2022 14th International Conference on COMmunication Systems & NETworkS (COMSNETS) | 978-1-6654-2104-1/22/$31.00 ©2022 IEEE | DOI: 10.1109/COMSNETS53615.2022.9668432

Automating the Selection of Container Orchestrators for Service Deployment

Pravar Chaurasia*, Shubha Brata Nath†, Sourav Kanti Addya‡ and Soumya K Ghosh§
Department of Computer Science and Engineering
*‡National Institute of Technology Karnataka, Surathkal, India †§Indian Institute of Technology Kharagpur, India
Email: praverchaurasia@gmail.com, nath.shubha@gmail.com, souravkaddya@nitk.edu.in, skg@cse.iitkgp.ac.in

***Abstract*—With the ubiquitous usage of cloud computing, the services are deployed as a virtual machine (VM) in cloud servers. However, VM based deployment often takes more amount of resources. In order to minimize the resource consumption of service deployment, container based lightweight virtualization is used. The management of the containers for deployment is a challenging problem as the container managers need to consume less amount of resources while also catering to the needs of the clients. In order to choose the right container manager, we have proposed an architecture based on the application and user needs. In the proposed architecture, we have a machine learning based decision engine to solve the problem. We have considered docker containers for experimentation. The experimental results show that the proposed system can select the proper container manager among docker compose based manager and Kubernetes.**

***Index Terms*—Containers, Docker Compose, Kubernetes, Service Deployment.**

I. Introduction

The increasing popularity of cloud computing [1] has happened as it provides an on-demand large pool of resources to the users. The users prefer to deploy their services in less amount of time in the cloud. Though virtual machine (VM) [2] based deployment is done in cloud computing, it has issues as it consumes much more amount of resources. That is why the developers have shifted from VM based deployment to container based deployment. Container [3]–[5] is a virtualization technology where the resource footprint is very less in comparison to VMs. With the increasing usage of container orchestration technologies over the cloud, choosing the container orchestrator that fits the application's requirement has become a challenging problem. Let us consider an example of a web application. Whenever the web application is serving limited number of clients, it is the container manager that has the less resource consumption would be chosen. However, with the increase with the number of clients, the container manager that has the best option of scaling up would dominate the other container managers. This choice of correct container orchestrator is a *motivation* in container based service deployment in cloud. Therefore, there is a requirement of designing a *deployment decision module* for picking up the right container manager that can easily handle different containers and their interactions.

In cloud service deployment, the delay of serving the client requests has to be very less. We mention the *challenges* of the container based service deployment problem in cloud computing as follows.

- Container based deployment becomes difficult when the container manager incurs more amount of time while communicating with its different modules.
- Another issue of taking more amount of resources while monitoring the deployed containers is present in the container managers.

The development of micro-service [6], [7] based applications in cloud computing requires multi-container application deployment. Among the different container managers present for service deployment, `Kubernetes`[1] is most popular due to its many advantages. However, the resource consumption is more in Kubernetes. The works of [8] have compared the performance of `docker swarm`[2] and Kubernetes. They concluded that the resource consumption is less in docker swarm than Kubernetes. However, Kubernetes [9], [10] has many more features. In [11], the authors have compared different container orchestrators (i.e. *Kubernetes*, *Cattle*, *Apache Mesos*, and *Docker Swarm*). The work showed that Kubernetes is superior for complex applications; however, other container managers are better for simple applications. Again, `docker compose`[3] is a tool for managing multi-container docker applications.

Contributions: In this work, we analyze the performance of the container managers [12] regarding service deployment. We perform an analysis considering *docker* [13] containers. The key contributions of this work are as follows.

- We propose a system architecture for choosing a container manager based on the application and user needs. In order to classify the applications suitably for deployment either in Kubernetes or in Docker Compose, we incorporate a *decision engine* module based on machine learning. We develop the application deployment modules using docker compose and Kubernetes respectively.
- A multi-container application is considered for evaluation where the front-end is a `PHP`[4] based web application container and the back-end is a `MySQL`[5] database container.

[1]https://kubernetes.io/
[2]https://docs.docker.com/engine/swarm/
[3]https://docs.docker.com/compose/
[4]https://www.php.net/
[5]https://www.mysql.com/

- We implement *K-Nearest Neighbour* [14] and *Logistic Regression* [15] as the machine learning models in experimentation to automate the selection of container orchestrator for an application.

