Reproduction Report of “Microservice Deployment in Edge Computing Based on Deep Q Learning”

Guo Jiawang

4th, January, 2025

Abstract

This reproduction report aims to reproduce the study titled“Microservice Deployment in Edge Computing Based on Deep Q Learning”.In the edge environment, container technology has been widely used in cloud computing and edge computing due to its lightweight, portability and efficient resource utilization.Although microservice deployment strategies are expected to shorten overall service response times. However, existing work has overlooked the impact of different interaction frequencies between microservices and the decrease in service execution performance caused by increased node load. This article proposes a method for deploying containers in edge environments based on the DQL algorithm RSDQL. DQL is an algorithm based on deep reinforcement learning that can handle control tasks in continuous action spaces. In the edge computing environment, the RSDQL algorithm can be used to optimize container deployment strategies to cope with complex environments and changing resource requirements.In the final part of the original paper, the author also proposed an algorithm called ES that scales containers automatically based on the resource load of each microservice.This report reproduces RSDQL as far as possible and replaced the ES algorithm proposed in the original text with Prometheus’ custom metrics

Keywords: Microservice,interaction awareness,load balancing,multi-objective model,DQL,Prometheus

1 Introduction

With the rapid development of the Internet of Things, traditional cloud computing can no longer meet the needs of service quality, latency sensitivity, and frequent interaction between IoT devices. Edge computing is a new computing paradigm composed of small distributed servers, which can make full use of the three-tier computing resources of terminal devices, edge nodes and cloud servers. In addition, driven by container technology, microservice architecture breaks down a monolithic application into multiple independent microservices that can coordinate with each other but also run independently. Loose coupled microservices are developed and maintained in a distributed and independent manner on computing platforms. As a result, service-oriented micro edge computing platform came into being, providing object detection micro services for driverless vehicles and low latency services for Internet of Things devices such as equipment fault detection for intelligent manufacturing systems. Although MSA has brought convenience to application development in edge computing, how to deploy microservices on resource constrained edge nodes is a major difficulty in current edge computing application development. Reducing service response time and improving service computing performance have become challenges. There have been many efforts to reduce service response time through service deployment strategies, where response time is the sum of processing time and data transmission time for each service. However, there are still some issues. Firstly, it is difficult to accurately calculate the data transmission time. It has been proven that the intuitive delay estimator used in existing service placement methods cannot accurately describe the transmission time affected by multiple microservices with complex dependencies. They proposed a data-driven end-to-end transmission time prediction method. However, in the process of building predictive models, collecting large-scale data through extensive experiments and generating training datasets after fine data processing is costly. The above tasks all aim to measure communication overhead by accurately calculating transmission time, which is a great challenge. On the contrary, Joseph used the fixed value of the relative ratio of communication scale as an indicator to measure communication overhead, which is simpler and more reasonable. Secondly, considering only interaction perception can lead to service deployment on one or a few nodes, resulting in a decrease in service execution performance. Therefore, in edge computing, the service execution performance degradation caused by the increase of node load must be considered.

2 Related works

In recent years, the deployment of microservices has received widespread attention. Capacity based cloud computing, as a new type of virtualization solution in cloud computing, provides a solution for microservices and offers a low-cost, secure, and isolated execution environment. Therefore, the deployment of containerized microservices has become a research hotspot. Researchers focus on developing deployment frameworks for specific scenarios to meet specific needs. For example, Wen et al. [1] proposed a reliable microservice orchestration that maximizes the satisfaction of security requirements and network QoS in cross regional distributed data centers. It is applicable to edge computing deployment with limited resources. To solve this problem, Deng et al. [2] proposed a method to minimize the cost of service deployment under the condition of mobile edge computing (MEC) resources. In addition, Li et al. [3] constructed the robust application deployment problem as a constrained optimization problem and proposed an approximate method for finding the approximate optimal solution. In addition, Faticant et al. [4] discussed how to deploy micro solutions in foggy environments in heterogeneous regions. The application of the service aims to minimize the throughput of the fog native application and maximize its utilization. Carlos et al. [5] considered goals such as load balancing, node reliability, and network overhead from the perspective of building a service deployment model to obtain the optimal container deployment strategy. Lin et al. [6] constructed a multi-objective optimization model with the goal of reducing network transmission overhead, balancing resource load, and improving business reliability. The above article only considers the physical distance difference between nodes when calculating the physical distance difference between nodes, ignoring the influence of the interaction frequency between microservices. Joseph et al. [7] first proposed a microservice deployment strategy for perceptual interaction, which focuses on the communication frequency between microservices rather than considering deployment nodes based on their physical distance. By deploying a large number of microservices on the same node, communication overhead between microservices can be minimized to the greatest extent possible. However, this may lead to resource imbalance between nodes and increase the number of applications. Total access latency. Therefore, based on interactive perception, we further consider load balancing and construct a multi-objective optimization model to avoid the influence of multiple factors. The problem of excessive utilization of resources by nodes leading to a decrease in microservice execution performance. From the perspective of solving models, reinforcement learning is important for solving strategic problems. The significant advantages in decision-making problems have been applied to microservice deployment. Chenit El. [8] Modeling the microservice deployment problem as a Markov decision process to find the optimal deployment strategy within a limited budget. Wang et al. [9] used an improved Q-learning algorithm to solve microservices on edge servers. Resolve issues to reduce overall service latency. Chen et al. [10] proposed a multi buffer depth deterministic policy gradient to provide service deployment solutions. To minimize the average response time. However, the above research overlooked the importance of inaccurate reward settings before deploying all microservices. Therefore, we propose RSDQL to calculate the reward at the last step of each episode and compare it with the steps it shares to achieve better model convergence.

