Automating the Selection of Container Orchestrators for Service Deployment

1st Suryam Arnav Kalra
Computer Science and Engineering
IIT Kharagpur
Kharagpur, India
suryamkalra35@gmail.com

2nd Kunal Singh
Computer Science and Engineering
IIT Kharagpur
Kharagpur, India
ksingh19136@gmail.com

3rd Aryan Agarwal
Computer Science and Engineering
IIT Kharagpur
Kharagpur, India
aryan.cs.kgp@gmail.com

***Abstract*—Cloud Computing has become the new norm and everyone is availing cloud services due to the added benefits it provides. Cloud services have traditionally been deployed as Virtual Machines (VM) in cloud provider servers. VMs are slow as they are fully isolated operating systems running on virtualized hardware. Due to this, there has been a shift towards Container as a Service (CaaS) computing model. Containers are lightweight Operating Systems that run directly on the host server and are faster than VMs. With this, comes the added responsibility of proper resource management of containers as container managers need to optimally use the computing resources along with satisfying the needs of the clients. The container managers need to be rightly chosen and for this we propose an architecture to appropriately select the manager based on application needs and user demands. The architecture that we propose uses a Machine Learning model to classify the right manager for containers. We have used docker containers for our experiments. Our experiments find the different parameters/important features of the application and the suitable ML models for effectively and efficiently choosing between Docker compose based manager and Kubernetes respectively.**

***Index Terms*—Docker, Docker Compose, Kubernetes, Containers, Virtual Machine**

I. Introduction

Cloud computing has become the new normal given broad network access, shared pool of resources, infinite storage all for a minimal pay-per-use costing. From the very beginning of cloud services, a virtual machine based deployment has been carried out using a hypervisor. But it has its own pitfalls due to which there has been a massive shift towards Container based deployment owing to the lightweight and faster creation of container images. This lead to a new service model Container as a Service (CaaS).

First of all, let us understand what is a Virtual Machine (VM). A VM is an abstraction of a physical environment by virtualizing the hardware resources. They are heavy packages that provide complete control of low level hardware resources. As shown in Figure 1, we have an Operating System (OS) installed in the physical machine. Over this OS, a hypervisor which captures and emulates the instructions from different VMs and allows for proper management of VMs is installed. Each VM is a standalone machine having its own Operating System, Network Stack, Memory and Compute resources. We can have multiple VMs running over the same physical hardware which led to its widespread use in Cloud computing.

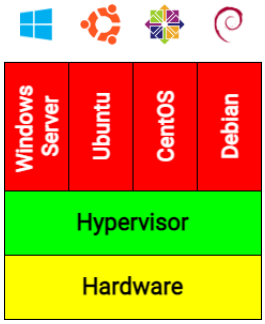

Fig. 1. Virtual Machine Architecture

A container on the other hand virtualizes the operating system installed over the physical machine. They are lightweight packages/Operating Systems that contain the required dependencies to execute the contained software. As shown in Figure 2, we have a Container Engine (Eg. *Docker*) installed over the Operating System. It manages and emulates the different containers which are currently being run.

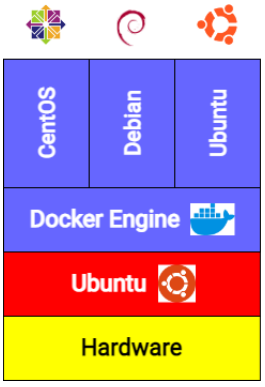

Fig. 2. Container Architecture

Virtual Machines have their own guest Operating System

with individual OS Kernels. They encompass a full stack system. Due to this, they take a long time to setup and use more of memory resources. Whereas, a container shares the Kernel with the Operating System the Container Engine runs on. They encompass only the required high-level software components which allow for their faster deployment and usage of exponentially less memory and compute resources.