To the *best of our knowledge*, this work is the *first attempt* to automate the selection of container manager among docker compose based manager and Kubernetes. We present an overview of the container managers and proposed system architecture in Section II. The machine learning models considered are also given in this section. After that, the experimental evaluation is presented in Section III. Finally, the paper is concluded in Section IV.

II. Proposed System for Choosing Container Manager

In this section, we discuss how container orchestration happens in the cloud. After that, we present our proposed system in detail.

A. Container Orchestration

The container orchestrators are used to deploy, manage, and remove the containers having different micro-services [16], [17]. Containerized application orchestration depends on multiple factors. These are as follows.

- We need to define all the independent segments of an application. These independent segments are deployed as containers. For example, a web based application has a front-end (HTML, PHP, etc.) that requires a web server (Apache-PHP container). The web application also has a back-end that requires a database server (MySQL container).
- There is a requirement to do a connection between the segments (containers).
- We need to perform backing up of the container's data. For example, if a container fails, all of its data can be lost. So, we require a volume to back up the data of the container. The data needs to be available as soon as we create a new container.

B. System Architecture

Based on the above discussion, we now present the proposed system architecture in detail. The system architecture is depicted in Figure 1. In the proposed system, we have four layers. The first layer gathers the information required to classify the applications for deployment using either *docker compose based manager* or *Kubernetes orchestrator*. These are the input features like *available resources*, *load balancing*, *multi host deployment*, *automation on container scaling*, *rolling update*, and *delay requirement*. The input features are forwarded to the second layer. The second layer has a *deployment decision engine* to predict a container manager based on the input features. The *deployment decision engine* has a machine learning model for prediction. The machine learning algorithms [18] are used in cloud computing domain to solve many complex problems due to its advantage of predicting outcome. The choice of container manager has to be performed in less amount of time. This choice is based on the performance of the system which is obtained by observation. In the third layer, we get as output a preferable container orchestrator that should be used for a specific type of application in the cloud. In the last layer, the applications are deployed using the predicted container orchestrator.

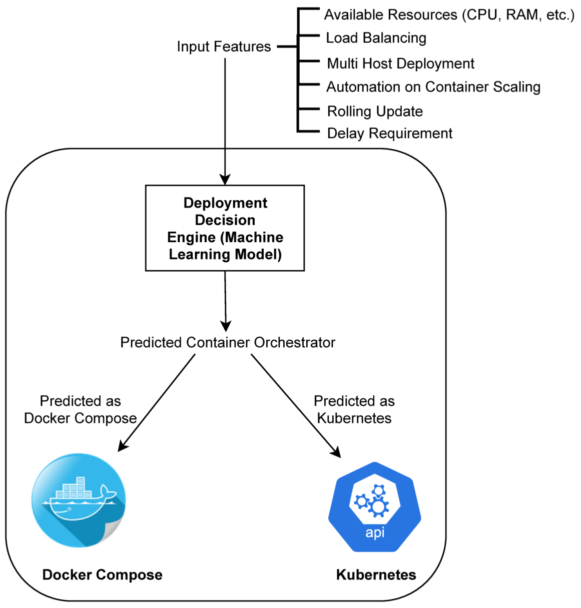
Input Features
Available Resources (CPU, RAM, etc.)
Load Balancing
Multi Host Deployment
Automation on Container Scaling
Rolling Update
Delay Requirement
Deployment Decision Engine (Machine Learning Model)
Predicted Container Orchestrator
Predicted as Docker Compose
Predicted as Kubernetes
api
Docker Compose
Kubernetes

Fig. 1. Proposed System

The deployment decision engine is based on different machine learning models. We have used *K-Nearest Neighbour* algorithm and *Logistic Regression* algorithm to build the machine learning model. These two models are compared using various performance measuring factors like accuracy, precision, recall etc.

i. **K-Nearest Neighbour**: K-Nearest Neighbour (KNN) is a supervised learning technique. KNN algorithm finds the similarity of new data point with the data points available and differentiate data points accordingly. It can be used for both classification and regression task. Here, we are using KNN algorithm for classification purpose. KNN algorithm is a *Lazy Learner Algorithm* as it does not learn from the training data but remembers the training dataset. KNN algorithm performs action on the dataset at the time of classification.
ii. **Logistic Regression**: Logistic Regression is a supervised learning technique that is used for predicting the categorical dependent variable using a set of independent variable. It uses the concept of predictive modeling as regression and used to classify samples. The output value lies within the range of 0 to 1. In logistic regression, we use a *sigmoid* function to map predicted value to probabilities.

