ChainsFormer: A Chain Latency-Aware Resource Provisioning Approach for Microservices Cluster

Chenghao Song[1], Minxian Xu[1(✉)], Kejiang Ye[1], Huaming Wu[2], Sukhpal Singh Gill[3], Rajkumar Buyya[4], and Chengzhong Xu[5]

[1] Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences, Shenzhen, China
{ch.song,mx.xu,kj.ye}@siat.ac.cn
[2] Tianjin University, Tianjin, China
whming@tju.edu.cn
[3] Queen Mary University of London, London, UK
s.s.gill@qmul.ac.uk
[4] Cloud Computing and Distributed Systems (CLOUDS) Lab, School of Computing and Information Systems, The University of Melbourne, Melbourne, Australia
rbuyya@unimelb.edu.au
[5] State Key Lab of IoTSC, University of Macau, Macau, China
czxu@um.edu.mo

Abstract. The trend towards transitioning from monolithic applications to microservices has been widely embraced in modern distributed systems and applications. This shift has resulted in the creation of lightweight, fine-grained, and self-contained microservices. Multiple microservices can be linked together via calls and inter-dependencies to form complex functions. One of the challenges in managing microservices is provisioning the optimal amount of resources for microservices in the chain to ensure application performance while improving resource usage efficiency. This paper presents *ChainsFormer*, a framework that analyzes microservice inter-dependencies to identify critical chains and nodes, and provision resources based on reinforcement learning. To analyze chains, ChainsFormer utilizes light-weight machine learning techniques to address the dynamic nature of microservice chains and workloads. For resource provisioning, a reinforcement learning approach is used that combines vertical and horizontal scaling to determine the amount of allocated resources and the number of replicates. We evaluate the effectiveness of *ChainsFormer* using realistic applications and traces on a real testbed based on Kubernetes. Our experimental results demonstrate that *ChainsFormer* can reduce response time by up to 26% and improve processed requests per second by 8% compared with state-of-the-art techniques.

Keywords: Microservice · Chain · Reinforcement learning · Kubernetes · Scaling

F. Monti et al. (Eds.): ICSOC 2023, LNCS 14419, pp. 197–211, 2023.
https://doi.org/10.1007/978-3-031-48421-6_14

1 Introduction

Microservice architecture is a popular approach for designing and developing modern applications. It involves breaking down monolithic applications into smaller, fine-grained components that can work together to provide services for users [15]. This approach allows development teams to focus on implementing different microservices, thereby speeding up the development process. Additionally, microservices can be updated or upgraded independently, making maintenance efforts more manageable. To ensure reliability and performance, microservices can be scaled and operated individually, depending on workload fluctuations and environmental variance.

Despite their independence, microservices are not entirely self-contained. Communication-based dependencies, such as remote procedure calls, exist between different microservices [7]. These dependencies can represent how requests are processed among different microservices. Based on these dependencies, microservices can be combined into chains to fulfill complex services. The length of a chain can vary from several nodes to tens of nodes. A single microservice, such as a database-related service, can also be shared by multiple chains to support the formation of different services. Additionally, microservice chains can be dynamic, scaling in or out as needed to accommodate new microservices. Given these features of microservices and the resource usage fluctuations, it is challenging to precisely pre-configure the amount of resources, provision and scale resources when deploying microservices in clusters.

Traditional approaches for improving application performance often rely on over-provisioning and autoscaling, which involve allocating more CPU and memory resources to microservices. These approaches typically use performance models, simple heuristics, static thresholds, or machine learning algorithms. However, these approaches have several limitations. Firstly, accurate performance models and efficient heuristic-based scheduling policies require significant manual efforts and training, which are infeasible for large-scale microservices with a large number of configurable parameters. Secondly, machine learning (ML) based approaches, such as support vector machines, rely on centralized graph databases, which can lead to scalability issues and inefficient scheduling when microservice chains are updated. Therefore, alternative approaches are needed to address these limitations and enable effective management of microservices.

This paper presents a solution to the limitations of traditional approaches with *ChainsFormer*, a chain latency-aware resource provisioning framework for microservices cluster based on chain feature analysis. *ChainsFormer* dynamically scales CPU and memory resources to microservices to ensure high-quality service. This framework utilizes online telemetry data, including requests information, application running data, and hardware resource usage, to capture the system state. By leveraging ML and reinforcement learning (RL) models, *ChainsFormer* can adapt to variances in the system and reduce the need for manual efforts. Overall, *ChainsFormer* provides an effective solution for managing microservices with a high degree of automation and accuracy.

