# CoScal: Multifaceted Scaling of Microservices With Reinforcement Learning

Minxian Xu, *Member, IEEE*, Chenghao Song, Shashikant Ilager, Sukhpal Singh Gill, Juanjuan Zhao, Kejiang Ye, *Member, IEEE*, and Chengzhong Xu, *Fellow, IEEE*

***Abstract*—The emerging trend towards moving from monolithic applications to microservices has raised new performance challenges in cloud computing environments. Compared with traditional monolithic applications, the microservices are lightweight, fine-grained, and must be executed in a shorter time. Efficient scaling approaches are required to ensure microservices' system performance under diverse workloads with strict Quality of Service (QoS) requirements and optimize resource provisioning. To solve this problem, we investigate the trade-offs between the dominant scaling techniques, including horizontal scaling, vertical scaling, and brownout in terms of execution cost and response time. We first present a prediction algorithm based on gradient recurrent units to accurately predict workloads assisting in scaling to achieve efficient scaling. Further, we propose a multi-faceted scaling approach using reinforcement learning called CoScal to learn the scaling techniques efficiently. The proposed CoScal approach takes full advantage of data-driven decisions and improves the system performance in terms of high communication cost and delay. We validate our proposed solution by implementing a containerized microservice prototype system and evaluated with two microservice applications. The extensive experiments demonstrate that CoScal reduces response time by 19%-29% and decreases the connection time of services by 16% when compared with the state-of-the-art scaling techniques for Sock Shop application. CoScal can also improve the number of successful transactions with 6%-10% for Stan's Robot Shop application.**

***Index Terms*—Cloud computing, workload prediction, microservices, reinforcement learning, brownout, scalability.**

Manuscript received 14 March 2022; revised 20 July 2022 and 20 September 2022; accepted 24 September 2022. Date of publication 28 September 2022; date of current version 31 January 2023. This work is supported by National Key R&D Program of China (No. 2021YFB3300200), National Natural Science Foundation of China (No. 62102408, 62072451), Shenzhen Science and Technology Program (Grant No. RCBS20210609104609044), Youth Innovation Promotion Association CAS (2019349), and CCF-Huawei Innovative Research Plan. The associate editor coordinating the review of this article and approving it for publication was N. Zincir-Heywood. *(Corresponding author: Kejiang Ye.)*

Minxian Xu, Chenghao Song, Juanjuan Zhao, and Kejiang Ye are with the Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences, Shenzhen 518055, China (e-mail: mx.xu@siat.ac.cn; ch.song@siat.ac.cn; jj.zhao@siat.ac.cn; kj.ye@siat.ac.cn).

Shashikant Ilager is with the Department of Informatics, Vienna University of Technology, 1040 Vienna, Austria (e-mail: shashikant.ilager@tuwien.ac.at).

Sukhpal Singh Gill is with the School of Electronic Engineering and Computer Science, Queen Mary University of London, E1 4NS London, U.K. (e-mail: s.s.gill@qmul.ac.uk).

Chengzhong Xu is with the State Key Laboratory of IoTSC, University of Macau, Macau, China (e-mail: czxu@um.edu.mo).

Digital Object Identifier 10.1109/TNSM.2022.3210211

## I. Introduction

THE CLOUD computing paradigm needs to satisfy strict performance requirements for diverse users hosting heterogeneous workloads. These cloud workloads are driven by applications belonging to various domains, including enterprise businesses and government services that require uninterrupted, reliable service delivery [1], [2]. The ever-increasing complexity of these large applications has recently stipulated moving towards microservice-based application development and deployment. The microservice paradigm has shifted the traditional monolithic application design into decomposed and self-contained standalone application components, which generally communicate through a RESTful Application Programming Interface (API) [3], [4]. The features of microservices include lightweight design, flexible development, continuous deployment, and independent management. These attractive features have promoted cloud computing providers, including Amazon, Google, and Alibaba, to adopt microservice-based application service delivery models and platforms.

The users request on-demand services from cloud service providers under the specified Quality of Service (QoS) requirements with the pay-as-you-go model [5]. If the QoS is unsatisfied, service providers may suffer revenue loss due to a Service Level Agreement (SLA) violation with their customers. The QoS is defined in terms of different performance metrics, including resource availability and latency. Satisfying the QoS of microservices-based applications is more challenging than traditional monolithic applications since the performance of microservices is more sensitive to latency. This is further complicated by dynamic workload levels. Thus, service providers aim to allocate sufficient resources to application workloads to avoid QoS degradation and performance bottlenecks and prevent huge revenue loss.

Over-provisioning is an effective approach to ensure the QoS of microservices. The over-provisioning method aims to allocate additional resources for user workloads to provide guaranteed performance [6]. However, provisioning as many resources as possible is cost-ineffective due to the finite hardware resources, energy budget, and operational costs of the cloud data centers [7]. Maintaining a large number of physical machines with lower utilization can lead to higher costs for service providers. Therefore, resource scaling approaches have been applied to optimize resource planning and provisioning in microservice-based applications. The

resource scaling methods are significant for infrastructure providers as they can contribute to cost reduction, increase resource utilization, and simultaneously improve the QoS parameters of microservice applications.

The dominant resource scaling approaches for microservices can be classified into two categories, horizontal scaling and vertical scaling [8]. Horizontal scaling adjusts the provisioned resource quantities by adding or removing microservice replicas to improve resource usage and QoS parameters (e.g., system availability, latency). In contrast, vertical scaling adjusts the capabilities of existing provisioned resources by increasing or decreasing the amount of CPU, memory, and network resources. These two approaches have both been validated to be effective for resource scaling.

Currently, Kubernetes (promoted by Google) has become the most popular platform for container-based microservice application deployment in Cloud [9]. Kubernetes supports horizontal scaling based on a runtime usage threshold. However, such a threshold-based scaling method only works well in simple cases, and achieves sub-optimal solutions in complex workload conditions (e.g., dynamic workload conditions). To address this issue, many studies have proposed different auto-scaling algorithms based on the default Kubernete's auto-scaling algorithms [10]. Nevertheless, these algorithms primarily focus on a single type of resource, e.g., CPU, memory, or bandwidth which are infeasible for many workload scenarios. In addition, Horizontal and vertical scaling has many practical limitations when applied individually. For instance, the communication overhead of horizontal scaling are non-trivial, which directly impacts application QoS, especially in microservice-based applications due to their shorter execution time and increased latency sensitivity. Although vertical scaling can increase the capacity of a provisioned physical machine's resource, upgrading the resource capacity in runtime is expensive. A recent study has considered combining these two scaling approaches together [8], although it does not address the complex real world scenarios such as infrequent bursty workloads that leads to overloading of the whole system, thus, affecting overall infrastructure and application QoS [11].

Some recent methods, such as brownout techniques, [12] have explored an alternative scaling method complementing both the horizontal and vertical scaling. Brownout is a self-adaptive approach for managing application components by dynamically activating and deactivating optional application components to be adaptive to the variance of workloads. The brownout can effectively address the limitations of vertical scaling (limited resource capacity of local machines) and horizontal scaling (the resource usage of replicating the microservice to other machines) in overloaded conditions. In the microservice-based application, a brownout can temporarily deactivate some of the optional microservices to reduce resource usage while ensuring the necessary functionalities of microservice applications. Thus, comprehensive scaling techniques addressing the limitations of existing individualistic approaches are required for adaptive and efficient scaling in microservice environments.

### A. Existing Challenges

The efficient scaling of microservices poses several significant challenges. Firstly, the micro-service based cloud workloads have high-variance in resource usage and are sensitive to resource types (CPU, network, memory). It is difficult to predict the accurate amount of workloads in a specific time period. An accurate workload prediction is an essential element for auto-scaling approaches in microservices, e.g., in horizontal scaling, the predicted workload can enable service providers to boot up and deploy applications beforehand to avoid startup cost and QoS degradation. Secondly, assigning resources to microservices is an NP-hard problem considering the multi-dimensional resources. Due to the various run-time parameters of microservices and hardware configurations, it is time-consuming to find the optimal results given the large solution space [13]. Thirdly, there are trade-offs considering when and how to trigger the auto-scaling algorithms to handle the situations with high dynamics while ensuring the QoS requirement. We propose multi-faceted, data-driven, and adaptive auto-scaling approaches for microservices to address the above discussed challenges.

### B. Our Contributions

In this paper, we investigate the promising approach called **CoScal** that combines horizontal scaling, vertical scaling, and brownout techniques to address the resource scaling problem in microservice-based cloud computing environments. The proposed approach exploits the advantages of above-mentioned individual techniques, including the high availability of horizontal scaling, fine-grained control of vertical scaling, and self-adaptability of brownout. This combined auto-scaling approach is more effective under diverse and complex workload scenarios derived from Alibaba traces [14]. The key **contributions** of this paper are:

- A performance analysis of horizontal scaling, vertical scaling, and Brownout investigates the trade-offs in execution costs and performance of microservices.
- A Reinforcement Learning (RL) based scaling algorithm for decision making to optimize the performance of microservices, such as response time, connection time, and the number of failed requests.
- A prototype system implementing the proposed approach is evaluated with realistic traces and extensive experiments.

The rest of the paper is organized as follows: Section II discusses the related work for auto-scaling microservices in the cloud computing environment. The motivation and performance analysis of the applied techniques in this work are introduced in Section III. Section IV depicts the system model of our proposed approach. The proposed algorithm based on RL is detailed introduced in Section V. The Section VI illustrates the details of our experiments conducted using a dataset derived from realistic traces and demonstrates the feasibility of our approach to improve the resource scaling of cloud data centers. Finally, conclusions along with the future directions are given in Section VII.

## II. Related Work

In this section, we discuss existing research works in microservices scaling. The current scaling approaches for performance optimization in microservices can be mainly categorized into three categories: the performance model, resource orchestration, and prototype systems.