III. Evaluation

In this section, we present the analysis done regarding containerized service deployment using container managers.

A. Experiments for Comparison of the Container Managers

1) Device Used for the Experiments: We have performed experiments on a computer having the configurations mentioned in Table I.

TABLE I
DEVICE CONFIGURATION

	Configuration
Model	Intel(R) Core(TM) i5-4210U CPU @ 1.70GHz
CPU Cores	4
Threads Per Core	2
Architecture	x86_64
OS	Ubuntu 18.04.5 LTS
RAM	12 GB

2) Setup Time: Our application consists of two components, a front-end and a back-end. The front-end container contains the docker base image of `nanasess/php7-ext-apache`[6]. The back-end contains the docker base image of `MySQL`[7]. In docker compose based manager, both are deployed using a *docker-compose* file that includes the linking of volumes, authentication to MySQL container, and the connection between the containers. The setup time can be referred to as the time that the container manager takes to finish up the deployment. In setup time, we do not consider the docker base image pulling time as it is already present in the system. Once the base image is pulled by the docker compose based manager, the next time it does not pull the image from docker hub. In Kubernetes, we need to remove the time of pulling the docker base images. We show the setup time of the container managers in Figure 2. We have found that the docker compose based manager is taking less amount of time for deployment than the Kubernetes based deployment for the application.

3) CPU Usage: After successful deployment, we find the CPU utilization by the docker compose based manager and Kubernetes respectively. In docker compose based manager, the CPU utilization is found using *docker stats* and *top* commands. We consider the amount of CPU used by the currently running container and the maximum CPU that can be required by docker service in the docker compose based manager. In Kubernetes, the *kubectl top nodes* command includes the CPU consumption of all the internal services, the external services, and the pods. Using the above approach, we collect output over multiple iterations and plot these in the graph. The CPU consumption of the container managers is shown in Figure 3.

4) Memory Consumption: The memory consumption is shown in Figure 4. In the docker compose based manager, we get memory consumed by each container and docker services using the *docker stats* and *top* commands respectively. Over multiple iterations, it is found that docker services consume a maximum of 1% of memory. In Kubernetes, we take the memory consumed by a node that gives the memory consumption of the pods and various services.

5) Discussions: From the above analysis, we conclude that Kubernetes has lots of overhead for container deployment than docker compose based manager. In Kubernetes, it creates a pod followed by the abstraction layer above it i.e. replication set layer and deployment layer. Also, Kubernetes implements the internal service as well as the external service. While in the docker compose based manager, it is very easy to deploy multiple containers as its setup is faster. The docker compose based manager consumes less amount of CPU and memory for the whole deployment in cloud computing. Though we find from the experiments that taking docker compose based manager is beneficial than Kubernetes for service deployment, Kubernetes is much more feature-rich than docker compose in terms of auto-scaling, self-healing, etc. This analysis is used to generate the data set using simulation.

[6]https://hub.docker.com/r/nanasess/php7-ext-apache
[7]https://hub.docker.com/_/mysql

B. Data Set

From the above subsection III-A, we have seen that docker compose is best preferred for limited Memory and limited CPU. Therefore, we have *available CPU* and *available memory* as the features for preparing the dataset. Apart from that, docker compose is more faster in setting up an application as seen in the subsection III-A. Therefore, *delay requirement* is a feature supporting docker compose. The features supporting kubernetes are *Auto Scalability*, *Multi Host*, *Rolling Update*, *Load Balancing*. Apart from *Auto Scalabilitiy*, all other features can be managed manually using docker compose. On the other hand, kubernetes does that efficiently and makes it less complex. Therefore, it is more likely to be feature supporting kubernetes. The value of a feature signifies the importance of the feature for an application. Highest value indicates more priority of the feature. In the dataset, the feature value is distributed from 0 (lowest priority) to 6 (highest priority). So, we have a total of 5040 data points in our dataset. The ratio of training data and test data is 3:1

C. Results from the Deployment Decision Engine

In our implementation of *deployment decision engine*, we import KNN class from `sklearn`[8] library where we have taken the value of K i.e. the required number of nearest neighbours as 5. Also, we implement the *deployment decision engine* by importing the *LogisticRegression* class from the *sklearn* library.