A. Motivation

Let us take an example of a service we use everyday such as a web application. From an end-user point of view, features such as response time, latency, availability seem to be the most important whereas looking from the eyes of a deployer, features including scability, realibility, low power consumption make the cut for the most desired ones. Consider a scenario where the web application has to only cater to a limited number of clients the most sought after traits would want the container manager to have optimum resource allocation and less latency while managing the container images. On the other hand, if the application needs to be available to a large number of users throughout the day a container manager able to easily scale the resources to meet the loads while keeping the application available would be desirable. Therefore, there is a need for a decision module which on inputs from the user about the application can effectively and efficiently decide the best orchestrator for the container-based application.

B. Objectives

In this work, we study the effectiveness of different container orchestrators according to application demands. We have used *docker* containers for our experiments. The key contributions of our work are as follows:

- We proposed an architecture for the effective selection of Container Orchestrators namely *Docker Compose* and *Kubernetes* based on user input including the requirements of the application.
- We study different features of the user and application needs such as *CPU usage*, *RAM usage*, *Setup Time* and how they effect the performance of container orchestrators.
- We consider multi-container based applications specifically replica of a web deployment server consisting of a front-end made using *PHP* and back-end made using *MySQL*.
- We implement and test two Machine Learning models namely *KNN* and *Logistic Regression* for selection of the container manager.

II. Related Work

A lack of consensus exists on choosing which container service to use for any particular application. At this point, there is not much work done on the automatic selection of container services. However, some of the research [1], [2], [3] has identified the various factors on which container application orchestration depends. Some of these are

- All the segments of the code are to be defined and microservices are needed to be listed, for example, if a mobile application has a front-end (React, Angular) and a backend in MySQL. So for both services, we require different containers.
- A proper connection between the containers needs to be established for consistent operation.
- In case of failures, data backup comes as an essential thing. Therefore, a backup of the container's data should also be created in volumes.

An emerging trend in the field of cloud computing research is observed between two topics, one is resource allocation in container based cloud platforms and the other is dividing the application into different microservices and the interoperability between these microservices. In recent years a plethora of studies have been conducted on how to allocate resources in container based cloud platforms in order to achieve maximum efficiency, different algorithms have come up as a result of these studies. For example, researchers have developed scheduling algorithms [4] for online as well as offline containers and also for different kinds of workloads, be it long-lasting training models of different machine learning and deep learning algorithms or some shorter microservice based applications. The above discussed algorithms are based on assumption of partial value for partial execution, specially for the long-lasting application. There have been studies on specific container managers, such as the study by researchers [5] in which they analysed a container-based cluster management tool of Docker and proposed algorithms for better load balancing and resource management.

III. Proposed Architecture

As the above discussion suggests that there is a need for an architecture which can be used to choose container manager rightly based on the application needs. Therefore, we propose an architecture consisting of 4 layers. The first and topmost layer of the architecture takes input from the user as what are the requirements of the application. The features like available resources (CPU, RAM, etc.), memory, load balancing, multi host deployment, automation on container scaling, rolling update and delay requirement will be taken as an input and will serve as the features for our decision model. At the second layer we have our decision model which is a machine learning algorithm, since the prediction of container manager should be fast enough machine learning algorithm are the best fit here due to their ability of predicting outcome. We have used two machine learning algorithms namely K-Nearest neighbour and Logistic Regression for this purpose, details of these are provided in Section IV. In the third layer, we get the output out of our decision engine whether to choose Docker compose based container manager or Kubernetes for the specific application of which the features are provided. The fourth and the last layer is deployment of the application on the container orchestrator that is predicted. The architecture is show in Figure 3.

