International Journal of Engineering Trends and Technology
ISSN: 2231–5381 / https://doi.org/10.14445/22315381/IJETT-V72I7P106

Volume 72 Issue 7, 58-68, July 2024

Original Article

Harmonizing Heterogeneous Hosts: A Strategic Framework for Docker Container Placement Optimization

Jalpa M. Ramavat[1], Kajal S. Patel[2]

[1]*Gujarat Technology University, Gujarat, India.*
[2]*Vishawakarma Government Engineering College, Gujarat, India.*

[1]*Corresponding Author : jalpa.ramavat.2012@vgecg.ac.in*

Received: 23 February 2024 Revised: 15 May 2024 Accepted: 10 June 2024 Published: 26 July 2024

Abstract *- Containerized applications are self-contained units of code executed within isolated environments called containers, encompassing all necessary dependencies like libraries and configuration files. Containerization gives portability, scalability, and efficiency. So, the rise in containerization will also increase the use of orchestration tools like Docker, Kubernetes, and others. Docker has simple deployment and is suitable for small numbers of containers. Docker Swarm is a management tool for Docker containers. Docker Swarm uses a spread strategy to place containers of services in a Docker cluster. Spread distributes containers evenly throughout the Docker swarm cluster, but load balancing in nodes with varying resources could be improved. So, a placement strategy is developed in this paper that considers the available resources of the node while placing a container on it. The results show improvements in load balancing and the completion time of service containers.*

Keywords *- Cloud computing, Containerization, Container scheduling, Docker swarm, Orchestration, Recourses, Spread strategy.*

1. Introduction

Cloud computing continues to dominate the IT landscape, providing scalable and on-demand access to computing resources, enabling organizations to enhance agility and efficiency in deploying and managing applications. The cloud ecosystem has diverse independent components such as containers, virtual machines, orchestrations, load balancers, applications, and security that function concurrently to provide the services [1][2]. Containers are ubiquitous in cloud computing, offering lightweight, portable, and scalable solutions. They empower microservices architectures and enhance resource efficiency. Containers drive agility and efficiency, enabling seamless application deployment and scaling. Containers have an advantage over standard virtual machines. This comparison is discussed in Section 2. With support from orchestration platforms like Kubernetes, Docker Swarm, Mesos, etc, containers facilitate easy management, rapid development, and robust isolation, fostering a dynamic and scalable cloud environment. Some famous orchestration tools are discussed in Section 3. Among this orchestration platform Docker Swarm offers vital features such as high availability, scalability, service discovery, load balancing, and security. Its seamless integration with Docker, user-friendly interface, and straightforwardness make it a compelling option for developers and organizations aiming to effectively manage and deploy containerized applications. The container placement and scheduling in Docker Swarm are discussed in Section 4.1. Docker Swarm Mode currently employs only the spread scheduling approach. This method distributes tasks evenly among the nodes in the cluster. When a new container arrives, it is assigned to the node with the fewest active containers without considering the node's resource utilization. As long as a node has enough resources to run a container and has the fewest running containers, the new container is placed there, regardless of the node's current resource load. Additionally, in Docker Swarm Mode, the scheduler attempts to assign new tasks to nodes that are not already executing a task for the same service. Suppose every cluster node is running at least one task for the service. In that case, the scheduler then selects the node with the fewest tasks related to that service before evaluating the overall task distribution across all nodes. Various researches have been done to optimize the container placement which are discussed in section 4.2. A short comparison of it is given in the table 1. The dynamic scheduling technique, in conjunction with the service performance framework for containerized clouds, is discussed in [16]. An ACO-based algorithm is used in [17], which improves resource utilization and performance. A hybrid form of the Lion Algorithm (LA) and the Whale Optimization Algorithm (WOA) was created as the Whale

Random update assisted Lion Algorithm (WR-LA), and it showed improvement in cost reduction [19]. The Makespan and completion time are improved, and results are compared to Kubernetes Scheduler by considering current and future resource utilization in Procon [20]. Availability-Aware container scheduling strategy is used in [21], and it improves application service availability in the cloud. To improve load balancing and response time algorithm is proposed that divides the containers into neighborhoods and improves the particle swarm algorithm [22]. To improve energy efficiency A genetic algorithm with mutation operations and control parameters changed is implemented in MATLAB [23]. Again, power consumption is improved in [24] by the Dynamic Container Placement (DCP) mechanism. The Resource-Aware Least Busy (RALB) [25] method is used to do load balancing in a containerized cloud. In most of the above work [19] [21] [23] [24] [25], real-time implementation is required. This research tries to improve load balancing and performance of service by decreasing the completion time of containers. The proposed model, Algorithm and Architecture of the system is discussed in Sections 4.3,4.4 and 4.5, respectively.

2. VM vs Container

Massively popular technology virtualization is possible due to virtual machines. The VMs are Infrastructure as a service (IaaS) that share the physical device with the host OS [4]. It requires its own physical memory and space. The fact that VMs work with the help of virtualization technology is not new. They use VM-specific software applications to recreate the virtual hardware. Hence, this way a hypervisor manages the resources like storage and memory. Containers are defined to be lightweight; as a result, they deliver higher efficiency. Unlike VMs, containers are platform-as-a-service (PaaS) components that work on the host OS. Recently, there has been a staggering peak in the development of containerized software. As per [5], nearly 79% of applications had implemented internal containerization in 2020. Since containers are capable of delivering real-time and real-world benefits, they are preferred over VMs. Under the hood, containers use a Linux utility called *cgroups* for resource sharing and a *namespace* for isolation [6]. Table 1 shows the comparison of Virtual Machine and Container.

3. Orchestration Tools

As more and more applications involve containers, it becomes essential to scale them rapidly. There was a demand for software that automatically regulates container scaling and management. Container orchestration is a tool designed to drive thousands of distributed containers. A few popular container orchestrators are Docker Swarm, Kubernetes, Apache Mesos, OpenShift, and Nomad.

3.1. Kubernetes

Kubernetes was introduced by Google in 2014. It is also known as K8s. It is an open-source orchestration tool for PaaS. It provides features like load balancing, deployment, and scalability, etc. It operates on a master and slave model to manage containerized applications [7]. It provides services like load balancing, storage management, rollouts, rollbacks, and self-healing. Pod is the smallest and fundamental unit of Kubernetes. A pod is a collection of one or more closely related containers that are placed next to one other and share resources. Alongside containers, pods encapsulate storage resources, a network IP, and a set of configurations dictating the operational aspects of the pod's container(s). The primary purpose of a pod is to execute a singular instance of an application; this facilitates horizontal scaling by utilizing multiple pods if needed. Information such as the CPU, memory, and storage for a container can be specified at the time of creating the pod. The scheduler uses this information to determine the optimal placement of pods. The allocation of compute resources can be expressed either as a requested amount or as a constraint on the maximum consumption allowed by the container [9]. The master consists of the API server, scheduler, controller manager, and Etcd for configuration storage. Worker nodes consist of Kubelet, Kube-proxy, and a container runtime and run containerized applications [10].