1) Training: After training the model using both *KNN* algorithm and *Logistic Regression* algorithm, the models are analyzed based on various factors. These factors are confusion matrix, accuracy, precision, recall, and cross-validation mean. These measurements show the model performance. We have shown the measurements for both of the algorithms in the following subsections.

2) Confusion Matrix: Confusion matrix provides statistics about the overall classification accuracy over the test data. Confusion matrix consists of two rows labelled with Kubernetes and Docker Compose. These rows are true labels. The two columns are labelled with Kubernetes and Docker Compose. The columns are predicted labels. To implement

[8]https://scikit-learn.org/stable/

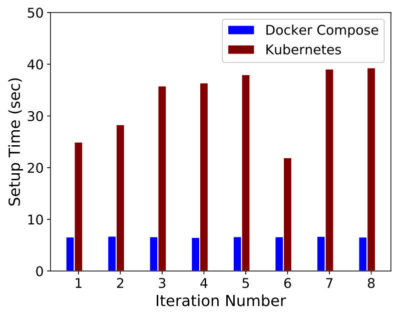

Fig. 2. Setup Time

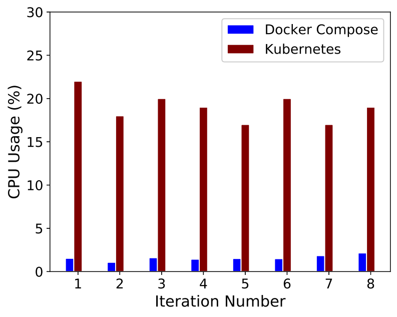

Fig. 3. CPU Usage

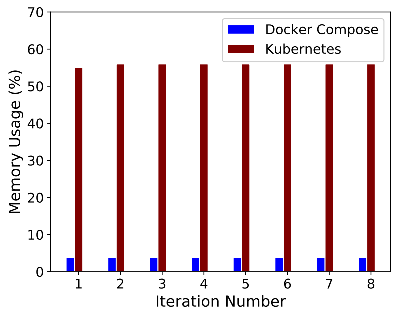

Fig. 4. Memory Usage

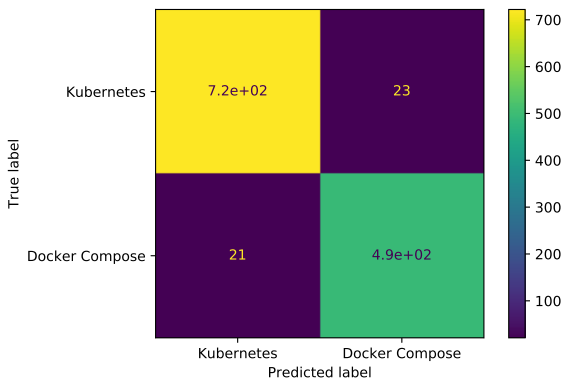

Fig. 5. Confusion Matrix for KNN Model

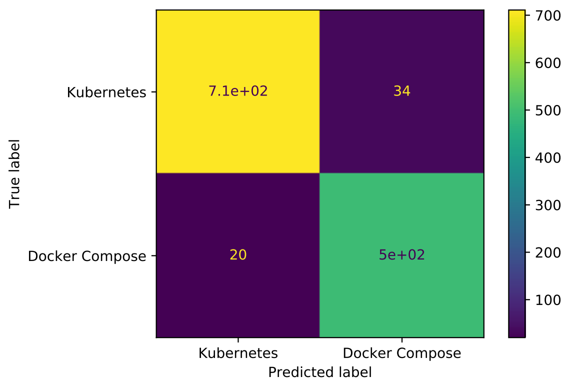

Fig. 6. Confusion Matrix for Logistic Regression Model

confusion matrix, we have imported confusion_matrix function from the *sklearn* library. Confusion Matrix for KNN and Logistic Regression have been shown in Figure 5 and Figure 6 respectively. We have considered Docker Compose as the positive class and Kubernetes as the negative class. In the confusion matrix of KNN model, the number of Kubernetes labels classified as Kubernetes i.e. *True Negative* is 722. Whereas, the number of Kubernetes labels classified as Docker Compose i.e. *False Negative* is 23. The number of Docker Compose labels classified as Docker Compose i.e. *True Positive* is 494. Whereas, the number of Docker Compose labels classified as Kubernetes i.e. *False Positive* is 21. In the confusion matrix of Logistic Regression model, the number of *True Negative* is 711 and the number of *False Negative* is 34. The number of *True Positive* is 495. Whereas, the number of *False Positive* is 20.