Table 1. Comparison of related work

Approach	Autoscaling	Workloads Prediction	Machine Learning based Resource Provisioning	Chain Analysis	**Quick Adaption to Dynamic Chains**	**SLO-awareness**
Sage [2]		✓	✓	Partial		✓
Firm [9]	✓		✓	✓		✓
Parslo [8]	✓			✓		✓
PEMA [4]	✓					✓
Autopilot [10]	✓	✓	✓			✓
Sinan [14]	✓		✓	✓		✓
Seer [3]			✓	✓	✓	✓
CoScal [12]	✓	✓	✓			✓
ChainsFormer (ours)	✓	✓	✓	✓	✓	✓

To efficiently manage the dynamic nature of microservice chains and adapt to changes quickly, *ChainsFormer* employs various techniques. It first identifies the critical chain using the calling graph and utilizes a decision tree to find the critical node that has a significant influence on microservice performance. This approach avoids the limitations of heavy machine learning techniques and centralized graph databases, which struggle with dynamic changes. Additionally, *ChainsFormer* utilizes RL to make efficient and optimized decisions regarding vertical and horizontal scaling. These decisions include when to conduct scaling actions, which microservice should be scaled with resources, and how many resources of each type should be scaled. Furthermore, these decisions can be further optimized through RL with updated decisions, resulting in even more efficient resource provisioning.

To evaluate the effectiveness of *ChainsFormer*, we deployed representative microservice applications on Kubernetes, which is the state-of-the-art container orchestration platform. We compared *ChainsFormer* with three state-of-the-art baselines and used realistic traces from Alibaba to measure application performance and response time. Our results show that *ChainsFormer* outperforms the baselines in terms of application performance and response time. These findings demonstrate the effectiveness of *ChainsFormer* in providing efficient resource provisioning and management for microservice chains.

In summary, we make the following key **contributions**:

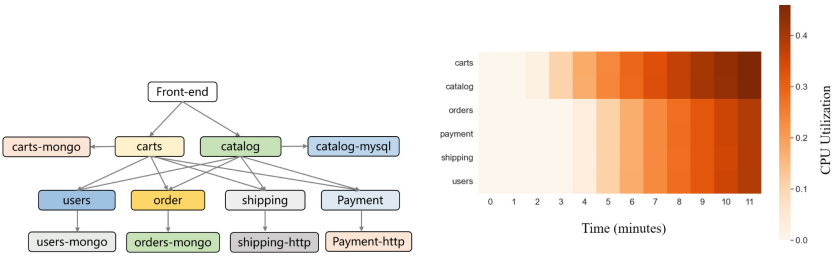
- We present the design of a framework that aims to handle the dynamic changes in microservice applications by identifying critical chains and nodes.
- We propose an RL-based approach for combining vertical and horizontal scaling to make decisions on efficient resource provisioning, which uses historical data for offline training and makes online decisions based on system states.
- We develop the designed framework on top of Kubernetes platform. Using realistic workload traces and real-world microservice, we demonstrate the efficiency of *ChainsFormer* compared to the state-of-the-art baselines.

2 Related Work

In this section, we will discuss the current state-of-the-art techniques that are designed to address the challenges of resource provisioning and autoscaling in microservices, in order to meet the desired quality of service levels.

Resource Provisioning for Microservices. Sage [2] aims to perform root cause analysis in microservice-based systems by utilizing causal Bayesian networks to identify the underlying reason for service level objective (SLO) violations. After identifying the root cause, Sage initiates autoscaling actions to mitigate the issue. One of the advantages of Sage is that it only requires lightweight tracking and is suitable for large-scale deployments. However, a major limitation of Sage is its heavy reliance on pre-trained machine learning models. Seer [3] employs deep learning models to predict quality of service (QoS) violations and dynamically adjusts allocated resources to each microservice to prevent such violations. It is particularly suitable for scenarios with frequent service updates and requires a large amount of tracking data. However, the accuracy of detection can be affected by significant application changes. Parslo [8] is a gradient descent-based approach that assigns partial SLOs to nodes in a microservice to provide resource configuration solutions quickly. One of Parslo's main advantages is its ability to achieve a globally optimal solution for large-scale services that have already been deployed. However, Parslo is limited in its support for only certain types of Directed Acyclic Graphs, and its performance may not be guaranteed in all circumstances. PEMA [4] uses iterative feedback-based tuning to optimize resource allocation to meet SLO requirements. Compared to other approaches, PEMA is lightweight and does not require any offline experiments or pre-training. However, PEMA's performance may be poor during resource update intervals, and its inability to capture the dependencies between microservices due to the lack of pre-training may limit its effectiveness. The fundamental limitation of this line of work is that they do not consider features of microservice chain, which can lead to inefficient actions and performance degradation.