### A. Performance Model

The performance model is applied for modeling the resource usage of the system and the performance of cloud services. Thus, resource management policies can be designed to fit the performance requirements. Nishtala *et al.* [15], [16] proposed scalable QoS-aware resource management approaches for latency-critical and co-located services in traditional cloud data centers. The proposed approaches are based on heuristic and reinforcement learning techniques to achieve energy-efficient goals, which require no service or system-specific information. Liu [17] analyzed the bottlenecks of the mainstream microservice applications and applied multiple machine learning models to schedule resources. The first model is used for finding the resource area that satisfies the microservice performance. The second model investigates the trade-offs between the QoS and allocated resources, and the third model dynamically adjusts resources according to the system states. Chang *et al.* [18] designed an automatic resource planning approach and modeled application performance based on an empirical model. Khazaei *et al.* [19] constructed a performance model via analyzing microservice platforms. This approach applies the Markov chain model to represent microservice resources, virtual machine (VM) resources, and physical machine resources. The number of tasks, number of microservices, and number of VMs have all been considered as the system states. Although the performance model is a popular approach for performance optimization, the model needs to be reconfigured or retrained when the service or environment changes dramatically. Gan *et al.* [20] exploited the prediction method for QoS violations, which utilizes a set of machine learning models and large-scale history data to locate the microservices that lead to QoS violations. The approach can relieve the QoS degradation by reallocating hardware resources. Qiu *et al.* [10] proposed a fine-grained control framework to relieve resource competition and optimize resource utilization and analyzed the microservice dependency relationship, microservice chains, and key call paths. Kannan *et al.* [21] modeled the multiple-stage tasks as Directed Acyclic Graph (DAG) and used DAG to estimate the task completion time. Yu *et al.* [22] proposed Microscaler framework by using service mesh to record the resource usage behaviors and applied online learning and heuristic approaches to obtain the near-optimal solutions for resource demands.

Scaling commands must be configured manually based on the performance model in all these approaches. Thus, they are far slower to react to load variations. We apply neural network-based workload prediction to achieve high adaptation to load variances to solve this challenge.

### B. Resource Orchestration

Other works apply vertical scaling or horizontal scaling that are not required to be configured manually based on the performance model, which utilizes resource orchestration to optimize resource provisioning for microservices by allocating and managing resources efficiently. Suresh *et al.* [23] investigated the overload control mechanism for a microservice cluster with complex dependency. The proposed approach considers the resource sharing problem under a multi-tenancy scenario, in which all the tasks are modeled as DAG. The scheduling is processed at the workflow level and requests level separately. Zhou *et al.* [24] pointed out that monitoring and collecting the data of each microservice under large-scale and high-dynamicity scenarios is not feasible. Therefore, authors proposed a workload control approach to maintain a self-adaptive threshold and algorithm for each microservice. Each microservice can shed the loads independently with quite limited communication costs. Rzdaca *et al.* [25] designed Auto-pilot as an auto-scaling approach to scale based on the change of workloads. Auto-pilot combines time-series analysis and scaling the number of microservices with related CPU and memory amount. Kwan *et al.* [8] designed an approach that combines vertical and horizontal scaling methods, which deploy microservices to suitable hardware to efficiently reduce service delay due to burst requests. Hou *et al.* [26] introduced a power-aware and latency-aware scheduling approach to scale resources from micro and macro perspectives. They also apply the decision tree and tagging method to achieve fast resources matching. He *et al.* [27] took advantage of genetic and heuristic algorithms to find the optimized microservice deployment place under an edge-cloud environment. Zhang *et al.* [28] proposed a predictive RL algorithm to horizontally scale containers based on the Autoregressive Integrated Moving Average (ARIMA) model and neural network model, which can ensure the predictability and accuracy of the scaling process. Rossi *et al.* [29] introduced RL-based approaches to control the horizontal and vertical scaling for containers to increase system flexibility under varying workloads, which also accelerate the learning process by exploiting different degrees of knowledge about the environment. However, this work was only evaluated with synthetic workloads. Gias *et al.* [30] proposed a model-driven scaling approach, named ATOM, for microservices via analyzing a layered queueing network of applications. This approach can dynamically adjust the number of replications via horizontal scaling.

In resource orchestration approaches, limited hybrid scaling for microservices has been proposed, and most of them focused on a single scaling technique, either vertical or horizontal, but not on both. The existing hybrid approaches are threshold-based or heuristic that requires heavy manual configurations. Unlike these efforts, we apply a hybrid scaling technique that combines vertical, horizontal, and brownout together, making adaptive decisions via RL.

### C. Prototype Systems and Tools

Kubernetes [31] and Docker Swarm have become the dominant systems for managing microservices. However, the

TABLE I
COMPARISON OF RELATED WORK

| Approach | Scaling Techniques | | | workload prediction | | Scheduling Policy | |
|---|---|---|---|---|---|---|---|
| | Vertical | Horizontal | Brownout | Linear | Non-Linear | Heuristic | Supervised Learning |
| Liu et al. [15] | √ | | | | √ | | √ |
| Qiu et al. [10] | √ | √ | | | | | √ |
| Kannan et al. [16] | √ | | | √ | | √ | |
| Yu et al. [17] | | √ | | | √ | √ | |
| Zhou et al. [18] | √ | | | √ | | √ | |
| Rzdaca et al. [19] | √ | √ | | √ | | √ | √ |
| Kwan et al. [8] | √ | √ | | | | √ | |
| Hou et al. [20] | √ | | | √ | | √ | |
| Zhang et al. [21] | | √ | | √ | | | √ |
| Rossi et al. [22] | √ | √ | | | | | √ |
| Gias et al. [23] | | √ | | | | | √ |
| Xu et al. [12] | | √ | √ | √ | | √ | |
| CoScal (Our Approach) | √ | √ | √ | | √ | | √ |

scaling techniques in these systems are primarily threshold-based or static policies. Kiss *et al.* [32] implemented a generic microservice orchestration platform for heterogeneous cloud clusters, which can auto-scale resources without binding to specific applications. Zhou *et al.* [33] analyzed and compared the interaction patterns of open-source microservice-based applications. They found that the investigated open source applications are small-scale and provided a medium-scale application called TrainTicket. Xu *et al.* [12] proposed a prototype system for managing co-located interactive and batch microservices based on the brownout approach and workloads deferral to achieve an energy-efficient cloud data center.

In these prototype systems and tools, threshold-based heuristic algorithms are applied popularly. However, heuristic algorithms can find a solution quickly, the performance needs of manually tuning various configurations, especially in an environment with high dynamics [29]. Unlike these approaches, we utilize the RL-based approach with an adaptive nature to learn and make good decisions via interactions with the environment.

### D. Critical Analysis

This paper contributes to the growing body of research related to the microservice area. The Table I compares the proposed approach **(CoScal)** with related works based on the scaling techniques, workload prediction techniques, and type of scheduling algorithms. Given the contributions of the existing works, it is important to highlight the key difference between our work and the prior ones. To the best of our knowledge, the proposed work is the first to offer a multi-faceted scaling approach based on vertical scaling, horizontal scaling, and brownout. Prior works only applied included one or two techniques. We also utilize gradient recurrent units for accurate workload prediction and apply an RL-based algorithm for resource management to offer flexible adaption for a highly dynamic environment.

## III. MOTIVATION: PERFORMANCE TRADE-OFFS IN INDIVIDUAL SCALING TECHNIQUES

In this section, we conduct experiments to investigate the effects of individual scaling techniques (vertical, horizontal, and brownout) on important performance metrics in microservice-based computing environments. The results illustrate that applying individual scaling techniques such as horizontal scaling or vertical scaling does not yield desired results due to trade-offs.

TABLE II
REQUEST DISTRIBUTION FOR SOCK SHOP APPLICATION

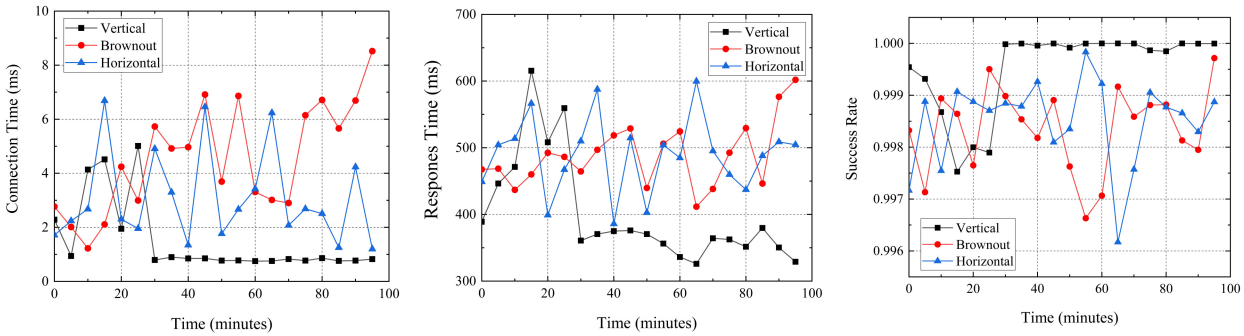
| | Request Distribution | | | |
|---|---|---|---|---|
| Microservice Type | Home | Orders | Catalog | Others |
| Percentage | 83.8% | 7.2% | 5.4% | 3.6% |

### A. Use Case Setup

In this motivational study, we have used a microservice application named Sock Shop,[1] which is an online e-commerce website for products sale. The Sock Shop application consists of multiple microservices representing different components of the application, including front-end User Interface (UI), catalog, carts, and some other supporting microservices, which are deployed on two Docker Swarm nodes.