TABLE II
MODEL SCORE

	K-Nearest Neighbour	Logistic Regression
Accuracy	0.965	0.957
Precision	0.956	0.936
Recall	0.959	0.961
CV Mean	0.936	0.945

3) Model Score: Model score consists of various model performance measures i.e. accuracy, precision, recall, and cross-validation mean. Accuracy can be defined as the total number correctly predicted classes over the total number of predicted classes.

$$Accuracy = \frac{Number\ of\ correct\ prediction}{Total\ number\ of\ prediction}$$

Accuracy of the KNN model and the Logistic Regression model are 96.5% and 95.7% respectively. Precision score of the KNN model and Logistic Regression model are 0.956 and 0.936 respectively. Recall score for the respective models are 0.959 and 0.961 respectively. Model performance measurement factors are shown in Table II.

4) CV Mean: Cross-Validation method is used to check skills of the model. It tells us how good our model is learning while predicting the new data. It keeps track of *overfitting* and *underfitting* while training our model. K-fold Cross Validation divides the training data into k subsets. CV mean can be calculated as the mean of all the k-subset's cross validation score. Our KNN model has the mean cross-validation score as 0.936. Whereas, the mean cross-validation score is 0.945 for the Logistic Regression model.

5) Precision-Recall Curve: Precision-Recall metric shows the classifier output quality where high area under the curve represent high *recall* and high *precision*. High precision shows low false positive rate i.e. more accurate classifier result. Whereas, high recall shows low false negative rate i.e. it is

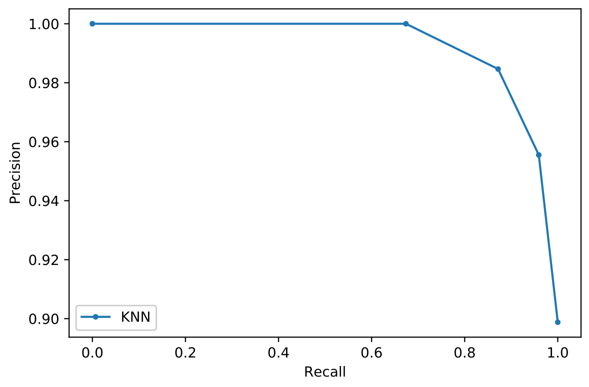

Fig. 7. Precision-Recall Curve for KNN Model

maximizing the positive results. The average precision-recall score for KNN model as well as Logistic Regression model is 0.99. Precision-Recall Curve for the KNN model and the Logistic Regression model are shown in Figure 7 and Figure 8 respectively.

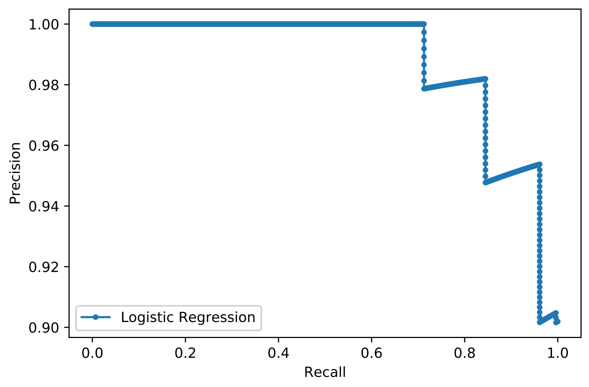

Fig. 8. Precision-Recall Curve for Logistic Regression Model

IV. Conclusion

The services in cloud computing [19] require proper deployment in a server with less amount of time so that the users are responded faster. In order to deploy services in the cloud data center, container based virtualization technology is preferred as it has less resource footprint than virtual machines. The deployment of multi-container applications demands proper management of the containers. The container manager needs to consume less amount of resources (i.e. CPU, RAM etc.) while providing less setup time. Again, based on the demand of serving many clients, the container manager needs to scale up the instances of the deployed containers. In this paper, we have automated the process of container manager selection to provide efficient deployment of an application. We have proposed a machine learning based *deployment decision engine* to automate the selection. The experiments show that the proposed deployment decision engine is effective for choosing an appropriate container manager for the user applications. In the future, we are planning to implement other machine learning models for the proposed deployment decision engine.