Table 1 . VM Vs. Container [3]

Aspect	Containers (CoaaS Model)	Virtual Machines (VMs)
Kernel Sharing	All containers share a single OS kernel with varying abstraction levels.	VMs require their own virtualized network, BIOS, CPU, and OS.
Usage	Preferred in PaaS and SaaS due to their lightweight nature and quick startup.	Typically less favored in PaaS and SaaS due to heavier resource overhead.
Size and Performance	Smaller in size, leading to quick startup, improved performance, and better compatibility.	Larger size, slower startup times, and potentially higher resource consumption.
Migration Characteristics	Supports lightweight and energy-efficient migration.	Requires hardware emulation for migration, potentially less energy-efficient.
Live System Updates	Facilitates live system updates using the underlying host OS, eliminating the need for hardware emulation.	Updates might involve more complex procedures due to the entire VM structure.
Energy Consumption	Container migration consumes less energy with minimal resource wastage.	VM migration may result in higher energy c onsumption and resource utilization.

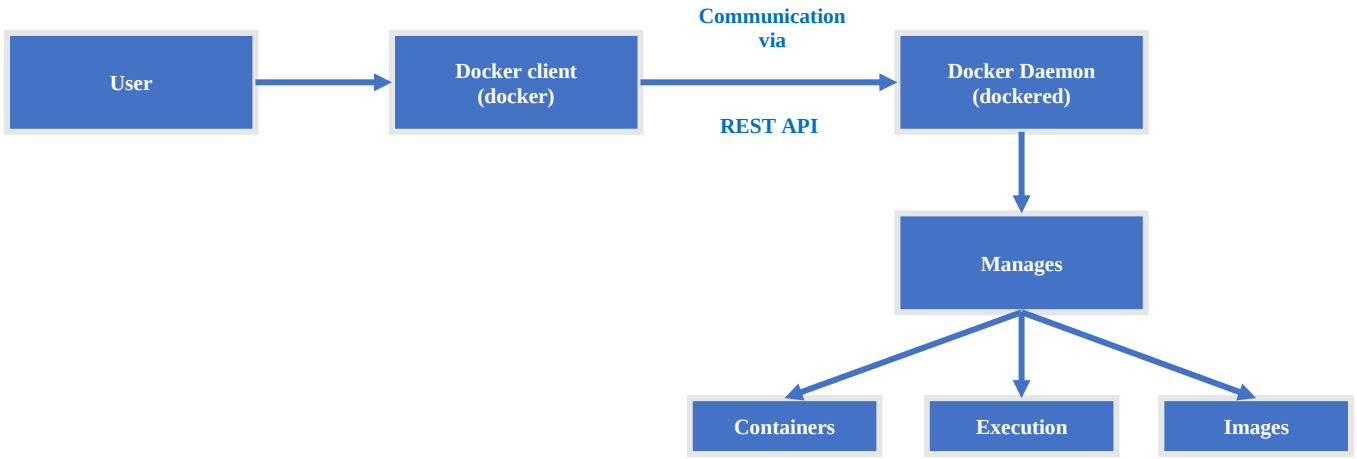

Fig. 1 Docker client-server architecture

3.2. Docker Swarm

Swarm serves as Docker's native platform for orchestrating containerized applications. It involves a collection of machines, whether physical or virtual, working collaboratively to run Docker applications. The Swarm Manager takes charge of swarm operations, overseeing container interactions across various host machines (nodes).

Docker Swarm maximizes the advantages of containers, facilitating the creation of highly portable and flexible applications, all the while ensuring redundancy for high application availability. Swarm managers are responsible for allocating workloads to the most suitable hosts based on the load of the node. The Swarm Manager manages scaling by adding or removing worker tasks to maintain the cluster's intended state [11].

3.3. MesOS

Apache Mesos is an open-source cluster manager used for data center environments. It deals with the difficulties of sharing resources across a range of workloads. Mesos provides fine-grained resource allocation and scalability. Its architecture consists of a master and slaves. A master for resource allocation and multiple slaves for task execution. Developers can write custom schedulers for their applications using the framework API. Mesos supports various resource allocation policies and offers fault tolerance. Real-world evaluations show its efficiency. Mesos is a scalable and extensible platform for resource sharing, benefiting data center environments and diverse workloads [12].

3.4. HashiCorp Nomad

Nomad is an open-source scheduler and orchestrator designed for deploying and managing applications at scale. It supports a range of workloads, including Docker containers, VMs, and standalone applications. It supports multiple workload types (containers, VMs, and more). It is a simple and flexible job specification. It has built-in high availability and fault tolerance[13].

4. Docker Container Placement and Scheduling in Docker Swarm

4.1. Docker Containers

Docker is open-source software for creating and running container applications. Docker mainly enforces client and server architecture. Hence, it has three central pieces - daemon, client, and REST API, which simultaneously are known as Docker Engine [13]. The client works as user-end software that allows users to connect to the daemon. The daemon is responsible for all the back-end handling of container creation, execution, and deployment. Moreover, the client and daemon communicate via the REST API interface. Figure 1 presents the architectural design of the Docker system [14]. The Docker daemon includes containers, images, configuration files, networking settings, and storage. While the daemon listens for Docker API, the docker client sends the request for the daemon. A Docker image is a layered read-only file that has the application software. Similarly, a container is an instance of the image. To meet the demands, Docker programmed Docker Swarm to govern a cluster of Docker nodes. Docker Swarm is the implementation of the Docker project called SwarmKit, which provides the core orchestration layer for Docker. Spread is the only scheduling approach that Swarm Mode presently uses [15]. In this approach, the tasks are distributed equally among the cluster's nodes. Therefore, when a new container arrives, it is assigned to the node with the fewest running containers. The resource utilization of the node is not considered. So if there are enough resources available on a selected node to run a container and the node has having least number of running containers, then it places the new container to the node irrespective of its current resource utilization. Another placement situation in Docker swarm mode is that the scheduler aims to assign a new task to a node, prioritizing nodes that are not currently executing a task for the same service. If all cluster nodes have at least one task for the service, the scheduler chooses the node with fewer tasks related to the same service before considering the overall task

distribution across all nodes. This scheduling approach is colloquially known as "HA scheduling" or High Availability scheduling [15]. These situations might lead to load imbalance within cluster nodes.