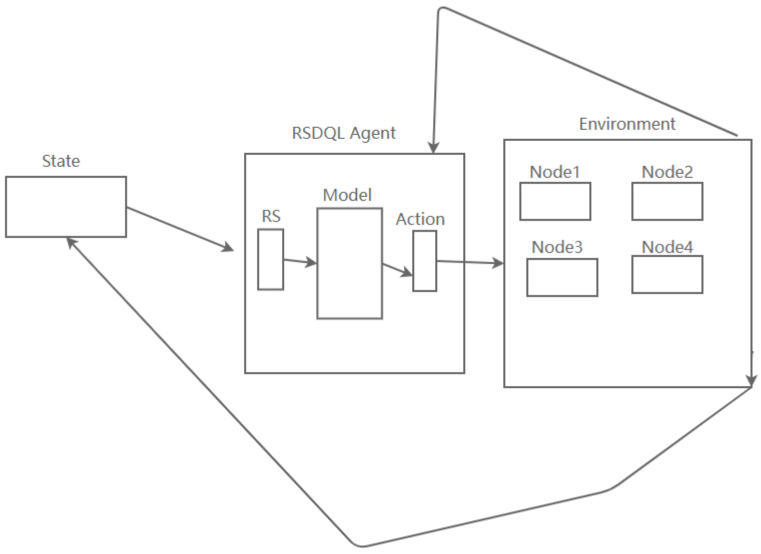


Figure 1. RSDQL framework

3 METHOD

3.1 DESIGN OF RSDQL

The specific contents of each part of our RSDQL algorithm are as follows,and its framework is shown in Figure 1.

State: The state space includes the states of containers and nodes in the cluster. We use S=(NodeS, NodeRes, PCont, UPCont, ContRes) to represent the state space. NodeS represents the container deployment status of the node, NodeRes represents the resource usage of all nodes, PCont represents deployed containers and their microservices, UPCont represents undeployed containers and their microservices, and ContRes represents the resource request status of the container.

Agent: RSDQL Agent takes an action to obtain a new state and reward, which is used to update the parameters of the Q network and make the neural network converge.

Reward: The RSDQL algorithm proposed in this article adds a mechanism called reward sharing on the basis of the DQL algorithm. Reward sharing refers to sharing the reward of the last step of a episode of microservice container deployment with other steps. Because the final deployment effect is decisive for the entire process of microservice deployment, and the rewards obtained from the previous deployment steps may not be accurate. The specific implementation is to assign the reward obtained in the last step to the previous steps after the end of a episode.Represent it as follows using a formula: $\mathrm{step}_i^j = rw_i, \forall j \in \{1, \ldots, M\}$.

Action: The action in RSDQL means deploying a container instance on top of the node.

3.2 DESIGN OF ES

Prometheus is an open-source monitoring system and time series database widely used for performance monitoring in cloud native environments. It periodically collects metric data from the target system by pulling models and stores this data in a local time series database. Prometheus provides a powerful query language PromQL, allowing users to flexibly query and aggregate data, generate charts and alerts. Prometheus Adapter is a native Kubernetes component that bridges the horizontal auto scaling capabilities of Prometheus and Kubernetes. Through Prometheus Adapter, HPA can dynamically adjust the number of Pods based on custom Prometheus metrics, achieving more refined resource management and automatic scaling. This allows applications to automatically expand based on actual load conditions, improving system resilience and efficiency.

