THE LATENCY-AWARE RESOURCE PROVISIONING FOR MICROSERVICE CLUSTER

COMP90055 RESEARCH PROJECT

Chenghao Song

Student ID: 1144743

Supervisor: Prof. Rajkumar Buyya

External Supervisor: Dr. Minxian Xu

A project report submitted for the 25-pts Research Project COMP90055

and degree of Master of Information Technology (Artificial Intelligence)

in the School of Computing and Information Systems

The University of Melbourne, Australia

May 2022

Contents

List of Tables

List of Figures

Abstract

As an important part of the new generation of information technology, user-oriented cloud computing-related services offer many conveniences to a wide range of users. The User-oriented microservices are typically supported by distributed clusters of servers. The service users are highly sensitive to server latency when using these services because server processing latency is directly related to the QoS(quality of services) of server cluster. The time delay within a microservice cluster is usually caused by the competition for resources generated by the microservices within the server cluster. To alleviate this problem, we propose a reinforcement learning based algorithm that makes different decisions (including horizontal scaling, vertical scaling, and brownout methods) to reallocate resources within the machine under different server load scenarios. The aim of the research project is to use this cluster load scheduling algorithm to address the impact of chain latency of microservices present in cloud servers.

In general, we need to face the problem of how to find the maximum critical chains and nodes of microservices and what scaling methods are needed to solve the time delay of microservice nodes.

Keywords Cloud computing, Microservice, Machine learning, Kubernetes

Declaration

I certify that

- *this thesis does not incorporate without acknowledgement any material previously submitted for a degree or diploma in any university; and that to the best of my knowledge and belief it does not contain any material previously published or written by another person where due reference is not made in the text.*

- *the thesis is 9910 words in length (excluding text in images, table, bibliographies and appendices).*

Signature: Chenghao Song

Date: 26 May 2022

ACKNOWLEDGEMENTS

First of all, I would like to thank my supervisor, Prof. Rajkumar Buyya, for his patience and guidance from the very beginning when I chose the topic of this article to the completion of the whole report and oral presentation. I would like to express my sincere respect and gratitude to Professor Buyya.

During my experiments, Dr. Minxian Xu gave me a lot of academic guidance and help in my life, he is a rigorous scholarship and has a lot of experimental experience. From the initial inspiration to the completed report, Dr. Xu has provided me with encouragement and guidance along the way. I would like to express my respect and gratitude to Dr. Xu. In addition to this, I also received assistance from SIAT (Shenzhen Institute of Advanced Technology), which provided me with the machines I needed for my experiments. In addition, I was a visiting student of Dr. Xu at SIAT. I didn't choose to go to the University of Melbourne offline because of COVID-19, and SIAT was like another campus for me. As well as the partners I met at SIAT, we often discuss computer-related information among ourselves and exchange ideas with each other, which enriches my daily life.

The support of my family and my friends has been part of what has enabled me to continue my studies. I would like to thank my family for their unconditional support of my studies and my friends for their encouragement. I would like to thank everyone who has supported me and encouraged me throughout my Master's career.

1 Introduction

As an important part of the new era of information technology, user-oriented cloud computing-related services offer many conveniences to a wide range of users. Nowadays, our lives are filled with user-oriented online services, such as the online shopping platform Amazon[1], the online chat platform Facebook[2], the online search engine Google[3], etc. These large user-oriented platforms usually consist of several microservices, which are usually supported by a distributed cluster of cloud servers and can be deployed in containers or virtual machines.

Users are often highly sensitive to server chain latency when using these services (Gan et al., 2019), as server chain latency, is directly related to the quality of service of the server and thus affects the user experience. Chain latency within a microservice cluster is caused by the microservices within the server cluster previously competing for each other's resources (Niu et al., 2018). The microservice scheduling algorithms for cloud server clusters have become one of the most important metrics for measuring the load capacity of cloud servers under highly variable workloads. A good microservice chain scheduling algorithm can reduce the waste of server resources for cloud server providers, as well as greatly improve the user experience of users, improve the stability of cloud servers, make the resource allocation of microservices in cloud servers more reasonable (Dragoni et al., 2017), and improve the robustness of microservices in the whole cloud server cluster.

The microservices are now typically deployed in virtual machines or containers in a cloud server cluster. Compared to virtual machines, containers are faster to start and the performance of the service is close to native, and a larger number of containers can be deployed simultaneously on a single physical machine (Salah et al., 2017). These advantages make container technology widely used in cloud computing and microservices. The introduction of container technology has solved the problem of slow and long interaction times caused by virtual machines, which affects the user experience (Barik et al., 2016). At the same time, container technology has become the choice of many cloud service providers because of its rapid deployment ability, scalability, and cross-platform compatibility (Potdar et al., 2020).

[1]Amazon: A online shopping platforms with free shipping on millions of items. https://www.amazon.com/
[2]Facebook: Social Media. https://www.facebook.com/
[3]Google: Online search engine. https://www.google.com

Kubernetes has become one of the most popular platforms for container-based microservices (Bernstein, 2014), and the container orchestration technology provides is facilitation to automate container management and scaling. It supports horizontal scaling based on usage thresholds. Threshold-based container scheduling algorithms can perform scheduling tasks pretty well in stable, less variable operating environments, but can only achieve the sub-optimal solutions in more complex workload conditions, such as in large user-facing online platforms, where the server load often rises or falls due to unexpected situations. To address this pain point, many researchers have proposed several proven algorithms. For example, Kubernetes default-based auto-scaling algorithms have been proposed, and Autopilot from Google (Reiss et al., 2012) has proposed scheduling algorithms based on threshold and historical load, based on the cluster trace record of the last few minutes. However, most of these algorithms focus on a single type of resource, such as CPU, memory, etc. In addition, many research groups propose algorithms that apply horizontal scaling (Nguyen et al., 2020) and vertical scaling (Balla et al., 2020). There are also many teams attempting to build on this by using a combination of horizontal and vertical scaling for scheduling clusters (Rossi et al., 2019a) and received some great effects. Hyscal (Kwan et al., 2019) is a hybrid scaling approach for microservices based on Kubernetes (Burns et al., 2019). Apart from scaling CPU utilization, the HyScal algorithm also scales memory resources.

To address the problem of high chain latency of microservices in cloud servers, this report proposes an efficient scheduling algorithm based on reinforcement learning for online adjustment in different cloud server load profiles, using methods including horizontal scaling, vertical scaling, and brownout method to reallocate resources within the cluster.

Horizontal scaling adjusts resources by adding or removing replicates of microservices on cloud servers within a cluster to improve resource usage and system availability. Usually increasing the number of replicates for a server can effectively alleviate overload conditions due to the pressure of transiently increasing server requests.

Vertical scaling adjusts the processing service capacity of the current cloud server by regulating the amount of resource regulation such as CPU, memory, network resources, and hard disk resources allocated to microservice instances. Cloud server clusters can achieve dynamic scheduling and resource allocation by increasing the resources of a single cloud server or multiple cloud servers within the cluster.

The existing vertical and horizontal adjustment mechanisms can adjust the workload size of the cloud server cluster to some extent, but both methods have a significant drawback that it is difficult to cope with the high-dimensional, high-variability cloud server workload change situation. Although based on the ability to give it some advance judgment, it is still difficult to cope with many unexpected events caused by instantaneous changes in cloud server workloads and other situations. To address the above drawbacks, this paper proposes a cloud server cluster load scheduling strategy based on the Brownout method, which dynamically turns on and off optional application components to alleviate the overload situation and achieve the purpose of reducing the cloud server cluster load according to the operation status of the system. The optional application components are usually some class of formation that are distinct from the core components of the overall system, and shutting them down in a short period does not affect the normal operation of the whole system.

Chain Latency-based Resources Provisioning for Cluster (CLRPC). CLRPC is a chain latency-based microservice resource scheduling system consisting of machine learning and reinforcement learning algorithms and scheduling policies, which covers a complete set of architectures from simulating user behavior to performing scheduling on a cluster of cloud servers. The three main contributions are summarized below:

- A cloud server load generation component based on Locust that simulates user behavior.
- A decision tree based machine learning model for determining critical chains and critical nodes for microservice chain latency.
- A reinforcement learning based cloud server scheduling algorithm for selecting the optimal solution of a scheduling policy under different states.
- A number of comprehensive experiments to evaluate the effectiveness of our proposed approach.

2 Background

Many of the existing mainstream microservice resource scheduling methods, while capable of simple scheduling of microservices in resource-sufficient situations, have difficulty handling resource contention in highly latency-sensitive cloud server clusters. According to the resource allocation scheduling algorithm, there are often dozens of containers running on a single node at the same time in a cloud server cluster, which leads to resource contention as the resources of these microservices are not allocated according to the pre-defined resources. This scheduling algorithm is not applicable in the current cloud server cluster architecture. According to the longest chain-based microservice scheduling algorithm, if the critical node of a microservice is not on the longest chain, the resources are not correctly allocated to the critical node, which results in microservice latency problems in the cloud server not being resolved promptly.