4.2. Related Work

H. Li [16] provides a thorough approach to tackling the problems associated with service performance by using dynamic scheduling techniques in conjunction with the service performance framework for container clouds in the big data era. Through the integration of real-time fundamental resource monitoring, service delays, and dynamic modifications grounded in particle swarm optimization, the framework seeks to optimize container-based service deployment under complex conditions. It is crucial to strike the ideal balance because the algorithm is sensitive to the weight assignment. In [17] uses a graph-based approach by mapping the scheduling problem to a graphical model and representing the service placement as a min-cost flow problem (MCFP), which allows encoding multi-resource requirements and affinities to other containers. ECSched presented concurrent container scheduling in heterogeneous clusters and showed that it enhanced the performance over queue-based container scheduling. The complexity of the algorithm is high and requires more study on resource dynamics and container dependencies. C. Kaewkasi et al. [18] describe an ACO and show how to utilize it in practice to create a new container scheduler for Docker. The ACO-based algorithm distributes application containers among Docker hosts in order to balance resource utilization overall, which improves application performance over the current greedy scheduler. The ACO-based algorithm outperformed the greedy one, according to the experimental data. It resulted in an approximate 15% improvement in overall application performance. But, the fine-tuning of parameters is required. K. N. Vhatkar et al. [19] introduce the Whale Random update assisted Lion Algorithm (WR-LA), which is the hybrid form of the Lion Algorithm (LA) and Whale Optimization Algorithm (WOA). With the use of this algorithm, they can get advantages of both algorithms by incorporating the WOA in LA in place of the fertilization function. Performance evaluation shows that WR-LA outperforms other models, demonstrating a cost reduction of 9.58% to 21.63% compared to SW-GA, SH-GA, GM-GA, LA, and WOA at various iterations. They simulated the algorithm. Actual environment container behaviour may be different and also, the computational time might increase. Y. Fu et al. [20] introduced the ProCon container placement scheme in the Kubernetes cluster. Procon considers both current and future resource utilization. The author introduces two algorithms: one for calculating the contention rate for workers based on resource utilization and remaining time, and the other places containers on available workers having lower contention rates. Procon demonstrates significant reductions in completion time (up to 53.3%) and improvement in Makespan (up to 37.4%) compared to the Kubernetes in-built scheduler. It requires monitoring the application and also increases response time. Y. Alahmad et al.[21] proposed an Availability-Aware container scheduling strategy. It improves application service availability in the cloud. Availability is measured by Mean Time To Fail (MTTF) and Mean Time To Repair (MTTR). The strategy finds availability for VMS and Hosts, selects them based on high availability, and places containers based on availability, functional resource constraints and dependency constraints. This strategy achieves the highest service availability and maintains acceptable host CPU utilization. Implementation is done on Cloudsim with simulated workloads. The simulated environment may have different behaviour than the actual environment, so the algorithm must be implemented in a real cloud environment. Y. Guo [22] proposed a method which focuses on Load balancing and Response time in order to improve the system performance. This algorithm divides the containers into neighbourhoods and improves the particle swarm algorithm in terms of results. It improves the efficiency at a rate of 20 to 25 percent more than commonly used swarm algorithms.

Chen et al. [23] use the Improved Genetic algorithm in which mutation operation and control parameter is changed. It has proved more efficient than the particle swarm algorithm, Conventional GA and first fit algorithm when Virtual machine resource utilization is high. It is also more efficient than binpack and spread in terms of energy efficiency. This algorithm has two mutation functions and also the control parameter to choose the Mutation operation to be performed on the data. It only tries to optimize only one objective energy efficiency. They implemented in a simulation environment of MATLAB, so implementation in a real cloud environment is required. Alwabel [24] presents a Dynamic Container Placement (DCP) mechanism. It is for energy-efficient management in Container-as-a-Service (CaaS) cloud systems. It extends the Whale Optimization Algorithm (WOA) to minimize power consumption by optimizing the placement of containers on virtual machines (VMs) and Physical Machines (PMs). DCP is compared with IGA (improved genetic algorithm) and DWO(discrete whale optimization) mechanisms for homogeneous and heterogeneous cloud systems. The results show that in homogenous clouds, DCP reduces the search time by around 50% and consumes approximately 78% less power. Whereas, in heterogeneous clouds, DCP reduces search time by around 30% and conserves power by 85%. More parameters should be considered for optimization and implemented in the real environment. Bouflous[25] proposed the Resource-Aware Least Busy (RALB) method. The main focus of this work is load balancing in a containerized cloud environment. RALB optimizes workload distribution by taking container migration time and server resource capabilities into account. RALB shows improvement over conventional random algorithms in terms of performance, resource consumption, and improved quality of service. However, it must be implemented in a real scenario. Table 2 shows a short comparison of all the above work done.

Table 2. Comparison of container scheduling strategies

Algorithm	Key Features	Benefits	Limitations
Dynamic scheduling technique [16]	Service performance framework using dynamic scheduling and particle swarm optimization	Resource monitoring and service delay management	Sensitive to the weight assignment
ECSched [17]	Graph-based approach uses a Min-Cost Flow Problem (MCFP)	Enhanced the performance over queue-based container scheduling in a heterogeneous cluster	The complexity of the algorithm is high
ACO [18]	Ant Colony Optimization for workload scheduling	Improvement in overall application performance	Fine-tuning of a parameter is required.
Whale Random update assisted Lion Algorithm (WR-LA)[19]	Uses a hybrid form of the Lion Algorithm (LA) and Whale Optimization Algorithm (WOA).	Cost Reduction	Implemented in a simulated environment.
ProCon [20]	Progress rate estimation. Considers both current and future resource utilization.	Reductions in completion time and improvement in Makespan	Requires monitoring the application and also increases response time. For Short-lived Containers
Availability-Aware container scheduling [21]	Enhances service availability using MTTF and MTTR	Achieves the highest service availability and maintains host CPU utilization	Needs real-world system implementation
Neighbourhood division [22]	Divides the containers into neighbourhoods and improves the particle swarm algorithm	Focus on Load balancing and Response time, improves the efficiency	Needs real-world system implementation
Improved GA [23]	It Improved the genetic algorithm in which mutation operation and control parameter is changed.	Improve energy efficiency	Only one parameter is considered: Needs real-world system implementation
DCP [24]	It extends the Whale Optimization Algorithm (WOA) for dynamic container placement.	Power savings and reduced search time.	Needs real-world system implementation
RALB [25]	Resource-Aware Least Busy algorithm for load balancing	Better resource utilization, improved performance, enhanced QoS	Needs real-world system implementation

4.3. Proposed Model

The Spread algorithm does not consider current resource utilization and may create load imbalance. To overcome this, some modification in the spread is introduced in the proposed algorithm. Before placing the container on a node, it checks the utilization of resources in all the nodes in the cluster and finds the node with minimum resource utilization. Currently, the algorithm only considers the CPU utilization of a node. And find the best node for initial container placement.The cluster of three nodes is created by using an Oracle VM virtual box. This cluster contains 3 nodes, of which one is the manager and the other two are worker nodes. Currently only considers CPU utilization.The manager node has 4GB RAM and 2 CPU cores. At the same time, worker nodes have 3GB RAM and 2 CPU cores.

