adopted in Kubernetes, primarily focusing on horizontal scaling. It is capable of dynamically adjusting

the number of replicas based on resource variations, such as CPU and memory, within the existing servers. The primary objective of HPA is to determine the number of replicas to be added or removed based on system running status. Kannan et al. [14] conceptualized multi-stage tasks using a Directed Acyclic Graph (DAG). By leveraging the DAG, they could predict the task's completion duration. Their method continually adapts thresholds and algorithms for each microservice. As a result, every microservice can manage its loads autonomously, incurring minimal communication expenses.

B. Machine Learning based Autoscaling

The ML-based autoscaling for microservices dynamically adjusts resources through ML algorithms that analyze historical data and workload patterns. This method optimizes resource utilization and ensures consistent performance by scaling services as required. Hou et al. [15] proposed an autoscaling approach that emphasizes both power and latency considerations for resource provisioning at micro and macro levels. By employing decision trees and a tagging methodology, they were able to expedite the resource-matching process. Yu et al. [16] introduced a framework called Microscaler that utilizes a service mesh to monitor resource usage patterns. By integrating online learning and heuristic methods, the framework achieves near-optimal solutions to satisfy resource needs and quality of service (QoS) requirements. Liu et al. [17] undertook a detailed analysis of bottlenecks in prevalent microservice applications and incorporated various machine learning models to facilitate resource scheduling. Meanwhile, Gan et al. [18] harnessed predictive methodologies to detect QoS violations. They leveraged a combination of machine learning models and expansive historical data to pinpoint the microservices responsible for these QoS infractions, enabling better resource reallocation to mitigate QoS degradation.

C. Deep Learning based Autoscaling

The emphasis has shifted towards employing DL for microservice autoscaling, utilizing neural networks for decision-making and pattern analysis. This approach dynamically adjusts resource allocation based on real-time demand and optimizes provisioning by learning from historical and current data, thereby minimizing resource wastage. Autopilot [19] accumulates historical server data and then utilizes two scheduling techniques: the sliding window algorithm based on past data, and the meta-algorithm inspired by RL. CoScal [20] classifies resource usage into four representative levels. It then applies an approximated Q-learning algorithm to these segments, establishing an estimated policy. This policy encompasses horizontal and vertical scaling, as well as brownout [24] capabilities that can dynamically activate and deactivate application component. With gated recurrent unit (GRU), CoScal predicts upcoming workloads and refers to the trained lookup table to determine the suitable scaling action for each pod, ensuring optimization across all pods. FIRM [21] employs a systematic approach to allocating resources in cloud systems. It pinpoints the crucial path in the microservice dependency by carefully examining the connections between components.

TABLE 1: Comparison of related work

Approach	Scaling Techniques			Workload Prediction		Scheduling Mechanism			Decision-making Pattern	
	Vertical	Horizontal	Brownout	Linear	Non-Linear	Heuristic	Machine Learning	Deep Learning	Distributed	Centralised
He et al. [12]	✓	✓				✓			✓	
Burns et al. [13]		✓				✓				✓
Kannan et al. [14]	✓			✓		✓				✓
Hou et al. [15]	✓			✓			✓			✓
Yu et al. [16]		✓			✓		✓			✓
Liu et al. [17]	✓				✓		✓			✓
Gan et al. [18]	✓				✓		✓			✓
Rzadca et al. [19]	✓	✓		✓				✓		✓
Xu et al. [20]	✓	✓	✓		✓			✓		✓
Qiu et al. [21]	✓							✓		✓
Zhang et al. [22]		✓						✓		✓
Rossi et al. [23]	✓	✓		✓				✓		✓
Our method (DRPC)	✓	✓	✓		✓			✓	✓	

Once this path is identified, FIRM employs a specialized Support Vector Machine (SVM) tailored to operate on both a per-critical-path and per-micro-service-instance basis. This helps in identifying specific microservice instances that require optimization. Zhang et al. [22] introduced a predictive RL algorithm for horizontal container scaling, which combines the Autoregressive Integrated Moving Average (ARIMA) model with a neural network model, ensures the predictability and precision of the scaling procedure. Rossi et al. [23] developed RL-based strategies to manage both horizontal and vertical scaling for containers. This approach enhances system adaptability in the face of fluctuating workloads and hastens the learning phase by harnessing varying levels of environmental knowledge.

D. Critical Analysis

Although the existing representative methods have brought valuable contributions, our proposed method advances the relevant area in several key points. First, unlike HPA and other threshold-based approaches that primarily focus on horizontal scaling, our approach leverages multi-faceted scaling approaches (horizontal, vertical and brownout that can dynamically activate or deactivate optional microservices [24]), ensuring optimal resource allocation even during non-overload-states. Second, compared with the machine learning-based approach, we have applied deep learning to capture the features of workloads and utilized RL to make scaling decisions to achieve more accurate and efficient resource provisioning decisions. The compared differences with the related work are highlighted in Table 1.

Our approach is most similar to CoScal [20] and FIRM [21] based on RL to auto-scale microservices while having significant differences compared with them. In contrast to CoScal's large-grained segmentation of system states, our method employs a deep neural network, offering nuanced decision-making and the capability for simultaneous multiple scaling actions, crucial for sudden load changes. Compared to FIRM, which focuses on optimizing resources of the critical path and nodes, our method ensures comprehensive optimization, with the advantage of supporting both horizontal and vertical scaling, and brownout. Furthermore, our advanced resource allocation framework provides precise solutions, overcoming the scalability limitations inherent in FIRM's centralized reinforcement learning approach, especially under a high volume of requests.

III. MOTIVATION

In this section, we perform motivational experiments with a use case that requests increase quickly to explore the system scalability under a centralized design, such as the centralized database capturing the dynamic change of microservices (FIRM) [21] and the centralized RL-based approach (CoScal) [20]. The investigation focuses on response time as the number of requests significantly increases. For these experiments, we utilize the TrainTicket [25] microservice application comprising approximately 40 services, including user, station, price, and route. The experiments are conducted on two nodes: one for the initial TrainTicket deployment and another for scaling, with a configuration of up to 8 CPU cores and 8 GB memory.

Fig. 1 shows the performance of FIRM and CoScal when the number of requests increases from 200 to 800 per second within a short time (e.g. 1 minute), and we can notice apparent performance fluctuations. For instance, for the FIRM approach, the response time increases from 160 ms to 290 ms when requests increase from 200 to 400 per second. This resulted from the limitation of the centralized design of FIRM in that its central node contains functionalities including data collection, training, inference, and service provisioning. The CoScal based on centralized RL design also suffers from the same issue when the number of requests increases significantly within a short time, CoScal might over-provision into the system to ensure response time but also incurs resource wastage. For instance, several replicas are added to provision resources when the number of requests increases and the response time is also reduced.