2.1 Related Work

Resource allocation scheduling algorithm: The resource allocation scheduling algorithm makes use of the pre-defined resource allocation to the microservice itself to make the allocation. This can achieve better results when resources are sufficient. In this resource allocation method, the competition for resources between microservices can be minimized and the stability of the cloud server can be improved to a certain extent.

Pod level scheduling algorithm: The algorithm focuses on obtaining the resource utilization of the dynamically changing containers and then ranking them in order and allocating resources according to that order. This method of resource allocation, due to its dynamic properties, allows for timely scheduling in response to changes in cloud server workload. This algorithm is often used in edge computing because there is no need to do much tuning, just to obtain the change in the number of containers/pods as a granularity (Toka et al., 2021).

Autopilot scheduling algorithm: Autopilot scheduling algorithm is a container scheduling algorithm based on Kubernetes (Rzadca et al., 2020b). This approach first requires collecting historical data information about the servers and then choosing two feasible methods to perform scheduling operations. One of them is a sliding window algorithm based on historical data, and the other is a meta-algorithm based on ideas borrowed from reinforcement learning, both of which aim to enable the scheduling of servers when the predicted results of the algorithm exceed a threshold.

Longest chain-based microservice scheduling algorithm: The longest chain-based microservice scheduling algorithm uses the chain relationship of microservices to allocate resources. For microservices in a cluster, the longest chain or chains can be obtained by analyzing their chain relationships, and resources are allocated based on this criterion. For all microservices on the microservice chains, the longest chain will be allocated resources first, then in order from the longest chain to the shortest chain. This method of resource allocation alleviates the competition for resources between microservices to a certain extent and prioritizes more resources for longer chains to reduce the overall time delay of microservices. The FIRM (Qiu et al., 2020a) uses this method and uses SVM(support vector machines) as their decision model, which receives great feedback from the scaled cluster.

2.2 Comparison with our approach

To overcome the above shortcomings, this project proposes a microservice scheduling algorithm for cloud server clusters based on reinforcement learning and chain latency of microservices, which uses the maximum latency chains in the microservice system to determine the location of critical nodes.

Compared with the Resource allocation scheduling algorithm, benefiting from the reinforcement learning model we use, the scheduling algorithm we use can make more reasonable scheduling measures in response to high latitude and high variability load states. Our algorithm also schedules the server when the load on the server is relatively stable, ensuring that it does not consume/waste too many resources.

Compared with the Pod level scheduling algorithm, our approach is not to focus on all Pods in a microservice system, but on those critical nodes on the critical path. In a highly variable load state, it would be time-consuming and wasteful of computing resources to monitor and schedule every microservice in real-time. Our approach cleverly solves this problem by not only optimizing the speed of microservice selection but also by preventing the high variability that causes too frequent scheduling from increasing the server's computational resource consumption.

Compared with the Autopilot scheduling algorithm, Autopilot's approach is based on historical data over a period, which is processed by a sliding window algorithm to get a prediction for the next time slice, and a threshold-based scheduling mechanism is set. The sliding window algorithm is difficult to cope with rapid changes in load conditions, as the average of adjacent workload records may be stable but with a large variance. The stable workload might not be detected by the threshold-based scheduling algorithm and the corresponding scheduling policy will not be triggered. Our approach can avoid this situation, by using the Approximate Q-Learning which always makes similar decisions in similar states (Kumar et al., 2019). This ensures that the decisions made in different situations must be specific to that situation, e.g. the scheduling policy for an overloaded server during training is similar to an overload situation in the real world.

Compared with the Longest chain-based microservice scheduling algorithm, the biggest difference between the two is the approach to selecting critical chains and critical nodes. Our approach is to select the chain with the highest latency by comparing the latency of the microservices-based chain, whereas the Longest chain-based microservice scheduling is to select the longest chain of all microservice chains, and if two of the same length are encountered, to select the last traversed chain. Although the scheduling strategies of these two algorithms are very similar, using both vertical and horizontal scaling, the actual results can be slightly different. For example, when the total latency of the longest chain in a microservice chain is relatively low, and the node with the highest latency is not on the longest chain, the two policies may vary. Our approach would be based on the maximum latency node on the maximum latency chain, i.e. the critical node, whereas the Longest chain-based microservice scheduling would be based on the "critical node" on the longest chain. Another feature is that our algorithm uses the Brownout method to schedule servers under overload, in addition to vertical and horizontal scaling.

2.3 Critical Analysis

Kubernetes Burns et al. (2019) and Docker Swarm has become the dominant systems for managing microservice. However, the scaling techniques in these systems are primarily threshold-based or static policies. Kiss et al. Kiss et al. (2019) implemented a generic microservice orchestration platform for heterogeneous cloud clusters, which can auto-scale resources without binding to specific applications. Zhou et al. Zhou et al. (2018b) 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 train ticket. Xu et al. Xu et al. (2021) 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

Table 1: Comparison of related work

Approach	Scaling Techniques			workload prediction		Scheduling Policy	
	Vertical	**Horizontal**	**Brownout**	**Linear**	**Non-Linear**	**Heuristic**	**Supervised Learning**
Liu et al. Liu (2019)	√				√		√
Qiu et al. Qiu et al. (2020b)	√	√					√
Kannan et al. Kannan et al. (2019)	√			√		√	
Yu et al. Yu et al. (2019)		√			√	√	
Zhou et al. Zhou et al. (2018a)	√			√		√	
Rzdaca et al. Rzadca et al. (2020a)	√	√		√		√	√
Kwan et al. Kwan et al. (2019)	√	√				√	
Hou et al. Hou et al. (2020)	√			√		√	
Zhang et al. Zhang et al. (2020)		√		√			√
Rossi et al. Rossi et al. (2019b)	√	√					√
Gias et al. Gias et al. (2019)		√					√
Xu et al. Xu et al. (2021)		√	√	√		√	
CLRPC (Our Approach)	√	√	√		√		√

to manually tune various configurations, especially in an environment with high dynamics Rossi et al. (2019b). Unlike these approaches, we utilize the RL-based approach with an adaptive nature to learn and make good decisions via interactions with the environment.

This paper contributes to the growing body of research related to the microservice area. The Table 1 compares the proposed approach **(CLRPC)** with related works based on the scaling techniques, workload prediction techniques, and type of scheduling algorithms. Given the contribution 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 provide 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 (GRU) Chung et al. (2014) for accurate workload prediction and apply an RL-based algorithm for resource management to offer flexible adaption for a highly dynamic environment.

3 System model

In this section, we introduce our system model, which contains the dataset we use, the platform we use to test our algorithm, and the architecture of our system.

3.1 Dataset

The dataset we use is the workload dataset cluster-trace-v2018 [4] from Alibaba, 2018, which contains workload data for a span of 8 days, generated from 4,034 homogeneous servers Chen et al. (2018). Each of these servers has 96 CPU cores and their memory size is the same. We selected machine_usage.csv from this dataset and analysed it to obtain a mapping between CPUs and the number of requests to be sent by training a deep learning model.

3.2 Microservices platforms

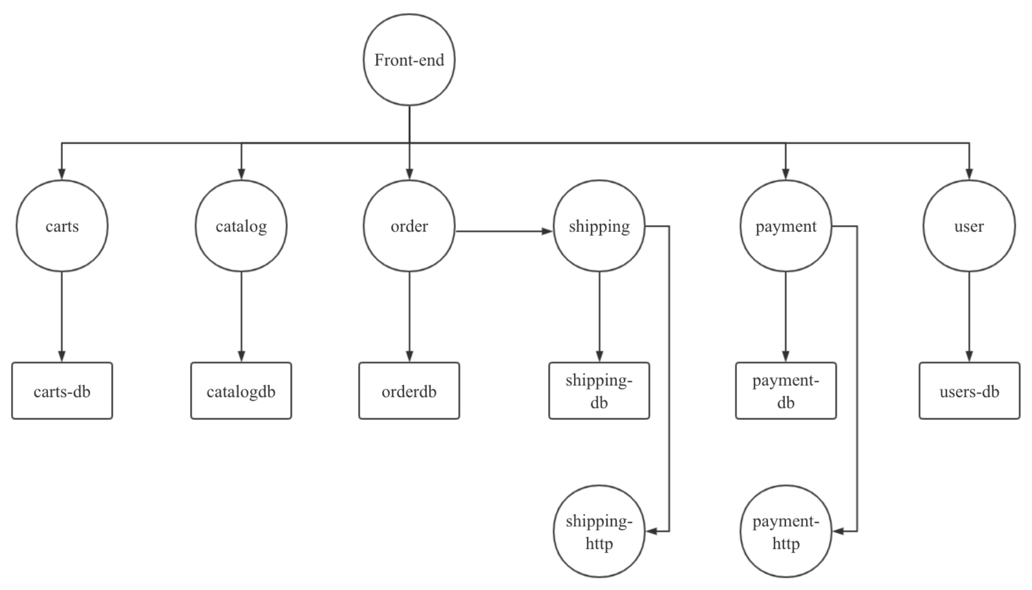

Figure 1: Sock shop dependency model

In this report, the microservices platform we use is the Sock Shop [5], which simulates the user-facing part of an e-commerce website selling socks. Sock Shop's microservices are designed to have minimal expected resource allocation, i.e. the quota for each microservice is as small as possible when it is initialized. The dependency model of Sock Shop shows in Fig.1. The Sock Shop is designed to help developers use it for demonstrating and testing microservices and cloud-native technologies. DNS is used between microservices to find other related connected microservices. A load balancer or service router can be inserted throughout the microservices framework depending on our needs when testing the scheduling of cloud servers. The version of Sock Shop used in this patent is the official Kubernetes-based version, which includes a Jaeger-based microservice monitoring