The algorithm uses Docker Python API for container placement if a set of Nodes are denoted as {n1,n2… ni}.

The resource considered is the CPU

Total CPU on node n_i is: Ci

Total no. of containers on node i is Ki

Resource used by container j on node i are RC_i^j.

So total resource(CPU) used by all containers running on node i is given by equation 1.

$$UC_i = \frac{\sum_{j=1}^{k_i} RC_i^j}{c_i} \quad (1)$$

Then, the algorithm is going to deploy a container on a node having minimum CPU utilization, as shown in Equation 2.

$$\text{Min(UC)} \quad (2)$$

4.4. Proposed Algorithm

Algorithm

Step 1. Start the Docker Swarm Cluster
Step 2. For each service in the list, do steps 3 to 8
Step 3. For each node ni in the cluster
Step 4. For each container running on the node
Step 5. Find the CPU used by container j on node i RC_i^j
Step 6. Total CPU used by all containers on node ni is

$$UC_i = \frac{\sum_{j=1}^{k_i} RC_i^j}{c_i}$$

Step 7. Select a node having minimum UC.
Step 8. Create a service for the image and place its container on the selected node

4.5. Proposed Architecture

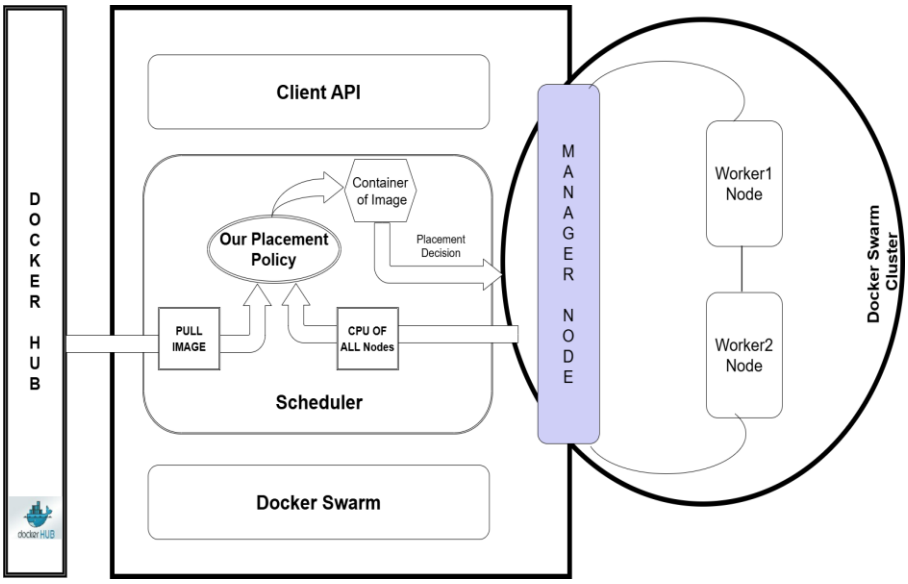

Fig. 2 Architecture of placement of container

Figure 2 describes the architecture of the proposed model. Here Docker Swarm cluster with 3 nodes is created. As factorials utilize more CPU in calculations, A Python script is created to find factorials of 5lac and 10lac. Then, it is converted to a Docker image and pushed to the Docker Hub. Now, the services from the same image are created and run on Docker cluster nodes selected by the proposed algorithm. To get the load of cluster nodes and other matrices, Prometheus and Graphana are used and deployed by using the Swarmprom[26] stack. Swarmprom has various monitoring service containers that are running on all three nodes to monitor the cluster. To see the workings of the inbuilt spread strategy in the Docker swarm cluster, some experiments are done.

4.6. Experiments and Results

Initially, nodes of the cluster have had the following containers (Swarmprom) running on them (Figure 3).

Now the service of factorial of 10lac is created from its image. The container of it is placed on the Worker1 node by default strategy (Figure 4).

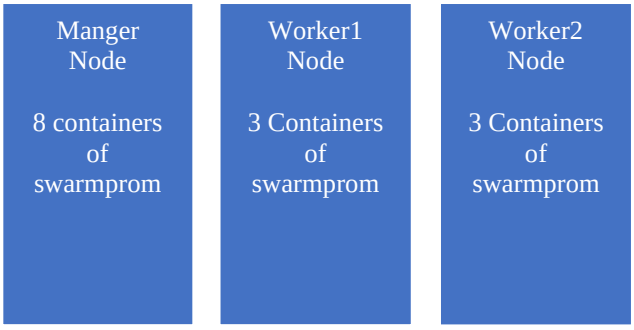

Fig. 3 Initial Docker swarm node status

Then, to understand the working of the default strategy of Docker Swarm, 12 services of nginx image are created, and Docker Swarm does container placement is observed in experiment 1 (Figure 5).

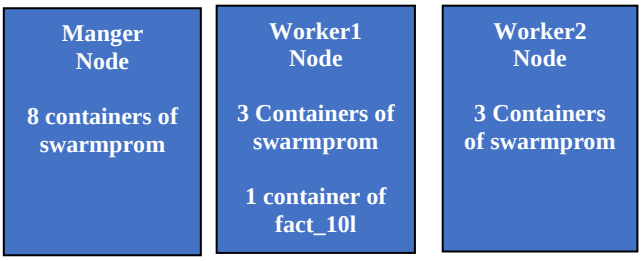

Fig. 4 Placement of factorial 10lac service container on worker1

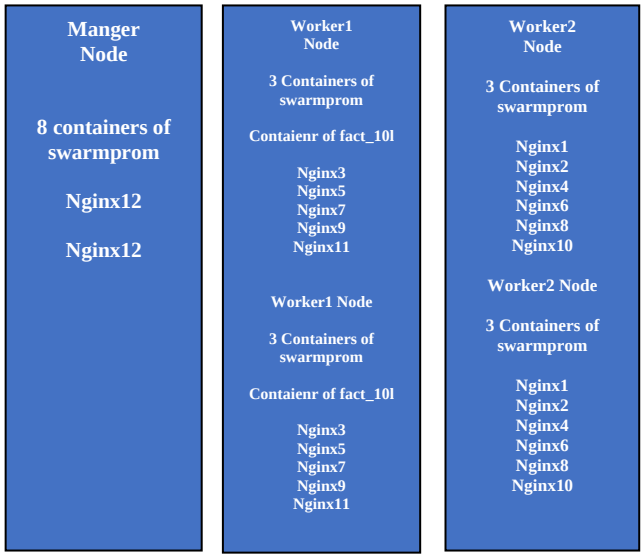

Fig. 5 Placement of Nginx services
(using default strategy to demonstrate the behaviour of spread)

As shown in Figure 5 initially, Worker2 has 3 containers of Swarmprom stack, which are the least among all nodes, so nginx1 and nginx2 services containers are placed on Worker2. Now, irrespective of the container of factorial 10lac CPU Usage on the Worker2 node, other Nginx containers are distributed between the Worker2 and Worker1 nodes. So spread will not consider the current CPU usage on the node while placing a new container.To compare the proposed minimum CPU usage strategy with spread, the CPU load is generated by creating 3 services of factorial 5lac and 1 service of factorial 10lac in experiment 2.