In this work, our objective is to provision resources efficiently via an RL-based approach, and utilize a distributed framework to overcome the limitation of centralized design, e.g. system performance bottleneck. A centralized module responsible for data collection and inference can exhibit drawbacks such as unreliability under a container-based environment, slow performance, and inability to adjust system resources asynchronously. The system could also potentially fail to adjust resources if the central nodes experience performance degradation. Hence, we propose a distributed framework utilizing multiple lightweight neural networks on distributed nodes to execute the operations sent from the central node to relieve
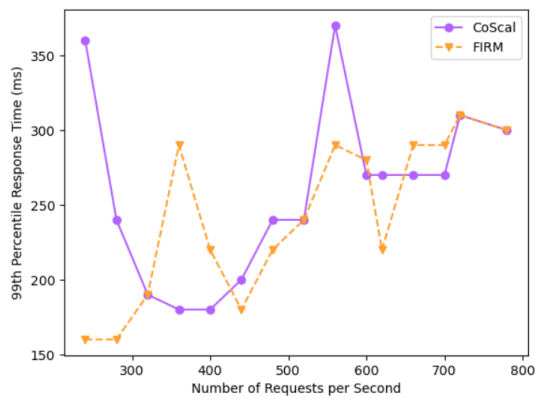
Fig. 1: Response time of centralized approaches when the number of requests increases significantly

the burden of the central node. This approach can potentially hasten resource allocation, and provide a more accurate resource usage prediction of the cloud system's behaviour. Moreover, it could facilitate differential resource adjustment, accommodating services with rapid changes more frequently than others. In the subsequent sections, we will introduce our detailed solution.

IV. SYSTEM MODEL

In this section, we will introduce the system model of DRPC, which adheres to the Monitor-Analyze-Plan-Execute framework over a shared Knowledge pool (MAPE-K), as depicted in Fig. 2. The model is composed of three key components: (1) a *Workload Processor and Predictor* that preprocess the raw workloads and predicts the future workloads, (2) a *Central Teacher Network* that aims to learn the globally optimal policy, and (3) a *Per-Deployment*[1] *Distributed Student Network* responsible for data collection as well as asynchronous for imitation learning.

A. Workload Processor and Predictor

The *Workload Processor* is composed of key components including a *Workload Preprocessor*, a *Load Generator* (e.g. Locust [26]), and a *Historical Database* containing historical workloads. It functions to extract necessary workload data and attributes from the *Historical Database*, preprocesses the dataset via *Workload Preprocessor*, and handles realistic user interactions with the *Load Generator*. It transforms these interactions into requests allocated to the microservice-based cluster. The *Historical Database* stores the historical data derived from realistic traces (e.g. Alibaba traces [27]). Detailed dataset information such as timestamps, machine identifiers and different types of resources are contained in these workloads.

The *Workload Predictor* is a critical component, responsible for forecasting future loads coming into the system. It communicates to the auto-scaler about the required amount

of resources to be scaled-in or scaled-out. The *Workload Predictor* obtains preprocessed workload data from the *Workload Processor* and employs predictive models such as Long Short-Term Memory (LSTM) [28] or Gated Recurrent Unit (GRU) to forecast future workloads. The responsibilities of the *Workload Predictor* include managing the training process of the model, updating the trained model when necessary, and deploying the trained models in their final stages. This module interacts with the Workload Analyzer and Workload Generator, receiving historical system data as input.

B. Central Teacher Network

The use of multi-agent RL often achieves sub-optimal solutions [29], indicating a need for a central agent ("teacher") module that suggests actions to another agent ("student") [30]. This module investigates the global state, formulating an optimal policy function to enhance system performance. The collected data is employed to train the decentralized student network via the technique of imitation learning [31].

Existing performance modelling-based or heuristic-based strategies suffer several limitations, including struggles with model reconstruction and retraining due to their inability to adapt to dynamic system statuses [32]. These strategies require considerable expert knowledge [33], demanding extensive efforts for the interpretation of microservice workloads and infrastructure. Compared with them, RL is suitable for formulating resource provisioning policies due to its capabilities of offering a close feedback loop, exploring scaling actions, and formulating optimal policies independent of erroneous assumptions. This allows for learning directly from actual various workloads and operational conditions, providing deeper insights into how changes in low-level resources influence application performance. We have incorporated two techniques below to achieve our objective:

1) Twin Delayed Deep Deterministic Policy Gradient (TD3) [34] : To overcome the limitation of overestimation of Q value due to the flexible cloud environment, we utilize the TD3 algorithm, an advanced model-free, actor-critic RL framework. Our RL formulation provides two distinct advantages:

- The model-free RL eliminates the need for precise modelling of the complete distribution of states or the environment dynamics (transitions between states). In the event of microservice updates, the simulations of state transitions used in model-based RL become inefficient.
- The actor-critic framework, merging policy-based and value-based methods, is suitable for continuous stochastic environments. This accelerates convergence, reduces variance, and provides a robust and efficient solution for managing the dynamic nature of container-based environments.

2) Retraining Notifier: The Retraining Notifier signals when the system requires retraining. There are instances where the imitation learner might not make efficient decisions, triggering the Retraining Notifier and initiating a retraining state. This is typically initiated by the following scenarios:

- Insufficient exploration by the teacher network: If the teacher network encounters a rarely seen state about

[1]Please note that the term *Per-Deployment* is inherited the naming conventions module of Kubernetes, which consists a set of pods to run an application workload, usually does not need to maintain state.
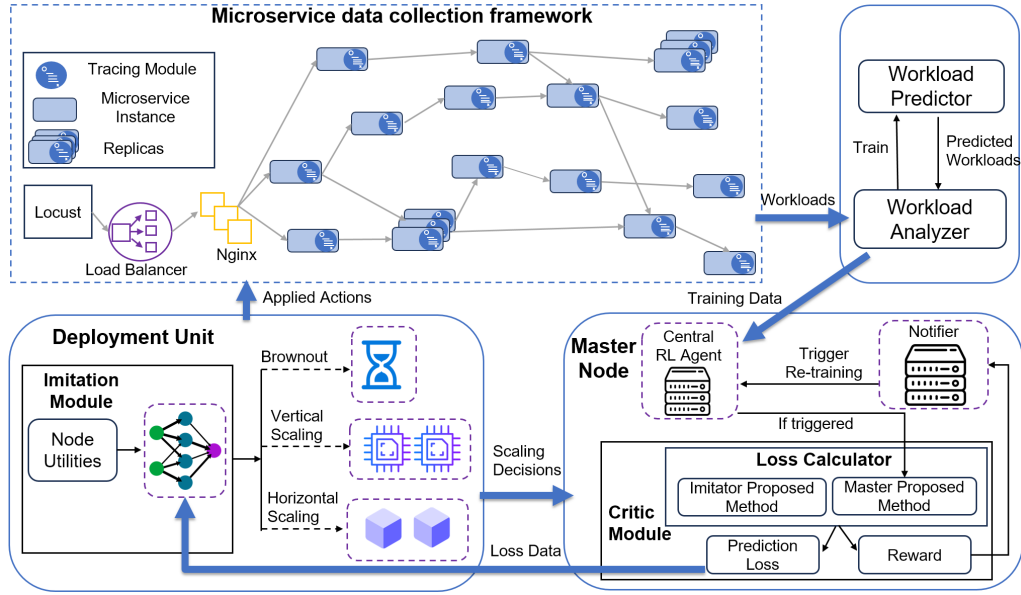
Fig. 2: System Model of DRPC

which it lacks confidence, it might fail to guide the student appropriately. This can potentially result in the cloud service scaler misjudging the system's scaling needs.