References

[1] A. Arunarani, D. Manjula, and V. Sugumaran, "Task scheduling techniques in cloud computing: A literature survey," *Future Generation Computer Systems*, vol. 91, pp. 407–415, 2019.

[2] Z. Li, H. Shen, and C. Miles, "Pagerankvm: A pagerank based algorithm with anti-collocation constraints for virtual machine placement in cloud datacenters," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2018, pp. 634–644.

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

[4] R. Ranjan, I. S. Thakur, G. S. Aujla, N. Kumar, and A. Y. Zomaya, "Energy-efficient workflow scheduling using container-based virtualization in software-defined data centers," *IEEE Transactions on Industrial Informatics*, vol. 16, no. 12, pp. 7646–7657, 2020.

[5] S. B. Nath, S. K. Addya, S. Chakraborty, and S. K. Ghosh, "Container-based service state management in cloud computing," in *2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)*. IEEE, 2021, pp. 487–493.

[6] S. N. Srirama, M. Adhikari, and S. Paul, "Application deployment using containers with auto-scaling for microservices in cloud environment," *Journal of Network and Computer Applications*, p. 102629, 2020.

[7] M. Grambow, L. Meusel, E. Wittern, and D. Bermbach, "Benchmarking microservice performance: a pattern-based approach," in *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, 2020, pp. 232–241.

[8] M. Großmann and C. Klug, "Monitoring container services at the network edge," in *2017 29th International Teletraffic Congress (ITC 29)*, vol. 1. IEEE, 2017, pp. 130–133.

[9] J. Santos, T. Wauters, B. Volckaert, and F. De Turck, "Towards network-aware resource provisioning in kubernetes for fog computing applications," in *2019 IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 2019, pp. 351–359.

[10] D. Balla, C. Simon, and M. Maliosz, "Adaptive scaling of kubernetes pods," in *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 2020, pp. 1–5.

[11] I. M. Al Jawarneh, P. Bellavista, F. Bosi, L. Foschini, G. Martuscelli, R. Montanari, and A. Palopoli, "Container orchestration engines: A thorough functional and performance comparison," in *ICC 2019-2019 IEEE International Conference on Communications (ICC)*. IEEE, 2019, pp. 1–6.

[12] M. Moravcik and M. Kontsek, "Overview of docker container orchestration tools," in *2020 18th International Conference on Emerging eLearning Technologies and Applications (ICETA)*. IEEE, 2020, pp. 475–480.

[13] B. Xu, S. Wu, J. Xiao, H. Jin, Y. Zhang, G. Shi, T. Lin, J. Rao, L. Yi, and J. Jiang, "Sledge: Towards efficient live migration of docker containers," in *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*. IEEE, 2020, pp. 321–328.

[14] H. Yang, S. Liang, J. Ni, H. Li, and X. S. Shen, "Secure and efficient k nn classification for industrial internet of things," *IEEE Internet of Things Journal*, vol. 7, no. 11, pp. 10 945–10 954, 2020.

[15] J. He, J. Wei, K. Chen, Z. Tang, Y. Zhou, and Y. Zhang, "Multitier fog computing with large-scale iot data analytics for smart cities," *IEEE Internet of Things Journal*, vol. 5, no. 2, pp. 677–686, 2017.

[16] T. Shiraishi, M. Noro, R. Kondo, Y. Takano, and N. Oguchi, "Real-time monitoring system for container networks in the era of microservices," in *2020 21st Asia-Pacific Network Operations and Management Symposium (APNOMS)*. IEEE, 2020, pp. 161–166.

[17] S. Wang, Z. Ding, and C. Jiang, "Elastic scheduling for microservice applications in clouds," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 1, pp. 98–115, 2020.

[18] T. L. Duc, R. G. Leiva, P. Casari, and P.-O. Östberg, "Machine learning methods for reliable resource provisioning in edge-cloud computing: A survey," *ACM Computing Surveys (CSUR)*, vol. 52, no. 5, pp. 1–39, 2019.

[19] M. Kumar, S. C. Sharma, A. Goel, and S. P. Singh, "A comprehensive survey for scheduling techniques in cloud computing," *Journal of Network and Computer Applications*, vol. 143, pp. 1–33, 2019.