A Fairness-aware Load Balancing Strategy in Multi-Tenant Clouds

Yu-Teng Chen[1] and Kuan-Chou Lai[1]

[1] National Taichung University of Education, Taichung, Taiwan. R.O.C.
kclai@mail.ntcu.edu.tw

Abstract. Load balancing is an important issue in multi-tenant clouds to ensure the load balanced on computing resources belonged to different tenants. A containerized multi-tenant environment could be managed by Kubernetes using different scheduling policies. For improving scheduling performance of Kubernetes, Apache Yunikorn project provides the fine-grain control by hierarchical resource queues to enhance the resource utilization. However, the fairness-aware load balance among tenants is missed in Yunikorn, which may result in poor resource utilization in some tenants. This paper proposes a fairness-aware load balance policy for multi-tenant environments to keep the balance of resources allocated in different tenants, and also the balance of the utilizations of different resources in a computing node. Experimental results show the superiority of the proposed policy.

Keywords: Fairness, Load Balance, Multi-tenants, Yunikorn, Resource Allocation.

1 Introduction

Adopting container technique to provide services is popular in cloud computing environments. In such environments, tenants could easily deploy containerized applications on these clouds. In general, Kubernetes [2, 5] is a high-availability distributed orchestration platform to maintain containers. Although Kubernetes allocates resources to tenants, the fairness issue of the resource allocation policy in Kubernetes still could be improved in the multi-tenant environment.

Apache Yunikorn [1] provides a fine-grained control over resources among tenants, which is missed in the Kubernetes. Apache Yunikorn adopts the hierarchical resource queues and the access control list (ACL) to manage resources among different tenants. The scheduling decision in Apache Yunikorn considers the specific order of applications and nodes; therefore, Apache Yunikorn has better scheduling performance because its scheduling cycle is shorter than that of Kubernetes.

However, neither Yunikorn nor Kubernetes considers both the load balance among nodes and the fairness among tenants. The fairness among tenants [4, 7] is an important issue to avoid resource conflict in the scheduling policy. In the meantime, neither Yunikorn nor Kubernetes considers the load balance among clusters. For example, the

NodeResourceFit procedure in the Kubernetes scheduler has three policies: most, least and balance. The most and least policies calculate the score based on average resource utilization and make decision. However, average resource utilization couldn't indicate the utilization gap among heterogeneous resources resulting in the resource waste. The utilization gap occurs when certain resources are depleted when other resources are plentiful in one computing node. Although the balance policy may reduce the utilization gap among heterogeneous resources in a node, but it doesn't consider the load balance among nodes even among clusters. The similar problem is also found in Apache Yunikorn. So, the schedulers in Apache Yunikorn and Kubernetes couldn't provide an efficient load balance approach in containerized clusters.

In order to improve the fairness and load-balance in multi-tenant environments, this work proposes a fairness-aware load balancing (FALB) strategy to minimize the difference of quantities of heterogenous resources among tenants, the utilization gap [3] of heterogeneous resources in a node and the deviation between heterogeneous resource utilizations among nodes.

In the rest of this paper, the fairness problem and the load balance issue are described first; and the pseudo code of the FALB is introduced. Experimental results are shown and analyzed finally.

2 Related works

Apache Yunikorn [1] adopts hierarchical queues to the fine-grained control and its scheduling performance is better than the Kubernetes one. However, current Yunikorn doesn't provide fairness among tenants. For evaluating fairness, Wang et al. [7] proposed the global dominant resource. But the indicator doesn't consider the elapsed time corresponding to this resource. Another previous work [6] keeps the fairness shared among tenants by computing resource quota. However, this previous work doesn't propose an indicator to solve the load balance. Pfreundschuh et al. [8] considers the profiling execution time of applications via neural network. With the profiling with neural network, this work indicates that the execution time of applications is predictable. The proposed DDRF approach in this work includes the execution time of the applications. To enhance the scheduling performance, the Apache Yunikorn project is proposed, and its scheduling strategies are simple. Carrión et al. [2] and Hilman et al. [4] describe the different scheduling objectives and they list indicators of each works such as CPU, memory, GPU. Chung et al. [3] proposed a method to minimize the resource waste. But it doesn't improve the load balance among nodes. Menouer et al. [5] adopted the Technique for Order of Preference by Similarity to Ideal Solution (TOPSIS) to achieve objectives including CPU utilization, memory utilization and the number of running container. However, it didn't reduce resource waste and the load-balance among nodes. Ramasamy et al. [6] proposed a priority queue scheduling algorithm and fair share strategies. Tenant's application is in high priority when the fair share is lower than the fair share quota. The algorithm focused on maintaining fair share among tenants but the resource utilizations did not be considered.