Micoservice Autoscaling. Autopilot [10] utilizes ML algorithms to analyze historical data on prior executions and performs a set of finely-tuned heuristics to adjust a job's resource requirements while it is running. The benefit of Autopilot is its ability to modify resource requirements on-the-fly, allowing it to adapt to changing workload demands. However, Autopilot's conservative approach can lead to overprovisioning and resource wastage. Sinan [3] leverages a set of machine learning models to determine the performance impacts of microservice dependencies and allocate appropriate resources for each tier. Sinan is an explainable approach and can be used for complex microservices, while it only monitors CPU resources and does not provide auto-tuning capabilities. CoScal [12] leverages data-driven decisions and enables multi-faceted scaling based on reinforcement learning. CoScal utilizes gradient recurrent units to accurately predict workloads, which assists in achieving efficient scaling. However, one limitation is that the model re-training required for adapting to new applications can be costly. FIRM [9] is a system that utilizes online telemetry data and machine learning methods to adaptively detect and locate microservices that lead to SLO violations. It can make decisions based on reinforcement learning to mitigate SLO violations via fine-grained and dynamic resource provisioning. FIRM proposes a two-level ML framework to locate critical microservice paths and nodes. However, FIRM has certain limitations. The scalability of centralized



(a) (b)

Fig. 1. (a) Microservice graph structure of Sock Shop application. (b) CPU utilization of each microservice from top tier to bottom tier. When workloads increase, the CPU utilization of all microservices increases.

graph databases is limited, and it cannot handle transient SLO violations that occur within an interval shorter than the minimum interval due to the heavy ML techniques.

The comparisons between *ChainsFormer* and other relevant work are presented in Table 1. Our work is most similar to CoScal and FIRM. However, there are notable differences between them. Firstly, CoScal is deployed on Docker Swarm, while *ChainsFormer* is designed specifically for Kubernetes. Secondly, CoScal does not incorporate chain analysis for resource management, whereas *ChainsFormer* leverages chain analysis techniques to optimize resource allocation within microservice chains. Thirdly, both CoScal and *ChainsFormer* employ reinforcement learning, but *ChainsFormer* utilizes the SARSA algorithm, which allows for faster convergence by updating Q-values based on the current policy. In comparison to FIRM, *ChainsFormer* employs lightweight ML techniques to handle transient SLO violations in microservice chains, a task that FIRM does not address due to its heavy ML models and the associated high costs of model re-training. Additionally, *ChainsFormer* does not require a centralized graph database like FIRM, which enhances its scalability by avoiding a central bottleneck caused by large amounts of data. In *ChainsFormer*, runtime data is stored on worker nodes and only fetched by the central node when model training or retraining is required, significantly reducing the overhead on the central node.

3 The ChainsFormer Framework

To motivate our design, we deployed the Sock Shop application[1] to observe how different microservices react to changes in workloads by monitoring utilization usage. As shown in Fig. 1a, a request sent to the Sock Shop application can be distributed to different microservices from front-end to back-end tiers. The processing of a request can form different calling chains, for example, a request

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

can go through different chains to complete different functionalities, e.g. checking items under a user account (front-end → carts → users), or paying for an item (front-end → catalog → payment). As shown in Fig. 1b, workloads increase from 0–110 requests per second during 0–11 minutes (requests per second is increased with 10 after each minute), and the CPU utilization also increases for all microservices, while the resource usage propagation among the nodes in a chain is not consistent. Thus, to achieve efficient resource provisioning of microservices, the scheduler should consider the features of the microservice chain properly.

To address the above observations, we propose the overview architecture design of *ChainsFormer* as shown in Fig. 2 and the key designs are as below:

- *ChainsFormer* first processes the incoming requests from users via Workload Generator by recording the number of requests and extracting the tracing data and performance counters.
- To make the resource provisioning more efficient, *ChainsFormer* applies the neural network-based prediction algorithm to estimate future workloads.
- *ChainsFormer* detects SLO violations and utilizes real-time data to dynamically identify critical chains and locate critical nodes that result in SLO violations. To support the quick adaption to the dynamic changes in chains, *ChainsFormer* includes an auto-adaptor that can quickly detect the changes.
- *ChainsFormer* analyzes the telemetry data collected by Workload Generator and node information identified by Chains Analyzer, and makes scaling decisions to provision resources for critical nodes. The decision is made automatically on the Kubernetes cluster by an RL-based resource scaler, which considers resource utilization, performance metrics, and future workloads.

3.1 Workload Generator

Workload Generator module in *ChainsFormer* is responsible for processing the raw workload trace to make fit with other modules, e.g. extracting the key information of workloads (e.g. timestamp and user id) and providing initial analyses for the workloads. Based on the required functionalities, workloads are distributed to different microservices that are deployed on different work nodes in the microservices cluster. For example, we have observed that the workloads of the Sock Shop application are distributed to Front-end (45.5%), Order (22.7%), Carts (22.7%), Catalog (5.7%) and Random item (3.4%) with different percentages. The processed workloads are also regularly stored in log files for workloads prediction, and the performance counters that indicate system performance are provided to resource scalers for autoscaling microservices.