[4]Alibaba Cluster Trace Program: https://github.com/alibaba/clusterdata/tree/v2018

[5]Sock Shop: A Microservices Demo Application. https://microservices-demo.github.io/

component in addition to the basic microservice components.

In the Sock Shop system model, a database per service architecture is used to ensure that each microservice is loosely coupled, thus ensuring that each service does not need to wait when using the database because other microservices are occupying it, and making it easier for developers to schedule it. Each microservice chain in Sock Shop, such as carts-cartsdb, catalog-catalogdb, etc., is directly connected to the front-end microservice. These corresponding microservice nodes will exist workload when users visit the home page or access the corresponding shopping cart interface, etc.

3.3 Architecture of the system

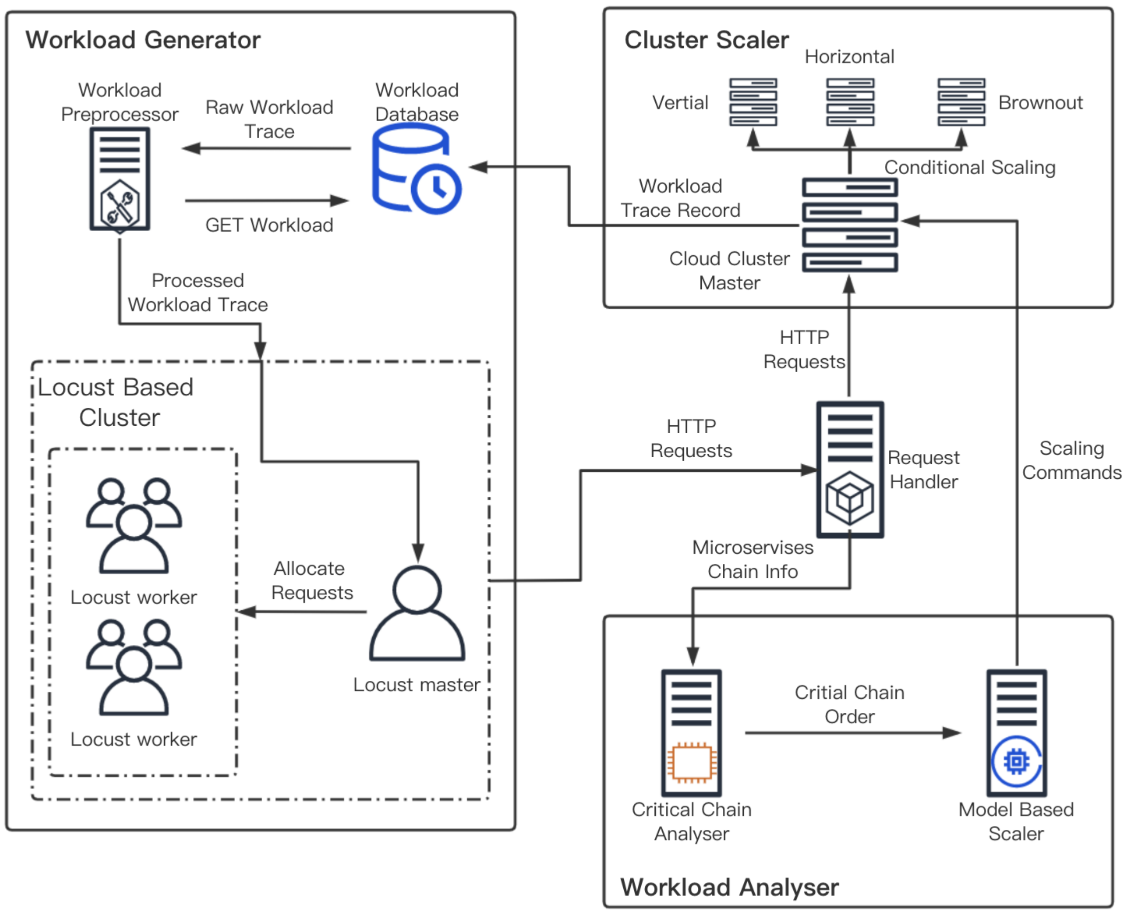

Figure 2: MAPE-K model of the system

The Monitor-Analyze-Plan-Execute over a shared Knowledge (MAPE-K) of the system model is shown in Fig. 2, consists of three sections: Workload Generator, Workload Analyser and Cluster Scaler.

3.3.1 Workload Generator

In our model, the Workload Generator is made up of a series of components, including workloads and processors, a Locust-based request generator and a database that stores historical workloads. **These components coordinate to extract the required workload data and characteristics from the historical database, preprocess workload dataset, simulate real user usage through the Load Generator and translate them into requests to the server cluster.**

The workload database contains the historical workloads from a running cloud server cluster. The workloads are regularly collected by specific servers or built-in programs and include information such as timestamps, machine numbers, CPU utilization, internal occupancy sizes, etc.

In order to generate different workloads for different parts of the microservices in the server cluster based on user behaviour more accurately, we used workload distribution (Zeng et al., 2019). The workload distribution is shown in the Table 2 . In addition to simulating user behaviour for the front-end, we also simulated user behaviour for different pages of the site. For example, we simulated users browsing an item screen, placing items in their shopping basket, viewing their shopping basket, etc. To better simulate the usage of multiple users, we used dynamic user generating rule, i.e. the user IDs were always different for the same time period, which simulated tests conducted in a highly concurrent environment.

3.3.2 Workload Analyser

The main role of the Workload Analyser is to allocate the requests sent from the cloud servers and to analyze the chain of the microservices to find out the critical nodes and critical chains, after which a cluster scheduler with a deep learning/reinforcement learning (RL) model is deployed to send scheduling requests to the cloud server cluster.

Decision tree-based chain analyzer : For the critical nodes, our project is judged and results are obtained using a decision tree-based classifier. Training on the dynamically

Table 2: The workload distribution of microservices

Microservices	**Workload Distribution**
Front-end	45.5%
Order	22.7%
Carts	22.7%
Catalog	5.7%
Random item	3.4%
Total	100%

changing chain relationships results in a decision tree model that accurately determines whether a node is critical or not. The decision tree has three decision outcomes: critical node, not a critical node, and potential critical node. Each of the three outcomes includes at least one critical node in each critical chain, while non-critical nodes and potential critical nodes may not exist in critical chains.

We first need to analyze the current microservice chain relationships of the cluster. This involves analyzing the connection status and the latency of all the nodes in the existing microservice chains and sorting out all the different chains that make up the service. Finally, these different chains are sorted by the maximum latency of the pods it contains. Then the decision tree model is used as a classifier to determine whether these pods in the selected chains are critical pods or not.

In the first step, we should get a set of critical chains, consisting of at least one chain, and a set of non-critical chains consisting of at least zero chains. Then analysis will be based on pods in these chains.

The decision tree model we use is based on a dataset of parameters for the current chain. The dataset we use includes the current CPU utilization, memory usage, and latency of the pods. We will train a binary classification model based on these parameters to classify pods into critical pods and non-critical pods. For example, if there is a pod with relatively high latency in the chain, and high CPU and memory usage, then this pod is more likely to be judged as a critical pod.

Neural network-based pod workload predictor: The Workload Predictor module is responsible for estimating the expected load on the next scheduling window, which guides scaling methods to decide the number of resources the current pod needs to use. This module can be constituted by different predictive models such as Multivariate Time series Forecaster Abadi et al. (2016). The Multivariate Time Series Forecaster accepts the pre-processed information of workloads from the Request Handler and predicts the future workload of pods based on suitable predictive models.