In this study, we consider the browsing workload for the Sock Shop application. The users can view the information of the items, add or remove items in the carts, pay for the items. The used microservices in the Sock Shop application and the request distribution are summarized in Table II. We have investigated the high load use case representing the system's over-utilized (when utilization is above a predefined threshold) condition from Alibaba's sample trace [14]. The sample trace contains 1200 lines of data representing resource utilization in 6000 seconds, where the data is collected every 5 seconds. We use JMeter[2] toolkit to generate workloads for the Sock Shop application, which simulates the scenario of users visiting the website for shopping. JMeter is an open-source and completely Java-based application. It is mainly used to conduct stress tests of the target server and validate test-case functionalities of service endpoints. Only one node is initially deployed with microservices in our setup, and the second node is kept in an idle state. In the horizontal scaling, the replicates of Sock Shop microservices are deployed on the second node. In the vertical scaling, the CPU is configured to be scaled to a maximum of 8 cores, and the memory can be scaled to a maximum of 3.5 GB. In the brownout approach, the optional microservices (3 out of 14, i.e., the recommendation engine)

[1] microservices-demo.github.io
[2] https://jmeter.apache.org/




(a) Connection time (b) Response time (c) Success rate

Fig. 1. Performance comparison of different scaling techniques when applied separately.

are temporarily deactivated (due to space limitation, please see the more detailed descriptions in Section V-A).

We apply the individual scaling techniques separately to evaluate system performance regarding connection time, response time, and success rate of requests. *Connection time* represents the time required to establish a connection between the user and the target microservice server. *Response time* represents the total time from the start of the request to the reception of the response. *Success rate* represents the percentage of successful requests (i.e., expected response is received for request) received by the user.

### B. Comparison Between Scaling Techniques

Fig. 1(a) represents the connection time achieved by three different scaling techniques. The collected data is obtained every 200 ms, and the average value is calculated based on every 5-minute interval. We choose 5-minute interval based on Alibaba's practice where 5-minute minimum time interval is considered for scheduling decisions. A smaller time interval can lead to frequent scaling costs, while a longer time interval cannot respond to the system change promptly. As this is a motivational example, we only conduct the experiments one time. We can observe that the vertical scaling converges faster to a stable connection time than horizontal and brownout scaling. However, at the initial period of scaling, its performance for connection time is not as good as brownout. The brownout approach initially achieves the best connection time as it deactivates optional parts that help provide additional resources for the front-end connections. After some periods, the connection time keeps growing because all the available optional components have been deactivated. Horizontal scaling can achieve better connections than brownout since more resources are provided regarding the number of nodes. However, the extra communication costs involving two replicas of services deployed on different hosts induce higher connection time in horizontal scaling than vertical scaling, as the communication in vertical scaling is negligible as it remain on the same host.

Figs. 1(b) and 1(c) represent the response time and success rate achieved by different scaling methods. The vertical scaling achieves the best performance with the lowest response time and the highest success rate. The reason is that the vertical scaling on the local machine can improve the system performance without incurring additional overheads. In contrast, Brownout can incur some delays induced by its resource optimization technique. Moreover, the horizontal scaling leads to poor performance in response time and success rate of user requests due to its associated overhead, such as boot-up cost of new instances and frequent scaling operations, which is crucial in highly time-sensitive microservice environments. Although vertical scaling achieves the best performance when compared to the other two techniques, it is not effective when the local machine is overloaded and constrained by the maximum resource capacity of an individual physical machine.

To summarize, individual scaling techniques may not yield the desired result in complex conditions. A multi-faceted scaling can solve the limitations of individual techniques. For instance, when a limited capacity is left on a hosted machine (i.e., vertical scaling hits resource limitations), the brownout technique can be triggered to maintain the connection time by relieving resource overloading. Meanwhile, the horizontal scaling can be applied when enough resources are available to host additional workload, and optional components can be reactivated further. However, the horizontal scaling requires more time to take effect compared to brownout as it requires significant time to activate host and initialize new container instances. Thus, under the high-variable workloads, the choice and combination of scaling techniques play an important role in delivering reliable services, especially in latency-sensitive microservice environments. Therefore, to overcome the limitations of individual scaling techniques, a joint, data-driven, and adaptive auto-scaling framework is necessary to provide optimal scaling decisions based on the current infrastructure and workload conditions.

## IV. System Model

In this section, we describe our proposed auto-scaling system named as *CoScal* that is composed of three modules, including *Workload Analyzer*, *Workload Predictor*, and *RL-based Resource Scaler*, as shown in Fig. 2. It is noted that this is a general system model, and the modules can be extended to new workload analysis techniques and more advanced RL-based resource scheduling solutions. The details of these modules are given in the following sections:

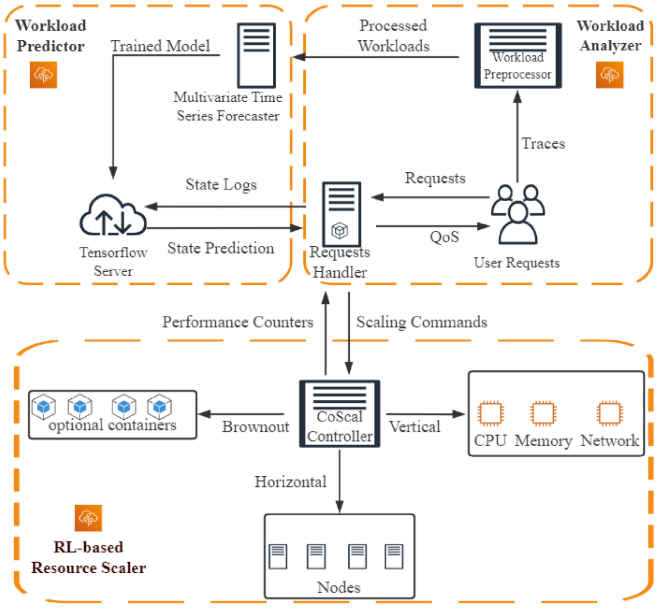



Fig. 2. The *CoScal* system model.

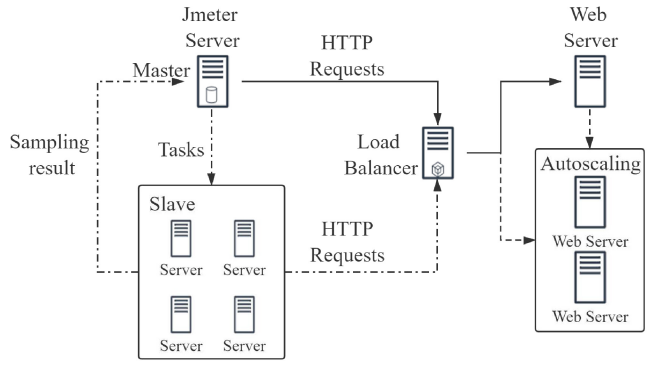



Fig. 3. Schematic diagram of requests generation by JMeter.

### A. Workload Analyzer

The *Workload Analyzer* module analyzes system workload and processes the raw data generated by log files. In this module, the *Requests Handler* handles the user-generated requests, which allocates the requests to hosts. The *Pre-processor* extracts the required meta information of workloads (e.g., time series and resource utilization metrics). Later, the *Workload Predictor* module takes our inputs to perform its stated operations (described in Section IV-B).

Fig. 3 shows a schematic diagram of load generation based on JMeter. In this case, The JMeter Master and Workers (Slaves) are regarded as part of the *User Requests* and *Requests Handler* components. Web servers are the infrastructure to provision resources. Based on our practice, when the number of requests is huge, a single JMeter server mode may not function well due to overloads, then the master-worker will be used collaboratively. With each test case, the data of collected metrics will be stored locally, such as timestamps, response time, connection time, and the number of failed requests.

### B. Workload Predictor

The *Workload Predictor* module in Fig. 2 is responsible for estimating the expected load in the next scheduling window, which guides scaling methods to decide the amount of resources to be added or removed. This module can be constituted by different predictive models such as *Multivariate Time Series Forecaster* [34], which accepts the pre-processed information of workloads from the *Workload Preprocessor* and predicts the future workload based on suitable predictive models, such as long short-term memory (LSTM) or gated recurrent unit (GRU) model. The responsibility of the *TensorFlow Server* is to manage the *Trained Model*, including the training process of the models and updating the trained models if necessary, and finally deploying trained models. As shown in Fig. 2, the communications between *Workload Analyzer* and *Workload Predictor* transfer the inputs with past system data to the *TensorFlow Server* (the left-bottom corner of this module). It provides feedback such as predicted workload level to *Requests Handler*.

To generate user requests for the prototype system, the load generator toolkits, e.g., JMeter, can be utilized. And the typical microservice application, like Sock Shop [35] as introduced in Section III, can be deployed as benchmark application. The profiling approach based on the stress test can establish a model to represent the relationship between the number of requests and resource utilization (see Section V-A for more details).

### C. RL-Based Resource Scaler

The *CoScal Controller* is the core component in this module which makes decisions on scaling strategies based on the RL framework. Compared with static performance models and heuristic-based approaches that suffer from model reconstructions and retraining problems, the RL-based approach is well suited for learning resource scaling policies to address dynamic system status [36]. The *RL-based Resource Scaler* in Fig. 2 receives the user requests from *Request Handler* and information like predicted workloads from *Workload Predictor* in our system. After executing the scaling techniques, the *CoScal Controller* collects the data of performance metrics, such as response time, and sends the data back to the users. The data is the response to the user request to the application.

In the *RL-based Resource Scaler* module, the supported techniques work in different manners. Vertical scaling is applied to the local machine with multiple resources, e.g., CPU, memory, and network capacity. The vertical scaling is faster compared with other techniques as shown in Section III, it can be leveraged initially. The brownout technique is also applied to the local machine, where the optional microservices can be dynamically deactivated to relieve the overloaded situation that vertical scaling cannot handle alone. Horizontal scaling is applied at the node level by adding or removing additional nodes into the system. Considering the communication costs of horizontal scaling, this technique can be applied when vertical scaling and brownout cannot handle the workloads level and keep the system in a normal state.

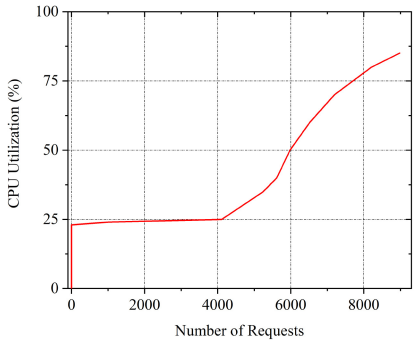



Fig. 4. The number of requests and the corresponding CPU utilization.