By using the Spread Strategy of the Docker swarm, the following distribution of containers is obtained among Docker cluster nodes (Figure 6).As shown in Figure 6, all 4 containers are distributed among Worker1 and Worker2 Node. As the manager contains 8 number of Docker Swarm monitoring Containers, no new container is placed on it.

Now, by using Graphana, the CPU usage of all nodes while using the Spread strategy is obtained (Table 3) and plotted in the graph (Figure 7). CPU usage by the proposed strategy is shown in Table 4 and plotted in Figure 9.

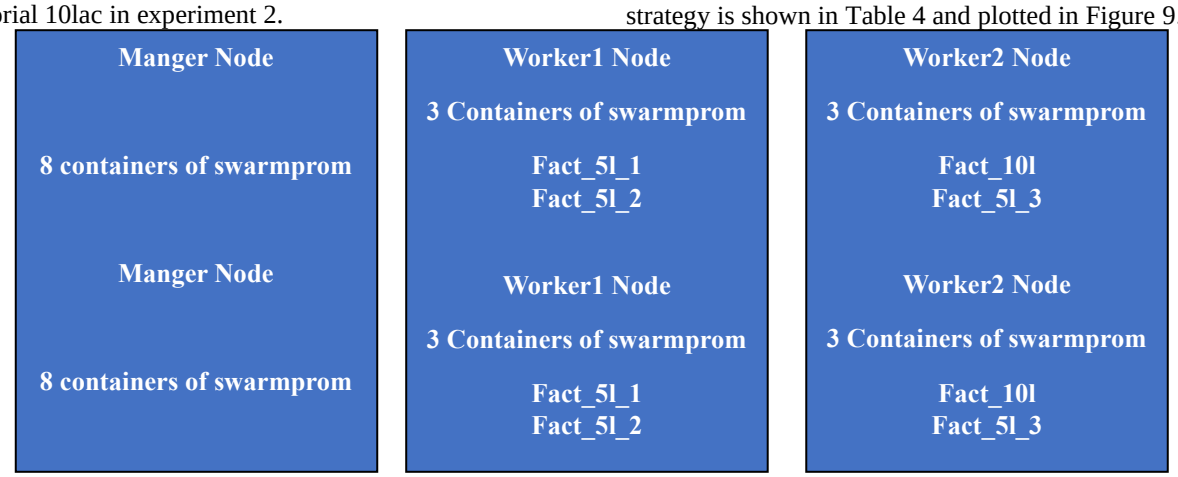

Fig. 6 Placement of 1 factorial 10lac service and 3 factorial 5lac service container(Using Default strategy)

Table 3. Container Utilization having 1 10lac factorial and 3 5lac factorial services containers (Default Strategy)

Time (seconds)	0	90	180	270	360	450	540	630	720	810	900
Manager(%)	20.233	18.322	20.333	10.567	32.667	12.5	12.3	12.318	24.377	15.445	13.539
Worker1(%)	100	97.2	93.4	50.027	99.7	51.533	97.6	55.303	96.299	85.874	95
Worker2(%)	56.17	99.5	99.067	98.967	67.349	88.299	58.003	99.8	85.916	98.633	96.667

Table 4. Container Utilization having 1 10lac factorial and 3 5lac factorial services containers (Proposed Strategy)

Time (Seconds)	0	90	180	270	360	450	540	630	720	810	900
Manager(%)	53.95	56.397	54.036	54.282	39.499	53.172	54.661	58.164	58.317	53.907	58.512
Worker1(%)	99.339	79.583	91.026	85.77	93.45	96.071	79.496	95.014	97.031	92.483	88.933
Worker2(%)	60.371	59.386	61.595	58.705	61.708	59.153	59.872	60.204	59.49	61	59.202

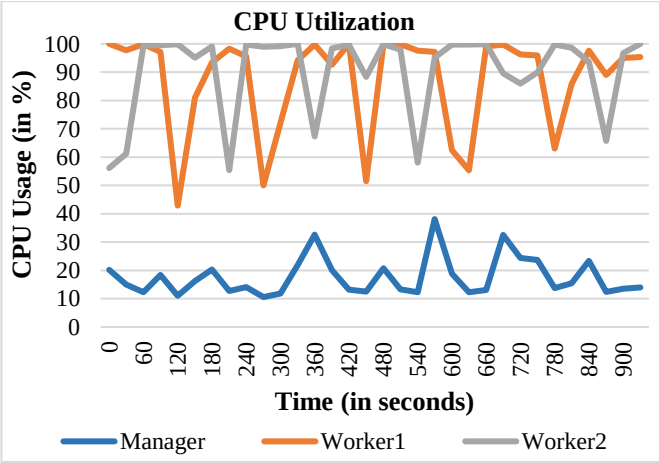

Fig. 7 CPU Usage of all Nodes having 1 10lac factorial and 3 5lac factorial services containers (Using Default strategy)

As seen in Figure 7, the Load is unbalanced among the three nodes Manager node has having least CPU Load where, whereas Worker1 and Worker2 are overloaded. Now some services placement is done by the proposed strategy in Figure 8.

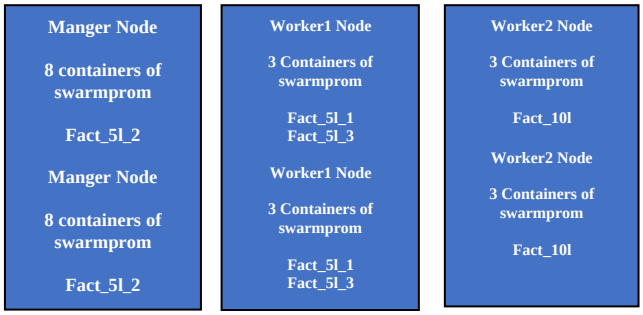

Fig. 8 Placement of 1 factorial 10lac service and 3 factorial 5lac service container(Using Proposed strategy)

As shown in Figure 8, the new containers are distributed among all nodes according to the current CPU load of nodes. Factorial 5lac services 1 and 3 are running on Worker1, and Factorial 5lac Service 2 is running on Manager. Again, the Factorial 10lac service is running on the Worker2 node. Figure 9 shows the CPU load is almost balanced among all three nodes by the proposed strategy. Completion time of containers of each service is calculated in both cases i.e. by using the Spread strategy and the proposed approach. Table 5 shows the Completion time of four containers of factorial 5lac services 1,2, 3 and 2 containers for factorial 10lac service and the average completion time using both the default and proposed placement approaches.