The workflow diagram of Prometheus and Prometheus Adapter is shown in Figure 2. Prometheus regularly collects various metrics from Kubernetes and stores them in the database. Prometheus Adapter queries the corresponding data from the Prometheus database based on our custom query statements and exposes the interface to Kubernetes. Kubernetes cluster implements HPA functionality based on this interface. In the end, we replaced the ES algorithm mentioned in the paper with HPA implemented using custom metrics.

We define a metric called custom_desource_usage, which is obtained by weighting the memory utilization and CPU utilization of the pod, and can be expressed as:$custom\_resource\_usage = U_{CPU} \times 0.5 + U_{mem} \times 0.5$.

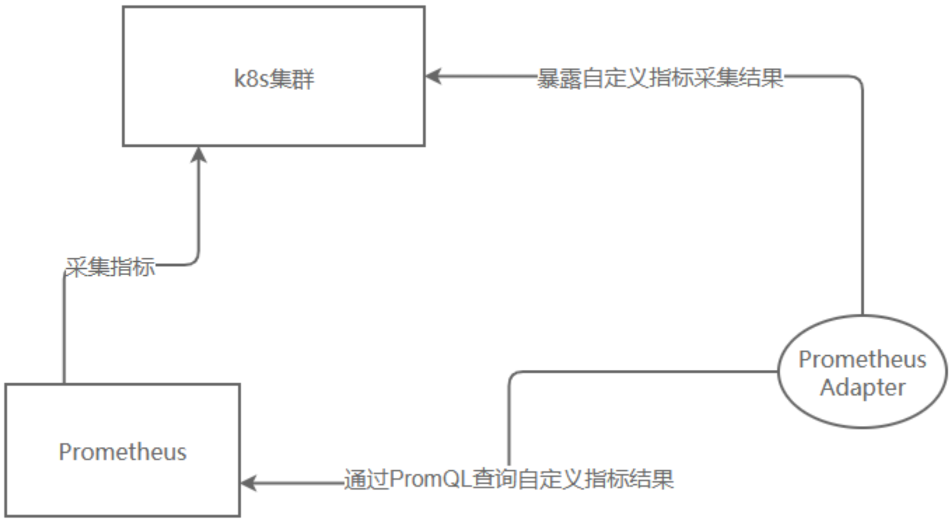


Figure 2. Prometheus workflow

4 Implementation details

4.1 Comparing with the released source codes

I borrowed some of the author's code in the process of implementing the paper. The code I borrowed was part of the design of the RSDQL algorithm, mainly the design of the virtual environment and the design of the neural network. I implemented the remaining parts based on borrowing the source code. In addition, I adjusted some code parameters based on my actual situation, such as buffer size, number of servers, and so on. Furthermore, I have implemented the interaction between the RSDQL algorithm and a real Kubernetes cluster. Through Kubernetes' Python API, I automatically convert the output results of RSDQL into container placement actions in the Kubernetes cluster. In addition, I also implemented the baseline algorithm IA mentioned in the paper, which reduces latency between microservices by deploying all microservices to the same node. Finally, I independently implemented custom metrics for Prometheus to achieve automatic scaling of microservice instances and replace the ES algorithm mentioned in the article.