To reduce the state space of our RL model, we process the workloads by dividing the workloads into a number of levels, e.g. using CPU utilization levels to represent the number of workloads, where the same scaling actions can be applied to the same level to reduce action space. For example, Fig. 3 shows the original continuous Alibaba's workloads converted to 10 discrete CPU utilization levels at per-day and per-minute intervals, and each level represents 10% utilization, e.g. level 0 represents utilization ranges from 0% to 10%.

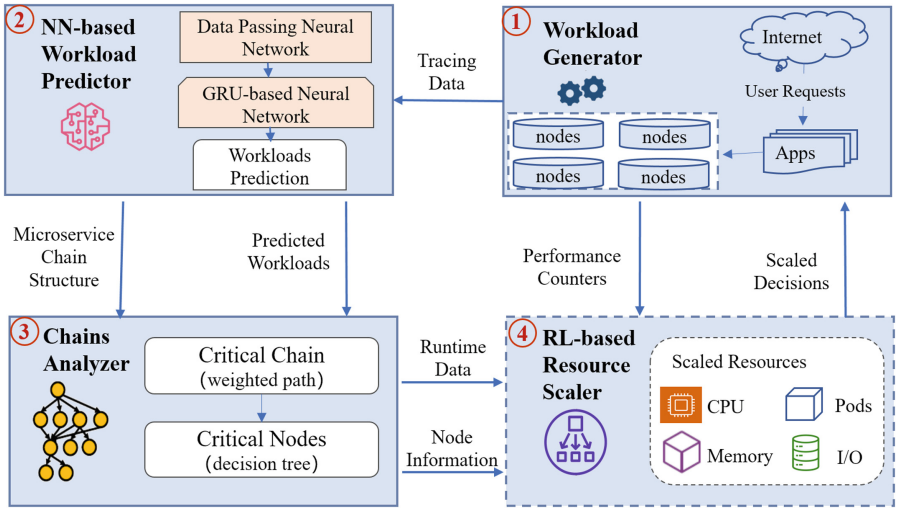


Fig. 2. Framework of *ChainsFormer*

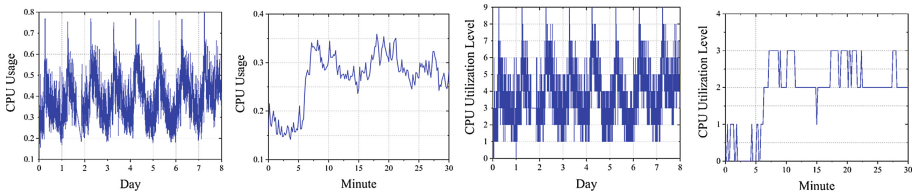


Fig. 3. (a) Original Alibaba per-day workloads. (b) Original Alibaba per-minute workloads. (c) Converted Alibaba per-day workloads. (d) Converted Alibaba per-minute workloads.

3.2 Neural Network-Based Workload Predictor

The Workload Predictor aims to accurately forecast the future workloads in system, and provides information for the RL-based Resource Scaler module to dynamically scale the number of pod replicates. The Workload Predictor module can be realized via different prediction approaches, such as ML-based prediction algorithms. *ChainsFormer* considers the workloads prediction as a category of multi-variate time series forecast problem, where the workloads are time-relevant and multiple variables (e.g. CPU usage, memory usage, network throughput, and hard disk read/write) can influence the final prediction results.

ChainsFormer utilizes a GRU-based neural network validated in [13], named esDNN, to predict future workloads, which can overcome the limitations of gradient explosion and disappearance when conducting long-term prediction. The esDNN can extract the key features of workloads, and convert multivariate time series forecasting into supervised learning to keep as much information as possi-

ble. The performance of esDNN has been validated to achieve good accuracy in predicting workloads.

3.3 Chains Analyzer

One of the main goals of *ChainsFormer* is to identify the critical chain efficiently and accurately based on tracing data and inter-dependencies, along with identifying the critical nodes that impact the latency of the critical chain. We define the critical chain as the one with the longest end-to-end latency, which represents the total time taken by a request to traverse the entire microservice chain, starting from the moment it enters the system until the user receives the response. Furthermore, the critical nodes (highlighted in Fig. 4a for Train-Ticket application) are defined as the nodes that have a substantial impact on the latency of the critical chain, and any performance degradation in these nodes can severely affect the performance of the microservices.

To identify the critical chain, *ChainsFormer* uses tracing data to construct an execution graph that shows the processing sequence of a user request. The graph includes all the microservices involved in processing the request. We then apply a weighted longest path algorithm [5] to find the critical chain, which is the chain with the longest end-to-end latency. The weight of each edge is the processing time between different nodes. This algorithm is lightweight and can adapt to changes in chains quickly. For example, if the blue chain in Fig. 4a has the highest latency, it can be identified as the critical chain. The critical chain will be changed to the red chain when its latency becomes to be the longest. We also identify critical nodes in the critical chain, which are the nodes that have a significant impact on latency. These critical nodes can significantly degrade the performance of microservices.