3 Fairness-aware Load Balancing Strategy

This section introduces the proposed fairness-aware load balancing strategy mechanism. This work adopts a profiling system [8] to capture the application execution time for supporting the scheduling decision making. In this work, containerized applications have different execution time by using different parameters to estimate the expected execution time.

Apache Yunikorn is a cloud-native, efficient, and cost-saving resource scheduling system. It could handle the resource scheduling for running big data and machine learning applications on the Kubernetes platform. Apache Yunikorn adopts the hierarchical resource queue structure and the access control list (ACL) to provide the fine-grained control, so administrators could build customized hierarchical queues to filter tenants and manage resources. Apache Yunikorn manages the resource accessed by tenants according to ACLs. When a new application is submitted, Apache Yunikorn accepts the application when the ACL is matched the permission.

The system components of Yunikorn are as follows:

- Scheduler interface

The scheduler interface defines the *grpc* protocol between the scheduler core and the Kubernetes shim. Common constraints for Yunikorn are also defined here.

- Scheduler core

The scheduler core encapsulates the whole scheduling algorithms, such as application sorting, node sorting and queue sorting. The scheduler core is responsible for making the scheduling decision according to the container allocation requests. There are three sorting policies for applications: FIFO, fair and stateAware. The policies of node-sorting include fair and bin packing approaches.

- Scheduler shim

Kubernetes shim is responsible for communicating with Kubernetes. Kubernetes shim watches events in Kubernetes clusters and translates Kubernetes events into corresponding information. Requests for resource allocation and the status of Kubernetes objects are translated based on the definitions of *grpc* protocol in the scheduler interface; and then the information is transmitted to the scheduler core.

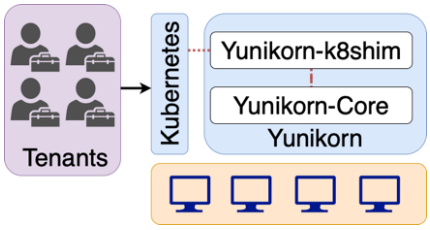

Fig. 1. System Architecture

For evaluating fairness among tenants, this work adopts the resource requirement dominant factor (RRDF) to present the dominant factor of the resource requirement.

Every tenant's RRDF is the sum of the product of resource requirements in each application request within the same tenant. The proposed approach keeps the balance of every tenant's RRDF in order to avoid that some tenants occupy excessive resources in a long time. Assume that there is a tenant set U = {tenant$_i$ ϵ the cluster} and $app(CPU, Mem, Exe\ time)$ represents the request of resource requirement of an application, where r_{CPU} is CPU request, r_{mem} is the memory request, and r_{exe} is the execution time. The variable, $scheduled_{app}$ is 1 when the application is scheduled. Eq. (1) sums up the production of each handled request to compute certain tenant's RRDF. The FALB uses Eq. (2) to find out the tenant with the least RRDF and try to schedule the tenant's application to increase the tenant's RRDF.

$RRDF_{tenant} = \sum_{app \in tenant}(scheduled * \prod_{r \in app} r)$, *where scheduled*=1 if *app is scheduled*; otherwise, it is 0. (1)

$$tenant_i = ARG\ Min_{tenant \epsilon U}(RRDF_{tenant})) \quad (2)$$

For example, a tenant submits three applications and they are scheduled. If each application request 1 CPU and 1KB memory, the RRDF of the tenant is 3*10^6 (3*1000(vcore)* 1000(bytes)). Load-balance minimizes the utilization gap in a node by minimizing the largest difference between any two resource requirements, as shown in Eq. (3) and minimizes the resource deviation among nodes, as shown in Eq. (4) Resource waste happens when there is a large utilization gaps and some resources are exhausted. For measuring resource waste in a node, Eq. (3) calculates the max utilization gap in a node. In the meanwhile, the FALB utilizes the deviation to measure inter-node load balance when calculating the utilization gap in a node. For improving the load balance of nodes within a cluster, Eq. (4) presents the max deviation of resource utilizations of heterogeneous resources in a cluster. FALB considers the utilization gap and the deviation to reduce the resource waste within a node, and improves load-balance among nodes. For example, there is a cluster that CPU deviation is higher than memory one. The proposed FALB mechanism would try to reduce the CPU deviation and to avoid increasing the utilization gap in a node. Eq. (5) finds the mean utilization of the node. This work finds the least mean utilization of a node, the minimal utilization gap in a node and the deviation of resource utilizations when there is a new request from applications, as shown in Eq. (6), (7) and (8). However, there would be a trade-off between Eq. (7) and (8) in some situations. This work adopts the TOPSIS (Technique for Order of Preference by Similarity to Ideal Solution) which is a multi-criteria decision analysis method to choose the best node after calculating the distance from current choice to best choice, as shown in Eq. (9).