TABLE III
DEFINED WORKLOAD LEVELS AND CORRESPONDING MEANINGS

| Workload Level | Meaning |
|---|---|
| Level 0 | No loads are allocated to the host, and the hosts can be switched into the low-power mode or turned off, i.e., the resource utilization is 0. |
| Level 1 | The host is under light load, with low latency and low CPU and memory usage, i.e., the resource utilization ranges from 0% to 25%. |
| Level 2 | The host is at a medium load level, the delay is low, and the QoS is not significantly affected, i.e., utilization is between 25% and 50%. |
| Level 3 | The host is potentially under heavy load, the CPU or memory usage is high, and QoS may have affected; thus, the scaling techniques may be triggered if needed, i.e., utilization is between 50% to 75%. |
| Level 4 | The host is overloaded, the QoS will be significantly impacted, and the effective scaling approaches should be performed, i.e., utilization is between 75% to 100%. |

## V. *CoScal*: A MULTI-FACETED AUTO-SCALING IN MICROSERVICE ENVIRONMENTS

This section introduces the key elements to realize our system model, including the performance profiling, neural network-based workload prediction and RL-based policy for scaling microservices.

### A. Performance Profiling

To estimate the approximate resource usage corresponding to a different amount of workloads with the Workload Analyzer module in Fig. 2, we use a deep neural network model to profile the performance of machines. To simulate the resource utilization in a realistic environment, we use the data derived from Alibaba traces, including workload traces of 4000 machines' containing resource usage data for 8 days [14].

The detailed profiling procedures are as follows: we consider the scheduling interval as 5 minutes, as 5 minutes are the minimum monitoring interval for data collection. A short scheduling interval can make too frequent scaling operations to influence system stability, while a long scheduling interval can delay the adaptive scaling decisions. Firstly, we apply stress test by gradually increasing the number of requests over 5 minutes with 200 requests in each test case and sending these requests to the host. As the number of requests increases, the host's resource utilization also increases. Then we use the nmon [37] performance monitoring toolkit to record the change of utilization in the host as per the number of requests sent. In this way, a detailed mapping relationship between resource utilization and the number of requests is obtained. Based on the profiled data, we apply a Multi-Layer Perceptron (MLP) with three layers to establish a model to represent the relationship efficiently. Compared with the traditional regression-based approaches, the MLP-based approaches can capture a more accurate relationship between workloads and utilization. Finally, we can convert the host utilization into the number of requests dispatching to our system for any utilization level with the trained model.

As shown in Fig. 4, we demonstrate the results between CPU utilization and the number of requests based on the tested host. We can observe the CPU utilization change under the different number of requests. The host consumes about 25% CPU utilization when it does not handle any requests. This is due to the resource consumption by the operating system and deployed applications. The CPU utilization increases gradually with the increased number of requests. In this experiment, a host can accept at most 9000 requests. We can also obtain the relationship between other resource types and the number of requests. Based on our experiments for the Sock Shop application, the CPU is a dominant resource type indicating that the deployed application is mainly compute-intensive.

### B. Neural Network-Based Workload Prediction

To address the prediction of resource usage levels in the Workload Predictor module in Fig. 2, we realize a neural network-based workload prediction approach. To reduce the state space of the RL model, we consider dividing the workloads into several levels (the exact number of levels configured) representing different levels of utilization. Moreover, this also helps to apply similar scaling techniques for workloads at the same level. This helps to significantly reduce the state-action space in the RL-based approach. We divide the workloads into five levels that can represent the degree of overloads, as shown in Table III. The overloaded threshold is configured as 75%, as it has been evaluated in our previous work [12] that it can achieve good trade-offs between resource utilization and QoS.

To accurately predict the server load status in the next time interval, we apply the Multivariate Time Series forecasting (MTFS) method, which converts multivariate time series forecasting into supervised learning. The MTFS algorithm can be applied to any time-related dataset, and it is highly correlated to temporal aspects and contains all the data from the previous time intervals. The MTFS makes each generated supervised learning sequence to have sample labeled datasets in the algorithm.

The Algorithm 1 shows the pseudocode of the MTFS algorithm, which generates labeled time series data for supervised learning. With the original time series dataset $E$, the algorithm first generates an empty matrix $S$ to store the supervised learning sequence. After the labeled data is generated for supervised learning, to achieve an accurate prediction for workloads, we apply the Gated Recurrent Unit (GRU) [38] derived from Recurrent Neural Network (RNN) [39], which

**Algorithm 1:** CoScal: Workload Preprocessing

**Input** : Multivariate time series dataset $E$, time intervals $T$, $k$ time-related variables, and a dataset $F$ needs to be forecast
**Output**: Supervised learning dataset $S$, each row of it has $2k + 3$ data

Initialize an empty matrix $S$ to record supervised time series data
**for** $t$ *from 1 to* $T$ **do**
  Record data in current time interval: $C(t) \leftarrow E(t)$
  **if** $t = 1$ **then**
    Record NONE
  **else**
    Record data in last time interval: $L(t-1) \leftarrow E(t-1)$
  **end**
  **if** $t = n - 1$ **then**
    Record NONE
  **else**
    Record data in next time interval: $F(t+1) \leftarrow E(t+1)$
  **end**
  Merge $L(t-1)$, $C(t)$ and $F(t+1)$ together into $S(t)$:
  $S(t) \leftarrow \{L(t-1), C(t), F(t+1)\}$
  $L(t-1)$, $C(t)$ are marked as samples in supervised learning
  $F(t+1)$ are marked as labels in supervised learning
  **if** *S(t) contains NONE* **then**
    Remove $S(t)$
  Update $S$
**end**

TABLE IV
MSE OF OUR WORKLOAD PREDICTION ALGORITHM

| Time (minutes) | 100 | 200 | 300 | 400 | 500 |
|---|---|---|---|---|---|
| MSE ($10^{-3}$) | 3.3 | 2.8 | 4.2 | 4.8 | 6.2 |

has been validated to have better performance in time series related prediction than traditional RNNs. Although RNN can use its memory to process a set of inputs sequentially, it is inefficient to learn long-term memory dependencies due to gradient vanishing, while GRU can overcome this limitation by merging the data processing elements (gates).

Table IV shows the mean square errors (MSE) of actual utilization and predicted utilization for Alibaba workloads during different periods. The MSE has lower values about $3 \times 10^{-3}$ to $7 \times 10^{-3}$ (in [40], the RNN-based and LSTM-based approaches have the MSE values about $4 \times 10^{-2}$), which demonstrates that our workload prediction algorithm has a good performance in utilization prediction and validates that the neural network based approach is suitable for cloud workloads prediction.

### C. RL-Based Resource Scaling

The RL-based approach solves sequential decision-making problems by modeling the problem as Markov Decision Process [41]. At each time interval $t$, the system is at a state $s_t \in S$, and performs an action $a_t \in A$ based on policy $\pi_\theta$, where $\theta$ are parameters configured in model. The state-space $S$ is mapped with action space $A$. In the following time intervals, the current system state can reach another state by taking actions and obtaining reward $r_t \in R$, calculated via reward function $r(s_t, a_t)$. The next state will only rely on the current state and the performed actions on it. The rewards represent the benefits that can be achieved by transiting the state from $s_t$ to $s_{t+1}$. When transferring states, there is also a transition probability of presenting the possibility to take different actions. The objective of RL is to maximize the expected cumulative reward by optimizing policy. $Q$-learning [42] is a typical type of RL to maximize the value function $Q_{\pi_\theta}(s, a)$. The value function estimates the expected cumulative reward of state $s$ with action $a$ under policy $\pi_\theta$. Consider action $a_t$ is selected at time interval $t$, and at time interval $t + 1$ with reward $r(s_t, a_t)$, the value of $Q$ function can be updated as:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r(s_t, a_t)] + \gamma max_{a'} Q\big(s_{t+1}, a'\big) - Q(s_t, a_t)\big] \quad (1)$$

where $\alpha \in (0, 1]$ is the learning rate and $\gamma \in [0, 1]$ is the discount factor. The Equation defines a mapping table containing states with actions and their expected value. The learning process happens in the form of $S * A- > R$ over time to achieve optimized results via iterative trials. We consider the state $S$ as workload level $S = \{0, 1, 2, \ldots, W\}$, where $W \in Z$ represents the maximum level of workloads and is a non-negative value.

To illustrate our RL-based multi-faceted scaling framework (*CoScal*), let us assume that we have a set of physical machines $P = (pm_1, pm_2, \ldots, pm_K)$ as infrastructure to provision resources for microservices. Each $pm_k$ can be represented with a tuple $U_k = (u_k^1, u_k^2, \ldots, u_k^I)$, where $u_k^i$ represents resource utilization of type $i$ with total $I$ types on physical machines. For each $pm_k$, the actions performed on it, which are denoted as $a_k^i = \{h_k, v_k^i, b_k\}$, where $h_k \in [-n, n]$ presents the $n \in Z$ number of replicates in horizontal scaling, $v_k^i \in [-m, m]$, where $m \in R$, represents the amount of resources via vertical scaling for resource type $i$, and $b_k \in \{0, 1\}$ represents the whether brownout is triggered $(b_k = 1)$ or not $(b_k = 0)$. The positive and negative values of $h_k$ and $v_k^i$ represent more resources are added and removed respectively, and value 0 means no change will be performed. Considering the total number of physical machines is $K$, the final set of actions is the Cartesian product of the sub-action sets as: $A = \prod_{k=1}^{K} \prod_{i=1}^{I} a_k^i$.

The objective of our technique is to find the suitable configuration of resources by dynamically adjusting the provisioned resources to adapt to changes in the environments, e.g., the load fluctuations. However, the amount of scaled resources is limited by the available resources on physical machine $pm_k$ and the minimum resources allocated to microservices. To avoid unnecessary vertical scaling, we consider adding an action $a_g$ to make decisions from the global view. Let us consider the scenario that when the system with $P = \{pm_1, pm_2\}$ is running at the normal states that vertical scaling is sufficient for adjusting resources. However, when unpredictable workloads arrive, the physical machines are overloaded. To handle such bursts, horizontal scaling must be performed on the system. If both $pm_1$ and $pm_2$ are completed with vertical scaling, two more physical machines are added. However, the system may only need one more physical machine to keep the system at the normal state. Therefore, to optimize resource usage, the action $a_g$ is required to make scaling decisions based on a global view.

