MULTICRITERIA SCHEDULING OF MICROSERVICES IN CONTAINER BASED CLOUD ENVIRONMENT

A Thesis submitted to Gujarat Technological University

For the award of

Doctor of Philosophy

In

Computer/ IT Engineering

by

Prajapati Anilkumar Amrutlal

[209999913024]

under supervision of

Dr. Manish M. Patel

Submitted to

Gujarat Technological University

Ahmedabad, Gujarat, India.

[SEPTEMBER-2026]

MULTICRITERIA SCHEDULING OF MICROSERVICES IN CONTAINER BASED CLOUD ENVIRONMENT

A Thesis submitted to Gujarat Technological University

For the award of

Doctor of Philosophy

In

Computer/ IT Engineering

by

Prajapati Anilkumar Amrutlal

[209999913024]

under supervision of

Dr. Manish M. Patel

Submitted to

Gujarat Technological University

Ahmedabad, Gujarat, India.

[SEPTEMBER-2026]

Certificate

I certify that the work incorporated in the thesis titled as "**Multicriteria Scheduling of Microservices in Container based Cloud Environment**" submitted by Shri. Prajapati Anilkumar Amrutlal was carried out by the candidate under my supervision/guidance. To the best of my knowledge: (i) the candidate has not submitted the same research work to any other institution for any degree/diploma, Associate ship, Fellowship or other similar titles (ii) the thesis submitted is a record of original research work done by the Research Scholar during the period of study under my supervision, and (iii) the thesis represents independent research work on the part of the Research Scholar.

Signature of the Supervisor: **Date:** 07/09/2026

Name of Supervisor: Dr. Manish M. Patel

Place: Ahmedabad.

Course-work Completion Certificate

This is to certify that Mr. **Prajapati Anilkumar Amrutlal** enrolment no:**209999913024** is a Ph.D. scholar for Ph.D. program in the branch **Computer/IT Engineering** of Gujarat Technological University, Ahmedabad.

(Please tick the relevant the relevant options(s))

☐ He has been exempted from the course-work (successfully completed during M.Phil Course)

☐ He has been exempted from Research Methodology Course only (successfully completed during M.Phil Course)

☑ He has successfully completed the PhD course work for the partial requirement for the award of PhD Degree. His performance in the course work is as follows-

Grade Obtained in Research Methodology (PH001)	Grade Obtained in Self Study Course (Core Subject) (PH002)
BB	AA

Signature of the Supervisor:

Dr. Manish M. Patel

Originality Report Certificate

It is certified that PhD Thesis titled "**Multicriteria Scheduling of Microservices in Container based Cloud Environment**" submitted by Mr. Prajapati Anilkumar Amrutlal has been examined by us. We undertake the following:

a. Thesis has significant new work / knowledge as compared already published or is under consideration to be published elsewhere. No sentence, equation, diagram, table, paragraph or section has been copied verbatim from previous work unless it is placed under quotation marks and duly referenced.

b. The work presented is original and own work of the author (i.e. there is no plagiarism). No ideas, processes, results or words of others have been presented as Author own work.

c. There is no fabrication of data or results which have been compiled / analyzed.

d. There is no falsification by manipulating research materials, equipment or processes, or changing or omitting data or results such that the research is not accurately represented in the research record.

e. The thesis has been checked using TurnitIn (copy of originality report attached) and found within limits as per GTU Plagiarism Policy and instructions issued from time to time (i.e. permitted similarity index <=25%).

Signature of Research Scholar: **Date:** 7/9/26

Name of Research Scholar: **Prajapati Anilkumar Amrutlal**

Place: Ahmedabad

Signature of Supervisor: **Date:** 07/09/2026

Name of Supervisor: **Dr. Manish M. Patel**

Place: Ahmedabad

Anil Prajapati

Thesis_209999913024

Anil Prajapati
SP
Gujarat Technological University

Document Details

Submission ID
trn:oid:::1:3637042734

Submission Date
Aug 31, 2026, 11:25 AM GMT+5:30

Download Date
Sep 5, 2026, 2:50 PM GMT+5:30

File Name
PHD_thesis_31_08_2026-Plagcheck_.pdf

File Size
4.5 MB

114 Pages

28,615 Words

185,635 Characters

8% Overall Similarity

The combined total of all matches, including overlapping sources, for each database.

Filtered from the Report

- Bibliography
- Quoted Text
- Cited Text
- Small Matches (less than 9 words)

Match Groups

- 192 Not Cited or Quoted 8%
  Matches with neither in-text citation nor quotation marks
- 0 Missing Quotations 0%
  Matches that are still very similar to source material
- 0 Missing Citation 0%
  Matches that have quotation marks, but no in-text citation
- 0 Cited and Quoted 0%
  Matches with in-text citation present, but no quotation marks

Top Sources

6% Internet sources
4% Publications
0% Submitted works (Student Papers)

Integrity Flags

1 Integrity Flag for Review

- Hidden Text
  12 suspect characters on 6 pages
  Text is altered to blend into the white background of the document.

Our system's algorithms look deeply at a document for any inconsistencies that would set it apart from a normal submission. If we notice something strange, we flag it for you to review.

A Flag is not necessarily an indicator of a problem. However, we'd recommend you focus your attention there for further review.

PhD Thesis Non-Exclusive License to

GUJARAT TECHNOLOGICAL UNIVERSITY

In consideration of being a PhD Research Scholar at GTU and in the interests of the facilitation of research at GTU and elsewhere, I, Prajapati Anilkumar Amrutal having Enrolment No: 209999913024 hereby grant a non-exclusive, royalty free and perpetual license to GTU on the following terms:

a. The University is permitted to archive, reproduce and distribute my thesis, in whole or in part, and/or my abstract, in whole or in part (referred to collectively as the "Work") anywhere in the world, for non-commercial purposes, in all forms of media;

b. The University is permitted to authorize, sub-lease, sub-contract or procure anyof the acts mentioned in paragraph (a);

c. The University is authorized to submit the Work at any National / International Library, under the authority of their "Thesis Non-Exclusive License";

d. The Universal Copyright Notice (©) shall appear on all copies made under the authority of this license;

e. I undertake to submit my thesis, through my University, to any Library and Archives. Any abstract submitted with the thesis will be considered to form part of the thesis.

f. I represent that my thesis is my original work, does not infringe any rights of others, including privacy rights, and that I have the right to make the grant conferred by this non-exclusive license.

g. If third party copyrighted material was included in my thesis for which, under the terms of the Copyright Act, written permission from the copyright owners is required, I have obtained such permission from the copyright owners to do the acts mentioned in paragraph (a) above for the full term of copyright protection.

h. I understand that the responsibility for the matter as mentioned in paragraph (g) rests with the authors/me solely. In no case GTU have any liability for any acts / omissions / errors / copyright infringement from the publication of the said thesis or otherwise.

i. I retain copyright ownership and moral rights in my thesis, and may deal with the copyright in my thesis, in any way consistent with rights granted by me to my University in this non-exclusive license.

j. GTU logo shall not be used/printed in the book (in any manner whatsoever) being published or any promotional or marketing materials or any such similar documents.

k. The following statement shall be included appropriately and displayed prominently in the book or any material being published anywhere: "The content of the published work is part of the thesis submitted in partial fulfillment for the award of the degree of Ph.D. in Computer/IT Engineering of the Gujarat Technological University."

l. I further promise to inform any person to whom I may hereafter assign or license my copyright in my thesis of the rights granted by me to my University in this non-exclusive license. I shall keep GTU indemnified from any and all claims from the Publisher(s) or any third parties at all times resulting or arising from the publishing or use or intended use of the book / such similar document or its contents.

m. I am aware of and agree to accept the conditions and regulations of PhD including all policy matters related to authorship and plagiarism.

Signature of the Research Scholar:

Name of Research Scholar: **Prajapati Anilkumar Amrutlal**

Date: 07/09/26

Place: Ahmedabad

Signature of Supervisor:

Name of Supervisor: **Dr. Manish M. Patel**

Date: 07/09/26

Place: Ahmedabad

Seal:

Thesis Approval Form

The viva-voce of the PhD Thesis submitted by Mr. **Prajapati Anilkumar Amrutlal.** (Enrolment No: **209999913024**) entitled **Multicriteria Scheduling of Microservices in Container based Cloud Environment** was conducted on .....7/9/26., 11.30 AM (day and date) at Gujarat Technological University.

(Please tick any one of the following options)

☑ The performance of the candidate was satisfactory. We recommend that he be awarded the PhD degree.

☐ Any further modifications in research work recommended by the panel after 3 months from the date of first viva-voce upon request of the Supervisor or request of Independent Research Scholar after which viva-voce can be re-conducted by the same panel again.

☐ The performance of the candidate was unsatisfactory. We recommend that he should not be awarded the PhD degree.

Name and Signature of Supervisor with Seal

1) (External Examiner 1) Name and Signature

PROF. SUDIP MISRA

Dr. Rashmi Ranjan Rout

(External Examiner 2) Name and Signature

Abstract

Cloud computing has emerged as a dominant paradigm for deploying scalable and distributed applications through the adoption of microservices architecture and containerization technologies. Containers provide lightweight and portable execution environments for microservice. The Container orchestration platforms such as Docker Swarm and Kubernetes offer automated deployment, scaling, and management of containerized microservices. Efficient scheduling of containerized microservices is essential to ensure performance, scalability, and quality of service in container-based cloud environments. Docker Swarm's default scheduling policies are based on static heuristics. They have lack of awareness of dynamic resource conditions and application-level performance. However, efficient container scheduling remains a significant challenge due to dynamic workload conditions, heterogeneous resource availability, Quality of Service (QoS) needs, and increasing energy consumption in cloud infrastructures.

To address these limitations, this research proposes a Multi-Criteria Scheduling of Containerized Microservices using Reinforcement Learning (MCSCM-RL) framework for Docker Swarm environments. The proposed framework integrates multi-criteria decision-making and Deep Reinforcement Learning techniques to enable intelligent and adaptive container scheduling for microservices. The proposed scheduling approach integrates node-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 multi-criteria node-ranking mechanism based on a relative closeness function is employed to evaluate the suitability of cluster nodes, while a DQN based reinforcement learning agent dynamically selects optimal deployment nodes based on real-time cluster conditions and learned scheduling mechanisms. The proposed scheduling framework was implemented and evaluated using a Docker Swarm cluster comprising one manager node and two worker nodes deployed on Ubuntu virtual machines. Extensive experimental evaluation is conducted and compared with Random, Spread, Binpack, and non-RL MCSCM container scheduling strategies under clean and cumulative deployment scenarios.

Experimental results exhibit that the proposed MCSCM-RL scheduler effectively improves workload distribution, resource utilization, startup time, response time, and energy efficiency compared with traditional container scheduling approaches. The integration of reinforcement

learning enables adaptive scheduling behaviour through continuous interaction with the environment, while the multi-criteria node-ranking mechanism enhances decision-making by considering multiple performance indicators simultaneously. Future work will focus on extending the work to large-scale cluster, Kubernetes, and public cloud platforms.

Keywords: Containerization, Docker Swarm, Microservices, Reinforcement Learning, Deep Q-Network (DQN), Quality of Service (QoS)

Acknowledgement

The successful completion of this doctoral research would not have been possible without the support, guidance, and encouragement of many individuals and organizations. First and foremost, I express my sincere gratitude to my research supervisor Dr. Manish Patel for providing invaluable guidance, continuous motivation, constructive suggestions, and unwavering support throughout the course of my research work. Their expertise, patience, and encouragement have been instrumental in shaping this research study and contributing to its successful completion.

I would also like to extend my heartfelt thanks DPC members of my course Dr. Kirit Modi and Dr. Rajan Patel for their guidance and giving directions for the research. I would like to extend my heartfelt thanks Dr. Safavan Vahora and Dr. Nilesh Maltare for the constant support and guidance for my doctoral studies. I gratefully acknowledge the support to the Principal Sir, faculty members of Computer Engineering/Information Technology, Government Engineering College, Modasa for their support and cooperation during my doctoral studies. Their valuable discussions and suggestions greatly enriched the quality of this research.

I gratefully acknowledge the support provided by my institution for offering the necessary academic environment, computational resources, and research facilities required for conducting this work. My sincere appreciation goes to my dear friend Manmitsinh Zala for constant support during my research journey, and all friends who provided encouragement, technical assistance, and constructive feedback throughout this journey.

Finally, I express my deepest gratitude to my wife Ankita and my family for their unconditional love, patience, understanding, and constant encouragement. Their unwavering support has been the source of strength and inspiration throughout my doctoral studies. This achievement would not have been possible without their sacrifices and belief in me.

I thank all those who directly or indirectly contributed to the successful completion of this thesis.

With Heartful Gratitude
Prajapati Anilkumar Amrutlal

Table of Contents

List of Abbreviations

ML	Machine Learning
RL	Reinforcement Learning
DQN	Deep-Q-Network
ACO	Ant Colony Optimization
PSO	Particle Swam Optimization
MCSCM	Multicriteria Scheduling of Microservices
MCSCM-RL	Multicriteria Scheduling of Microservices- Reinforcement Learning
SLA	Service Level Agreement
DRL	Deep Reinforcement Learning
API	Application Programming Interface
QoS	Quality of Service

List of Figures

List of Tables

Abstract

Cloud computing has emerged as a dominant paradigm for deploying scalable and distributed applications through the adoption of microservices architecture and containerization technologies. Containers provide lightweight and portable execution environments for microservice. The Container orchestration platforms such as Docker Swarm and Kubernetes offer automated deployment, scaling, and management of containerized microservices. Efficient scheduling of containerized microservices is essential to ensure performance, scalability, and quality of service in container-based cloud environments. Docker Swarm's default scheduling policies are based on static heuristics. They have lack of awareness of dynamic resource conditions and application-level performance. However, efficient container scheduling remains a significant challenge due to dynamic workload conditions, heterogeneous resource availability, Quality of Service (QoS) needs, and increasing energy consumption in cloud infrastructures.

To address these limitations, this research proposes a Multi-Criteria Scheduling of Containerized Microservices using Reinforcement Learning (MCSCM-RL) framework for Docker Swarm environments. The proposed framework integrates multi-criteria decision-making and Deep Reinforcement Learning techniques to enable intelligent and adaptive container scheduling for microservices. The proposed scheduling approach integrates node-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 multi-criteria node-ranking mechanism based on a relative closeness function is employed to evaluate the suitability of cluster nodes, while a DQN based reinforcement learning agent dynamically selects optimal deployment nodes based on real-time cluster conditions and learned scheduling mechanisms. The proposed scheduling framework was implemented and evaluated using a Docker Swarm cluster comprising one manager node and two worker nodes deployed on Ubuntu virtual machines. Extensive experimental evaluation is conducted and compared with Random, Spread, Binpack, and non-RL MCSCM container scheduling strategies under clean and cumulative deployment scenarios.

Experimental results exhibit that the proposed MCSCM-RL scheduler effectively improves workload distribution, resource utilization, startup time, response time, and energy efficiency compared with traditional container scheduling approaches. The integration of reinforcement learning enables adaptive scheduling behaviour through continuous interaction with the environment, while the multi-criteria node-ranking mechanism enhances decision-making by considering multiple performance indicators simultaneously. Future work will focus on extending the work to large-scale cluster, Kubernetes, and public cloud platforms.

Keywords: Containerization, Docker Swarm, Microservices, Reinforcement Learning, Deep Q-Network (DQN), Quality of Service (QoS)

Acknowledgement

The successful completion of this doctoral research would not have been possible without the support, guidance, and encouragement of many individuals and organizations. First and foremost, I express my sincere gratitude to my research supervisor Dr. Manish Patel for providing invaluable guidance, continuous motivation, constructive suggestions, and unwavering support throughout the course of my research work. Their expertise, patience, and encouragement have been instrumental in shaping this research study and contributing to its successful completion.

I would also like to extend my heartfelt thanks DPC members of my course Dr. Kirit Modi and Dr. Rajan Patel for their guidance and giving directions for the research. I would like to extend my heartfelt thanks Dr. Safavan Vahora and Dr. Nilesh Maltare for the constant support and guidance for my doctoral studies. I gratefully acknowledge the support to the Principal Sir, faculty members of Computer Engineering/Information Technology, Government Engineering College, Modasa for their support and cooperation during my doctoral studies. Their valuable discussions and suggestions greatly enriched the quality of this research.

I gratefully acknowledge the support provided by my institution for offering the necessary academic environment, computational resources, and research facilities required for conducting this work. My sincere appreciation goes to my dear friend Manmitsinh Zala for constant support during my research journey, and all friends who provided encouragement, technical assistance, and constructive feedback throughout this journey.

Finally, I express my deepest gratitude to my wife Ankita and my family for their unconditional love, patience, understanding, and constant encouragement. Their unwavering support has been the source of strength and inspiration throughout my doctoral studies. This achievement would not have been possible without their sacrifices and belief in me.

I thank all those who directly or indirectly contributed to the successful completion of this thesis.