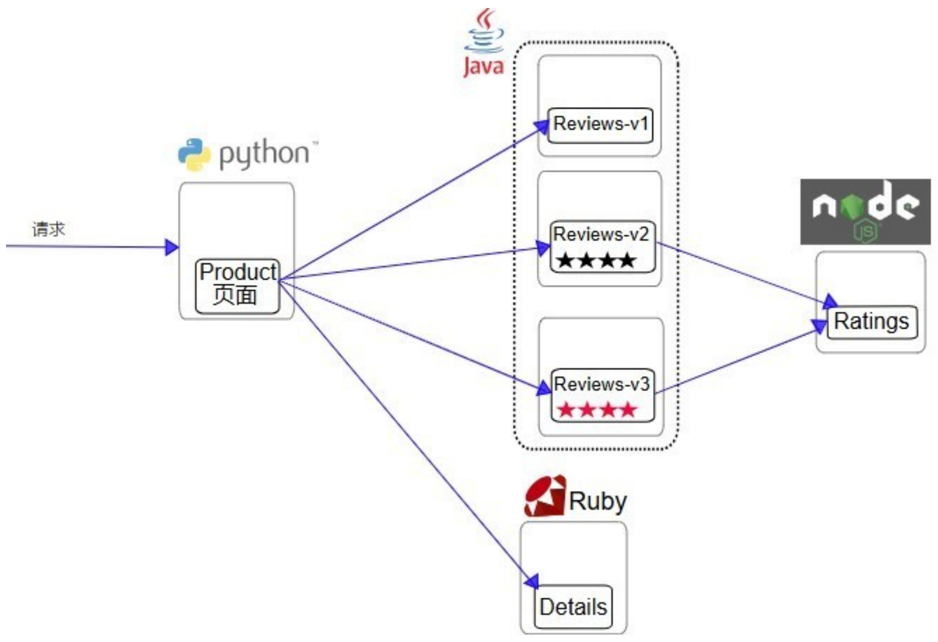


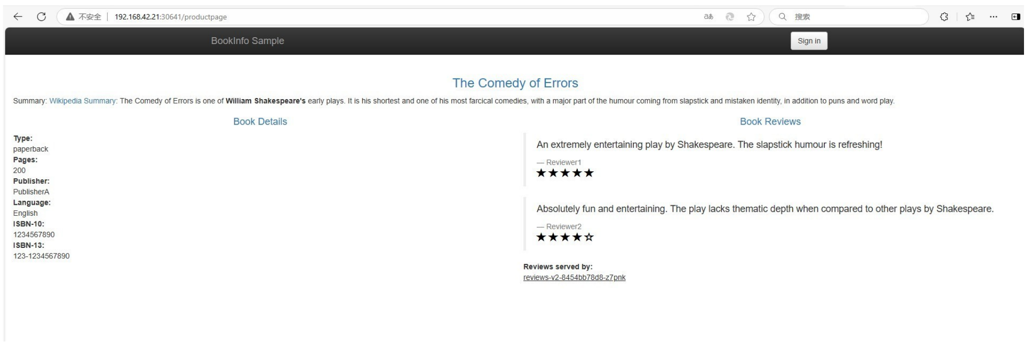
Figure 3. BookInfo framework

4.2 Experimental environment setup

We have built a Kubernetes cluster with one master and five slaves on VMware. The configuration of the node is 4-core, 4GB of memory, 20GB of disk, and 64 bit CentOS 7.4 operating system. The deployed application is the bookinfo application of Istio, which is an application containing 4 microservices and 6 containers. Its function is a display page for detailed book information. The architecture diagram of the Bookinfo application is shown in Figure 3.In addition, we use the Jmeter tool to conduct stress testing on the Bookinfo application. Jmeter is an interface testing tool developed in Java language, mainly used for performance testing. The baseline methods we use for comparison include IA method and k8s default scheduler method. The IA method requires deploying all nodes on one or two nodes to minimize the latency of microservices. The default scheduler method of k8s tends to deploy containers based on the resource usage of each node.

4.3 Interface design

As can be seen, Figure 4 shows the main interface of the bookinfo application. The left side of the interface displays detailed information about the book, while the right side shows the book's rating and comments. We can obtain detailed information interfaces for different books by constantly refreshing the page.



The Comedy of Errors

Summary: Wikipedia Summary: The Comedy of Errors is one of **William Shakespeare's** early plays. It is his shortest and one of his most farcical comedies, with a major part of the humour coming from slapstick and mistaken identity, in addition to puns and word play.

Book Details

Type:
paperback
Pages:
200
Publisher:
PublisherA
Language:
English
ISBN-10:
1234567890
ISBN-13:
123-1234567890

Book Reviews

An extremely entertaining play by Shakespeare. The slapstick humour is refreshing!
— Reviewer1
★★★★★

Absolutely fun and entertaining. The play lacks thematic depth when compared to other plays by Shakespeare.
— Reviewer2
★★★★☆

Reviews served by:
reviews-v2-8454bb78d8-z7pnk

Figure 4. BookinfoProductpage

4.4 Main contributions

My contributions include designing the RSDQL algorithm and implementing its interaction with Kubernetes clusters. In addition, the parameters of the algorithm were adjusted according to the actual situation, such as the number of nodes, buffer size, and so on. Also, I implemented custom scaling metrics through Prometheus Adapter and used this metric to automatically scale the number of microservices, replacing the ES algorithm in the paper.

5 Results and analysis

5.1 Deployments ofThree Strategies

As shown in Figure 5, the deployment nodes of containers under different strategies are presented. It can be seen that under the baseline method IA, all containers are deployed to nodes with index 0, as the IA method aims to minimize latency. Under other methods, the container was deployed to different nodes.