*CoScal* incorporates the offline training and online training approach together to achieve optimized actions. Once the load change is identified, *CoScal* can select an action in response to the QoS or performance degradation of deployed applications. Given $s_t$ as the observed state, a policy exploits the knowledge of previous decisions (offline). It evaluates the performance of new selected actions (online), improving the mapping relationship between states and actions. We apply the $\epsilon$-greedy policy, [36] a standard policy to balance exploration and exploitation [43]. In $\epsilon$-greedy policy, a random action with probability equals $\epsilon$ is selected. Otherwise, it chooses the action with the maximum $Q$ value.

The final objective of *CoScal* is to improve the QoS of services and utilization of physical machines. The reward is modeled based on this objective which composes of two parts. For the QoS, we choose to use response time as the metric, representing the latency from submitting the request to completion. We model the reward of response time $R_{qos}(rt)$ based on the maximum acceptable response time with $RT_{max}$. As shown in Equation (2), when the system is working at the normal status, the reward is 1. However, when the system performance violates the $RT_{max}$, the reward converges to 0, punishing the actions that lead to overloaded situations.

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

As for the reward of resource utilization (please note that the utilization can also be referred as resource costs in our model), we model it as shown in Equation (3). Here, $U_k^{max}$ defines the maximum utilization threshold of $pm_k$, which is also the highest utilization of all resource types (CPU and memory utilization are both considered). $u_k$ is the current utilization of $pm_k$. Higher utilization without violating the threshold can contribute positively to the reward, while utilization above the threshold can undermine the reward.

$$R_{util}(u_k) = \begin{cases} \frac{\sum_{k=1}^{K} U_k^{max} - u_k}{K} + 1, & u_k \leq U_k^{max} \\ \frac{\sum_{k=1}^{K} u_k - U_k^{max}}{K} + 1, & u_k > U_k^{max} \end{cases} \tag{3}$$

Equation (4) shows the final reward value based on response time and resource utilization, in which the higher values of $R_{qos}$ and lower values of $R_{util}$ can increase the total reward.

$$r(s_t, a_t) = \frac{R_{qos}(rt)}{R_{util}(ut)} \tag{4}$$

Algorithm 2 shows the general procedure of *CoScal*. The algorithm initializes the monitoring model to collect system status for the RL process (line 1), including workloads level, utilization, and relevant metrics at each time interval (lines 4-5). These observations constitute a state in the RL framework. If the workload level changes (line 6), scaling approaches should be applied to ensure QoS or optimize resource usage. Thus, the $Q$-learning process will be started by choosing actions from the experience pool and transiting the current system state to another one (line 7). The actions are executed based on the Algorithm 3 (line 8). *CoScal* also supports online training, after *CoScal* transits the state $s_t$ to $s_{t+1}$, *CoScal* firstly stores the transition $(s_t, a_t, r_t, s_{t+1})$ into the experience pool. The reward $r_t$ is applied to evaluate the goodness of $(s_t, a_t)$ (lines 9-14).

**Algorithm 2:** CoScal: General Procedure

**Input** : Table $Q(s, a)$ contains all state/action pairs from experience pool by offline training, time intervals $T$, probability of random action $\epsilon$, learning rate $\alpha$, discount factor $\gamma$

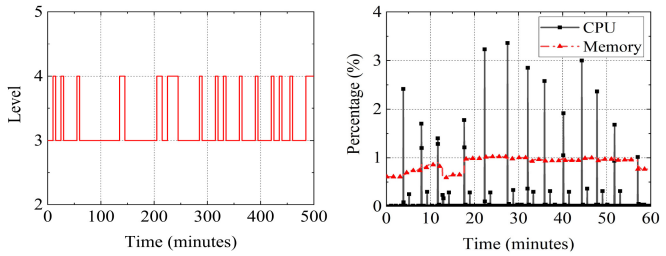
1 Initialize system status, monitoring model;
2 **for** *t from 1 to T* **do**
3  $W_{t-1} \leftarrow$ Workloads level at time interval $t-1$;
4  $W_t^p \leftarrow$ Predicted workload level according to Algorithm 1;
5  $U_k^t \leftarrow$ Resource utilization of $pm_k$ at time interval $t$;
6  **if** $W_t^p - W_{t-1} \neq 0$ **then**
7  Choose a action from action set $A$ with $\epsilon$ probability, or select an action with the $\max(Q(s_t, a_t))$;
8  Execute $a_t$ according to Algorithm 3;
9  **if** *online training is triggered* **then**
10  $s_{t+1} \leftarrow$ system state at time interval $t+1$;
11  $r_t \leftarrow$ reward calculation by Equation (4);
12  Update $Q$ value: $Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r(s_t, a_t)] + \gamma max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)]$;
13  **end**
14  Store transition $(s_t, a_t r_t, s_{t+1})$in experience pool;
15  **end**
16 **end**

**Algorithm 3:** CoScal: Action Execution

**Input** : Time interval $t$, action sets $a_t = \{a_k^i(t) | i \in \{0, 1, \ldots, I\}, k \in \{1, 2, \ldots, K\}\}$, $a_k^i(t) = \{h_k(t), v_k^i(t), b_k(t)\}$, selected action at time $t$ with horizontal scaling operation $h_k(t) = n$, vertical scaling operation $v_k^i(t) = m$, and brownout operation $b_k(t) \in \{0, 1\}$

1 **for** *k from 1 to K* **do**
2  /*Vertical scaling*/
3  **for** *i from 1 to I* **do**
4  **if** $m > 0$ **then**
5  Add $m$ resources of type $i$ on local machine;
6  **else**
7  Remove $m$ resources of type $i$ on local machine;
8  **end**
9  **end**
10 **end**
11 /*Brownout control*/
12 **if** $b_k(t) > 0$ **then**
13  Deactivate microservices via brownout;
14 **else**
15  Brownout will not be triggered;
16 **end**
17 /*Horizontal scaling*/
18 **if** $n > 0$ **then**
19  Add $n$ microservices replicates;
20 **else**
21  Remove $n$ microservices replicates;
22 **end**

Algorithm 3 shows the action execution process of *CoScal*. The system states and corresponding actions are provided by the Algorithm 2, which includes the decisions of horizontal scaling, vertical scaling, and brownout. The algorithm will first check the decision for vertical scaling. If the vertical scaling should be performed, then resources of the specific type will be added or removed (lines 1-10). After that, the brownout will be examined. If the brownout is triggered, the optional microservices are deactivated in this time interval (lines 11-16). And finally, the horizontal scaling will be checked and executed by increasing or decreasing the number of replications. We consider the execution sequence based on the execution costs




(a) The fluctuations of workloads (b) Resource usage (auto-adapter)

Fig. 5. Auto-adapter resource consumption to handle workloads fluctuations.

of different scaling techniques as discussed in Section III, as the vertical scaling can be completed within the shortest time while the horizontal scaling brings much longer processing time and communication costs (inter-process communication costs between processes on different hosts for user authentication, database access, remote function call, image retrieval and etc.) than the other two techniques as shown in the motivational example in Section III.

*Decision-making complexity of CoScal:* After the RL model is trained based on historical data, the system states and corresponding actions can be stored in a lookup table. At each iteration to access and modify the lookup table, *CoScal* applies the open addressing technique to resolve hash collisions to ensure the access and modification operations to take negligible time. The open addressing technique has the computational complexity as $O(1)$.

*Notes on model re-training:* To be noted, when microservices are updated and new optional components are added, the RL model can be re-trained to improve the performance, as the brownout part in *CoScal* is influenced by the identification of the optional parts. If the model is not re-trained, the added components will be regarded as the mandatory ones.

### D. Auto-Adapter for Unexpected Workloads Change

Although our workload prediction algorithm can achieve high accuracy, fluctuations of cloud workload level during the stable period (workload level remain unchanged) are usually unpredictable. For instance, Fig. 5(a) shows a sample of workload changes within 500 minutes. It can be observed that during the first 100 minutes, the workloads change from workload level 3 to level 4 and return to level 3 within a short time (as the workload level is the average level during a period of time, the changes represent the total amount of workloads change significantly). This often happens when there are bursts in the system. Such workload changes are difficult to be predicted due to randomness.

Microservices are more sensitive to resource fluctuations than traditional applications [10]. Therefore, bursts can cause QoS violation, resource wastage, and system overload. To address this challenge and complement the prediction algorithm, we introduce a component named *auto-adapter* that detects and adapts to these changes. The auto-adapter is integrated into the *CoScal Controller* module in Fig. 2. Auto-adapter collects the information at the first minute of each time interval and examines whether the actual workload level (collected via performance counters) equals the predicted one (via workload predictor module). If not equal, auto-adapter applies vertical scaling to optimize resource usage by adding or removing resources according to the predicted and actual workload difference.

The auto-adapter needs to be lightweight to avoid excessive resource usage incurred by the auto-adapter. We measure the resource usage in one hour caused by auto-adapter as shown in Fig. 5(b), which shows that auto-adapter only costs about a maximum of 3% extra CPU and 1% extra memory resources. This additional limited resource usage is acceptable considering its optimization effects on resource usage.

## VI. Performance Evaluation

To evaluate the performance of our proposed approach for scaling microservices, we conduct experiments in the container-based prototype system. We first present the experimental settings in Section VI-A and then introduce the baselines in Section VI-B. The results and analysis are presented in SectionVI-C.