Figure 10 shows the graphical representation of the above values of Completion Time. As shown in Figure 10, four containers of fact_5l_1, fact_5l_2, and fact_5l_3 services and two containers of fact_10l services. As shown in the result, minor improvement in completion time is found as only factorial services are running. Another experiment, 3, is done in order to see the effect of resource-based placement. Now, mixed services are taken, and the results are obtained. Two nginx services named Nginx and Nginx1 are created. Created 5 tasks (containers) for each of these services and placed them on the manager and Worker1 Node by using placement constraint of node id in Docker swarm as shown in following Figure 11.

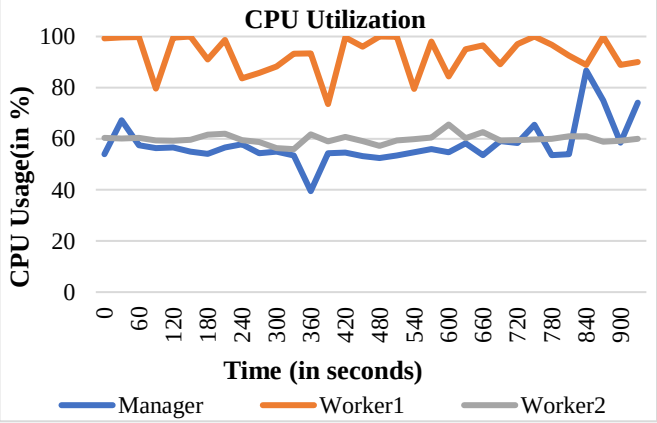

Fig. 9 CPU Usage of all Nodes having 1 10lac factorial service and 3 5lac factorial service container(Using Proposed strategy)

Table 5. Completion time of containers of factorial services (Experiment 2)

	Services	Containers				AVG
		1	2	3	4	
Propose Strategy	Fact_5l_1	145	142	136	139	140.5
	Fact_5l_2	125	137	133	129	131
	Fact_5l_3	148	142	147	142	144.75
	Fact_10l	485	582	-	-	533.5
Default Strategy	Fact_5l_1	132	144	150	151	144.25
	Fact_5l_2	147	142	147	152	147
	Fact_5l_3	153	140	140	147	145
	Fact_10l	710	660	-	-	685

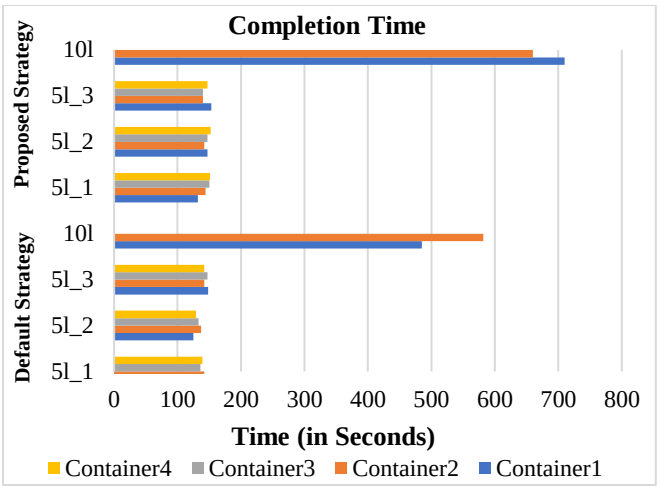

Fig. 10 Comparison of completion time for factorial 5lac and 10lac services containers (Using default strategy and proposed strategy)

Now, three services for factorial of 5lac are created and placed by default spread strategy of the Docker Swarm. All of the three services are placed on the worker2 node (Figure 12) because the manager node has having total of 13 containers of Swarmprom and Ngnix services, and the Worker 1 node has having total of 8 containers of Swarmprom and Ngnix1 services. Worker 2 has only 3 containers of Swarmprom, so it has having least number of running containers, and all new Factorial 5lac services are placed on the Worker 2 node.

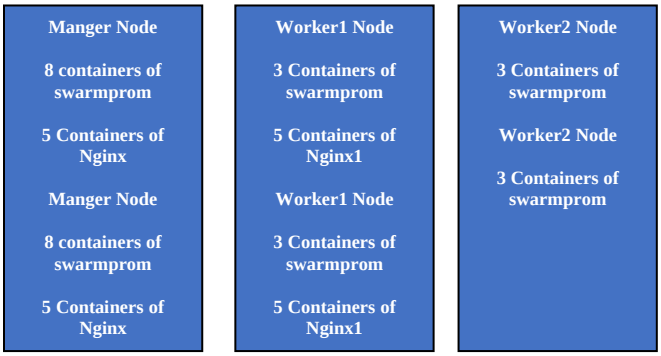

Fig. 11 Placement of Nginx services containers (Using placement criteria on manager and worker1)

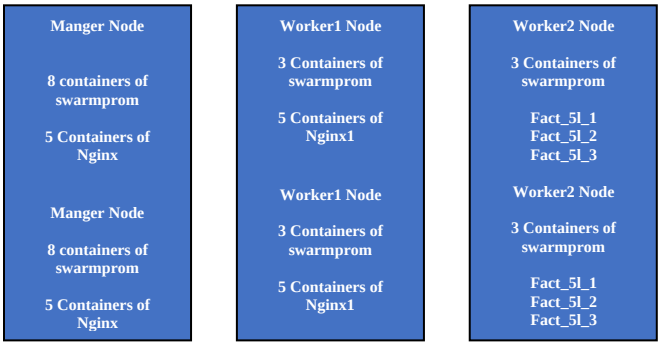

Fig. 12 Placement of Nginx, Nginx1 services and 3 factorial 5lac services containers (Nginx and Ngnix1 services are placed using placement criteria on node manager and worker1 and Factorial services are placed using Default strategy)

Table 6. CPU Utilization of nodes having 5 Nginx, 5Nginx1 and 3 Factorial 5lac services (Using Default Strategy)