With Heartful Gratitude
Prajapati Anilkumar Amrutlal

Table of Contents

List of Abbreviations

ML	Machine Learning
RL	Reinforcement Learning
DQN	Deep-Q-Network
ACO	Ant Colony Optimization
PSO	Particle Swam Optimization
MCSCM	Multicriteria Scheduling of Microservices
MCSCM-RL	Multicriteria Scheduling of Microservices- Reinforcement Learning
SLA	Service Level Agreement
DRL	Deep Reinforcement Learning
API	Application Programming Interface
QoS	Quality of Service

List of Figures

List of Tables

Chapter 1

Introduction

Cloud computing has emerged as one of the most dominant computing paradigms in recent years. This paradigm offers on-demand access to shared computing resources and services over the Internet. The need for scalable and efficient cloud infrastructures has grown dramatically due to the quick development of Internet-based applications, Internet of Things (IoT) devices, machine learning applications, video streaming platforms, and cloud storage services. Cloud computing enables users and organizations to access computing resources such as processing power, storage, networking, and software services. It offers different service models including Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). The foundation of cloud computing is virtualization technology which isolates the underlying hardware infrastructure and make it possible to share computing resources effectively in an isolated environment. The flexibility, scalability, and cost-effectiveness offered by cloud computing platform have encouraged enterprises and industries over the world to shift their workloads to cloud platforms in order to increase the performance and business agility[1][2].

Microservices Architecture (MSA) has gradually replaced traditional monolithic application architecture due to the growing need of agile software development and continuous service delivery. In microservices-based systems, applications are decomposed into multiple lightweight, loosely coupled, and independently deployable services. In which each microservice performs a specific business functionality and interacts with other services using lightweight APIs. Microservices allows the ease of application development, maintenance, scalability, and continuous deployment by enabling independent updates and quick service delivery. The emergence of containerization technologies, have expedited the use of microservices by offering lightweight and portable execution environments for deploying applications and their dependencies. Compare to traditional virtual machines, containers offer quick startup time, near-native performance, portability, and efficient resource utilization. As a result, container orchestration platforms like Docker Swarm and Kubernetes are widely adopted for managing large-scale deployments of containerized microservices in cloud environments[3][4].

1.1 State of the arts of the research

Traditional software applications were basically developed using monolithic application architecture, which tightly integrates all application functionalities such as user interface, business logic, and database operations into a single deployable unit. Monolithic systems were initially easy to design, develop, and deploy for small-scale applications. However, monolithic applications started encounter a number of drawbacks as applications became larger and complex, such as poor scalability, increased maintenance complexity, longer deployment cycles, and reduced fault isolation. In monolithic systems, a minor change in one module often required redeployment of the entire application. It results in increased downtime and reduced development agility. Additionally, the tightly coupled nature of monolithic systems made it difficult for software developers to work independently and slowed down the process of continuous integration and continuous deployment (CI/CD).

The software industry gradually shifted toward microservices architecture to overcome these limitations. It enables modular application development, independent scaling of services, quick deployment, improved fault tolerance, and enhanced maintainability. Unlike monolithic application architecture, microservices permits to update and redeploy specific services without affecting the entire application. The adoption of microservices has become increasingly popular in cloud-based environments due to its support for agility, scalability, resilience, and continuous software delivery. Large technology organizations like Amazon, Netflix, and Google have widely used microservices architecture to efficiently handle large-scale distributed applications and dynamic workload conditions.

Containerization is a lightweight virtualization technology that enables applications and their dependencies to be packaged into isolated and portable execution environments. Compared to traditional virtual machines, containers guarantee process-level isolation. It shares the host operating system kernel, which lowers resource overhead and boost startup times. By encapsulating application code, libraries, runtime environments, and configuration files into a single deployable unit, containerization makes it possible for applications to operate reliably across many computing platforms. Because of its consistency and portability, containerization is a crucial technology for modern cloud-based applications, making software development, testing, deployment, and easy maintenance processes [5][6].

Containers offer several advantages over conventional virtualization technologies, including rapid deployment, reduced infrastructure overhead, improved scalability, and near-native performance. Containerization is well suited for microservice architecture, where applications are decomposed into multiple loosely coupled services which can be independently deployed and scaled. Each microservice can be packaged as a container, that enables continuous delivery and flexible deployment[3][7].

A container platform is a software architecture that allows the effective creation, deployment, management, and operation of containerized applications across various computer environments. Containerization platforms like Docker offer simplicity, portability, scalability, and effective use of computing resources, they have significantly increased the adoption of container-based applications in cloud computing environments. As a result, containerization has become a foundational technology for building scalable, agile, and efficient cloud-native systems. As the number of containers increases in large-scale cloud systems, container orchestration platforms like Docker Swarm and Kubernetes are used to automate deployment, scheduling, auto-scaling, networking, and management of containers across distributed cluster nodes[8].

The key process in container orchestration platforms is container scheduling that determines how and where containers are deployed across the nodes of a distributed cluster environment. In cloud-based systems, applications are typically deployed as multiple containerized microservices running on clusters managed by orchestration platforms such as Docker Swarm and Kubernetes. The container scheduler is responsible for selecting the most suitable node of the cluster to deploy the containers based on available resources and predefined scheduling policies. In large-scale cloud-based systems, effective container scheduling is very crucial for maintaining system stability, resource balancing, enhancing application performance, and guaranteeing dependable service delivery.

Existing container scheduling approaches of container orchestration platforms like Docker and Kubernetes often fail to consider runtime metrics such as CPU utilization, memory availability, startup delay, response time, dynamic workloads situations, dependencies among microservices, and energy consumption. Inefficient container scheduling decisions may lead to resource contention, increased latency, service degradation, underutilization of resources, and excessive power consumption in cloud datacentres[9]. In cloud datacentres, inefficient

container scheduling choices can result in resource contention, higher latency, service failure, underutilization of resources, and excessive power utilization. Modern cloud-based systems are becoming more sophisticated and dynamic, which has made need of intelligent and adaptable container scheduling algorithms. In addition, microservices-based applications need continuous scaling and dynamic resource management to maintain Quality of Service (QoS) under dynamic workload situations. Therefore, there is a requirement of advanced and intelligent container scheduling mechanisms, that can dynamically adapt to changing system states and optimize multiple objectives simultaneously. In order to achieve effective, QoS-aware, and energy-aware container orchestration in modern cloud-based systems, intelligent container scheduling approaches based on machine learning and reinforcement learning techniques have emerged as potential solutions.

1.2 Problem Definition

The deployment of containerized workloads in cloud-based systems has greatly grown due to the quick adoption of cloud-based applications and microservices architecture. Containers are ideal for modern applications like the Internet of Things (IoT), artificial intelligence, streaming services, and real-time distributed applications because they offer lightweight virtualization, scalability, portability, and quick deployment capabilities. To automate the deployment, scheduling, networking, and management of containerized microservices across cluster nodes, container orchestration technologies like Docker Swarm and Kubernetes are widely utilized. One of the most crucial features of these orchestration systems is container scheduling, which determines where containers should be placed on available cluster nodes in order to maximize resource consumption and improve application performance.

However, existing container scheduling strategies of Docker Swarm such as Spread, Binpack, and Random scheduling are mainly depended on static heuristic-based scheduling approaches, that are unable to handle effectively the dynamic nature of cloud environments. These traditional container scheduling techniques often consider only limited resource parameters (CPU, memory), while ignoring application-level metrics such as startup time, response time, workload fluctuation, energy consumption, and Quality of Service (QoS) needs. However, inefficient scheduling decisions may lead to resource contention, load imbalance, increased service latency, higher energy consumption, poor scalability, and degradation of overall system performance. In addition, in dynamic workload conditions microservices environments need

adaptive and intelligent container scheduling mechanisms that is capable of making real-time decisions based on current cluster states.

Improper workload placement and inefficient resource allocation can lead to excessive power consumption and operational costs. Existing container schedulers are not energy-aware and fail to optimize the compromise between performance and energy efficiency. Furthermore, most conventional container scheduling methods used for deploying containerized microservices do not possess self-learning or autonomous adaptation capabilities, that makes them less effective in highly dynamic and heterogeneous cloud environments.

Looking at these challenges, there is a critical need for an intelligent, adaptive, QoS-aware, and energy-aware container scheduling framework that is capable of dynamically selecting optimal nodes to deploy containerized microservices. The proposed research addresses these challenges by developing a Multi-Criteria Scheduling of Containerized Microservices using Reinforcement Learning (MCSCM-RL) framework for Docker Swarm environments. The proposed container scheduling approach integrates multi-criteria decision-making with Deep Reinforcement Learning techniques to optimize container placement decisions based on parameters such as CPU utilization, memory utilization, container count, startup time, and response time. Proposed scheduling approach is aimed to improve resource utilization, reduce latency, enhance QoS, and minimize energy consumption to ensure efficient orchestration of containerized microservices in cloud-based environments.

1.3 Objective and Scope of Work

Developing an intelligent and adaptive container scheduling system for the effective deployment of containerized microservices in Docker Swarm-based cloud environments is the main goal of this research work. By combining multi-criteria decision-making and reinforcement learning techniques for dynamic resource allocation and optimal container placement for microservices, the research proposes to overcome the limitations of conventional container scheduling approaches. Enhancing system performance, Quality of Service, and energy-efficient resource usage in cloud-based systems are the main objectives of the proposed scheduling framework.

The objectives of the research are as follows:

- To investigate and evaluate existing container scheduling strategies such as Random, Spread, and Binpack in container orchestration platforms.

- To develop a scheduling framework for Multi-Criteria Scheduling of Containerized Microservices (MCSCM) in Docker Swarm environments.
- To integrate resource-aware and application-aware parameters such as CPU utilization, memory utilization, container count, startup time, and response time into the scheduling decision process.
- To develop a Reinforcement Learning-based intelligent scheduling mechanism using Deep Q-Network (DQN) for autonomous and adaptive container placement.
- To implement dynamic node ranking and workload-aware scheduling for improving resource balancing and Quality of Service (QoS).
- To reduce startup time, response time, and energy consumption while improving overall cluster performance and resource utilization.
- To perform comparative performance evaluation of the proposed MCSCM-RL scheduling approach against existing container scheduling strategies including Random, Spread, and Binpack scheduling.
- To analyse the effectiveness of the proposed framework using various performance metrics such as CPU utilization, memory utilization, startup time, response time, and number of containers deployed on the cluster node.

The scope of this research primarily focuses on intelligent container scheduling in cloud-based environments using Docker Swarm container orchestration system. The work covers the design, implementation, and evaluation of an energy-aware and QoS-aware reinforcement learning-based scheduling framework for deploying containerized microservices on docker swarm cluster nodes. The experimental setup consists of a Docker Swarm cluster deployed using virtual machines, including swarm manager and two swarm worker nodes, for evaluating scheduling performance under different workload situations.

The proposed research also focuses on microservices-based applications deployed using Docker containers and does not extensively address virtual machine scheduling. The application of Deep Reinforcement Learning algorithms for dynamic scheduling decisions in small- to medium-sized distributed cloud system is the main focus of this study. However, the proposed framework can be extended in future work toward large-scale Kubernetes environments, edge-cloud systems, and public cloud platforms.

1.4 Research Contribution

The rapid growth of cloud-native applications and containerized microservices has created significant challenges in efficient resource management, Quality of Service (QoS) maintenance, and energy-efficient workload deployment in cloud environments. Existing container scheduling approaches are basically depending on static heuristic-based methods and often not able to adapt dynamically to changing workload situations and runtime resource needs. To tackle the issues with container scheduling, this research study proposes an intelligent scheduling system that uses reinforcement learning for Docker Swarm environments. The main research contributions of this thesis are summarized like this:

1. **Development of Multi-Criteria Container Scheduling Framework**
   This research proposes Multi-Criteria Scheduling of Containerized Microservices using Reinforcement Learning (MCSCM-RL) framework for Docker Swarm environments. The proposed scheduling framework performs intelligent and adaptive container deployment by considering multiple resource-aware and application-aware metrics.
2. **Combining QoS-aware and Energy-aware Scheduling approach**
   The proposed scheduling approach combines Quality of Service (QoS) and energy-aware metrics into the scheduling decision. In contrast to conventional container scheduling methods that mainly focus on resource availability. To achieve balanced and optimal container deployment, performance metrics like startup time, response time, CPU utilization, memory utilization, and container count are incorporated.
3. **Dynamic Node Ranking using Multi-Criteria Decision Making**
   A dynamic node-ranking mechanism based on multi-criteria decision-making techniques is designed to examine the suitability of cluster nodes for the deployment of containerized microservices. The node ranking process continuously updates node rank based on real-time cluster conditions and workload.
4. **Application of Deep Reinforcement Learning for Container Scheduling**
   For autonomous container scheduling in Docker Swarm environment, the study employs a Deep Q-Network (DQN)-based reinforcement learning scheduling method. Through constant interaction with the swarm cluster, the RL agent learns the best deployment policies and dynamically adjusts to changes in workload and resource availability. The RL agent is given a unique reward function that maximizes resource use and load balancing efficiency while minimizing response time, startup latency, and

energy consumption. The reward mechanism enables the container scheduler to make intelligent deployment decisions under dynamic situations.

5. **Implementation of Docker Swarm-based Testbed**
   In order to evaluate the proposed container scheduling method, a full experimental environment with a Docker Swarm manager and two worker nodes is implemented using virtual machine-based infrastructure. Automated deployment, monitoring, metric collecting, and comparative analysis of various container scheduling techniques are all supported by the testbed.
6. **Comparative Analysis with Existing Container Scheduling Strategies**
   The proposed MCSCM-RL framework is experimentally compared with existing container scheduling strategies including Random, Spread, and Binpack. Detailed performance analysis is conducted using performance metrics such as startup time, response time, CPU utilization, memory utilization, response time, startup time and number of containers deployed on the cluster node. Performance evaluation is measured under clean and cumulative deployment scenarios.
7. **Improvement in Resource Utilization and System Performance**
   The proposed scheduling framework exhibits enhanced workload distribution, reduced latency, better QoS, and enhanced energy efficiency compared to existing container scheduling methods. The framework effectively adapts to dynamic cluster conditions and improves overall system performance in cloud-based environments.
8. **Contribution toward Autonomous Cloud-native Resource Management**
   The development of autonomous cloud orchestration systems with intelligent decision-making for large-scale containerized environments is facilitated by this research work. Future studies in public cloud systems, edge-cloud orchestration, Kubernetes scheduling, and AI-driven datacentre management can build upon this proposed methodology.

1.5 Organization of the Thesis

The thesis is organized into six chapters that present the background, research methodology, implementation, experimental evaluation, and conclusions of the proposed research work.

Chapter 1: Introduction presents the background and motivation of the research work. It discusses cloud computing, microservices architecture, containerization technologies, and the importance of efficient container scheduling in cloud-based environments. The chapter also

highlights the limitations of existing scheduling approaches, defines the research problem, outlines the objectives and scope of the research work, presents the research contributions, and provides the overall organization of the thesis.

Chapter 2: Literature Review provides a comprehensive review of existing research related to cloud computing, containerization, microservice architecture, Docker Swarm, Kubernetes, and container scheduling strategies for microservices. The chapter discusses traditional scheduling techniques such as Random, Spread, and Binpack scheduling, along with recent machine learning and reinforcement learning-based scheduling approaches used in scheduling of containerized microservices. Furthermore, the chapter identifies research gaps and limitations in existing container schedulers that motivate the proposed work.

Chapter 3: Research Methodology and Problem Formulation describe the proposed Multi-Criteria Scheduling of Containerized Microservices using Reinforcement Learning (MCSCM-RL) framework. It presents the overall system architecture, problem formulation, workload model, and scheduling workflow. The chapter also explains the node-ranking mechanism, state-space representation, action-space design, reward function, and integration of Deep Q-Network (DQN)-based reinforcement learning into the container scheduling process.

Chapter 4: System Design and Implementation discuss the implementation details of the proposed framework in a Docker Swarm environment. The chapter explains the experimental setup, Docker Swarm cluster configuration, deployment of containerized microservices, monitoring framework, resource metric collection, and implementation of various scheduling strategies including Random, Spread, Binpack, MCSCM, and MCSCM-RL schedulers.

Chapter 5: Experimental Evaluation and Performance Analysis present the experimental results and comparative performance analysis of the proposed scheduling framework. Using performance metrics including startup time, response time, CPU utilization, memory utilization, and container count, the chapter compares the proposed MCSCM-RL scheduler against current container scheduling techniques. The effectiveness of the proposed scheduling strategy is illustrated by a number of graphs, tables, and statistical studies.

Chapter 6: Conclusion and Future Research Directions summarizes the main conclusions and contributions of the suggested research project to wrap up the thesis. The chapter highlights potential future research directions and highlights the efficiency of the MCSCM-RL scheduling system.

Chapter 2
Literature Review

The literature review chapter presents a comprehensive analysis of existing research related to cloud computing, microservices architecture, containerization, container orchestration, and intelligent container scheduling techniques. This chapter examines the evolution of cloud-based computing environments and discusses the role of Docker, Docker Swarm, and Kubernetes in managing large-scale containerized microservices. Various traditional container scheduling approaches such as Random, Spread, and Binpack scheduling are analysed along with recent advancements in heuristic-based, QoS-aware, energy-aware, and machine learning-based scheduling mechanisms. Furthermore, the chapter reviews reinforcement learning and Deep Q-Network (DQN)-based approaches applied to cloud environment for container scheduling. The strengths, limitations, and research gaps identified from the existing literature provide the foundation and motivation for the proposed Multi-Criteria Scheduling of Containerized Microservices using Reinforcement Learning (MCSCM-RL) framework presented in this thesis.