5.2 Comparison of Response Time

As shown in Figure 6, the latency of requests under different deployment strategies is displayed at different concurrent request counts. From the graph, we can see that our RSDQL algorithm is superior to the default scheduling method of Kubernetes, but slightly inferior to the IA algorithm. This is because the IA algorithm deploys containers on a single node to minimize transmission latency. The RSDQL algorithm simultaneously balances latency and resource load, so it cannot achieve optimal performance.

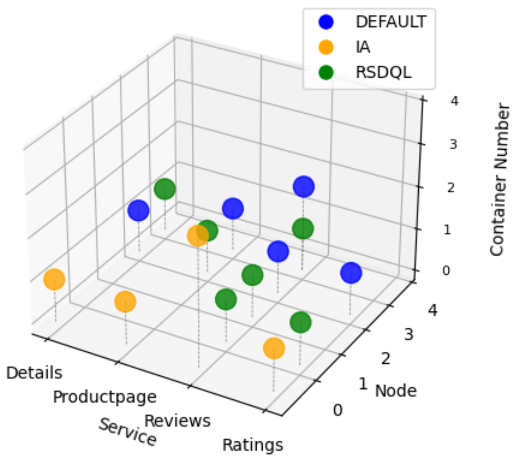


Figure 5. Deployments of three strategies

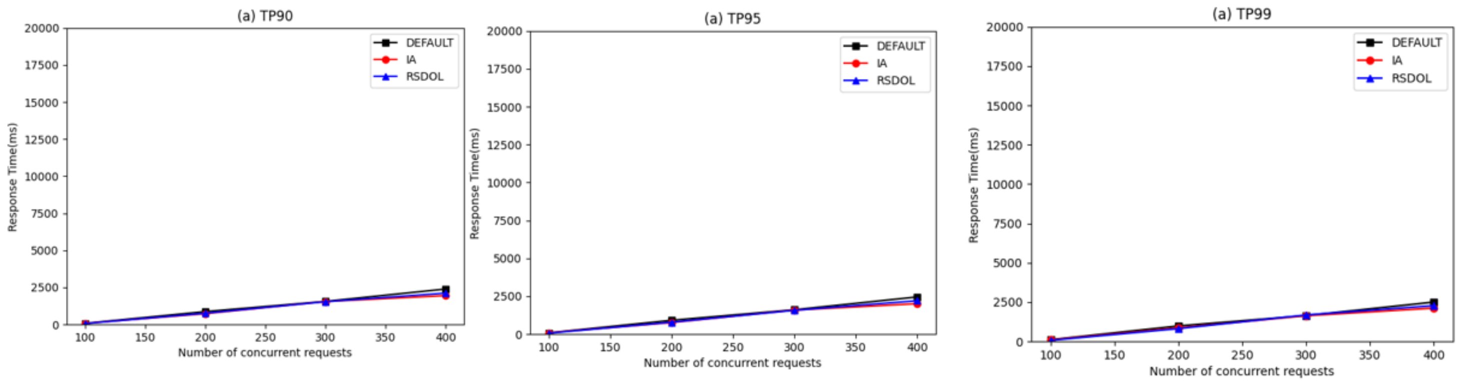


Figure 6. Performance comparison with different top percentile indicators

5.3 Memory Occupancy and Load Balancing

As shown in Figure 7, the total memory capacity occupied by containers and the memory occupancy variance of each container under different deployment strategies are presented for a concurrent request rate of 100 per second. From the graph, we can see that under the IA deployment strategy, the total memory occupied by the container is the highest, and the variance of container memory is the largest. The surface IA method performs the worst in considering resource load balancing, while our RSDQL algorithm has similar memory usage and variance to the default scheduler algorithm of Kubernetes.

5.4 Scalability

Due to limited computing resources allocated to containers, some container resources are depleted as the number of requests increases. Subsequently, ES is applied to ensure the scalability of service execution. Our custom metric custom_desource_usage enables real-time scaling of containers based on their CPU and memory utilization. The changes in the number of instances of different microservices in Bookinfo at different concurrent request counts per second are shown in Figure 8. It can be seen from the figure that as the number of concurrent requests per second increases, our container instances also increase, achieving automatic container scaling.