- Sub-optimal learning state of the student deployment network: In this case, despite the central network learning an optimal policy function, the distributed deployment network fails to establish a correlation between the optimal policy and the domain information it possesses. This can occur if the information provided by the central teacher network is over-complex, requiring an adjustment in the number of epochs that the distributed deployment net runs to achieve convergence.
- Outdated knowledge held by the student deployment network: For instance, if the majority of users previously searched for tickets, the student net may over-rely on this pattern. If the request composition changes, such as users predominantly purchasing tickets, the deployment of student nets would require retraining.

C. Distributed Student Deployment Network

The Distributed Student Deployment Network module operates in a distributed fashion, enabling the parallel execution of scaling actions for microservices. Rather than maintaining global states, it relies on local information to function. This module consists of two key components: the *Deployment Buffer*, which stores the policies communicated by the teacher node for training the student network, and the *Imitation Learner*, which scales each deployment independently.

1) Deployment Buffer: It maintains the policy transmitted by the Central Module, and stores the state of the deployment when the policy is recorded. This policy-state pairing is utilized to train the Deployment Student Network via imitation learning, which aims to ensure replicas' successful behaviours in a given context.

2) Imitation Learner: The Distributed Student Deployment Net is characterized by its lightweight nature and lower resource requirements compared to the Central Teacher Network. This characteristic facilitates more frequent and adaptable decision-making. The system operates in two distinct modes: learning mode, where it receives guidance from the Central Network, and acting mode, where it autonomously makes decisions based on its state, including factors like resource utilization. This dual-mode configuration enables continuous learning and adaptation, leading to optimized resource allocation and enhanced system performance.

V. DISTRIBUTED RL ALGORITHM FOR SCALING MICROSERVICE

In this section, we introduce the algorithm design of our proposed DRPC approach, which can achieve efficient distributed decisions.

In DRPC, the RL-based resources provisioning is modelled as a Markov Decision Process, it views the state $s_t \in S$ as the current microservices system status and interprets the action $a \in A$ as a scaling operation modifying system status and allocated resources. We use $M = (m^1, m^2, \ldots, m^i)$ to denote the physical machines provisioning resources for the microservices in our system. For each physical machine, say m^i, the amount of resources that can be allocated are denoted as $R^i = (r^{i,1}, r^{i,2}, \ldots, r^{i,j})$, where the j represents the type of resources, such as CPU and memory. Finally, the set of actions that can be performed on each physical machine is denoted as $A^i = (a^{i,1}, a^{i,2}, \ldots, a^{i,k})$. The actions indicate the amount of resources that can be altered on each machine. The supported actions are *Horizontal_scaling*, *CPU_scaling*, *Memory_scaling*, and *Brownout*. A positive or negative sign before an action implies the addition or reduction of resources to a specific machine, respectively. The term 'Brownout' is a binary indicator for whether a brownout can be triggered

---

Algorithm 1: DRPC: System-wide Action Execution

Input : A central module for controlling scaling actions
Output: None

1 Initialize $trainingMode$ as True or False based on the system state;
2 **while** *True* **do**
3   **if** $trainingMode$ **then**
4   // Stage 1: The central module explores the state and chooses actions $actions$ = central module.detection();
5   centralModule.Scale($actions$) based on Alg. 2;
6   **foreach** *deploymentAction in getActionByDeployment(actions)* **do**
7   // Training deployment networks deployment.train(deploymentAction);
8   **end**
9   **else**
10   // Stage 2: Deployments make independent scaling decisions **foreach** *deployment in allDeployments* **do**
11   deployment.Scale() based on Alg. 2;
12   **end**
13   **end**
14   // Check if the retraining notifier is triggered and update $trainingMode$ $trainingMode$ = retrainingNotifier();
15 **end**

---

on the machine. If this value is set to True, the minimum allowable replicas for horizontal scaling would be set to 0. The collective action space for the system, $A = \prod_{i=1}^{I} \prod_{k=1}^{k} A^{i,k}$, is the product of the action spaces for each machine.

A. System-wide Action Execution

As illustrated in Algorithm 1, the execution of the system-wide action plan progresses through two distinct stages: an exploration phase and a distributed provision phase.

In Stage 1, the exploration phase, (lines 3-9), the central teacher node explores the state space, choosing multi-dimensional scaling actions (horizontal, CPU, memory scaling, or maintaining the current state) for each deployment as described in the section V-C. Simultaneously, deployment-level networks initiate training.

In Stage 2, the distributed provision phase, (lines 10-12), decision-making transitions to deployment-level networks once the central node's exploration is deemed sufficient, with the same scaling options. This phase continually updates the teacher network's buffer with action-state pairs, as described in section V-E. The system alternates between these stages based on the retraining notifier (line 14), enabling the central node to re-explore states or delegate decision-making based on its readiness.

Algorithm 2 illustrates the general scaling procedure of DRPC. The algorithm seeks advice from the Scaling Agent to

TABLE 2: Workloads Prediction Accuracy

Predicted length (mins)	5	10	15	20	25
MSE	0.0025	0.0027	0.0033	0.0049	0.0057

obtain Q-values that guide resource adjustments (line 1). This includes CPU usage (lines 2-4), and memory utilization (lines 5-7). Such guidance manifests as actions within the DRPC framework. If the recommended scaling for CPU or memory is significant, adjustments are made proportionally to ensure system stability or enhance efficiency. Adding or removing replicas (lines 9-13) follows a similar process as above, except if a microservice is pre-configured to support brownout, the minimum number of replicas can be set to 0.

B. Workload Processor and Predictor

The connection between different levels of resource usage and workloads is determined using a deep neural network model for the workload analysis, as shown in Fig. 2. To mimic realistic resource utilization, we apply data from Alibaba traces, which contain workload traces from 4000 machines, encompassing eight days of resource usage data. The performance profiling procedure is as follows: we define a scheduling interval of 5 minutes, over which we gradually increase the number of requests, with each test case comprising 200 requests sent to the host. A three-layer Multi-Layer Perceptron (MLP) is applied to the profiling data, thereby efficiently representing this relationship. Consequently, we can translate the host utilization into the number of requests to our system for any given level of utilization. Then, our workload generator produces these requests in accordance with the current user type composition.

We employ multivariate time series forecasting (MTFS), which converts MTFS problem into a supervised learning task. The Gated Recurrent Unit (GRU) [35], a type of recurrent neural network (RNN), is then used as the prediction method, as it has been validated with good performance in [36] for MTFS. The GRU solves the vanishing gradient problem encountered in conventional RNNs through the utilization of gating mechanisms.

Table 2 shows the mean square errors (MSE) of actual utilization versus predicted utilization for Alibaba workloads over different predicted length. With MSE values ranging between 0.002 and 0.006, it shows that the workload prediction algorithm can achieve good performance in resource utilization prediction.

