MULTICRITERIA SCHEDULING OF MICROSERVICES IN CONTAINER BASED CLOUD ENVIRONMENT

A Synopsis

For the award of

Doctor of Philosophy

In

Computer/ IT Engineering

Submitted By:

Prajapati Anilkumar Amrutlal

[209999913024]

Supervisor:

Dr. Manish M Patel

Professor

Sankalchand Patel College of Engineering, Visnagar

Sankalchand Patel University,Gujarat,India

Submitted to

Gujarat Technological University

Ahmedabad, Gujarat, India.

Supervisor	**DPC Member-1**	**DPC Member-2**
Dr. Manish M. Patel	Dr. Kirit Modi	Dr. Rajan Patel

Abstract

The rapid development of cloud computing and microservice oriented application architectures has increased the need for efficient container scheduling and intelligent resource management in cloud computing environments. Containerization technologies such as Docker and container orchestration platforms like Docker Swarm offers scalable and flexible deployment of containerised microservices. However, efficient container scheduling is remains a major challenge due to dynamic workloads, imbalanced resource usage and QoS requirements. Traditional Docker Swarm scheduling mechanisms such as Random, Spread, and Binpack scheduling are static in nature and often lacks to improve startup time, response time, and resource utilization in container-based cloud environments.

The research proposes a Multi-Criteria Scheduling of Containerized Microservices using Reinforcement Learning (MCSCM-RL) framework for intelligent and adaptive container scheduling in Docker Swarm environments. The proposed framework integrates Deep Q-Network (DQN) based reinforcement learning with multi-criteria node ranking using performance metrics such as CPU utilization, memory utilization, container count, startup time, and response time. The research work is implemented and validated on a three node Docker Swarm cluster using CPU intensive microservice applications under clean load and cumulative load deployment conditions. Experimental results shows that the proposed MCSCM scheduler improves startup time, response time, resource utilization, and energy efficiency compared to existing Docker Swarm scheduling mechanisms. The research provides an energy-aware and QoS-aware container scheduling for modern cloud-based systems.

Keywords: *Containerization, Microservices, Docker Swarm, Reinforcement Learning*

Table of Content

1.
Brief Description on the state of the art of the research topic

1.1 Background

Now a days, Microservice Architecture as a modern software development paradigm has been adopted due to the rapid development of cloud computing and large-scale internet applications. In microservice architecture, an application is decomposed into multiple small, independent, and loosely coupled services that communicate through lightweight protocols such as REST APIs. Each microservice performs a business functionality and it can be independently developed, deployed, scaled, and maintained. Compared to traditional monolithic applications, microservices offer improved scalability, fault isolation, flexibility, and continuous deployment support. Due to these, today microservice-based applications are widely used in cloud-based environments, enterprise systems, and large-scale web applications. Figure-1 illustrates the evolution of software application architecture from traditional monolithic architecture to modern microservice-based architecture [1,2].

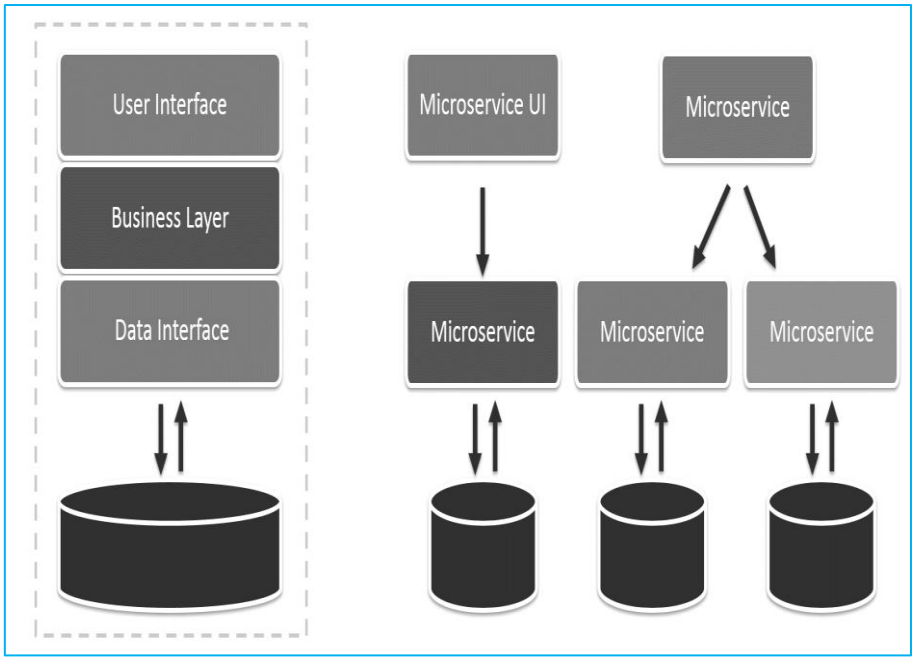


Figure 1- Monolithic to Microservice [1]

To efficiently deploy and manage microservices, containerization has become a key crucial solution. Microservices are packaged in a container along with their dependencies, libraries, and runtime environments into lightweight and portable units. In contrast to traditional virtual machines, containers share the host operating system kernel, and offer

reduced overhead, quick startup time, better portability, and high deployment frequency. Figure-2 illustrates the architectural difference between traditional VM based virtualization and container-based virtualization.

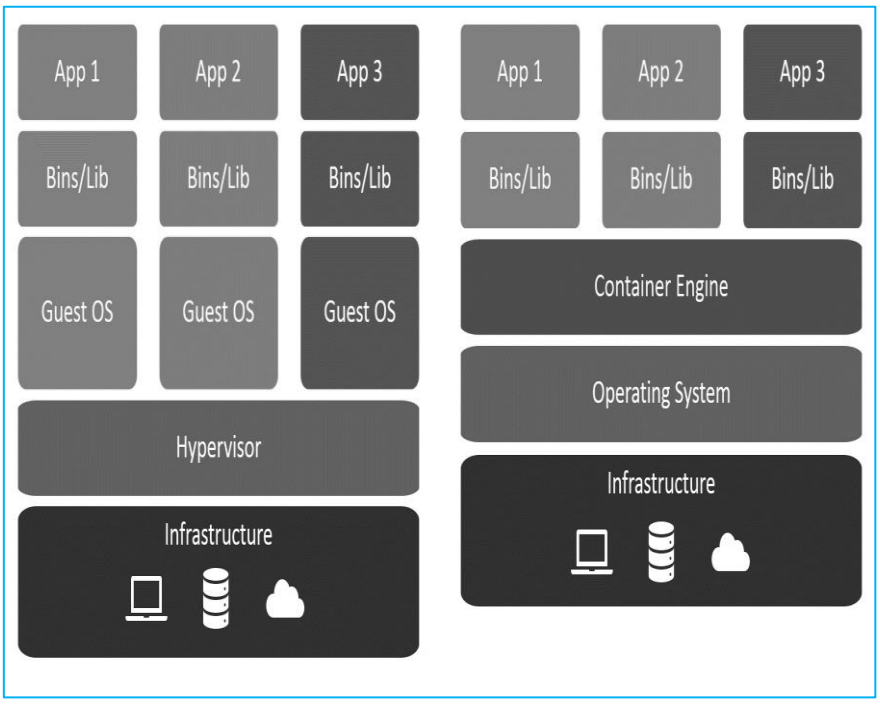


Figure 2-Architecture of Virtual Machine and Container [2]

The application deployment has significantly simplified using platforms like Docker and Kubernetes. In large-scale environments, container orchestration platforms such as Docker Swarm and Kubernetes are widely adopted to automate deployment, scaling, and monitoring of containerized microservices. Figure-3 shows the deployment model of containerized microservices in a cloud environment [3,4].

1.2 Container Scheduling for Microservices

The decision of where to place each container, known as the container scheduling problem. Container scheduling is very complex and multi-objective optimization problem in large scale cloud-based environments.

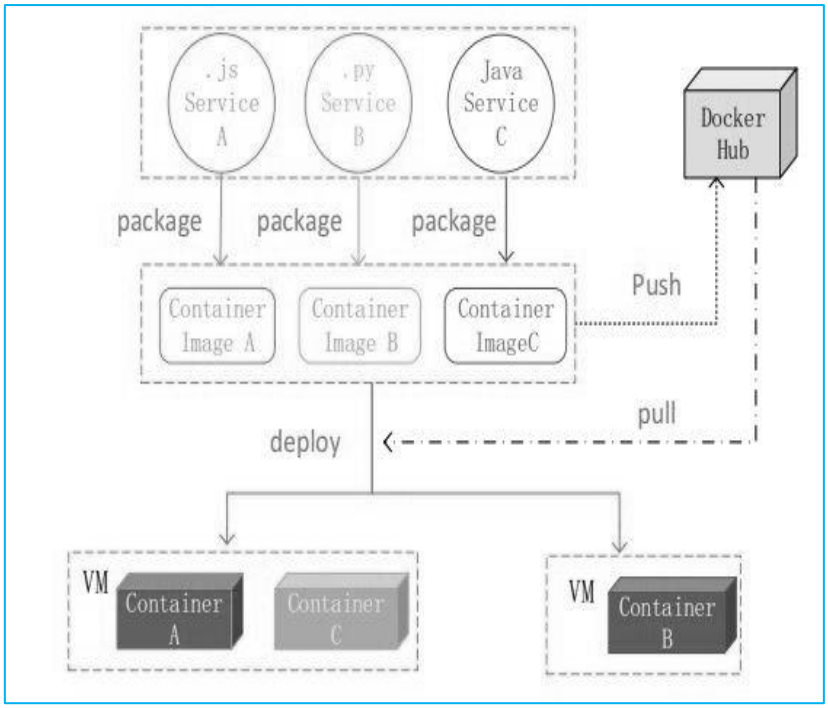


Figure 3-Deployment of containerized microservices [4]

Container scheduling is the process of selecting the most suitable node for deploying and executing containerised microservices. Efficient container scheduling is crucial because microservices generate workloads dynamically that constantly consume node resources such as CPU, memory, storage, and network bandwidth. Improper container scheduling may lead to resource imbalance and poor QoS. Hence, intelligent container scheduling mechanisms are required to ensure energy efficiency and improved QoS in container-based cloud environments [5,6].

The container orchestration systems like Docker Swarm commonly use scheduling strategies such as Random, Spread, and Binpack. Random scheduling deploys containers without considering system state, which may lead to inefficient resource utilization. Spread scheduling deploys containers evenly across nodes to achieve load balancing, whereas Binpack scheduling attempts to maximize resource utilization by packing containers into a smaller number of nodes. Although these scheduling approaches are mostly static and do not adapt dynamically to changing workload conditions and QoS needs. Table-1 presents a comparative analysis of various container orchestration platforms used for managing and deploying containerized microservices.

Orchestrator	Swarm	Kubernetes	Apache Mesos	Aurora	Marathon	YARN	Omega	Fuxi
Organization	Docker	Google	UC Berkeley	Twitter	Mesosphere	Apache	Google	Alibaba
Open Source	✓	✓	✓	✓	✓	✓	-	-
Technology of container	Docker	Docker	Mesos containers, Dokcer	Apache Mesos, Docker	Mesos containers, Docker	Linux cgroups-based, Docker	N/S	Linux cgroups-based
Pre-emption	-	-	-	✓	-	-	✓	-
Rescheduling	-	-	-	✓	-	-	✓	-
Scheduling constraints	Label and affinity-based	Label and affinity-based	N/A	Value and limit-based	Value and limit-based	Value and limit-based	N/S	Value based
Scalability	✓	✓	✓	✓	✓	✓	✓	✓
High Availability	✓	✓	-	-	-	-	✓	✓
High Utilization	✓	✓	-	-	-	-	✓	✓
High Throughput	✓	✓	-	-	-	-	✓	✓
Resource granularity	Fine-grained	Fine-grained	Fine-grained	Fine-grained	Fine-grained	coarse-grained	Fine-grained	Fine-grained
Application workload	Long running tasks	All	All	Long running tasks	Long running tasks	Batch tasks	All	Batch tasks

Table 1 –Classification of container orchestrators [3-10]

1.3 Machine Learning based approaches in Container Scheduling

Recent research has focused on intelligent and adaptive scheduling techniques based using Machine Learning (ML). Reinforcement Learning based schedulers, particularly Q-learning and Deep Q-Network approaches, enable dynamic scheduling decision by learning environmental interactions. These approaches have demonstrated improvements in efficient container scheduling for cloud and edge computing environments. Table-2 presents a comparative overview of machine learning-based approaches proposed for container scheduling in cloud computing environments.

Problem in scheduling of containerized applications	Machine Learning based solution	Advantage/ Limitations
How to analyze inter-dependency between micro services?	BO, GP, CNN, and LSTM	Improved prediction accuracy Ignorance of dependency updates
How to analyze dependency of tasks in containerized applications?	SVM	High Accuracy Implicit explanation of the time overhead and computational costs
How to achieve energy efficient resource scheduling for containers?	MDP, Q-learning, and SARSA	Cost saving Minimizing task execution time Limited scalability
How to reduce the communication delay when moving micro services to cloud container?	DNN, Q-learning	Minimizing task execution time Limited scalability
How to characterize workloads by modeling and predicting requests behavior and resource usage pattern?	LSTM, K-means++, ARIMA, Bi-LSTM, GRU, TSNNR, and GBR	Optimized Resource utilization Scheduling delays
How to reduce the communication delay when moving micro services to cloud container?	DNN, Q-learning	Minimizing task execution time Limited scalability
How to characterize workloads by modeling and predicting requests behavior and resource usage pattern?	LSTM, K-means++, ARIMA, Bi-LSTM, GRU, TSNNR, and GBR	Optimized Resource utilization Scheduling delays

Table 2-Machine learning based solutions in container scheduling [11-15]

1.4 Research Gap

However, several research gaps still exist in current state-of-the-art approaches. Many existing container scheduling techniques do not effectively integrate QoS metrics and energy efficiency. Some methods lack real-time workload and need adaptation capabilities, while others are evaluated only in simulation environments rather than real Docker Swarm clusters. Limited attention has been given to combining QoS-aware and energy-aware container scheduling within a unified framework. Therefore, there is a strong need for an intelligent, adaptive, and multi-criteria scheduling container mechanism which is capable of optimizing both application performance and energy efficiency in containerized microservice environments.

2.
Definition of the Problem

The increasing adoption of cloud computing and microservice-based applications has significantly changed modern software application development and deployment practices. Container scheduling is the process of selecting the most appropriate deployable node in a cluster for deploying a containerized microservice. The performance of a microservice is depends on how effectively containers are scheduled and balanced among available resources of the cluster. Existing container scheduling techniques particularly in Docker Swarm, such as Random, Spread, and Binpack scheduling. They mainly focus on basic resource allocation approaches. Random scheduling deploys containers without considering node conditions, which may lead to uneven resource utilization and degraded system performance. Spread scheduling deploys containers uniformly across nodes for load balancing but may result in underutilization of resources. Binpack scheduling tries to maximize resource usage by allocating containers on small number of nodes, however it may result node overloading and performance bottlenecks under dynamic workloads conditions.

Modern cloud container environments are highly dynamic in nature, where workload characteristics, resource utilization, and application need continuously change. Traditional container scheduling approaches are failed to adapt dynamic workload conditions. Furthermore, existing container scheduling methods generally consider limited parameters such as CPU and memory usage while ignoring QoS metrics. As a result, these approaches may lead to higher service delay and inefficient resource utilization in containerized microservice environments.

Recent research has explored machine learning and reinforcement learning based scheduling techniques for improving scheduling decisions in container orchestration systems. Although these approaches show better adaptability and decision-making capabilities, many research studies are experimented and evaluated in simulation environments and lack of implementation on real Docker Swarm clusters. Moreover, most reinforcement learning-based schedulers do not effectively integrate multi-criteria decision-making performance metrics with dynamic node ranking. Existing solutions also provide limited consideration for energy-aware scheduling and QoS optimization simultaneously.

Therefore, there is a need to develop an intelligent, adaptive, and multi-criteria container scheduling framework which is capable of dynamically selecting optimal deployable nodes for microservice deployment in Docker Swarm environments. The proposed research addresses this problem by designing a Multi-Criteria Scheduling of Containerized Microservices using Reinforcement Learning (MCSCM-RL) framework. The proposed system integrates DQN based reinforcement learning with multi-criteria node ranking mechanism to make container scheduling decisions based on CPU utilization, memory utilization, container count, startup time, and response time for microservices. The aim is to improve QoS and energy efficiency while reducing startup delay and response time in containerized microservice environments.

3.
Objective and Scope

3.1 Objectives of the Research

The primary objective of this research is to develop an intelligent and adaptive container scheduling framework for microservice-based applications deployed in a cloud environment using Docker Swarm. The research aims to improve QoS and energy efficiency by integrating Reinforcement Learning (RL) with multi-criteria decision-making for efficient scheduling of containerised microservices.

The specific objectives of the proposed research work are as follows:

- The first objective is to analyse the theoretical and practical aspects of microservice architecture, containerization, and container orchestration platforms such as Docker Swarm and Kubernetes. The study includes understanding of container deployment mechanisms and resource management in cloud-based container environments.
- The research aims to investigate existing container scheduling strategies such as Random, Spread, and Binpack scheduling used in Docker Swarm. Their performance will be evaluated based on Node and Application parameters.
- The objective is to identify the limits of current container scheduling methods in handling dynamic workload conditions and QoS-aware microservice deployment requirements.
- The research aims to develop a node ranking using QoS and resource parameters. The aim is to select optimal nodes dynamically for efficient deployment of containerized microservices.
- The primary objective is to implement a Reinforcement Learning based scheduling model using DQN. The RL agent learns optimal scheduling decisions dynamically from system node states and application parameters to improve scheduling efficiency in Docker Swarm clusters.

- The proposed work aims to integrate QoS parameters and Resource utilization during deployment of containerized microservices. The custom scheduler MCSCM-RL will try to improve startup time, response time, and energy efficiency.
- The research objective also includes implement proposed custom schedular on a real Docker Swarm cluster of three nodes and comparing its performance with existing container scheduling strategies.
- The final objective is to perform comparative performance analysis between the proposed scheduler and traditional docker swarm container scheduling methods such as Random, Spread, and Binpack under clean and cumulative load conditions.

3.2 Scope of the Research

The scope of this research is focused on intelligent scheduling of containerized microservices in cloud environments using reinforcement learning and multi-criteria decision-making mechanisms. This research primarily aims to improve resource utilization and application QoS in container-based cloud systems.

The scope of the proposed work includes the following aspects:

- The research focuses on deployment and management of containerized microservices. The microservices are deployed in a docker cluster environment.
- The implementation and experimentation are limited to the Docker Swarm orchestration platform. A Docker Swarm cluster of 3 nodes consisting of a manager and two worker nodes is used for container deployment.
- The proposed scheduler considers multiple node level and application level QoS parameters for optimal node selection for deployment of containerised microservice deployment.
- The research scope includes the application of Reinforcement Learning, using DQN, for adaptive and intelligent container scheduling decisions in dynamic workload conditions.
- The proposed framework focuses on improving application QoS and energy consumption through intelligent container deployment.
- The research includes comparative analysis of the proposed scheduling approach with existing container scheduling techniques such as Random, Spread, and Binpack scheduling under clean and cumulative load.

- The research does not focus on large-scale public cloud deployment or heterogeneous hardware environments. Although, the research focuses on Docker Swarm, the proposed work can be extended in the future to Kubernetes and public cloud platforms for large-scale intelligent scheduling applications.

4.
Original Contribution by the Thesis

The proposed thesis contributes to the field of scheduling of containerized microservice by developing a novel container scheduler framework for Docker Swarm environments. The research introduces an energy-aware and QoS-aware scheduling approach that integrates Reinforcement Learning with multi-criteria node ranking mechanism for efficient scheduling of containerized microservices in cloud environment. The major original contributions of the thesis are summarized as follows:

1. The thesis proposes a novel container scheduling framework that integrates DQN based Reinforcement Learning with multi-criteria decision-making mechanism. The proposed approach dynamically learns optimal container scheduling decision unlike traditional container schedulers that rely on static decisions.

2. A significant contribution of this research is the integration of relative closeness-based node ranking into the reinforcement learning model. The proposed scheduler evaluates nodes dynamically using multiple parameters and incorporates ranking information into the state representation of the RL agent to improve scheduling accuracy.

3.The proposed research work introduces energy-aware and QoS-aware container scheduling by considering performance metrics such as:

- *Startup time*
- *Response time*
- *CPU utilization*
- *Memory utilization*
- *Container count*

The proposed scheduler aims to minimize service latency and improving application performance in microservice based systems.

4. An existing research and studies rely mainly on simulation environments, where the proposed research work implements and validates the scheduling framework on a real Docker Swarm cluster of three nodes. The scheduler dynamically calculates node rankings and adapts scheduling decisions based on continuously changing resource conditions.

5. The thesis provides an extensive comparative analysis between the proposed MCSCM-RL scheduler and existing Docker Swarm scheduling strategies such as

Random, Spread, Binpack, and MCSCM (custom scheduler). The evaluation is performed using real deployment experiments and QoS metrics to present improvements in resource utilization, startup time, response time, and energy efficiency.

6. The research presents a unified framework that combines QoS-aware scheduling and Energy-aware optimization. It offers Reinforcement learning-based decision-making with real-time node monitoring. This integrated approach provides a more efficient and scalable solution for container orchestration in distributed cloud environments.

7. The proposed methodology can be extended to large-scale cloud-native platforms, Kubernetes, and public cloud computing environments.

5.
Research Methodology

The proposed research focuses on the development of an intelligent and adaptive container scheduling framework for microservice based applications deployed in a cloud environment using Docker Swarm. The research methodology consists of system environment, development of container scheduling strategies, resource monitoring and node ranking, reinforcement learning model development, experimental deployment, performance evaluation, and comparative analysis. The research methodology aims to improve Quality of Service (QoS), resource utilization, and energy efficiency in containerized microservice environments.

5.1 System Environment

A real distributed container orchestration environment is created using Docker and Docker Swarm. The experimental environment consists of:

- *One swarm manager node*
- *Two swarm worker nodes*
- *Ubuntu based Virtual Machines hosted on a Windows system*

The Docker Swarm cluster is configured to support deployment and monitoring of containerized microservices. Different microservice applications are containerized and deployed using Docker images hosted on docker hub. Figure 4 presents experiment environment used in this research work.

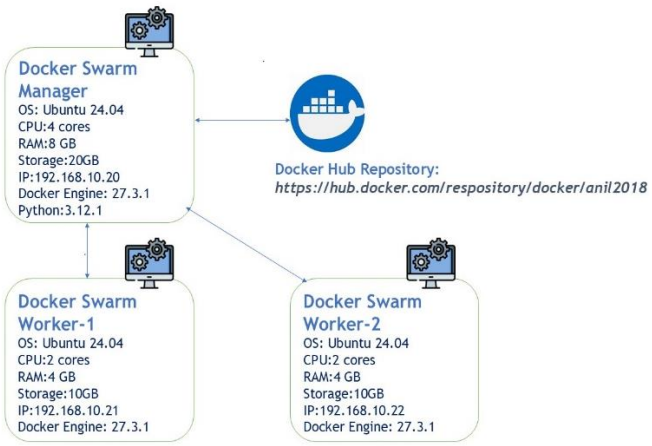


Figure 4- Experiment Environment

5.2 Development of existing docker swarm container scheduling policies

To establish comparisons, traditional Docker Swarm scheduling policies are implemented and evaluated. These include:

- *Random Scheduling*
- *Spread Scheduling*
- *Binpack Scheduling*

Each scheduler is analysed using parameters such as:

- *CPU utilization*
- *Memory utilization*
- *Startup time*
- *Response time*
- *Container count*

The results obtained from these existing schedulers are used for comparison with the proposed scheduler.

5.3 Resource Monitoring and Metric Collection

A monitoring framework is developed to collect real-time node-level and application-level metrics from the Docker Swarm cluster. The collected metrics are periodically updated and used for scheduling decision and performance evaluation. The real time metric collection helps the proposed scheduler to understand the current state of the swarm cluster and enables adaptive scheduling decisions.

5.4 Multi-Criteria Node Ranking

A multi-criteria decision-making mechanism is developed to rank swarm nodes in the Docker Swarm cluster. The node ranking mechanism considers QoS and resource parameters to calculate swarm node priority. The ranking mechanism identifies the most suitable swarm node for container deployment under dynamic workload conditions. The node ranking framework enables balanced workload distribution and efficient resource utilization across the swarm cluster.

5.5 Reinforcement Learning-based Scheduling Model

The main contribution of this research is the development of a Reinforcement Learning based intelligent scheduler using DQN.The RL environment consists of State Space, Action Space and Reward Function.

State Representation

The state vector includes performance metrics.

Action Space

The action represents the selection of one node from the available Docker Swarm nodes for deploying a microservice container.

Reward Function

The reward function is designed to improve QoS and energy efficiency. The DQN model continuously learns from environmental interactions and updates scheduling policies dynamically to achieve optimal deployment decisions.

5.6 Proposed MCSCM-RL Scheduling Framework

The proposed framework, named MCSCM-RL, integrates:

- *Dynamic node monitoring*
- *Multi-criteria ranking*
- *QoS-aware scheduling*
- *Energy-aware optimization*
- *Reinforcement Learning-based decision-making*

The scheduling workflow operates as follows:

1. *Collect real-time node metrics*
2. *Calculate node ranking values*
3. *Generate current system state*
4. *RL agent selects optimal node*
5. *Deploy container on selected node*
6. *Measure startup and response time*
7. *Update reward and train DQN model*
8. *Repeat scheduling cycle dynamically*

This adaptive container scheduling approach allows the system to continuously optimize deployment decisions according to dynamic workload conditions.

5.7 Experimental Evaluation

The proposed work is evaluated using real deployment scenarios in the Docker Swarm cluster. Performance analysis is carried out using performance metrics used in the research. Multiple deployment experiments are conducted for clean/cumulative workloads and scheduling strategies. Figure 5 shows workflow of the proposed research work.