Our approach is to use a GRU-based neural network to analyze the various information metrics of pods. In the case of pods belonging to the same microservice, i.e., multiple pods usually generated by the microservice through extensions, we analyze these pods as a single microservice.

In general, our deep learning model can analyze the historical data of the microservice, including historical CPU usage, memory usage, network throughput, and hard disk read/write, to determine the load of the microservice. We will discuss further in Section 4, including how to classify the load state.

RL-based cluster scaler: With the microservice chain information and microservice latency information, we can determine the critical chains and critical nodes in the entire microservice. With the overload status of each node, we can further determine the status of the microservices in the whole cluster.

The cluster scaler 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 Cao et al. (2020). The Resource Scaler receives the user requests from Request Handler and information like predicted workloads from Workloads Predictor in our system. Unlike existing solutions, the CLRPC jointly applies three scaling techniques, including vertical scaling, horizontal scaling, and brownout. The techniques can be used separately or jointly based on the system status and workload characteristics. After executing the scaling techniques, the CLRPC scaler collects the performance metrics, such as resource utilization and response time, and sends the data back to the users.

In the RL-based Resource Scaler module, the supported techniques work in different manners. The 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, 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. The Horizontal scaling is applied at the node level by adding or removing additional nodes into the system. Considering the costs of horizontal scaling, this technique can be applied when vertical scaling and brownout cannot handle the workload level and keep the system in a normal state.

If vertical scaling functions well and the system is working into the normal states again, the other techniques are not required to be executed.

3.3.3 Cluster Scaler

The Cluster Scheduler works on a cluster of cloud servers and accepts scheduling requests from the Workload Analyser to schedule microservices in the corresponding cloud servers. There are three main types of scheduling: Vertical, Horizontal, and Brownout. The implementation of these three scheduling policies will be described in more detail in Section 4.

4 CLRPC: A Chain Latency-based Resources Provisioning for Cluster

In this section we present the model and implementation of the algorithms used in CLRPC. It contains the transformation of the original dataset, deep learning based model analysis and reinforcement learning based approaches, and an automatic scheduler based on thresholds.

4.1 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 Chen et al. (2018). The Fig. 3 shows the trace from the Alibaba dataset. We have divided them into per-day and

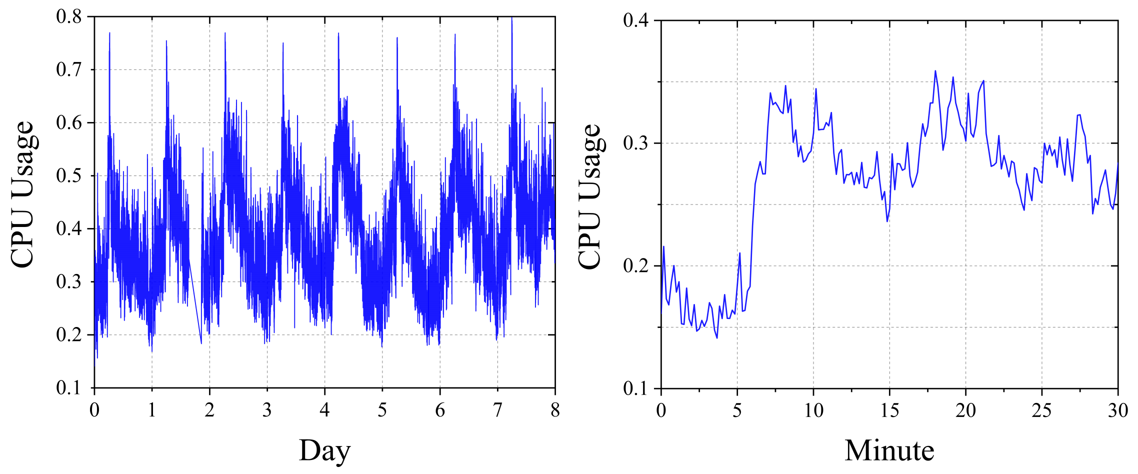

Figure 3: Alibaba (a) per-day workload (b) per-minute workload fluctuations

per-minute workload fluctuations of machines so that we can see the fluctuations of CPU usage more clearly over time. We can notice that both the datasets show high variance and random features.

The detailed profiling procedures are as follows: we consider the scheduling interval as 5 minutes. Firstly, we apply a 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 Mendoza (2008) 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. 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 900 requests per seconds. 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.

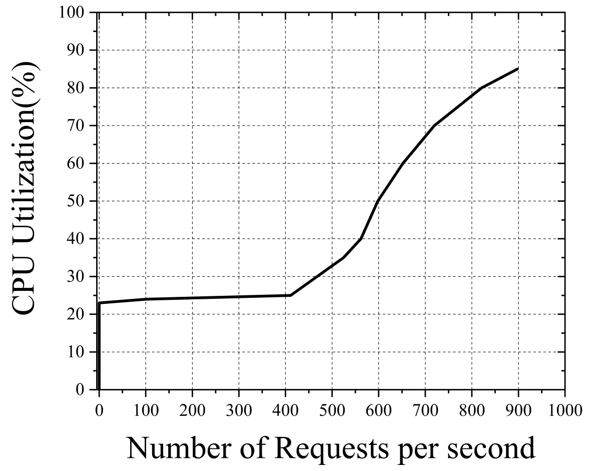

Figure 4: CPU utilization and the corresponding number of requests based on tested host

4.2 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 pod resources utilization. Moreover, this also helps to apply similar scaling techniques for pod workloads at the same level. This helps to significantly reduce the state-action space in the RL-based approach.Here, we use pods to describe the smallest unit that can be scheduled in Kubernetes. But the difference is that we specify that each individual microservice is measured separately as a pod, i.e., our scheduling algorithm is based on the microservice hierarchy for analysis, based on the smallest unit that can be scheduled, pod, for scheduling. We divide the pod workloads into five levels that can represent the degree of overloads, as shown below:

Level 0: No loads are allocated to the pod, and the hosts can be switched into the low-power mode or turned off, i.e., the resource utilization is 0.

Level 1: The pod 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 pod 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 pod 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 pod is overloaded, the QoS will be significantly impacted, and the effective scaling approaches should be performed, i.e., utilization is between 75% to 100%.

To accurately predict the pod 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 (line 1). To convert the time series into a supervised learning sequence, the algorithm needs to arrange the time series ordered by time and then store the data of the previous time slot $L(t-1)$ and the current time slot $C(t)$ into the corresponding places in the matrix (lines 2-11). After this, the predicted data $F(t+1)$ are also included in the matrix (lines 12-13). Then, the algorithm combines these data to form a new supervised learning data sequence, where the data of the previous and the current time slots are labeled as a sample, and the data of the next time slot is used as labels (lines 14-17). Finally, the empty data must be detected and removed to improve prediction accuracy (lines 19-22).

After the labelled data is generated for supervised learning, to achieve the accurate prediction for workloads, we apply the Gated Recurrent Unit (GRU) Chung et al. (2014) derived from Recurrent Neural Network (RNN) Mikolov et al. (2011), which 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). The structure of