2.1 Cloud Computing and Virtualization

Cloud computing is a modern computing paradigm that offers on-demand access to shared computing resources such as servers, storage, networking, software, and applications via Internet. Without having to maintain physical infrastructure, it allows users and businesses to take advantage of flexible and scalable computing services as and when required. Cloud computing offers several advantages including cost reduction, resource scalability, high availability, elasticity, and efficient resource management. Depending on user requirements, cloud services are generally categorized into Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). The rapid adoption of cloud computing across various organizations has significantly transformed the deployment and management of large-scale cloud-based applications[10][11].

Cloud computing is built on virtualization technology, which makes it possible to share and abstract physical hardware resources effectively. Virtualization allows multiple isolated virtual environments to run simultaneously on the same physical machine, which improves hardware

utilization and reducing infrastructure costs. Through virtualization, applications and operating systems become decoupled from the underlying hardware architecture, enabling portability, flexibility, and simplified resource management. Virtualization technology has played a major role in enabling cloud service providers to support multi-tenant environments and dynamic resource provisioning[11].

Virtual Machines

A Virtual Machine is a software-based emulation of a physical system that runs its own OS and applications independently within an isolated virtual environment. Each virtual machine contains hardware resources such as memory, CPU, storage, and network in virtualized form, allowing multiple OS to execute concurrently on a single physical server. Virtual machines offer strong isolation, security, and fault tolerance between applications running on the same host system. They also support workload consolidation and migration, which are crucial aspects in cloud datacentres.

Virtual machines are widely used in cloud computing environments because they allow cloud service providers to efficiently allocate resources among multiple users while maintaining workload isolation. Software such as Xen,VMware, KVM, and Microsoft Hyper-V are commonly used virtualization platforms for creating and managing virtual machines.VM-based deployment offers flexibility and isolation, but because each VM instance must run a different guest operating system, there is a significant overhead[8][12].

Hypervisors

A hypervisor is a software layer which is responsible for creating, managing, and monitoring VMs on a physical computer system. It is also known as Virtual Machine Monitor. It abstracts the underlying hardware resources and allocates them to multiple virtual machines while ensuring isolation and controlled resource sharing. The Hypervisor enables multiple OS to run concurrently on a single physical host system by virtualizing memory, CPU, storage, and network resources.

Basically, Hypervisors are classified into Type-1 and Type-2 hypervisors. Type-1 hypervisors, run directly on the physical hardware without requiring a host OS. Type-1 is also known as bare-metal hypervisor. Examples include Xen, VMware ESXi, and Microsoft Hyper-V. These hypervisors provide improved performance, scalability, and security, making them suitable for enterprise cloud datacentres. Type-2 hypervisors run on top of a host OS and are mainly used

for desktop virtualization and development purposes. Type-2 also known as hosted hypervisors. VMware Workstation and Oracle VirtualBox are examples of Type-2 hypervisors.

Limitations of VM-based Deployment

- Despite their widespread adoption in cloud computing, VM-based deployments suffer from several limitations that affect performance, scalability, and efficiency of the resources. One of the primary drawbacks of virtual machines is the inclusion of separate guest operating systems for each VM instance that causes high resource overhead. When compared to lightweight virtualization solutions like containers, each virtual machine needs dedicated operating system resources, memory consumption, and storage requirements[13].
- Another limitation of VM-based deployment is the slow startup and resource provisioning time associated with initializing full operating system environments. Virtual machines typically require several minutes to boot, which reduces deployment agility and limits rapid scaling capabilities required by modern cloud-native applications. Additionally, VM-based infrastructures often experience inefficient resource utilization because hardware resources may remain underutilized when workloads fluctuate dynamically[14][15].
- In addition, maintaining and managing large numbers of virtual machines increases operational complexity in cloud environments. The heavyweight nature of VMs also impacts system performance and increases energy consumption in cloud datacentres. These limitations have motivated the transition toward lightweight containerization technologies, which provide faster startup time, improved portability, lower overhead, and better scalability for microservices-based cloud applications.

2.2 Containerization

When an application is being upgraded on a VM, there is significant service outages. However, containers are a potential lightweight virtualization approach for cloud-based services, especially for microservices, edge computing, and IoT-based applications, because of their quick startup times and better performance. Fig. 1 illustrates the extremely different architectural approaches for VMs and container technologies. A container engine is the top layer of the stack's resource management system when it comes to container architecture. In

contrast to the layer in the VM architecture known as the hypervisor, it is located directly on top of the host OS [16]. One notable feature of the container-based architecture is that containers can run without the guest OS. Because the hypervisor layer is eliminated, containerization provides portability, efficiency, and lower overhead when compared to virtualization.

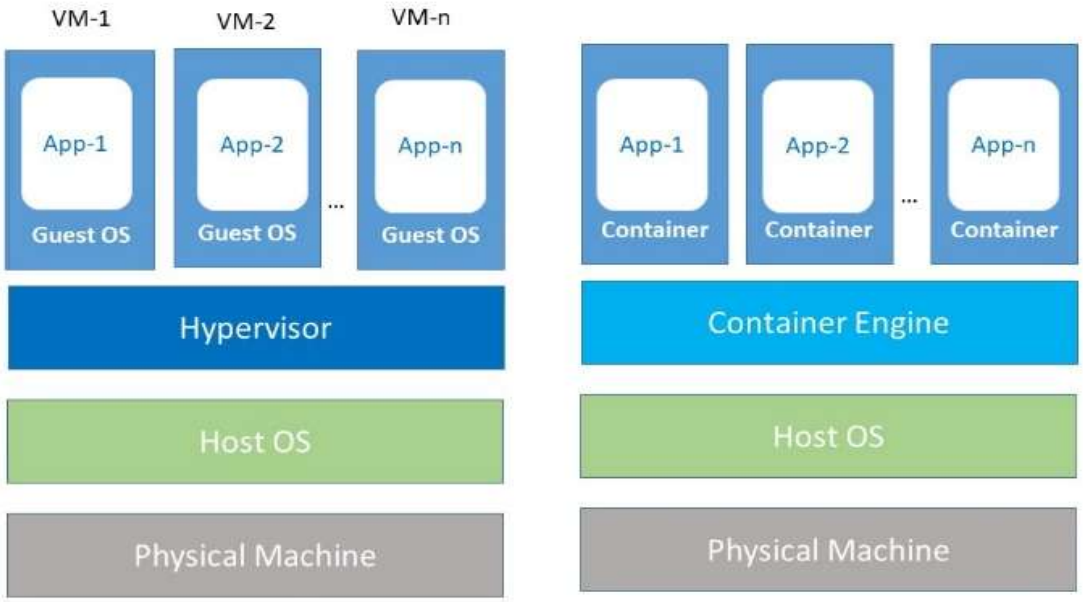


Fig. 1. Virtual Machines and Containers

Additionally, it is discovered that services implemented using containers outperform virtual machines in terms of throughput. The deployment of VMs for real-time applications was also found to have a large performance overhead, as seen by the memory and CPU usage. Furthermore, it is stated that AWS launches Docker containers on top of Amazon EC2 virtual machines, which differs from the typical application deployment process depicted in Fig. 2 shows how Amazon Cloud Platform deploys containers.

Container technologies are widely used by developers to deploy different microservices. Although containers are widely used in cloud computing, there isn't a thorough study that looks at container scheduling strategies from the perspective of containerized microservices. Developers that create and implement application containers must enhance the performance of the infrastructure when it comes to application deployment. This challenge to cloud providers promotes the need for different types of container scheduling algorithms for the deployment of containerized microservices.

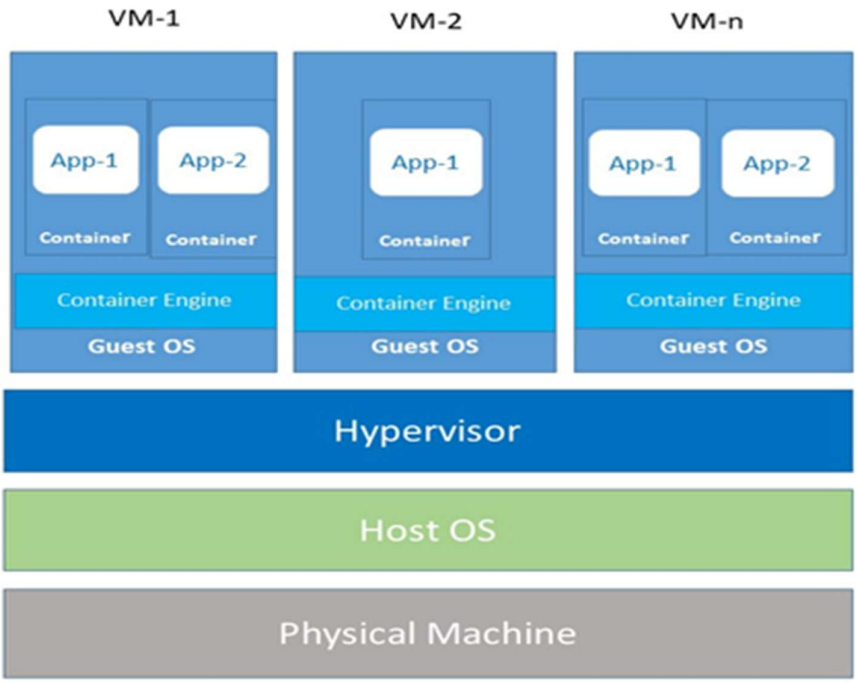


Fig. 2. Architecture of Container Hosting in Public Cloud Environments

2.3 Container Engine Architecture

Software developers create and build container images. Container images are files that have the data required to run a containerized microservices. Developers generate read-only, non-modifiable template of containers using containerization engines. Containerization tools allow users to build containers on top of any infrastructure. Docker is currently the most popular container solution on the market. Here, the idea of containerization is shown using the Docker engine [17].

Docker is an open-source containerization management tool that makes it easy to build and deploy container-based services. By using Docker containers to isolate the applications from the infrastructure, users can deploy microservices or applications more rapidly. The core elements of Docker are Docker Engine, Docker Image, Docker Swarm, Docker Compose, and Docker Daemon. This infrastructure might be managed similarly to an application. Fig. 3 depicts the architecture of the Docker container engine [13].

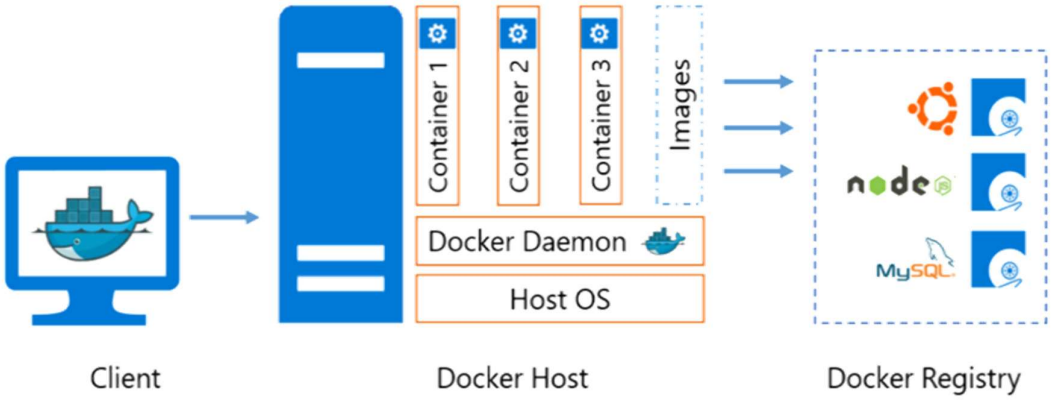


Fig. 3. Architecture of Docker

The container is a running instance of a Docker image. The instructions for creating a Docker container are included in a read-only template called a Docker image. The Docker engine is the part of Docker that builds and maintains Docker containers. The containers are managed and controlled by a process called the Docker daemon. The Docker client is used by Docker users to communicate with Docker containers. A Docker registry is where the Docker image is kept. Containers can be started, stopped, or removed using the Docker CLI (Command Line Interface) [13].

Container Lifecycle in Docker Engine:

The container lifecycle represents the sequence of states through which a container passes from image creation to container removal. Here Docker-based environment is considered for illustration of container life cycle. The Docker Engine is the core runtime responsible for building, running, and managing containers. It provides the required functionalities for container creation, execution, monitoring, networking, storage management, and termination. Understanding the container lifecycle in Docker Engine is essential for efficient deployment and management of containerized microservices in cloud environment.

Making a Docker image is the first step in the lifecycle. A read-only template known as a Docker image contains the runtime environment, libraries, dependencies, application code, and configuration files required to run an application or microservices. A Docker file, which contains a collection of building instructions, is used to create Docker images. After the Docker image is created, it is either pushed to a container registry like Docker Hub or saved locally, from where it may be fetched and accessible for container deployment [2][18].

Once the Docker image is available, Docker Engine creates an instance of container from the Docker image. It can be done using Docker commands. During this stage, Docker allocates resources like CPU, memory, networking interfaces, and storage to the created container. Initially, the container enters the created state, where the container exists but the application process has not yet started. When the container execution begins, it transitions to the running state, where the main application process inside the container becomes active. Since containers share the host operating system kernel, Docker containers start much faster and consume fewer resources compared to conventional virtual machines.

Docker Engine continuously manages the operational state of containers. A running container may communicate with other containers through Docker networking mechanisms. Docker also provides monitoring and logging functionalities to track container performance, and resource utilization. If required, containers can be temporarily suspended using the paused state, where all running processes inside the container are frozen without terminating the container. The container can later resume execution from the same state.

When the application process inside the container completes or receives a termination signal, the container transitions to the stopped or exited state. In this state, the container no longer executes any process but its metadata, file system changes, and logs remain available. Docker Engine allows stopped containers to be restarted if needed. Finally, containers that are no longer required can be permanently deleted using Docker commands such as docker *rm*. Once container is removed, all resources associated with the container are released, although the original Docker image remains available for creating new container instances.

In large-scale microservices environments, Docker Engine works together with container orchestration platforms such as Docker Swarm to automate container lifecycle management operations including deployment, scheduling, scaling, restart policies, failure recovery, and load balancing. Efficient lifecycle management in Docker Engine is critical for achieving scalability, fault tolerance, rapid deployment, and optimized resource utilization in cloud-based applications.

2.4 Microservice Architecture

Microservices present a new paradigm for developing cloud-based applications. Microservice design divides an application into multiple loosely coupled, fine-grained services. A microservice is an independently scalable, deployable component of an application. The microservice-based approach is contrasted with the conventional "monolithic" approach to application development, where each application is a single independent unit. Due to their massive user bases and high scalability requirements, several companies, such as Uber, Netflix, and Amazon, are moving from traditional monolithic architecture to microservice architecture. Fig. 4 compares the monolithic and microservice architecture. Microservices are scaled by distributed among cloud servers and replicated as needed, whereas monolithic programs are scaled by replicating them on several servers.

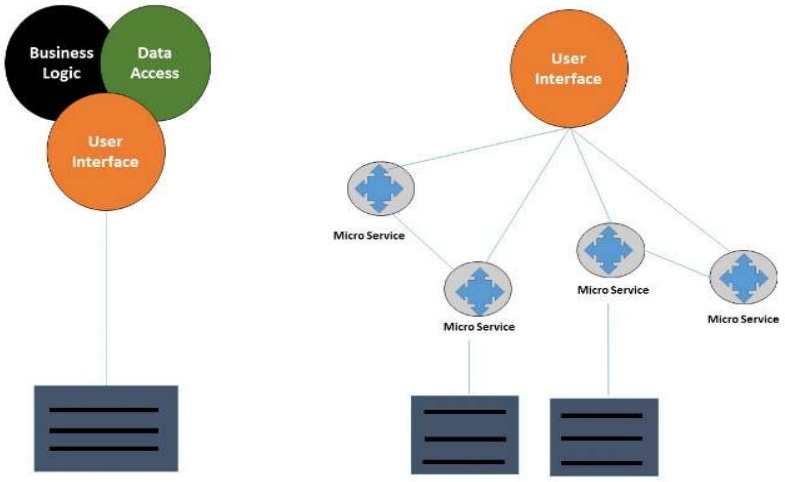


Fig. 4. Evolution from Monolithic and Microservice architecture

Microservices are autonomous, compact, and user-friendly components of an application as opposed to monolithic architecture. The autonomy of the microservices refers to their ability to change and be executed independently. Microservices, which are usually managed by distinct development teams and organized around business logic, are autonomous in their creation, deployment, testing, and operation. Furthermore, a variety of technologies and programming languages may be used to develop the microservices. Container-based virtualization makes microservices scalable and deployable. Fig. 5 shows how a microservice instance is deployed in a cloud container.

A new microservice have to register itself. It can be done by saving runtime configurations of it and modify the deployment information, it makes possible the communication between microservices of the same application. Each microservice will be built separately and will have its own development framework. The deployment server pulls the source code from the repository, compiles it, and tests it to produce microservice binaries. Using these binaries, the deployment server creates a container image that is stored in the container repository. Once built, these container images are deployed to the deployment server. One or more containers would host the microservice [19].