The critical nodes are identified based on a decision tree, as shown in Fig. 4b. This tree classifies the nodes into critical and non-critical based on real-time data from the selected critical microservice chain and a trained model using historical running data. To reduce the overhead on the central node, the runtime data is stored on worker nodes and only fetched by the central node when model training or retraining is required. Nodes with high latency, CPU, and memory usage are more likely to be classified as critical nodes. In case the identification has a high error rate (e.g. 5%), a model updating mechanism is triggered to update the decision tree.

3.4 RL-Based Resource Scaling

The resource scaler uses RL techniques to determine the optimal scaling actions. Compared to static and meta-heuristic approaches, the RL-based approach can effectively explore a larger solution space and respond to dynamic status changes. The RL-based resource scaler employs a hybrid scaling approach that includes both vertical scaling and horizontal scaling. Vertical scaling is used to quickly adjust resources such as CPU, memory, and network on the local machine, while horizontal scaling adds or removes active nodes in the system.

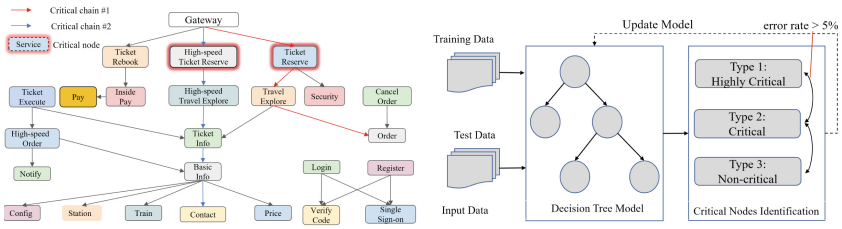


Fig. 4. (a) Train-Ticket application with critical chain (https://github.com/FudanSELab/train-ticket). (b) Decision tree model for critical node identification

In *ChainsFormer*, the problem of RL-based resource scaling is modeled as a Markov Decision Process [11]. At each time interval t, the system state is represented by $s_t \in S$, and an action $a_t \in A$ can be taken to transition the state to s_{t+1}, yielding a reward of R_{t+1} based on the policy π_θ, which has configurable parameters θ. The state space S is associated with an action space A, and a transition matrix captures the probability of taking different actions during state transitions. The goal of RL is to optimize the policy to maximize the expected cumulative reward.

To achieve this, *ChainsFormer* employs the SARSA algorithm [6] to learn the policy for the Markov decision process and estimate the expected cumulative reward of state-action pairs using the action-value function $Q_t(s, a)$. When action a_t is taken at time interval t, the value of Q_{t+1} is updated using the reward R_{t+1} and propagated to the next time interval as:

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha[R_{t+1} + \gamma max_{a'} Q_t(s_{t+1}, a') - Q_t(s_t, a_t)], \quad (1)$$

where $\alpha \in (0, 1]$ is the learning rate and $\gamma \in [0, 1]$ is the discount factor. To address the curse of dimensionality associated with updating the Q Table with a large solution space, we train the model offline to minimize the loss function and reduce training time. Online training is used to make decisions and update actions with rewards. We also employed the divided load levels to reduce the state space, as discussed in Sect. 3.1. In addition, we use the SARSA algorithm to further reduce computational costs by using $R_{t+1} + \gamma max_{a'} Q_t(s_{t+1}, a')$ as the update target to guide the estimate of the true action-value function. This approach considers only the sampling of successive s_{t+1}, a_{t+1}, and immediate reward R_{t+1}. The estimation of the action-value function at the time interval $t + 1$ is given by:

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha[R_{t+1} + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t)], \quad (2)$$

where the $Q_t(s_{t+1}, a_{a+1})$ and each update can be obtained via one-step transition $(s_t, a_t, R_{t+1}, s_{t+1}, a_{t+1})$ of the state-action-reward-state-action pair.

To implement the RL-based resource scaler module, we utilize various parameters of the current pod as inputs to the RL model. These parameters include the

load state, the position of the pod in the chain, and the latency of the microservice. The RL model considers the state $s_t \in S$ to represent the current status of the microservice chain, and action $a_t \in A$ comprises scaling operations that adjust the chain status and provisioned resources by a specific amount. We also assume the presence of a set of physical machines $P = (M_1, M_2, \ldots, M_K)$ in the system that provides resources. Each physical machine M_k is represented by a tuple $U_k = (u_k^1, u_k^2, \ldots, u_k^I)$, where u_k^i represents the resource utilization of type i out of a total of I resource types on physical machine M_k. For each M_k, we denote the set of possible actions as $a_k^i = \{h_k, v_k^i\}$, where $h_k \in [-n, n]$ represents the number of horizontal replicates that can be added or removed, $v_k^i \in [-m, m]$ represents the amount of vertical scaling that can be applied to resource type i. A positive value of h_k or v_k^i indicates that more resources are added, whereas negative values indicate resource removal. Given K as the total number of physical machines, the final set of actions is represented as the Cartesian product of the sub-action sets: $A = \prod_{k=1}^{K} \prod_{i=1}^{I} a_k^i$.