Algorithm 1: CLRPC: pod level 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
1  Initialize an empty matrix $S$ to record supervised time series data
2  for $t$ from $1$ to $T$ do
3      Record data in current time interval: $C(t) \leftarrow E(t)$
4      if $t = 1$ then
5          Record NONE
6      else
7          Record data in last time interval: $L(t-1) \leftarrow E(t-1)$
8      end
9      if $t = n - 1$ then
10         Record NONE
11     else
12         Record data in next time interval: $F(t+1) \leftarrow E(t+1)$
13     end
14     Merge $L(t-1)$, $C(t)$ and $F(t+1)$ together into $S(t)$:
15     $S(t) \leftarrow \{L(t-1), C(t), F(t+1)\}$
16     $L(t-1), C(t)$ are marked as samples in supervised learning
17     $F(t+1)$ are marked as labels in supervised learning
18     if $S(t)$ contains NONE then
19         Remove $S(t)$
20     Update $S$
21 end
```

GRU is demonstrated in Fig. 5. The parameters in the GRU include the reset gate r_t, the update gate z_t, the candidate hidden layer y'_t and the output gate y_t. The updates of these parameters are shown as follows:

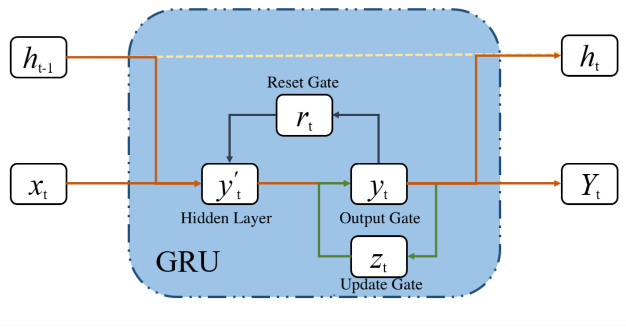

Figure 5: The structure of GRU

Table 3: 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

$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right), \tag{1}$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right), \tag{2}$$

$$y'_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right), \tag{3}$$

$$y_t = (1 - z_t) * h_{t-1} + z_t * y'_t, \tag{4}$$

where x_t is the input data, h_t is output data, Y_t is predicted data after processed by the gates in GRU, σ is the sigmoid function and W are the parameters of neural network.

The Table 3 shows the mean square errors (MSE) of actual utilization and predicted utilization for Alibaba workloads during different periods. The MSE has lower values about 3×10^{-3} to 7×10^{-3}, which demonstrates that our workload prediction algorithm has a good performance in utilization prediction.

4.3 RL-based Resource Scaling

The RL-based approach solves sequential decision-making problems by modeling the problem as Markov Decision Process Wang et al. (2021). 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 π_θ, where θ 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 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 Yin et al. (2008) 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 π_θ. 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 can be updated as:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r(s_t, a_t)] + \gamma max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)] \tag{5}$$

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. However, updating the Q Table can have the curse of dimension due to the large solution space and size of the Q Table ($S * A$). Therefore, it is promising to train the model offline with historical data. The network parameters can be trained offline to minimize the loss function and reduce the training time significantly. Then the model can be deployed in a system for online decisions and update the actions with rewards via online training. In addition, as introduced in Section 4.2, we also use representative load levels to reduce the state space.

Classical Q-Learning models often encounter problems such as too large Q-table and slow Q-value parameter transfer due to too many input states for decision making. Moreover, as the dimension of the input becomes larger, the state space of the model will grow exponentially, which is not what we want for a good model. A good reinforcement learning model must balance the training accuracy and pre-training speed.

To solve the above problem, we use Approximate Q-Learning. To reduce the size of the incoming state space, we use the information of the current pod as parameters to pass into the model, such as the current load state, e.g. level 1, and the current position of the pod in the chain, the current latency of the microservice, etc. We limit the size of the state space by reducing the incoming information We limit the size of the state space by narrowing down the incoming information and ignoring some irrelevant parameters. In the model input of our reinforcement learning, the state $s_t \in S$ represents the current microservice chain that contains the information of each microservice. And action $a_t \in A$ consists of three scheduling policies.

To illustrate our RL-based multi-faceted scaling framework (CLRPC), 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 machine pm_k. For each pm_k, the actions can be performed on it 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, $v_k^i \in [-m, m]$ represents the amount of $m \in R$ 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 values of h_k and v_k^i represent more resources are added, negative values mean that resources are removed, and 0 represents no change will be performed. Considering the total number of physical machines is K, the final set of actions of as 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 its normal state. Therefore, to optimize resource usage, the action a_g is required to make scaling decisions based on a global view.

CLRPC incorporates the offline training and online training approach together to achieve optimized actions. Once the load change is identified, CLRPC 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 ϵ-greedy policy, Kardani-Moghaddam et al. (2021) a standard policy to balance offline and online training. In ϵ-greedy policy, a random action with probability equals ϵ is selected. Otherwise, it chooses the action with the maximum Q value.

The final objective of CLRPC 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 (6), 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.

$$R_{qos}(rt) = \begin{cases} e^{-(\frac{rt-RT_{max}}{RT_{max}})^2} & , rt > RT_{max} \\ 1 & , rt \leq RT_{max} \end{cases} \tag{6}$$

As for the reward of resource utilization, we model it as shown in Equation (7). 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.

$$R_{util}(u_k) = \begin{cases} \frac{\sum_{k=1}^{K} U_k^{max} - u_k}{K} + 1 & , u_k \leq U_k^{max} \\ \frac{\sum_{k=1}^{K} u_k - U_k^{max}}{K} + 1 & , u_k > U_k^{max} \end{cases} \tag{7}$$

Equation (8) shows the final reward value based on response time and resource utilization, in which the higher values of RT_{qos} and lower values of RT_{util} can increase the total reward.

$$r(s_t, a_t) = \frac{R_{qos}(rt)}{R_{util}(ut)} \tag{8}$$

The Algorithm 2 shows the general procedure of CLRPC. 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 utilization. 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). CLRPC also supports online training, after CLRPC transits the state s_t to s_{t+1}, 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

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

goodness of (s_t, a_t) (lines 9-14).

The Algorithm 3 shows the action execution process of CLRPC. 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 horizontal scaling will be examined. If the horizontal scaling 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.

4.4 Workload and microservice chain analysis

After the request has been generated and sent to the Request Handler, we send the request to the target cluster on the one hand and collect the workload from the cloud server cluster on the other. These workload profiles include information such as the latency of the microservice itself, the resource consumption of the cloud server, the request latency of the microservice, and the success rate of the request. By passing this information to the Workload Analyzer, the critical chain analysis of the microservices, the chain relationship

Algorithm 3: CLRPC: 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 /*Horizontal scaling*/
12 if $n > 0$ then
13     Add $n$ microservices replicates ;
14 else
15     Remove $n$ microservices replicates;
16 end
17 /*Brownout control*/
18 if $b_k(t) > 0$ then
19     Deactivate microservices via brownout;
20 else
21     Brownout will not be triggered;
22 end
```

of the microservices in the current state can be obtained.

4.5 Cluster Scaler Scheduling Policy

Following critical chain and node determination and model-based scheduling decisions, we use three algorithms to schedule the cloud server cluster: Horizontal scaling, Vertical scaling, and Brownout.

Horizontal scaling adjusts resources by adding or removing copies (replicates) of microservices on cloud servers within the cluster to improve resource usage and system availability. A cloud server cluster can achieve a reduced workload on the cloud server cluster by adding copies of microservices.

Vertical scaling adjusts the processing service capacity of the current cloud server by adjusting the amount of CPU, memory, or network resources allocated to microservice instances. Cloud server clustering can reduce the workload of a cloud server cluster by increasing the resources of a single cloud server or multiple cloud servers within the cluster to a single or multiple microservices.

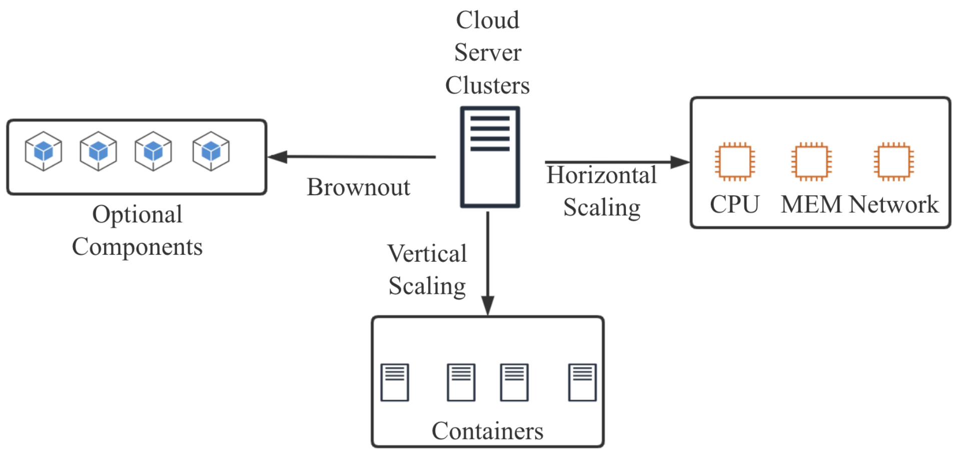

Figure 6: Three scheduling policies in the system

The three scheduling policies are shown in Fig. 6.

4.6 Auto-adapter for unpredictable workloads change

Although our workload prediction algorithm can achieve high accuracy, sudden fluctuations of cloud workloads during the stable period (workloads remain unchanged) are usually unpredictable.

Microservices are more sensitive to resource fluctuations than traditional applications Qiu et al. (2020b). 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 sudden changes. 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 is integrated into the Cluster Scaler module in Fig. 2. 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. 7, 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.