•

$UG_i = |Max_{r \in node_i}(u_{r1}) - Min_{r \in node_i}(u_{r2})|$,where take two resource types as example, but could be extended to multiple resource types. (3)

$\sigma_r = Max_{\ r \in nodes}(\sigma_r)$,*where* σ *is standard deviation of utilization* of nodes (4)

$$MU_i = Average(\sum_{r \epsilon node_i} \frac{utilizated\ quantity_r}{Capcity_r}) \quad (5)$$

$$MU_{min} = Min_{i \in nodes}(MU_i) \quad (6)$$

$$UG_{min} = Min_{i \epsilon nodes}(UG_i), \text{where UG is the utilization gap in } node\ i \quad (7)$$

$$\sigma_{min} = Min_{i \in nodes}(\sigma_{r,i}) \quad (8)$$

$$node_{best} = ARG\ Max_{i \in nodes}(TOPSIS(MU, UG, \sigma)) \quad (9)$$

The following is an example to illustrate above equations. For example, there are two nodes (one has 1 CPU,1GB memory, one has 2 CPU, 1GB memory) in the cluster. The both initial mean resource utilizations are 0. When a new application request (0.5 CPU, 0.2 GB memory) is submitted, assigning the application to the specific node makes utilization gaps and standard deviations in the FALB. Utilization gaps of nodes would be 30% (i.e., (0.5/1 – 0.2/1) *100%) and 10% (i.e., (0.5/2-0.2/1) *100%) separately. Their standard deviations are 25 and 10. In the FALB approach, Algorithm 1 finds the application from the tenant with the minimal *RRDF*. For assigning the application to a node, Algorithm 2 chooses a best node based on the trade-off between utilization gaps, standard deviation, and the mean node's utilization. Finally, FALB assigns the tenant's application according to the starting time and node ID by TOPSIS.

Algorithm 1: Fairness in FALB

Inputs:
- *apps, the applications in the cluster.*
- *tenants, who submit applications in cluster*

Outputs:
- *Application ID, which is from tenant with the minimal RRDF*

1:Initialize every $RRDF_{tenant}$ *with 0*
2:For tenant in U:
3: For app in apps:
4: if app belongs to tenant and app is scheduled:
5: $RRDF_{tenant} += \prod_{r \in app} r$
6:tenantID := $ARG\ MIN_{tenant \in U}(RRDF_{tenant})$
7:Initialize heaps H, which order is increasing order of submitted timestamp of applications.
8:Put tenant's unscheduled apps into H separately.
9:return the top of H

Algorithm 2: Load-balance in FALB
Inputs: - *nodes, nodes in cluster allow to run the application* - *req, request of the submitted application* *Outputs:* - *node ID, the node to run the submitted application* *1:nodes:= find nodes allowing request to run* *2:for node ϵ nodes:* *3: UG_{node} := Eq. (3) computes the utilization gap if the app runs on the node* *4: MU_{node} := Eq. (5) calculates mean utilizations of the node* *5: σ_{node} := Eq. (4) finds the standard deviation if the app runs on the node* *6: Put $UG_{node}, MU_{node}, \sigma_{node}$ to $UG_{nodes}, MU_{nodes}, \sigma_{nodes}$* *7: $\grave{MU}_{nodes}, \grave{UG}_{nodes}, \grave{\sigma}_{nodes}$:=normalize $MU_{nodes}, UG_{nodes}, \sigma_{nodes}$ and then divide them with 3 separately.* *10: # Based on $\grave{MU}_{nodes}, \grave{UG}_{nodes}, \grave{\sigma}_{nodes}$* *11:$A^{+}_{MU}, A^{+}_{UG}, A^{+}_{\sigma}$:=finding the minimal values Eq. (6)(7)(8) # best point* *12:$A^{-}_{MU}, A^{-}_{UG}, A^{-}_{\sigma}$:=finding the max values # worst point* *13:for n in nodes:* *14: the point $(\grave{wait}_n, \grave{UG}_n, \grave{\sigma}_n)$ when choosing node n* *15: append Euclidean distance between the point and best point to SM^{+}* *16: append Euclidean distance between the point and worst point to SM^{-}* *17: append $SM^{-}_n/(SM^{-}_n + SM^{+}_n)$ to RC* *18:return the node with the minimal RC value*

4 Experiment Results

This section introduces the experiment to show the performance improvement of FALB. **Table 1** shows the machine specification in this experiment. The whole system consists of ten workstations and the node10 is the master node for Apache Yunikorn. Yunikorn on the master node assigns tenants' containerized applications to the slave workstations.