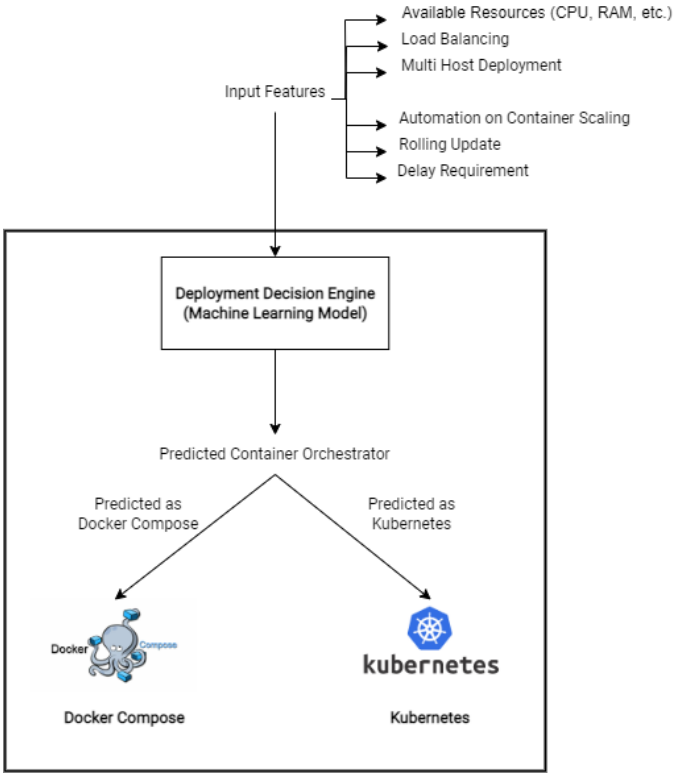

Fig. 3. Proposed System

IV. Implementation

A. Approach

1) Device used for the experiments: Table I contains the configuration of the device used for our experiments.

TABLE I
Device Configuration

	Configuration
Model	Intel® Core™ i5-8250U CPU @ 1.60GHz
CPU Cores	4
Threads per Core	2
Architecture	x86_64
OS	Ubuntu 22.04.1 LTS
RAM	8 GB

2) Application background: We made a full stack web application for our experiments. PHP with docker base image at *nanasess/php7-ext-apache*[1] and MySQL with docker base image at *MySQL*[2] were used in the frontend and backend of our application respectively. Both the container managers, docker-compose and kubernetes, used a configuration file which had the details of environment variables, ports and volume linking. We used *docker-compose up* and *kubectl apply* commands for the deployments for our containers/pods. After the successful deployment and measurement of setup time, CPU and memory usage we deleted them using the *docker-compose down* and *kubectl delete* commands. We repeated this process for 10 iterations and plotted the graphs.

3) Setup Time: We define setup time as the time taken by a container manager to finish the deployment of the conatiners/pods including the linking of volumes and database authentication. The time taken to pull the images from internet is discarded, as once the image is pulled it remains locally on the computer and does not get pulled subsequent times. The setup time is measure with the linux *time* command for docker compose and by checking the logs of the pod using the *kubectl get pod* command in kubernetes. The setup time of the container managers is shown in Figure 4. We found that the setup time for the docker-compose based manager is very less as compared to the Kubernetes.

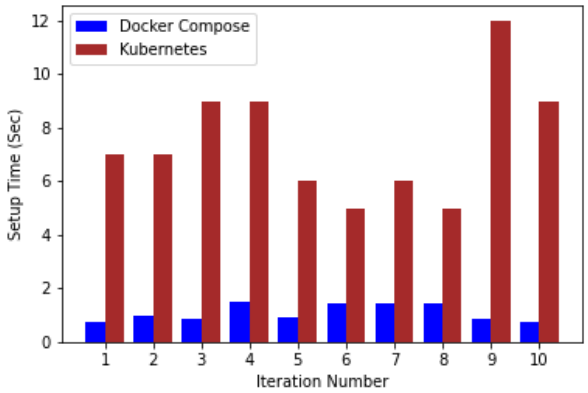

Fig. 4. Setup Time

4) CPU Usage: The CPU usage is measure with the *docker stats* command for docker compose and with *kubectl top nodes* command in kubernetes. The cpu usage of the container managers is shown in Figure 5. We found that the cpu usage for the docker-compose based manager is very less ($\sim$0.5% in all iterations) as compared to the Kubernetes. Since our application had two services (database and php) we took the average of the two cpu usages from the containers in docker compose. Similarly, we took the cpu usage of only the master node in the kubernetes cluster.

5) Memory Consumption: The memory consumption is measure with the *docker stats* command for docker compose and with *kubectl top nodes* command in kubernetes. The memory consumption of the container managers is shown in Figure 6. We found that the memory consumption for the docker-compose based manager is very less as compared to the Kubernetes. Since our application had two services (database and php) we took the average of the two memory consumption from the containers in docker compose. Similarly, we took the memory consumption of only the master node in the kubernetes cluster.