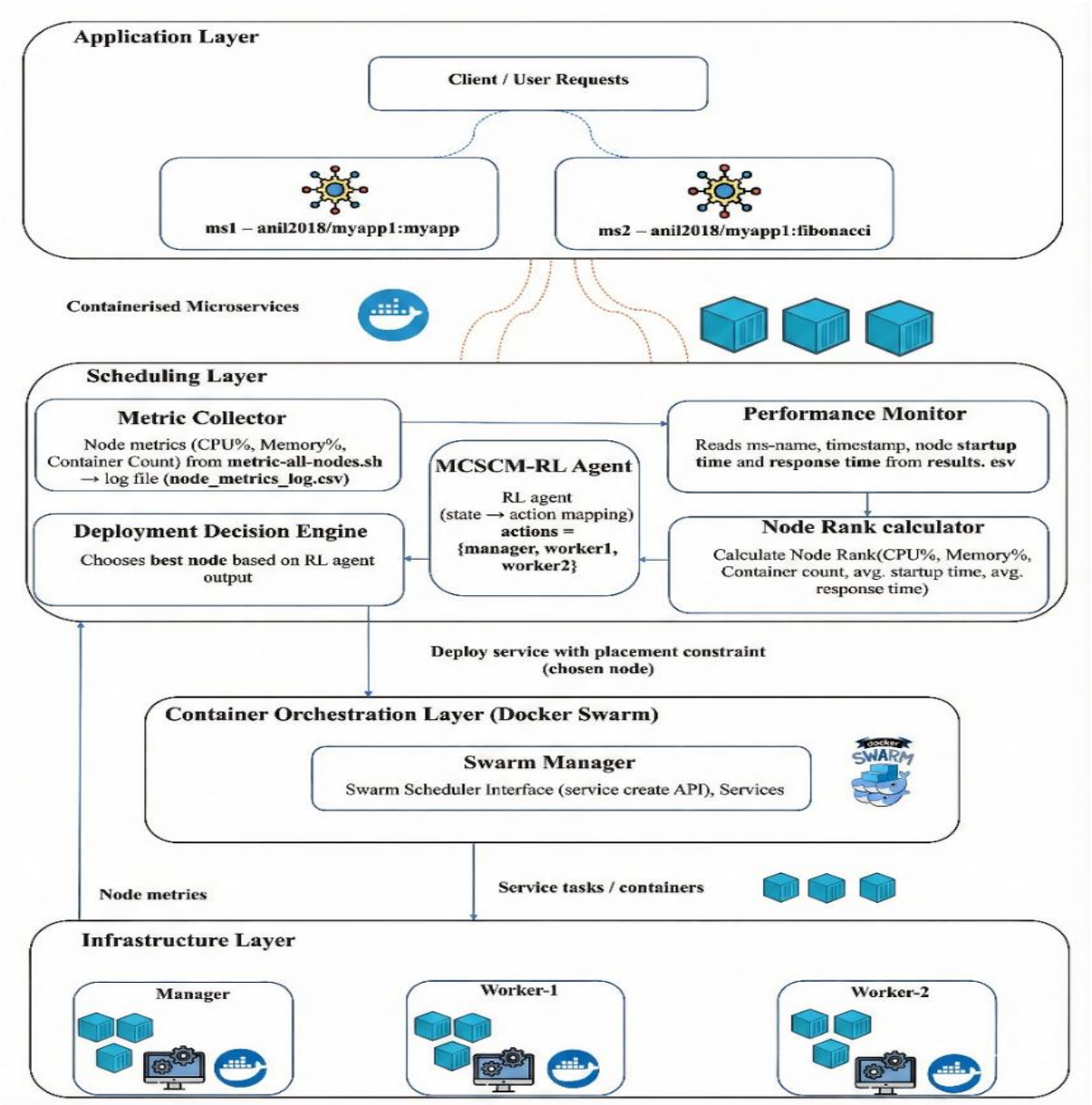


Figure 5- Workflow of Proposed research

The proposed MCSCM-RL scheduler is compared with:

- *Random Scheduler*
- *Spread Scheduler*
- *Binpack Scheduler*
- *MCSCM (custom)*

5.8 Application Model

The application model used in the proposed research is based on a microservice-oriented architecture deployed in a Docker Swarm environment. The model is designed to evaluate proposed MCSCM-RL framework under different workload conditions. The application

environment consists of lightweight and CPU-intensive microservices deployed as Docker containers across multiple Swarm nodes. Figure 6 shows microservices hosted on my docker hub repository.

The experimental setup includes two microservices that simulate different workload characteristics commonly observed in cloud-native applications. These microservices are deployed using Docker images hosted on Docker Hub and are dynamically scheduled on swarm cluster by different scheduling strategies during experimentation.

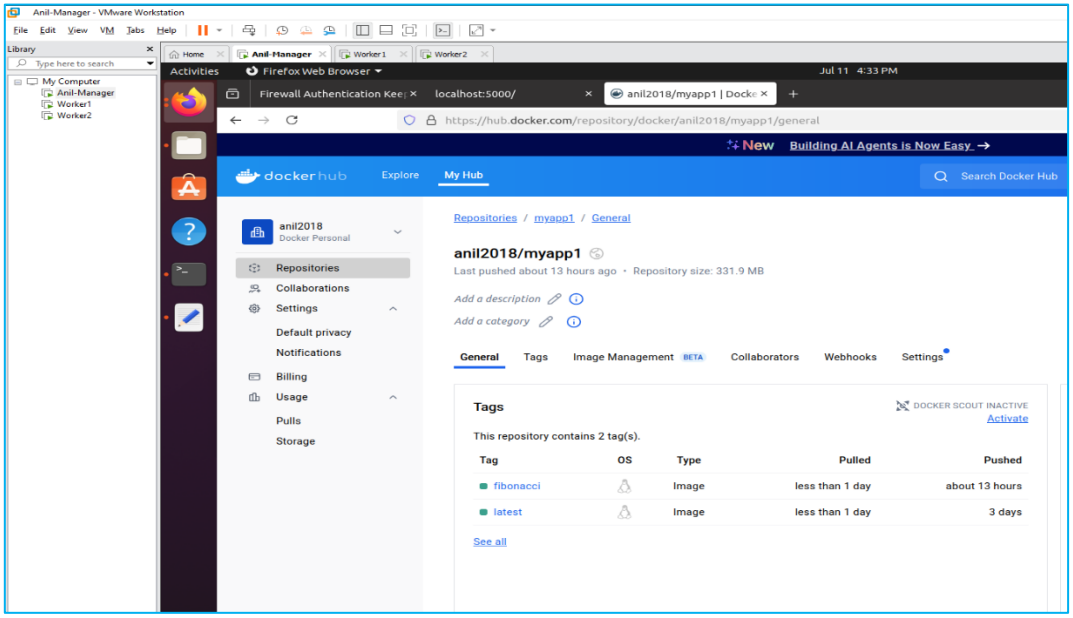


Figure 6- Microservices on my docker hub

The first microservice, **ms1**, uses the Docker image **anil2018/myapp1:myapp** and acts as a stateless request-processing service. This microservice simulates lightweight application behavior similar to REST API-based services. The service processes client requests independently without maintaining application state, making it suitable for evaluating response time, startup latency, and scheduling overhead in microservice environments.

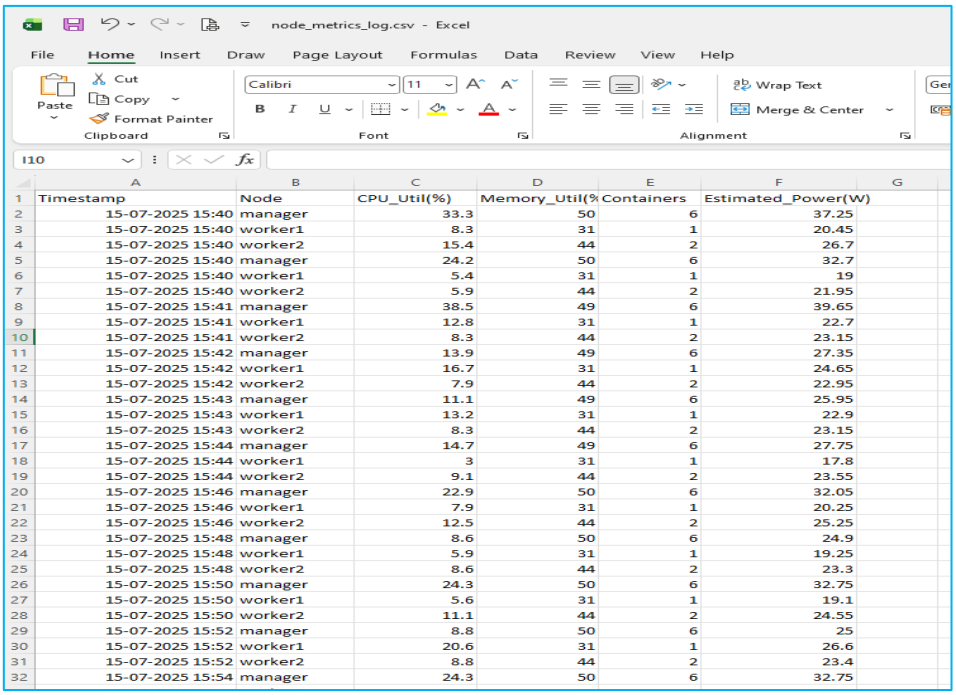
The second microservice, **ms2**, uses the Docker image **anil2018/myapp1:fibonacci** and represents a CPU-intensive computational workload. This microservice performs Fibonacci number calculations and is used as a synthetic workload generator to create high CPU utilization conditions within the Docker Swarm cluster. The Fibonacci service helps evaluate scheduler behavior under resource-intensive conditions and enables analysis of CPU utilization, resource balancing, and QoS-aware scheduling performance.

The application model supports deployment of multiple instances of these containerized microservices to simulate dynamic cloud workloads. During experimentation, services are

deployed repeatedly under both clean load and cumulative load conditions. In clean load deployment, previously running containers are removed before deploying new services, whereas cumulative deployment continuously increases swarm cluster workload by retaining active containers. This enables evaluation of scheduler adaptability, scalability, and resource management under varying workload conditions.

5.9 Result

The proposed MCSCM-RL scheduler was experimented and evaluated on a real Docker Swarm cluster consisting of one manager node and two worker nodes. The performance of the proposed scheduler was compared with existing scheduling strategies including Random Scheduling, Spread Scheduling, Binpack Scheduling and MCSCM. Figure-7 and Figure-8 presents node-level metrics collection at regular interval and application metrics collection after each deployment respectively.



	A	B	C	D	E	F	G
1	Timestamp	Node	CPU_Util(%)	Memory_Util(%	Containers	Estimated_Power(W)	
2	15-07-2025 15:40	manager	33.3	50	6	37.25	
3	15-07-2025 15:40	worker1	8.3	31	1	20.45	
4	15-07-2025 15:40	worker2	15.4	44	2	26.7	
5	15-07-2025 15:40	manager	24.2	50	6	32.7	
6	15-07-2025 15:40	worker1	5.4	31	1	19	
7	15-07-2025 15:40	worker2	5.9	44	2	21.95	
8	15-07-2025 15:41	manager	38.5	49	6	39.65	
9	15-07-2025 15:41	worker1	12.8	31	1	22.7	
10	15-07-2025 15:41	worker2	8.3	44	2	23.15	
11	15-07-2025 15:42	manager	13.9	49	6	27.35	
12	15-07-2025 15:42	worker1	16.7	31	1	24.65	
13	15-07-2025 15:42	worker2	7.9	44	2	22.95	
14	15-07-2025 15:43	manager	11.1	49	6	25.95	
15	15-07-2025 15:43	worker1	13.2	31	1	22.9	
16	15-07-2025 15:43	worker2	8.3	44	2	23.15	
17	15-07-2025 15:44	manager	14.7	49	6	27.75	
18	15-07-2025 15:44	worker1	3	31	1	17.8	
19	15-07-2025 15:44	worker2	9.1	44	2	23.55	
20	15-07-2025 15:46	manager	22.9	50	6	32.05	
21	15-07-2025 15:46	worker1	7.9	31	1	20.25	
22	15-07-2025 15:46	worker2	12.5	44	2	25.25	
23	15-07-2025 15:48	manager	8.6	50	6	24.9	
24	15-07-2025 15:48	worker1	5.9	31	1	19.25	
25	15-07-2025 15:48	worker2	8.6	44	2	23.3	
26	15-07-2025 15:50	manager	24.3	50	6	32.75	
27	15-07-2025 15:50	worker1	5.6	31	1	19.1	
28	15-07-2025 15:50	worker2	11.1	44	2	24.55	
29	15-07-2025 15:52	manager	8.8	50	6	25	
30	15-07-2025 15:52	worker1	20.6	31	1	26.6	
31	15-07-2025 15:52	worker2	8.8	44	2	23.4	
32	15-07-2025 15:54	manager	24.3	50	6	32.75	

Figure 7-Node metric collection

Figure-7 illustrates the process of application metric collection performed after deployment of each container in the proposed scheduling framework. The system continuously monitors both node-level and application-level performance parameters. The collected metrics are used to evaluate the current state of the cluster and support intelligent scheduling decisions for subsequent container deployments.

The experimental evaluation was conducted under two deployment scenarios:

1. **Clean Load Deployment**
2. **Cumulative Load Deployment**

The experiments were performed using containerized microservice applications deployed repeatedly under varying workload conditions. Performance evaluation was carried out using QoS and resource utilization metrics.

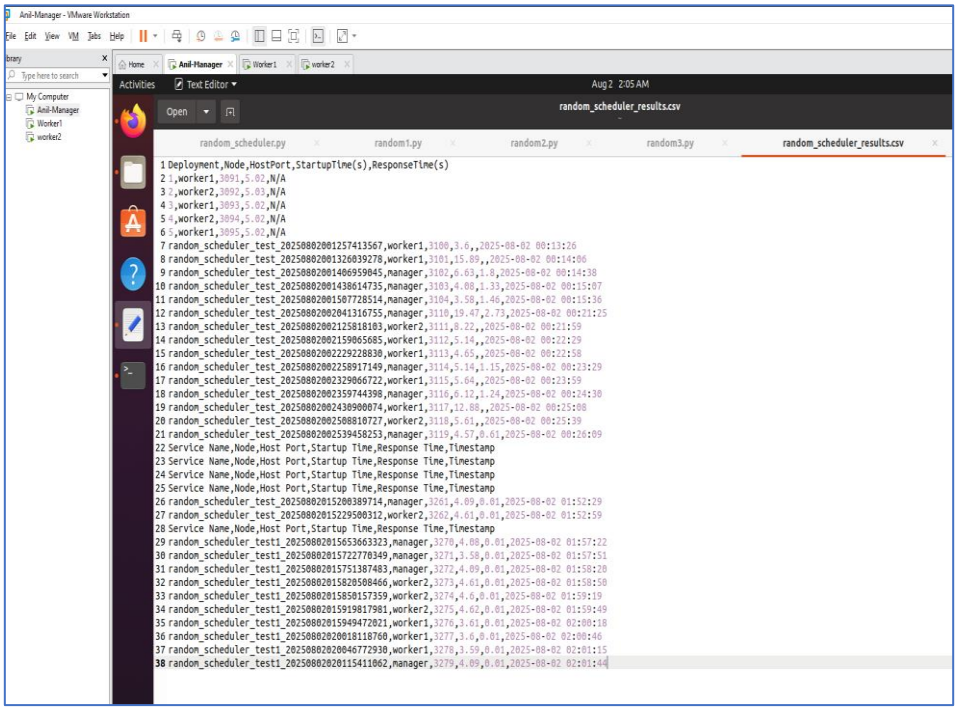


Figure 8- Application metrics collection after each deployment

Clean Load Deployment Analysis

In the clean load deployment scenario, all existing microservices and containers were removed from node before each deployment. This ensured that every container scheduling policy was evaluated under identical initial swarm cluster conditions without interference from previously deployed workloads.

The experimental results demonstrated that traditional container scheduling techniques such as Random, Spread, Binpack, and MCSCM showed variations in startup time and response time due to their static decision-making mechanisms. Random scheduling frequently selected nodes without considering current resource conditions, which leads to inconsistent deployment performance. Spread scheduling distributed workloads uniformly across nodes, which improved load balancing but did not always optimize resource

utilization. Binpack scheduling achieved better resource packing but occasionally caused localized resource congestion on selected nodes.

The proposed scheduler achieved best performance in clean load deployment experiments due to its adaptive and intelligent scheduling capability. Figure-9 shows average resource utilization for each container scheduling policy under clean load scenario. As a result, the proposed scheduler reduced average startup delay and improved response time compared to existing scheduling policies.

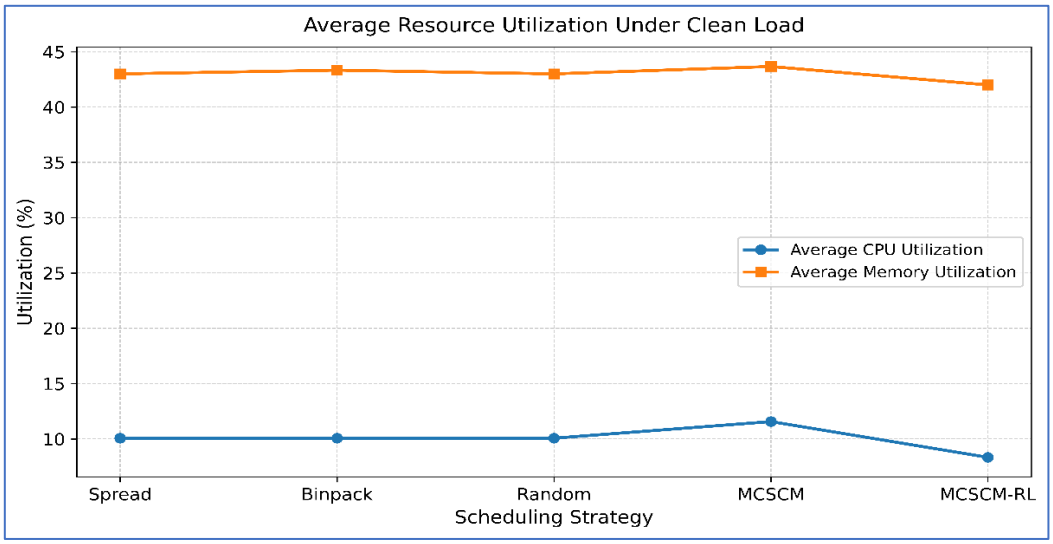


Figure 9-Average Resource Utilization under clean load

The results further indicated that integrating multi-criteria node ranking with reinforcement learning significantly improves scheduling decisions even in low-load conditions.

Cumulative Load Deployment Analysis

In the cumulative load deployment scenario, microservices were continuously deployed without removing previously running containers. This experiment simulated realistic cloud environments where workloads dynamically increase over the time and node resources progressively utilized. The cumulative deployment was conducted to evaluate the scalability and resource-awareness of the container scheduling policies under increasing swarm cluster workload.

Under cumulative workload scenario, traditional container scheduling strategies showed significant performance degradation. Random scheduling resulted in uneven workload distribution and frequent node overloading. Spread scheduling maintained balanced

distribution initially but failed to optimize resource usage efficiently as cluster utilization increased. Binpack scheduling densely allocated containers on fewer nodes, which increased resource contention and degraded response time under heavy workload situations. Figure-10 presents average resource utilization for each scheduling mechanism under cumulative load scenario.

The proposed scheduler demonstrated improved adaptability and robustness in cumulative deployment scenarios. The reinforcement learning agent continuously monitored node states and dynamically updated scheduling policies based on changing swarm cluster conditions. The multi-criteria ranking mechanism enabled intelligent workload distribution by considering both resource utilization and QoS parameters during swarm node selection for the deployment.

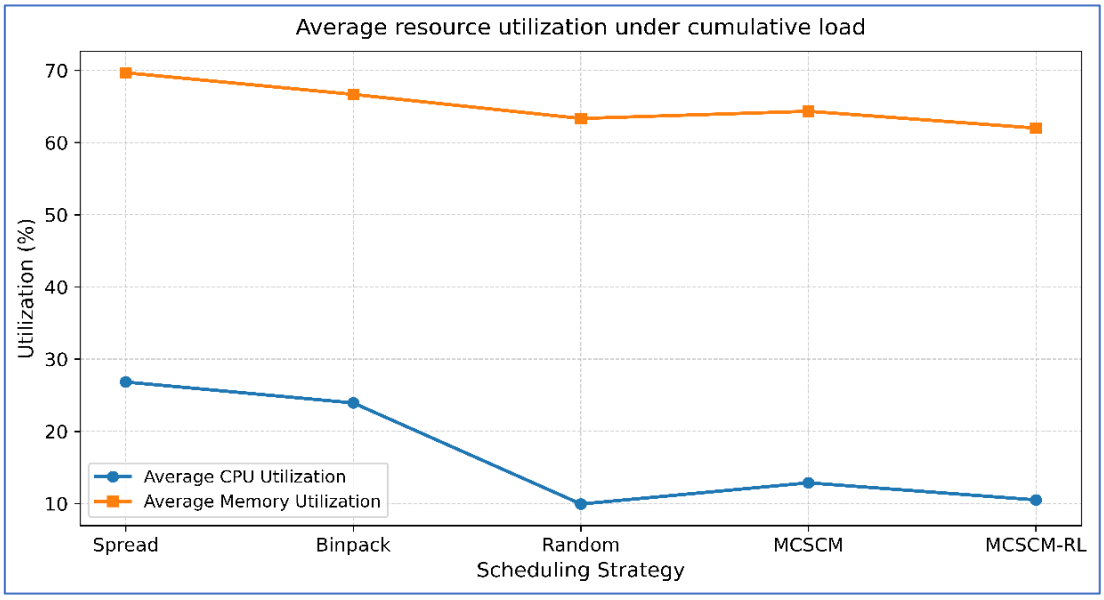


Figure 9-Average Resource Utilization under cumulative load

The cumulative deployment results demonstrated the learning capability of the DQN-based scheduler in adapting to dynamic workload conditions. As the number of deployed containers increased, the RL agent learned efficient scheduling patterns that minimized resource contention and improved system stability.

Comparative Validation of Scheduling Strategies

The comparison between Random, Spread, Binpack, and the proposed scheduler validated the effectiveness of the proposed framework. Figure -11 shows comparative analysis of Clean and Cumulative Deployment Scenarios for all container scheduling policies.

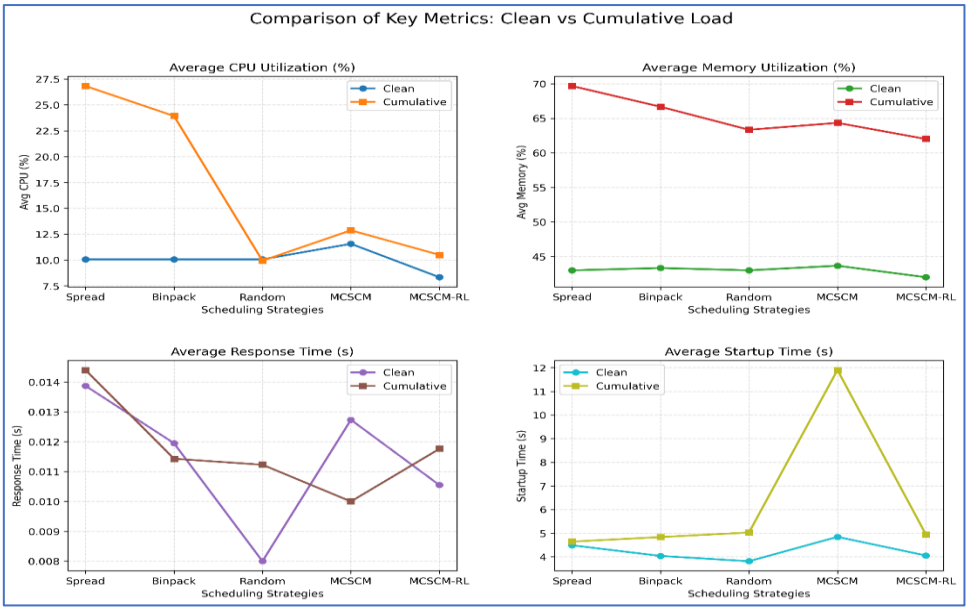


Figure 11-Comparative Analysis of Clean and Cumulative Deployment Scenarios

The comparative analysis showed that:

- Random scheduling produced inconsistent performance and poor resource optimization.
- Spread scheduling improved load balancing but lacked QoS optimization.
- Binpack scheduling improved resource packing but increased node congestion under heavy load.
- The proposed scheduler achieved better QoS, balanced resource utilization, and adaptive scheduling performance.

6.
Achievements with respect to objectives

The proposed research successfully achieved the main objectives of intelligent scheduling of containerized microservices in cloud environments using Reinforcement Learning and multi-criteria decision-making method.

The research successfully analysed the theoretical and practical aspects of microservice architecture, containerization, and container orchestration technologies. An extensive study of Docker and Docker Swarm was conducted to understand container deployment, resource management, and container scheduling strategies in cloud environments.

Implementation of Docker Swarm-based Experiment Setup

A real multi-node Docker Swarm cluster environment was successfully developed using Ubuntu virtual machines on windows-based host system. Docker swarm cluster consisting of:

- *One swarm manager node*
- *Two swarm worker nodes*

The practical validation of the proposed scheduler under realistic cloud-native deployment conditions was implemented.

Development and Evaluation of Existing Container Scheduling Strategies

Traditional Docker Swarm scheduling approaches including:

- *Random Scheduling*
- *Spread Scheduling*
- *Binpack Scheduling and*

  MCSCM Scheduling(custom) were successfully implemented and evaluated. Experimental analysis identified limitations of these scheduling approaches in terms of:

- *Resource imbalance*
- *Higher startup time*
- *Increased response time*
- *Poor adaptability under dynamic workloads*
- *Node overloading in cumulative deployment scenarios*

In order to validate the suggested scheduling framework, the baseline performance was determined for comparative evaluation.

Development of Multi-Criteria Node Ranking System

A multi-criteria decision-making dynamic node ranking framework was effectively developed. The node-level and application-level metrics were considered by ranking system to identify optimal container deployment nodes dynamically.

The ranking mechanism improved:

- *Load balancing*
- *Resource utilization*
- *QoS-aware scheduling decisions*

Design and Implementation of Reinforcement Learning-based Scheduler

A major achievement of the research is the successful development of a DQN based reinforcement learning container scheduler in Docker Swarm environments.