Table 1. machine specification

Node number	Machine Specification		
	Product Name	CPU	Memory
node1	IBM System x	16	36G
node2	IBM System x	16	42G
node3	IBM System x	16	32G
node4	BladeCenter HS23	8	32G
node5	BladeCenter HS23	8	32G
node6	ProLiant DL360 G6	16	36G
node7	ProLiant DL360 G6	16	30G
node8	ProLiant DL360 G6	16	36G
node9	ProLiant DL360p Gen8	24	32G
node10	Pro E500 G6_WS720T	16	40G

Table 2 shows the software information. Apache Yunikorn is responsible to schedule the pod which is basic scheduling unit in Kubernetes. Pods sharing the same application ID belong to the same application. A pod contains multiple containers and Docker is a well-known container runtime to provide operations of containers. Kubernetes is a container management platform, and a pod is a basic scheduling unit. Helm chart is the tool to manage configuration and developers could deploy applications to Kubernetes by helm.

Table 2. software version

Software	Version
Apache Yunikorn	1.1.0
Docker	20.10.17
Kubernetes	1.21.0-00
Ubuntu	18.04
Helm	3.9.0

Table 3 indicates the resource requirement of four tenants who submits 50 applications separately. Each application is encapsulated in a pod. User1 and user2 prefer CPU. The others prefer memory.

Table 3. tenants' application information

tenants	CPU	memory(G)	execution(s)
user1	2	8	50
user2	1	4	200
user3	8	2	50
user4	4	1	200

There are two application scenarios: the stream scenario submits applications of tenants sequentially after deploying Yunikorn; In the batch scenario, Yunikorn deploys all application when all of them are submitted.

Table 4 indicates what objective strategies adopt. The mean utilization is the main scheduling objective in the original Yunikorn. The Fairness-aware Yunikorn (FA-YK), FALB-2 and FALB-3 implement the Algorithm 1 to maintain the *RRDF*. Comparing to the original Yunikorn and FA-YK, FALB series use TOPSIS to find the best node. The mean utilization and the utilization gap are the objectives in FALB-2 and FALB-3. Additionally, FALB-3 objectives include the resource deviation.

Table 4. active objectives among strategies

Strategies	DDRF	Mean utilization	Utilization gap	Deviation
Original Yunikorn		x		
FA-YK	x	x		
FALB-2	x	x	x	
FALB-3	x	x	x	x

Fig. 2 indicates the total execution time of FALB series is better than the execution times of Yunikorn and the FA-YK. In the both scenario, two phenomena describe the benefits from *DDRF* and the effect of objectives. Maintaining tenants' *DDRF* reduces the specific resource exhaustion when a lot of same kind applications are submitted. Objectives in FALB series make the total execution time shorter than the execution time of Yunikorn and FA-YK.

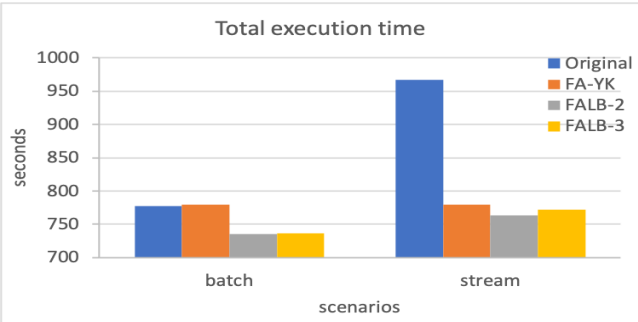

Fig. 2. Total execution time with different strategies

In the stream scenario, the FALB provides better fairness among tenants. Comparing to the FALB in **Fig. 4**, original Yunikorn in the **Fig. 3** did not try to keep every tenant's *RRDF* close.