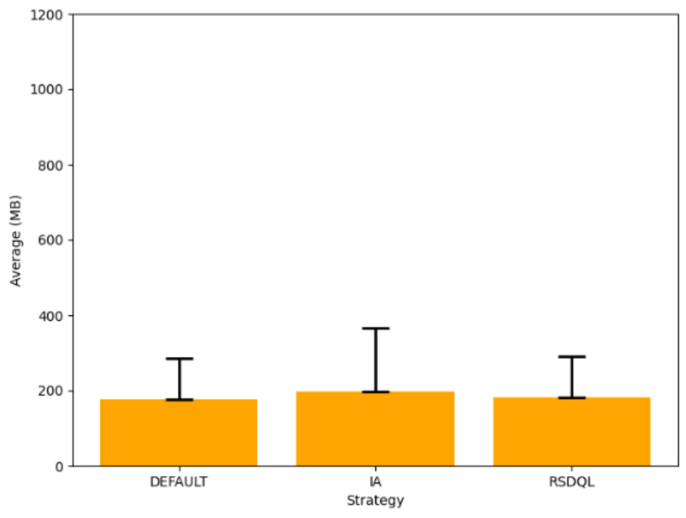


Figure 7. The comparison of memory occupancy and variance

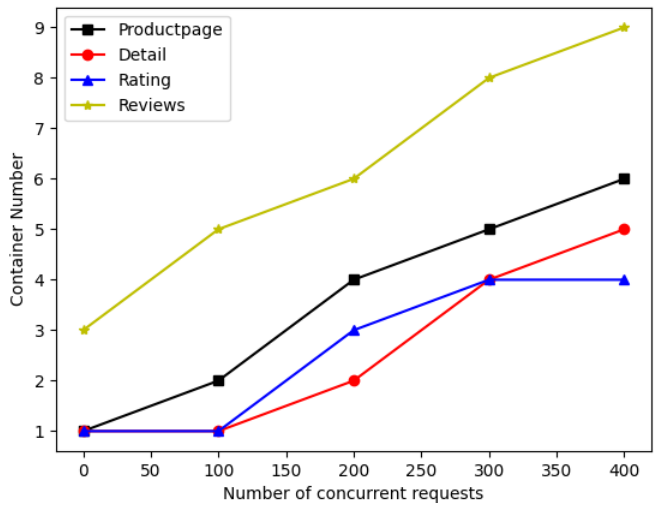


Figure 8. The comparison of memory occupancy and variance

6 Conclusion and future work

The original paper introduced the deployment of multi-objective microservices in edge computing, and further considered load balancing to build a multi-objective optimization model, designed a deep Q learning algorithm RSDQL based on reward sharing mechanism to solve the MMDP problem. The experimental results on Kubernetes clusters show that RSDQL can reduce average response time, balance load, and improve service scalability. Custom Prometheus metrics can automatically scale based on container resource load. However, there is still room for improvement in deploying microservices in edge computing. Our future work is to further research better reward sharing methods to replace the overly simplistic and crude assignment methods in the original text. In addition, we can consider more constraints to improve the deployment effectiveness of microservices and achieve better service response latency. Finally, we can also consider more advanced reinforcement learning algorithms to solve this microservice problem. These tasks will be carried out step by step in the future.