The Proposed scheduler:

- *Dynamically learned optimal scheduling policies*
- *Adapted to changing workload conditions*
- *Selected nodes intelligently using current state of system*
- *Integrated multi-criteria ranking into state representation*

The RL model successfully improved scheduling efficiency compared to static container scheduling policies.

Integration of Energy-aware and QoS-aware Scheduling

The proposed scheduling framework successfully integrated QoS-aware and energy-aware scheduling mechanisms. The scheduler considered both resource efficiency and application performance during container deployment decisions.

The implemented framework achieved:

- *Reduced startup time*
- *Improved response time*
- *Balanced workload distribution*
- *Efficient resource utilization*

The research demonstrated that integrating QoS and energy optimization greatly enhance the overall system performance.

Successful Evaluation under Clean and Cumulative Workload scenarios

The proposed scheduler was experimentally evaluated under:

- *Clean load deployment*
- *Cumulative load deployment*

The results demonstrated that the proposed scheduler maintained better performance stability and adaptability under increasing workloads compared to existing container schedulers.

The research provides a practical and scalable framework for intelligent container scheduling in cloud-native environments.

The proposed research methodology can be extended to:

- *Large-scale cloud platforms*
- *Public cloud computing environments*
- *Kubernetes based container orchestration system*

7.

Conclusion

In cloud computing environments, the importance of efficient container orchestration and intelligent resource management are crucial due to rapid growth of cloud-based applications and microservice architectures. Traditional container scheduling strategies used in Docker Swarm, such as Random, Spread, and Binpack scheduling, offer basic workload distribution mechanisms but have limitations when it comes to handle dynamic workloads, QoS requirements, and energy efficiency. Under constantly changing workload conditions, these container scheduling approaches often results in resource imbalance, increased response time, higher startup delay, and inefficient resource utilization.

By developing a Multi-Criteria Scheduling of Containerized Microservices using Reinforcement Learning (MCSCM-RL) framework for Docker Swarm environments the proposed research successfully has addressed these issues. The proposed approach combines multi-criteria node ranking with Deep Q-Network based reinforcement learning to provide adaptive and intelligent scheduling decisions. The scheduler dynamically evaluates cluster nodes using node-level and application-level QoS parameters such as CPU utilization, memory utilization, container count, startup time, and response time.

Experimental evaluation was conducted under both clean deployment and cumulative load deployment scenarios using lightweight and CPU-intensive microservice. The experimental results demonstrated that the proposed MCSCM-RL scheduler outperformed all existing container scheduling strategies.

The integration of QoS-aware and energy-aware optimization mechanisms enabled the proposed framework to maintain stable performance even under cumulative workloads. The research also validated that combining reinforcement learning with multi-criteria decision-making significantly enhances container scheduling efficiency in distributed microservice environments. All things considered, the proposed research offers an intelligent, scalable, and adaptive scheduling framework appropriate for current cloud-native systems. The work contributes toward efficient container orchestration, QoS optimization, and energy-efficient resource management in cloud computing environments.

8.
Future Work

The proposed MCSCM-RL scheduling framework can be further extended to support large-scale and real-world cloud computing environments. To evaluate its performance in enterprise-level container orchestration systems the proposed scheduling framework may be experimented on Kubernetes in future. The scheduling framework can also be deployed on public cloud computing platforms and fog computing environments where workload distribution and latency optimization are very critical. Additionally, advanced Deep Reinforcement Learning techniques and Actor-Critic models can be explored to improve learning stability and scheduling accuracy under highly dynamic workload situations.

Further enhancements may include the integration of auto-scaling, fault-tolerance and security-aware container scheduling mechanisms to improve reliability microservice environments. These future enhancements can further strengthen the effectiveness of intelligent container scheduling for next-generation cloud systems.

References

[1] A Practical Guide to Choosing between Docker Containers and VMs, January 16,2020 [Online] .Available: https://www.weave.works/blog/a-practical-guide-to-choosing-between-docker-containers-and-vms , [Accessed : January 11,2021]

[2] Ye Wu ,Haopeng Chen,"ABP Scheduler :Speeding up service spread in Docker swarm" , 2017 IEEE International journal on Parallel and distributed Processing with Applications", 0-7695-6329-5/17 2017 IEEE

[3] A Practical Guide to Choosing between Docker Containers and VMs, January 16,2020 [Online] .Available: https://www.weave.works/blog/a-practical-guide-to-choosing-between-docker-containers-and-vms , [Accessed : January 11,2021]

[4] Vindeep Singh,Sateesh K Peddoju,"Container-based Microservice Architecture for Cloud Applications" , IEEE International Conference on Computing, Communication and Automation (ICCCA2017), ISBN: 978-1-5090-6471-7/17/$31.00 ©2017 IEEE

[5] Guozhi Liu, Bi Huang, Zhihong Liang, "Microservices: architecture, container, and challenges", 978-1-7281-8915-4/20/$31.00 ©2020 IEEE DOI 10.1109/QRS-C51114.2020.00107

[6] Sanath Mugeraya, Kailas Devadkar," Dynamic Task Scheduling and Resource Allocation for Micro services in Cloud", Journal of Physics: 2325 (2022) 012052 doi:10.1088/1742-6596/2325/1/012052

[7] Abdul Saboor, Mohd Fadzil Hassan, Rehan Akbar, Syed Nasir Mehmood Shah," Containerized Microservices Orchestration and Provisioning in Cloud Computing: A Conceptual Framework and Future Perspectives",MDPI, applied Appl. Sci. 2022, 12, 5793. https://doi.org/10.3390/app12125793

[8] Mehmet Söylemez , Bedir Tekinerdogan ,Ayça Kolukısa Tarhan," Challenges and Solution Directions of Microservice",MDPI, Appl. Sci. 2022, 12, 5507. https://doi.org/10.3390/app12115507 Architectures: A Systematic Literature Review

[9] Devki Nandan Jha,Saurabh Garg, Prem Prakash Jayaraman,Rajkumar Buyya,Rajiv Ranjan Celest, " A Holistic Evaluation of Docker Containers for Interfering Microservices" , 2018 IEEE International Conference on Services Computing,2474-2473/18/$31.00 ©2018 IEEEDOI 10.1109/SCC.2018.00012

[10] Chris Richardson of Eventuate,Introduction to microservices[Online] .Available: https://www.nginx.com/blog/introduction-to-microservices/ [Accessed : July 08,2022]

[11] Container-Based Microservice Scheduling Using Reinforcement Learning in Distributed Cloud Computing, Marcos Carvalho,IEEE 2024-Latin-Americal Conference

[12] A deep reinforcement learning-based optimization approach for containerized microservice scheduling in Hybrid Fog/Cloud environments, Ameni Kallel,Journal of engineering applications of Artificial Intelligence,December-2024

[13] Automated Cloud Provisioning on AWS using Deep Reinforcement Learning, Zhiguang Wang, Chul Gwon,http://doi.org/10.48550/arXiv:1709.04305

[14] A Reinforcement Learning Approach, Hanshuai Cui1, 2 , Zhiqing Tang1 , Jiong Lou3, 1 , and Weijia Jia] http://dx.doi.org/10.48550/arXiv.2307.12121

[15] An efficient deep reinforcement learning based task scheduler in cloud-fog environment, Prashanth Choppara1 , Sudheer Mangalampalli, Springer-Cluster Computing (2025) 28:67

[16] DRL4HFC: Deep Reinforcement Learning for Container-Based Scheduling in Hybrid Fog/Cloud System, Ameni Kallel1 a , Molka Rekik, ICAART 2024 - 16th International Conference on Agents and Artificial Intelligence

[17] Enhancing Kubernetes Automated Scheduling with Deep Learning and Reinforcement Techniques for Large-Scale Cloud Computing Optimization, Zheng Xu1* Yulu Gong1,Yanlin Zhou2,Qiaozhi Bao3,Wenpin Qian4

[18] TOMMASO PRATURLON," Intelligent autoscaling in Kubernetes: the impact of container performance indicators in model-free DRL methods",Thesis, Stockholm, Sweden, 2023

[19] Yi Wei ,Daniel Kudenko, Shijun Liu ,Li Pan2018," A Reinforcement Learning Based Auto-Scaling Approach for SaaS Providers in Dynamic Cloud Environment", Hindawi,Mathematical Problems in Engineering,Volume 2019, Article ID 5080647, 11 pages,https://doi.org/10.1155/2019/508064

[20] Haoyu Bai, Minxian Xu, Senior Member, IEEE, Kejiang Ye, Senior Member, IEEE, Rajkumar Buyya, Fellow, IEEE, Chengzhong Xu, Fellow, IEEE," DRPC: Distributed Reinforcement Learning Approach for Scalable Resource Provisioning in Container-based Clusters", EEE TRANSACTIONS ON SERVICE COMPUTING, :2407.10169v1 [cs.DC] 14 Jul 2024

[21] Jose Santos,Mattia Zaccarini,Filippo Poltronieri,Mauro Tortonesi, 'Efficient Microservice Deployment in Kubernetes Multi-Clusters through Reinforcement Learning'

[22] Tom Danino 1,Yehuda Ben-Shimol and Shlomo Greenberg," Container Allocation in Cloud Environment Using Multi-Agent Deep Reinforcement Learning", Electronics 2023, 12, 2614. https://doi.org/10.3390/electronics12122614,MDPI

[23] Peng Liu, Weisen Zhao*, Baoliang Zhang and Jing Wang, "Hybrid Elastic Scaling Strategy for Container Cloud based on Load Prediction and Reinforcement Learning" Journal of Physics: Conference Series 2732 (2024) 012014 IOP Publishing doi:10.1088/1742-6596/2732/1/012014

[24] Fabiana Rossi, Matteo Nardelli, Valeria Cardellini,” Horizontal and Vertical Scaling of Container-based Applications using Reinforcement Learning”, 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), 2159-6190/19/$31.00 ©2019 IEEE DOI 10.1109/CLOUD.2019.00061

[25] Zhiheng Zhong, Minxian Xu, Maria Alejandra Rodriguez, Chengzhong Xu, and Rajkumar Buyya. 2022. Machine Learning-based Orchestration of Containers: A Taxonomy and Future Directions. ACM Comput. Surv.54, 10s, Article 217 (September 2022), https://doi.org/10.1145/3510415

[26] Joaquín Entrialgo · Manuel García,” Joint Autoscaling of Containers and Virtual Machines for Cost Optimization in Container Clusters”, Journal of Grid Computing (2024) 22:17

[27] ZHIHENG ZHONG and RAJKUMAR BUYYA, “A Cost-Efficient Container Orchestration Strategy in Kubernetes-Based Cloud Computing Infrastructures with Heterogeneous Resources” ACM Transactions on Internet Technology, Vol. 20, No. 2, Article 15. Publication date: April 2020. https://doi.org/10.1145/3378447

[28] Rajkumar Buyya1, Maria A. Rodriguez1, Adel Nadjaran Toosi, Jaeman Park,” Cost-Efficient Orchestration of Containers in Clouds: A Vision, Architectural Elements, and Future Directions”, Cloud Computing and Distributed Systems (CLOUDS) Lab

[29] Nikhil Marathe , Ankita Gandhi , Jaimeel M Shah ,“Docker Swarm and Kubernetes in Cloud Computing Environment”, Third International Conference on Trends in Electronics and Informatics (ICOEI 2019),978-1-5386-9439-8/19/$31.00 ©2019 IEEE

[30] Jingze Lv, Mingchang Wei,Yang Yu,,” A Container Scheduling Strategy Based on Machine Learning in Microservice Architecture”, 2019 IEEE International Conference on Services Computing (SCC), 2474-2473/19/$31.00 ©2019 IEEEDOI 10.1109/SCC.2019.00023

[31] Chanwit Kaewkasi , Kornrathak Chuenmuneewong,” Improvement of Container Scheduling for Docker using Ant Colony Optimization” ,*IEEE cloud computing* , 978-1-4673-9077/4/17/ © 2017 IEEE October 2017

International Journal of
INTELLIGENT SYSTEMS AND APPLICATIONS IN ENGINEERING
ISSN:2147-6799 www.ijisae.org Original Research Paper

Container Scheduling: A Taxonomy, Open Issues and Future Directions for Scheduling of Containerized Microservices in Cloud Environments

Anil Prajapati[1] *, Dr.Manish M Patel[2]

Submitted: 03/05/2024 **Revised**: 16/06/2024 **Accepted**: 23/06/2024

Abstract: Containerization offers lightweight virtualization and modern applications are adopting containers because of their scalability, portability, and flexible deployment, particularly with microservices. In contrast to monolithic architecture, microservice design is a new paradigm that delivers granular and loosely coupled services. With containerization, it is feasible to create a scalable application architecture made up of several microservices. For their production environment, companies like Netflix, Google, Microsoft, and others have been using cloud environments based on containers. We have presented an extensive review of containerization, and microservice architecture in this paper. The study also discusses container orchestration, classification of container scheduling techniques, optimization objectives for scheduling of container-based microservices, and a comparison of different container orchestration platforms. Container scheduling strategies are widely developed using heuristic and meta-heuristic techniques. Designing a resource efficient scheduling technique for containerized microservice is a key challenge due to various factors such as dynamic workload and diverse resource requirements. Machine learning has great potential and machine learning-based techniques have been employed for the optimized scheduling of containerized microservices in recent years. It is possible to implement an intelligent container scheduling approach to forecast performance. Machine learning-based multi-objective container scheduling solutions can be proposed to obtain effective resource usage of cloud environment. We mention areas that still need investigation in the field of container scheduling for containerized microservices.

Keywords: *Cloud Computing, Containerization, Container scheduling, Microservice, Machine learning*

1. Introduction

These days, cloud computing is becoming more and more popular. Many apps are moving from private infrastructures to cloud-based environments to take advantage of its features, which include scalability, flexibility, agility, and cost-effectiveness. When physical resources of the cloud, such as memory, CPU, storage, and network resources, are allocated to different cloud-based applications and computational resources are used effectively, the cost of application deployment and operation is decreased. One of the main concerns for cloud service providers is how to deploy applications on the cloud at a reasonable cost. Resource management, scheduling, and allocation have all been extensively researched for the deployment of a variety of dynamic applications to address these problems. To streamline processes and minimize costs associated with developing and deploying cloud-based services, we have investigated the multiple architectural paradigms [1].

Resource virtualization—or sharing of resources amongst applications—is made possible by virtualization technologies. Applications on the same system can therefore operate in many execution environments. To provide the necessary resource isolation, hypervisor-based virtualization incurs expense by performing various duplicate functionalities to abstract the system resources. CPU overhead, memory overhead, network overhead, and Disk overhead are the overheads that occur in virtual machines [2].

Containers are quickly gaining popularity and replacing virtual machines in recent years due to their several promising characteristics, like shared host operating system, quick launch time, scalability, portability, and quick deployment. With containers, applications can boost productivity and portability by wrapping all required dependencies as code, runtime, code, system libraries, and system tools, into a single box. This creates a run-time environment that is platform-independent [3]. Software applications that offer virtualization at the operating system level are called containers. Container-based computing platforms can be installed, terminated, replicated, recovered from, and relocated in milliseconds owing to their architecture [4].

The paradigm of cloud computing is being revolutionized by containerization [3, 4, 5], and in response to the increasing demand for these services, numerous cloud service providers have begun to offer services based on containers. A few instances are Amazon Elastic Container Service, Google Container Engine, and Azure Container Services. Modern applications must communicate in real-

[1]*Research Scholar (209999913024), Gujarat Technological University, Gujarat, India*
ORCID ID : 0009-0002-6314-4987
[2] *Professor, Department of Information Technology, Sankalchand Patel College of Engineering, Visnagar, Gujarat, India*
** Corresponding Author Email: anil.apit18@gmail.com, it43manish@gmail.com*

time, get regular updates, and deal with fluctuating traffic patterns.

A new paradigm for developing applications for cloud computing is microservices. It separates an application into a collection of fine-grained, loosely connected services. It is a deployable and independently scalable application component. The traditional "monolithic" approach for developing applications, where every application is a stand-alone, independent component, is contrasted with the microservice-based method. For example, in client-server applications, the server is a single unit that executes logic, gets and/or changes data, and responds to HTTP requests. Therefore, the challenge with these monolithic architectures is that every time the logic of the application changes, an updated version of the whole code base must be deployed. Because each microservice only handles a single subtask or service, there is less overhead in communication and less computing power needed.

Numerous major corporations, such as Netflix, Amazon, IBM, Uber, Alibaba, and others, have moved their apps to this development framework because of microservices [6]. These days, cloud-based applications are frequently developed using the microservice architecture, which is growing in popularity in application design and development [7]. Microservices can be deployed and encapsulated in a cloud environment using containers, a lightweight virtualization technology. Certain scheduling techniques for containers have been suggested as a result of the advancements in container technology and the growing popularity of microservice architecture. Nonetheless, there are still a few significant issues with scheduling of containerized microservice in cloud environments that need to be resolved.

The growing popularity of microservice architecture and container architecture for cloud computing offers the chance to enhance the elasticity and scalability of application development. Compared to virtual machines, containers offer more portability, faster deployment, and reduced utilization of resources. As a result, it's a great tool for scheduling, encapsulating, and deploying microservices. At present, Google Kubernetes, Apache Mesos, and Docker Swarm are the most popular container cluster management tools [8, 9, 10].

To the diverse workloads and resources available on the cloud, container scheduling has become essential for the cost-effective operation of microservices on the cloud platform. Researchers have proposed various container scheduling approaches to accomplish different performance objectives, including response time, energy consumption, resource utilization, and availability, load balancing, and cost. There was some investigation into the practical deployment of microservice-based applications using containers. In a cloud system based on containers, we investigated container scheduling for microservice deployment. One researcher has developed approaches to figuring out how much resources to allocate to each microservice and how to scale effectively when workload varies.

In addition to meeting the cloud cluster's load requirements, an efficient container resource allocation strategy guarantees the cluster's reliability and efficiency. Additional study is necessary to investigate the performance of microservice applications, cloud cluster reliability, and network transmission overhead between microservices, all of which can be enhanced [7].

The remainder of the paper has been arranged in this manner. A brief history of containerization, container architecture, and microservices architecture is given in the second section. The third section presents a comparison of several container orchestration technologies, classifies container scheduling strategies, and explains the optimization objectives used to schedule microservices based on containers. The scheduling of containerized microservices is discussed in the fourth section along with the relevant container scheduling work for microservices. In the fifth section, the significance of machine learning is discussed, along with an in-depth investigation of the techniques based on machine learning that are employed in container scheduling. Additionally, it shows how well machine learning techniques perform in different optimization objectives and the scheduling container technology. The sixth section presents research and the future. The future and research directions in the field of containerized microservices scheduling are presented in the sixth section. The seventh section brings survey to its conclusion.

2. Background

Significant background information on container architecture, microservices architecture, and container engines is presented in this section. We have also showcased the associated research that compares the performance of virtual machines and containers made with Docker.

2.1 Containerization

High service downtime occurs while an application is being upgraded on a virtual machine. On the other hand, due to their faster startup times and improved performance, containers are promising lightweight virtualization technology for cloud-based services, particularly when it comes to microservices, edge computing, smart cars, and the Internet of Things [10]. The architectural approaches for virtual machines and container technologies are distinct, as seen in Figure 1. Considering the container architecture, a container engine represents the top layer of the stack's resource management system. It is situated directly on top of the host operating system, whereas the comparable layer

in the virtual machine architecture is called the hypervisor. The fact that a guest operating system is not needed for running containers is a fascinating characteristic of the container-based design. Compared to virtual machines, it offers portability, efficiency, and less overhead because the hypervisor layer is removed [11].

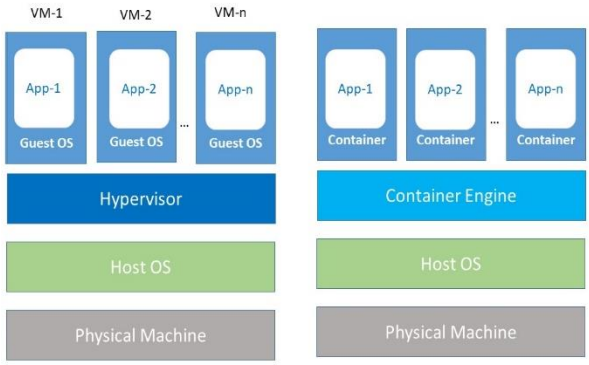


Fig 1. Hosting Approach: Virtual Machines and Containers Furthermore, the authors in [11][12] found that in terms of total throughput, services deployed using containers perform far better than virtual machines. Virtual machine deployment of real-time applications was also found to have a significant performance overhead, as shown by the CPU utilization and memory consumption. Moreover, Docker containers are allegedly launched on top of Amazon EC2 virtual machines by Amazon Cloud, which goes against the standard procedure for application deployment shown in Figure 1. The container deployment strategy used by Amazon Cloud Platform is depicted in Figure 2.

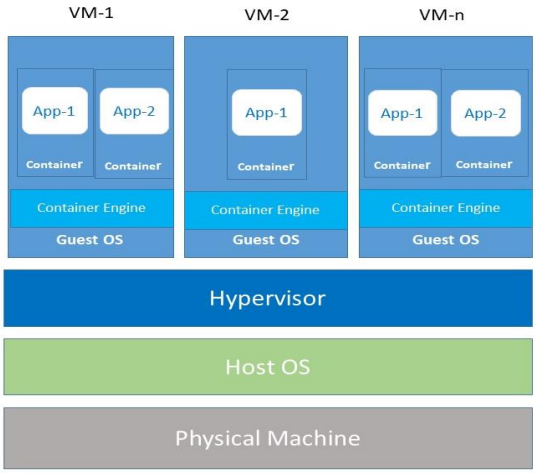


Fig 2. Container hosting approach in Public Cloud

Developers use container technologies extensively for deploying various microservices. Despite containers' enormous popularity in cloud computing, no comprehensive study is present that addresses scheduling approaches for containers from the perspective of containerized microservices [12,13]. When it comes to application deployment, developers who build and deploy application containers must make the most of the infrastructure's performance. The demand for various types of container scheduling algorithms for the deployment of containerized microservices is driven by this resistance to cloud providers.

2.2. Container Engine Architecture

Software developers build and distribute container images, which are files that include the data needed to run a containerized application. Containerization tools are used by developers to create read-only, non-modifiable images in containers. On top of any infrastructure, users can create containers with the use of containerization tools. The most widely used container solution available right now is called Docker, and here it is used to illustrate the concept of containerization.

Docker is an open-source platform that makes it simpler to build, deploy, and run services that use containers. Users can deploy applications more quickly by separating the applications from the infrastructure through the use of Docker containers. Docker Swarm, Docker Compose, Docker Images, Docker Daemon, and Docker Engine are major components of Docker. We may manage this infrastructure in the same manner that we would an application. The Docker container engine's architecture is depicted in Figure 3.

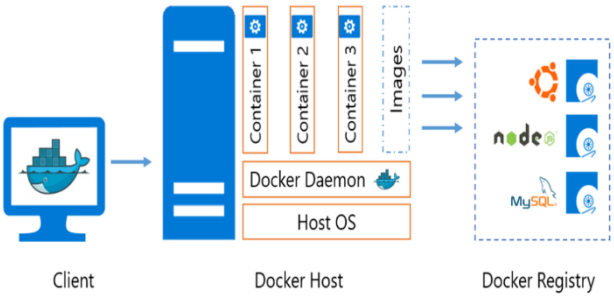


Fig 3. Architecture of Container Engine [14]

The component of Docker that creates and manages Docker containers is the Docker engine. The container is an instance of a Docker image that is currently running. A Docker image is a read-only template which contains the instructions on how to build a Docker container. The Docker daemon serves as a process that is used to manage and control the containers. The primary service that Docker users use to interact with Docker containers is the Docker client. Every Docker image is stored in a Docker registry. The Docker CLI (Command Line Interface) enables us to start, stop, or remove containers [14].

2.3. Microservice Architecture

A new paradigm for developing cloud applications is microservices. An application is divided into several fine-grained, loosely coupled services using microservice architecture. A microservice is a deployable, independently scaled application component. The traditional "monolithic" approach to application development, in which each

application is a single autonomous unit, is compared with the microservice-based approach. Many organizations, including Uber, Netflix, and Amazon, are switching from traditional monolithic architecture to microservice architecture because of their high scalability requirements and massive user counts. A comparison of the microservice and monolithic architectures is shown in Figure 4. Monolithic applications are scaled by replicating them on multiple servers, while microservices are scaled by distributing across cloud servers and replicating them as and when required.

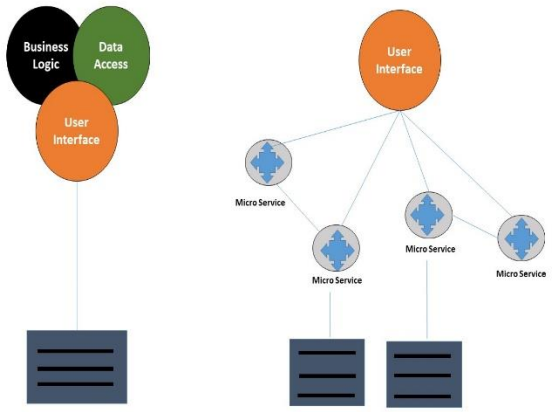


Fig 4. Monolithic and Microservice architecture

Compared to monolithic architectural approaches, microservices are independent components of an application that are small and easy to work with. The flexibility to adapt and be implemented on their own is referred to as the services' autonomy. Microservices can independently create, deploy, test, and operate; they are typically run by separate developer teams and structured around business logic. Moreover, microservices may utilize many technologies and be developed using different programming languages. Microservices are deployable and scalable using container-based virtualization [16]. The deployment of a microservice instance in a cloud container is depicted in Figure 5.

A newly created microservice should be able to register itself by saving its runtime configurations and updating the deployment information so that it can communicate with other microservices in the same application. Microservices will have their development framework and each will be developed independently. To create microservice binaries, the deployment server retrieves the source code from the repository, compiles, and tests the microservice code. The deployment server generates a container image with these binaries, which is then kept in the repository of containers. These container images are deployed to the deployment server after they are created. The developed microservice would be deployed in one or more containers.