Deployment of Microservices in container

The deployment of microservices in containers is a fundamental approach in modern cloud-based application development, where each microservice is packaged along with its dependencies into an isolated and lightweight container. In a microservices architecture, applications are divided into multiple independent services, with each service responsible for a specific business functionality. Each microservice would have multiple instances. Containers provide a deployment platform for microservices because they ensure portability, scalability, consistency, and rapid deployment across different computing environments as shown in Fig. 5. Using Docker, developers can package application code, runtime libraries, configuration files, and dependencies into Docker images, which can then be instantiated as containers on any host system running Docker Engine. This container-based deployment model simplifies continuous integration and continuous deployment (CI/CD) processes by enabling independent development, testing, and deployment of individual microservices without affecting the entire application.

Container orchestration systems like Docker Swarm and Kubernetes are used in large-scale cloud systems to automate the deployment and management of containerized microservices. Container Orchestration systems schedule containers among cluster nodes and distribute resources according to scheduling strategy and resource availability during deployment of container. Loosely connected interactions between microservices are become possible by containers using lightweight network protocols and APIs. Containerized deployment also supports dynamic scaling, fault tolerance, service discovery, and load balancing, which are essential for maintaining application performance and availability under varying workload situations. The lightweight containers permit microservices to start quickly and consume fewer resources compared to traditional virtual machines. That makes containerized microservice deployment highly suitable for scalable and agile cloud-native computing environments[20].

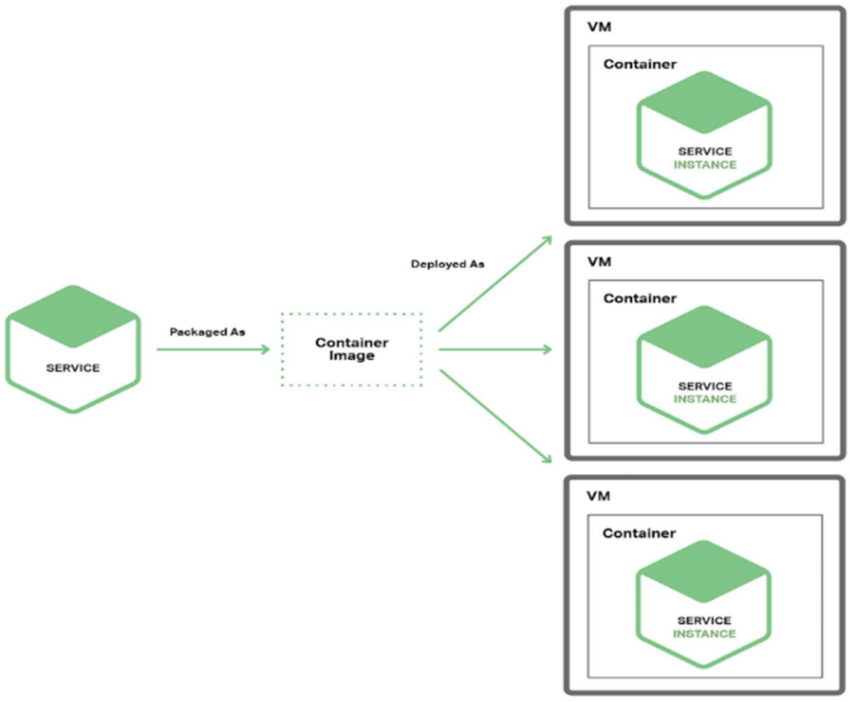


Fig. 5. Deployment of Microservice in container

The developers can update, modify, or redeploy individual microservices without affecting the operation of the entire system or application. The microservices offers improved application flexibility, maintainability, and scalability. That makes it most suitable for modern cloud-native environments. Additionally, containerized deployment provides several essential capabilities, that help to maintain application performance and availability under dynamic workload situations. The ability to quick instantiate and terminate, containers enable organizations to respond efficiently to dynamic user needs, it optimizes the infrastructure utilization at the same. Thus, containerized microservice deployment has become a best choice for building scalable, resilient, and agile cloud-based applications.

Challenges in Deployment of Microservices in Containers:

Despite the flexibility, scalability, and quick deployment capabilities that containerized microservices offer, deploying them in distributed cloud systems presents a number of operational and technical issues. Effective resource management and container deployment are two of the main issues. Several containers competing for limited resources like CPU, memory, storage, and network bandwidth in large-scale microservice-based systems. Resource contention, load imbalance, longer response times, and poor application performance can result from improper container deployment. Ineffective distribution among cluster nodes may further raise network latency and communication overhead because microservices frequently connect

with one another. Additionally, traditional static container scheduling techniques struggle to maintain appropriate resource allocation and Quality of Service (QoS) due to dynamic workloads.

Managing the complexity of container orchestration, scalability, monitoring, and fault tolerance in containerized environments is another major issue. Maintaining service dependencies, inter-container communication, service discovery, and data consistency gets more difficult as the number of microservices rises. If container orchestration frameworks fail to adequately handle container failures, node breakdowns, and network disruptions may impact application availability and reliability. Furthermore, quick container creation and continual scaling may result in higher startup time and energy usage in cloud datacentres. Because containers share the host operating system kernel and are therefore more susceptible to some security risks than fully isolated virtual machines, security and isolation are still major considerations. Therefore, intelligent orchestration and adaptive scheduling mechanisms are essential for ensuring efficient, secure, and reliable deployment of containerized microservices in cloud-based environments.

2.5 Scheduling for Containerized Microservices

In order to ensure the performance of deployed containerized microservices of the application, the container-based architecture imposes emerging requirements for container scheduling. The physical machine or cloud cluster should offer the underlying resources requested by container, such as memory, CPU, network, etc. Containers inside an application often have a strong affinity for one another due to frequent upgrades and data transfers. Therefore, affinity needs to be considered while scheduling containers. The container scheduling techniques are frequently used during the application execution, particularly when scaling out or recovering from failure, which usually entails critical time limitations. As a result, scheduling overhead should be minimal [21][22].

Classification of Container Scheduling

Recent research work on scheduling of containers has been reviewed by us and it was found that, there are basically the four types of container scheduling methods under studied and each had different performance and quality features. The many of the container scheduling approaches are based on Mathematical modelling-based, Heuristics based, Meta-heuristics-based, and ML-based techniques [22].

Using a set of linear constraints, mathematical modelling allows for the optimization of a linear function; this method is known as integer linear programming. It does not provide optimal solutions in a timely manner for the container scheduling. The majority of research studied have suggested linear programming models for the scheduling of containers that take different optimization criteria in consideration and give effective outcomes. Mathematical modelling has been found to be appropriate for modest problem sizes.

After an extensive literature studies, it is observed that compared to other modelling, the heuristic techniques are preferable for the scheduling of containerized microservices. To determine the best container scheduling, most studies in recent years have employed some sort of heuristic. Heuristic algorithms are often easy to apply and provide fast and efficient container scheduling. The majority of these heuristic methods integrate existing container scheduling strategies and implement them in container orchestration platforms like Docker Swarm and Kubernetes. A method for the effective use of resources has been demonstrated by some researchers. When compared to Docker Swarm, the Docker-based Container scheduling framework has demonstrated an improved energy-efficient solution and suggested the best-fit container scheduling that provides dynamic monitoring. A multi-objective approach that combines all three (Spread, Binpack, Random) of Docker Swarm to provide efficient container scheduling is proposed with considerations of CPU, memory, network delay, and communication between cluster nodes and containers, and showed efficient performance. It is observed that these approaches provide fast optimization. Additionally, these methods can be combined with other optimization approaches for enhanced container scheduling performance.

Nowadays, it is observed that to get optimum solution in a variety of domains, meta-heuristic methods are very promising. These scheduling approaches are adaptive to the environment and they can be mainly based on Genetic algorithms, ACO, and PSO. Kawasaki (2017) presented ACO based approach to improve resource utilization and balanced load. The scheduling mechanism was tested on Docker Swarm and showed improved performance. For the scheduling of containerized microservice, Lin (2019) presented a multi-objective ACO scheduling technique that is used to optimize network transmission overhead, failure rate, and resource usage of containerized microservices. Results exhibit better resource utilization and reliability. Genetic algorithms and Particle Swarm Optimization-based container scheduling methods have also been presented in recent years by the researchers. Li (2018) proposed a PSO-

based technique to for enhanced load balancing and resource utilization. Results exhibit improved performance compared to the default Spread scheduling technique of Docker Swarm.

Additionally, it is observed that meta-heuristic methods are appropriate for the multi-objective optimization. A rapidly developing field of research, machine learning has demonstrated enormous potential in several applications across a wide range of industries. Container scheduling has a lot of potential with machine learning. Machine learning techniques are particularly studied in relation to scheduling the containerized microservice.

Performance Goals and Evaluation Criteria for Container Scheduling

Scheduling decisions usually need to achieve many performance goals based on specific user requirements. When deploying microservices in containers, to maintain a balance between optimization objectives required for container scheduling is a major issue. The objectives given below are most typical for scheduling containers in cloud environment. The following goals and evaluation criteria are believed to be the common for scheduling of containers in cloud environment.

- **Energy efficiency:** As cloud datacentres continue to grow in size, the high quantity of electricity needed by cloud computing infrastructure is being a key issue in the domain of cloud computing. Energy consumption is the amount of electricity needed when containers are deployed on nodes of the cloud datacentres. The objective is to reduce the overall power consumption of the cluster.
- **Cost:** The price of various services, like processing, storage, and communication, determines the overall cost of running an application. The compute cost is the amount of time required to run an application or microservice on the node of cloud cluster. An application grows more expensive the longer it runs.
- **Availability:** It indicates how longtime a user has access to an application or microservice. Availability is a crucial objective of cloud computing infrastructure, where users should be able to access a microservice or application all-time. The goal is to ensure that the created scheduler offers fault tolerance for the various microservices that the application offers, necessitating redundancy in the quantity of cloud containers.
- **Resource Utilization:** It is the efficiency with which a cluster node makes use of its network bandwidth, memory, and CPU. A cluster node's cost-effectiveness and energy efficiency rise with its resource utilization to achieve the objective. Infrastructure-level resource utilization metrics, such as CPU and memory, are usually considered key performance indicators for energy and cost efficiency.

- **Load Balancing:** It is the process of distributing the workload equally among the available resources so that no node is overworked. The load balancing affect response time, cost, and throughput. This precaution is essential, especially for container systems that rely on microservices.
- **Scalability:** A crucial parameter, that indicates the capacity of containers to sustain the necessary service even when system ask need for the application or service increases.
- **Latency:** It is the amount of time required for a microservice to run from start to end. The goal of effective scheduler is always to reduce latency. This goal has a significant impact on throughput and cost.
- **Security:** Capacity to use encryption and access control techniques to shield data and services from malicious activities or software defects. Essential security must be provided by container orchestration solutions.
- **Network Bandwidth:** Network bandwidth is the quantity of bits that are carried over a network in a one second. It would help in managing congestion that arise during container communication and evaluating network delays.
- **Carbon Emission:** It is the amount of carbon dioxide released into the atmosphere when electricity is used.
- **SLA assurance:** Strict performance criteria, such as throughput, start time, completion time, and response times, are set up for most containerized microservices. Most of the limitations are specified in SLA contracts, and violating them could result in penalties.

2.6 Container Orchestration

Organizations frequently employ containers to implement cutting-edge apps like Internet of Thing and big data in cloud datacentres. Consequently, container orchestration has been developed. The ability to define, choose, deploy, monitor, and dynamically control the configuration of containerized microservices in the cloud is offered by container orchestration systems. The Container orchestration tools, like Docker Swarm and Google Kubernetes, performs the containers scheduling, choose the best node, and make sure the container is operating in the intended state. By delivering the same application or microservice without having to rebuild it in the cloud environment, container orchestration solutions allow organizations to expedite application development [23].

The difficult container scheduling challenge has a major impact on containerized microservices deployment, cost, and performance. Container orchestration does more than merely move applications between servers and turn them on and off. Orchestration systems for containers allows users to decide how to manage the containers of microservices when a multi-container application is launched. The initial deployment of containers and their management as a single unit are referred to as container orchestration [9][21][22][23].

The section explains the taxonomy of container orchestration in the perspective of the scheduling of the containerized microservices. With concerning container technique used, the application model, placement criteria, and resource granularity, we have analysed the existing container orchestration systems. Container orchestration platforms for the objectives such as scalability, availability, throughput, and resource utilization are also thoroughly studied. The summary of the study is presented in Table-1. Widely used Container orchestration tools are as follows.

- **Docker Swarm:** The application or service operating in a Docker container is coordinated by Docker Swarm. It is an integrated container scheduling and clustering system for Docker containers. Applications are handled as services; they can be deployed in one or more containers and decomposed into microservices. Swarm workers are used to deploy the containers, and the Swarm manager is used to manage the whole life cycle of containerized services in Docker containers. By default, it supports three baseline container scheduling techniques. The first scheduler is Spread, and it selects the newly deployed containers to run on the node with the lowest load. Second, Binpack scheduler, which elects the most loaded node with sufficient resources to deploy the containers. The third is the Random scheduler, which elects any random node for the deployment.
- **Kubernetes:** Kubernetes is an open-source container orchestration platform developed by google that offers the deployment and scaling of containerized applications or microservices. The primary building block of the Kubernetes container orchestration platform is the pod; it is a group of containers. Kubernetes container scheduler offers placement for each pod based on resources available on the cloud cluster, filtered by user-based needs, and ranked according to application or service affinities defined. The Kubernetes container cluster consists of Master and Worker nodes, where the master node is the basic component and the worker nodes are used to run pods of containers.
- **Mesos:** Container scheduling is done in two steps by Apache Mesos. During the first stage, known as the framework, it assigns the physical resources of node to each

application sequentially. The framework can then schedule its tasks based on the obtained resources of the node using its built-in framework scheduler. Service discovery is not provided by Mesos. In order to fill this gap, Kubernetes or Swarm scheduler are incorporated. Mesos is used by the orchestration frameworks Aurora and Marathon to control the resources of container cluster.

- **Aurora:** is designed by Twitter, an orchestrator that runs on top of Mesos and offers long-running microservices to be deployed on the container.
- **YARN:** It is developed for orchestration of Hadoop operations, it offers support for frameworks like Giraph, Spark, and Storm. The execution flows and optimizations of each application framework that runs on top of YARN are coordinated.
- **Omega:** It is Google's next-generation cluster management system. It gives users a lot of flexibility by offering a framework for the development of specialized and customized container schedulers.
- **Fuxi:** It is an Alibaba's proprietary data platform is supported by the scheduling and resource management system. The physical resources of Linux clusters in a datacentre are managed by their Aspara system's resource management module [20].

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

2.7 State-of-the-Art Container Scheduling Techniques

There are several difficulties in deploying containerized microservices, the most of them are associated with configuration and planning of container deployment. An ideal container scheduling strategy is necessary to satisfy the diverse needs of microservices. For the application to be predictable and available, the microservices must be monitored and their resources automatically adjusted in response to dynamic workload conditions. The scheduling of containers in cloud environments is complicated by these needs. The available resources must be scheduled for effective use, and the chain of microservices must be organized and controlled. Less efficient container scheduling directly affects the system's dependability and availability. Additionally, it is often necessary to research container scheduling for microservices in order to analyse and improve it further. This section presents the research and development regarding to containerized microservices scheduling.

A model to enhance the container resource allocation strategy for microservice-based applications was created by Lin et al. [24]. This model illustrates the decreased load, network transmission between microservices, and improved cloud cluster service dependability. The mean value of microservice request failures, the high resource consumption rate, and the network transmission overhead across containerized microservices were employed by the researcher to do this. Ant colony optimization (ACO) was used to build the proposed technique, which was then compared to the current container scheduling strategies for resource consumption and the average of microservice request failures. The result shows that performance has improved.

Fard et al. [25] introduced a microservices scheduling strategy that uses a multi-objective optimization approach to solve the scheduling problem, which is described as an improved version of the knapsack problem. The proposed scheduling method is extremely scalable and simultaneously boosts CPU and memory utilization, which improves throughput, according to the results. The proposed approach works better than the standard Docker Swarm container scheduling methods, Binpack and Spread in terms of throughput.

Zhao et al. [26] introduced a new Microservice-based container framework to execute delay-sensitive applications at the lowest possible cost. The results exhibit that the proposed scheduling approach have decreased costs, improved resource utilization, and minimum service delay. Menouer et al. [20] developed a container scheduling approach that integrates the Binpack scheduler and the Spread scheduler. The suggested scheduling mechanism and the

node that strikes the right balance between the three performance criteria are tested in Docker and contrasted with Spread, Binpack, and Random scheduling methods. The outcome shows the improved performance in different workload situations.

Kaewkasi et al. [27] introduced the first ACO-based method for container scheduling to increase resource utilization through load balancing. The proposed scheduling technique was tested using Docker Swarm. When compared to the Swarm Greedy approach, the results showed an improvement in performance. However, only optimization parameters, such as load balancing and resource utilization, have been taken into consideration. The issues of container scheduling and scalability of containerized microservices in cloud systems are the main emphasis of Soulemez et al.'s research [28]. According to the researcher, streaming workloads cause baseline container scheduling techniques to function poorly. In order to overcome these challenges, researchers designed an elastic scheduling system based on a variable-sized Binpacking problem that manages job scheduling and auto-scaling.