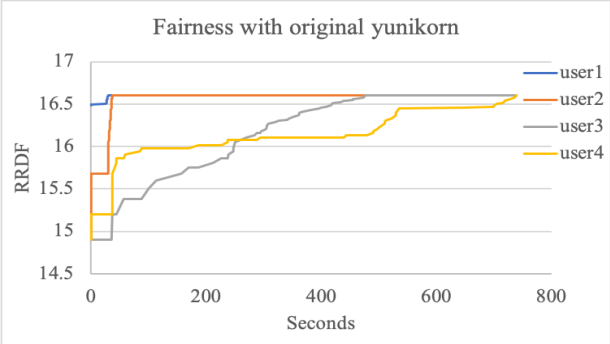

Fig. 3. fairness in stream scenario (original)

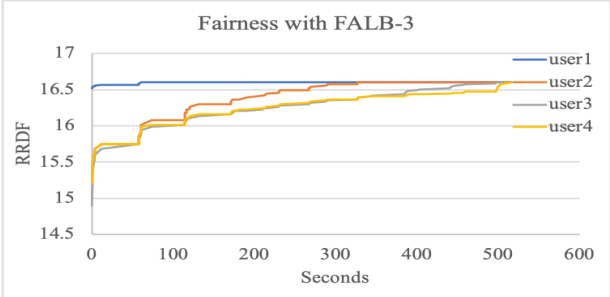

Fig. 4. fairness in stream scenario (adopting FALB)

Fig. 5 indicates the max resource deviations among original Yunikorn and FALB are between 5 and 28. Although the max resource deviation of the FALB is higher than 28, the total execution of FALB is better than that of the original one. The original Yunikorn without DDRF causes other tenants' applications to wait when a large number of applications from a tenant consume certain resources. Moreover, this also increase the resource waste.

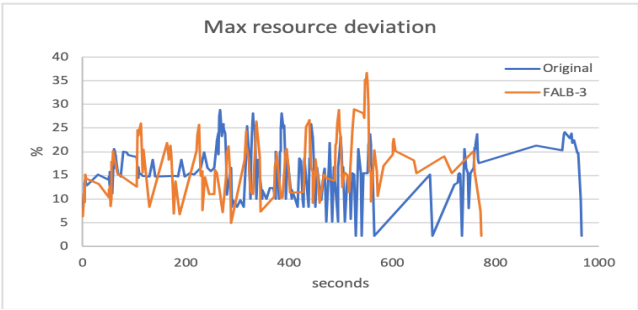

Fig. 5. deviations in the stream scenario

Fig. 6 shows the mean resource utilizations in the stream scenario. The memory utilization with FALB is stably higher than that of original Yunikorn after 270 seconds. FALB considering RRDF avoids scheduling a lot of same tenant's applications to exhaust specific resource. Comparing to the FALB, the original Yunikorn leads that other tenants' applications are waiting. The original Yunikorn increases the waiting time of application and the total execution time.

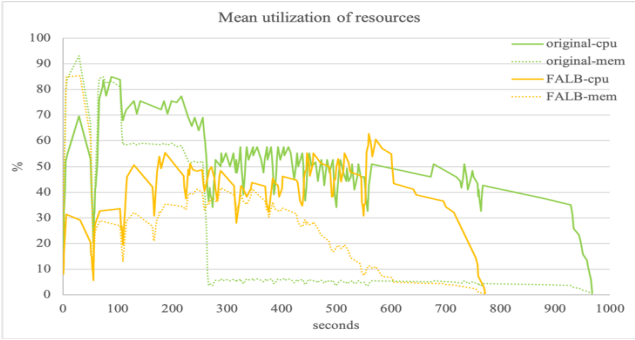

Fig. 6. mean utilizations of nodes in stream scenario

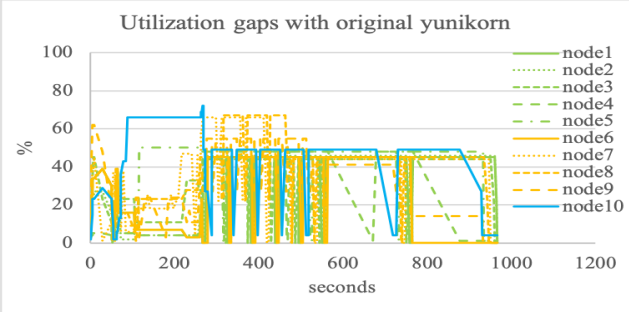

Fig. 7. utilization gaps in stream scenario (original)

Fig. 7 and **Fig. 8** show the utilization gaps by different strategies. A small utilization gap of a node is better when the node can't run new applications. **Fig. 8** indicates average median value of utilization gaps is around 47. Compared to **Fig. 7**, the RRDF mechanism avoids the tenants' application waiting and some utilization gaps are reduced. **Fig. 9** shows that the FALB keeps the difference of every tenant's *RRDF* close. In hence, the curves of different tenants would be close each other. No tenant owns too many resource quantities to violate the fairness.