The main objective of the *ChainsFormer* system is to enhance resource utilization while ensuring QoS requirements are met. Therefore, the reward function is designed to consider two key metrics: resource utilization and response time. The reward for resource utilization is formulated in Eq. (3).

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

where U_k^{max} represents the highest utilization threshold of all resource types for physical machine M_k, and u_k is the current utilization of M_k. The system receives a positive reward when the utilization is below the threshold, and the reward decreases when the utilization is higher or significantly lower than the predefined threshold.

The reward for response time, denoted as $R_q(rt)$, is modeled based on the maximum acceptable response time RT_{max}.

$$R_q(rt) = \begin{cases} e^{-(\frac{rt - RT_{max}}{RT_{max}})^2}, & rt > RT_{max}, \\ 1, & rt \leq RT_{max}, \end{cases} \tag{4}$$

which shows that when the system is operating normally, the reward is 1. However, as the system's performance degrades and violates the RT_{max}, the reward gradually decreases and converges to 0.

The final reward value is based on the resource utilization R_u and response time R_q at time interval t, which is formulated as follows:

$$r(s_t, a_t) = \frac{R_q^t}{R_u^t}, \tag{5}$$

where higher values of R_q^t and lower values of R_u^t can increase the total reward.

Algorithm 1 outlines the overall procedure of *ChainsFormer*. Initially, the algorithm collects the system status to enable the RL process (line 1), which

Algorithm 1: *ChainsFormer*: Overall Procedure

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

1 Initialize system status, and monitoring model;
2 **for** t *from* 1 *to* T **do**
3 $U_t^k \leftarrow$ Resource utilization of M_k at time interval t;
4 $W_{t-1} \leftarrow$ Workloads level at time interval $t-1$;
5 $\hat{W}_t \leftarrow$ Predicted workload level;
6 **if** $\hat{W}_t \neq W_{t-1}$ **then**
7 Choose a action from action set A with ϵ probability, or select an action with the $\max(Q_t(s_t, a_t))$;
8 Conduct $a_t = \{h_k(t), v_k^i(t)\}$ with horizontal scaling and vertical scaling
9 **if** *online training is triggered* **then**
10 $s_{t+1} \leftarrow$ system state at time interval $t+1$;
11 $R_{t+1} \leftarrow$ reward calculation by Eq. (5);
12 Update Q value:
$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha[R_{t+1} + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t)]$;
13 **end**
14 Store transition $(s_t, a_t, R_{t+1}, s_{t+1}, a_{t+1})$in experience pool;
15 **end**
16 **end**

includes monitoring the workloads level, resource utilization, and metrics at each time interval to construct the complete system states (lines 3-5). Upon a change in workload level (line 6), resources are dynamically scaled to optimize resource usage while maintaining the required QoS. The SARSA algorithm commences by selecting actions randomly with a probability of ϵ from the experience pool and transitions to another state (line 7). The chosen actions entail vertical and horizontal scaling to allocate resources effectively (line 8). *ChainsFormer* facilitates online training by storing the transition $(s_t, a_t, R_{t+1}, s_{t+1}, a_{t+1})$ in the experience pool and subsequently updating the decisions based on rewards with better outcomes (lines 9–14).

4 Performance Evaluations

4.1 Experimental Settings

We use the workload dataset provided by Alibaba[2] as demonstrated in Sect. 3.1, which includes 8-day data traces from homogeneous 4,034 servers. We utilize the Locust toolkit to generate resource usage based on profiled data of machines. We evaluate the performance of the Train-Ticket application (a larger application

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

than the Sock Shop used for motivation in Sect. 3) and use the Jaeger monitoring toolkit to track the distribution of requests. The application is deployed on a Kubernetes-based cluster consisting of five nodes, each with an Intel Xeon E5-2660 processor and 64 GB of RAM. One physical machine serves as the master of the cluster, while the others serve as workers.

4.2 Baselines and Metrics

We have compared *ChainsFormer* (**CF**) with 3 state-of-the-art baselines implemented by us.

KS [1]**:** it is employed by native Kubernetes and mainly relies on horizontal scaling, which involves dynamically adding or removing the number of replicas. It follows a threshold-based approach based on resource usage metrics such as CPU and memory, where more replicas are added when the pre-defined resource threshold is exceeded (e.g. CPU utilization > 0.7) and vice versa.