Machine Learning-Based Approaches for Container Scheduling

These days, automation of container orchestration for complex and heterogeneous workload situations in cloud computing is highly unclear. In the literature survey, we observed that machine learning has been applied in orchestration of virtual machines. Existing container scheduling strategies are basically developed with heuristic scheduling techniques that do not consider dynamic workload situations and QoS needs of the microservice-based applications. Many of these scheduling mechanisms are used for small-scale systems that are configured statically based on specific workload situations only. Particularly for container scheduling, such techniques are not able to handle highly dynamic workloads where services required to be scaled at any time according to specific behaviour patterns. Heuristic scheduling techniques may perform noticeably worse as the system expands, and component dependencies are taken in account for resource provisioning of containerized systems. Within an application, many microservices may contain internal relationships and depend on each another. Increased resource needs and communication costs have a negative impact on microservice-based application.

Current container scheduling techniques focus mainly on assessing infrastructure-level metrics; application-level metrics and Quality of Service requirements are not sufficiently taken into account. Cloud service providers have been forced to employ machine-learning based

approaches to optimize their container orchestration strategies due to the growing complexity of applications on cloud systems.

Thus, parameters at the infrastructure or application level, such as workload characteristics, system resource consumption, and application performance, could be modelled and predicted using machine learning techniques. Heuristic methods could be directly replaced by machine learning algorithms, which provide more accuracy and reduced time overhead in large-scale systems for making resource management decisions. We looked into research studies that take machine-level resource utilization into account to enhance service level agreement (SLA) and energy efficiency of the datacentres. The proposed model was experimented on ContainerCloudSim and demonstrated lower energy consumption [29]. Thus, parameters at the infrastructure-level or application-level, such as workload characteristics, resource utilization, and application performance, could be used to model and forecast using machine learning techniques. Heuristic-based scheduling methods have traditionally been used for container resource management due to their simplicity and low computational requirements. However, these approaches often struggle to adapt to dynamic workload variations and complex cloud-native environments. Machine learning-based scheduling techniques offer a more intelligent alternative by learning from historical and real-time system data to make adaptive deployment decisions.

An evaluation of machine learning-based container scheduling techniques was provided by Muniswamy et al. [30]. In 2016, the resource usage of containerized microservices was estimated using K-nearest neighbour. Nevertheless, the application model solely took into account the infrastructure-level resource metric's time series behaviour. The long short-term memory (LSTM) model, which is based on neural networks and appropriate for processing, forecasting, and classification, was used for dependency analysis of microservices.

Machine learning has become extremely popular in recent years and is used extensively in numerous fields of study. We discovered that the popular machine learning models in the container scheduling field are regression, classification, and decision-making, based on their optimization goals. The overall performance of the application and resource efficiency are directly impacted by the quality of selections of scheduling mechanisms. Our survey revealed that some researchers have been using machine learning-based scheduling models in recent

years, especially for the scheduling of the containerized application. The study on machine learning-based container scheduling was presented by Nanda et al. [31]. Optimizing resources under dynamic workload situations is extremely challenging. To enhance resource utilization, the author has suggested a reinforcement learning-based method for container consolidation. When compared to Random and Shortest Job First scheduling methods, the proposed method produced better results. A random forest regression model has been proposed by one of the authors to forecast future container requirements. The future resource requirements of containers were correctly forecasted by the model.

While genetic methods frequently encounter long convergence times, Liu et al. [32] presented a multi-task genetic programming strategy for the multi-objective container placement in heterogeneous cloud clusters. Lee et al. [33] presented a functionality-aware container offloading technique was developed for edge computing, which exhibits enhanced service latency performance but need extensive application-level annotations. In an extensive literature survey of AI based approaches applied in the microservices lifecycle, Moreschini et al. [34] identified RL and meta-learning as new areas of research. Research is forecasting and learning based container orchestration has been growing. Al Qassem et al. [35] developed an AI-based predictive container orchestration system that enhanced container placement stability but lacks runtime learning system.

Zhang et al. [36] presented a pattern-based scheduler for microservice workflows, which exhibits enhanced throughput but basically focused on workflow-structured services or applications. For resource-constrained edge computing, Huang et al. [37] developed a graph-based improved DQN model, which shows excellent flexibility with the significant computational overhead. For adaptive microservice resource allocation, Wang et al. [38] presented a model that is combination meta learning and super heuristic approaches, which offered fast task-based learning. Fraihat, et al. [39] presented an extensive literature review of AI driven job scheduling methods, it shows the industry wide transition toward Reinforcement Learning-based and QoS-aware container orchestration systems.

The results of the comprehensive literature review on machine learning-based container scheduling approaches are summarized in Table 2. The table presents a comparative analysis of various intelligent scheduling techniques proposed for the scheduling of containerized microservices in cloud environments. Consequently, many researchers have explored the application of machine learning and artificial intelligence techniques to enable adaptive, data-

driven, and intelligent scheduling approaches in cloud environments. Furthermore, the findings from Table 2 provide valuable insights into the strengths and limitations of existing machine learning-based container scheduling techniques used for the scheduling of containerized microservices.

A summary of the meta-heuristic and machine learning-based container scheduling techniques, along with their corresponding optimization objectives, is presented in Table 3. The table provides a comprehensive overview of the intelligent scheduling approaches proposed by various researchers for managing containerized microservices in cloud environments. As container orchestration platforms continue to support increasingly complex and dynamic microservice applications, traditional scheduling strategies have proven inadequate in addressing multi-dimensional optimization needs.

Table 2. Machine Learning-Based Scheduling for Containerized Microservices

Problem in scheduling of containerized applications	Machine Learning based approach	Benefits/ Limitations
How can the interactions and dependencies among microservices be modelled and evaluated?	BO, GP, CNN, and LSTM	Improved forecast accuracy/ Ignorance of dependency updates
What approaches can be used to analyse task dependencies in containerized microservice applications?	SVM	Enhanced Accuracy/Implicit explanations of computational costs and time overhead
What strategies can be employed to minimize energy consumption during container scheduling and deployment?	MDP, Q-learning, and SARSA	Cost saving, Reducing task execution time/ Limited scalability
How can communication performance be improved during the migration of microservice-based applications to cloud-native container infrastructures?	DNN, Q-learning	Reducing task execution time/Limited scalability
How can request behaviour and resource usage patterns be utilized for workload characterization?	ARIMA, Bi-LSTM, LSTM, K-means++, GRU, TSNNR	Optimized Resource utilization/ Scheduling latency

Analysis has revealed that methods based on machine learning and meta-heuristics are more appropriate for multi-objective optimization.

Table 3. Literature Review of Container Scheduling and its Optimization Objectives

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	Just forecast the resources and absence of real-time testing.
✓		✓	✓			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

Reinforcement Learning-Based Approaches for Container Scheduling and Orchestration

In recent years, machine learning (ML) and reinforcement learning (RL) driven scheduling have essentially replaced static heuristics to handle resource heterogeneity, dynamic workloads, and inter-service dependencies. Kubernetes is the most widely used platform for managing large-scale containerized microservice. Rao and Li. [40] presented an energy-aware microservice scheduling approach for Kubernetes using an Improved Sparrow Search Algorithm (ISSA) to prioritize pod placement based on parameters like communication patterns, and CPU. The result presented improved energy reduction over default Kubernetes schedulers. Turin et al. [41] developed a CPU and Memory prediction model, which provides calibrated workload estimation under dynamic execution scenarios. Tomarchio et al. [42] proposed a load-aware Kubernetes orchestrator, which exhibits in reduction in end-to-end latency under dynamic workload conditions across shared clusters.

Several reinforcement learning-based schedulers have advanced the capabilities of the Kubernetes. Bai et al. [43] presented REACH, a Soft Actor Critics based container scheduler that reduces microservice latency in edge computing systems. Cai et al. [44] developed a multi-dimensional RL-based Kubernetes scheduler which is capable of adjusting weights automatically according to the need of Input/Output, CPU and GPU. The result shows faster response times over the default container scheduler.

Jian et al. [45] developed a DRL based Kubernetes scheduler developed as a Markov Decision Process, which exhibits the improvements in resource utilization and load imbalance.

Senjab et al. [46], presented a classified Kubernetes scheduling improvements into generic, multi-objective, and ML based techniques. It highlights potential of RL to enhance container scheduling under dynamic workload conditions. Santos et al. [47] presented optimal latency with deep Reinforcement Learning in Kubernetes clusters. Alongside the rapid advancements driven by Kubernetes, numerous studies have examined metaheuristic and evolutionary approaches for optimizing containerized microservices. Numerous research has examined at metaheuristic and evolutionary scheduling methods for optimizing containerized microservices in addition to the rapid advancement fostered by Kubernetes.

Docker Swarm has a simpler architecture and a smaller number of scheduler extension points, research using reinforcement learning for Docker Swarm is very limited, as opposed to Kubernetes [48]. However, Docker Swarm is suitable for small-scale containerized microservice deployments and lightweight Reinforcement Learning-based experimentation due to its steady performance under heavy demand and reduced container orchestration overhead. These studies demonstrate how RL can effectively adapt container scheduling decisions in response to dynamic resource usage and application performance.

Table 4 summarizes the reinforcement learning-based container scheduling approaches studied in the literature. It highlights the approach used, optimization objectives considered, experiment environments, and outcomes achieved. The comparative analysis highlights research potential for creating multiple optimization objective-based adaptive container scheduling frameworks and offers insightful information about the efficiency of reinforcement learning for intelligent container orchestration.

While a lot of research has been done on reinforcement learning-based scheduling in Kubernetes, very little has been done in the field of Docker Swarm. However, Docker Swarm consistently provides useful benefits for controlled testing, such as steady behaviour at high cluster saturation levels and minimal orchestration costs. It makes it ideal for RL-based scheduling and small-scale containerized microservice deployments. When taken as a whole, these studies show tremendous progress, but they also highlight enduring research gaps: (i) limited integration of real-time metrics; (ii) the lack of support for multi-criteria scheduling that

simultaneously takes resource consumption and application-level QoS into consideration; and (iii) the difficulties in implementing RL based models on lightweight container orchestration platforms like Docker Swarm.

Table 4. RL based approaches used for container scheduling

Approach Used	Optimization Objectives	Experiment Environment	Outcome
DEEP REL	CPU, No. of Pods	Kubernetes	Better response time and resource utilization
REL-Q-LEARNING	VM instances used	Cloud Simulation	Effective Auto-scaling of SAAS services environment
DISTRIBUTED REL	CPU, Memory	Kubernetes based testbed	Reduces average response time and failed requests, validating improved scalability as the number of requests increases.
REL	Latency, CPU, Memory	Kubernetes	Optimal scheduling of microservices
MULTI-AGENT REL	Computation power, memory, Network	Private cloud environment	Reduce execution time, Outperforms K8s and Docker Swarm
REL	CPU, Memory, Response time	Kubernetes on Private Cloud	Minimise cost and SLA violation
Q-BASED REL	CPU usage, application instances	Simulation environment-Docker Swarm EDS	Reduced deployment cost
REL	CPU, Memory, Time	Simulation	Reduce the overall cost
REL	CPU, Memory	Kubernetes based Australian National Cloud Infrastructure (Nectar)	Reduce load imbalance and resource utilization
DEEP REL	CPU, Memory, Transmission rate, Read/Write rate of disk	Kubernetes (Physical cluster)	Reduce power consumption of cloud cluster
TOPSIS	CPU, Memory, Disk, Containers, container Images, Power consumption	Kubernetes-Virtual Testbed	Effective Auto-scaling of SAAS services environment

These research gaps motivate for the development of the suggested MCSCM-RL container scheduler, which enables adaptive, resource-efficient, and application-aware container deployment in small-scale cloud clusters by combining real-time multi-criteria analysis (CPU, memory, container count, startup time, response time) with a reinforcement learning based framework.

The present research work focuses on Docker Swarm and proposes the MCSCM-RL scheduler, a multi-criterion, resource-aware and application-aware RL based container scheduler. The scheduler that has been presented is made for small scale but still active Swarm clusters and it tunes five critical performance parameters: CPU usage, memory usage, number of containers, startup time and response time. As a result, MCSCM-RL is light, interpretable, and can be immediately implemented in situations where the complexity of Kubernetes is not necessary. Overall, the MCSCM-RL framework shows how intelligent, multi-criteria RL policies may be added to Docker Swarm to improve application performance with limited resource utilization.

2.8 Container Scheduling Frameworks in Docker Swarm and Kubernetes

Among all container orchestration frameworks, Docker Swarm remains one of the most integrated and simple container orchestration platforms for microservices. Docker swarm has built-in clustering capability to transform a group of Docker engines into a single virtual system for container orchestration and management. 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. It is capable of automatically deploying, scaling, and managing containerized applications and microservices, 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. 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.

Docker swarm default scheduling strategies are so established that they serve as the baselines for evaluating new and custom container scheduling strategies. These are heuristic based approaches that aim to simplify microservice placement decisions.

Binpack is a container deployment strategy widely used in container orchestration platforms to maximize resource utilization by placing containers on the most heavily utilized nodes that still have sufficient available resources. The prime objective of the Binpack algorithm is to consolidate workloads onto a smaller number of nodes, thereby minimizing resource fragmentation and reducing the number of active nodes or servers. In this approach, containers are scheduled onto nodes with the highest resource usage that can still accommodate the incoming workload. By concentrating workloads on fewer nodes, Binpack scheduling can improve overall cluster utilization, reduce energy consumption, and free underutilized nodes for future deployments and power-saving operations. However, excessive consolidation may lead to resource hotspots, increased contention among containers, and potential performance degradation under high workloads. Consequently, while Binpack scheduling is effective for achieving high resource efficiency and energy savings, it must be carefully balanced with Quality of Service (QoS) requirements and workload distribution considerations in modern container-based cloud environments.

Spread is a container deployment strategy in Docker swarm that aims to distribute workloads evenly across all available nodes in a swarm cluster to achieve balanced resource utilization and improve system reliability. Unlike Binpack scheduling, which consolidates containers onto fewer nodes, the Spread scheduler places new containers on nodes with the lowest resource utilization or the fewest running containers. This strategy minimizes resource hotspots, reduces the risk of node overloading, and enhances fault tolerance by preventing excessive concentration of workloads on a single cluster node. Spread scheduling is particularly beneficial for applications or microservices need high availability, load balancing, and consistent performance across the swarm cluster. By evenly distributing containers, the scheduler can improve resource fairness and reduce the impact of node failures. However, this strategy may lead to lower overall resource utilization and increased energy consumption because more nodes remain active. Therefore, Spread scheduling is commonly used in container orchestration platforms when workload distribution, scalability, and service reliability are prioritized over resource consolidation and energy efficiency.

Random is one of the simplest container deployment strategies used in Docker container orchestration environments, where incoming containers are assigned to available nodes randomly without considering resource utilization, workload characteristics, or performance metrics. The primary benefit of this strategy is its low computational overhead and ease of

implementation, making scheduling decisions extremely fast. Since node selection is performed randomly, the scheduler does not require continuous monitoring of cluster resources or complex optimization algorithms. Random scheduling is often used as a baseline strategy for evaluating the effectiveness of more advanced container scheduling techniques. However, because placement decisions are made without awareness of node and cluster situations. Thus, it can lead to uneven workload distribution, resource imbalance, increased response times, and underutilization or overutilization of certain cluster nodes. In large-scale containerized environments, these limitations may negatively impact Quality of Service (QoS), resource efficiency, and overall system performance. Therefore, while Random scheduling provides simplicity and less scheduling overhead, it is generally less effective than scheduling approaches such as Binpack, Spread, and intelligent schedulers in optimizing cluster performance and resource utilization.

To illustrate the operational differences among Random, Spread, and Binpack container scheduling strategies, consider a Docker Swarm cluster consisting of two worker nodes with heterogeneous resource capacities. Node 1 is equipped with 32 CPUs and 80 GB of memory, while Node 2 provides 32 CPUs and 40 GB of memory. Four containers having microservices with varying resource need are submitted to the container queue: Container 1 (16 CPUs, 20 GB memory), Container 2 (8 CPUs, 20 GB memory), Container 3 (8 CPUs, 10 GB memory), and Container 4 (8 CPUs, 20 GB memory). The scheduling strategy determines how these containers are assigned to the available nodes of the docker swarm cluster. In Random scheduling, containers are placed arbitrarily without considering current resource utilization, which may result in workload imbalance and inefficient resource allocation. Spread scheduling distributes containers evenly across the available nodes to achieve balanced resource utilization, improved load distribution, and enhanced fault tolerance. In contrast, Binpack scheduling consolidates containers onto the most utilized node with sufficient available resources, thereby maximizing resource utilization and potentially reducing energy consumption by minimizing the number of active nodes. Figure 6 demonstrates the container deployment decisions and resulting resource utilization patterns for the three container scheduling strategies, highlighting their distinct approaches to workload placement and cluster resource management.