### A. Experimental Setup

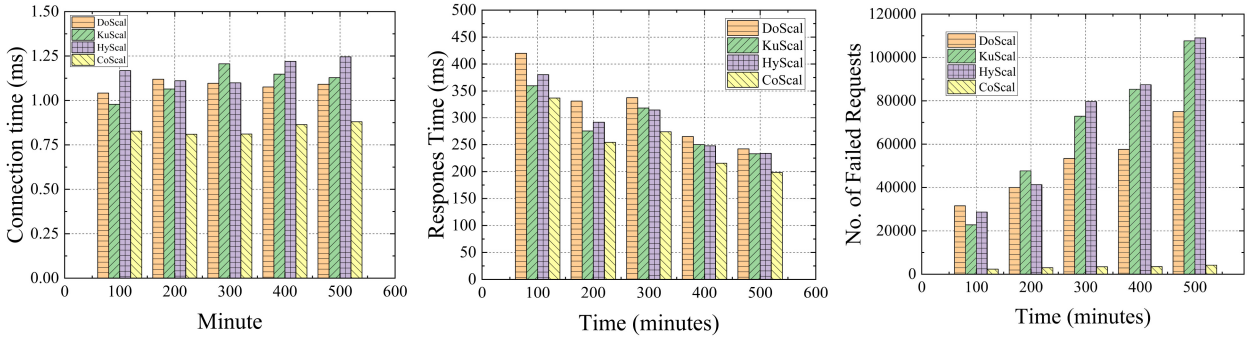
We built a cluster with heterogeneous nodes for the microservice-based application deployment. The cluster consists of four physical machines, including a machine with an Intel Core i7-9700 CPU and 16GB of RAM, two machines with Intel Core i7-4790 and 4GB of RAM, and a machine with two Intel Xeon E5-2660 CPUs and 48GB of RAM. The containers or microservices are managed through Docker Swarm, a container orchestrating tool. We deployed the Sock Shop application, a microservice-based application. The computing environment has been configured with Java (version 1.8), Python (version 3.7), and TensorFlow (version 2.2.0) development toolkits. We use the cluster-trace-v2018 of Alibaba dataset[3] that contains workloads of machines. And we use 500-minute data for evaluations.

To enable our scaling algorithms to support the brownout mechanism, we categorize the individual microservices in Sock Shop into two categories: mandatory microservices and optional microservices. Mandatory microservices include microservices that are the key components (e.g., database-related microservices) of the system, and the optional microservices (e.g., recommendation engine) can be dynamically activated or deactivated, In our testbed, the CPU can be scaled from one core to a maximum of 8 cores, memory can be scaled from 2 GB to 3.75 GB, and for horizontal scaling, the maximum number of hosts is set to 3.

### B. Baselines

We compare the performance of *CoScal* with several state-of-the-art algorithms for scaling microservices. Some baselines have been used in dominant microservice platforms, including Docker Swarm and Kubernetes.

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




(a) Average connection time (b) Average response time (c) Number of failed requests

Fig. 6. Performance Comparison of DoScal, KuScal, HyScal, and CoScal: (a) Average Connection Time. (b) Average Response Time. (c) Number of Failed Requests.

*DoScal [29]:* it is derived from the resource scaling approach implemented in native Docker Swarm, which is mainly based on vertical scaling. For instance, when one of the microservice instances is overloaded, DoScal can reallocate the microservices equipped with sufficient resources.

*KuScal [31]:* this scaling approach has been used in Kubernetes, which is mainly based on horizontal scaling that can dynamically increase or decrease the number of replicates. KuScal decides on how many replicates should be added or removed based on the resource fluctuations, e.g., CPU and memory, on the current servers.

*HyScal [8]:* it is a hybrid scaling approach for microservices. Apart from scaling CPU utilization, the HyScal algorithm also scales memory resources. The HyScal algorithm applies a similar approach to the KuScal, allocating resources and monitoring resource utilization.

### C. Experiment Analysis

The results are collected from JMeter log files, including connection time, response time, and a number of failed requests. The lower values of connection time and response time represent better QoS performance. A lower number of failed requests is desired, indicating a higher number of successful requests processed by the deployed applications.

We conducted 500-minute experiments to accurately observe and analyze the advantages and disadvantages of the scaling approaches in different periods. We demonstrated the average value of each metric with an interval of 100 minutes (i.e., 100 minutes, 200 minutes, to 500 minutes) to represent the statistical change, and the experiments are repeated 3 times to avoid randomness.

*Connection Time:* Fig. 6(a) shows the comparison of the average value of connection time during 500 minutes. The results of the baselines vary between 0.75 ms and 1.25 ms. More specifically, HyScal keeps the connection time at a high level, from 1.1 ms to 1.25 ms, and finally reaches around 1.25 ms at peak since the number of requests increases during the observed period. KuScal can achieve a better performance than HyScal with about 1.1 ms. The reason lies in that KuScal can scale resources more sufficiently than HyScal that pre-configures some limitations on resources allocated to microservices. The connection time achieved by DoScal is 1.1 ms. *CoScal* can achieve the best performance compared with the baselines, and it can reach around 0.8 ms within 500 minutes. Although the results of the *CoScal* vary slightly, the connection time is still 19.3% to 29.3% lower than the average value of the other baselines.

*Response Time:* Fig. 6(b) illustrates the comparison of the average value of response time in the first 500 minutes. The average response time values for all four scaling approaches show a decreasing trend. DoScal remains at the highest value but shows an overall decreasing trend from 419.9 ms to 242.1 ms. KuScal and HyScal show irregular fluctuations, which are caused by the different number of requests during different time periods, and their values worse than *CoScal*. For *CoScal*, it maintains the lowest response time throughout the observed time and finally drops to 198.4 ms. *CoScal* can reduce 16.1% response time compared with the baselines, which shows that combining different techniques can achieve better performance.

*Number of Failed Requests:* Fig. 6(c) demonstrates the comparison of failed requests during the experiments. The results show that *CoScal* can significantly decrease the failed requests. For instance, after 100 minutes experiments, DoScal has 31574 failed requests, HyScal brings 28666 failed requests, KuScal reduces the number to 22756, and our *CoScal* only has 2345 failed requests. For other observations with longer time, *CoScal* always has the lowest number of failed requests with 92% to 96% reduction compared with other baselines. In conclusion, *CoScal* can ensure more requests are processed successfully.

*Transaction Per Second (TPS)*: it is an important metric to that shows the number of requests per second a system can handle. The higher TPS value represents the better performance of the algorithm. As DoScal has achieved better performance than KuScal and HyScal, we compare DoScal and *CoScal* in Fig. 7. We can observe that the TPS value of CoScal is greater than DoScal. *CoScal* can handle requests stably under heavy loads, and its TPS value fluctuates slightly.

As experimental results of connection time and response time during first 200 minutes are quite close, to better demonstrate the differences between the values of the four scaling approaches, we use the Cumulative Distribution Function

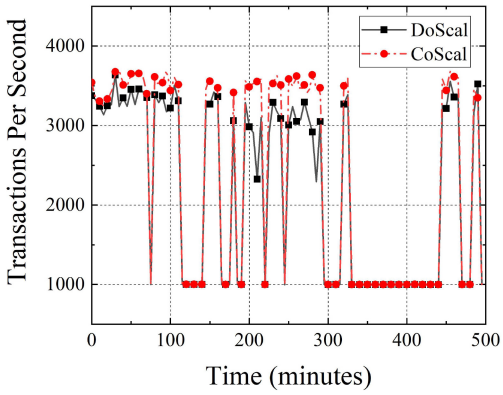



Fig. 7. Comparison of TPS for DoScal and CoScal.

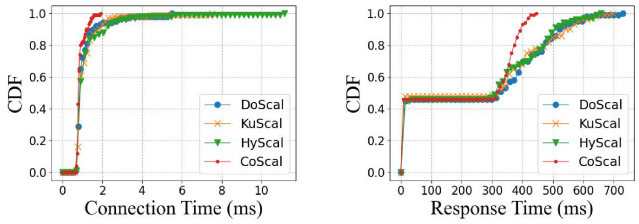



Fig. 8. CDF of connection time and response time curves.

TABLE V
CONNECTION TIME (MS) COMPARISON BASED ON CDF PERCENTILE

| Percentile | DoScal | KuScal | HyScal | CoScal |
|---|---|---|---|---|
| 20th (ms) | 0.758 | 0.780 | 0.782 | **0.752** |
| 40th (ms) | 0.805 | 0.812 | 0.828 | **0.775** |
| 60th (ms) | 0.847 | 0.887 | 0.936 | **0.807** |
| 80th (ms) | 1.208 | 1.3 | 1.241 | **0.896** |
| 95th (ms) | 2.469 | 2.129 | 2.651 | **1.407** |
| 99th (ms) | 5.336 | 4.437 | 4.614 | **1.550** |

(CDF) as the metric which is the integral of the probability density function representing the sum of the probability of occurrence of all values less than or equal to $x$. The formula of the CDF function is $F_X(x) = \mathrm{P}(X \leq x)$.

Fig. 8 shows the CDF curves of connection time and response time. There is a clear gap between the *CoScal* and the baselines. Almost all requests can establish the connection within 2 ms by *CoScal*, while other baselines can only have about 90% requests in this range. *CoScal* can respond within 400 ms for most of the requests, while other baselines need more than 500 ms. The increase in response time around 0.43 results from the change of workload level. Therefore, we can conclude that *CoScal* achieves better performance overall compared to the other baselines.

Tables V and VI also demonstrate the connection time and response time at different percentiles. From Table V, we can notice that *CoScal* outperforms other baselines during the observed period, for instance, the connection time of 95% requests falls into 1.407 ms in *CoScal*, while other baselines require 2.129 to 2.651 ms. When comparing the response time in Table VI, we can observe that although *CoScal* is not the best one before 60th percentile, it performs the best at 80th, 90th and 99th percentile. For example, the response time of

TABLE VI
RESPONSE TIME (MS) COMPARISON BASED ON CDF PERCENTILE

| Percentile | DoScal | KuScal | HyScal | CoScal |
|---|---|---|---|---|
| 20th (ms) | 3.26 | **3.22** | 3.22 | 3.38 |
| 40th (ms) | 4.25 | **4.23** | 4.30 | 4.37 |
| 60th (ms) | 378.46 | 353.99 | **336.84** | 342.46 |
| 80th (ms) | 378.46 | 476.36 | 464.52 | **370.56** |
| 95th (ms) | 567.20 | 583.55 | 561.84 | **412.70** |
| 99th (ms) | 629.21 | 641.63 | 640.09 | **425.51** |

TABLE VII
MULTI-FACETED SCALING TECHNIQUES ADOPTED UNDER DIFFERENT LOADS BY *CoScal* (THE FIRST ROW REPRESENTS THE CURRENT WORKLOAD LEVEL, AND THE FIRST COLUMN REPRESENTS THE NEXT WORKLOAD LEVEL)

| Next Level \ Current Level | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| **1** | NA | H | H | NA |
| **2** | V | NA | H | NA |
| **3** | V+B | V | NA | NA |
| **4** | V+B+H | V+B+H | V+B | NA |

80% requests are 370.56 ms in *CoScal*, while other baselines need 464.53 to 476.36 ms.