C. Central Module

Conventional RL models such as Sarsa [37] and Q-learning encounter challenges when dealing with infinite or continuous states and actions, a common scenario in microservices characterized by variable CPU and memory usage. To address this issue, we employ the TD3 algorithm, a more sophisticated approach well-suited for continuous, high-dimensional spaces. TD3 utilizes twin critic networks to reduce bias, incorporates delayed policy updates for stable learning, and employs target policy smoothing to prevent overfitting. These attributes render

---

Algorithm 2: DRPC: General Scaling Procedure

Input: Scaling Agent: $Agent$, CPU usage: Cpu_usage, Memory usage: $memory\_usage$, number of replicas: $replicas$, step size for CPU: Cpu_step, step size for memory: $Memory\_step$, Microservice state: $thisMicroservice.Brownout$

Output: Updated Cpu_usage, $memory\_usage$, $replicas$

1 $[Cpu\_Scaling, Memory\_Scaling, Horizontal\_Scaling] \leftarrow Agent.getQvalue()$

2 **if** $|Cpu\_Scaling| > 0.5$ **then**

3    $Cpu\_usage \leftarrow Cpu\_usage + Cpu\_Scaling.Clip(-1,1) \times Cpu\_step$

4 **end**

5 **if** $|Memory\_Scaling| > 0.5$ **then**

6    $memory\_usage \leftarrow memory\_usage + Memory\_Scaling.Clip(-1,1) \times Memory\_step$

7 **end**

8 **if** $|Horizontal\_Scaling| > 0.5$ **then**

9   **if** $replicas + \lceil Horizontal\_Scaling \rceil < 0$ *and* $thisMicroservice.Brownout == False$ **then**

10   exit

11   **else**

12       $replicas \leftarrow replicas + \lceil Horizontal\_Scaling \rceil$

13   **end**

14 **end**

Algorithm 3: DRPC: TD3 Algorithm

Result: TD3 (Twin Delayed Deep Deterministic policy gradient)

1 Initialize actor network π and critic networks $Q_{\theta_1}, Q_{\theta_2}$ with random parameters $\theta, \theta_1, \theta_2$

2 Initialize target networks π' and Q' with weights $\pi' \leftarrow \pi, Q'_{\theta_{1'}} \leftarrow Q_{\theta_1}, Q'_{\theta_{2'}} \leftarrow Q_{\theta_2}$

3 Initialize replay buffer R

4 **for** *episode = 1 to M* **do**

5   Observe state s

6   **for** *t = 1 to T* **do**

7   Select action $a = \text{clip}(\pi(s) + \epsilon, a_{High}, a_{Low})$ where $\epsilon \sim \mathcal{N}$ and execute it

8   Observe next state s', reward r and done signal d to end episode

9   Store (s, a, r, s', d) in R

10   **if** *it is time to update* **then**

11   Sample a batch B of transitions (s, a, r, s', d) from R

12       $a' \leftarrow \text{clip}(\pi'(s') + \text{clip}(\epsilon, -c, c), a_{High}, a_{Low})$

13       $y \leftarrow r + \gamma(1-d) \min_{i=1,2} Q'_{\theta_{i'}}(s', a')$

14   Update Q_{θ_i} by minimizing the loss: $L(\theta_i) = \frac{1}{|B|} \sum_{(s,a,r,s',d)\in B} (Q_{\theta_i}(s,a) - y)^2$

15   **if** $t \mod policy\_update == 0$ **then**

16   Update π by one step gradient ascent using the loss: $L(\theta) = \frac{1}{|B|} \sum_{s\in B} -Q_{\theta_1}(s, \pi(s))$

17   Soft update the target networks: $\theta_{i'} \leftarrow \rho\theta_{i'} + (1-\rho)\theta_i$ (for both actor and critic)

18   **end**

19   **end**

20     $s \leftarrow s'$

21   **end**

22 **end**

TD3 robust and efficient, making it an ideal choice for optimizing autoscaling in dynamic microservices environments.

1) TD3 modelling: Our dual objectives are improving the Quality of Service (QoS) and maximizing the utilization of physical machines. Our reward model, encompassing response time $R_{qos}(rt)$ and resource utilization $R_{util}(u)$ are accordingly formulated in Equations (1) and (2):

$$R_{qos}(rt) = \begin{cases} e^{-\left(\frac{rt - RT_{max}}{RT_{max}}\right)^2} & , rt > RT_{max} \\ 1 & , rt \leq RT_{max}, \end{cases} \quad (1)$$

where the maximum tolerant latency, RT_{max}, is pre-defined into the QoS reward function. Normal system operation rewards 1, while performance that exceeds RT_{max} is penalized, gradually approaching 0, thus discouraging SLO violation.

Resource utilization $R_{util}(u)$, measured using a devised model shown in Equation (2),

$$R_{util}(u) = \begin{cases} \frac{\sum_{k=1}^{K}\sum_{r=1}^{R}(U_{r,k}^{pred} - u_{r,k})^3}{K} + 1 & , u_{rk} \leq U_{r,k}^{pred} \\ \frac{\sum_{k=1}^{K}\sum_{r=1}^{R}(u_{r,k} - U_{r,k}^{pred})^3}{K} + 1 & , u_{r,k} > U_{r,k}^{pred}, \end{cases} \quad (2)$$

which guides the system towards resource conservation. Here, K and R represent the kth physical machine and rth resource type residual on the kth physical machine, respectively, for example, the $u_{1,1}$ indicate the 1st adjustable resources on the 1st machine. $U_{r,k}^{pred}$, the predefined utility,

represents the ideal utility for a certain resource type r on machine k. Proximity to this ideal utilization is rewarded; under-provisioning or wastage is discouraged.

The final reward value as shown in Equation (3) is a combination of both response time and resource utilization, The objective is:

$$r(s_t, a_t) = \frac{R_{qos}(rt)}{R_{util}(u)}. \quad (3)$$

The final objective is to decrease the response time while keeping the system running in a stable state U^{pred} as shown in Equation (4).

$$\min_{u \in Total\_Resources} |U^{pred} - R_{util}(u)| \wedge \min_{rt \in RT} R_{qos}(rt). \quad (4)$$

2) TD3 Implementation Details: As shown in Algorithm 3, TD3 begins by initializing the actor and critic networks, as well as the replay buffer (lines 1-3). It then selects an action from the actor network, executes the action, and observes the subsequent state, reward, and episode termination status (lines 4-8). This tuple (s, a, r, s', d) is stored in the replay buffer (line 9). Once enough data has accumulated and it is time for an update, TD3 samples a batch of transitions from the buffer, which includes data from both the central and distributed modules (lines 10-11). In certain instances, scale-out may considerably diminish the SLO violation, while scale-in, by circumventing the communication time, could realize superior response times after several scaling operations. The robust reward resulting from scale-out might engender a fragile, or erroneous, "sharp peak," potentially obscuring our model's capacity to investigate the long-term impact of Scale-in. Furthermore, due to the dynamic nature of cloud environments, where internet connections fluctuate, some reallocation actions may reduce QoS violations more effectively than anticipated. Ultimately, this can lead to an overestimation of Q values. To alleviate this, a clipped (limits values to a range) colour black noise, between bound $-c$ and c, ϵ is incorporated into the target policy $\pi'(s')$ to get a final policy a' and that policy is ensured to lie within a_{Low} and a_{High} as shown in Equation (5) (line 12).