References

[1] Z. Wen et al., “GA-PAR: Dependable microservice orchestration framework for geo-distributed clouds,” IEEE Trans. Parallel Distrib. Syst., vol. 31, no. 1, pp. 129–143, Jan. 2020.
[2] S. Deng et al., “Optimal application deployment in resource constrained distributed edges,” IEEE Trans. Mobile Comput., vol. 20, no. 5, pp. 1907–1923, May 2021.
[3] B. Li et al., “READ: Robustness-oriented edge application deployment in edge computing environment,” IEEE Trans. Services Comput., to be published, doi: 10.1109/TSC.2020.3015316.
[4] F. Faticanti, F. De Pellegrini, D. Siracusa, D. Santoro, and S. Cretti, “Throughput-aware partitioning and placement of applications in fog computing,” IEEE Trans. Netw. Service Manag., vol. 17, no. 4, pp. 2436–2450, Dec. 2020.
[5] C. Guerrero, I. Lera, and C. Juiz, “Genetic algorithm for multi-objective optimization of container allocation in cloud architecture,” J. Grid Comput., vol. 16, no. 1, pp. 113–135, 2018.
[6] M. Lin, J. Xi, W. Bai, and J. Wu, “Ant colony algorithm for multi-objective optimization of container-based microservice scheduling in cloud,” IEEE Access, vol. 7, pp. 83088–83100, 2019.
[7] C. T. Joseph and K. Chandrasekaran, “IntMA: Dynamic interaction-aware resource allocation for containerized microservices in cloud environments,” J. Syst. Archit., vol. 111, 2020, Art. no. 101785.
[8] L. Chen, J. Xu, S. Ren, and P. Zhou, “Spatio–temporal edge service placement: A bandit learning approach,” IEEE Trans. Wireless Commun., vol. 17, no. 12, pp. 8388–8401, Dec. 2018.
[9] S. Wang, Y. Guo, N. Zhang, P. Yang, A. Zhou, and X. S. Shen, “Delay-aware microservice coordination in mobile edge computing: A reinforcement learning approach,” IEEE Trans. Mobile Comput., vol. 20, no. 3, pp. 939–951, Mar. 2021.
[10] L. Chen et al., “IoT microservice deployment in edge-cloud hybrid environment using reinforcement learning,” IEEE Internet Things J., vol. 8, no. 16, pp. 12610–12622, Aug. 2021.
[11] Q. He et al., “A game-theoretical approach for user allocation in edge computing environment,” IEEE Trans. Parallel Distrib. Syst., vol. 31, no. 3, pp. 515–529, Mar. 2020.
[12] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, “Edge computing: Vision and challenges,” IEEE Internet Things J., vol. 3, no. 5, pp. 637–646, Oct. 2016.
[13] C. Zhang, X. Liu, X. Zheng, R. Li, and H. Liu, “FengHuoLun: A federated learning based edge computing platform for cyber-physical systems,” in Proc. IEEE Int. Conf. Pervasive Comput. Commun. Workshops, 2020, pp. 1–4.
[14] I. Nadareishvili, R. Mitra, M. McLarty, and M. Amundsen, Microservice Architecture: Aligning Principles, Practices, and Culture. Newton, MA, USA: O’Reilly Media, Inc., 2016.
[15] L. Bao, C. Wu, X. Bu, N. Ren, and M. Shen, “Performance modeling and workflow scheduling of microservice-based applications in clouds,” IEEE Trans. Parallel Distrib. Syst., vol. 30, no. 9, pp. 2114–2129, Sep. 2019.
[16] S. Wang, Y. Guo, N. Zhang, P. Yang, A. Zhou, and X. S. Shen, “Delay-aware microservice coordination in mobile edge computing: A reinforcement learning approach,” IEEE Trans. Mobile Comput., vol. 20, no. 3, pp. 939–951, Mar. 2021.
[17] Y. Wang et al., “MPCSM: Microservice placement for edge-cloud collaborative smart manufacturing,”

IEEE Trans. Ind. Informat., vol. 17, no. 9, pp. 5898–5908, Sep. 2021.
[18] M. Alam, J. Rufino, J. Ferreira, S. H. Ahmed, N. Shah, and Y. Chen, “Orchestration of microservices for IoT using docker and edge computing,” IEEE Commun. Mag., vol. 56, no. 9, pp. 118–123, Sep. 2018.
[19] A. Brogi, S. Forti, C. Guerrero, and I. Lera, “How to place your apps in the fog: State of the art and open challenges,” Softw.: Pract. Experience, vol. 50, no. 5, pp. 719–740, 2020.
[20] P. Smet, B. Dhoedt, and P. Simoens, “Docker layer placement for on-demand provisioning of services on edge clouds,” IEEE Trans. Netw. Service Manag., vol. 15, no. 3, pp. 1161–1174, Sep. 2018.
[21] A. Samanta and J. Tang, “Dyme: Dynamic microservice scheduling in edge computing enabled IoT,” IEEE Internet Things J., vol. 7, no. 7, pp. 6164–6174, Jul. 2020.
[22] L. Yin, J. Luo, and H. Luo, “Tasks scheduling and resource allocation in fog computing based on containers for smart manufacturing,” IEEE Trans. Ind. Informat., vol. 14, no. 10, pp. 4712–4721, Oct. 2018.
[23] X. Li, J. Wan, H.-N. Dai, M. Imran, M. Xia, and A. Celesti, “A hybrid computing solution and resource scheduling strategy for edge computing in smart manufacturing,” IEEE Trans. Ind. Informat., vol. 15, no. 7, pp. 4225–4234, Jul. 2019.
[24] Y. Wang, K. Wang, H. Huang, T. Miyazaki, and S. Guo, “Traffic and computation co-offloading with reinforcement learning in fog computing for industrial applications,” IEEE Trans. Ind. Informat., vol. 15, no. 2, pp. 976–986, Feb. 2019.
[25] X. Wan, X. Guan, T. Wang, G. Bai, and B.-Y. Choi, “Application deployment using microservice and docker containers: Framework and optimization,” J. Netw. Comput. Appl., vol. 119, pp. 97–109, 2018.
[26] S. Tully, GitHub Repository, Github, 2013. [Online]. Available: https://github.com/shanet/Irrational/
[27] Y. Mao, J. Oak, A. Pompili, D. Beer, T. Han, and P. Hu, “DRAPS: Dynamic and resource-aware placement scheme for docker containers in a heterogeneous cluster,” in Proc. IEEE 36th Int. Perform. Comput. Commun. Conf., 2017, pp. 1–8.
[28] RK3399, “RK3399,” Firefly, 2014. [Online]. Available: https://www.t-firefly.com/product/clusterserver.html/
[29] M. Abdullah, W. Iqbal, and A. Erradi, “Unsupervised learning approach for web application auto-decomposition into microservices,” J. Syst. Softw., vol. 151, pp. 243–257, 2019.
[30] B. Tan, H. Ma, and Y. Mei, “A NSGA-II-based approach for multi-objective micro-service allocation in container-based clouds,” in Proc. 20th IEEE/ACM Int. Symp. Cluster Cloud Internet Comput., 2020, pp. 282–289.
[31] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, “A review of auto-scaling techniques for elastic applications in cloud environments,” J. Grid Comput., vol. 12, no. 4, pp. 559–592, 2014.
[32] R. S. Sutton and A. G. Barto, Reinforcement Learning: An Introduction. Cambridge, MA, USA: MIT press, 2018.
[33] Z. Wang and M. O’Boyle, “Machine learning in compiler optimization,” Proc. IEEE, vol. 106, no. 11, pp. 1879–1901, Nov. 2018.
[34] Istio, “Bookinfo application,” 2021. [Online]. Available: https://istio.io/latest/docs/examples/bookinfo/
[35] V. Yussupov, J. Soldani, U. Breitenb€ucher, A. Brogi, and F. Leymann, “Faasten your decisions: A classification framework and technology review of function-as-a-service platforms,” J. Syst. Softw., vol. 175, 2021, Art. no. 110906.
[36] L. Toka, G. Dobreff, B. Fodor, and B. Sonkoly, “Machine learning-based scaling management for kubernetes