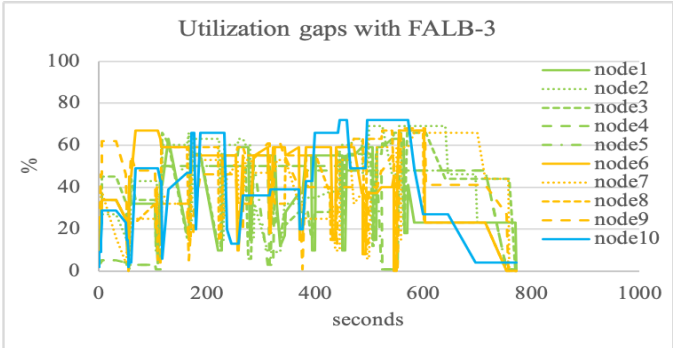

Fig. 8. utilization gaps in stream scenario (adopting FALB)

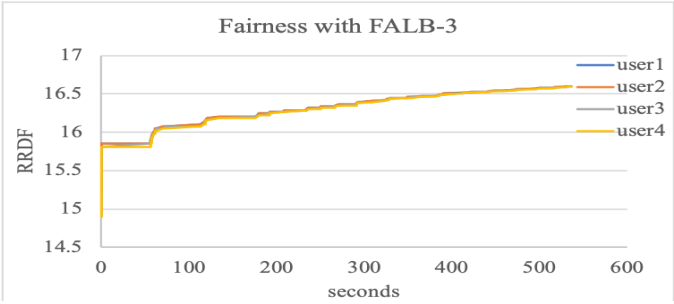

Fig. 9. fairness in batch scenario (adopting FALB)

5 Conclusions

This study improves the fairness by keeping the utilizations of tenants' *RRDF* close. After keeping the fairness among tenants, FALB provides the reduction of utilization gaps in a node, the decreament of max. deviation of resources in clusters to reduce the execution time. FALB evaluates nodes by TOPSIS in order to find a node to allocate an application. The results show that total execution time of the FALB is better than that of original Yunikorn.

6 Acknowledgement

This study was sponsored by the Ministry of Science and Technology, Taiwan, R.O.C., under contract numbers: MOST 111-2221-E-142-004-, and by the "Intelligent Manufacturing Research Center" (iMRC) from the Featured Areas Research Center Program within the framework of the Higher Education Sprout Project by the Ministry of Education, Taiwan, R.O.C.

References

1. Apache Yunikorn [Online]. (2022, December 8) Available: https://yunikorn.apache.org
2. Carrión, C. “Kubernetes scheduling: Taxonomy, ongoing issues and challenges,” *ACM Computing Surveys*, vol. 55, no. 7, pp. 1-37 2022.
3. Chung, W. C., Wu, T. L., Lee, Y. H., Huang, K. C., Hsiao, H. C., and Lai, K. C, “Minimizing resource waste in heterogeneous resource allocation for data stream processing on clouds,” *Applied Sciences*, vol 11, no. 1, pp. 149, 2020.
4. Hilman, M. H., Rodriguez, M. A., and Buyya, R. “Multiple workflows scheduling in multi-tenant distributed systems: A taxonomy and future directions,” *ACM Computing Surveys (CSUR)*, vol. 53, no. 1, pp.1-39, 2020.
5. Menouer, T. ”KCSS: Kubernetes container scheduling strategy,” *The Journal of Supercomputing*, vol. 77, no. 5,pp. 4267-4293, 2021.
6. Ramasamy, M., Balakrishnan, M., and Thangaraj, C.” Priority Queue Scheduling Approach for Resource Allocation in Containerized Clouds,” In *Inventive Computation Technologies 4*, Springer International Publishing, 2020, pp.758-765.
7. Wang, W., Liang, B., and Li, B. ”Multi-resource fair allocation in heterogeneous cloud computing systems,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 10,pp. 2822-2835, 2015.
8. Pfreundschuh, S., Brown, P. J., Kummerow, C. D., Eriksson, P.,and Norrestad, T.”GPROF-NN: A neural network based implementation of the Goddard Profiling Algorithm,” *Atmospheric Measurement Techniques Discussions*, vol. 15, no. 17, pp.5033-5060, 2022.