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

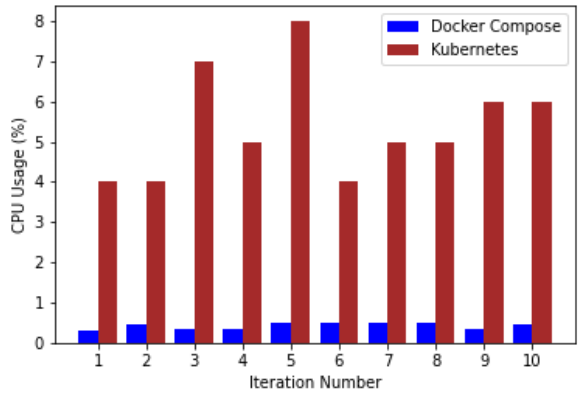

Fig. 5. CPU Usage

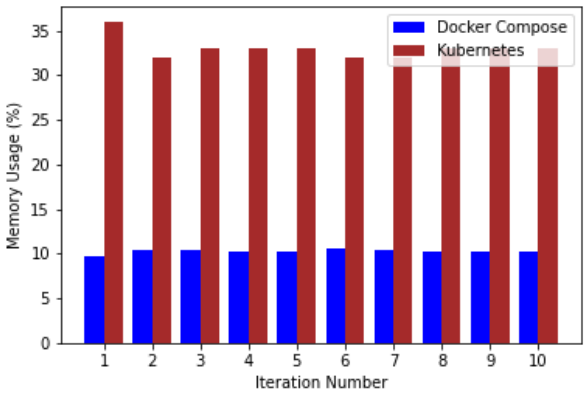

Fig. 6. Memory Usage

B. Details of existing work

From what we know, this effort is the first attempt in developing a system architecture for automating the selection process of container manager services between docker compose and Kubernetes. Earlier attempts in the field of cloud computing research focused on different other aspects, such as algorithms to make the creation, deployment and scaling of containers efficient. The primary challenge in automating the selection process of container managers is the evaluation of these container services on various features which the users are concerned about while choosing the container managers. The decision model should have enough data as close as possible to the real-world use cases in order to predict excellently. The data that can be used to feed the decision engine could be of two types, one is collecting the data of which container manager do users generally choose or prefer when they have a certain set of requirements and the second one is generating data on the basis of comparison between the container managers and this data will show what users should choose if they have a certain set of requirements. We have followed the second approach, and we have compared Setup Time, CPU usage, Memory consumption, etc between the container managers for dataset generation, details of which are provided in the next section.

V. Experimental Results

A. Dataset Description

On the grounds of the results obtained in Section IV, we have considered seven features into account in our dataset, namely Limited CPU (for CPU usage), Limited Mem (for memory usage), Auto Scalability, Multi Host, Rolling Update, Load Balancing and Delay Requirement. The features limited CPU and limited Memory are in the dataset due to the fact shown in subsection IV-A that docker compose performs better on these features. The feature Delay Requirement is also there because decker compose takes less time to set up an application as seen in subsection IV-A. Kubernetes handles efficiently the features, Rolling Update, Multi Host, Auto scalability and load balancing. In Docker also, we can manage the these features other than Auto Scalability manually, but if we want to do in more efficient and less complex manner Kubernetes should be preferred. In the dataset the value for each feature signifies its importance and the values are distributed from 0 to 6 where 0 implies lowest priority and 6 implies highest priority. Other than feature, there is an output label for each data point which can have a value either 0 which means docker compose should be preferred or 1 which means Kubernetes should be preferred. Since for each feature there can be 7 values from 0 to 6 therefore, 7 factorial arrangements are possible and that is why we have 5040 data points in the dataset. The ratio of train and test split is 3:1. A sample of our dataset is given in Table II.