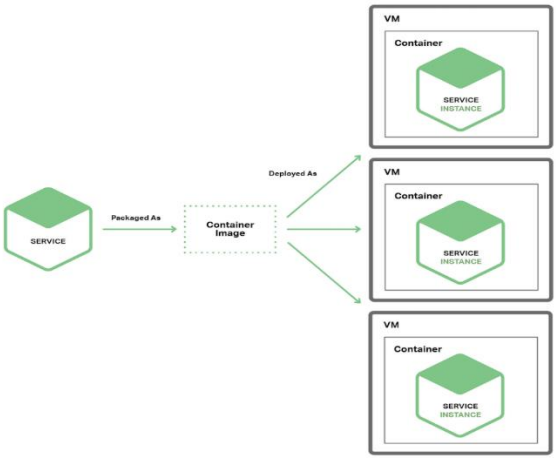


Fig 5. Deployment of Microservice in container [16]

A newly created microservice should be able to register itself by saving its runtime configurations and updating the deployment information so that it can communicate with other microservices in the same application. Microservices will have their development framework and each will be developed independently. To create microservice binaries, the deployment server retrieves the source code from the repository, compiles, and tests the microservice code. The deployment server generates a container image with these binaries, which is then kept in the repository of containers. These container images are deployed to the deployment server after they are created. The developed microservice would be deployed in one or more containers.

3. Container Scheduling for Microservices

Emerging requirements for container scheduling are imposed by the container-based infrastructure to guarantee the performance of deployed microservice-based containerized applications. The underlying physical machine or cloud cluster must provide the resources that a container requests, which are frequently a combination of several resources like CPU, memory, network, etc. Since there are frequent updates and data transfers, containers inside a distributed application frequently have a strong affinity for one another. Thus, affinity needs to be considered when planning container schedules. During the execution of the application, the scheduling algorithms are routinely called upon, especially when scaling out or recovering from failure, which typically involves crucial time constraints. As a result, the scheduling overhead ought to be minimal [17,4].

3.1 Container Scheduling Classification

Numerous research papers on container scheduling have been reviewed by us. It was discovered that the four kinds of container scheduling approaches under investigation each had different performance and quality characteristics. The majority of the suggested approaches fall into Mathematical modeling-based techniques, Heuristics based techniques, Meta-heuristics-based technique, and Machine learning-

based techniques.

Mathematical modeling offers the optimization of a linear function using a set of linear constraints, the technique is called integer linear programming. From the perspective of container scheduling, it does not offer optimum solutions promptly. Many researchers have proposed linear programming models for container deployment considering various optimization objectives to obtain optimum container scheduling and showed efficient results as well. It has been observed that mathematical modeling is suitable to the problem in small size.

After a detailed survey it is found that compared to mathematical modeling, the heuristic techniques are preferable in container scheduling. Most of the approaches investigated in recent years use some kind of heuristic to find the optimum container scheduling. In the majority of situations, heuristic algorithms are simple to use and offer effective scheduling in a small amount of time. Most of these approaches combine current container scheduling techniques and apply them in Kubernetes and Docker Swarm. [3] Summarized the heuristic techniques employed in container scheduling. A researcher has showed a scheme for efficient resource utilization. Container scheduling framework for the Docker environment also proposed best-fit scheduling that offers dynamic monitoring and showed a 23% energy-efficient solution as compared to Docker Swarm. A multi-objective approach that combines all three (Spread,Binpack,Random) of Docker Swarm to provide efficient container scheduling is proposed with considerations of memory utilization, network delay, CPU utilization, and interaction between cluster nodes and containers. The approach showed efficient performance. After the survey, it is observed that heuristic techniques generally offer quick optimization, furthermore, they can be integrated with other optimization approaches for enhanced container scheduling.

Meta-heuristic approaches are promising in getting the optimum solution in a variety of sectors nowadays. Ideally, these approaches are adaptive to the environment and they can be based on Genetic algorithms, Ant Colony Optimization, and Particle Swarm Optimization. [3] Summarizes meta-heuristic approaches used in container scheduling. Kaewsaki(2017) presented ACO(Ant colony optimization) based approach presented to improve resource usage and load balancing. The strategy was experimented on Docker Swarm and showed 15% enhanced performance. From the perspective of the microservice, Lin(2019) presented a multi-objective ACO scheduling technique that is used to optimize network transmission overhead, failure rate, and resource utilization of containerized microservices. Results showed enhanced resource usage and reliability. Genetic algorithms and Particle Swarm Optimization-based container scheduling techniques have also been proposed in recent years. Li(2018) proposed a PSO-based technique to improve load balancing and resource usage. Results showed enhanced performance by 20% compared to the spread technique of Docker Swarm.

After an extended literature survey, it is observed that researchers have proposed hybrid scheduling techniques for container scheduling in which ACO and PSO are combined to offer efficient scheduling mechanisms. Furthermore, it is found that meta-heuristic approaches are more suitable for multi-objective optimization. The field of machine learning has been an emerging field of study that has shown tremendous potential in numerous applications across various fields and it holds great potential for container scheduling. Particularly when it comes to scheduling the containerized microservice, machine learning approaches are thoroughly investigated. We have investigated the machine learning algorithms employed for the scheduling of the containerized microservices and discussed in Section-5 of the paper.

3.2. Performance Objectives for Container Scheduling

According to particular user requirements, the scheduling decisions typically need to accomplish several performance objectives. When deploying applications in containers, striking a balance between competing optimization objectives is still a major challenge. The objectives mentioned below are thought to be the most typical ones for scheduling cloud containers [3,4].

Energy efficiency: The huge amount of electricity used by cloud environments has become a major concern in the area of cloud computing due to the constantly expanding scale of cloud data centers. When deploying containers on nodes that are working, the amount of power used is referred to as energy consumption. The objective seeks to reduce the cluster's overall power consumption.

Cost: The total cost of running an application is determined by the price of several services, including computing, storage, and communication. The amount of time needed to run an application on the cloud cluster's available cores is referred to as the computation cost. The more time an application runs on a processor, the more costly it becomes.

Availability: It specifies how long a user has access to an application. Availability is a key component of cloud computing, where users should always be able to access a service or application. The objective is to ensure that the schedule that is generated offers some kind of fault tolerance for the different services that the application provides, requiring redundancy in the number of containers that are deployed in the cloud.

Resource Utilization: It is the effectiveness with which a worker node uses its memory, core, and network bandwidth. To achieve this goal, a cluster node's energy efficiency and

cost-effectiveness increase with its resource utilization. For energy and cost efficiency, infrastructure-level resource utilization measures like CPU and memory usage are typically regarded as vital application performance indicators.

Load Balancing: The task of dividing up the workload among available resources equally so that no one node becomes overloaded is known as load balancing. Throughput, cost, and response time are all impacted by this objective. This measure is

critical, particularly for container applications that take advantage of microservice architectures.

Scalability: The scalability of containers to maintain the required service even in the face of growing system demand for the application or service is another important parameter.

Latency: The duration needed for an application to run from start to finish is known as latency. Reducing latency is always the objective of a good scheduler. Cost and throughput are greatly impacted by this objective.

Security: An ability to protect data and services from malicious attacks or software bugs by encryption and access control methods. Orchestration tools must be able to offer essential security.

Network Bandwidth: The number of bits transferred in one second on the network is referred as network bandwidth. It would assist in assessing network delays and handling congestion that occurred during the communication of containers.

Carbon Emission: It is the volume of Carbon Dioxide that is emitted into the atmosphere as a result of using electric power that is specifically created when fossil fuels are burned.

SLA assurance: The majority of containerized applications are set up with precise performance parameters, including throughput, launch time, completion time, and response times. The majority of these constraints are outlined in SLA contracts, and breaking them may result in a penalty.

3.3. Container Orchestration

The containers are widely employed by organizations to deploy modern applications such as IoT and Big data in cloud data centres. As a result, container orchestration has emerged. Container orchestration facilitates defining, selecting, deploying, monitoring, and dynamically control the configuration of containerized services in the cloud. The container orchestrator performs the scheduling of containers, selecting the optimal node and ensuring the container is running in the desired state. Container orchestration system enables organizations to streamline application development by deploying the same application without having to rebuild it in a cloud computing environment.

The deployment of containerized microservices, cost, and performance are all significantly impacted by the intricate container scheduling problem. Applications are not just moved between servers and turned on and off with container orchestration. When a multi-container application is deployed, users can specify how to manage the cloud's containers through the use of container orchestration. Container orchestration describes the way multiple containers are managed as a single unit in addition to how they are initially deployed [18]. The section discusses taxonomy of container orchestration and analyses the orchestration systems in the context of the scheduling of the containerized applications. [19] Analyses the existing container orchestration systems concerning container technology used, the application model it follows, placement constraints, and resource granularity. Orchestration systems for the system objectives such as scalability, availability, throughput, and resource utilization are also investigated. The result of the investigation is summarized in Table 1.

Container orchestration tools available today are as follows.

Docker Swarm

The built-in scheduling and clustering mechanism for Docker containers is called Docker Swarm which orchestrates the application or service running in a Docker container. Applications are treated as services; they might be decomposed in microservices and may be deployed in one or more containers. The Swarm manager is used for controlling the entire life cycle of applications containerized in Docker containers. It supports three scheduling techniques by default: First, there is the Spread approach, which chooses the newly deployed containers for operation on the least loaded hosts; second, there is the BinPack strategy, which chooses the most loaded host with sufficient resources to run the containers. In addition to this, the third is the Random method, which chooses the node at random.

Kubernetes

An open-source container orchestrator called Kubernetes was first created by Google that facilitate the automated deployment and scaling of containerized applications. The pod is a group of containers and the basic building block of the Kubernetes orchestration framework. Kubernetes scheduler provides schedules for each pod based on available resources, filtered by user-specified requirements, and ranked according to application affinities that are individually defined. The Kubernetes cluster contains Master and Worker nodes, where the master node is the fundamental component and the worker nodes perform services to run pods.

Mesos

Two steps are available for scheduling with Apache Mesos. It allocates the physical node's resources to each application in order

during the first stage, which is referred to as the framework. After accepting the offer, the framework can use its built-in framework scheduler to plan its tasks on the obtained resources. Mesos does not offer service discovery. Hence, Kubernetes or Swarm is integrated to address this lack. Orchestration frameworks Aurora and Marathon depend on Mesos to manage the cluster resources of the container cluster.

Aurora is developed by Twitter, a scheduler that runs on top of Mesos and enables long-running services to be deployed on the container.

Table 1 . Taxonomy of Container orchestration tools

Orchestrator	Swarm	Kubernetes	Apache Mesos	Aurora	Marathon	YARN	Omega	Fuxi
Organization	Docker	Google	UC Berkeley	Twitter	Mesosphere	Apache	Google	Alibaba
Open Source	✓	✓	✓	✓	✓	✓	-	-
Technology of container	Docker	Docker	Mesos containers, Dokcer	Apache Mesos, Docker	Mesos containers, Docker	Linux cgroups-based, Docker	N/S	Linux cgroups-based
Pre-emption	-	-	-	✓	-	-	✓	-
Rescheduling	-	-	-	✓	-	-	✓	-
Scheduling constraints	Label and affinity-based	Label and affinity-based	N/A	Value and limit-based	Value and limit-based	Value and limit-based	N/S	Value based
Scalability	✓	✓	✓	✓	✓	✓	✓	✓
High Availability	✓	-	-	-	-	-	✓	✓
High Utilization	✓	-	-	-	-	-	✓	✓
High Throughput	✓	-	-	-	-	-	✓	✓
Resource granularity	Fine-grained	Fine-grained	Fine-grained	Fine-grained	Fine-grained	coarse-grained	Fine-grained	Fine-grained
Application workload	Long running tasks	All	All	Long running tasks	Long running tasks	Batch tasks	All	Batch tasks

Marathon is a framework for Mesos that is developed for the orchestration of long-running services. It offers fault tolerance and high availability; it ensures that deployed applications will continue running even in the situation of node failures.

YARN is designed for orchestration of Hadoop tasks, it also supports frameworks like Giraph, Spark, and Storm. Each application framework running on top of YARN coordinates its execution flows and optimizations as it sees fit.

Omega is Google's next-generation cluster management

system

that offers a platform that enables specialized and customized schedulers to be developed, providing users with great flexibility.

Fuxi is a resource management and scheduling system that supports Alibaba's proprietary data platform. It is the resource management module on their Aspara system, which is responsible for managing the physical resources of Linux clusters within a datacenter [20].

4. Container Scheduling for Microservices

In addition to fulfilling user service needs, efficient container scheduling minimizes running overhead and guarantees the performance of the application and the cloud environment underneath it [9,20]. We have reviewed the open issues within the scheduling of containerized microservices and presented herein this paper, different scheduling methods employed for the containerized microservices are investigated and discussed in this section.

4.1. Issues in Scheduling of Microservices

The number of independently running microservices must fulfil the necessary function, handling thousands of requests for access and ensuring high availability, scalability, and tolerance over network failures [21]. The services that comprise the application must communicate with one another constantly to use the microservices architecture. In order to complete the task, each service must simultaneously use the interface for interaction between its services. The communication between microservices is extremely difficult because an enormous number of microservices instances run on a large number of containers, and the placement of microservices instances is also constantly changing High complexity and dynamicity pose many challenges for handling microservices.For managing microservices networked together such as 2-tier or 3-tier web applications and Internet of Things (IoT) based applications, we have yet to discover a conventional large-scale, optimized scheduling platform. Following a thorough survey, we discovered the following issues with microservices scheduling [22].

Configuration and Management

Typically, a cloud application integrates several interdependent microservices, such as a web server, database server, and load balancer, to provide a variety of functionality. These microservices also have interference and dataflow dependencies. Dealing with different microservice configurations and cloud data center resources operated by diverse performance needs presents some issues.

Application Composition

The workloads associated with various microservices are interdependent, meaning that modifications to one microservice dataflow and execution will have an impact on others. All things considered, the application composition must address the entire life cycle, including deploy, patch, monitor, reconfigure, and shutdown, all of which are influenced by each of the microservices' objectives for performance.

Monitoring of microservice

A clear and updated understanding of objectives for performance across microservices and data center resources is necessary for optimal application performance. Measures of performance include, for instance, throughput and latency for storage resources, utilization and throughput for CPU resources, and query response time for microservices based on NoSQL and SQL databases. Thus, there is still work that needs to be done to define and construct performance objectives across microservices in a coherent manner that provides a comprehensive perspective of data and control flows.

Elastic Scheduling and runtime Adaptation

Microservice scheduling is challenging due to several runtime uncertainties. Microservice workload behavior, including request arrival rate, type, and processing time, as well as Input/output system behavior and the number of users connecting to different microservice types and mixtures, is difficult to estimate. Developing workload models specifically for microservices presents a significant challenge: accurately determining and fitting statistical functions for observed distributions, including those pertaining to request arrival patterns, CPU and memory utilization patterns, I/O system behaviors, request processing time distributions, and network usage patterns. Manually solving the issue wouldn't be possible.

Microservice Interference

If several microservices are deployed in a single container, there may be resource conflict and interference because various microservices may have similar resource requirements. To minimize interference and make the best use of available resources, it's essential to understand how microservices running in tandem fluctuate in terms of performance.

Service Discovery

During the running time of microservices, discovering the proper microservice needs the orchestration which needs to be aligned with the essential quality of the service. An essential quality parameter is the latency of the discovered and triggered microservice.

Performance isolation

In a containerized system, a single container can handle several heterogeneous microservices that offer different application-specific functionalities. Unexpected conflict and interference are possible in this scenario. Certain microservices have more storage requirements than others, some have more computational requirements than others (such as transactional query processing by database servers), and some have more communication requirements than storage requirements.

Communication and integration

These issues have arisen because of the distributed architecture of the microservices. It is challenging to ensure that the communication system is dependable and that the protocol to be used for communication and integration can manage challenging processes, even when microservices communicate with a more lightweight protocol. Reliability and durability are the two most crucial requirements for both problems; if these are not satisfied, the system's capacity to function properly and reliably will be impacted, perhaps leading to cascading failures.

4.2 Approaches for scheduling of containerised microservices

The deployment of containerized microservices poses some challenges, most of which are related to planning and deployment configuration in the container. The heterogeneous requirements of microservices demand an optimal container scheduling mechanism to meet these requirements. The microservices need to be monitored and automatically adjusted resources according to varying loads for the predictability and availability of the application. These requirements bring a lot of challenges for the scheduling. The chain of services must be arranged and managed, and the available resources must be scheduled for efficient utilization. The system's dependability and availability are directly impacted by ineffective scheduling. Additionally, container scheduling for microservices frequently needs to be reviewed for effective analysis and, further improvement. In this section, we have presented the studies and work related to the scheduling of microservices.

[MIAO LIN, 9] presented a model to enhance the microservice-based container resource allocation approach. This model includes reducing overhead of network transmission between microservices, load, and enhancing the cluster services' reliability. The researcher used the average number of microservice request failures, the maximum value of the resource utilization rate, and the network transmission overhead between microservices to achieve this. This was implemented using an Ant colony optimization (ACO_MCMS) and it was presented to address the issue of containerized microservices in the cloud platform. It is compared with the current container scheduling policies for an average number of microservice request failures and computing resource utilization. The result shows enhanced performance.

For microservices in clouds, [Sheng Wang, 23] suggested and developed a task scheduling algorithm of microservices as a cost optimization. Based on the workload statistics, the proposed strategy selects the container and adapts to the streaming load based on the statistics of the workload. The strategy was tested in simulation-based environment and results show reduced the cost in comparison with existing scheduling mechanisms.

A microservices scheduling method called LWFF presented by [H M Fard, 24] in which the scheduling problem is characterized as an advanced version of the knapsack problem and solved using a multi-objective optimization technique, Findings indicate that, in comparison to the state-of-the-art, the suggested mechanism is extremely scalable and concurrently boosts CPU and memory usage, which improves throughput. The suggested container scheduling technique outperforms the Spread and Binpack scheduling strategies in terms of throughput.

A novel Microservice-based container framework proposed by [XUEHUA ZHAO, 25] for running mobility and delay-sensitive applications at the lowest possible cost. The findings demonstrate that the suggested approaches can decrease costs, improve utilization of resources, and minimize service delay.

Using the TOPSIS algorithm, [Tarek, 26] presented a novel scheduling approach that combines the principles of Bin-Packing and the Spread techniques. The suggested approach aimed to identify, from a group of nodes that comprise a cloud infrastructure, the node with the best balance between three factors: (i) the total number of containers on the node; (ii) the total number of CPUs available; and (iii) the total amount of RAM accessible. The proposed scheduling mechanism is experimented with in Docker Swarm and compared with Spread, Binpack, and Random strategies. The result shows improved performance under various scenarios.

The first Ant Colony Optimization technique for container scheduling was introduced by [Kaewkasi, 3] to improve the utilization of resources through appropriate load balancing. Docker Swarm was used to integrate and test the proposed strategy. The outcomes demonstrated a 15% improvement in performance as compared to the Swarm Greedy technique. Nevertheless, the strategy only took into account a small number of optimization objectives, like load balancing and utilization of resources.

[Mehmet,27] focuses on the challenges of scheduling and auto-scaling of containerized microservices. The authors claimed that existing scheduling mechanisms underperform with streaming workloads and in architecture consisting of virtual machines and containers. Thus, to overcome these challenges, researchers proposed an elastic scheduling mechanism that handles task scheduling and auto-scaling, which is based on a variable-sized bin packing problem (VSBPP). The proposed mechanism improves the success ratio and cost.

C.T Joseph (2021) proposed a microservice rescheduling framework to address performance degradation and response time challenges. The author says, to define the quality of any microservice, response time is an important measure. The author claimed that the effect of configuration parameters of containerized microservices has not been handled well by the researchers.

5. Machine Learning in scheduling of microservice based containers

Automation of the container orchestration process for complex and heterogeneous workloads under cloud computing systems is very uncertain today. In the studies, we found that machine learning has already been employed in virtual machine orchestration. Current container scheduling mechanisms are typically designed with heuristic scheduling policies that do not take into account the variety of workload situations and QoS needs of an application. The majority of these techniques are used for small-scale systems that are configured offline based on specific workload scenarios. Such scheduling techniques cannot handle highly dynamic workloads where applications need to be scaled at runtime according to specific behavior patterns. When the system scales up, heuristic techniques may perform significantly worse. When resource provisioning, dependency structures between components of containerized systems are not taken into consideration. Within an application, several microservice units contain internal relationships and depend on one another. Microservice architecture suffers as a consequence of increased resource requirements and communication expenses.

The majority of the attention of current container orchestration approaches is on evaluating infrastructure-level metrics; application-level metrics and particular QoS (Quality of Service) needs are not given enough weight. The increasing complexity of managing applications on cloud platforms has compelled cloud service providers to use machine-learning approaches to optimize their container

Table 2. Machine learning-based models used for scheduling containerized microservices

Problem in scheduling of containerized applications	Machine Learning based solution	Advantage/ Limitations
How can the interdependencies between micro services be analyzed?	BO, GP, CNN, and LSTM	Increased accuracy of predictions/ Ignorance of updates regarding dependencies
How can task dependencies in containerized applications be analyzed?	SVM	Increased Accuracy/ The time overhead and computational expenses are explained implicitly
How can resource scheduling for containers be made energy-efficient?	MDP, Q-learning, and SARSA	Cost saving, Minimizing task execution time/ Limited scalability
When migrating micro services to cloud containers, how can the communication delay be minimized?	DNN, Q-learning	Minimizing task execution time/Limited scalability
How can workloads be characterized by forecasting and modeling the behavior of requests and the pattern of resource consumption?	ARIMA, Bi-LSTM, LSTM, K-means++,GRU, TSNNR	Optimized Resource utilization/ Scheduling delays

orchestration strategies [28]. Therefore, machine learning techniques could be used to model and forecast metrics at the infrastructure or application level, including workload characteristics, system resource consumption, and application performance. However, machine learning algorithms, which offer better accuracy and less time overhead in large-scale systems, could generate resource management decisions directly in place of

heuristic techniques. We investigated the research work

[3] Reported the work related to machine learning-based container scheduling. Resource optimization is very challenging in diverse workload scenarios. To boot resource usage, Nanda(2018) proposed a reinforcement learning-based approach for container consolidation. The proposed technique showed enhanced results compared SJF and random scheduling techniques. Lv(2019) presented a random forest regression model for the prediction of the need of containers. The model accurately predicted the future resource needs. Liu(2020) proposed a model to

Table 3. Container scheduling techniques and optimization objectives have been used by the researchers

Objectives						ML approach used	Container Technology used	Limitations
Energy	**Availability**	**Utilization**	**Load Balancing**	**Cost**	**Network**			
		✓	✓			ACO	Docker	Require parameter tuning
	✓	✓	✓		✓	First multi-objective GA	Kubernetes	No energy reduction
✓	✓	✓	✓			GA	Docker	Slow
		✓	✓			Random Forest	Kubernetes	Only predict the resources and lack of testing in real time
✓		✓	✓			K-means	Docker	Few optimization objectives are used
✓						NN based model	Docker	Performance is not considered
✓		✓				Linear Regression	Cloudsim	Cost is not taken in consideration

where

machine learning algorithms are utilized in container scheduling.

In past years, machine learning has gained immense popularity and is widely employed in many research areas. According to their optimization objectives, we found that the most often used machine learning models in the container scheduling area are Regression, Classification and Decision-making. The overall performance of the application and resource efficiency are directly impacted by the quality of scheduling selections. Following our survey, it was found that a few researchers used machine learning-based models in the last few years, particularly for scheduling of containerized applications.

predict the physical machine-level resource usage are considered to improve energy efficiency and Service Level Agreement of datacenters. The proposed model was tested in ContainerCloudSim and showed reduced energy consumption.

[28] Shows an evaluation of machine learning-based container scheduling technologies. In 2016, K-nearest neighbor was employed to estimate the resource consumption of containerized applications. However, the application model only considered the time series pattern of infrastructure-level resource metrics. In Shah(2017) long short-term memory (LSTM) model was applied to microservices dependency analysis, based on neural networks and fitted for classification, processing, and forecasting. The approach assessed the time series pattern of resource metrics as

well as the internal relationships among microservices. Tang(2018) presented a model for performance analysis of important resource metrics which was carried out using the bidirectional LSTM (Bi-LSTM) model. This model was employed to forecast application throughput and workload arrival rates. Comparing their training module to ARIMA and LSTM models, it has shown a notable gain in accuracy in terms of time series prediction. Podolskiy(2019) used Lasso regression (LASSO) to forecast the service level indicators (SLI) for the predictions of application response time and throughput. The ANN model was trained using the reinforcement learning technique, which provided the best scheduling mechanism with the least amount of performance interference.

In 2020, many Reinforcements Learning based approaches were suggested. Qiu (2020), employed a model for microservice dependency analysis and identification of the essential components most likely to encounter resource shortages and performance degradations. This model was implemented using SVM. Zhang (2021), proposed a model combination of Convolutional neural networks (CNNs) and boosted trees (BT) to analyze the dependability and performance of microservices, with promising results.

Table 2 displays findings following our thorough analysis of the machine learning-based container scheduling model. It demonstrates how different machine learning-based scheduling techniques have been designed for containerized microservices. A summary of meta-heuristics and machine learning-based container scheduling techniques summarized with the optimization objectives have been used by the researcher and shown in Table 3. Analysis has shown that approaches based on machine learning and meta-heuristics are better suited for multi-objective optimization.

To summarize, the majority of research studies use reinforcement learning models for scheduling decision-making to improve utilization of resources and reduce task completion time.

6. Discussion and Future Research Directions

Applications like the Internet of Things (IoT), microservices, smart infrastructure, and containers are growing at a rapid pace and are widely used in cloud computing environments. Containerization is a lightweight virtualization solution that facilitates microservice application encapsulation, scheduling, and deployment. For this reason, a lot of cloud service providers have integrated container technologies into their infrastructure to automate applications. Container scheduling is proposed as a key research challenge to address the automation of deployment, scheduling, auto-scaling, and networking of containerized applications. Applications that are highly diverse and dynamic, significantly increase the complexity of container scheduling.

Conventional scheduling and optimization techniques are insufficient for scheduling containerized applications, according to an analysis of different container scheduling methods. All of the important objectives for performance cannot be achieved by a single algorithm. Therefore, there are still several issues that need to be resolved as important areas for future research in the field of scheduling the deployment of containerized applications in cloud environments. With the rise of the fog computing paradigm, processing and storage are being pushed closer to the user for real-time applications resulting in reduced energy consumption and quicker response time. The essential technology used to provide such services are containerization, which allows for dynamic workloads and applications. To use this new field of microservice applications, more resource-aware and energy-efficient container scheduling strategies are also needed.