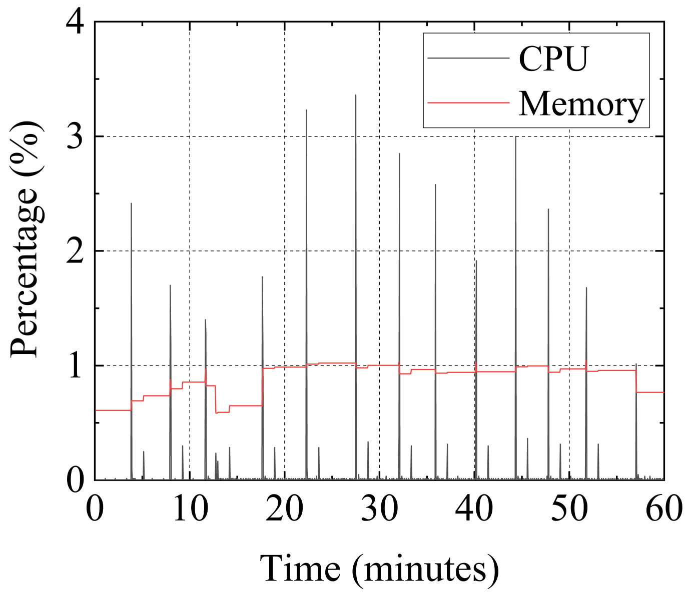

Figure 7: Resource consumption of auto-adapter

5 Performance Evaluation

To evaluate the performance of our system and the algorithms in the real environment, we used a Kubernetes-based cluster and some servers to complete this experiment. The comparative algorithms of our experiments and the analysis of the results will be presented in this section.

5.1 Experimental environment

We built a cluster of the homogeneous cluster for the microservices-based application deployment, which means that all machines within this cluster are configured identically. The cluster consists of five physical machines, all configured with Intel Xeon E5-2660 processors and 64 GB of RAM, with one physical machine as the master of the cluster and the remaining as a worker. we used the Kubernetes-based version of Sock Shop for the deployment. The list of environments for deep learning and reinforcement learning is as follows: Python (version 3.9) and TensorFlow (version 2.2.0), Sklearn (version 1.1) (Pedregosa et al., 2011), Locust (version 2.8.0).

The dataset we use is from the Section. 4.1, where we use the preprocess method and get the 8 days of requests based on our cluster. We selected 5000 minutes from the dataset to analyze. The Locust platform we used uses five minutes as a period and the number of requests in each period is modeled as the sum of all requests in that period corresponding to the number of users.

5.2 Baseline

KuScal: KuScal has been used in Kubernetes, which is mainly based on horizontal scaling that can dynamically increase or decrease the number of replicates Burns et al. (2019). The 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. In KuScal, if the server's CPU utilization is larger than the pre-configured threshold, KuScal will create more microservice replicates and vice versa. To obtain the real-time resource utilization of the server, KuScal periodically collects the resource utilization data from each node in the cluster. The algorithm formulas of Kubernetes scaling are as follows:

$$U_{avg} = \frac{U_{total}}{R_{total}}, \tag{9}$$

$$\frac{U_{avg}}{Tr} - 1 > 0.1, \tag{10}$$

where U_{avg} is the average CPU utilization in the cluster, Tr is a CPU threshold predefined by Kubernetes, U_{total} is the sum of resource usage values of all microservice nodes, and

R_{total} is the total number of replicates.

Auto: Auto is an abbreviation for Auto pilot scale algorithm from Google (Reiss et al., 2012), and in the following Auto refers to Auto pilot scale algorithm. When we perform this scheduling algorithm, we collect information about the current liabilities of the server at certain intervals and save it in a specific file. When we need to schedule a server, we first read the last 5 recorded data from this file and process this data using the sliding window algorithm. We average the server load information over the previous five sampling points, and if this average is greater than the threshold we have set, we schedule the server, for example, by increasing or decreasing the server's resources (replicates).

FIRM (Qiu et al., 2020a): FIRM is implemented based on the Longest chain-based microservice scheduling algorithm. Its approach consists of the following steps. First, we need to obtain basic information about all the microservices in the system, and then we arrange these microservices in terms of length, i.e. we analyze them by counting the longest one among the microservice chains. If there are multiple chains of the same length in the same microservice chain, we only analyze the last microservice chain obtained. After this, we need to schedule the information for the whole cluster, which involves using horizontal versus vertical scaling for scheduling. To ensure consistency of the experiment, the size of the total available resources in this algorithm is kept consistent with the other methods.

5.3 Experiment Analysis

Requests per second: The Fig. 8 shows the requests per second of four algorithms. As direct analysis of the 5000-minute results was a little too difficult to observe trends, we divided the 5000 minutes of the experimental results into 5 slices, each time slice representing the average of the results over the current period. After this point, the results were sliced in this method to facilitate reading and analysis.

We can determine the load and QoS of the four algorithms to a certain extent by looking at their requests per second, and when analyzing the parameter requests per second. we generally consider that the larger the requests per second, the better the QoS of the server and the better the control of the load state of the server. It is because the higher the RPS of the server, the better the QoS of the server. A larger server RPS means that the server

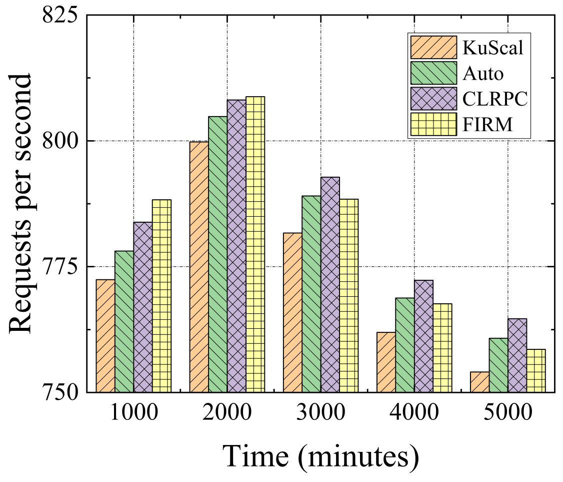

Figure 8: Comparison of requests per second

can handle more requests per second. From the user's point of view, the server is more responsive to its requests, and from the cloud service provider's point of view, a larger RPS also means better service results.

The KuScal is not very good at scheduling the next time slice based on the available data when dealing with highly variable load situations, and the threshold-based algorithm does not provide a good solution in this case. In contrast, Auto uses a processing step based on historical data information, which is used as a reference under high variability conditions. If the load on the server rises or falls continuously over a certain period, then Auto's algorithm can accurately determine the change in load and give a response strategy. As a result, the performance of the Auto algorithm is somewhat better than KuScal over the five-time periods but still worse than CLRPC and FIRM.

We observed that at each of the five time slice, FIRM had a higher RPS relative to CLRPC in the first two time periods, but CLRPC was higher in the next three time periods. Moreover, the gap between CLPRC and FIRM becomes smaller and smaller over the first two time periods until it overtakes FIRM. FIRM held first place for just two of the earliest time slices but was slightly worse than Auto in the next three time periods. The effect of CLRPC's scheduling strategy will be better over time better and better with RPS as the reference variable, with CLRPC having a more stable RPS over a longer period.

Regarding the gap between CLRPC and FIRM, it is reasonable to conclude that the scheduling algorithm relying solely on the scheduling with the longest chain becomes more difficult to cope with the scheduling of server clusters in the case of high variability over time. This is because the longest chain usually implies a higher load at the beginning of the experiment where there is less variation in the chains. But over time and as the load changes, the real targets for optimization are the chains with high load, which may not be the longest chain. At the same time, it is becoming increasingly difficult for FIRM to cope with this more complex chain variation due to possible chain variations with the appearance of chains of the same length. The CLRPC algorithm, on the other hand, solves the problem of scheduling decisions due to chain variation. CLRPC is also optimized to a certain extent compared to the relatively static approach of KuScal and Auto. As a result, CLRPC performs better in short periods than FIRM's scheduling performance but is effective in scheduling cloud server clusters for long periods of time under load. overall, CLRPC optimized the requests per second from 1.2% to 8.1% compared to baselines.

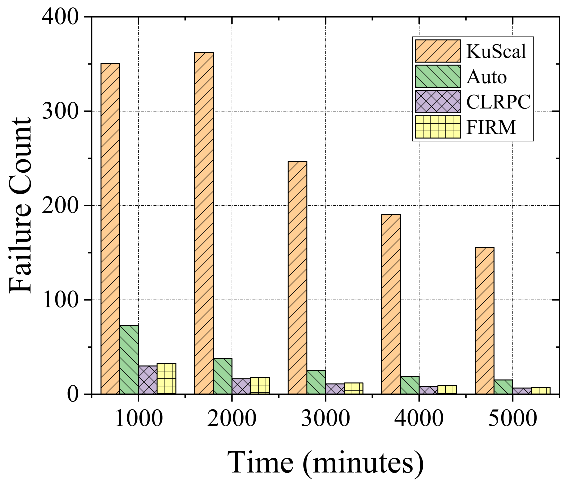

Figure 9: Comparison of failure count

Failure count: The image of Failure Count is shown in Fig. 9. The failure count represents the number of error requests in a given time slice, where the Failure Count in this figure is obtained by calculating the average of the error rate over a period of time.