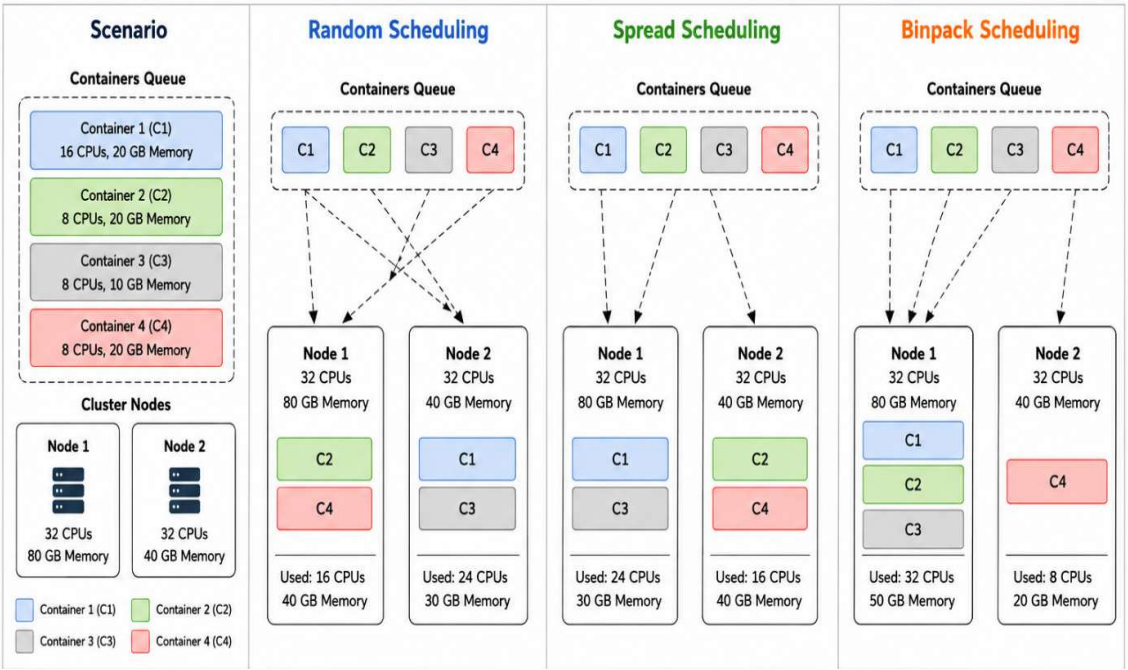


Fig. 6. Docker swam default schedulers

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.

Kubernetes-Scheduling strategy for containerized microservices

Kubernetes is one of the most widely adopted container orchestration platforms for deploying, managing, and scaling containerized microservices in cloud-native environments. It provides automated mechanisms for container deployment, service discovery, load balancing, scaling, fault recovery, and resource management across distributed cluster nodes. In Kubernetes, applications are deployed in the form of pods, where a pod represents the smallest deployable unit consisting of one or more tightly coupled containers. The Kubernetes scheduler is a core component responsible for assigning pods to appropriate worker nodes based on resource availability, scheduling policies, and application requirements. Efficient scheduling is essential for maintaining scalability, Quality of Service (QoS), resource utilization, and application performance in large-scale microservices environments.

Unlike simple scheduling approaches such as Random, Spread, or Binpack, Kubernetes utilizes a multi-stage scheduling process consisting of filtering and scoring phases. In the filtering

phase, the scheduler identifies a list of feasible nodes capable of running a pod by evaluating resource requirements such as CPU, memory, storage, network constraints, and node conditions. Nodes that do not satisfy the pod requirements are eliminated from consideration. In the scoring phase, the feasible nodes are ranked based on predefined scheduling policies and priority functions. The scheduler then selects the node with the highest score for pod deployment. Kubernetes supports several built-in scheduling strategies including least requested resource scheduling, balanced resource allocation, node affinity, anti-affinity scheduling, taints and tolerations, and topology-aware scheduling. These mechanisms help distribute workloads efficiently across cluster nodes while considering application constraints and resource balancing.

For containerized microservices, Kubernetes scheduling plays a critical role in maintaining Quality of Service (QoS), minimizing response time, ensuring high availability, and optimizing cluster resource utilization. However, the default Kubernetes scheduler primarily relies on predefined heuristics and static scoring functions. Despite its advanced container orchestration capabilities, Kubernetes scheduling still faces some challenges in dynamic cloud-based environments. As modern cloud environments become increasingly dynamic and heterogeneous, these conventional scheduling policies may not adequately address multiple conflicting objectives such as CPU utilization, memory consumption, startup time, response time, energy efficiency, and application-specific QoS requirements. Consequently, recent research has focused on integrating machine learning and reinforcement learning techniques into Kubernetes scheduling frameworks to achieve intelligent, adaptive, and multi-objective scheduling decisions for containerized microservices.

The Fig. 7 illustrates the working mechanism of the Kubernetes scheduler for deploying containerized microservices across worker nodes in a cluster environment. In Kubernetes, applications are deployed as pods, and the scheduler is responsible for selecting the most suitable worker node for pod deployment based on resource availability and scheduling constraints. Initially, the scheduler identifies all possible worker nodes capable of hosting the incoming pod. During the filtering phase, nodes that do not satisfy the resource requirements are eliminated from consideration. As shown in the figure, Worker 3 is excluded because it does not have sufficient resources, while Worker 2 is removed due to conflicts of volume. After filtering, only feasible worker nodes remain available for scheduling.

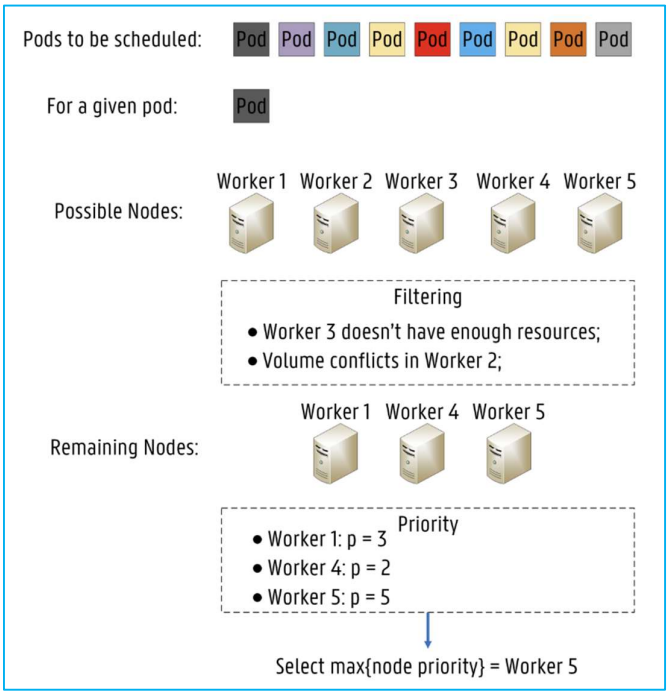


Fig. 7. Container scheduling in Kubernetes

In the next stage, known as the priority or scoring phase, Kubernetes assigns priority scores to the remaining worker nodes based on predefined scheduling policies and resource optimization criteria. The scheduler evaluates factors such as CPU utilization, memory availability, workload distribution, affinity rules, and overall resource balancing to calculate node priorities. In the figure, Worker 1 receives a priority value of 3, Worker 4 receives 2, and Worker 5 receives the highest priority value of 5. Finally, the scheduler selects the node with the maximum priority score, resulting in the deployment of the pod on Worker 5. This two-stage scheduling process helps Kubernetes efficiently allocate resources, improve load balancing, and ensure reliable deployment of containerized microservices in distributed cloud-native environments.

2.9 Research Gap

The rapid adoption of cloud-native applications and microservices architecture has significantly increased the deployment of containerized workloads in cloud environments. Container orchestration platforms such as Docker Swarm and Kubernetes provide automated mechanisms for deployment, scheduling, scaling, and management of container lifecycle. However, efficient container scheduling remains a major research challenge due to the dynamic and heterogeneous workloads of cloud environments. Existing container scheduling strategies of Docker swarm

such as Random, Spread, and Binpack scheduling primarily rely on static heuristic-based approaches and are unable to adapt effectively to continuously changing workload conditions. These baseline container schedulers basically focus on limited resource parameters such as CPU or memory availability while ignoring important application-level and crucial parameters like startup time, response time, workload behaviour, inter-service communication, and energy consumption.

Many research studies have proposed heuristic-based, QoS-aware, and machine learning-based container scheduling approaches for improving container deployment decisions. Although these approaches demonstrate improvements in specific performance metrics, most existing solutions still suffer from many limitations. Many container scheduling strategies optimize only a single optimization objective such as load balancing, resource utilization, or latency, without considering the multiple parameters simultaneously. Furthermore, a majority of existing container schedulers lack dynamic adaptation capabilities and fail to learn from fluctuating runtime environments. Limited research has been conducted on integrating multi-criteria decision-making with reinforcement learning for autonomous container scheduling in Docker Swarm environments. Additionally, energy-aware scheduling and power consumption optimization remain insufficiently explored in existing Docker Swarm-based orchestration frameworks. There is also a lack of comprehensive experimental evaluation involving startup time, response time, and QoS-aware scheduling in reinforcement learning-driven container orchestration systems. Therefore, there exists a significant research gap in developing an intelligent, adaptive, multi-criterion, QoS-aware, and energy-efficient container scheduling framework capable of dynamically optimizing deployment decisions in distributed cloud-native environments.

Consequently, there remains a need for an adaptive and intelligent scheduling framework that can continuously learn from cluster situations, optimize multiple conflicting objectives simultaneously, and improve both resource efficiency and QoS.

Motivation: The increasing complexity of cloud-based applications and large-scale deployment of containerized microservices have created a strong demand for intelligent and adaptive resource management mechanisms in cloud environments. Modern applications such as IoT systems, machine learning services, streaming platforms, and real-time analytics require high scalability, low latency, continuous availability, and efficient resource utilization. Traditional container scheduling strategies are insufficient for handling dynamic workloads

because they do not consider runtime variations, application-level QoS requirements, or energy consumption during deployment decisions. As a result, inefficient scheduling often leads to increased startup delay, higher response time, resource contention, poor load balancing, and excessive power consumption in cloud datacentres.

Another important motivation for this research is the growing operational cost and environmental impact associated with energy consumption in modern cloud infrastructures. Large-scale datacentres consume enormous amounts of electrical power, making energy-efficient workload management an essential research problem. Existing scheduling mechanisms in Docker Swarm environments generally lack energy-awareness and intelligent adaptation capabilities. Recent advancements in machine learning and reinforcement learning techniques have demonstrated significant potential for intelligent decision-making and adaptive optimization in dynamic environments. Reinforcement learning enables scheduling agents to continuously learn optimal deployment policies through interaction with the cluster environment and improve scheduling decisions over time. Motivated by these challenges and opportunities, this research aims to develop a Multi-Criteria Scheduling of Containerized Microservices using Reinforcement Learning (MCSCM-RL) framework that integrates QoS-aware, energy-aware, and resource-aware scheduling mechanisms for improving overall system performance, resource utilization, and energy efficiency in Docker Swarm-based cloud environments.

Summary of Literature Review

The literature review presented in this chapter provides a comprehensive study of cloud computing, virtualization, containerization, microservices architecture, container scheduling, container orchestration platforms and container scheduling techniques for microservices in cloud-based environments. The review highlights the evolution of cloud computing as a scalable and flexible computing paradigm that enables on-demand access to shared computing resources through virtualization. Traditional virtual machine-based deployment approaches were analysed along with their limitations, including high resource overhead, slow startup time, and inefficient scalability, which motivated the adoption of lightweight containerization technologies.

The shift from monolithic to microservices architecture which divides applications into loosely connected, independently deployable services were also covered in this chapter. It was

addressed how containerization technologies in particular, Docker allow for the effective deployment and administration of microservices through portable and lightweight execution environments. Along with the operational difficulties related to orchestration, scalability, fault tolerance, and resource management, the container lifecycle, Docker Engine architecture, and deployment mechanisms of containerized microservices were also examined.

The chapter has also included the scheduling techniques of widely used container orchestration platforms such as Docker Swarm and Kubernetes, focusing on their scheduling mechanisms and resource allocation strategies. Existing container scheduling approaches of Docker swarm including Random, Spread, and Binpack were thoroughly studied. The Kubernetes scheduling workflow involving filtering and priority-based node selection was also discussed to understand modern orchestration strategies for distributed containerized applications.

Furthermore, the literature review investigated recent researches in Quality of Service (QoS)-aware, energy-aware, and machine learning-based scheduling approaches for container orchestration. Several studies demonstrated the effectiveness of intelligent scheduling mechanisms in improving resource utilization and application performance. However, the analysis identified significant research gaps in existing approaches, including limited consideration of multiple runtime parameters, lack of dynamic adaptation to workloads, insufficient energy-aware optimization, and minimal integration of reinforcement learning techniques in Docker Swarm environments.

Based on the identified research gaps, the review establishes the need for an intelligent, adaptive, and multi-criteria container scheduling framework capable of optimizing resource utilization, startup time, response time, load balancing, and energy efficiency simultaneously. The findings from the literature review offers the foundation and motivation for the proposed Multi-Criteria Scheduling of Containerized Microservices using Reinforcement Learning (MCSCM-RL) framework presented in this thesis.

Chapter 3
Research Methodology and Problem Formulation

This chapter presents the research methodology and problem formulation employed for developing an intelligent container scheduling framework for cloud-based containerized microservices environments. The chapter focuses on the design and implementation of the proposed Multi-Criteria Scheduling of Containerized Microservices using Reinforcement Learning (MCSCM-RL) framework in Docker Swarm environments. It describes the system architecture, scheduling workflow, experimental setup, and the methodology used for dynamic monitoring of resources, node ranking, and adaptive deployment of containerized microservices. The chapter includes formulation the container scheduling problem by considering resource-aware and application-aware parameters such as CPU utilization, memory utilization, container count, startup time, and response time. Furthermore, the integration of reinforcement learning techniques, with Deep Q-Network (DQN) for the container scheduling, is discussed for autonomous and intelligent scheduling decision in dynamic cloud environments. The proposed methodology aims to improve Quality of Service (QoS), optimize resource utilization, reduce deployment latency, and enhance energy efficiency in container-based cloud environments.

3.1 Problem Description

The rapid growth of cloud computing and cloud-based applications has significantly increased the deployment of containerized microservices in cloud computing environments. Highly scalable, adaptable, and highly accessible computing infrastructures are necessary for modern applications including Internet of Things (IoT) systems, real-time analytics, machine learning applications, multimedia streaming services, and large-scale web applications. To fulfil these needs, organizations have increasingly embraced microservices architecture and containerization technologies. The microservice architecture decomposes the application into multiple lightweight and independently deployable services running inside containers. The containerization is a light weight virtualization technique. Container orchestration platforms such as Docker Swarm and Kubernetes are widely used to automate the container deployment, scaling, networking, and management of these containerized microservices across cluster nodes.

Among the various functions of container orchestration systems, container scheduling plays a important role in determining the overall performance and efficiency of cloud environments. Container scheduling refers to the process of selecting the most appropriate node of the cloud cluster for deploying a container based on resource availability and scheduling policies. Efficient container scheduling is required for maintaining optimal resource utilization, load balancing, low response time, high throughput, and Quality of Service (QoS). However, the dynamic and heterogeneous nature of cloud workloads makes container scheduling a highly complex optimization problem.

Existing container scheduling strategies implemented in container orchestration platforms such as Docker Swarm primarily include Spread, Binpack, and Random scheduling approaches. Spread scheduling aims to distribute containers evenly across cluster nodes to achieve load balancing. Binpack scheduling concentrates on maximizing resource utilization by packing containers into fewer cluster nodes. Random scheduling selects nodes randomly without considering resource utilization or characteristics of the microservice Although these scheduling approaches are simple and computationally efficient, they rely mainly on static heuristic-based policies and fail to adapt effectively to dynamic workload situations in cloud environments.

Kubernetes is also widely adopted container orchestration platform by organizations and researchers for container scheduling of microservices. Kubernetes scheduler primarily relies on predefined heuristics and static scoring mechanisms that basically focus on resource availability, and overlook application-level Quality of Service (QoS) metrics such as startup time, response time, and service delay. Furthermore, the Kubernetes scheduler does not support multi-objective optimization involving resource utilization, energy efficiency, workload characteristics, and application performance simultaneously.

One of the major limitations of existing scheduling techniques is their inability to simultaneously consider multiple system-level and application-level parameters during deployment decisions. Most baseline container schedulers focus only on basic resource metrics such as CPU and memory availability, while ignoring critical application performance parameters such as startup time, response time, container count, workload fluctuation, communication overhead among microservices, and energy consumption. As a result, inefficient scheduling decisions may lead to resource contention, increased latency, poor load

balancing, service degradation, underutilization of resources, and higher operational costs in cloud datacentres.

Another significant issue is the growing energy consumption of modern cloud infrastructures. Large-scale datacentres consume significant amounts of electrical energy to support constantly running applications and services. Improper workload placement and inefficient resource management leads to excessive power consumption and environmental damage. Existing container scheduling approaches in Docker Swarm environments generally lack energy-awareness and do not optimize the trade-off between performance and energy efficiency.