edge clusters," IEEE Trans. Netw. Service Manag., vol. 18, no. 1, pp. 958–972, Mar. 2021.

[37] Kubernetes, "Kubernetes," 2020. [Online]. Available: https://kubernetes.io/zh/docs/home/

[38] O. Mart, C. Negru, F. Pop, and A. Castiglione, "Observability in kubernetes cluster: Automatic anomalies detection using prometheus," in Proc. IEEE 22nd Int. Conf. High Perform. Comput. Commun., IEEE 18th Int. Conf. Smart City, IEEE 6th Int. Conf. Data Sci. Syst., 2020, pp. 565–570.

[39] Apache JMeter, "JMeter," 2021. [Online]. Available: https://jmeter.apache.org/

[40] J. Dean and L. A. Barroso, "The tail at scale," Commun. ACM, vol. 56, no. 2, pp. 74–80, 2013.

[41] W. Voorsluys, J. Broberg, S. Venugopal, and R. Buyya, "Cost of virtual machine live migration in clouds: A performance evaluation," in Proc. IEEE Int. Conf. Cloud Comput., 2009, pp. 254–265.

[42] R. Ramakrishnan and A. Kaur, "Performance evaluation of web service response time probability distribution models for business process cycle time simulation," J. Syst. Softw., vol. 161, 2020, Art. no. 110480.

[43] T. Zheng et al., "Smartvm: A sla-aware microservice deployment framework," World Wide Web, vol. 22, no. 1, pp. 275–293, 2019.

[44] C. Qu, R. N. Calheiros, and R. Buyya, "Auto-scaling web applications in clouds: A taxonomy and survey," ACM Comput. Surv., vol. 51, no. 4, pp. 1–33, 2018.

[45] Amazon elastic compute cloud, 2021. [Online]. Available: http://aws.amazon.com/ec2

[46] Y. Gan, 2019. "DeathStarBench," GitHub Repository, 2019. [Online]. Available: https://github.com/delimitrou/Deat

[47] Y. Gan et al., "An open-source benchmark suite for microservices and their hardware-software implications for cloud edge systems," in Proc. 24th Int. Conf. Archit. Support Program. Lang. Oper. Syst., 2019, pp. 3–18.

[48] Y. Zhang, W. Hua, Z. Zhou, G. E. Suh, and C. Delimitrou, "Sinan: ML-based and QoS-aware resource management for cloud microservices," in Proc. 26th ACM Int. Conf. Archit. Support Prog. Lang. Oper. Syst., 2021, pp. 167–181.

[49] G. Tene, "wrk2," GitHub Repository, 2019. [Online]. Available: https://github.com/giltene/wrk2/