B. Results from the Machine Learning models

1) Training: We trained the K-nearest neighbour (KNN) and Logistic Regression models on the dataset that we created according to the subsection V-A. The value of K for KNN after hyperparameter tuning was set to be 5. The results and scores of both the models are compared accross various heuristics such as confusion matrix, test set accuracy, precision-recall curve and cross-validation mean.

2) Confusion Matrix: The confusion matrix is used to study the performance of the algorithms. We have created the confusion matrices for both KNN and logistic regression using the *sklearn* library functions. The matrix has dimensions 2×2 where the rows depict the true labels with the first row being denoted by Kubernetes and second row by Docker Compose. Similarly, the two columns depict the predicted labels with the first column being Kubernetes and second column being Docker Compose. The Kubernetes class has been considered the negative class whereas the Docker Compose class is assigned the positive class. The $(1,1)$ entry signifies the *True Negative* meaning our model predicted Kubernetes and its correct. The $(1,2)$ entry signifies *False Positive* meaning our

TABLE II
DATASET DETAILS

Limited CPU	Limited Memory	Auto Scalibility	Multi-Host	Rolling Update	Load Balancing	Delay Requirement	Output
6	4	5	0	2	1	3	0
3	5	2	0	6	1	4	1
2	0	1	5	6	3	4	0
1	5	2	0	3	6	4	1

TABLE III
MODEL RESULTS

	KNN	Logistic Regression
Accuracy	0.965	0.957
Precision	0.956	0.936
Recall	0.959	0.961
CV Mean	0.936	0.945

model predicted Docker Compose but its incorrect. The $(2, 1)$ entry signifies *False Negative* meaning our model predicted Kubernetes and its incorrect. The $(2, 2)$ entry signifies *True Positive* meaning our model predicted Docker Compose and its correct. The confusion matrix for KNN is given in Figure 7 and for Logistic Regression is given in Figure 8.

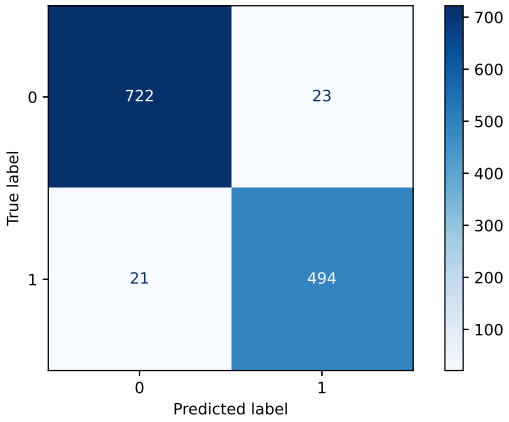

Fig. 7. Confusion Matrix for KNN Model

3) Model Score: We have used four scoring heuristics to test our models namely accuracy, precision, recall and cross-validation mean.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

The score values for both the models are calculated using the *sklearn* library and have been summarized in Table III.

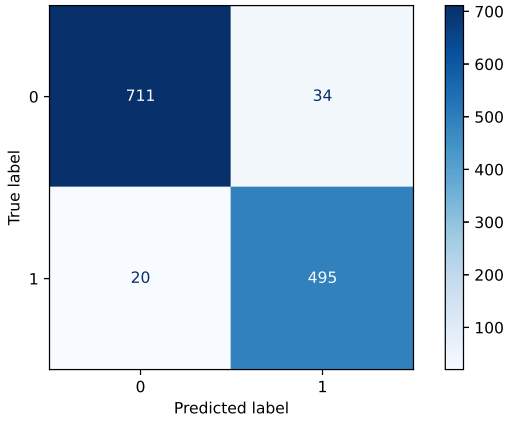

Fig. 8. Confusion Matrix for Logistic Regression Model

4) CV mean: A Machine Learning model is good if it can predict the unforeseen data points with high accuracy. This model score helps us to keep a check on whether the model is underfitting or overfitting the data. We use K-fold cross validation from the *sklearn* library and the CV mean is calculated by taking the mean of the score of the K hold-out sets. We can see in Table III the CV mean score for KNN is 0.936 and for Logistic Regression it is 0.945 respectively.