$$a' \leftarrow \text{clip}(\pi'(s') + \text{clip}(\epsilon, -c, c), a_{High}, a_{Low}), \quad (5)$$

where both Q-functions employ a singular target. Moreover, as microservices are deployed in a distributed manner and communicate via cable, certain scaling actions may lead to a significant reduction in communication time due to internet fluctuations, ultimately generating a substantial reward. To circumvent the overestimation of rewards, Clipped double-Q learning is employed. Both Q-functions utilize a singular target $y(r, s', d)$, calculated by summing the immediate reward r and the minimum value obtained from the two Q-functions $Q'_{\theta_{i'}}(s', a')$, multiplied by the discount factor γ. If the subsequent state is a terminal state ($d = 1$), no future reward is considered (line 13) as shown in Equation (6):

$$y(r, s', d) \leftarrow r + \gamma(1 - d) \min_{i=1,2} Q'_{\theta_{i'}}(s', a'). \quad (6)$$

The loss indicated as $L(\theta_i)$ in line 14, for the i_{th} critic network, is calculated as the average mean of the sum of squared differences between the target Q-value y, and the critic network predicted Q-value $Q_{\theta_i}(s, a)$, across a mini-batch experiences B containing a set of (s, a, r, s', d) experience. The number of such experiences is denoted by $|B|$ as formulated in Equation (7):

$$L(\theta_i) = \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} (Q_{\theta_i}(s, a) - y)^2. \quad (7)$$

To minimize the variability in microservice communication and to learn a more stable Q function, the policy will only be updated by one step of gradient ascent once every *policy_update* (line 15) times using the loss function (line 16) with Equation (8):

$$L(\theta) = \frac{1}{|B|} \sum_{s \in B} -Q_{\theta_1}(s, \pi(s)). \quad (8)$$

Finally, the target network will be updated (lines 17-20).

D. Deployment Unit with Imitation Learning

Imitation learning involves training an agent to perform tasks by emulating the actions of an expert. This approach has gained popularity due to its capability to teach complex behaviors without the need for explicit programming or heavy reliance on reward signals, which is common in traditional RL. In our scenario, the deployment unit utilizes its information to imitate the resource adjustment actions of the Central module. This objective, as shown in Eq. (9), is accomplished by minimizing the mean square error between the Q-values $Q_{\text{Ctr}}(s)$ generated by the central network and those $Q_{\text{Dtr}}(s)$ produced by the distributed network under the same state s.

$$\min (Q_{\text{Ctr}}(s) - Q_{\text{Dtr}}(s))^2 \quad (9)$$

When the retraining notifier signals deployment readiness, scaling actions, guided by human knowledge-driven frequency, are executed simultaneously and asynchronously. State-action-reward pairs are then gathered, combined with data from other deployments, and utilized to fine-tune the policy of the central network.

E. Retraining Notifier

When the RL-based model needs to be re-trained based on the conditions as introduced in Section IV.B, the *Retraining Notifier* as shown in Algorithm 4 informs the teacher network to repeat the training phase whenever the student network faces insufficient information or the average reward from recent actions falls below a predetermined threshold. It initializes an array, $rewardHistory$, to record the last npr rewards and the iterator number $iter$; continuously, the algorithm calculates the current action's reward, storing it cyclically in $rewardHistory$ (lines 1-7). If collected rewards are below the defined number npr, retraining will not be triggered due to insufficient data (line 8). However, if the average of $rewardHistory$ drops below the threshold TH, the retraining process will be triggered, indicating suboptimal performance (lines 10-12). Otherwise, the system continues its distributional operations (lines 13-15).

VI. PERFORMANCE EVALUATIONS

To assess the efficacy of DRPC in the autoscaling of microservices, we conduct experiments on a realistic testbed based on Kubernetes. We detail the experimental settings, benchmarks, and a comprehensive analysis of the results. The primary objective is to verify that a distributed asynchronous framework can improve performance, potentially offering potential directions for future research.
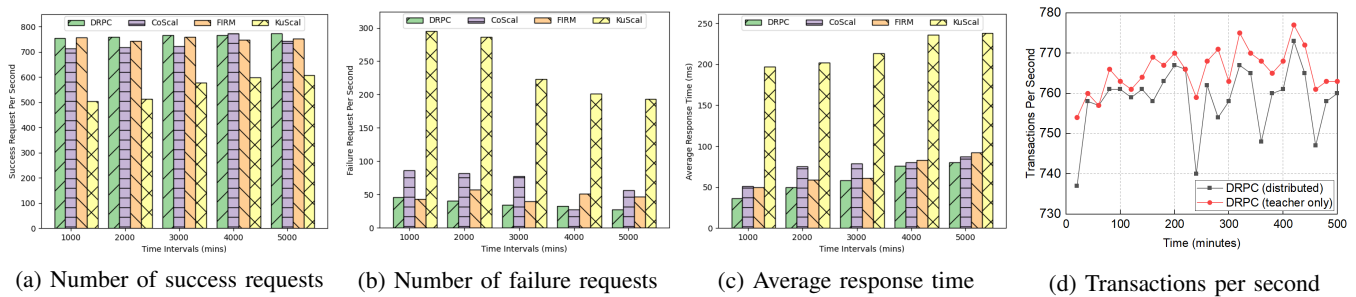
(a) Number of success requests   (b) Number of failure requests   (c) Average response time   (d) Transactions per second

Fig. 3: Performance comparison of KuScal, FIRM, CoScal, and DRPC

Algorithm 4: DRPC: Retraining Notifier

Input : Retraining Threshold TH, number of past rewards to monitor npr, action sets $a_t = \{a_k^i(t)|i \in \{0, 1, \ldots, I\}, k \in \{1, 2, \ldots, K\}\}$

Output: Boolean indicating whether retraining mode is triggered

1   Initialize an array $rewardHistory$ of size npr to store the rewards from past actions;

2   Initialize an Iterator $iter = 0$;

3   **while** *True* **do**

4   Calculate the current action reward $currentReward$ using get_reward(a_t);

5   Store the current reward at the position $iter \% npr$ in $rewardHistory$;

6     $rewardHistory[iter\%npr] = currentReward$;

7     $iter$ ++;

8   **if** $iter < npr$ **then**

9   // Not enough data gathered yet, do not trigger retraining return False;

10   **if** *average(rewardHistory)* $< TH$ **then**

11   // The average reward is below the threshold, trigger retraining return True;

12   **else**

13   // The average reward is above the threshold, do not trigger retraining return False;

14   **end**

15 **end**

A. Experimental Setup

We utilize TrainTicket as a testbed in our motivational example in Section III. It provides functionalities such as ticket purchases, availability checks, cancellations, and news browsing, simulating backend processes like verification code generation, user login checks, and database ticket availability searches. The platform establishes latency-measuring chains using domain name service for service connectivity. Load balancers or service routers based on cloud server tests are employed. Each TrainTicket microservice has its own database, reducing wait times and facilitating scheduling for developers.