In the last few years, machine learning-based techniques have also been used for containerized application scheduling improvement. Nonetheless, we have shown how different strategies compare in terms of performance and optimization objectives. Only machine learning would be useful for dynamic and varied workload-specific scheduling strategies. It would also be able to predict future consumption of resources and workload, which would enable intelligent scheduling decisions. Microservices have become widely used in many domains, and most cloud-native applications these days might have a large number of microservices.

The number of user requests, their interdependence, interference, and scalability all affect how an application behaves, which increases overhead and lowers performance. Therefore, an intelligent and efficient container scheduling that takes into consideration every microservice architecture issue would be needed. Multi-dimensional metrics for performance would be predicted using machine learning methods. The quality of scheduling and resource provisioning decisions made in response to shifting user demands in complex systems with diverse resource utilization and dynamic workloads may be further enhanced by these insights.

7. Conclusion

In this paper, we showed that the most appropriate method for deploying containerized microservices in a cloud context is containerization, which offers lightweight virtualization. The growing popularity of cloud-based containers highlights how crucial microservices management and orchestration are to the entire microservices architecture. Based on the optimization strategy used for containerized application scheduling, we have presented the classification of scheduling approaches and discussed several container

orchestration tools.

Docker and Kubernetes would provide the best performance in container orchestration and management from the perspective of application developers. Our research indicates that estimating the behaviour of microservice workload concerning processing time, request arrival rate, type, and number of users connecting to various microservice types is challenging. For predicting performance objectives, the microservice-based workload model can be employed. In our studies, we found that Machine Learning has a great potential for implementing intelligent scheduling mechanisms. Our survey results show that machine learning techniques are the most effective for the scheduling of containerized microservices by making the most use of the underlying cloud resources. Based on these findings, we have suggested potential further studies in this paper.

Furthermore, to offer efficient container scheduling, multi-objective container scheduling techniques can be employed to manage resources efficiently. Hybrid machine learning-based approaches can also be employed. Our survey would help researchers identify the key characteristics of Machine learning-based approaches and choose the most suitable method for efficient container scheduling for the microservices.

Conflicts of interest

The authors declare no conflicts of interest.

References

[1] XiliWana , XinjieGuana,∗ , TianjingWanga , GuangweiBaia , Baek-Yong Choi, "Application deployment using Microservice and Docker containers: Framework and optimization", Journal of Network and Computer Applications 119 (2018) 97–109

[2] M. SRIRAGHAVENDRA1 & PRATEEK JAIN2," VIRTUAL MACHINE VS CONTAINER: AN APPLICATION PERFOMANCE" ,International Journal of General Engineering and Technology (IJGET) ,ISSN(P): 2278-9928; ISSN(E): 2278-9936 Vol. 6, Issue4, Jun – Jul 2017; 29-40

[3] Imtiaz Ahmad, Mohammad Gh. AlFailakawi , AsayelAlMutawa, LatifaAlsalman," Container scheduling techniques: A Survey and assessment", Journal of King Saud University – Computer and Information Sciences

[4] Y. Hu, H. Zhou, C.d. Laat et al.," Concurrent container scheduling on heterogeneous clusters with multi-resource constraints" ,Future Generation Computer Systems 102 (2020) 562–573

[5] MIAO LIN 1 , JIANQING XI1 , WEIHUA BAI 2 , AND JIAYIN WU ,"Ant Colony Algorithm for Multi-Objective Optimization of Container-Based Microservice Scheduling in Cloud"IEEE Access VOLUME 7, 2019

[6] Qian Qu, Ronghua Xu, SeyedYahyaNikouei, Yu Chen "An Experimental Study on Microservices based Edge Computing Platforms", IEEE Xplore 2017

[7] VindeepSingh,Sateesh K Peddoju,"Container-based Microservice Architecture for Cloud Applications", IEEE International Conference on Computing, Communication and Automation (ICCCA2017), ISBN: 978-1-5090-6471-7/17/$31.00 ©2017 IEEE

[8] OMOGBAI OLEGHE, "Container Placement and Migration in Edge Computing: Concept and Scheduling Models",IEEE Access 2021

[9] Guisheng Fan1,2 , Liang Chen1 , Huiqun Yu1 , and Wei Qi1,"Multi-Objective Optimization of Container-Based Microservice Scheduling in Edge Computin", Computer Science and Information Systems 2020

[10] Ouafa Bentaleb1,2,3 · Adam S. Z. Belloum3 · Abderrazak Sebaa4,5 Aouaouche El-Maouhab,"Containerization technologies: taxonomies, applications and challenges", The Journal of Supercomputing 2021

[11] Salah, M. Jamal Zemerly, Chan YeobYeun, Mahmoud Al-Qutayri, Yousof Al-Hammadi," Performance Comparison between Container-based and VM-based Services", 978-1-5090-3672-1/17/$31.00 ©2017 IEEE

[12] Sheng Wang, Zhijun Ding, Senior Member, IEEE, ChangjunJiang,"Elastic Scheduling for Microservice Applications in Clouds", IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS:1045-92(19)-2020

[13] JunmiBo An, Donggang Cao, Xiangqun Chen," Comparing Container-based Microservices and Workspace as a Service:Which One to Choose?", 2018 IEEE Symposium on Service-Oriented System Engineering", 0-7695-6394-5/18/$31.00 ©2018 IEEE

[14] Docker Containers and VMs.https://www.weave.works/blog/a-practical-guide-to-choosing-between-docker-containers-and-vms.Accessed January 2022

[15] Microservice vs Monolithic. https://www.suse.com/c/rancher_blog/microservices-vs-monolithic-architectures/. Accessed June 2023

[16] Chris Richardson of Eventuate,Building Microservices, https://www.nginx.com/blog/building-microservices-using-an-api-gateway/. Accessed: December 2023

[17] Tarek Menouer,PatriceDarmon ,"New Scheduling Strategy based on Multi-CriteriaDecision Algorithm" 2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), 2377-5750/19/$31.00 ©2019 IEEE

[18] Rajkumar Buyya1, Maria A. Rodriguez1, Adel NadjaranToosi, Jaeman Park," Cost-Efficient Orchestration of Containers in Clouds: A Vision, Architectural Elements, and Future Directions", Cloud Computing and Distributed Systems (CLOUDS) Lab

[19] Claus Pahl∗ and Antonio Brogi‡ and Jacopo Soldani‡ and PooyanJamshidi† "Cloud Container Technologies: a State-of-the-Art Review", IEEE Transactions on Cloud computing 2017

[20] Maria A. Rodriguez RajkumarBuyya," Container-based cluster orchestration systems: A taxonomy and future directions", © 2018 John Wiley & Sons, Ltd

[21] Abdul Saboor 1,* ,MohdFadzil Hassan 2, Containerized Microservices Orchestration and Provisioning in Cloud Computing: A Conceptual Framework and Future Perspectives", Appl. Sci. 2022, 12, 5793

[22] Maria Fazio,AntonioCelest,RajivRanjan,LydiaChen,Chang Liu," Open Issues in Scheduling Microservices in the Cloud" ,IEEE cloud computing , 2325-6095/16/$33.00 © 2016 IEEE September/October 2016

[23] Kaewkasi , KornrathakChuenmuneewong," Improvement of Container Scheduling for Docker using Ant Colony Optimization" ,IEEE cloud computing , 978-1-4673-9077/4/17/ © 2017 IEEE October 2017

[24] Hamid MohammadiFard ,"Dynamic Multi-objective Scheduling of Microservices in the Cloud" 2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)

[25] XUEHUA ZHAO," Microservice Based Computational Offloading Framework and Cost Efficient Task Scheduling Algorithm in Heterogeneous Fog Cloud Network",IEEE Access VOLUME 8, 2020

[26] Tarek Menouer Patrice Darmon," New Scheduling Strategy based on Multi-Criteria Decision Algorithm" , 2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing ,2377-5750/19/$31.00 ©2019 IEEE DOI 10.1109/PDP.2019.00022

[27] Mehmet Söylemez1 ,BedirTekinerdogan 2,* and AyçaKolukısaTarhan 1," Challenges and Solution Directions of Microservice Architectures: A Systematic Literature Review", Appl. Sci. 2022, 12, 5507. https://doi.org/10.3390/app12115507

[28] Z. Zhong, "Machine Learning-based Orchestration of Containers: A Taxonomy and Future Directions" ACM Computing Surveys, Vol. 54, No. 10s, Article 217. Publication date: September 2022

[29] NandanJha,Saurabh Garg, Prem Prakash Jayaraman,RajkumarBuyya,RajivRanjanCelest, " A Holistic Evaluation of Docker Containers for Interfering Microservices" , 2018 IEEE International Conference on Services Computing,2474-2473/18/$31.00 ©2018 IEEEDOI 10.1109/SCC.2018.00012

[30] MingchangWei,Yang Yu,," A Container Scheduling Strategy Based on Machine Learning in Microservice Architecture", 2019 IEEE International Conference on Services Computing (SCC), 2474-2473/19/$31.00 ©2019 IEEEDOI 10.1109/SCC.2019.00023

[31] Ye Wu ,HaopengChen,"ABP Scheduler :Speeding up service spread in Docker swarm" , 2017 IEEE International journal on Parallel and distributed Processing with Applications", 0-7695-6329-5/17 2017 IEEE

[32] Junming Ma, Bo An, Donggang Cao, Xiangqun Chen," Comparing Container-based Microservices and Workspace as a Service:Which One to Choose?", 2018 IEEE Symposium on Service-Oriented System Engineering", 0-7695-6394-5/18/$31.00 ©2018 IEEE

[33] Christophe C´ erin, Tarek Menouer, WalidSaad and Wiem Ben AbdallahChris," A New Docker Swarm Scheduling Strategy", 2017 IEEE 7th International Symposium on Cloud and Service Computing, 978-0-7695-6328-2/17 $31.00 © 2017 IEEE

[34] Ayman Noor∗†, DevkiNandanJha∗, Karan Mitra‡, Prem Prakash Jayaraman§, Arthur Souza¶, Rajiv Ranjan∗andSchahramDustdar, "A framework for monitoring microservice-oriented cloud applications in heterogeneous virtualization environments", 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), 2159-6190/19/$31.00 ©2019 IEEE

[35] Michel Gokan Khan, Member, IEEE, Javid Taheri, Senior Member, IEEE,"PerfSim: A Performance Simulator for Cloud Native Microservice Chains:, arXiv:2103.08983v2 [2017]

[36] Guo ,"A Container Scheduling Strategy Based on Neighborhood Division in Micro Service" 978-1-5386-3416-5/18/$31.00 ©2018 IEEE

[37] Raj Gunasekaran,"Characterizing Bottlenecks in

Scheduling Microservices on Serverless Platforms", 2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)

[38] Rohan Mahapatra,Exploring Efficient ML-based Scheduler for Microservices in Heterogeneous Clusters

[39] 1st Guozhi Li," Microservices: architecture, container, and challenges", 2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C)

[40] Qilong Li, "Multi-Algorithm Collaboration Scheduling Strategy for Docker Container", 2017 International Conference on Computer Systems, Electronics and Control (ICCSEC)

[41] Christina Terese Joseph K. Chandrasekaran," Straddling the crevasse: A review of microservice software architecture foundations and recent advancements", DOI: 10.1002/spe.2729

[42] Yanjun Shi *, YijiaGuo, LinglingLv and Keshuai Zhang," An Efficient Resource Scheduling Strategy for V2X Microservice Deployment in Edge Servers", Future Internet 2020, 12, 172; doi:10.3390/fi12100172

[43] 1Pratham Jangra, 2Anuttam Anand, 3Dr. Amit Kumar Tyagi, "Survey on Container Systems and Their Efficient Orchestration Algorithms", International Journal of Creative Research Thoughts (IJCRT)

[44] Ebook for .Net microservice for containerized applications, Microsoft book

[45] Yiren Li1,2, Tieke Li1 , Pei Shen2 , Liang Hao2," Sim-DRS: a similarity-based dynamic resource scheduling algorithm for microservice-based web systems", PeerJComput. Sci. 7:e824 DOI 10.7717/peerj-cs.824

[46] Vishal Rao1 , Vishnu Singh1 "Scheduling Microservice Containers on Large Core Machines through Placement and Coalescing"

Original/Research Article

Advances in Engineering and Intelligence Systems

e-ISSN:2821-0263
Volume 05, Issue 01, 2026

Journal Web Page: https://aeis.bilijipub.com

Multi-Criteria Deep Reinforcement Learning Energy-Aware and QoS-Driven Scheduling for Containerized Microservices in Docker Swarm Clusters

Anil A. Prajapati [1] *, Manish M. Patel [2]

[1] *Scholar, Gujarat Technological University, Ahmedabad, Gujarat, India*
[2] *Professor, Sankalchand Patel College of Engineering, Sankalchand Patel University, Visnagar, Gujarat, India*

Highlights

- Proposing a multi-criteria DRL scheduler for containerized microservices.
- Integrating resource utilization and QoS metrics for adaptive node selection.
- Training Deep Q-Network policies using real-time cluster data.
- Reducing CPU usage by up to 25% and improving memory efficiency.
- Enhancing QoS by lowering latency and stabilizing response time behavior.

Article Info

Received: 30 December 2025
Received in revised: 17 February 2026
Accepted: 14 March 2026
Available online: 31 March 2026

Keywords

Container
Microservices
Docker Swarm
Reinforcement Learning
DQN

Abstract

Efficient scheduling of containerized microservices is essential to ensure performance, scalability, and quality of service in cloud native environments. Docker Swarm comes with three container scheduling strategies by default: Spread, Binpack, and Random. These policies are based on static heuristics and lack awareness of dynamic resource conditions and application-level performance metrics. As a result, they often lead to resource imbalance and degraded performance under cumulative and chronic workloads. This paper proposes MCSCM-RL (Multicriteria Scheduling of Containerized Microservices-Reinforcement Learning), a scheduler evaluated on Docker Swarm clusters. The new scheduling approach integrates container-level metrics, such as CPU and memory utilization, and container counts, with application-level QoS metrics, such as average startup time and average response time. A three-node Docker Swarm cluster with one manager and two worker nodes are used to gather real-time data, which is then used to continuously train the best node selection policies using a Deep Q-Network. Extensive experimental evaluation is conducted under both clean and cumulative deployment scenarios for containerised microservices. The results show that the proposed scheduler consistently outperforms default Docker Swarm schedulers across all evaluated metrics. Against the reference container scheduling strategies, the proposed scheduler can attain a reduction of CPU utilization by up to 25%, improve memory utilization, have lower startup latency, and provide consistent response time behavior. The proposed scheduler avoids the instability associated with the Random scheduler while effectively mitigating resource peaks commonly observed with Spread and Binpack schedulers. The proposed scheduler offers an adaptive, quality of service-aware, and energy-efficient solution for scheduling of containerized microservices in Docker Swarm architecture. Future work will focus on applying predictive workload modelling, extending to larger heterogeneous clusters or Kubernetes.

1. Introduction

Microservice architecture is a software development approach that splits the entire application into many small, independent, and loosely connected service units [1]. Each microservice is a distinct business function and communicates with other services via lightweight protocols, primarily RESTful APIs or message queues. Unlike monolithic architectures, in which all features are combined into a single deployment unit, microservices enable modularity, independent scaling, and continuous delivery [2,3]. Internally, this architectural pattern perfectly fits cloud computing as each service can be

* Corresponding Author: Anil A. Prajapati
Email: anil.apit18@gmail.com

https://doi.org/10.22034/aeis.2026.568553.1405

deployed, scaled, and updated independently without the risk of the changes affecting other parts of the system. Microservices can be scaled horizontally based on the load behaviours. Besides, they can be developed in different programming languages or with different frameworks, based on the functionality needs. Being fault-isolated, microservices do not cause failures to propagate to the entire system. The fast growth of the cloud environments, including microservices-based architectures, has led to a very big change in the software development, application deployment, and their scaling as it is known [3,4].

Containerization has become the backbone of cloud-native computing, which basically means the building of applications that are containerized together with all the things necessary to be able to run in an extremely minimal and portable manner [4,5]. Docker and similar technologies provide better isolation, quicker startup times, and a smaller footprint than the usual virtual machines [6]. This is why containers are very well adapted to microservice architectures, where the services are conceived, deployed, and scaled independently. Moreover, microservices go hand in hand with containerization. A container is used to package every microservice so as to have the best use of resources, quick start-up, and a standard platform for all the environments.

Container orchestration platforms oversee the life-cycle of containerized microservices. For instance, in the cloud, container orchestration platforms like Docker Swarm and Kubernetes not only manage container deployment manually but also automate such operations, including scaling and networking [7]. When containers are coordinated through platforms such as Docker Swarm that offer very lightweight container isolation, portability, and fast deployment, all these qualities are quite suitable for the new and modern applications. In distributed systems, container scheduling, i.e., selecting the appropriate node to run a newly created container among other nodes, is the major factor in achieving top resource utilization, premium service quality, and cheap running of operations [7,8]. The usage of containers to deploy microservices, like with Docker Swarm, has become the norm for the development of scalable, maintainable, and portable applications [5,6]. Each microservice operates inside its own container, with different and even changing resource demands.

Figuring out the right spot for every container is a problem called container scheduling. Making sure the system runs smoothly, is dependable, and efficient is a must [5,8]. Scheduling containers in huge cloud-native environments is not only really complicated but also involves optimizing multiple objectives at the same time. In large-scale deployments, especially those with dynamic workloads, intelligent scheduling becomes critical to balance Quality of Service (QoS) requirements that directly affect response time and availability with operational efficiency for latency-sensitive applications [5,6,8]. However, creating an effective container scheduling plan is really a lot more than a mere puzzle. Cutting down on energy use is still the main theme in reducing the operating costs [7,8].

There are tremendous advancements in container orchestration and scheduling strategies, but several limitations remain in place in existing container scheduling strategies. First, existing scheduling strategies, such as, Spread, and Random in Docker Swarm, or predicate–priority scheduling in Kubernetes are static and rule-based [5,6]. These scheduling approaches are single-criteria, reactive in nature, and effective in basic deployments. Besides, they do not implement a changing and multi-dimensional decision-making based on the current real-time system metrics, and also do not change through the adaptive learning from the past scheduling that they perform. As a result, these schedulers may decrease resource efficiency and application performance [9]. Second, existing learning-based schedulers focus only on resource utilization metrics (e.g., CPU or memory utilization) without integration of application-level QoS indicators such as startup time and response time. Third, the lightweight container orchestration frameworks like Docker Swarm are comparatively unexplored. Many reinforcements learning-based approaches are tested mostly in Kubernetes environments and may require complex integration mechanisms.

On the other hand, scheduling methods with learning at their base allow for making decisions in an adaptable way by constantly exchanging with the environment and updating scheduling rules according to the system states that have been seen. By using learning-based schedulers, it is possible to change work pattern following and to work towards maximizing a system's efficiency and robustness over a longer period. That is why, the incorporation of ML methods into container scheduling becomes a must in coping with the scalability and adaptability problems of present-day orchestration platforms. Machine learning can identify trends in historical data and also can react to changing circumstances instantly [9]. Regarding container scheduling, use of machine learning has been considered for prediction of resource requirements, workflow categorization, decision making with multiple criteria support, and scheduling with consideration to energy consumption [8-10].

Existing research has explored machine learning for scheduling in large-scale cloud systems, but Docker Swarm specific reinforcement learning (RL) approaches remain scarce [10].Furthermore, few studies consider multi-criteria scheduling that simultaneously optimizes

CPU usage, memory usage, and energy consumption [9,10].This gap is significant because containerized microservice deployments increasingly require energy-aware and application-aware scheduling policies that can dynamically balance microservice performance and energy efficiency.

These limitations highlight the need for a multi-criterion, and learning-based scheduler. The scheduler, which primarily determines the optimal placement of containers guided by metrics such as CPU/memory usage, number of containers, power consumption, startup and response times, is capable of adjusting to change in workload on-the-fly. To address these limitations, this paper proposes MCSCM-RL scheduler, a multi-criteria Deep Q-Network based scheduling framework that combines both resource-level and QoS-level metrics into a unified decision-making process. The proposed scheduler, unlike the current scheduling strategies, changes itself to the system changes regularly by reinforcement learning and as a result, load balancing is better, startup latency is less and there is more robustness even when deployment had been done cumulatively.

The major contributions of this paper are as follows:

1. **Multi-Criteria RL-Based Scheduling Framework**

A DQN-based container scheduling framework, MCSCM-RL, was proposed, which combines both resource-level metrics (CPU utilization, memory utilization, container count) and application-level metrics (startup time and response time) into a unified decision-making state representation. Unlike traditional heuristics schedulers that rely only on resource thresholds, the proposed scheduler optimizes system efficiency and service performance.

2. **Adaptation to changing Workload**

The proposed scheduler continuously updates the scheduling policy through reinforcement learning, enabling adaptation to changing workloads and resource conditions. On the other hand, Docker swarm default scheduling strategies like Spread, Binpack, and Random, do not possess feedback-driven modification mechanisms.

3. **Energy and QoS Aware Decision mechanism**

The framework integrates multi-criteria evaluation principles into the RL state design. It increases the overall robustness of the system in scenarios of cumulative container deployment by making good trade-offs between the application performance and resource utilization that are well – informed.

4. **Extensive Comparative Evaluation**

The effectiveness of MCSCM-RL scheduler is validated through extensive experiments on a Docker Swarm cluster under both clean and cumulative deployment scenarios. The proposed scheduler is compared with default Docker Swarm schedulers (Random, Spread, Binpack) using multiple performance metrics.

5. **Consideration of Scalability**

Although the framework is evaluated on Docker Swarm. The framework is allowed to be extended to other container orchestration systems in future, such Kubernetes, through custom scheduling mechanisms

These contributions aim to bridge the current gap in multi-objective, learning based scheduling for Docker Swarm, providing both a practical deployment and a concrete performance study that might direct future research in adaptive scheduling of containerized microservices.

The rest of this manuscript is organized as follows. Section-2 reviews existing literature on container orchestration systems, with emphasis on Docker Swarm–based scheduling and recent advances in containerized microservice scheduling, including reinforcement learning approaches in Kubernetes environments. Section-3 discusses the problem addressed by this research, defines the aims of the system, and presents the system architecture, application model, and criteria for assessment. Section-4 describes the implementation and system integration of the proposed MCSCM-RL framework, including an overview of the scheduling algorithm, the heuristic information and metric formulations, and the complete MCSCM-RL scheduling procedure. Section-5 describes the setup of the experiments, the deployment scenarios, and the evaluation methods for the new approach. Section 6 presents and analyzes the results obtained under clean and cumulative deployment conditions, along with a comparative assessment across all scheduling strategies. Finally, Section-7 concludes the paper and highlights potential directions for future research.

2. Related Works

This section reviews prior contributions across three key areas: container orchestration systems, container scheduling in Docker Swarm, and recent advances in machine learning and reinforcement learning approaches for scheduling of containerized microservice. First, established container orchestration platforms were examined. Then it focused specifically on Docker Swarm, summarizing its native scheduling policies. Finally, the Kubernetes-based work that applies machine learning (ML) and reinforcement learning (RL) to container scheduling and microservice orchestration was discussed, emphasizing the limitations of heuristic and rule-based schedulers and motivating the need for adaptive, data-driven decision-making frameworks.

2.1. Containers Orchestration Systems

Containers represent a lightweight and versatile execution environment that isolates different applications from one another while the underlying operating system kernel is still shared. This level of isolation facilitates deployment, ensures that portable devices can work across diverse infrastructures, and makes testing pipelines easier [3,5,6]. A number of studies have revealed that running microservices inside containers only results in a very slight performance overhead, whereas the efficiency of resource utilization is notably improved when compared to traditional virtualization methods. Moreover, a container can start up quicker than a virtual machine, thus making them suitable for latency sensitive and dynamic applications [10,11]. Due to the rapid increase in the use of containers, their popularity in large-scale application deployment is increasing and thus the demand for container orchestration platforms has emerged [11]. Essentially, a container orchestration platform is a platform that automates the deployment, scaling, and management of containerized microservices over the cluster. Additionally, the scheduling mechanisms of these platforms are designed to optimize the usage, to ensure service continuity without any interruption, and to keep fault tolerance [12]. Actually, container orchestration platforms are essential to running cloud-native applications that are both dynamic and resource limited (i. e., running the application without interruptions and with minimum faults) [13].

A few modern container orchestration systems [13,14] were assessed in terms of design, scheduling policies, workload support, and management capabilities. According to the literature, Kubernetes and its contemporaries have advanced scalability, resilience, and a highly mature ecosystem, making large-scale cloud-native deployments quite viable [15]. Lightweight orchestrators like Docker Swarm, for example, are very simple to use, require little configuration, and deploy very quickly, which are features that may make them a very good fit for small- to medium-scale deployments [15]. Several other systems, such as Apache Mesos, Google Aurora, and Hadoop YARN are only available for special-purpose big data analytics, batch processing, and long-running distributed workloads, respectively. But they don't have broad adoption seen in Kubernetes-driven ecosystems [16]. Given these considerations, Docker Swarm is adopted in this study as a practical and transparent experimental platform. Along with that, its lightweight architecture and simple scheduling behavior are perfect for not only creating but also combining and thoroughly testing the proposed multi-criteria, reinforcement-learning-based scheduling framework.

2.2. Container Scheduling Strategies in Docker Swarm

Among all container orchestration frameworks, Docker Swarm remains one of the most integrated and minimalist container orchestration platforms for microservices. It provides a built-in clustering capability to transform a group of Docker engines into a single virtual system for container orchestration and management [17]. Docker Swarm is Docker's built-in tool for container orchestration. It is a tool that allows multiple Docker hosts (nodes) to be managed and coordinated as one cluster [18]. It is capable of automatically deploying, scaling, and managing containerized applications, which means that it can provide high availability and efficient use of resources. Docker Swarm comes with three different scheduling options. While Kubernetes dominates large-scale orchestration, Docker Swarm remains attractive due to its simplicity, native. Docker Swarm follows a manager-worker model. A Docker Swarm cluster consists of two primary node types where manager nodes control and manage the cluster's state and worker nodes execute containers (services) as scheduled by the manager nodes [18,19].