5) Precision-Recall Curve: Precision-Recall curve is used to show the tradeoff between precision and recall for different thresholds. Mainly, the area under the precision-recall curve is of higher importance as higher the area means the model is performing better. A high value of precision implies that out of the positive values predicted by our model most are true positives. On the other side, a high value of recall implies that our model is able to predict most of the positive class labels correctly. The precision-recall curve for KNN is given in Figure 9 and for Logistic Regression is given in Figure 10.

VI. DISCUSSION

In this paper, we proposed a new system architecture to automate the process of selecting cloud manager services based on application demands. We carried out experiments for comparison of the container managers in order to identify features to be fed to the deployment decision model and the results shows:

- On testing setup time, CPU usage and memory usage with an application consisting of two components, a front-end

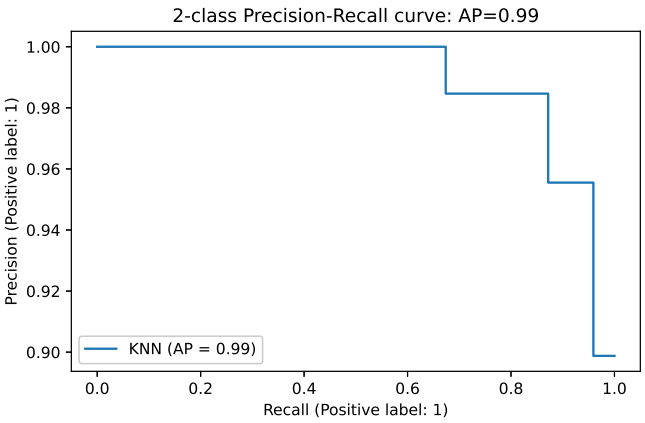

Fig. 9. Precision-Recall Curve for KNN Model

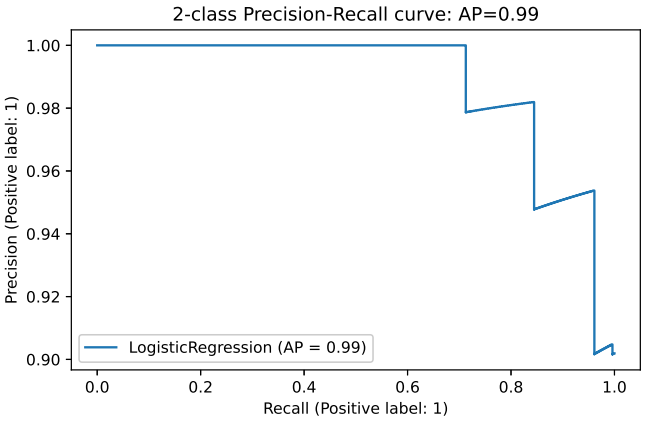

Fig. 10. Precision-Recall Curve for Logistic Regression Model

and a back-end, docker compose was better with CPU and memory utilization and took less amount of time for deployment than Kubernetes.

- Experiments showed that Kubernetes has lots of overheads and is not efficient with CPU and memory utilization. But Kubernetes is far more feature rich than docker compose when it comes to auto-scaling, rolling update, load balancing, etc. That is why all these features were considered in our dataset.
- We experimented with two machine learning models namely K-nearest neighbours and Logistic Regression as both of them are state-of-the art for classification tasks.
- Based on accuracy and precision, KNN is slightly better than Logistic Regression whereas the opposite trend is noticed for recall and CV mean. The difference is not too significant and thus one can use any of the two ML models in the deployment decision engine for correctly predicting the container orchestrator to use based on the application needs.

VII. Conclusion

Traditionally cloud services has been deployed using Virtual Machines but they have a large footprint in memory and time both as they are stand alone systems with their own virtualized hardware. In the present day scenario, the trend has moved towards providing cloud services through containers as they only virtualize the operating system and have lower memory footprint and are faster in deployment. This brings us to the next issue of selecting the best container orchestrator with respect to varying application demands. In this paper, we propose an architecture by creating a machine learning based decision deployment engine. This engine takes inputs from the user about the application demands and outputs the orchestrator between Kubernetes and Docker Compose for that specific application. Our machine learning models namely KNN and Logistic Regression have been able to achieve a high accuracy of 95% on the dataset with varying priorities of the application demands.