We use a cluster with five machines (one master and four workers) for microservice-based cluster. Each machine has 8 CPU and 8GB memory. This prototype system is implemented using a suite of toolkits, including Cgroup v2, Python, TensorFlow, PyTorch, Sklearn, and Locust, which support deep learning and reinforcement learning environments.

We simulate four user types (Normal Query Users, Ticket Buyers, Cancel Ticket Users, and Admin Users) with various ratios in our cluster using the Alibaba work trace, each assigned with a unique ID for tracking. This emulates real-world user behavior dynamics, where service requests fluctuate over time. Data is collected every 200 milliseconds and averaged over 1000-minute intervals in accordance with the existing work [20] [21].

B. Baselines

Several state-of-the-art approaches from both industry and academia have been selected as baselines.

KuScal [13] is a Kubernetes mechanism using horizontal scaling to adjust replicas dynamically. It determines replica quantity based on resource fluctuations, adjusting microservice numbers via continuous real-time resource utilization tracking and predefined CPU utilization thresholds. The threshold is set to be 75% as it is evaluated as the most effective in some existing work surveyed in [38].

CoScal [20] divides resource usage into quarterlies, applies an approximated Q-learning algorithm to devise a policy, and supports horizontal, vertical scaling, and brownout. Using the GRU unit for workload prediction, it forecasts workload, consulting a trained table to guide pod-level scaling actions, and even optimizing non-critical path pods.

FIRM [21] systematically allocates cloud resources by identifying the critical microservice path and using machine learning to target specific instances needing optimization. The process strategically moves from the longest to the shortest chain, prioritizing longer chains to reduce microservice time delays and boost efficiency.

C. Experiment Analysis

Several widely used metrics, including the number of success requests, failure rate and response time have been utilized to evaluate the performance of our proposed approach and the baselines.

1) Computation Efficiency Analysis: Based on the efficient design of our distributed student network, DRPC necessitates considerably fewer computational resources compared to FIRM, with trainable parameters being 643 for our model and
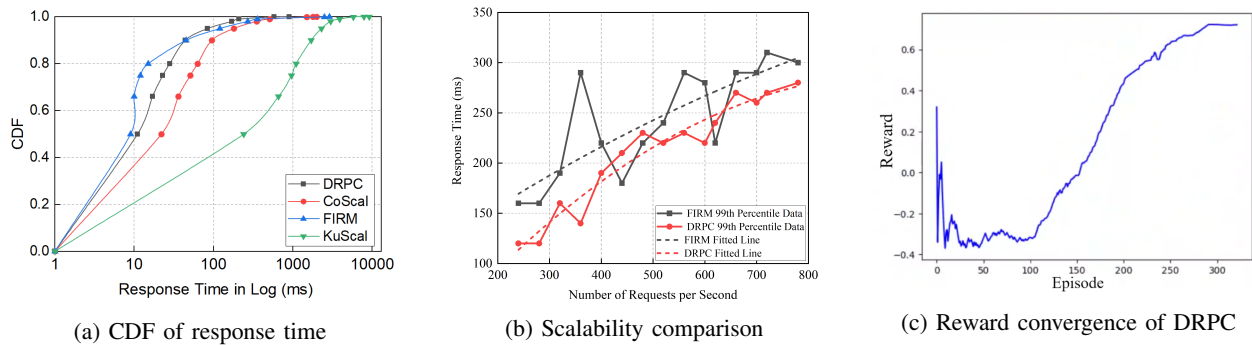
(a) CDF of response time   (b) Scalability comparison   (c) Reward convergence of DRPC

Fig. 4: CDF comparison and reward convergence

11,352 for FIRM. DRPC utilizes only 6×10^{-5} of CPU time to determine the subsequent action, compared to 4×10^{-3} taken by FIRM. Our model also showcases superior efficiency, with a peak CPU utilization of 0.8 as opposed to FIRM's maximum usage of 2.0. This illustrates that our approach is markedly more resource-efficient than FIRM. DRPC incorporates 40 services, and enables simultaneous application of CPU, memory, and horizontal scaling for each deployment, executing up to 120 actions per time step. KuScal ranks second, when CPU utilization for all services surpasses the threshold, and all services undergo horizontal scaling. However, FIRM can only scale one resource type per step, as FIRM's Actor-critic network estimates the Q-value for scaling actions and applies the action with the highest Q-value.

The offline training phase required approximately 10 days to converge, with the majority of this time dedicated to the data collection process within the cluster. However, once the initial training is completed, the model can be fine-tuned within minutes via online learning to adapt to changes in the composition of user requests, which can handle the significant changes in realistic system.

2) Comparison of the Number of Success Requests per Second : Fig. 3a presents the requests per second processed by four distinct algorithms. We partitioned the 5000-minute data into five time periods, with each period representing the average results over the corresponding period. We observed that, across the five time periods, FIRM is capable of processing a higher number of requests compared to our method during the initial periods. However, our method outperformed FIRM over the subsequent four periods. Likewise, CoScal also demonstrated the ability to adapt and improve over time, but it did not outperform our method or FIRM consistently. Furthermore, the difference between our method, CoScal, and FIRM was relatively small during the initial periods, until our method started surpassing them. It can be also reasoned that a scheduling algorithm solely reliant on the longest chain becomes increasingly challenging to handle workloads under significant variances in request rate over time. This might be caused by the longest chain typically signifying more requests at the onset of the scenario when chain variations are minimal. However, as the load changes over time, the chains experiencing high loads, which may not necessarily be the longest chains, become the actual targets for optimization. In

summary, FIRM and Coscal have achieved good performance in handling requests successfully. Compared to FIRM, DRPC improves the success rate of requests by around 2%.

3) Comparison of Failure Rate: As shown in Fig. 3b, DRPC has a failure rate of 4.5%, which is notably lower than FIRM's 5.9%, CoScal's 8.2%, and Kuscal's significantly high rate with 29%. When we evaluate the improvement of DRPC over the other methods, it reduced the failure requests by 24% compared to FIRM, 44% in relation to CoScal, and exhibited an impressive 85% reduction when compared to KuScal. In summary, DRPC demonstrates a significant performance improvement over the benchmarks. This enhancement is attributed to our design's ability to adjust deployments more frequently, thereby reducing the likelihood of failures compared to other methods.

4) Comparison of Response Time: Fig. 3c demonstrates the average response time for the four algorithms under the same configurations. KuScal shows the longest average response time across all five periods, being three to four times longer than others, as its inability to predict forthcoming requests inhibits it from conducting pre-emptive horizontal scaling to accommodate these incoming requests. In contrast, CoScal performs better than KuScal but trails behind FIRM and DRPC. The CoScal model demonstrates an average response time that is generally similar to that of FIRM, except for the last two time periods, where FIRM shows a slight gap. We observed that the DRPC achieves the shortest response time, typically ranging between 40 ms and 70 ms, for most of the time intervals, thereby reducing the average response time by around 15% compared to FIRM and 24% compared to Coscal. Remarkably, it reduces the response time by 72.4% compared to Kuscal.