Time(Seconds)	0	90	180	270	360	450	540	630	720	810	900
Manager(%)	12.922	6.933	61.867	12.954	13.806	25.228	16.3	9.152	11.327	9.006	9.748
Worker1(%)	19.8	23.033	20.611	16.672	14.933	14.933	16.356	12.933	14.194	12.798	13.6
Worker2(%)	85	99.667	99.767	98.6	99.767	96.833	98.267	99.8	100	99.933	98.767

Table 7. CPU Utilization of nodes having 5 Nginx, 5Nginx1 and 3 Factorial 5lac services (Using Proposed Strategy)

Time(Seconds)	0	90	180	270	360	450	540	630	720	810	900
Manager(%)	95.672	67.508	68.916	69.771	62.099	66.672	85.09	93.667	74.375	59.091	84.852
Worker1(%)	55.138	53.064	52.912	47.745	52.662	55.063	52.046	51.716	50.129	52.825	51.48
Worker2(%)	73.103	51.404	52.63	38.648	57.164	52.956	53.221	53.34	51.631	51.486	51.33

The CPU load Utilization taken from Prometheus and Graphana for the Default strategy of Experiment 3 is shown in Table 6. and its graphical representation is in Figure 13. As displayed in Figure 13, the Worker2 node is overloaded, as all CPU-oriented tasks are running on it. Now, proposed placement criteria are used as shown in equation 1 to find out minimum CPU utilization and then place tasks accordingly. So, the factorial 5lac services are distributed among all three nodes, as shown in Figure 14. The CPU Utilization is shown in Table 7 and plotted in Figure 15. As shown in Figure 15, CPU load is more evenly distributed among all nodes. The manager has the highest CPU utilization due to its monitoring tasks container. But load balancing is found among all cluster nodes.

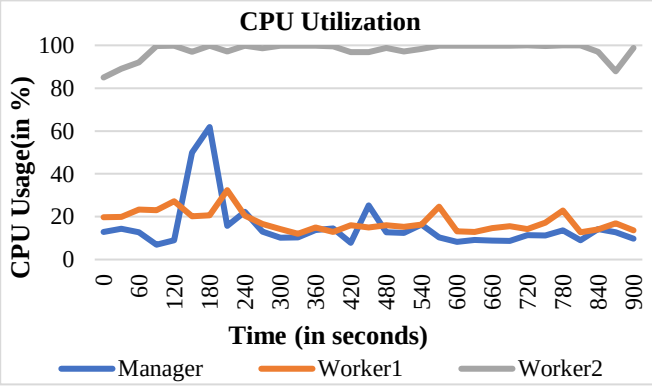

Fig. 13 CPU usage of all nodes having 5 Nginx, 5 Nginx1 and 3 factorial 5lac services containers(Using default strategy)

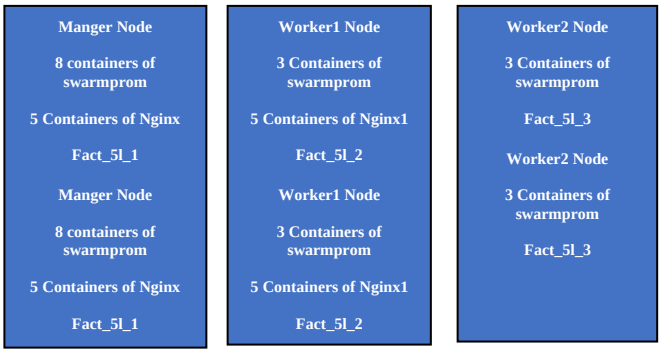

Fig. 14 Placement of Nginx, Nginnx1 services and 3 factorial 5lac services containers (Nginx and Ngnix1 services are placed using placement criteria on node manager and worker1 and Factorial services are placed using Proposed Strategy)

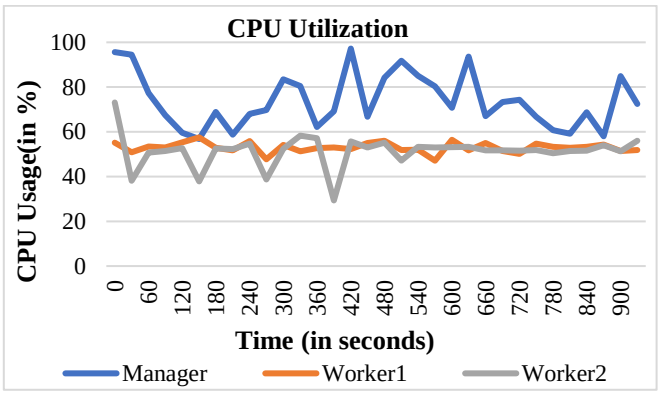

Fig. 15 CPU usage of all nodes having 5 Nginx, 5 Nginx1 and 3 factorial 5lac services containers (Nginx and Ngnix1 services are placed using placement criteria on node manager and worker1 and Factorial services are placed using proposed strategy)

Table 8. Completion time of containers of factorial services (Experiment 3)

Services		Containers				AVG
		1	2	3	4	
Proposed Strategy	Fact_5l_1	108	123	115	126	118
	Fact_5l_2	119	121	122	124	121.5
	Fact_5l_3	111	119	117	120	116.75
	Average Completion Time of 4 Containers (Proposed Strategy)					118.75
Default Strategy	Fact_5l_1	160	187	177	152	169
	Fact_5l_2	164	184	157	162	166.75
	Fact_5l_3	161	181	160	161	166.75
	Average Completion Time of 4 containers (Default Strategy)					167.17

Now, the completion time of the container of service is calculated. Table 8 shows the completion time of 4 containers for each factorial 5lac 1, 2 and 3 services. The major difference is seen in Completion time. The proposed strategy (minimum CPU utilization) shows improvement in completion time as compared to Docker's default Spread strategy.

The average in Table 8 shows containers in the proposed strategy are taking almost 49 seconds less time than the default Spread strategy. The Graphical representation of Table 8 is shown in Figure 16.

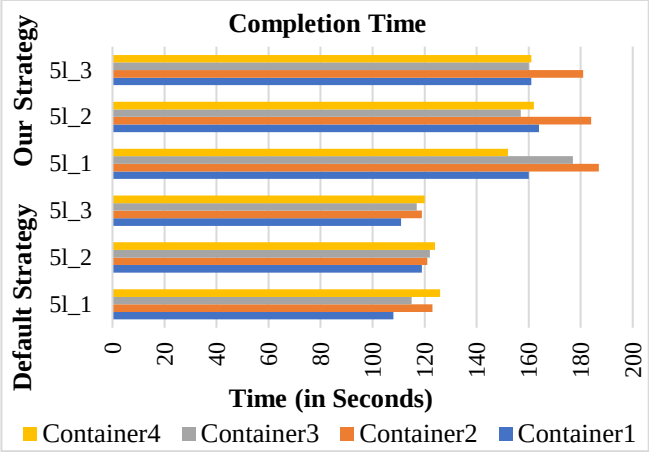

Fig. 16 Comparison of completion time for 3 factorial 5lac services containers (Using default strategy and proposed strategy)

5. Conclusion

In this paper container placement scheme base on resource usage is proposed. The proposed approach considers resource usage by current task and availability of resources on the node.