In addition, cloud-based environments are highly dynamic, where workload patterns and resource availability are highly dynamic. Static container scheduling mechanisms are unable to adapt effectively to these dynamic changes, resulting in suboptimal deployment decisions. Therefore, there is a need for intelligent and adaptive container scheduling frameworks which is capable of learning from runtime cluster environments and autonomously optimizing deployment policies. Recent advancements in machine learning and reinforcement learning techniques offer promising opportunities for developing adaptive container scheduling mechanisms that can continuously learn and improve their decision-making capabilities through interaction with the environment.

To address these challenges, this research proposes a Multi-Criteria Scheduling of Containerized Microservices using Reinforcement Learning (MCSCM-RL) framework for Docker Swarm environments. The proposed framework integrates multi-criteria decision-making and Deep Reinforcement Learning techniques to dynamically select optimal nodes for container deployment based on application performance and resource optimization metrics. The objective of the proposed container scheduler is to improve resource utilization, reduce startup delay and response time, enhance QoS, achieve efficient load balancing, and minimize energy consumption in cloud environments.

3.2 Research Objectives

The main goal of this research is to design and develop an intelligent, adaptive, QoS-aware, and energy-aware container scheduling framework for Docker Swarm-based cloud environments using reinforcement learning techniques. The proposed framework aims to optimize container

scheduling decisions by considering resource and application metrics simultaneously, and dynamically adapting to fluctuating workload conditions.

The specific objectives of the research are as follows:

1. To Study Existing Container Scheduling Techniques

Analysing current container scheduling techniques and orchestration mechanisms utilized in Docker Swarm and Kubernetes environments is the primary objective. This entails studying the strengths and limitations of heuristic-based and machine learning-based scheduling techniques applied in scheduling of the containerized microservices.

2. To Develop a Multi-Criteria Container Scheduling Framework

The research aims to develop a multi-criteria container scheduling framework that considers both resource-aware and application-aware parameters during container deployment decisions. Parameters such as CPU utilization, memory utilization, container count, startup time, and response time are integrated into the scheduling process for achieving optimized container deployment.

3. To Implement Dynamic Node Ranking Mechanism

Another objective is to develop a dynamic node-ranking mechanism capable of evaluating the suitability of cluster nodes based on real-time resource and performance metrics. The ranking system is intended to continuously update node priorities according to dynamic workload conditions in the cluster environment.

4. To Integrate Reinforcement Learning into Container Scheduling

The research aims to implement a Deep Reinforcement Learning-based scheduling model using Deep Q-Networks (DQN) for intelligent and autonomous decision-making. The reinforcement learning agent learns optimal scheduling decision through continuous interaction with the cluster environment and adapts dynamic workload conditions.

5. To Design an Efficient Reward Function

An important objective is to formulate a reward mechanism that guides the reinforcement learning agent toward optimized container scheduling behaviour. The reward function is designed to minimize startup time, response time, and energy consumption while maximize resource utilization.

6. To Improve Quality of Service (QoS)

The proposed scheduling framework aims to improve Quality of Service (QoS) in containerized microservices environments by reducing start-up time, improving response time, minimizing deployment delay, and maintaining stable system performance under dynamic workload conditions.

7. To Achieve Energy-efficient Resource Management

The objective is to reduce operational cost in cloud datacentres by implementing energy-aware scheduling strategies that optimize balanced container placement and resource allocation.

8. To Perform Comparative Performance Evaluation

The research aims to experimentally evaluate the proposed MCSCM-RL scheduling framework against existing container scheduling strategies including Random, Spread, and Binpack scheduling. Comparative analysis is performed using performance metrics such as CPU utilization, memory utilization, startup time, and response time.

9. Enhancing Container Orchestration through Intelligent Scheduling

Finally, the research aims to contribute toward the development of intelligent and autonomous container orchestration systems capable of adaptive resource management and self-optimized container scheduling for future cloud computing environments.

3.3 Research Methodology

In order to design, develop, and evaluate an intelligent container scheduling framework for Docker Swarm-based cloud environments, this research employs an experimental and analytical methodology. The objective of the research is to create a Multi-Criteria Scheduling of Containerized Microservices using Reinforcement Learning (MCSCM-RL) framework that may enhance energy efficiency, Quality of Service (QoS), and resource utilization in container-based cloud infrastructures. In order to examine the efficiency of intelligent scheduling mechanisms under dynamic workload conditions, the research methodology integrates container orchestration-based experimentation, multi-criteria optimization, reinforcement learning methodologies, and comparative performance analysis.

Experimental and Analytical Research Methodology

The proposed research employs a realistic experimental methodology supported by analytical assessment methods. Initially, an extensive literature review is carried out to analyse existing container scheduling strategies, container orchestration platforms, and reinforcement learning-based optimization approaches. Based on the identified research gaps, a multi-criteria intelligent scheduling framework is formulated for the deployment of containerized microservices in Docker Swarm environments. The scheduling problem, optimization goals, reward functions, and node-ranking mechanisms, various resource and application-aware characteristics are taken in consideration and all mathematically formulated as part of the research's analytical aspect.

The experimental methodology includes implementation of the proposed scheduling framework in a real Docker swarm-based cluster environment. Multiple container scheduling strategies including Random, Spread, Binpack, MCSCM, and the proposed MCSCM-RL approach are implemented and experimentally tested under clean and cumulative workload conditions. Performance metrics such as startup time, response time, CPU utilization, memory utilization, and container count are constantly monitored and analysed. The effectiveness of the proposed scheduling framework is further confirmed by statistical and comparative analysis of the experimental outcomes.

Docker Swarm-based Experimental Setup

The research is conducted using a Docker Swarm-based container cloud experimental environment deployed on virtualized infrastructure. The experimental setup consists of three Ubuntu virtual machines configured as one manager node and two worker nodes forming a Docker Swarm cluster. Docker Engine and Docker Swarm services are installed and configured on all nodes to support distributed container deployment and orchestration of containerized microservices.

Containerized microservices, including Fibonacci microservices and other test workloads, are deployed across the swarm cluster for evaluating container scheduling behaviour under different deployment conditions. Resource monitoring scripts and performance logging tools are integrated into the experimental framework to collect real-time system metrics such as CPU usage, memory utilization, startup delay, response time, and node-level container count. The real Docker swarm-based cloud setup enables realistic evaluation of dynamic workload

distribution, resource management, and container scheduling efficiency in containerized environments.

Reinforcement Learning-based Optimization Approach

The proposed research uses reinforcement learning as the optimization mechanism for intelligent and adaptive scheduling of containerized microservices. Reinforcement learning enables the scheduling agent to interact constantly with the Docker Swarm environment, observe system states, perform scheduling actions, and learn optimal container deployment policies through the reward feedback. In this research, the container scheduling problem is modelled as a sequential decision-making process where the reinforcement learning agent dynamically selects the most appropriate node for deploying incoming containers.

A Deep Q-Network based reinforcement learning model is implemented to handle the complexity of dynamic resource allocation in container-based cloud environments. The state space includes multiple runtime metrics such as CPU utilization, memory utilization, container count, startup time, response time, and node-ranking scores. The action space consists of selecting one of the available swarm worker nodes for deployment of the container. A customized reward function is developed to minimize response time, startup delay, and energy consumption while maximizing resource utilization. Through continuous training and interaction with the environment, the RL agent learns intelligent and adaptive container scheduling strategies, which is capable of improving overall performance of the swarm cluster.

Comparative Performance Evaluation Methodology

Comparative performance evaluation is carried out with current container scheduling strategies, such as Spread scheduling, Binpack scheduling, Random scheduling, and the multi-criteria MCSCM approach in order to verify the efficiency of the proposed MCSCM-RL framework. To assess the robustness and flexibility of the suggested container scheduler, clean and cumulative deployment scenarios are carried out.

The performance evaluation is conducted using system-level and application-level metrics including CPU utilization, memory utilization, startup time, response time, and number of containers deployed on cluster node. Experimental observations are collected over multiple deployment iterations to ensure consistency and reliability of results. Graphical analysis, statistical comparisons, and improvement calculations are used to analyse performance of the

scheduler and demonstrate the advantages of the proposed reinforcement container scheduling framework. The comparative analysis helps assess the capability of the proposed system in achieving efficient resource allocation, QoS optimization, and energy-aware container orchestration in cloud-native environments. The comparison study helps in evaluating the performance of the proposed scheduler to accomplish energy-efficient container orchestration, QoS optimization, and effective resource allocation in container-based cloud environment.

3.4 Research Workflow

The proposed Multi-Criteria Scheduling of Containerized Microservices using Reinforcement Learning (MCSCM-RL) framework is designed to achieve intelligent, adaptive, and energy-efficient container scheduling in a Docker Swarm environment. The overall research workflow is organized into four hierarchical layers, namely the Application Layer, Scheduling Layer, Container Orchestration Layer, and Infrastructure Layer. Each layer performs a specific function within the scheduling process and collectively contributes to the efficient deployment and management of containerized microservices. Fig. 8 illustrates the architecture and workflow of the proposed framework.

Application Layer: The Application Layer represents the topmost layer of the proposed framework and serves as the source of workload generation. In this layer, microservice are created following the microservices architecture paradigm, where a CPU intensive microservice is developed. Each microservice performs a specific functionality and can be developed, deployed, and scaled independently. Each microservice is packaged into a Docker container image. Containerization encapsulates the application code along with all required dependencies, libraries, and runtime environments. These containerized microservices constitute the deployment requests that are submitted to the scheduling layer. As new service requests arrive, the scheduler must determine the most appropriate cluster node for deployment based on resource availability and application performance requirements.

Scheduling Layer: The Scheduling Layer is the core component of the proposed MCSCM-RL scheduling framework and represents the primary contribution of this research work. This layer is responsible for collecting system information, monitoring application performance, ranking cluster nodes, making intelligent scheduling decisions, and selecting the most suitable node for

container deployment. To accomplish these objectives, the layer consists of five interconnected components.

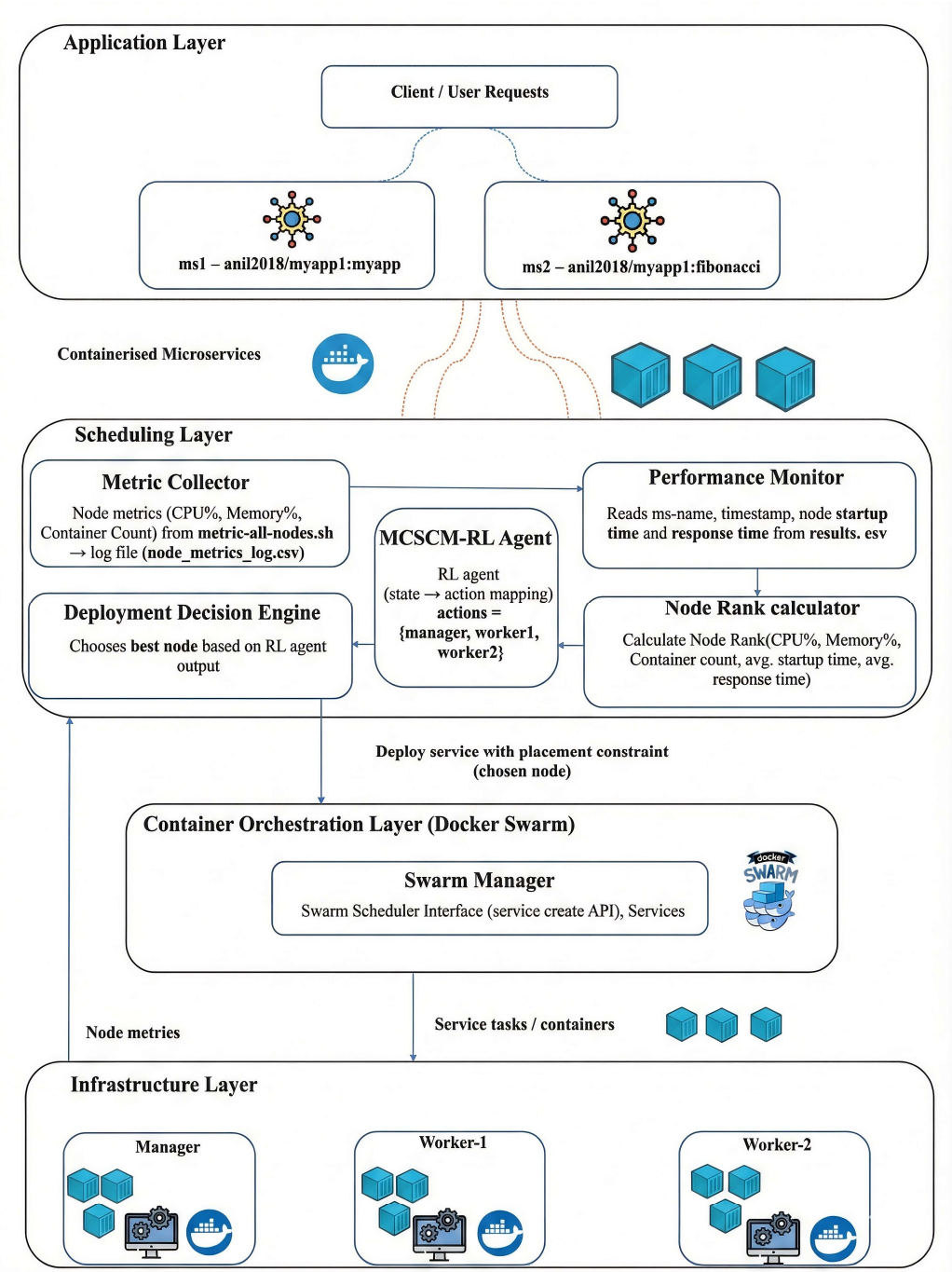


Fig. 8. Research Workflow

Metric Collector: It continuously gathers real-time resource utilization information from all cluster nodes within the Docker Swarm cluster. The collected node metrics include CPU

utilization percentage, memory utilization percentage, and the number of running containers on each swarm node. The collected information is periodically logged into monitoring files and serves as the foundation for subsequent scheduling decisions. By continuously monitoring node status, the system can identify overloaded or underutilized nodes and make informed container scheduling decisions accordingly.

Performance Monitor is responsible for collecting application-level performance metrics associated with deployed containerized microservices. After a container is deployed, this component records important Quality of Service (QoS) parameters, including startup time and response time. Startup time measures the duration required for a deployed containerized microservice to become operational and ready to serve requests. Response time measures the latency experienced when processing microservice requests. These performance indicators are logged for every deployment and are subsequently used to evaluate the effectiveness of container scheduling decisions.

Unlike traditional container schedulers that rely only on infrastructure-level metrics, the inclusion of startup time and response time allows the proposed framework to incorporate application-level performance into the scheduling process.

Node Ranking Calculator performs multi-criteria evaluation of all swarm cluster nodes based on both resource utilization and application performance metrics. This component collects the CPU utilization percentage**,** Memory utilization percentage**,** Container count**,** Average startup time, and Average response time. Using these metrics, the calculator computes a ranking score for each swarm node of the cluster. The ranking process identifies nodes that provide an optimal balance between resource availability and application performance. Nodes with lower resource contention and better QoS characteristics receive higher rankings. The calculated node rankings will be used as a representation of cluster status and form the state information used by the reinforcement learning agent.

MCSCM-RL Agent constitutes the intelligent decision-making component of the proposed framework. This agent employs a Deep Reinforcement Learning (DRL) approach to learn optimal container scheduling strategy through continuous interaction with the swarm cluster environment. The state of the reinforcement learning model is represented using the node rankings generated by the Node Ranking Calculator. Based on the observed state, the agent

selects an action corresponding to the cluster node on which the incoming container should be deployed.

After deployment, the agent receives feedback in the form of rewards based on scheduling outcomes such as resource utilization efficiency, startup time, response time, and overall QoS performance. Through repeated interactions with the environment, the agent continuously updates its policy and learns to make increasingly effective scheduling decisions.

Unlike traditional container scheduling mechanisms such as Spread, Binpack, and Random, the MCSCM-RL agent adapts dynamically to changing workload conditions and continuously improves its decision-making capability over time.

Deployment Decision Engine acts as the final decision layer within the scheduling framework. It receives the selected node from the MCSCM-RL agent and validates the deployment decision before execution. This component translates the scheduling action generated by the reinforcement learning model into an actual deployment command. It ensures that the chosen node satisfies deployment constraints and forwards the deployment request to the Docker Swarm manager. The Deployment Decision Engine serves as the interface between the intelligent scheduling mechanism and the underlying container orchestration platform.

Deployment Layer: The Deployment Layer is responsible for executing the scheduling decisions generated by the Scheduling Intelligence Layer. Once the Deployment Decision Engine selects the target node, the deployment request is forwarded to the Docker Swarm manager. The Docker Swarm manager coordinates the deployment process across the cluster and creates the corresponding service instance on the selected worker node. The deployment process includes:

- Service creation
- Container initialization
- Resource allocation
- Network configuration
- Service activation

The swarm manager ensures that containers are launched according to the scheduling decision generated by the MCSCM-RL framework. This layer therefore acts as the execution environment for container placement and microservice deployment.

Infrastructure Layer: The Infrastructure Layer forms the foundation of the proposed framework and represents the physical and virtual computing resources on which the entire system operates. This layer consists of a Docker Swarm cluster comprising one manager node and two worker nodes.