5) Scalability Analysis: In comparing the transactions per second (TPS) of distributed networks with a centralized teacher network (as shown in Figure 3d), it was observed that asynchronous, frequently updated distributed models consistently surpass the teacher model (centralized) in performance. These models not only emulate the teacher network's patterns but also demonstrate reduced volatility. In Figure 4b, an analysis shows that as request numbers significantly increase, our method outperforms FIRM with 30% improvement in terms of response time in average. Additionally, DRPC's lower quadratic coefficient in fitted line than FIRM confirms its effectiveness, scalability, and stability.

TABLE 4: Response time at different percentile

	50th	66th	75th	80th	90th	95th	99th	99.99th
FIRM	**9**	**10**	**12**	**15**	45	120	360	2900
CoScal	22	36	51	63	95	180	510	1800
DRPC	11	17	23	28	**44**	**84**	**210**	**900**
Kuscal	240	660	960	1100	1700	2300	3900	7900

6) Cumulative Distribution Function (CDF) of Response Time Analysis: We also utilize the CDF in Table 4 to highlight the differences between different approaches, as well as the results in Figure 4a. Until the 90th percentile, the CDF curve of FIRM consistently outperforms DRPC. This difference appears because FIRM allocates requests to specific nodes rather than central ones. Our approach, which releases resources on non-critical chains or nodes, introduces delays on these nodes. Since FIRM does not allocate resources to these nodes, resulting in over-provisioning, it handles certain simple requests more quickly than our method. However, beyond the 90th percentile, our strategy efficiently redistributes the released resources to nodes critical for QoS enhancement. In summary, DRPC significantly reduces tail latency by reallocating resources to nodes crucial for enhancing QoS, outperforming baselines consistently.

7) Reinforcement Learning Convergence Analysis: As illustrated in Figure 4c, the system requires approximately 300 episodes to reach convergence. Beyond this point, the reward per action experiences no significant increase, prompting us to halt the operation. The reward coverage settles at approximately 0.7, as seen from the figure. This indicates that CPU, memory usage, and latency are approaching an optimal configuration by analyzing the reward function. In terms of actual performance, memory utilization is around 75%, while CPU utilization is approximately 60%. The QoS is guaranteed, ensuring that 99% of requests can be handled within 210 ms.

VII. CONCLUSIONS AND FUTURE WORK

In this work, we introduced a novel framework for distributed resource provisioning, named DRPC. By adopting an asynchronous, parallel, and differential approach, DRPC facilitates the global optimal allocation of resources. This accommodates the dynamic nature of microservice-based clusters while ensuring QoS. Notably, DRPC incorporates DL methodologies for workload prediction, achieving a higher level of accuracy compared to conventional gradient-based methods. Additionally, it leverages a distributed RL algorithm to make informed decisions on scaling strategies, effectively managing the infinite action-states space associated with microservices. The results, based on realistic testing and comparisons with state-of-the-art algorithms, demonstrate that DRPC outperforms the baselines in terms of successful request rate and average response time, particularly under significantly increased requests.

The limitation of this approach is that it increases network usage due to the essential communication of distributed RL. Future research will focus on automating the asynchronous updating process, which currently requires manual setting of timing intervals for specific microservices, to improve system efficiency. In addition, we would like to explore anomaly-aware workloads management under container-based environment, and incorporate our approach into large-scale and production environment (e.g. Alibaba Cloud) with further validations.

SOFTWARE AVAILABILITY

The codes have been open-sourced to https://github.com/vincent-haoy/DRPC for research usage.

REFERENCES

[1] S. Newman, *Building microservices: Designing fine-grained systems.* O'Reilly, 2021.
[2] G. Quattrocchi, D. Cocco, S. Staffa, A. Margara, and G. Cugola, "Cromlech: Semi-automated monolith decomposition into microservices," *IEEE Transactions on Services Computing*, vol. 17, no. 2, pp. 466–481, 2024.
[3] O. M. A. Khan and A. Chandaka, *Developing Microservices Architecture on Microsoft Azure with Open Source Technologies.* Microsoft Press, 2021.
[4] M. Xu, L. Yang, Y. Wang, C. Gao, L. Wen, G. Xu, L. Zhang, K. Ye, and C. Xu, "Practice of alibaba cloud on elastic resource provisioning for large-scale microservices cluster," *Software: Practice and Experience*, vol. 54, no. 1, pp. 39–57, 2024.
[5] S. Pallewatta, V. Kostakos, and R. Buyya, "Placement of microservices-based iot applications in fog computing: A taxonomy and future directions," *ACM Computing Surveys*, 2023.
[6] S. N. A. Jawaddi, M. H. Johari, and A. Ismail, "A review of microservices autoscaling with formal verification perspective," *Software: Practice and Experience*, vol. 52, no. 11, pp. 2476–2495, 2022.
[7] S. Xie, J. Wang, B. Li, Z. Zhang, D. Li, and P. C. K. Hung, "Pbscaler: A bottleneck-aware autoscaling framework for microservice-based applications," *IEEE Transactions on Services Computing*, vol. 17, no. 2, pp. 604–616, 2024.
[8] V. Kalavri, J. Liagouris, M. Hoffmann, D. Dimitrova, M. Forshaw, and T. Roscoe, "Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 783–798.
[9] S. Luo, C. Lin, K. Ye, G. Xu, L. Zhang, G. Yang, H. Xu, and C. Xu, "Optimizing resource management for shared microservices: A scalable system design," *ACM Transactions on Computer Systems*, 2023.
[10] S. Padakandla, "A survey of reinforcement learning algorithms for dynamically varying environments," *ACM Computing Surveys (CSUR)*, vol. 54, no. 6, pp. 1–25, 2021.
[11] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 2015.
[12] X. He, Z. Tu, X. Xu, and Z. Wang, "Re-deploying microservices in edge and cloud environment for the optimization of user-perceived service quality," in *International Conference on Service-Oriented Computing*. Springer, 2019, pp. 555–560.
[13] B. Burns, J. Beda, and K. Hightower, *Kubernetes: up and running: dive into the future of infrastructure.* O'Reilly Media, 2019.
[14] R. S. Kannan, L. Subramanian, A. Raju, J. Ahn, J. Mars, and L. Tang, "Grandslam: Guaranteeing slas for jobs in microservices execution frameworks," in *Proceedings of the Fourteenth EuroSys Conference 2019*, ser. EuroSys '19. New York, NY, USA: Association for Computing Machinery, 2019.
[15] X. Hou, C. Li, J. Liu, L. Zhang, Y. Hu, and M. Guo, "Ant-man: towards agile power management in the microservice era," in *2020 SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Computer Society, 2020, pp. 1098–1111.
[16] G. Yu, P. Chen, and Z. Zheng, "Microscaler: Automatic scaling for microservices with an online learning approach," in *2019 IEEE International Conference on Web Services (ICWS)*, July 2019, pp. 68–75.
[17] L. Liu, "Qos-aware machine learning-based multiple resources scheduling for microservices in cloud environment," *arXiv preprint arXiv:1911.13208*, 2019.