*RL Decisions Analysis:* to investigate the essential reasons for resource optimization by applying different techniques in *CoScal*, we summarize the adopted techniques by our RL agent under different workload levels as shown in Table VII, which are collected by the log files that record the adopted operations. The first row represents the original workload level and the first column represents the transferred workload level. The contents in other cells represent the techniques that have been applied, where V stands for vertical scaling, B stands for brownout, H stands for horizontal scaling, and NA means no action is conducted. For example, if the workload level needs to be transferred from level 1 to 4, vertical scaling, horizontal scaling, and brownout should be applied together, since the workloads change significantly. While if the workloads are increased from level 2 to 3, only vertical scaling is required. When the workload level is 4, the response time is high thus the reward function in Equation (4) will not guide horizontal scaling to reduce available resources. It indicates that our RL agent in *CoScal* effectively learns the environment complexity and workload characteristic and accordingly takes the suitable combinations of scaling decisions to increase application QoS and optimize resource usage of physical infrastructure.

*RL Convergence Analysis:* to observe the convergence behavior of our RL-based approach, we train the RL model following the same steps of resource scaling according to Section IV-C. Fig. 9 shows that the approach updates its scaling policy as the training progresses and improves the total reward value. As the RL agent spends initially in the exploration phase, it exponentially increases the reward. After the 16522 epochs, it converges to a stable reward.

### D. Experimental Analysis With Stan's Robot Shop

To further evaluate the effectiveness of *CoScal*, we also conduct experiments with another Docker-based microservice

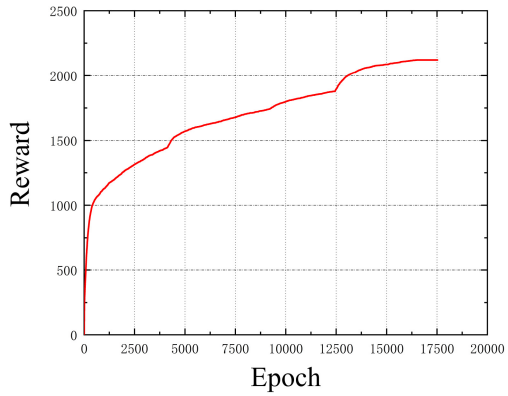



Fig. 9. Reward value during learning process.

TABLE VIII
EXPERIMENTAL RESULTS WITH STAN'S ROBOT SHOP

| Metrics | DoScal | KuScal | HyScal | CoScal |
|---|---|---|---|---|
| **TPS** | 909 | 915 | 882 | **980** |
| **Successful TPS** | 825 | 841 | 815 | **892** |
| **No. of Failures per Second** | 74 | 74 | **68** | 88 |
| **Response Time (ms)** | 45.1 | **43.8** | 61.4 | 47.2 |

application, named Stan's Robot Shop,[4] which is an online system composed of 12 microservices for robots sale. We use Locust,[5] a Python-based testing tool similar to JMeter, to generate requests based on Alibaba's trace [14], and send the requests to the homepage of Stan's Robot Shop. The experiments are conducted in a Docker Swarm cluster with 5 homogeneous machines equipped with Intel Xeon CPU E5-2630 v3 and 64GB RAM.

We compare several metrics in Table VIII collected from Locust, including TPS, successful TPS, response time and the number of failed requests, which have also been evaluated in our previous experiments with Sock Shop application. The final results of each approach are the average value of 300 minutes, and the experiments have been repeated 3 times to avoid randomness. The results show that *CoScal* can improve 7%-11% TPS compared with the other three baselines, and *CoScal* can achieve the highest successful TPS. In terms of the number of failures, *CoScal* is very close to the best result of HyScal. Moreover, *CoScal* can obtain the lower response time than HyScal, and achieve close results with KuScal. In conclusion, the experiments with Stan's Robot Shop application also validate the effectiveness of our proposed approach in improving system performance. *CoScal* can achieve good performance as it is adaptive to the change of environments and it searches large solution space.

## VII. CONCLUSION AND FUTURE WORK

The microservice-based applications have been widely adopted in cloud computing environments, converting monolithic applications into lightweight, flexible, and loosely-coupled application components. Such a modular design of the application helps achieve CI/CD (continuous integration, continuous delivery, and continuous deployment) of the application life cycle in cloud environments. However, efficient algorithms for resource scaling in cloud infrastructure are required to ensure the sustainable development of microservice technology. In this paper, we proposed a multi-faceted scaling approach, named *CoScal*, for scaling resources for microservices that combine the three techniques, including horizontal scaling, vertical scaling, and brownout. It ensures cloud service providers optimize their resource usage while guaranteeing QoS. *CoScal* utilizes deep learning approaches for workload prediction to predict load more accurately compared to traditional regression-based prediction approaches. Furthermore, leveraging the RL framework takes decisions on scaling strategies, adapting to the load changes, and allocating the appropriate resources. To evaluate the performance of *CoScal*, we deployed our algorithms on a prototype system with representative microservices application. The results are compared with popular state-of-the-art algorithms from research and industry domains with two microservice applications. The experimental results have demonstrated that *CoScal* can outperform the baseline algorithms in connection time, response time and number of failed requests for Sock Shop application, and improve the number of successful transactions with 6%-10% for Stan's Robot Shop application with slight degradation in response time.

As for future work, we would like to investigate network resource scaling to extend to the suitable scenarios of our scaling approach. As the size of Q-table can increase significantly when the state-action space increases largely, we would also like to investigate Deep Q-Learning with workloads prediction to improve the speed and accuracy of inference, which can also address the limitation in Q-Learning based approach that heavily relies on Q-table. Moreover, we plan to migrate our system to a Kubernetes-based platform to evaluate medium-scale microservice applications, such as TrainTicket and Hotel Reservation applications.

## SOFTWARE AVAILABILITY

The source codes of *CoScal* project can be found at the GitHub repository: https://github.com/Kminassch/CoScal

## ACKNOWLEDGMENT

The authors would like to thank Dr. Zhicheng Cai for his suggestions to improve this work.

## REFERENCES

[1] M. Armbrust *et al.*, "A view of cloud computing," *Commun. ACM*, vol. 53, no. 4, pp. 50–58, 2010.
[2] M. Kumar, S. Sharma, A. Goel, and S. Singh, "A comprehensive survey for scheduling techniques in cloud computing," *J. Netw. Comput. Appl.*, vol. 143, pp. 1–33, Oct. 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1084804519302036
[3] S. Newman, *Building Microservices*. Sebastopol, CA, USA: O'Reilly Media, 2015.
[4] F. Al-Doghman, N. Moustafa, I. Khalil, Z. Tari, and A. Zomaya, "AI-enabled secure microservices in edge computing: Opportunities and challenges," *IEEE Trans. Services Comput.*, early access, Mar. 1, 2022, doi: 10.1109/TSC.2022.3155447.
[5] M. Kumar, A. Kishor, J. Abawajy, P. Agarwal, A. Singh, and A. Zomaya, "ARPS: An autonomic resource provisioning and scheduling framework for cloud platforms," *IEEE Trans. Sustain. Comput.*, vol. 7, no. 2, pp. 386–399, Apr.–Jun. 2022.

[4]https://www.instana.com/blog/stans-robot-shop-sample-microservice-application/

[5]https://locust.io/

[6] J. Son and R. Buyya, "SDCon: Integrated control platform for software-defined clouds," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 1, pp. 230–244, Jan. 2019.

[7] M. S. Hasan, F. Alvares, T. Ledoux, and J.-L. Pazat, "Investigating energy consumption and performance trade-off for interactive cloud application," *IEEE Trans. Sustain. Comput.*, vol. 2, no. 2, pp. 113–126, Apr.–Jun. 2017.

[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 39th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, 2019, pp. 80–90.

[9] L. Toka, G. Dobreff, B. Fodor, and B. Sonkoly, "Machine learning-based scaling management for kubernetes edge clusters," *IEEE Trans. Netw. Service Manag.*, vol. 18, no. 1, pp. 958–972, Mar. 2021.

[10] 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 *Proc. 14th USENIX Symp. Oper. Syst. Design Implement. (OSDI)*, Nov. 2020, pp. 805–825. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/qiu

[11] L. Tomás, C. Klein, J. Tordsson, and F. Hernández-Rodríguez, "The straw that broke the camel's back: Safe cloud overbooking with application brownout," in *Proc. IEEE Int. Conf. Cloud Auton. Comput.*, 2014, pp. 151–160.

[12] M. Xu, A. N. Toosi, and R. Buyya, "A self-adaptive approach for managing applications and harnessing renewable energy for sustainable cloud computing," *IEEE Trans. Sustain. Comput.*, vol. 6, no. 4, pp. 544–558, Oct.–Dec. 2021.

[13] Z. Zhong, M. Xu, M. A. Rodriguez, C. Xu, and R. Buyya, "Machine learning-based orchestration of containers: A taxonomy and future directions," *ACM Comput. Surveys*, vol. 54, no. 10S, p. 217, Jan. 2022. [Online]. Available: https://doi.org/10.1145/3510415

[14] W. Chen, K. Ye, Y. Wang, G. Xu, and C. Xu, "How does the workload look like in production cloud? Analysis and clustering of workloads on Alibaba cluster trace," in *Proc. IEEE 24th Int. Conf. Parallel Distrib. Syst. (ICPADS)*, 2018, pp. 102–109.