VIII. Future Work

The future work in this domain entails a lot of possibilities. One of the things that we plan to do is implement new Machine Learning models like Artificial Neural Networks (ANN) and Support Vector Machines (SVM) to aim for better accuracies than the already implemented models. In addition to it, we are also planning to identify new features such as Container Self-healing, Realibility and Easy Linking of containers in our dataset for better selection of container orchestrators with respect to new application demands.

References

[1] P. Chaurasia, S. B. Nath, S. K. Addya, and S. K. Ghosh, "Automating the selection of container orchestrators for service deployment," *IEEE*, 2022.
[2] T. Shiraishi, M. Noro, R. Kondo, Y. Takano, and N. Oguchi, "Real-time monitoring system for container networks in the era of microservices," *IEEE*, 2020.
[3] S. Wang, Z. Ding, and C. Jiang, "Elastic scheduling for microservice applications in clouds," *IEEE*, 2020.
[4] R. Zhou, Z. Li, and C. Wu, "Scheduling frameworks for cloud container services," *IEEE/ACM*, 2018.
[5] L. Li, J. Chen, and W. Yan, "A particle swarm optimization-based container scheduling algorithm of docker platform." *ACM*, 2018.
[6] D. Elliott, C. Otero, M. Ridley, and X. Merino, "A cloud-agnostic container orchestrator for improving interoperability," *IEEE*, 2018.
[7] P. Chaurasia, S. B. Nath, S. K. Addya, and S. K. Ghosh, "Automating the selection of container orchestrators for service deployment," *IEEE*, 2022.
[8] B. Tan, H. Ma, and Y. Mei, "A nsga ii based approach for multi-objective micro-service allocation in container-based clouds," *IEEE*, 2020.
[9] H. Zhang, H. Ma, G. Fu, X. Yang, Z. Jiang, and Y. Gao, "Container based video surveillance cloud service with fine-grained resource provisioning," *IEEE*, 2016.
[10] G. Ambrosino, G. B. Fioccola, R. Canonico, and G. Ventre, "Container mapping and its impact on performance in containerized cloud environments," *IEEE*, 2020.
[11] Y. Lei and P. S. Yu, "Container scheduling in blockchain-based cloud service platform," *IEEE*, 2020.
[12] S. B. Nath, S. K. Addya, S. Chakraborty, and S. K. Ghosh, "Container-based service state management in cloud computing," *IEEE*, 2021.
[13] O. Katz, D. Rawitz, and D. Raz, "Containers resource allocation in dynamic cloud environments," *IEEE*, 2021.

[14] Z. Nikdel, B. Gao, , and S. W. Neville, "Dockersim: Full-stack simulation of container-based software-as-a-service (saas) cloud deployments and environments," *IEEE*, 2017.
[15] J.-D. Jhan, Y.-C. Lai, Y.-L. Chen, and F.-H. Kuo, "Enhanced quality of service measurement mechanism of container-based cloud network architecture," *IEEE*, 2021.
[16] S. B. Nath, S. K. Addya, S. Chakraborty, and S. K. Ghosh, "Green containerized service consolidation in cloud," *IEEE*, 2020.
[17] S. B. Nath, S. Chattopadhyay, R. Karmakar, S. K. Addya, S. Chakraborty, and S. K. Ghosh, "Ptc: Pick-test-choose to place containerized micro-services in iot," *IEEE*, 2019.
[18] D. K. Kim and H.-G. Roh, "Scheduling containers rather than functions for function-as-a-service," *IEEE*, 2021.
[19] B. Xu, S. Wu, J. Xiao, H. Jin, Y. Zhang, G. Shi, T. Lin, J. Rao, and J. J. Li Yi, "Sledge : Towards efficient live migration of docker containers," *IEEE*, 2020.
[20] D. Zhao, N. Mandagere, G. Alatorre, M. Mohamed, and H. Ludwig, "Toward locality-aware scheduling for containerized cloud services," *IEEE*, 2015.