Therefore, the larger the value of Failure Count, the higher the number of error requests occurring in the same period, and the worse the quality of service and the higher load on the server. Because when the error rate in the same period is high, the failure rate of requests in the current period will increase accordingly, and the direct feedback to the user is that the server cannot connect.

We can first see that KuScal's Failure Count is the highest in all five time periods, which is consistent with the result in the previous analysis of Requests per second, i.e., KuScal has difficulty coping with high-dimensional and high-variability load states, and has difficulty responding promptly, which leads to frequent server overloads and greatly affects the QoS of the server, resulting in the Failure The number of Failure Count increases.

In contrast, the Auto method greatly alleviates this problem because Auto introduces a scheduling policy based on historical data, and Auto can respond effectively compared to KuScal when there is an upward or downward trend in historical data. Auto is very effective in controlling the value of Failure Count and keeping it in an acceptable range.

However, the Failure Count of the Auto method is still higher than that of CLRPC and FIRM. In the Failure Count comparison, both CLRPC and FIRM have a good performance, but it is still difficult to analyze further in this picture because their values are at a very low level compared to KuScal. We will discuss this issue further in the next Section 5.4. Overall, CLRPC reduces 8.3% of failure count compared to FIRM.

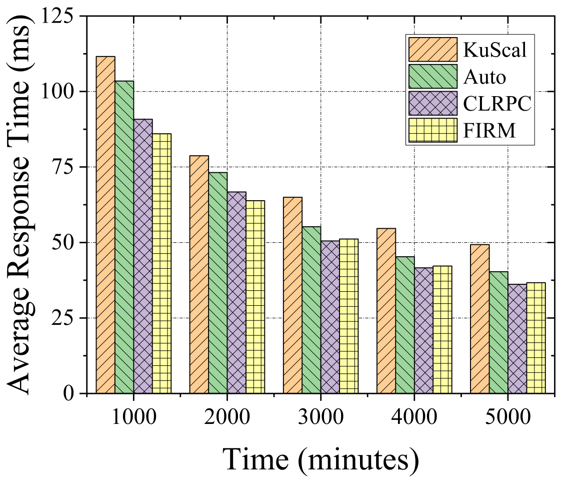

Figure 10: Comparison of the average response time

Average response time: The Fig. 10 shows the average response timeof four algorithms. The average response time is also one of the most important indicators of QoS, and the lower the average response time, the better the QoS of the server.

The average response time of KuScal is at the highest value in all five time periods, and we consider it as a benchmark test to analyze the performance of the other three algorithms. The Autopliot method is optimized compared to the KuScal, and it stays at the second-highest in all five time periods. The results of CLRPC and FIRM are similar to the result of RPS analyzed before, in that in the starting two time periods, FIRM outperforms CLRPC. But over time, CLRPC outperforms FIRM and maintains its optimization in the last three time periods. It shows that CLRPC outperforms FIRM from 3000 minutes to 5000 minutes. Overall, CLRPC optimized from 1.4% to 26.6% compared with baselines.

Although we can see the difference over the five time periods, it is difficult to see the actual distribution of algorithm optimization. For example, in the average delay time, we can see the optimization of each algorithm over the five time periods, but it is difficult to get the data distribution over the whole period, so we introduce the CDF in the next Section 5.4.

5.4 CDF Analysis

To better demonstrate the differences between CLRPC and FIRM, we use the Cumulative Distribution Function (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 $F_X(x) = \mathrm{P}(X \leq x)$. We still used the three metrics from the previous analysis and transformed them into CDF curves for further analysis. Overall, CLRPC optimized the average response time by 1.4% compared to FIRM.

Requests per second: The Fig. 11 shows the requests per second figure of CLRPC and FIRM. In the CDF curve for requests per second, the FIRM and CLRPC curves are very close to each other. For the same number of requests per second, the vertical coordinate of the curve represents the distribution of all values less than or equal to the current number of requests per second. For values between 400 and 800 requests per second, the CDF curve for FIRM stays above that of CLRPC, but the difference between the two is not very large. In the dense area of the requests per second CDF distribution, where the slope of the CDF curve is larger, this means the optimization effect of CLRPC and FIRM on the cluster is very close. However, we observe a clear difference be-

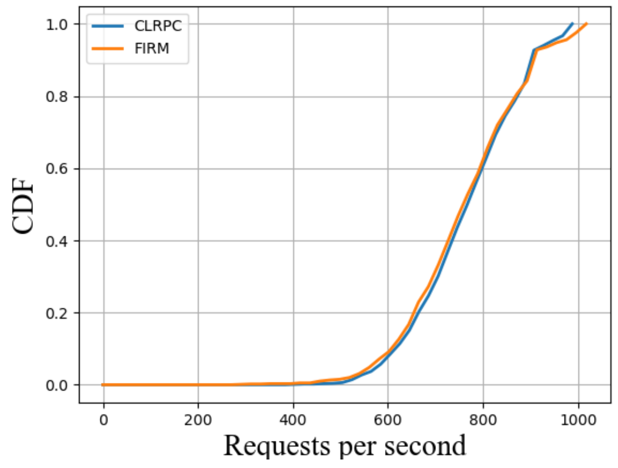

Figure 11: CDF of requests per second

tween the CLRPC and FIRM curves when requests per second reach the range of 800 to 100.

As the requests per second increase, CLRPC is the first to reach where CDF value equal to 1.00, while FIRM reaches where CDF value equal to 1.00 for larger requests per second. In this case, a higher request per second means a higher risk of error rate with the same resources.

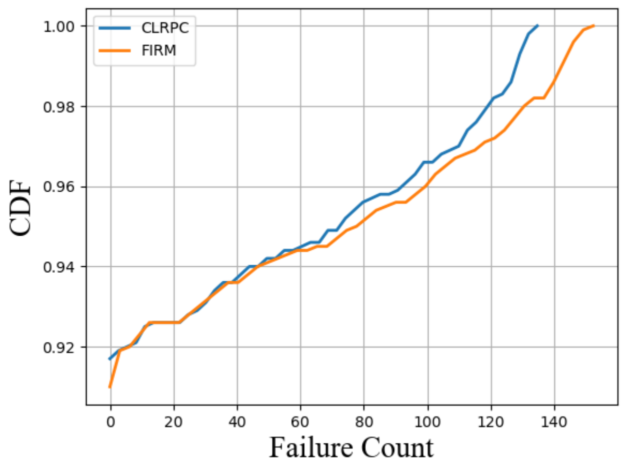

Figure 12: CDF of failure count

Failure count: The Fig. 12 shows the failure count of CLRPC and FIRM. We can see that as the number of Failure counts rises, the CDF curve also increases, while the CLRPC curve is the first to reach where CDF value equal to 1.00.

The characteristic of the CDF curve in this figure is that the higher the CDF value corresponding to the same failure count, the more percentage of the failure count is within the corresponding failure count. Since the metric of failure count is a measure of the number of errors that occur in the whole cluster, the lower the number of corresponding errors, the better the performance of the algorithm.

In the whole CDF curve, the CDF curve of CLRPC is basically above the FIRM curve for all the failure counts, i.e., the horizontal coordinate. This indicates that for the distribution of the number of error rates, CLRPC will have fewer errors than FIRM most of the time. At the same time, the FIRM curve reaches where CDF value equal to 1.00 with a higher failure count than CLRPC, which also indicates that the cluster under FIRM scheduling has a higher error rate, which does not happen in CLRPC.

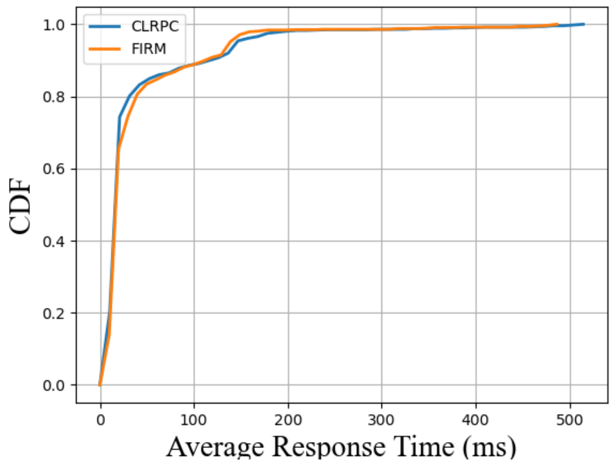

Figure 13: CDF of the average response time

Average response time The Fig. 13 shows the average response time of CLRPC and FIRM. The CDF curves for FIRM and CLRPC are also very close in average response time. However, we can see that CLRPC's CDF curve appears slightly higher than FIRM's CDF curve when the average response time lies in the interval from 0 to 100. For the average response time, this suggests that CLRPC has more of its average response time spread in the lower range, which means that throughout the tests of the experiment (5000 minutes of testing), CLRPC was more effective in optimizing the cluster overall, as a lower average response time means a higher QoS.