AUTO [10]**:** it is derived from Google Autopilot and uses a hybrid approach to scale resources based on workloads. The approach combines horizontal scaling and vertical scaling to dynamically adjust the allocated resources to tasks based on historical data.

FIRM [9]**:** it utilizes machine learning techniques, specifically support vector machine and reinforcement learning, to identify and mitigate microservices responsible for SLO violations.

We have adopted three widely used metrics to evaluate the performance: *1) Requests per second (RPS)* represents the system's ability to process requests within a specific time period, and a higher value shows better performance. *2) Number of failures* indicates the number of requests that were not processed or did not receive a response due to an overloaded situation. A lower value for this metric represents a more reliable system. *3) Average response time:* is a dominant metric to measure performance, and a good autoscaling algorithm should aim to reduce it.

4.3 Experiment Analyses

Due to page limitations, we present key results. Figure 5a compares the average requests per second over different time periods. To highlight differences among periods, we analyze results over 5 periods (e.g. 1,000 minutes, 2,000 minutes, and 5,000 minutes), covering short-term and long-term comparisons. It is noteworthy that the loads significantly vary during different time periods. For instance, the highest loads were observed during the first 2,000 minutes, and the average load during the 5,000 minutes period was much lower. It is observed that the KS approach performs the worst in terms of RPS compared to other baselines. This could be due to the limited capability of the threshold-based approach. The AUTO approach, which leverages ML-based techniques, can process larger RPS compared to KS. The FIRM approach can obtain better RPS during the

first 3,000 minutes, but during the 4,000-5,000 minutes, it performs worse than AUTO. Our proposed approach, *ChainsFormer*, can achieve the best RPS in the long-term, i.e., when the time period is larger than 3,000 minutes. This optimization comes from our more accurate identification of critical chains and nodes. At the early stage of request processing, FIRM performs well when the critical path is identified. However, after load changes, the identified critical path may not be critical anymore. Additionally, it is reasonable to note that FIRM with a static critical path does not fit workloads with high variances well. In conclusion, CF optimized the requests per second up to 8.1% compared to the baselines.

Figure 5b illustrates the comparison of the number of failures, presenting the average results in five different time periods. It is observed that KS has the highest number of failures compared to other baselines due to its static policy, which shows that it struggles to handle high-variant workloads. AUTO significantly reduces the number of failures. For instance, during the first 1,000 minutes, AUTO reduces the failures from 350 to 80 by leveraging historical data. FIRM and CF further optimize the failures by utilizing critical chains and nodes, where the results are quite close. Overall, CF can reduce the number of failures by 8.3% compared to FIRM.

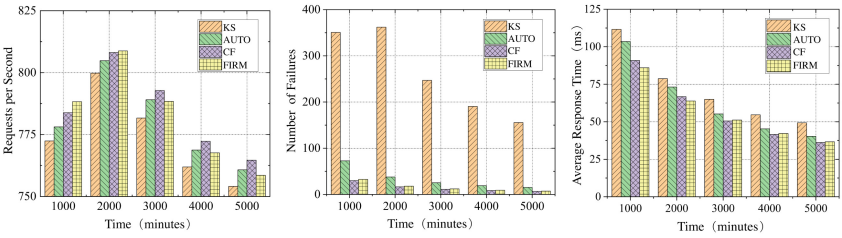
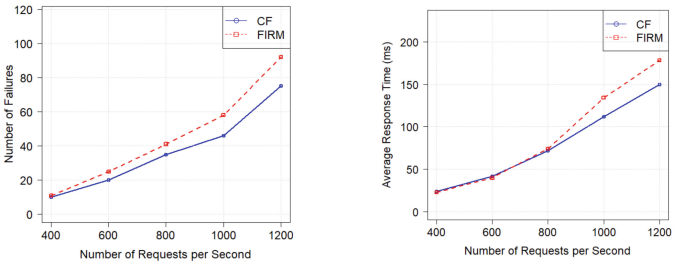


Fig. 5. Comparison of (a) requests per second, (b) number of failures, and (c) average response time.

Figure 5c depicts the comparison of average response time. Among all five time periods, the average response time of KS is at the highest value, which we consider as a benchmark test to analyze the performance of the other three algorithms. AUTO is optimized compared to KS and maintains the second-highest response time. It optimizes around 10% of response time, for example, decreasing from 110 ms to 100 ms during the first 1,000 minutes. The results of CF and FIRM are consistent with the analyses of RPS. In the early stage, FIRM slightly outperforms CF, while in the long run (e.g. 3,000 to 5,000 minutes), CF achieves a lower response time than FIRM. Overall, *ChainsFormer* optimizes response time by 1.4% to 26.6% compared to the baselines.