These options are so established that they serve as the baselines for evaluating new scheduling strategies. These are heuristic based approaches that aim to simplify microservice placement decisions [15,17,18].

- **Binpack** schedules containers onto the node with the least available resources, effectively packing containers into fewer nodes to reduce fragmentation. While this can minimize the number of active nodes, it often results in performance bottlenecks under high load.
- **Spread** balances containers across nodes to distribute load more evenly, reducing the likelihood of hot-spots but increasing the number of active nodes and overall energy consumption.
- **Random** assigns containers to random eligible nodes, which provides a simple baseline but lacks predictability and can cause temporary load imbalances.

Table 1 presents a comparative summary of Docker Swarm's default schedulers. The comparison shows the differences in how resource-awareness, adaptivity, and QoS considerations are handled.

Table 1. Strengths and Limitations of Docker Swarm scheduling strategies.

Scheduler	Resource Awareness	QoS Awareness	Adaptivity	Load Stability	Scalability
Spread	Partial	No	No	Moderate	Limited
Binpack	Partial	No	No	Low (hotspots)	Limited
Random	None	No	No	Low (variable)	Poor

These limitations of default schedulers motivate the need for adaptive, multi-criteria learning-based schedulers, which are capable of balancing resource efficiency and application-level performance needs.

2.3. Recent Advances in Scheduling for Containerized Microservices

Heuristic and optimization-based scheduling have been widely explored for scheduling containerized microservices in cloud environments. Usually, these schedulers use metaheuristic algorithms and rule-based techniques. They use genetic algorithms, particle swarm optimization, ant colony optimization, or multi-objective optimization models for better resource utilization, latency, and energy efficiency.

Traditional heuristic based schedulers (e.g., Kubernetes and Docker Swarm defaults) often lack to adapt to fluctuating workloads and the complex interdependencies among microservices, leading to suboptimal placement decisions. Consequently, recent research (2020–2025) published in leading journals has explored advanced optimization techniques including deep learning, reinforcement learning, and hybrid multi-objective algorithms to improve container placement efficiency, scalability, and Quality of Service (QoS).The following subsection reviews these contributions, emphasizing their core methodologies, optimization goals, and limitations relative to real time multi-criteria scheduling of containerized microservice in cloud environments.

Shi et al. [20] presented a multi-objective model for containerized microservice deployment in edge computing that uses multiple fitness genetic algorithms to optimize resource utilization, load balancing, and inter-microservice dependencies. In order to minimize distance vehicle application and balance utilization, the work integrates dynamic container migration during crossover and mutation. Simulations on Container CloudSim validated improvements in system performance over Kubernetes' default.

In order to maximize completion time and resource costs in hybrid clouds, Lin et al. [21] modelled microservice workflows as directed acyclic graphs (DAGs) and suggested an ACO based algorithm. Their approach improves pheromone update techniques for multi-objective heuristics. Comparative evaluations against genetic algorithms and particle swarm methods showed better convergence and reduced costs in private and public cloud integration. Multi-objective optimization frameworks have been extensively applied to container scheduling in edge environments. In order to reduce network latency between microservices, improve application stability, and achieve cluster load balancing while maintaining node resource capacities, Fan et al. [22] developed container-based microservice scheduling as a multi-objective problem in edge computing. They presented the particle swarm optimization (PSO)-based Latency, Reliability, and Load Balancing Aware Scheduling (LRLBAS) algorithm, which adaptively reserves ineffective particles and separates fitness functions from restrictions.

The integration of intelligent scheduling algorithms to improve resource efficiency and service performance has been the focus in recent studies. Munisami and Vignesh [23] presented DSTS, a hybrid deep learning based dynamic task scheduler for container-based cloud systems. It shows greater scalability but relying on high quality training data. Rao and Li [24] presented energy aware microservice scheduling in Kubernetes clouds, showing considerable power savings but limited adaptability to multi-criteria QoS. While genetic methods frequently encounter long convergence times, Liu et al. [25] developed a multi-task genetic programming strategy for online multi-objective placement in heterogeneous clusters.

A functionality-aware container offloading technique was developed by McEnery and Lee [26] for edge infrastructures, which improved latency performance but required extensive application-level annotations. In a thorough mapping investigation of AI approaches in the microservices lifecycle, Moreschini et al. [27] identified RL and meta-learning as new areas of research. Research momentum in prediction and learning based orchestration keeps growing. An AI-based predictive container orchestration architecture that improves placement stability but lacks real-time learning mechanisms was identified by Al Qassem et al. [28].

A pattern-learning scheduler for microservice workflows was proposed by Zhang et al. [29], which improved throughput but mainly focused on workflow-structured applications. For resource-constrained edge computing, Shi et al. [30] presented a graph transformer improved DQN model, which achieved excellent flexibility at the expense of significant computational overhead. For adaptive microservice resource allocation, Wang et al. [31] combined meta learning with super heuristic algorithms, which provided quick task-specific learning but necessitated intricate model building. Finally, Zhou et al. [32] provided an extensive review of AI driven job scheduling techniques, underscoring the industry wide transition toward RL-based and QoS-aware orchestration frameworks.

2.4. Reinforcement Learning–Based Scheduling in Container Orchestrations

Kubernetes has emerged as the dominant platform for large-scale microservice management. The default Kubernetes scheduler relies primarily on predefined scoring and filtering mechanisms, and struggles with dynamic workloads. In result, it gives suboptimal

resource utilization. Recent years (2023–2025) have seen a significant shift from static heuristics to ML and reinforcement learning (RL) driven scheduling to address the challenges of resource heterogeneity, changing workload, and inter-service dependencies.

Rao and Li. [33] presented an energy-aware microservice scheduling approach for Kubernetes using an Improved Sparrow Search Algorithm (ISSA) to prioritize pod placement based on communication patterns, CPU demand, and heartbeat overhead. It shows more than 5% energy reduction over default Kubernetes schedulers. Similarly, Turin et al. [34] proposed a predictive CPU and Memory resource model, which provides calibrated workload estimation under changing execution intensities. Marchese and Tomarchio [35] extended this work with a load-aware Kubernetes orchestrator, which reduces end-to-end latency by up to 20% under dynamically balancing workloads across shared clusters.

Several RL-driven schedulers further advance Kubernetes' capabilities. Bai et al. [36] presented REACH, a Soft Actor Critic (SAC) based scheduler that reduces microservice end-to-end latency by 7.9–10% through adaptive rescheduling in cloud-edge systems. Cai et al. [37] developed a multi-dimensional RL-based Kubernetes scheduler capable of adjusting weights according to CPU, GPU, and I/O demands, achieving up to 36% faster response times over the default scheduler. Jian et al. [38] introduced DRS, a DRL enhanced Kubernetes scheduler designed as a Markov Decision Process (MDP), improving resource utilization by 27.29% and reducing load imbalance by 2.9%.

Kallel et al. [39] presented Multi agent RL approaches such as DRL4HFC to optimize microservice deployments in hybrid Fog/Cloud environments, learning resource task interactions to minimize execution time and communication overhead. Senjab et al. [40], classified Kubernetes scheduling advancements into generic, multi-objective, and ML based strategies. It highlights RL's potential to improve scheduling under heterogeneity and dynamic workload. Santos et al. [41] highlights near optimal latency cost trade-offs with deep RL in federated Kubernetes clusters. Alongside the rapid advancements driven by Kubernetes, numerous studies have examined metaheuristic and evolutionary approaches for optimizing containerized microservices.

Because Docker Swarm has a simpler architecture and a smaller number of scheduler extension points, research using reinforcement learning for Docker Swarm is quite limited, as opposed to Kubernetes. However, Docker Swarm offers stable performance under high load and lower orchestration overhead, making it attractive for lightweight RL experimentation and small-scale microservice deployments. These studies show that RL can effectively adapt scheduling decisions based on dynamic resource utilization and application performance [33,42].

Though a great deal of research investigating reinforcement learning based scheduling in Kubernetes has been done, research in the parallel area of Docker Swarm is quite limited. Nevertheless, Docker Swarm constantly offers practical advantages for controlled experimentation, including low orchestration overhead and stable behavior in high cluster saturation levels. It makes it well suited for small-scale microservice deployments and RL-based scheduling.

Collectively, these studies demonstrate significant progress but also reveal persistent gaps: (i) limited integration of real-time metrics into the state model, (ii) inadequate support for multi-criteria scheduling that jointly considers resource usage and application-level QoS, and (iii) challenges in deploying RL models on lightweight container orchestration platforms such as Docker Swarm. These gaps motivate the development of the proposed MCSCM-RL scheduler, which combines real-time multi-criteria analysis (CPU, memory, container count, startup time, response time) with a reinforcement learning framework to enable adaptive, resource-efficient, and application-aware container deployment in small-scale cloud clusters.

The present work focuses on Docker Swarm and proposes the MCSCM-RL scheduler, a multi-criterion, resource and application aware RL based scheduler. The scheduler that has been presented is made for small but still active Swarm clusters and it tunes five critical performance metrics: CPU usage, memory usage, number of containers, startup time and response time. Therefore, MCSCM-RL is light, interpretable, and can be immediately implemented in situations where the complexity of Kubernetes is not required. Overall, MCSCM-RL framework demonstrates that Docker Swarm can be enhanced with intelligent, multi-criteria RL policies to deliver effective performance in resource and performance constrained scenarios.

3. Problem Formulation and System Objectives

The rapid adoption of microservice-based architectures in containerized environments such as Docker Swarm has created new challenges in efficient scheduling across distributed nodes. Hence, a smart, adaptable, and multi-criteria scheduling method is required that can keep up with the changes in the system state and take the best placement decisions on the spot (real-time) by constantly learning. This section outlines the system architecture of the proposed intelligent scheduler, followed by the application model used to characterize microservice behavior and QoS requirements. It concludes with the evaluation metrics

that measure resource efficiency, performance, and load-balancing effectiveness across the Docker Swarm cluster.

This research proposes the Multi-Criteria Scheduling of Containerized Microservices using Reinforcement Learning (MCSCM-RL) strategy. MCSCM-RL integrates both resource-level metrics (CPU%, Memory%, container count) and application-level QoS metrics (average startup time and average response time) within a Deep Q-Network (DQN) learning framework. This scheduler dynamically learns optimal node-selection policies using continuous feedback and reward-based adaptation, leveraging relative closeness for state representation. By getting continual knowledge of environmental feedback, the scheduler is able and willing to dynamically choose the best node for container deployment so that three things are achieved in the best way: resource utilization is well balanced, latency is minimized and at the same time, the overall system efficiency is enhanced.

3.1.System Architecture

The system that was proposed was run on a Docker Swarm cluster with three nodes: one Manager node and two Worker nodes (Worker-1 and Worker-2). Each node operates as an independent virtual machine (VM) running on an Ubuntu system hosted on Windows System, with Docker Swarm configured to orchestrate containerized microservices across the cluster. The Manager Node manages the cluster, keeps track of the global states, performs resource monitoring, and runs the scheduling strategies such as Random Spread Binpack, as well as the new MCSCM-RL scheduler. The Worker Nodes are the agents for executing the tasks and they will have microservice containers deployed there as per the node selection by the scheduler.

The MCSCM-RL scheduler at runtime makes very accurate decisions on the best node for microservice deployment with the help of the Deep Q-Network (DQN) based reinforcement learning model. It selects the node in a multi-criteria manner by considering state parameters like CPU usage, memory usage, no. of containers, average startup time, and average response time, along with the relative closeness ranking. By the way, Fig. 1 shows the architectural design of the proposed system in layers.

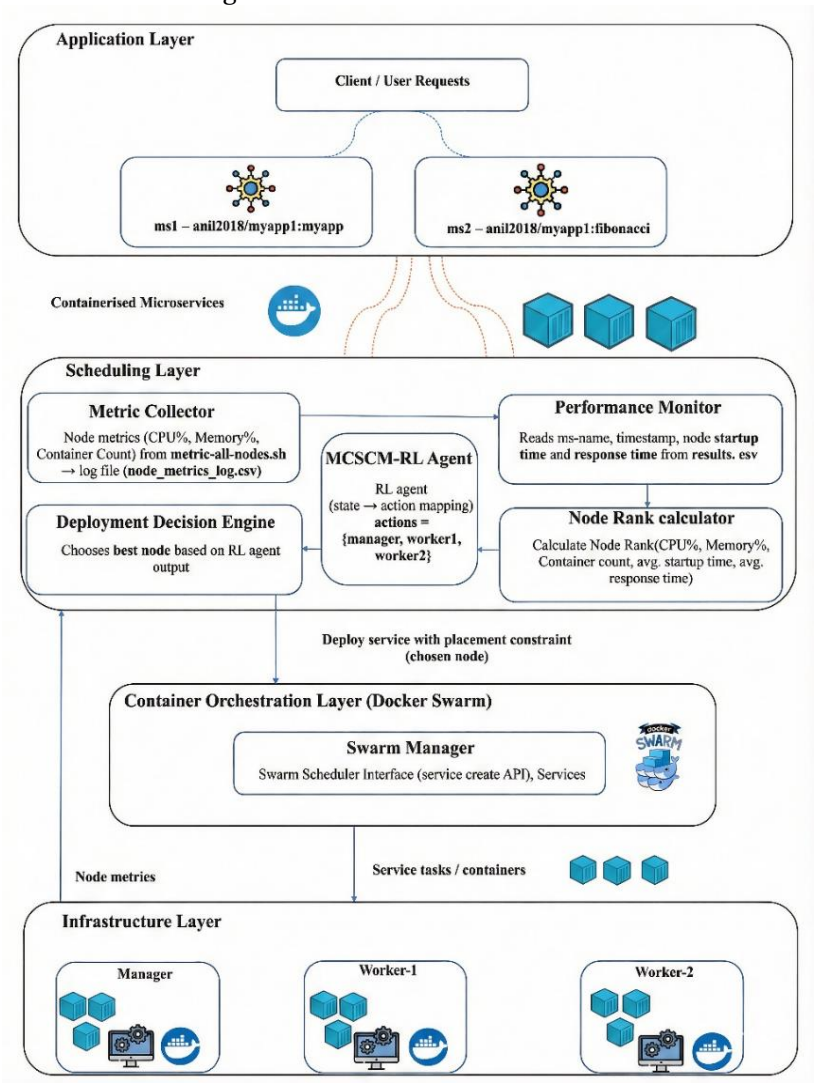


Fig. 1. System architecture of the proposed MCSCM-RL based container scheduling framework on a Docker Swarm cluster.

• The Docker Swarm manager continues to handle service definitions, cluster state, and communication with worker nodes worker-1 and worker-2.

• The MCSCM-RL scheduler operates as a decision-making layer between the user's docker service create requests and Swarm's task placement logic.

• This scheduler continuously monitors real-time node metrics, ranks nodes using a multi-criteria decision-making (MCSCM) approach, and feeds the ranked states into a Deep Q-Network (DQN) agent to select the optimal target node for the next container deployment.

• Final placement is enforced via Docker API constraints to pin the container to the chosen node.

3.2. Application Model

The Swarm cluster is deployed on a Windows host using three Ubuntu virtual machines and each node runs the Docker Engine and participates in the container orchestration of the cluster.

Two microservices were containerized and published on Docker Hub. All container images are hosted in the public Docker Hub repository **anil2018** for testing the default and MCSCM-RL scheduling. Table 2 presents the details of microservices used for evaluation.

Table 2. Microservices used in the experimental evaluation.

Microservice	Image (with tag)	Description
ms1	anil2018/myapp1: myapp	Stateless request-processing service (e.g., REST API)
ms2	anil2018/myapp1: fibonacci	CPU-intensive Fibonacci calculator (used as a synthetic workload)

These microservices were deployed repeatedly across different nodes under various scheduling strategies. Scheduling strategies evaluated in this study are detailed in Table 3. In each deployment, startup time and response time are recorded for clean deployment (no existing containers) and cumulative deployment (progressive accumulation of containers) environments.

Table 3. Scheduling strategy tested.

Scheduling Policy	Docker Swarm Built-in	Description
Spread	Yes	Maximizes distribution across nodes
Binpack	Yes	Packs containers onto the node with least remaining resources
Random	Yes (fallback)	Uniform random placement
MCSCM-RL	**Custom**	Reinforcement-Learning based multi-objective scheduler (resource, latency, prediction)

Table 4 outlines the deployment scenarios employed for performance comparison. The MCSCM-RL scheduler dynamically selects the target node using Deep Q-Network (DQN) reinforcement learning integrated with multi-criteria ranking, which considers CPU utilization, memory utilization, container count, average startup time, and average response time. This setup allows fair and repeatable comparison of the proposed MCSCM-RL with standard Swarm scheduling policies.

Table 4. Deployment scenarios evaluated.

Deployment Scenario	Description
Clean	Fresh Docker Swarm cluster with no pre-existing containers or workloads. All nodes start with identical resource availability.
Cumulative	Containerized microservices are deployed incrementally while the cluster already hosts running workloads, leading to increasing resource contention over time.

To analyse short-term performance and long-term scalability of scheduling strategies needs a distinction between clean deployment scenario, which presents an ideal environment with minimal interference, and the cumulative deployment scenario, which simulates realistic operational conditions, where successive deployments increase system load and resource contention.

3.3. Evaluation Metrics

The efficiency of the MCSCM-RL, must be validated using a rigorous set of performance metrics that quantify both resource efficiency and application Quality of Service (QoS). The selection of the metrics CPU utilization, memory utilization, container count, average startup time, and average response time is specifically intended to evaluate the MCSCM-RL algorithm's ability to outperform native Docker Swarm scheduling policies (Spread, BinPack, Random). The formal mathematical

equations of these metrics and Learning Dynamics of MCSCM-RL are provided in Section-4 (Algorithm Implementation).

CPU Utilization and Memory Utilization

CPU and memory utilization serve as fundamental metrics for assessing load balancing effectiveness and overall resource optimization within a heterogeneous Docker Swarm cluster. CPU utilization is an indicator of how much processing a node undergoes and can be used as an indirect way to reveal whether nodes are crowded or not used enough. On the other hand, memory utilization is a reflection of the application memory usage as well as the container density that together impact the service latency and responsiveness. These metrics directly address the intrinsic limitations of Docker Swarm's native scheduling strategies, such as the Spread policy, which largely rely on container counts and do not incorporate real-time CPU or memory availability into placement decisions.

To achieve balanced resource utilization is substantially more challenging than simply equalizing container distribution. The proposed MCSCM-RL scheduler is explicitly designed to be resource-aware, leveraging dynamic CPU and memory measurements as primary inputs to get optimal placement decisions. One of the major signs of its success lies in how much it lowers the overall resource imbalance across the cluster, thus making the load distribution more even when compared to the native Docker Swarm scheduling approaches.

Container Count

Container count is a very important precision point of measure for characterizing the default behavior of Docker Swarm's scheduling policies. It shows how workloads are spread across swarm nodes and reveals those where the heavy few could lead to the cluster's over- or under-utilization. The native Spread strategy in Docker Swarm attempts to equalize the number of running containers per node, whereas the BinPack strategy, by concentrating tasks on the least-loaded node, often increases the variance in container distribution. But focusing only on balancing the number of containers per node will not guarantee that the resource utilization is balanced, especially in the case of heterogeneous clusters where nodes have different CPUs and memory capacities. Measuring container count alongside CPU and memory utilization enables the identification of the clean imbalance. This distinction is important, as the proposed MCSCM-RL scheduler aims to achieve low variance in resource utilization even when the resulting container count distribution may be uneven. In this context, the behavior and limitations of the native scheduling policies are measured by the container count, and allows for an extensive comparison of MCSCM-RL's resource-aware enhancements.

Average Response Time

Mean response time is the simplest indicator of end-user Quality of Service (QoS) and therefore a key metric in the assessment of microservice performance. Given that containerized microservices in cloud setups are typically bound by stringent latency targets, a reduction in response time essentially implies an improved user experience. Poor or ill-conceived container placement may result in resource contention, which can then cause queuing delays and consequently increase the service latency. That is why one of the main goals behind this scheduler is to reduce response time. MCSCM-RL scheduler has its strength in lessening resource contention by means of intelligent, resource-aware placement choices. Through placing containers on nodes, which have good resource availability, the algorithm cuts down processing delays and hence, raises system responsiveness. A successful scheduling strategy is expected to produce a clearly lower average response time, especially for CPU-bound microservices.

Average Startup Time

Average startup time measures scheduling overhead and deployment efficiency, shows how quickly new services can be instantiated under varying resource conditions. Speedier service startup times are indispensable for allowing rapid scaling up and deployment efficiency, and are a basic characteristic of containerized microservice architectures. Apart from being a major factor in the productivity of the underlying resource selection strategy, this metric also plays a crucial role in determining the scheduler's responsiveness. In the MCSCM-RL framework, startup time is a major playing alongside runtime performance metrics like response time. Although the RL-based scheduler is designed to make intelligent placement decisions that reduce runtime contention. In highly dynamic cloud environments, a successful scheduling policy should improve runtime QoS while maintaining startup overhead at acceptable levels to maintain overall system responsiveness.

4. Proposed Scheduling Framework and Implementation

This section presents the complete implementation of the proposed scheduling framework. It includes an overview of the algorithmic workflow, the extraction and processing of heuristic node information and the formulation of QoS-aware resource metrics, DQN-Model Design and Learning Dynamics of MCSCM-RL.The mathematical definitions of CPU, memory, startup time, and response time are provided alongside the computation of the closeness coefficients used for multi-criteria ranking. This chapter describes the parts of

reinforcement learning, for example, the depiction of the state, the development of the reward, and the choice of action, and shows how they are merged with the MCSCM-RL framework that was suggested.

4.1.Workflow of Proposed Scheduling Algorithm

The proposed MCSCM-RL algorithm is designed for intelligent scheduling of containerized microservices across heterogeneous nodes in a Docker Swarm environment. Unlike traditional static scheduling strategies such as Random, Spread, and Binpack, which rely on predefined heuristics, MCSCM-RL employs a Deep Q-Network (DQN) based reinforcement learning agent to learn optimal scheduling decisions from experience and evolving system states. This merger harnesses multi-criteria decision-making (MCDM) and deep reinforcement learning (DRL) to optimally allocate resources, reduce response time and schedule the work in an energy-efficient way. The environment of a DRL agent is characterized by the combination of five major factors: the level of CPU utilization, the level of memory usage, the number of containers, the average startup time, and the average response time. State is basically a full picture of all the nodes at the swarm cluster during a particular moment. Both resource-level and application-level features of the containerized microservices are reflected in these measures. Before each microservice deployment, these metrics are normalized and compute a relative closeness score for each docker swarm node. These scores collectively form the state vector input to the DQN agent.

Using an ε-greedy policy, the DQN agent selects a Swarm node (action) for deploying the next microservice. The deployed microservice's startup time and response time are then measured to compute the reward, which reflects the efficiency and QoS performance of the scheduling decision. After deployment, the agent observes the updated cluster state, stores the experience tuple (*si, ai, ri, si+1*) in a replay buffer, and performs training through mini-batch gradient updates to refine its policy over time. By continuously learning from real-time feedback, the MCSCM-RL scheduler dynamically adapts to varying workloads, balances swarm node utilization, and minimizes response times of the microservice deployment. The final trained model encapsulates an intelligent policy capable of making energy-aware and performance aware scheduling decisions, outperforming conventional heuristics in terms of both efficiency and QoS compliance.

4.2. Heuristic information

This subsection defines the heuristic information used by the proposed scheduler, including the formulation of node-level resource metrics (CPU, memory, and container density) and QoS metrics (startup and response time). This section also describes how the closeness coefficient is calculated, which offers a way to rank cluster nodes using multiple criteria. Finally, the integration of this heuristic into the Deep Q-Network (DQN) model is described, outlining how the RL agent leverages the metric vector to perform intelligent container placement.

Node Metrics Calculation

To quantify node-level performance characteristics, the CPU utilization, memory utilization, container count, and estimated power consumption were computed and used as input features for the proposed scheduling framework.

i. The **CPU utilization** of a docker swarm node represents the proportion of time the CPU spends performing non-idle tasks.

$$U_{CPU} = 100 - C_{idle} \quad (1)$$

Where;

U_{CPU} denotes the CPU utilization (%)

C_{idle} is the CPU idle percentage obtained from the system monitor (***top*** command)

ii. The **memory utilization** represents the percentage of memory currently in use relative to the total available memory:

$$U_{Mem} = \frac{M_{used}}{M_{total}} \times 100 \quad (2)$$

Where;

U_{Mem} = memory utilization (%)

M_{used} = used memory (in MB),

M_{total} = total available memory (in MB)

The values are extracted using the ***free -m*** command.

iii. The **container count** represents the total number of active containers running on the node at a given time:

$$N_{cont} = count(running\ containers) \quad (3)$$

Where;

N_{cont} denotes the number of running containers obtained using the Docker command: "docker ps -q | wc -l"

QoS Metrics Calculation

To evaluate the quality of service (QoS) perceived by users, two performance indicators were measured for each microservice deployment — the average startup time and average response time.

Average Startup Time

Startup time means the length of time from when a container deployment is started to the time the microservice is up, running, and ready to serve requests. If $t_{start,i}$ and $t_{ready,i}$ denote the start and ready timestamps of the i^{th} container, respectively, then the **average startup time** across n deployments is defined as:

$$T_{\text{startup,avg}} = \frac{1}{n}\sum_{i=1}^{n}(t_{\text{ready},i} - t_{\text{start},i}) \quad (4)$$

Where;

$T_{startup,avg}$= average container startup time (in seconds),

$t_{start,i}$= timestamp when deployment of container i begins,

$t_{ready,i}$= timestamp when container i becomes ready,

n= total number of deployed containers or trials.

Average Response Time

Response time means the time duration between the moment a request is sent to a deployed microservice and the moment its response is received. If $t_{req,j}$and $t_{res,j}$denote the request and response timestamps for the j^{th}request, the average response time is given by:

$$T_{\text{response,avg}} = \frac{1}{m}\sum_{j=1}^{m}(t_{\text{res},j} - t_{\text{req},j}) \quad (5)$$

Where;

$T_{response,avg}$= average response time (in seconds),

$t_{req,j}$= timestamp when the request is sent,

$t_{res,j}$= timestamp when the response is received,

m= total number of requests measured.