6 Conclusions 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 deep learning and reinforcement learning based scaling approach, named CLRPC, for scaling resources for microservices that combine the three techniques, including horizontal scaling, vertical scaling, and brownout. It ensures cloud service providers increase their resource utilization while guaranteeing the QoS. CLRPC utilizes deep learning approaches for workload prediction to predict pod level workload more accurately compared to traditional regression-based prediction approaches. Furthermore, leveraging the Reinforcement Learning framework takes decisions on scaling strategies, adapting to the load changes, and allocating the appropriate resources. To evaluate the performance of CLRPC, 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. The experimental results have demonstrated that CLRPC can outperform the baseline algorithms in different performance metrics, including RPS (requests per second), failure count, and average response time.

As for future work, we would like to investigate the scaling of network resources and the improvement of scheduling algorithms. We also intend to consider the energy-efficient scaling of microservices and to apply the proposed approach in large-scale environments as well as in larger clusters of cloud servers.

References

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul

Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, Savannah, GA. USENIX Association.

David Balla, Csaba Simon, and Markosz Maliosz. 2020. Adaptive scaling of kubernetes pods. In *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*, pages 1–5. IEEE.

Rabindra K Barik, Rakesh K Lenka, K Rahul Rao, and Devam Ghose. 2016. Performance analysis of virtual machines and containers in cloud computing. In *2016 international conference on computing, communication and automation (iccca)*, pages 1204–1210. IEEE.

David Bernstein. 2014. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84.

Brendan Burns, Joe Beda, and Kelsey Hightower. 2019. *Kubernetes: up and running: dive into the future of infrastructure*. O'Reilly Media.

Y. Cao, Y. Zhao, J. Li, R. Lin, J. Zhang, and J. Chen. 2020. Multi-tenant provisioning for quantum key distribution networks with heuristics and reinforcement learning: A comparative study. *IEEE Transactions on Network and Service Management*, 17(2):946–957.

W. Chen, K. Ye, Y. Wang, G. Xu, and C. Xu. 2018. How does the workload look like in production cloud? analysis and clustering of workloads on alibaba cluster trace. In *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 102–109.

Junyoung Chung, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. 2014. Empirical evaluation of gated recurrent neural networks on sequence modeling. In *NIPS 2014 Workshop on Deep Learning, December 2014*.

Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. 2017. Microservices: yesterday, today, and tomorrow. *Present and ulterior software engineering*, pages 195–216.

Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. 2019. 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*, pages 19–33.

Alim Ul Gias, Giuliano Casale, and Murray Woodside. 2019. Atom: Model-driven autoscaling for microservices. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 1994–2004.

Xiaofeng Hou, Chao Li, Jiacheng Liu, Lu Zhang, Yang Hu, and Minyi Guo. 2020. Ant-man: towards agile power management in the microservice era. In *2020 SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1098–1111. IEEE Computer Society.

Ram Srivatsa Kannan, Lavanya Subramanian, Ashwin Raju, Jeongseob Ahn, Jason Mars, and Lingjia Tang. 2019. Grandslam: Guaranteeing slas for jobs in microservices execution frameworks. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA. Association for Computing Machinery.

S. Kardani-Moghaddam, R. Buyya, and K. Ramamohanarao. 2021. Adrl: A hybrid anomaly-aware deep reinforcement learning-based resource scaling in clouds. *IEEE Transactions on Parallel and Distributed Systems*, 32(3):514–526.

Tamas Kiss, Péter Kacsuk, József Kovács, Botond Rakoczi, Ákos Hajnal, Attila Farkas, Gregoire Gesmier, and Gabor Terstyanszky. 2019. Micado—microservice-based cloud application-level dynamic orchestrator. *Future Generation Computer Systems*, 94:937–946.

Aviral Kumar, Justin Fu, Matthew Soh, George Tucker, and Sergey Levine. 2019. Stabilizing off-policy q-learning via bootstrapping error reduction. *Advances in Neural Information Processing Systems*, 32.

Anthony Kwan, Jonathon Wong, Hans-Arno Jacobsen, and Vinod Muthusamy. 2019. Hyscale: Hybrid and network scaling of dockerized microservices in cloud data centres. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 80–90. IEEE.

Lei Liu. 2019. Qos-aware machine learning-based multiple resources scheduling for microservices in cloud environment. *arXiv preprint arXiv:1911.13208*.

Alfredo Mendoza. 2008. Using nmon to monitor sas applications on aix servers. In *SAS Global Forum*, pages 16–19. Citeseer.

T. Mikolov, S. Kombrink, L. Burget, J. Černocký, and S. Khudanpur. 2011. Extensions of recurrent neural network language model. In *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5528–5531.

Thanh-Tung Nguyen, Yu-Jin Yeom, Taehong Kim, Dae-Heon Park, and Sehan Kim. 2020. Horizontal pod autoscaling in kubernetes for elastic container orchestration. *Sensors*, 20(16):4621.

Yipei Niu, Fangming Liu, and Zongpeng Li. 2018. Load balancing across microservices. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 198–206. IEEE.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

Amit M Potdar, DG Narayan, Shivaraj Kengond, and Mohammed Moin Mulla. 2020. Performance evaluation of docker container and virtual machine. *Procedia Computer Science*, 171:1419–1428.

Haoran Qiu, Subho S Banerjee, Saurabh Jha, Zbigniew T Kalbarczyk, and Ravishankar K Iyer. 2020a. {FIRM}: An intelligent fine-grained resource management framework for {SLO-Oriented} microservices. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 805–825.

Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2020b. FIRM: An intelligent fine-grained resource management framework for slo-oriented microservices. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 805–825. USENIX Association.

Charles Reiss, Alexey Tumanov, Gregory R Ganger, Randy H Katz, and Michael A Kozuch. 2012. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the third ACM symposium on cloud computing*, pages 1–13.

Fabiana Rossi, Matteo Nardelli, and Valeria Cardellini. 2019a. Horizontal and vertical scaling of container-based applications using reinforcement learning. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 329–338. IEEE.

Fabiana Rossi, Matteo Nardelli, and Valeria Cardellini. 2019b. Horizontal and vertical scaling of container-based applications using reinforcement learning. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 329–338.

Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. 2020a. Autopilot: Workload autoscaling at google. In *Proceedings of* *the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA. Association for Computing Machinery.

Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, et al. 2020b. Autopilot: workload autoscaling at google. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16.

Tasneem Salah, M Jamal Zemerly, Chan Yeob Yeun, Mahmoud Al-Qutayri, and Yousof Al-Hammadi. 2017. Performance comparison between container-based and vm-based services. In *2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN)*, pages 185–190. IEEE.

László Toka, Gergely Dobreff, Balázs Fodor, and Balázs Sonkoly. 2021. Machine learning-based scaling management for kubernetes edge clusters. *IEEE Transactions on Network and Service Management*, 18(1):958–972.

S. Wang, Y. Guo, N. Zhang, P. Yang, A. Zhou, and X. Shen. 2021. Delay-aware microservice coordination in mobile edge computing: A reinforcement learning approach. *IEEE Transactions on Mobile Computing*, 20(3):939–951.

M. Xu, A. N. Toosi, and R. Buyya. 2021. A self-adaptive approach for managing applications and harnessing renewable energy for sustainable cloud computing. *IEEE Transactions on Sustainable Computing*, 6(4):544–558.

G Yin, C. Z Xu, and L. Y Wang. 2008. Q-learning algorithms with random truncation bounds and applications to effective parallel computing. *Journal of optimization theory and applications*.

G. Yu, P. Chen, and Z. Zheng. 2019. Microscaler: Automatic scaling for microservices with an online learning approach. In *2019 IEEE International Conference on Web Services (ICWS)*, pages 68–75.

Ming Zeng, Hancheng Cao, Min Chen, and Yong Li. 2019. User behaviour modeling, recommendations, and purchase prediction during shopping festivals. *Electronic Markets*, 29(2):263–274.

Shubo Zhang, Tianyang Wu, Maolin Pan, Chaomeng Zhang, and Yang Yu. 2020. A-sarsa: A predictive container auto-scaling algorithm based on reinforcement learning. In *2020 IEEE International Conference on Web Services (ICWS)*, pages 489–497.

Hao Zhou, Ming Chen, Qian Lin, Yong Wang, Xiaobin She, Sifan Liu, Rui Gu, Beng Chin Ooi, and Junfeng Yang. 2018a. Overload control for scaling wechat microservices. In

Proceedings of the ACM Symposium on Cloud Computing, SoCC '18, page 149–161, New York, NY, USA. Association for Computing Machinery.

Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chenjie Xu, Chao Ji, and Wenyun Zhao. 2018b. Poster: Benchmarking microservice systems for software engineering research. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 323–324. IEEE.