It is compared with the existing spread scheme of the docker swarm cluster.The improvements in Completion time and overall performance of service are found. It also has a more balanced CPU load across all nodes compared to spread.

Right now, the algorithm is only considering the CPU resource of Docker Swarm cluster nodes. In the future algorithm can be improved by including memory and other parameters for container placement in the Docker swarm cluster.

References

[1] Cloud Ecosystem, 2021. [Online]. Available: https://www.includehelp.com/cloud-computing/cloud-ecosystem.aspx

[2] Wei-Tek Tsai, Xin Sun, and Janaka Balasooriya, "Service-Oriented Cloud Computing Architecture," *2010 Seventh International Conference on Information Technology: New Generations*, Las Vegas, NV, USA, pp. 684–689, 2010. [CrossRef] [Google Scholar] [Publisher Link]

[3] Neeraj Kumar et al., "Renewable Energy-Based Multi-Indexed Job Classification and Container Management Scheme for Sustainability of Cloud Data Centers," *IEEE Transactions on Industrial Informatics*, vol. 15, no. 5, pp. 2947-2957, 2019. [CrossRef] [Google Scholar] [Publisher Link]

[4] Rabindra K. Barik et al., "Performance Analysis of Virtual Machines and Containers in Cloud Computing," *2016 International Conference on Computing, Communication and Automation*, Greater Noida, India, pp. 1204–1210, 2016. [CrossRef] [Google Scholar] [Publisher Link]

[5] IBM Market Development & Insights, Containers in the Enterprise Rapid Enterprise Adoption Continues, 2020. [Online]. Available: https://www.ibm.com/downloads/cas/VG8KRPRM

[6] Understanding the Docker Internals, Nitin AGARWAL 2017. [Online]. Available: https://medium.com/@BeNitinAgarwal/understanding-the-docker-internals-7ccb052ce9fe

[7] Kubernetes Documentation, Overview | Kubernetes, 2023. [Online]. Available : https://kubernetes.io/docs/concepts/overview/

[8] Kubernetes Doc, Concepts | Kubernetes, 2020. [Online]. https://kubernetes.io/docs/concepts/

[9] Maria Rodriguez, and Rajkumar Buyya, "Container Orchestration with Cost-Efficient Autoscaling in Cloud Computing Environments," *Handbook of Research on Multimedia Cyber Security, IGI global*, pp. 190-213, 2020. [CrossRef] [Google Scholar] [Publisher Link]

[10] Kubernetes Components, 2023. [Online]. Available: https://kubernetes.io/docs/concepts/overview/components/

[11] Swarm Mode Overview, Docker. Docs. [Online]. Available: https://docs.docker.com/engine/swarm/

[12] Benjamin Hindman et al., "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center," *8th USENIX Symposium on Networked Systems Design and Implementation*, 2011. [Google Scholar] [Publisher Link]

[13] Docker Overview, Docker Docs. [Online]. Available: https://docs.docker.com/get-started/overview/

[14] Docker Architecture, Docker Docs. [Online]. Available: https://docs.docker.com/get-started/overview/#docker-architecture

[15] Scheduling Services on a Docker Swarm Mode Cluster, 2017. [Online]. Available: https://semaphoreci.com/community/tutorials/scheduling-services-on-a-docker-swarm-mode-cluster

[16] Han Li et al., "A Service Performance Aware Scheduling Approach in Containerized Cloud," *2020 IEEE 3rd International Conference on Computer and Communication Engineering Technology*, Beijing, China, pp. 194-198, 2020. [CrossRef] [Google Scholar] [Publisher Link]

[17] Yang Hu et al., "ECSched: Efficient Container Scheduling on Heterogeneous Clusters," *Euro-Par 2018: Parallel Processing: 24th International Conference on Parallel and Distributed Computing*, Turin, Italy, pp. 365-377, 2018. [CrossRef] [Google Scholar] [Publisher Link]

[18] Chanwit Kaewkasi, and Kornrathak Chuenmuneewong, "Improvement of Container Scheduling for Docker Using Ant Colony Optimization," *2017 9th International Conference on Knowledge and Smart Technology*, Chonburi, Thailand, pp. 254-259, 2017. [CrossRef] [Google Scholar] [Publisher Link]

[19] Kapil N. Vhatkar, and Girish P. Bhole, "Optimal Container Resource Allocation in Cloud Architecture: A New Hybrid Model," *Journal of King Saud University - Computer and Information Sciences*, vol. 34, no. 5, pp. 1906–1918, 2022. [CrossRef] [Google Scholar] [Publisher Link]

[20] Yuqi Fu et al., "Progress-Based Container Scheduling for Short-Lived Applications in a Kubernetes Cluster," *2019 IEEE International Conference on Big Data*, Los Angeles, CA, USA, pp. 278–287, 2019. [CrossRef] [Google Scholar] [Publisher Link]

[21] Yanal Alahmad, Tariq Daradkeh, and Anjali Agarwal, "Availability-Aware Container Scheduler for Application Services in Cloud," *2018 IEEE 37th International Performance Computing and Communications Conference*, Orlando, FL, USA, pp. 1–6, 2018. [CrossRef] [Google Scholar] [Publisher Link]

[22] Yanghu Guo, and Wenbin Yao, "A Container Scheduling Strategy Based on Neighborhood Division in Micro Service," *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*, Taipei, Taiwan, pp. 1–6, 2018. [CrossRef] [Google Scholar] [Publisher Link]

[23] Rong Zhang et al., "A Genetic Algorithm-Based Energy-Efficient Container Placement Strategy in CaaS," *IEEE Access*, vol. 7, pp. 121360–121373, 2019. [CrossRef] [Google Scholar] [Publisher Link]

[24] Abdulelah Alwabel, "A Novel Container Placement Mechanism Based on Whale Optimization Algorithm for CaaS Clouds," *Electronics*, vol. 12, no. 15, pp. 1-19, 2023. [CrossRef] [Google Scholar] [Publisher Link]

[25] Zakariyae Bouflous, Mohammed Ouzzif, and Khalid Bouragba, "Resource-Aware Least Busy (RALB) Strategy for Load Balancing in Containerized Cloud Systems," *International Journal of Cloud Applications and Computing*, vol. 13, no. 1, pp. 1-14, 2023. [CrossRef] [Google Scholar] [Publisher Link]

[26] Swarmprom, 2021. [Online]. Available: https://github.com/stefanprodan/swarmprom