Closeness function

The raw metric matrix X = [xij] is first subjected to min–max normalization to obtain the normalized matrix rij. These normalized values are then weighted to produce the weighted decision matrix vij. For each criterion, the ideal and anti-ideal points are identified, and the Euclidean separations S+ i and S– i are computed for every node. The relative closeness coefficient Ci, which is defined on the interval [0,1], combines the information of different criteria into one single numeric score that is then used to order the nodes from most to least suitable. The scheduler selects the node corresponding to the highest Ci, while the RL agent may incorporate the vector of closeness values St into its state representation. Fig. 2 illustrates the computational workflow of the proposed MCSCM-RL scheduling framework.

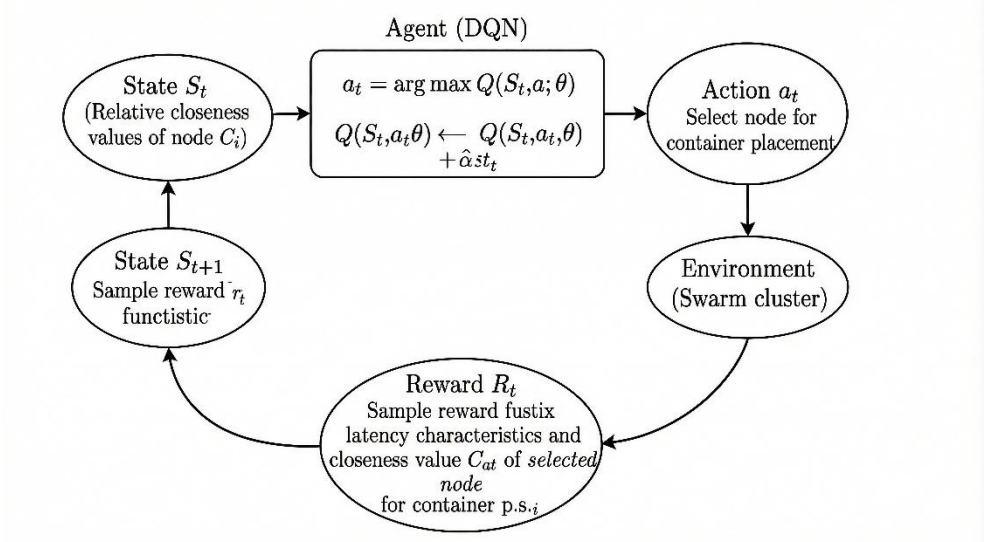


Fig. 2. Computational workflow.

4.3. DQN Model Design and Learning Dynamics of MCSCM-RL

This subsection presents the design of the Deep Q-Network (DQN) model and explains its learning dynamics. The formulation sets the state portrayal, action domain, and reward schema that are exploited to lead the node picking decisions.

RL State Design Rationale

The state representation of the proposed MCSCM-RL model is designed to catch resource availability and QoS of docker swarm cluster nodes. CPU utilization, memory usage, and container count depict the consumption of resources, whereas average startup time and average response time show the application's level of performance. Combining these performance metrics within a single state vector permits the DQN agent to develop intelligent scheduling policies that achieve high system efficiency and application QoS requirements. This unified state permits the scheduler to avoid scheduling decisions that are optimal with respect to resource utilization alone but give degraded application performance, which is a typical weakness of heuristic schedulers.

Action Space and Decision-Making

At every scheduling step, the action space of the DQN agent selects one swarm node from the available swarm cluster nodes for the container deployment. The Q-network predicts the potential reward for every candidate node considering the current environment situation, and the node displaying the highest Q-value will be chosen for the deployment, while using an -greedy approach for the selection of an action.

The ε-greedy(epsilon-greedy) strategy is selected for the swarm cluster node selection in the proposed MCSCM-RL scheduler for the container deployment. The ε-greedy policy balances exploration and exploitation by choosing a random node with probability ε and the node with the highest estimated Q-value with probability (1–ε). Taking this choice is inspired by its simplicity, low

computational overhead, and effectiveness in environments having limited action space, such as Docker Swarm clusters with a small number of nodes.

Additionally, three exploration–exploitation strategies were considered for the proposed work. First, Softmax exploration, which requires careful temperature tuning and it is sensitive to reward scaling, which disrupts stable learning under fluctuating workload conditions [43]. Second, Upper Confidence Bound (UCB) methods take relatively stationary reward distributions, an assumption that is not suitable in containerized environments with fluctuating resource needs [44]. Third, Thompson Sampling is inappropriate to integrate with value-based deep reinforcement learning [45]. On the other hand, ε-greedy offers predictable behavior, stable conjunction through ε decay, and minimal tuning requirements, making it suitable for container scheduling on docker swarm cluster [43]. On this basis, ε-greedy was selected ε-greedy, as the most practical and reliable exploration strategy for the proposed MCSCM-RL scheduler.

Learning and Adaptation to Dynamic Workloads

One of the main means by which DQN agent adapts to dynamic workloads is by continuously interacting with the cluster environment. After each container deployment, updated resource metrics and QoS metrics are collected and used to develop the next state. The reward indicates the impact of the scheduling decision on resource utilization and application performance. As cluster workload characteristics change over the time, accordingly reward and state transitions also change, prompting the agent to update Q-values. In that way, MCSCM-RL can regularly evaluate its own scheduling policy in accordance with the changes in resource contention, workload variations, and performance fluctuations through this ongoing interaction. In that way MCSCM-RL offers improved robustness compared to static container scheduling approaches.

Reinforcement Learning -DQN-based Container Placement

1. State Representation

At each decision step t, the environment (Docker Swarm cluster) is represented by a state vector S_t composed of the normalized performance metrics of all nodes:

$$S_t = [\, C_1,\, C_2,\, \dots,\, C_m \,] \tag{6}$$

or equivalently, when detailed per-node metrics are used,

$$S_t = [\, U_{CPU,1},\, U_{Mem,1},\, N_{cont,1},\, T_{startup,avg,1},\, T_{response,avg,1},\, \dots,\, U_{CPU,m},\, U_{Mem,m},\, N_{cont,m},\, T_{startup,avg,m},\, T_{response,avg,m} \,] \tag{7}$$

Where;

m= total number of nodes in the Swarm cluster

C_i= relative closeness value of node $i$obtained

2. Action Selection

The agent selects an action at ∈ {1,2, ..., m}, where each action corresponds to choosing a Swarm node Ni for container placement. An ε-greedy policy is used to balance exploration and exploitation:

$$a_t = \begin{cases} \text{random}(1, \dots, m), & \text{with probability } \varepsilon_t \\ \arg\max_a Q(S_t, a; \theta), & \text{with probability } 1 - \varepsilon_t \end{cases} \tag{8}$$

Where,The exploration rate εt decays exponentially over time as*:*

$$\varepsilon_t = \varepsilon_{min} + (\varepsilon_{max} - \varepsilon_{min}) e^{-t/\tau} \tag{9}$$

Where, θ denotes the trainable weights of the DQN policy network.

3. Reward Function

After placing a container on swarm node a_t, the agent observes the resulting performance change and computes a **reward** to guide learning. A sample reward function combining QoS and efficiency objectives is defined as:

$$R_t = \lambda_1 \left(1 - \frac{T_{startup,avg}}{T_{startup,max}}\right) + \lambda_2 \left(1 - \frac{T_{response,avg}}{T_{response,max}}\right) + \lambda_3\, C_{a_t} \tag{10}$$

Where;

$T_{startup,avg}$ and $T_{response,avg}$ are the average startup and response times

$C_{a_t}$is the relative closeness of the selected swarm node

$\lambda_1, \lambda_2, \lambda_3$are weighting factors such that $\lambda_1 + \lambda_2 + \lambda_3 = 1$.

This formulation ensures higher rewards for deployments achieving lower startup and response times, while also favouring nodes with higher closeness scores.

4. Q-Value Estimation

The DQN approximates the optimal action–value function $Q^*(S_t, a_t)$using a neural network parameterized by θ:

$$Q(S_t, a_t; \theta) \approx E[R_t + \gamma max_{\,a'}\, Q(S_{t+1}, a'; \theta^-)] \tag{11}$$

Where;

$\gamma \in (0,1)$is the discount factor,

$\theta^-$are the parameters of the **target network** (periodically updated).

5. Q-Learning Update

The temporal difference (TD) target is defined as:

$$y_t = R_t + \gamma max_{\,a'}\, Q(S_{t+1}, a'; \theta^-) \tag{12}$$

The policy network parameters $\theta$are updated by minimizing the mean-squared TD error:

$$L_t(\theta) = E_{(S_t, a_t, R_t, S_{t+1}) \sim D}[(y_t - Q(S_t, a_t; \theta))^2] \tag{13}$$

Where,D is the replay buffer containing past experience tuples (S_t, a_t, R_t, S_{t+1}).

6. Gradient Descent Optimization

The parameters $\theta$are updated through stochastic gradient descent:

$$\theta \leftarrow \theta - \eta \nabla_\theta L_t(\theta) \quad (14)$$

Where, η is the learning rate.

7. Target Network Update

To stabilize learning, the target network parameters are periodically synchronized with the policy network:

$$\theta^- \leftarrow \theta every\ N_{update}\ steps. \quad (15)$$

8. Experience Replay Buffer

At each step, the agent stores experience tuples (*St, at, Rt, St+1*) into a replay buffer D. During training, mini-batches are sampled uniformly from D to break temporal correlations and improve stability. The replay buffer is defined as:

$$D = \{(S_i, a_i, R_i, S_{i+1}) \mid i = 1, \dots, N_{buffer}\} \quad (16)$$

9. Container Placement Decision

The swarm node with the highest estimated Q-value is selected for container deployment:

$$N_{selected} = arg\ max_{\ i\in\{1,\dots,m\}} Q(S_t, a_i; \theta) \quad (17)$$

4.4. MCSCM-RL Algorithm

This subsection presents the pseudocode of the proposed MCSCM-RL scheduling algorithm, which summarizes the operational flow of the MCSCM-RL scheduling algorithm, state construction, action selection, reward calculation, and policy update steps. Pseudocode for the algorithm is provided in Algorithm-1.

Algorithm-1. Multi-Criteria Scheduling of Containerized Microservices using Reinforcement Learning (MCSCM-RL).

Require: P_n: Number of Docker Swarm nodes; C_n: Number of microservice instances to deploy
Ensure: Scheduled deployment decisions and performance logs
1: Initialize a Deep Q-Network (DQN) agent
2: State dimension ← 5
3: Action dimension ← P_n
4: for i = 1 to C_n do
5: Collect latest resource metrics for each Swarm node:

6: CPU usage, Memory usage, Container count, Power consumption
7: Retrieve historical average startup time and average response time
8: Form a 5-parameter feature matrix:
{CPU%, Memory%, Containers, Avg.Startup, Avg.Response}
9: Normalize the matrix and compute relative closeness coefficients
10: Construct state vector s_i from the closeness values
11: Select node n_i using ε-greedy policy:

$$n_i = \begin{cases} \text{random node}, & \text{with probability } \varepsilon \\ \arg\max_a Q(s_i, a; \theta), & \text{with probability } 1 - \varepsilon \end{cases}$$

12: Deploy microservice *i* on node n_i
13: Measure startup time and response time of the deployed microservice
14: Compute reward: $r_i = -$(response time)
15: (Or apply a penalty if deployment fails)
16: Obtain next state s_i+1 by recomputing relative closeness after deployment
17: Store transition (s_i, a_i, r_i, s_i+1) in replay buffer D
18: Train DQN on sampled mini-batches from D
19: Log deployment details: selected node, startup time, response time, updated metrics
20: **end** for
21: **return** Final DQN-trained scheduling policy and performance logs

5. Performance Evaluation

This section presents the details of the experimental platform that was used to validate the MCSCM-RL scheduling approach presented in this article. The experiments took place on a three-node Docker Swarm cluster made of one manager and two worker nodes, which were virtual machines running Ubuntu. The cluster was designed to automatically monitor the resource and application-level metrics like CPU usage, memory usage, number of containers, starting time, and response time. Two types of deployment scenarios, clean and cumulative, were planned to measure the schedulers effectiveness in both situations of low-load and a workload that increases step by step. The proposed MCSCM-RL scheduler was evaluated alongside three

native Docker Swarm strategies (Spread, Random, and Binpack) and the non-RL multi-criteria scheduler (MCSCM). For each strategy, identical deployment procedures and workload characteristics were applied to ensure methodological fairness and reproducibility. The following paragraphs explain the cluster setup, deployment scenarios, and experimental procedure that have been used to produce the performance data, which have been discussed in Section-6.

5.1. Experiment Environment and Setup

The experimental environment consisted of a three-node Docker Swarm cluster deployed on Ubuntu-based virtual machines hosted within a Windows system, com prising one manager node and two worker nodes, as illustrated in Fig. 3. All nodes were connected through a private virtual network to ensure isolated and deterministic communication. This configuration provided a controlled, resource-constrained testbed suitable for evaluating container scheduling behavior. It further enabled consistent collection of CPU, memory, and QoS-related metrics across the swarm cluster throughout all experimental iterations.

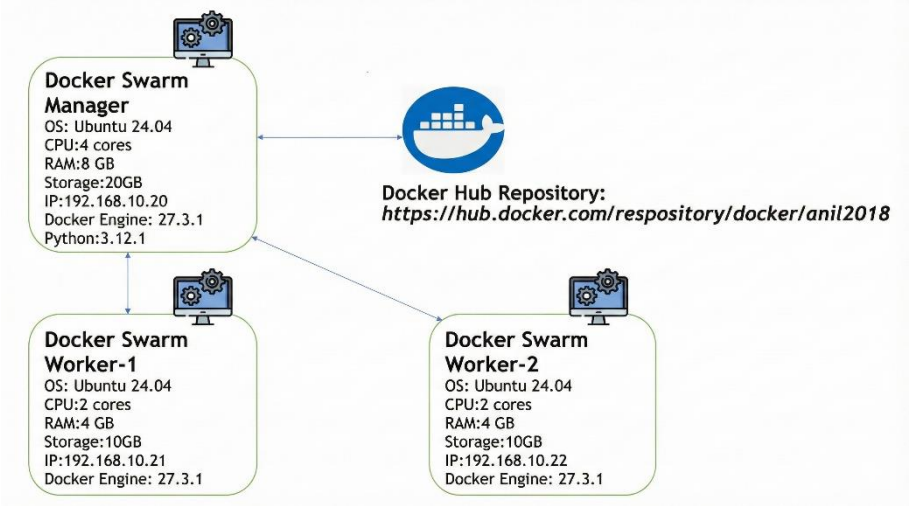


Fig. 3. Docker Swarm three-node cluster (1 manager + 2 workers) used for experiments.

Due to the lightweight architecture, Docker Swarm was selected, as the experimental platform. It offers integrated clustering support, ease of deployment, simplified scheduling workflow and fitness for controlled cluster experimentation. These features make Docker Swarm appropriate for experimenting reinforcement learning–based scheduling approaches, where frequent container deployments and fine-grained metric collection are required with low orchestration overhead [11].

The Kubernetes is the foremost container orchestration platform. Its scheduling framework applies multiple layers of plugins, constraints, and filtering stages. Hence, it can complicate the impact of a learning-based scheduling approach. However, Docker Swarm provides a simple and transparent scheduling model, which enables clear analysis of the learning behavior and decision-making process of the proposed MCSCM-RL scheduler. Kubernetes applies a manifold scheduling pipeline. It supports custom schedulers, which enables advanced scheduling logic but brings additional architectural complexity [12].

Whereas, the existing literature on RL-based scheduling in Docker Swarm is very limited and this gap further motivates the present work. The proposed MCSCM-RL work operates on cluster node-level and application-level metrics. Due to that the work is not essentially tied to Docker Swarm specific mechanisms. The insights received from the study can be transferred to more complex orchestration platforms, such as Kubernetes, through custom schedulers or scheduling extenders is identified as an important direction for future research.

With concentration on Docker Swarm, this study highlights algorithmic clarity and reproducibility, which are required for isolating the effects of reinforcement learning in container scheduling.

5.2. Deployment Scenarios

To thoroughly assess the performance of Random, Spread, Binpack, MCSCM, and the proposed MCSCM-RL scheduling strategies, two experimental deployment scenarios were developed: Clean Deployment and Cumulative Deployment. These scenarios were executed systematically across the Docker Swarm cluster to assess both the isolated and progressive behavior of each strategy under varying system loads.

- **Clean deployment:** In this scenario, each scheduling strategy was tested in an isolated environment to ensure unbiased performance evaluation. Prior to creating a new set of deployment, all the running containers, and services were removed from each of the nodes in the Swarm cluster. In this way, the cluster was returned to its original clean condition. A

fixed number of microservices were then deployed sequentially, following the respective scheduling policy.

• **Cumulative deployment:** The overall scenario was intended to study how well each scheduling strategy could scale and adapt their performance in the presence of varying workloads. Unlike the clean deployment case, containers and services were not removed between successive deployment cycles. Instead, microservices were incrementally deployed to simulate a gradually intensifying workload on the cluster.

5.3. Experiment Procedure

Random Spread Binpack, and MCSCM-RL scheduling strategies were independently run and their performances were measured against the same database in order to maintain fairness and allow reproducibility. The benchmark microservices were stored in a Docker Hub repository (namespace ***anil2018***); the Fibonacci microservice (***anil2018/myapp1: fibonacci***) provided a CPU-bound workload for each deployment. Prior to each trial, the cluster was restored to the required baseline state (empty for clean runs). During each trial the scheduler under test determined the target swarm node for deployment.

For every deployed microservice, all metrics were logged in CSV files to enable direct comparison for clean and cumulative deployment scenarios for each scheduling strategy. This approach provides insights into how each scheduling strategy manages resource contention, QoS stability, and deployment scalability as the cluster load increases over time.

Throughout each deployment, node-level metrics including CPU utilization, memory utilization, and container count were continuously monitored by all nodes, while application-level metrics such as startup-time and response-time were recorded after each container deployment to evaluate microservice responsiveness and Quality of Service (QoS). The CSV file collects node-level metrics at a sampling interval of two seconds.

For every scheduling strategy, dedicated CSV files systematically capture application-level metrics after each microservice deployment. To minimize the impact of transient variations and network fluctuations, all recorded measurements were averaged over multiple experimental repetitions. The aggregated results were subsequently used to assess the performance of the proposed MCSCM-RL scheduler relative to Docker Swarm's built-in strategies, with emphasis on improvements in response time, startup latency, and resource-aware scheduling efficiency.

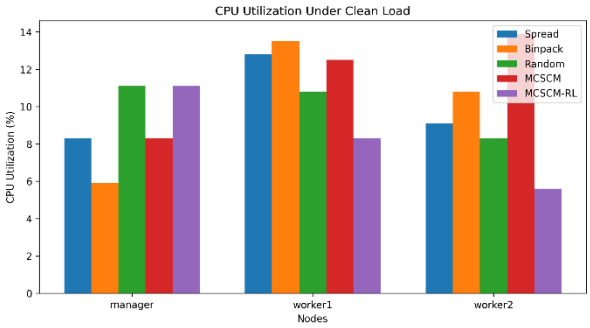
6. Results and Analysis

This section presents the performance evaluation of the proposed MCSCM-RL scheduler in comparison to Docker Swarm default schedulers: Spread, Binpack, Random, and the non-RL multi-criteria scheduler (MCSCM). The assessment takes into account both clean and cumulative deployment scenarios in order to record the performance under low-load and gradually increasing cluster utilization. All results were obtained via controlled experiments on a three-node Docker Swarm cluster setup, which consisted of one manager node and two worker nodes. In order to make a fair and consistent comparison, each scheduling strategy was tested on similar deployment iterations.

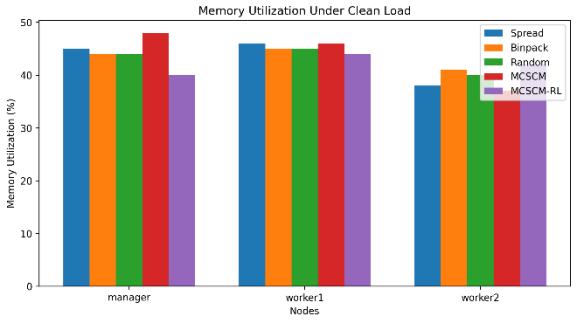
To ensure a fair and broad comparison, the performance of the MCSCM-RL scheduler and Docker Swarm's default schedulers was evaluated by multiple container deployments of microservices. Performance metrics, including CPU utilization, memory utilization, container distribution, average startup time, and average response time across nodes, are considered. These metrics collectively capture both system-level efficiency and application-level quality of the microservice.

6.1. Performance Evaluation under Clean Deployment Conditions

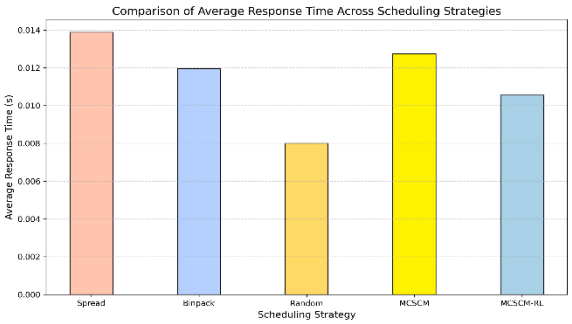
Under a clean deployment scenario, performance differences among scheduling strategies are relatively small due to uniform resource availability. However, the MCSCM-RL scheduler constantly achieves slightly lower startup and response times while maintaining balanced CPU and memory utilization. These comparative results across all performance metrics are summarized in Fig. 4. To enable an accurate evaluation of scheduling behavior, Figures (a), (b),(c), and (d) presents the results obtained under clean deployment conditions, where each scheduling strategy was evaluated on a Docker Swarm cluster fully reset to an empty state before the deployment of containerised microservices.



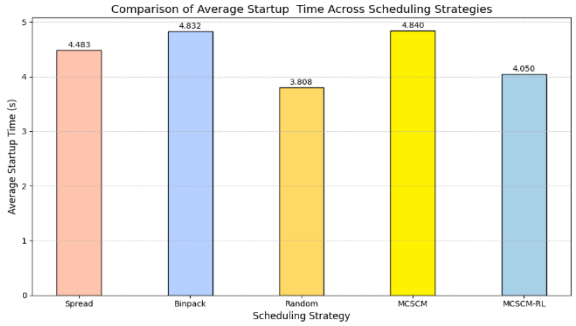
(a) CPU Utilization under clean load



(b) Memory Utilization under clean load



(c) Average Response Time under clean load



(d) Average Startup Time under clean load

Fig. 4. Results under Clean load deployments.

Evaluating performance in a clean environment provides a clear baseline for understanding how each scheduling strategy deploys containers and utilizes cluster resources in the absence of prior deployments.

Fig. 4 (a) presents the CPU utilization across the three swarm nodes for all scheduling strategies. The results indicate Spread and Binpack scheduler distribute the load evenly across the manager and worker nodes, but worker1 exhibits comparatively higher CPU usage (approximately 12–13%), reflecting their default binpacking and spreading behavior. Random scheduler gives imbalanced CPU distribution, with worker1 reaching 10.8% utilization and worker2 8.3%, confirming the non-deterministic nature of this strategics increases utilization on worker nodes, especially worker2 (approximately 13.9%), due to its static multi-criteria scoring preference. MCSCM-RL results in the lowest average CPU utilization among all techniques (approximately 8–11%), and more importantly, distributes CPU demand across the nodes more evenly. Its adaptive decision-making avoids repeated selection of a single node, making it more efficient even under clean load conditions. Overall, these results indicate that MCSCM-RL reduces unnecessary CPU overhead by dynamically selecting nodes based on learned system states rather than static placement heuristics.

Fig. 4 (b) presents memory utilization for each scheduling strategy across the three nodes. The results indicate Spread, Binpack, and Random scheduler maintain relatively similar memory utilization (44–46% on manager and worker1), as expected under clean deployment. MCSCM shows higher memory utilization on manager and worker1 (47–48%), caused by static metric-based selection that sometimes favors already stronger nodes. MCSCM-RL scheduler achieves more balanced memory utilization, particularly with lower overhead on the manager (≈40%). This demonstrates that RL not only considers current memory consumption but also anticipates future congestion, resulting in better overall load distribution. The memory balancing behavior of MCSCM-RL shows its ability to prevent early memory accumulation on specific nodes compared to deterministic schedulers.

Fig. 4 (c) shows the average response time of the containerized microservice under each scheduler. The response-time analysis shows clear distinctions in latency behavior. Random achieves the lowest response time (0.008 s), but this performance is inconsistent and highly dependent on chance. Spread (0.0139 s) and MCSCM (0.0127 s) demonstrate higher latencies, as both distribute services without adaptation to real-time node conditions. Binpack performs slightly better (0.01195 s) but still lacks adaptive balancing. MCSCM-RL achieves 0.01055 s, improving response time over Spread and MCSCM while maintaining stable and predictable performance. The RL-based scheduler learns to avoid nodes with even slightly higher CPU/memory loads, yielding more consistent and lower latency compared to deterministic approaches.

Fig. 4 (d) shows the average startup time under clean load. Random once again records the lowest startup time (3.808 s), but due to its non-deterministic behavior this result is not stable or repeatable. Spread and Binpack show moderate startup delays (4.48 s and 4.83 s), reflecting their respective default load allocation behaviors. MCSCM has the highest startup time (4.84 s),

caused by static multi-criteria bias toward specific nodes, which increases container initialization overhead.MCSCM-RL significantly improves startup time (4.05 s), and outperforms Spread, Binpack, and MCSCM.

The improved startup time in MCSCM-RL results from its dynamic intervention that avoids nodes with higher current service counts or ongoing container creation operations, enabling faster service initialization.

The average resource utilization (CPU and memory) across the three node Docker Swarm cluster for all scheduling strategies under clean deployment is presented in Fig. 5.