[15] R. Nishtala, P. Carpenter, V. Petrucci, and X. Martorell, "Hipster: Hybrid task manager for latency-critical cloud workloads," in *Proc. IEEE Int. Symp. High Perform. Comput. Architecture (HPCA)*, 2017, pp. 409–420.

[16] R. Nishtala, V. Petrucci, P. Carpenter, and M. Sjalander, "Twig: Multi-agent task management for colocated latency-critical cloud services," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, 2020, pp. 167–179.

[17] L. Liu, "QoS-aware machine learning-based multiple resources scheduling for Microservices in cloud environment," 2019, *arXiv:1911.13208*.

[18] M. A. Chang, A. Panda, Y.-C. Tsai, H. Wang, and S. Shenker, "ThrottleBot-performance without insight," 2017, *arXiv:1711.00618*.

[19] H. Khazaei, C. Barna, and M. Litoiu, "Performance modeling of microservice platforms considering the dynamics of the underlying cloud infrastructure," 2019, *arXiv:1902.03387*.

[20] Y. Gan *et al.*, "Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices," in *Proc. 24th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2019, pp. 19–33. [Online]. Available: https://doi.org/10.1145/3297858.3304004

[21] R. S. Kannan, L. Subramanian, A. Raju, J. Ahn, J. Mars, and L. Tang, "GrandSLAm: Guaranteeing SLAs for jobs in Microservices execution frameworks," in *Proc. 14th EuroSys Conf.*, 2019, p. 34. [Online]. Available: https://doi.org/10.1145/3302424.3303958

[22] G. Yu, P. Chen, and Z. Zheng, "Microscaler: Automatic scaling for Microservices with an online learning approach," in *Proc. IEEE Int. Conf. Web Services (ICWS)*, Jul. 2019, pp. 68–75.

[23] L. Suresh, P. Bodik, I. Menache, M. Canini, and F. Ciucu, "Distributed resource management across process boundaries," in *Proc. Symp. Cloud Comput.*, 2017, pp. 611–623. [Online]. Available: https://doi.org/10.1145/3127479.3132020

[24] H. Zhou *et al.*, "Overload control for scaling WeChat Microservices," in *Proc. ACM Symp. Cloud Comput.*, 2018, pp. 149–161. [Online]. Available: https://doi.org/10.1145/3267809.3267823

[25] K. Rzadca *et al.*, "Autopilot: Workload autoscaling at Google," in *Proc. 15th Eur. Conf. Comput. Syst.*, 2020, pp. 1–16. [Online]. Available: https://doi.org/10.1145/3342195.3387524

[26] X. Hou, C. Li, J. Liu, L. Zhang, Y. Hu, and M. Guo, "ANT-Man: Towards agile power management in the microservice era," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal. (SC)*, 2020, pp. 1098–1111.

[27] 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 *Proc. Int. Conf. Service-Oriented Comput.*, 2019, pp. 555–560.

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

[29] F. Rossi, M. Nardelli, and V. Cardellini, "Horizontal and vertical scaling of container-based applications using reinforcement learning," in *Proc. IEEE 12th Int. Conf. Cloud Comput. (CLOUD)*, 2019, pp. 329–338.

[30] A. U. Gias, G. Casale, and M. Woodside, "ATOM: Model-driven autoscaling for microservices," in *Proc. IEEE 39th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, 2019, pp. 1994–2004.

[31] B. Burns, J. Beda, and K. Hightower, *Kubernetes: Up and Running: Dive into the Future of Infrastructure*. Sebastopol, CA, USA: O'Reilly Media, 2019.

[32] T. Kiss *et al.*, "MICADO—Microservice-based cloud application-level dynamic orchestrator," *Future Gener. Comput. Syst.*, vol. 94, pp. 937–946, May 2019.

[33] X. Zhou *et al.*, "Poster: Benchmarking microservice systems for software engineering research," in *Proc. IEEE/ACM 40th Int. Conf. Softw. Eng. Compan. (ICSE-Companion)*, 2018, pp. 323–324.

[34] M. Abadi *et al.*, "TensorFlow: A system for large-scale machine learning," in *Proc. 12th USENIX Symp. Oper. Syst. Design Implement. (OSDI)*, Nov. 2016, pp. 265–283.

[35] "Sock shop: A microservices demo application." 2017. [Online]. Available: https://microservices-demo.github.io/

[36] S. Kardani-Moghaddam, R. Buyya, and K. Ramamohanarao, "ADRL: A hybrid anomaly-aware deep reinforcement learning-based resource scaling in clouds," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 3, pp. 514–526, Mar. 2021.

[37] A. Mendoza, "Using NMON to monitor SAS applications on AIX servers," in *Proc. SAS Global Forum*, 2008, pp. 16–19.

[38] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," in *Proc. Workshop Deep Learn.*, Dec. 2014, pp. 1–9.

[39] T. Mikolov, S. Kombrink, L. Burget, J. Černocký, and S. Khudanpur, "Extensions of recurrent neural network language model," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, 2011, pp. 5528–5531.

[40] Z. Chen, J. Hu, G. Min, A. Y. Zomaya, and T. El-Ghazawi, "Towards accurate prediction for high-dimensional and highly-variable cloud workloads with deep learning," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 4, pp. 923–934, Apr. 2020.

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

[42] G. Yin, C. Z. Xu, and L. Y. Wang, "Q-learning algorithms with random truncation bounds and applications to effective parallel computing," *J. Optim. Theory Appl.*, vol. 137, pp. 435–451, May 2008.

[43] S. Levine, A. Kumar, G. Tucker, and J. Fu, "Offline reinforcement learning: Tutorial, review, and perspectives on open problems," 2020, *arXiv:2005.01643*.


**Minxian Xu** (Member, IEEE) received the B.Sc. and M.Sc. degrees in software engineering from the University of Electronic Science and Technology of China in 2012 and 2015, respectively, and the Ph.D. degree from the University of Melbourne in 2019. He is currently an Associate Professor with the Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences. He has coauthored 40+ peer-reviewed papers published in prominent international journals and conferences, such as *ACM Computing Surveys*, *ACM Transactions on Internet Technology*, IEEE TRANSACTIONS ON SUSTAINABLE COMPUTING, IEEE TRANSACTIONS ON CLOUD COMPUTING, IEEE TRANSACTIONS ON AUTOMATION SCIENCE AND ENGINEERING, IEEE TRANSACTIONS ON GREEN COMMUNICATIONS AND NETWORKING, *Journal of Parallel and Distributed Computing*, *Journal of Systems and Software*, and ICSOC. His research interests include resource scheduling and optimization in cloud computing. His Ph.D. Thesis was awarded the 2019 IEEE TCSC Outstanding Ph.D. Dissertation Award. More information can be found at: minxianxu.info.

**Chenghao Song** received the B.Sc. degree from the University of Electronic Science and Technology of China. He is currently pursuing the master's degree with the University of Melbourne. He is also a visiting student with the Shenzhen Institutes of Advanced Technology, Chinese Academy of Science. His research interest includes deep learning for cloud resource optimization.

**Shashikant Ilager** received the Ph.D. degree from the University of Melbourne in 2021. He is a Postdoctoral Researcher with the Vienna University of Technology, Austria. He published several papers in leading journals and conferences, including IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, IEEE TRANSACTIONS ON MOBILE COMPUTING, and IEEE/ACM INTERNATIONAL SYMPOSIUM ON CLUSTER, and *Cloud and Internet Computing*. His research interests covers the boundaries of large scale distributed systems and machine learning. He was a recipient of the Best Paper Award from CCGRID 2020 Conference.

**Sukhpal Singh Gill** is a Lecturer (Assistant Professor) of Cloud Computing with the School of Electronic Engineering and Computer Science, Queen Mary University of London, U.K. Prior to his present stint, he has held positions as a Research Associate with the School of Computing and Communications, Lancaster University, U.K., and also as a Postdoctoral Research Fellow with CLOUDS Laboratory, The University of Melbourne, Australia. He is serving as an Associate Editor in *Transactions on Emerging Telecommunications Technologies* (Wiley) and *IET Networks Journal*. His research interests include cloud computing, fog computing, Internet of Things, and energy efficiency. For further information, please visit: http://www.ssgill.me.

**Juanjuan Zhao** received the M.S. degree from the Department of Computer Science, Wuhan University of Technology in 2009, and the Ph.D. degree from the Shenzhen College of Advanced Technology, University of Chinese Academy of Sciences in 2017, where she is an Associate Professor. Her research topics include data-driven urban systems, mobile data collection, cross-domain data fusion, and heterogeneous model integration.

**Kejiang Ye** (Member, IEEE) received the B.Sc. and Ph.D. degrees in computer science from Zhejiang University in 2008 and 2013, respectively. He was also a joint Ph.D. student with the University of Sydney from 2012 to 2013. He works as a Postdoctoral 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 Institutes of Advanced Technology, Chinese Academy of Science. His research interests focus on the performance, energy, and reliability of cloud computing and network systems.

**Chengzhong Xu** (Fellow, IEEE) received the B.Sc. and M.Sc. degrees in computer science and engineering from Nanjing University in 1986 and 1989, respectively, and 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, and a Chair Professor of Computer and Information Science. He published two research monographs and more than 300 peer-reviewed papers in journals and conference proceedings; his papers received about 10K citations with an H-index of 59. His main research interests lie in parallel and distributed computing and cloud computing, in particular, with an emphasis on resource management for system's performance, reliability, availability, power efficiency, and security, and in big data and data-driven intelligence applications in smart city and self-driving vehicles. He serves or served on a number of journal editorial boards, including IEEE TRANSACTIONS ON COMPUTERS, IEEE TRANSACTIONS ON CLOUD COMPUTING, IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, and *Journal of Parallel and Distributed Computing*. He has been the Chair of IEEE Technical Committee on Distributed Processing since 2015.