To evaluate the scalability of *ChainsFormer*, we conducted experiments comparing it with FIRM under different numbers of requests as shown in Fig. 6. We gradually increased the number of requests (from 400 to 1200 per second)



(a) (b)

Fig. 6. Scalability comparison of (a) number of failures, and (b) average response time when the number of requests increase.

and monitored the system's performance. The number of failures in *ChainsFormer* exhibited a slower rate of increase compared to FIRM as the number of requests grew. Similarly, the average response time in *ChainsFormer* remained relatively stable as the number of requests grew. In contrast, FIRM experienced a more pronounced increase in average response time under the same conditions. These findings validate the scalability of *ChainsFormer* and its ability to handle larger workloads while maintaining good performance. The results suggest that *ChainsFormer* is a promising solution for scaling microservice-based systems in scenarios with dynamic and growing request loads.

5 Conclusions

In this paper, we propose *ChainsFormer*, a microservice scaling approach that combines deep learning and reinforcement learning techniques to dynamically adjust resource allocation based on workload predictions and critical chain identification. By leveraging decision trees for rapid identification of critical chains and nodes, and using reinforcement learning to make real-time scaling decisions, *ChainsFormer* optimizes resource usage while maintaining high-quality of service in terms of response time, number of failures, and requests per second. Our experiments, conducted on a representative microservices application, show that *ChainsFormer* outperforms state-of-the-art algorithms from research and industry in terms of QoS optimization. Our approach has the potential to significantly improve the efficiency and reliability of microservices-based applications in cloud computing environments.

Acknowledgments. This work is supported by National Key R & D Program of China (No.2021YFB3300200), the National Natural Science Foundation of China (No. 62072451, 62102408), Shenzhen Industrial Application Projects of undertaking the National key R & D Program of China (No. CJGJZD20210408091600002), Shenzhen Science and Technology Program (No. RCBS20210609104609044), and Alibaba Group through Alibaba Innovative Research Program.

References

1. Burns, B., Beda, J., Hightower, K.: Kubernetes: up and running: dive into the future of infrastructure. O'Reilly Media (2019)
2. Gan, Y., Liang, M., Dev, S., et al.: Sage: practical and scalable ml-driven performance debugging in microservices. In: Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2021, pp. 135–151 (2021)
3. Gan, Y., Zhang, Y., Hu, K., et al.: Seer: leveraging big data to navigate the complexity of performance debugging in cloud microservices. In: Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systemsm ASPLOS 2019, pp. 19–33 (2019)
4. Hossen, M.R., Islam, M.A., Ahmed, K.: Practical efficient microservice autoscaling with qos assurance. In: Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2022, pp. 240–52 (2022)
5. Ioannidou, K., Nikolopoulos, S.D.: The longest path problem is polynomial on cocomparability graphs. Algorithmica **65**, 177–205 (2013)
6. Kardani-Moghaddam, S., Buyya, R., Ramamohanarao, K.: Adrl: a hybrid anomaly-aware deep reinforcement learning-based resource scaling in clouds. IEEE Trans. Parallel Distrib. Syst. **32**(3), 514–526 (2021)
7. Luo, S., Xu, H., Lu, C., et al.: An in-depth study of microservice call graph and runtime performance. IEEE Trans. Parallel Distrib. Syst. **33**(12), 3901–3914 (2022)
8. Mirhosseini, A., Elnikety, S., Wenisch, T.F.: Parslo: a gradient descent-based approach for near-optimal partial slo allotment in microservices. In: Proceedings of the ACM Symposium on Cloud Computing, SoCC 2021, pp. 442–457 (2021)
9. Qiu, H., Banerjee, S.S., Jha, S., et al.: {FIRM}: an intelligent fine-grained resource management framework for {SLO-Oriented} microservices. In: 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), pp. 805–825 (2020)
10. Rzadca, K., Findeisen, P., Swiderski, J., et al.: Autopilot: workload autoscaling at google. In: Proceedings of the Fifteenth European Conference on Computer Systems. EuroSys 2020 (2020)
11. Wang, S., Guo, Y., Zhang, N., et al.: Delay-aware microservice coordination in mobile edge computing: a reinforcement learning approach. IEEE Trans. Mob. Comput. **20**(3), 939–951 (2021)
12. Xu, M., Song, C., Ilager, S., et al.: Coscal: multifaceted scaling of microservices with reinforcement learning. IEEE Trans. Netw. Serv. Manage. **19**(4), 3995–4009 (2022)
13. Xu, M., Song, C., Wu, H., et al.: Esdnn: deep neural network based multivariate workload prediction in cloud computing environments. ACM Trans. Internet Technol. **22**(3) (2022)
14. Zhang, Y., Hua, W., Zhou, Z., et al.: Sinan: Ml-based and qos-aware resource management for cloud microservices. In: Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2021, pp. 167–181 (2021)
15. Zhong, Z., Xu, M., Rodriguez, M., et al.: Machine learning-based orchestration of containers: A taxonomy and future directions. ACM Comput. Surv. **54**(10s) (2022)