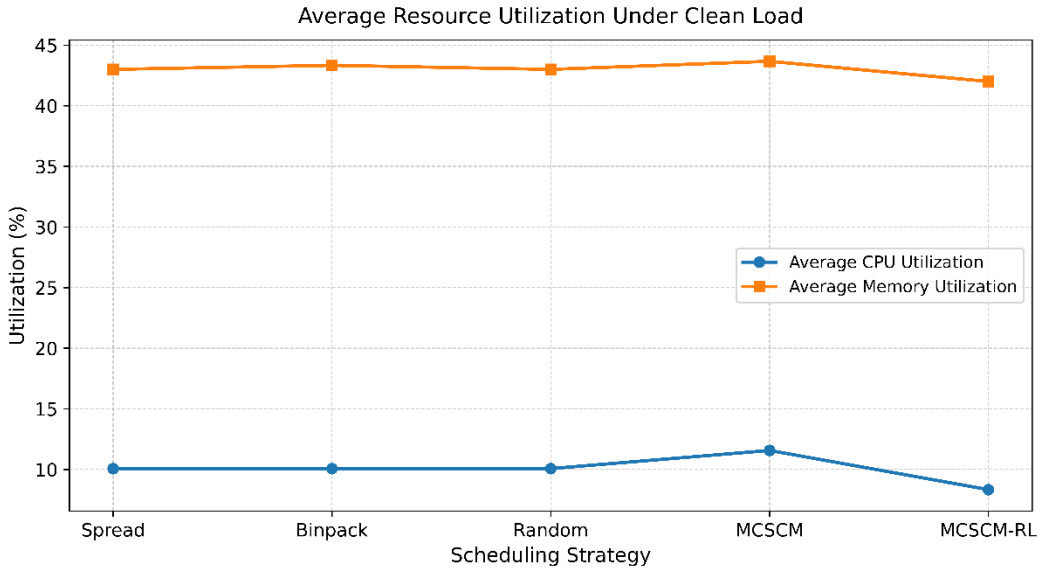


Fig. 5. Average Resource utilization of swarm cluster under clean load.

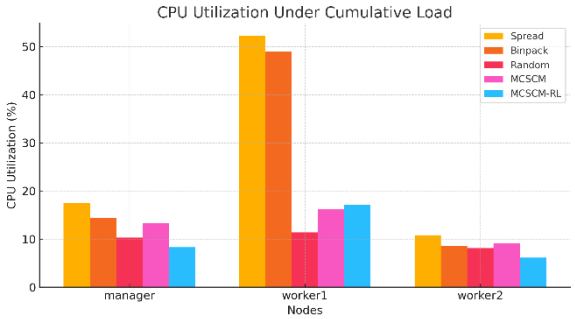
The results demonstrate that MCSCM-RL provides the most resource-efficient load distribution, and reduces average CPU utilization by 10–25% compared to the other scheduling strategies. Docker swarm strategies (Spread Binpack Random) are showing similar utilization patterns as they are non-adaptive by nature. The RL-based method recognizes the nature of the cluster and starts to learn it, thus giving the early benefits that eventually result in measurable improvements in cluster-level efficiency.

In the clean deployment scenario, Random scheduling sometimes shows competitive or better performance in startup time and response time. The lack of resource contention is responsible for these results, where initially all swarm nodes have the same resource availability. Under such conditions, random placement unconditionally deploys containers across cluster nodes, rather than considering workloads and possibility of resource saturation. As a result, for a while Random scheduling may do better from balanced resource usage without the need of explicit decision logic.

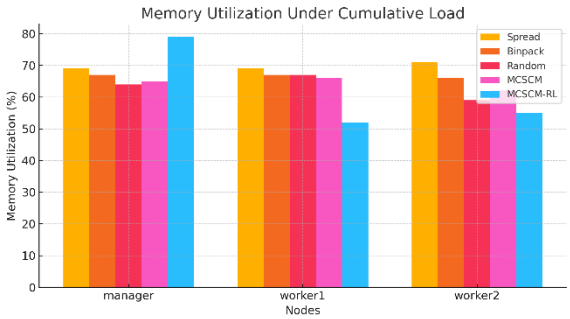
Overall, under clean deployment, the similarity in resource availability limits the performance gap between schedulers, although MCSCM-RL maintains better stability.

6.2. Performance Evaluation under Cumulative Deployment Conditions

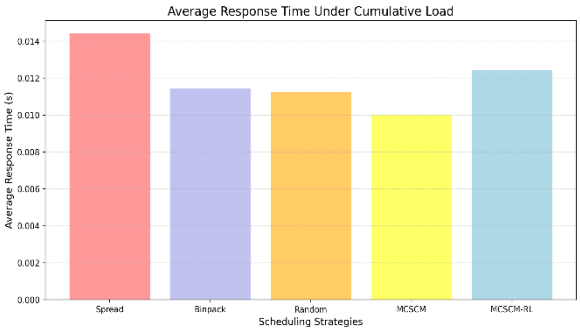
Under cumulative deployment scenarios, each new microservice is deployed without removing the previously deployed ones. This is a resource balancing scenario where the scheduler has to respond to the increase of the workload and, at the same time, maintain a good level of resource balance, minimize latency, and not allow resource saturation. Under this condition, MCSCM-RL consistently outperforms all strategies, delivering lowest average CPU usage, balanced memory utilization, reduced hotspots, stable response latency and lowest startup time. These comparative results across all performance metrics are summarized in Fig. 6. To enable an accurate evaluation of scheduling behavior, Figures (a), (b), (c), and (d) present the results obtained under clean deployment conditions.



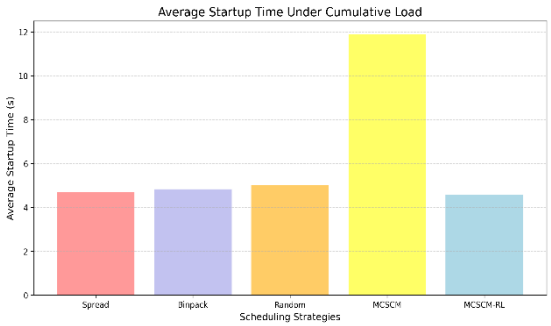
(a) CPU Utilization under cumulative load



(a) CPU Utilization under cumulative load



(c) Average Response Time under cumulative load



(d) Average Startup Time under cumulative load

Fig. 6. Results under Cumulative load deployments.

Fig. 6 (a) presents CPU utilization on the Swarm manager and two worker nodes. Spread and Binpack generate extremely high CPU utilization on worker1(52% and 49%, respectively), which indicates heavy load imbalance. Random performs better (worker1 at 11%) but still shows unbalanced utilization across nodes due to random clustering patterns. MCSCM increases CPU load on worker2 (14%), showing static bias in node scoring. MCSCM-RL maintains the most balanced CPU distribution. Despite worker1 being inherently more powerful, MCSCM-RL prevents extreme hotspots. In contrast, Spread and Binpack show 5 times higher CPU load on worker1 compared to other nodes.

Fig. 6 (b) presents memory utilization under cumulative load. Spread shows the highest memory consumption, with worker2 reaching 71% and all nodes staying near or above 69%. Binpack exhibits similar memory escalation (66–67%), caused by its consolidation-oriented behavior. Random slightly reduces memory on worker2 (59%), but the distribution remains inconsistent. MCSCM keeps memory usage moderately balanced but inflation persists due to repeated selection of static nodes. MCSCM-RL delivers the most balanced memory usage with worker1 and worker2 maintaining significantly lower utilization (52% and 55%, respectively). Only the manager bearing a higher load due to its control-plane duties. These results prove that MCSCM-RL effectively prevents memory hotspots and supports sustained multiple microservice deployments without degrading cluster performance.

Fig. 6 (c) presents the average response time for each scheduler. Spread performs the worst (0.0144 s), due to growing contention on worker1 and memory exhaustion across nodes. Binpack and Random give moderate performance (0.01143 s and 0.01123 s, respectively), but both exhibit fluctuations due to unstable node selection. MCSCM achieves the lowest response time (0.0100 s) under cumulative load, benefiting from multi-criteria evaluation. MCSCM-RL offers competitive performance (0.01244 s), with deviations arising from adaptive exploration. However,MCSCM-RL offers more consistent response times with far fewer high-latency spikes even as deployments accumulate.

Fig. 6 (d) presents the average startup time for each scheduler. Spread, Binpack, and Random show startup times in the range of 4.6–5.0 s, and gradually increasing as cluster resource load. MCSCM suffers heavily, with startup time reaching 11.904 s, confirming that static ranking leads to node overloading. MCSCM-RL provides the best startup performance (4.57 s), significantly outperforming all other strategies except Random. Its adaptive node selection helps avoid overloaded nodes, reducing container initialization delays. This demonstrates MCSCM-RL's ability to adaptively maintain QoS even under heavy cumulative deployment.

The average resource utilization (CPU and memory) across the three node Docker Swarm cluster for all scheduling strategies under cumulative deployment is presented in Fig. 7.

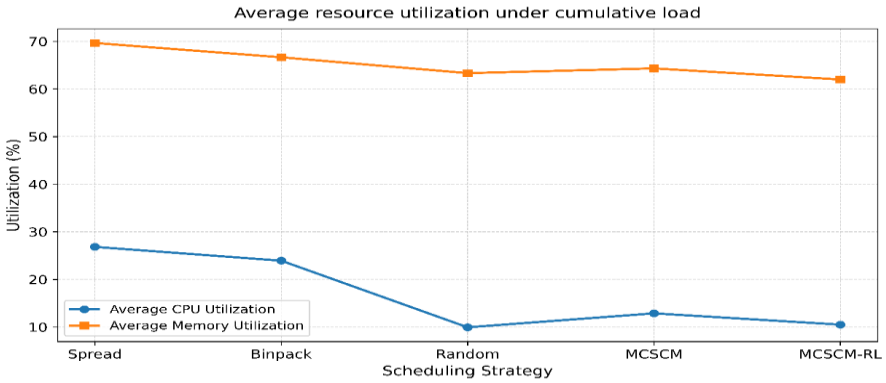


Fig. 7. Average Resource utilization of swarm cluster under cumulative load.

The results indicate a clear divergence in behavior. Spread demonstrates the highest resource consumption, with CPU averaging 27% and memory reaching 70%. As microservices accumulate, Spread continues distributing replicas uniformly, causing worker1 to become heavily loaded due to its higher baseline capability. Binpack shows slightly lower memory usage than Spread but pushes CPU usage to nearly 24%, as it continues deploying microservices onto nodes that initially appear least loaded. Over time, this creates a substantial resource load on worker1. Random maintains lower CPU usage nearly 10% but memory increases beyond 66%. Because random selection frequently targets the same nodes. MCSCM shows balancing but suffers from static ranking. As deployments accumulate, the same preferred nodes are repeatedly selected, causing CPU spikes (worker1 at 16%) and memory usage across the cluster. MCSCM-RL demonstrates the lowest average CPU usage (approximately 10–11%) and lowest memory growth (approximately 62%), consistently outperforming all other schedulers. The real-time learning of MCSCM-RL enables it to avoid overload nodes and mitigate the cumulative buildup of microservices.

Overall, MCSCM-RL provides the most resource-efficient behavior. Under cumulative deployments MCSCM-RL consistently outperforms all scheduling strategies, delivering lowest average CPU usage, balanced memory utilization, reduced hotspots, near-stable response latency and lowest startup time. MCSCM-RL is the only strategy capable of sustaining long cumulative deployment sequences without cluster instability.

6.3. Comparative Analysis of Clean and Cumulative Deployment Scenarios

Fig.8 presents a combined comparison of average CPU utilization, memory utilization, response time, and startup time across all five scheduling strategies under clean and cumulative deployment conditions.

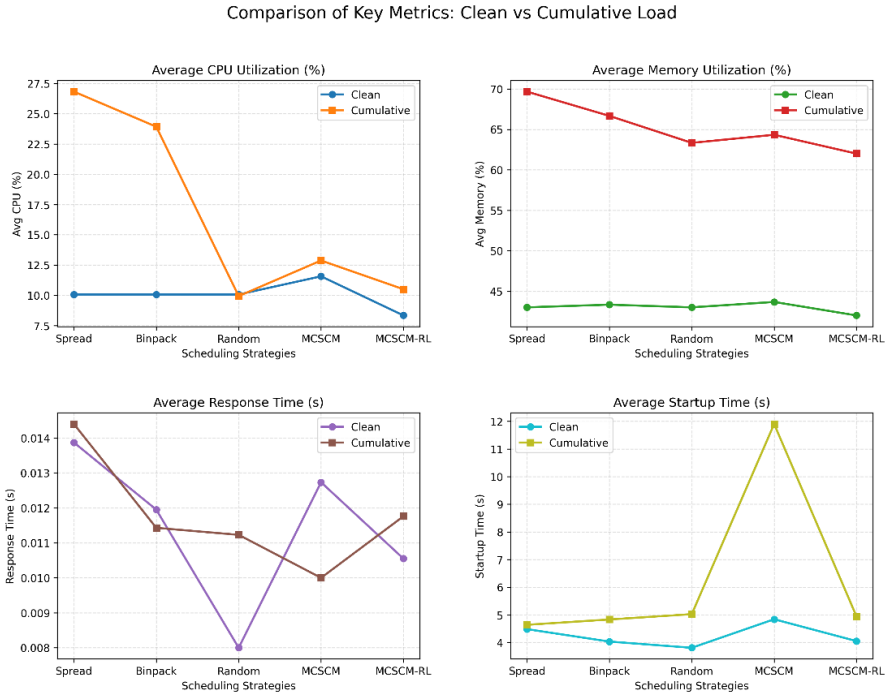


Fig. 8. Comparison of key metrics under Clean and Cumulative load.

Results show that Spread and Binpack experience CPU and memory inflation on worker nodes and are remarkable in response time and startup performance due to their deterministic allocation mechanism.

Random avoids such deterministic bias but offers no meaningful optimization. The MCSCM demonstrates improved decision-making under clean load; however, its reliance on fixed ranking leads to progressively unbalanced placements under cumulative deployment. As services accumulate, MCSCM repeatedly selects the same high-scoring nodes, causing resource saturation and increase in startup time, thus limiting its applicability in dynamic, real-world microservice environments.

In the clean deployment scenario, Random scheduling sometimes shows competitive or better performance in startup time and response time. The root cause of these results is the absence of resource contention, where at the beginning, all swarm nodes have the same level of resource availability. Under such conditions, random placement unconditionally deploys containers across cluster nodes, rather than considering workloads and possibility of resource saturation. As a result, for a while Random scheduling may do better from balanced resource usage without the need of explicit decision logic. On the other hand, as the deployment of containers substantially grows and the load on the cluster rises, the weakness of Random scheduling is revealed without delay. Random scheduling increasingly deploys containers to already congested nodes, because deployment decisions are taken without considering current resource utilization or historical application performance. Such a situation results in bigger startup times and worse response times. In contrast, schedulers that are conscious of resources and those based on learning make scheduling decisions from a measured system state perspective and thus are able to handle resource contention more effectively even when the load is changing.

This transition from short-term effectiveness to long-term inefficiency shows the importance of adaptive container scheduling. Random scheduling can perform well in homogeneous cluster conditions and minimal load. It has a lack of workload limits awareness, scalability and robustness in cumulative container deployment scenarios. Whereas, MCSCM-RL scheduler exhibits consistent performance improvements through adaptive node selection for deployment.

Random scheduling is commonly used as a baseline in container scheduling studies. Previous studies showed similar behaviors, where random scheduling performs well under low-load and homogeneous cluster conditions but degrades as system load increases [46].

In contrast, the proposed reinforcement learning based scheduler, constantly offers excellent scalability and robustness across all metrics. By continuously learning from real-time cluster conditions, MCSCM-RL maintains the lowest CPU and memory usage and preserves stable response latencies even under sustained workload expansion. Furthermore, it achieves the most favourable startup-time behavior among all scheduling strategies. Altogether, these results support MCSCM-RL as the sole approach that can maintain high quality of service (QoS), energy efficiency, and even resource utilization under both clean and cumulative deployment scenarios. It proves that it is most suitable for large-scale, production-grade microservice orchestration.

Observed performance differences are statistically meaningful. The results indicate that the performance improvements achieved by MCSCM-RL scheduler are statistically significant across key performance metrics, especially under cumulative deployment scenarios.

More comprehensive statistical study could further improve the evaluation. The results demonstrate that MCSCM-RL outperforms default swarm schedulers in dynamic workload and variable executions. Future work will include more comprehensive testing across larger clusters and diverse workload conditions.

7. Conclusion and Future Work

This research presented an extensive evaluation of multiple scheduling strategies, including Spread, Binpack, Random, MCSCM, and the proposed MCSCM-RL, for efficient and QoS-aware orchestration of containerized microservices in Docker Swarm environments. Experimental outcomes have revealed that conventional schedulers, while perfectly good when operating under the clean deployment, lead to serious performance drops when the number of services deployed grows. Spread and Binpack suffer from severe disparity between CPU and memory under cumulative load as a result of their predictable allocation strategies. Random scheduler achieving lower response times initially, provides inconsistent and non-optimal performance as system complexity grows. The static multi-criteria scheduler, MCSCM, improves upon these approaches under clean conditions but remains sensitive to workload accumulation, showing severe startup-time inflation and node saturation. Under clean deployment, the static multi-criteria scheduler, MCSCM, outperforms these methods. However, it is vulnerable to workload increase and shows significant node saturation and startup-time inflation.

On the other hand, the MCSCM-RL, consistently outperformed all existing scheduling strategies under clean and cumulative deployment scenarios. By leveraging state-aware learning and continuous feedback from real-time cluster metrics, MCSCM-RL achieved superior CPU and memory balancing, minimized response-time variability, and significantly reduced startup delays. Its ability to dynamically adapt to evolving node states makes it highly effective in maintaining QoS guarantees and energy-efficiency. It presents itself as a robust and scalable solution for modern cloud-native

microservice infrastructures across diverse deployment conditions.

There are still a number of opportunities for further research. First, evaluating the scheduler on larger and heterogeneous clusters, both within Docker Swarm and on widely adopted orchestration platforms such as Kubernetes, would offer deeper insights into its scalability and applicability to real-world cloud environments. Moreover, integrating workload prediction mechanisms, such as LSTM, might make it possible to make proactive scheduling option that predict resource fluctuations rather than only responding to them.

Conflict of Interest

The authors declare there is no conflict of interest.

Funding

Self-funded.

Acknowledgments

Not applicable.

Availability of Data and Materials

The datasets used during the current study are available from the corresponding author on request.

Using Artificial Intelligent Chabot's

None.

Final Credit Author Statement:

Anil A. Prajapati: Literature review conceptualization methodology, formal analysis investigation experimental design, and writing of the original draft.

Manish M. Patel: Supervision, data curation, validation, and final manuscript revision.

References

[1] de Lauretis L. From monolithic architecture to microservices architecture. Proc IEEE Int Symp Softw Reliab Eng Workshops 2019:93-96.

[2] Doe J, Smith A. The evolution of microservices and containerization. J Softw Eng 2023;12:45-60.

[3] Bentaleb O, Belloum ASZ, Sebaa A, El-Maouhab A. Containerization technologies: Taxonomies, applications and challenges. J Supercomput 2022;78:1144-81.

[4] Kavis M. Challenges and solution directions of microservice architectures. Tech Rep Wageningen Univ 2020.

[5] Souppaya M. Container-based cluster orchestration systems: A taxonomy. ACM Comput Surv 2021;54:129.

[6] Tariq M, Waqas M, Ahmad J, Khan A, Alazab M, Mustapha A, Khan S, Anuar NB. A comprehensive feature comparison study of open-source container orchestration frameworks. J Cloud Comput 2022;11:1.

[7] Zhou N, Zhou H, Hoppe D. Containerization for high performance computing systems: Survey and prospects. IEEE Trans Softw Eng 2023;49:2722-40.

[8] Sharma S, Vishwakarma R. An efficient container scheduling framework for resource allocation in cloud computing environments. Commun Appl Electron 2025;7:39-48.

[9] Tariq M, Asfand-e-Yar M, Waqas M, Ahmad J, Alazab M, Mustapha A, Khan S, Anuar NB. A survey on container orchestration and scheduling in cloud computing environments. IEEE Trans Cloud Comput 2022;10:1-20.

[10] Chen X, Xiao S. Multi-objective and parallel particle swarm optimization algorithm for container-based microservice scheduling. Sensors 2021;21:6212.

[11] Pahl C, Brogi A, Soldani J, Jamshidi P. Cloud container technologies: A state-of-the-art review. IEEE Trans Cloud Comput 2021;7:677-92.

[12] Cao R, Zhang P, Wu Y, Liu J, Su H. Adaptive container scheduling based on reinforcement learning in Kubernetes. CCF Trans High Perform Comput 2025;7:291-304.

[13] Rodríguez J, Buyya R. A taxonomy and survey of container orchestration systems for cloud computing. Softw Pract Exp 2022;52:4-33.

[14] Pérez de Prado R, García-Galán S, Muñoz-Expósito JE, Marchewka A, Ruiz-Reyes N. Smart containers schedulers for microservices provision in cloud-fog-IoT networks: Challenges and opportunities. Sensors 2020;20:1714.

[15] al Qassem LM, Stouraitis T, Damiani E, Elfadel IM. Containerized microservices: A survey of resource management frameworks. IEEE Trans Netw Serv Manag 2024;21:3775-96.

[16] Li C, Bai J, Ge Y, Luo Y. Heterogeneity-aware elastic provisioning in cloud-assisted edge computing systems. Future Gener Comput Syst 2020;112:1106-21.

[17] Achrudin MR, Muslikh AR. Optimization of application deployment architecture in container orchestration. J Appl Inform Comput 2025;9:45-53.

[18] Marella V. Comparative analysis of container orchestration platforms: Kubernetes vs. Docker Swarm. Int J Sci Res Sci Technol 2024;11:122-30.

[19] Singh N, Hamid Y, Juneja S. Load balancing and service discovery using Docker Swarm for

microservice-based big data applications. J Cloud Comput 2023;12:4.
[20] Shi Y, Guo Y, Lv L, Zhang K. An efficient resource scheduling strategy for V2X microservice deployment in edge servers. Future Internet 2020;12:172.
[21] Lin M, Xi J, Bai W, Wu J. Ant colony algorithm for container-based microservice scheduling in hybrid cloud. J Phys Conf Ser 2021;1994:012028.
[22] Fan G, Chen L, Yu H, Qi W. Multi-objective optimization of container-based microservice scheduling in edge computing. Comput Sci Inf Syst 2023;18:1345-65.
[23] Muniswamy S, Vignesh R. DSTS: A hybrid optimal and deep learning for dynamic scalable task scheduling on a container cloud environment. J Cloud Comput 2022;11:33.
[24] Rao W, Li H. Energy-aware scheduling algorithm for microservices in Kubernetes clouds. J Grid Comput 2025;23:2.
[25] Liu R, Lv H, Yang P, Bie R. A multi-task genetic programming approach for online multi-objective container placement in heterogeneous clusters. Complex Intell Syst 2024.
[26] Nkenyereye L, Lee S. Functionality-aware offloading technique for scheduling container applications in edge infrastructures. J Cloud Comput 2025;14:28.
[27] Moreschini S, Pour S, Lanese I, Balouek D, Bogner J, Li X, Pecorelli F. AI techniques in the microservices life-cycle: A systematic mapping study. Computing 2025;107:1235-68.
[28] John A, Kawash J, Alhajj R, Al Qassem LM, Elfadel IAM. Predictive container orchestration in the cloud using artificial intelligence techniques. Computing 2025.
[29] Li W, Li X, Chen L. Pattern learning for scheduling microservice workflow to cloud containers. Int J Mach Learn Cybern 2024.
[30] Shi K, Huang Y, Xu H. Resource-constrained scheduling in containerized edge computing using graph transformer-enhanced DQN. 2025:210-20.
[31] Xia H, Hao J, Cao R. Adaptive resource allocation for cloud-native microservice via meta-learning and hyper-heuristic algorithms. J Grid Comput 2025.
[32] Sanjalawe Y, Al-E'mari S, Fraihat S, Makhadmeh S. AI-driven job scheduling in cloud computing: A comprehensive review. Artif Intell Rev 2025;58:197.
[33] Kallel A, Rekik M, Khemakhem M. A deep reinforcement learning-based optimization approach for containerized microservice scheduling in hybrid fog/cloud environments. Eng Appl Artif Intell 2025;141:109745.
[34] Turin G, Borgarelli A, Donetti S, Damiani F, Johnsen EB, Tapia Tarifa SL. Predicting resource consumption of Kubernetes container systems using resource models. J Syst Softw 2023;203:111750.
[35] Marchese A, Tomarchio O. Enhancing the Kubernetes platform with a load-aware orchestration strategy. SN Comput Sci 2025;6:224.
[36] Bai X, Islam MT, Buyya R, Toosi AN. REACH: Reinforcement learning for adaptive microservice rescheduling in the cloud-edge continuum. arXiv 2025;2510.06675.
[37] Cai J, Zeng H, Liu F, Chen J. Intelligent dynamic multi-dimensional heterogeneous resource scheduling optimization strategy based on Kubernetes. Mathematics 2025;13:1342.
[38] Jian Z, Xie X, Fang Y, Jiang Y, Lu Y, Dash A, Li T, Wang G. DRS: A deep reinforcement learning enhanced Kubernetes scheduler for microservice-based system. Softw Pract Exp 2024;54:2102-26.
[39] Kallel A, Rekik M, Khemakhem M. DRL4HFC: Deep reinforcement learning for container-based scheduling in hybrid fog/cloud system. Proc Int Conf Agents Artif Intell 2024:231-42.
[40] Senjab K, Abbas S, Ahmed N, Khan A ur R. A survey of Kubernetes scheduling algorithms. J Cloud Comput 2023;12:87.
[41] Santos J, Zaccarini M, Poltronieri F, Tortonesi M, Stefanelli C, di Cicco N, de Turck F. HephaestusForge: Optimal microservice deployment across the compute continuum via reinforcement learning. Future Gener Comput Syst 2025;166:107680.
[42] Lorido-Botran T, Bhatti MK. Adaptive container scheduling in cloud data centers: A deep reinforcement learning approach. 2021:572-81.
[43] Sutton RS, Barto AG. Reinforcement learning: An introduction. 2nd ed. Cambridge: MIT Press; 2018.
[44] Auer P, Cesa-Bianchi N, Fischer P. Finite-time analysis of the multiarmed bandit problem. Mach Learn 2002;47:235-56.
[45] Osband I, Russo D, Van Roy B. More efficient reinforcement learning via posterior sampling. Adv Neural Inf Process Syst 2013;26.
[46] Xu J, Fortes JAB, Carpenter T. On the use of random scheduling for load balancing in distributed systems. IEEE Trans Parallel Distrib Syst 2012;23:c4-c4.