[18] Y. Gan, Y. Zhang, K. Hu, D. Cheng, Y. He, M. Pancholi, and C. Delimitrou, "Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices," in *Proceedings of the twenty-fourth international conference on architectural support for programming languages and operating systems*, 2019, pp. 19–33.

[19] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand *et al.*, "Autopilot: workload autoscaling at google," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–16.

[20] M. Xu, C. Song, S. Ilager, S. S. Gill, J. Zhao, K. Ye, and C. Xu, "Coscal: Multifaceted scaling of microservices with reinforcement learning," *IEEE Transactions on Network and Service Management*, vol. 19, no. 4, pp. 3995–4009, 2022.

[21] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, "{FIRM}: An intelligent fine-grained resource management framework for {SLO-Oriented} microservices," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 805–825.

[22] S. Zhang, T. Wu, M. Pan, C. Zhang, and Y. Yu, "A-sarsa: A predictive container auto-scaling algorithm based on reinforcement learning," in *2020 IEEE International Conference on Web Services (ICWS)*, 2020, pp. 489–497.

[23] F. Rossi, M. Nardelli, and V. Cardellini, "Horizontal and vertical scaling of container-based applications using reinforcement learning," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, 2019, pp. 329–338.

[24] M. Xu and R. Buyya, "Brownout approach for adaptive management of resources and applications in cloud computing systems: A taxonomy and future directions," *ACM Comput. Surv.*, vol. 52, no. 1, jan 2019.

[25] X. Zhou, X. Peng, T. Xie, J. Sun, C. Xu, C. Ji, and W. Zhao, "Benchmarking microservice systems for software engineering research," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, Eds. ACM, 2018, pp. 323–324.

[26] Locust Contributors, "Locust," 2023, accessed: 2023-06-05. [Online]. Available: https://locust.io/

[27] J. Guo, Z. Chang, S. Wang, H. Ding, Y. Feng, L. Mao, and Y. Bao, "Who limits the resource efficiency of my datacenter: An analysis of alibaba datacenter traces," in *Proceedings of the international symposium on quality of service*, 2019, pp. 1–10.

[28] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[29] L. Buşoniu, R. Babuška, and B. De Schutter, "Multi-agent reinforcement learning: An overview," *Innovations in multi-agent systems and applications-1*, pp. 183–221, 2010.

[30] M. Zimmer, P. Viappiani, and P. Weng, "Teacher-student framework: a reinforcement learning approach," in *AAMAS Workshop Autonomous Robots and Multirobot Systems*, 2014.

[31] A. Hussein, M. M. Gaber, E. Elyan, and C. Jayne, "Imitation learning: A survey of learning methods," *ACM Computing Surveys (CSUR)*, vol. 50, no. 2, pp. 1–35, 2017.

[32] Y. Cao, Y. Zhao, J. Li, R. Lin, J. Zhang, and J. Chen, "Multi-tenant provisioning for quantum key distribution networks with heuristics and reinforcement learning: A comparative study," *IEEE Transactions on Network and Service Management*, vol. 17, no. 2, pp. 946–957, 2020.

[33] J. Rahman and P. Lama, "Predicting the end-to-end tail latency of containerized microservices in the cloud," in *2019 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2019, pp. 200–210.

[34] S. Fujimoto, H. Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," in *International conference on machine learning*. PMLR, 2018, pp. 1587–1596.

[35] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," in *NIPS 2014 Workshop on Deep Learning, December 2014*, 2014.

[36] M. Xu, C. Song, H. Wu, S. S. Gill, K. Ye, and C. Xu, "Esdnn: Deep neural network based multivariate workload prediction in cloud computing environments," *ACM Trans. Internet Technol.*, vol. 22, no. 3, aug 2022.

[37] C. Song, M. Xu, K. Ye, H. Wu, S. S. Gill, R. Buyya, and C. Xu, "Chainsformer: A chain latency-aware resource provisioning approach for microservices cluster," in *Service-Oriented Computing*, F. Monti, S. Rinderle-Ma, A. Ruiz Cortés, Z. Zheng, and M. Mecella, Eds. Cham: Springer Nature Switzerland, 2023, pp. 197–211.

[38] C. Carrión, "Kubernetes scheduling: Taxonomy, ongoing issues and challenges," *ACM Computing Surveys*, vol. 55, no. 7, pp. 1–37, 2022.
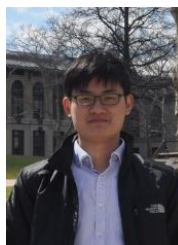
Haoyu Bai received his BSc degree from the University of Melbourne. Now he is a PhD student at the University of Melbourne, he is also a visiting student at Shenzhen Institute of Advanced Technology, Chinese Academy of Science. His main research interest includes efficient microservice-based application and system management.

Minxian Xu (Senior Member, IEEE) is currently an Associate Professor at Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences. his PhD degree from the University of Melbourne in 2019. His research interests include resource scheduling and optimization in cloud computing. He has co-authored 60+ peer-reviewed papers published in prominent international journals and conferences. His PhD thesis was awarded the 2019 IEEE TCSC Outstanding Ph.D. Dissertation Award. He was also awarded the 2023 IEEE TCSC Award for Excellence (Early Career Award).

Kejiang Ye (Senior Member, IEEE) received the BSc and PhD degrees from Zhejiang University, in 2008 and 2013, respectively. He was also a joint PhD student with the University of Sydney from 2012 to 2013. After graduation, he works as post-doc researcher with Carnegie Mellon University from 2014 to 2015 and Wayne State University from 2015 to 2016. He is currently a professor with the Shenzhen Institute of Advanced Technology, Chinese Academy of Science. His research interests focus on the performance, energy, and reliability of cloud computing and network systems.

Rajkumar Buyya (Fellow, IEEE) is currently a Redmond Barry distinguished professor and director with Cloud Computing and Distributed Systems (CLOUDS) Laboratory, University of Melbourne, Australia. He has authored more than 625 publications and seven textbooks including "Mastering Cloud Computing" published by McGraw Hill, China Machine Press, and Morgan Kaufmann for Indian, Chinese and international markets, respectively. He is one of the highly cited authors in computer science and software engineering worldwide (h-index=168, g-index=322, 149,000+citations).

Chengzhong Xu (Fellow, IEEE) received the Ph.D. degree in computer science and engineering from the University of Hong Kong in 1993. He is the Dean of Faculty of Science and Technology and the Interim Director of Institute of Collaborative Innovation, University of Macau. He published two research monographs and more than 300 peer-reviewed papers in journals and conference proceedings; his papers received about 17K citations with an H-index of 72. His main research interests lie in parallel and distributed computing and cloud computing.