The cluster is implemented within a virtualized environment, where each node is deployed as a separate virtual machine. The virtualized architecture while enabling realistic experimentation and performance evaluation. Each worker node contributes computational resources such as CPU, memory, storage, and network bandwidth to the swarm cluster. The Docker Swarm manager coordinates resource management and scheduling activities across these worker nodes. The Infrastructure Layer provides the execution platform for running containerized microservices and supplies the resource metrics utilized by the proposed scheduling framework.

The framework integrates application deployment, resource monitoring, multi-criteria node evaluation, reinforcement learning-based decision-making, and container orchestration into a unified scheduling process. Initially, containerized microservice requests are generated from the application layer and submitted for deployment. The scheduling intelligence layer continuously collects resource utilization and application performance metrics from the cluster, evaluates the suitability of available nodes, and determines the optimal deployment target using the MCSCM-RL agent. Based on the selected action, the deployment decision engine instructs the Docker Swarm manager to deploy the container on the chosen worker node. Following deployment, performance metrics such as startup time and response time are recorded and fed back into the scheduling framework to support future decision-making. The complete sequence of operations performed by the proposed framework is described in the following steps.

Step 1: Docker Swarm Cluster Setup

The research workflow begins with the configuration of a Docker Swarm-based cloud environment. Three Ubuntu virtual machines are created and configured as one manager node and two worker nodes. Docker Engine is installed on all virtual machines, and the Docker Swarm cluster is initialized using the swarm manager node. The worker nodes join the cluster

using swarm tokens provided by the manager node. This setup forms a container orchestration environment, which is capable of deploying and managing containerized microservices across swarm cluster.

Step 2: Deployment of Microservices

After cluster initialization, containerized microservices are prepared for deployment. Docker images containing the application code and dependencies are created and stored in Docker Hub or local repositories. Microservices such as Fibonacci-based applications are deployed on the Docker Swarm cluster using service deployment commands. Multiple deployment requests are generated dynamically to simulate realistic cloud-based workload conditions. These deployments form the workload environment used for evaluating container scheduling strategies.

Step 3: Metric Collection

During runtime, the system continuously monitors resource utilization and application performance metrics from all swarm cluster nodes. Monitoring scripts and Docker APIs are used to collect metrics such as CPU utilization, memory utilization, container count, startup time, and response time. The collected metrics represent the current state of the cluster environment and are used as inputs for node ranking and reinforcement learning-based scheduling decisions.

Step 4: Node Ranking

Once the metrics are collected, a multi-criteria node-ranking mechanism is applied to evaluate the suitability of each swarm node for deploying incoming containers. The ranking process considers multiple resource-aware and application-aware parameters simultaneously, including CPU utilization, memory utilization, container count, startup time, and response time. A relative closeness used to calculate ranking scores for each swarm node. The ranked node information helps identify nodes capable of providing better QoS and resource efficiency.

Step 5: RL-based Node Selection

The ranked node information and collected runtime metrics are passed to the Reinforcement Learning (RL) agent for intelligent scheduling decisions. A Deep Q-Network based reinforcement learning model is used as the scheduling agent. The RL agent observes the current cluster state, including node metrics and node ranking, and selects the most suitable

cluster node for deploying the container. The node selection process balances exploration and exploitation to continuously improve scheduling policies during training.

Step 6: Container Deployment

Based on the container scheduling decision generated by the RL agent, the selected cluster node receives the container deployment request. Docker Swarm manager deploys the microservice container on the chosen node using service deployment mechanisms. The deployed container enters the execution phase, where the application becomes operational. Multiple containers may be deployed dynamically across the cluster during experimental execution.

Step 7: QoS Measurement

After the container deployment, the performance of the deployed microservice is continuously monitored to evaluate Quality of Service (QoS). Important QoS parameters such as startup time, response time are measured. These metrics help determine the effectiveness of the scheduling decision and provide runtime feedback regarding the overall system performance and application behaviour.

Step 8: Reward Calculation

Based on the measured QoS and resource utilization metrics, a reward value is calculated for the reinforcement learning agent. The reward function is developed for optimal scheduling behaviour by minimizing startup delay, response time, while maximizing resource utilization. Positive rewards are assigned for efficient scheduling decisions, whereas penalties are applied for poor performance or resource imbalance.

Step 9: RL Model Update

The calculated reward and updated cluster environment state are used to train and update the Deep Q-Network model. The reinforcement learning agent learns from previous scheduling experiences and gradually improves its deployment policy through continuous interaction with the cluster environment. Experience replays and Q-value updates are performed iteratively to improve learning stability.

Step 10: Performance Evaluation

The overall performance of the proposed MCSCM-RL scheduling framework is evaluated through extensive experimentation and comparative analysis. The proposed approach is

compared with existing container scheduling strategies such as Spread, Binpack, Random and custom scheduler MCSCM using performance metrics including startup time, response time, CPU utilization, memory utilization, and number of containers deployed on cluster node. Statistical analysis, graphs, and comparative results are used to validate the effectiveness of the proposed scheduling framework in container-based cloud environments.

3.5 Problem Formulation and System Objectives

Container scheduling in cloud-based environments is a complex multi-objective optimization problem. It involves efficient allocation of containerized microservices across distributed cluster nodes while satisfying resource constraints and Quality of Service (QoS) needs. In modern Docker Swarm based container environments, multiple microservice containers are deployed on worker nodes with changing resource capacities and workload conditions. The container scheduler must constantly decide the most suitable node for deploying containers to achieve balanced resource utilization, low response time, minimal startup time, and energy-efficient operation. Due to dynamic workloads and heterogeneous resource needs, traditional container scheduling strategies are often unable to provide optimal deployment decisions under real-time cloud conditions.

The proposed research formulates the container scheduling problem as an adaptive and intelligent multi-criteria optimization problem where scheduling decisions are based on system-level and application-level parameters. The objective is to develop an adaptive reinforcement learning-based container scheduling framework, which is capable of dynamically choosing optimal worker nodes for deploying containerized microservices. Simultaneously it is aimed to minimize performance overhead and maximize resource efficiency.

The rapid adoption of microservice-based architectures in containerized environments such as Docker and Kubernetes have created new challenges in efficient container scheduling across in container-based cloud environments. Hence, an intelligent, adaptive, and multi-criteria scheduling method is required that can adopt dynamic changes in the system state and take the best container placement decisions by continuously learning the environment. This section outlines the system architecture of the proposed intelligent container scheduler, followed by the application model used to characterize microservice behaviour and QoS requirements. It

concludes with the evaluation metrics that measure resource efficiency, and performance 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 metrics (average startup time, average response time) within a Deep Q-Network based learning framework. This custom container scheduler dynamically learns optimal cluster node-selection policies using continuous feedback and reward-based adaptation, including relative closeness for state representation. By getting continuous environmental feedback, the scheduler is able to dynamically select the best node for container deployment. Using proposed scheduler three things can be achieved: better resource utilization, minimum startup time, enhanced system efficiency.

Optimization Objectives

Performance metrics that measure both resource efficiency and Quality of Service (QoS) of application must be used to evaluate the effectiveness of MCSCM-RL. In order to validate whether the MCSCM-RL algorithm can outperform baseline Docker Swarm container scheduling strategies like Spread, BinPack, Random, the metrics CPU utilization, memory utilization, container count, average startup time, and average response time are selected. Section 3.7 contains the explicit mathematical formulas for these metrics as well as MCSCM-RL's Learning Dynamics.

CPU Utilization and Memory Utilization

In a Docker Swarm cluster, CPU and memory utilization are essential metrics for evaluating the efficiency of load balancing and resource optimization. CPU utilization is a measure of the amount of processing a node goes through and can be used to indirectly determine if a node is overloaded or underutilized. However, memory usage is a reflection of both container density and application memory usage, both of which affect microservice responsiveness and latency. The drawbacks of Docker Swarm's native container scheduling techniques, such the Spread and Binpack policy, which mostly depend on container numbers and do not take real-time CPU or memory availability into account when taking container deployment decisions, are immediately addressed by these measurements.

Container Count

It displays the distribution of workloads among swarm cluster nodes and identifies those where the heavy few may cause the cluster to be overused or underutilized. While the Binpack scheduler concentrates deployments on the least-loaded node, which frequently increases the variance in container distribution. The Spread scheduler in Docker Swarm aims to equalize the number of operating containers per cluster node. But focusing only on balancing the number of containers per node will not guarantee that the resource utilization of the cluster is balanced. Especially in the case of clusters where nodes have different CPUs and memory capacities. Measuring container counting the number of containers deployed on the node, alongside CPU and memory utilization enables the identification of the clean imbalance. This distinction is very 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 behaviour and limitations of the default container scheduling policies are measured by the container count, and allows for an extensive comparison of MCSCM-RL's resource-aware enhancements.

Average Response Time

Average response time is the simplest indicator of end-user Quality of Service. It is a key metric in the evaluation of performance of microservices. In cloud environments, containerized microservices are usually subject to strict latency standards; a faster response time basically means a better user experience. Inadequate container placement may lead to resource contention, which may result in latency of the service. Thus, reducing response time is one of the primary objectives of MCSCM-RL scheduler. By making intelligent, resource-aware container placement decisions, the MCSCM-RL scheduler excels at reducing resource contention. The scheduler reduces processing delays and improves system responsiveness by putting containers on cluster nodes with good resource availability. An effective container scheduling strategy is expected to provide a low average response time, especially for CPU-bound microservices.

Average Startup Time

Average startup time measures scheduling overhead and deployment efficiency. It shows how quickly new microservices can be started under dynamic resource conditions. Fast microservice startup times can allow rapid scale up and deployment efficiency. These are a basic characteristic of containerized microservice architectures. Apart from being a major parameter

in the productivity of the underlying resource selection strategy, this metric also plays an important role in determining the container scheduler's responsiveness. In the MCSCM-RL framework, startup time plays major role alongside runtime performance metrics like response time. In highly dynamic container-based cloud environments, an effective scheduling policy should improve runtime QoS while maintaining startup overhead at acceptable levels to maintain overall responsiveness of the system.

3.6 System Architecture

The system was proposed to run on a Docker Swarm cluster with three nodes: one Manager node (Manager) and two Worker nodes (Worker-1 and Worker-2). Each swarm node operates as an independent virtual machine (VM) running on an Ubuntu system hosted on Windows System. With Docker Swarm configured for the orchestration of the containerized microservices across the swarm cluster. The Manager Node manages the cluster, keeps track of the states, performs monitoring of the resources, and runs the scheduling strategies such as Spread, Binpack, Random as well as the custom 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 container scheduler executed by the swarm manager.

At runtime, the MCSCM-RL scheduler makes scheduling decisions on the best swarm node for the deployment of containerized microservices with the help of the Deep Q-Network (DQN) based reinforcement learning technique. It selects the node in a multi-criteria manner by considering state parameters like CPU usage, memory usage, number of containers, average startup time, and average response time.

Docker Swarm Cluster Setup

The experimental environment for the proposed research is implemented using a Docker Swarm cluster deployed on multiple Ubuntu virtual machines. Docker Swarm is a native container orchestration platform provided by Docker that enables management of distributed containerized microservices across multiple nodes of the cluster. The Swarm cluster is may be the combination of physical and virtual machines. The Docker Swarm cluster allows automated deployment, scaling, load balancing, and scheduling of containerized microservices. In the proposed experimental environment, the Docker Swarm cluster consists of one manager node and two worker nodes configured within a distributed environment. This experiment setup

provides a realistic cloud-based infrastructure for evaluating the proposed container scheduling framework.

Manager Node

The Manager Node acts as the central control unit of the Docker Swarm cluster. It is responsible for managing the overall cluster operations, maintaining cluster state information, scheduling services, and coordinating communication among worker nodes. The manager node initializes the Docker Swarm environment using the docker swarm *init* command and generates authentication tokens that allow swarm worker nodes to join the cluster securely.

The primary responsibilities of the manager node include:

- Maintaining cluster membership information
- Monitoring the status of all worker nodes
- Managing container orchestration operations
- Scheduling and deploying microservices
- Handling resource allocation decisions
- Performing load balancing and fault management
- Managing service replication and scaling

In the proposed scheduling framework, the manager node also hosts the intelligent scheduling module responsible for collecting runtime metrics, node ranking, and executing reinforcement learning-based scheduling decisions. The manager continuously collects the information of metrics such as CPU usage, memory usage, container count, startup time, and response time, from all worker nodes. Based on these metrics, the proposed MCSCM-RL scheduler dynamically selects the most suitable node for deploying microservice containers.

The manager node maintains the desired state of the cluster and ensures that deployed microservices remain operational even in the event of node failures. If a worker node becomes unavailable, the manager automatically redistributes workloads to healthy nodes to maintain service availability and fault tolerance.

Worker Nodes

Worker Nodes are the execution units of the Docker Swarm cluster where containerized microservices are deployed and executed. After initialization of the manager node, worker nodes join the swarm cluster using the authentication token generated by the manager. Each

worker node runs Docker Engine and participates in distributed container execution under the coordination of the swarm manager node.

- The primary functions of swarm worker nodes include:
- Executing containerized microservices
- Providing CPU, memory, and storage resources
- Reporting runtime resource utilization to the manager
- Supporting distributed workload execution
- Participating in service scaling and load balancing

In the proposed experimental environment, two worker nodes are configured using Ubuntu virtual machines. Each worker node contains a different runtime state depending on current workload situations and deployed containers. During execution, worker nodes continuously host microservice containers and process incoming microservice requests. Performance metrics including CPU utilization, memory utilization, container count, startup time, and response time are periodically monitored from each swarm worker node.

The proposed MCSCM-RL scheduler dynamically selects manger and worker nodes for deploying incoming microservice containers based on current availability of the resources and QoS parameters. The worker nodes execute the assigned microservice containers and provide feedback regarding runtime performance and resource consumption. This execution environment enables evaluation of startup time, response time, and energy-aware scheduling behaviour under dynamic workload situations.

The combination of manager and worker nodes in the Docker Swarm cluster provides a realistic cloud-based container orchestration platform suitable for implementing and evaluating intelligent reinforcement learning-based container scheduling strategies for microservices deployment.

3.7 Proposed MCSCM-RL Scheduling workflow

The proposed MCSCM-RL algorithm is designed for intelligent scheduling of containerized microservices across heterogeneous cluster nodes in a Docker Swarm environment. Unlike traditional container scheduling strategies such as Spread, Binpack, and Random, which are depend only on predefined heuristics, MCSCM-RL uses a DQN based reinforcement learning

agent to learn optimal scheduling decisions from experience and system states. This combination of multiple performance metrics and deep reinforcement learning (DRL) for optimal resource allocation, minimal response time, fast startup time and schedule the work in an energy-efficient way. The environment of a DRL agent is characterized by the combination of five major parameters: percentage of CPU utilization, percentage of memory utilization, the number of containers, average startup-time, and average response time. State is basically a picture of all the cluster nodes at the swarm cluster. Both resource-level and application-level features of the containerized microservices are reflected in these measures. Before the deployment of each containerized microservices, these metrics are normalized and compute a relative closeness score for each node of the docker swarm cluster. 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 containerized microservice. The startup time and response time of the containerized microservices are then measured to compute the reward, which reflects the efficiency and QoS performance of the container scheduling decision. After container deployment, the agent observes the updated state of the cluster, 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 the time. By constant learning from real-time feedback, the MCSCM-RL scheduler dynamically adapts to changing workloads, balances swarm node resource utilization, and minimizes response times of the microservice deployment. The final trained model encapsulates an intelligent policy capable of making energy-aware and application-aware container scheduling decisions. It outperforms conventional heuristics in terms of both efficiency and QoS.

Metrics Collection

Metric collection is an important component of the proposed MCSCM-RL scheduling framework. An accurate runtime information is required for intelligent scheduling decisions and adaptive resource management. In container-based cloud environments, the performance and resource utilization of worker nodes of the cluster are continuously change due to dynamic workload execution and container deployment activities. Thus, constant monitoring and collection of system-level and application-level metrics are necessary to evaluate the current state of the Docker Swarm cluster. The collected metrics are used for node ranking, reinforcement learning-based scheduling, QoS analysis, and performance evaluation of the proposed scheduling framework. In this research work, performance metrics such as CPU

utilization, memory utilization, container count, startup time, and response time are runtime collected from all swarm worker nodes using monitoring scripts and Docker APIs. These metrics provide real-time information regarding workload conditions of the swarm node and application performance, which enables the proposed scheduler to dynamically allocate containers to the most appropriate worker nodes of the cluster while optimizing resource utilization, load balancing, QoS, and energy efficiency.

Node Metrics Collection

To quantify node-level performance characteristics, the CPU utilization, memory utilization, and number of containers deployed on the node, were computed and used as input parameters for the proposed scheduling framework.

i. The CPU utilization of a docker swarm node shows 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** shows the percentage of memory currently in use relative to the total available memory on the node:

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

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 cluster 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 Metric Collection

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

Average Startup Time

Startup time refers to the duration between the initiation of a container deployment and time the microservice becomes operational and ready to accept user 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 is defined as the time between sending a request to a deployed microservice and receiving its response. 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.

3.8 Multi-Criteria Node Ranking

In the proposed MCSCM-RL scheduling framework, the closeness function is used as a multi-criteria node-ranking mechanism for evaluating the suitability of worker nodes before container deployment. Since the Docker Swarm cluster consists of two worker nodes with dynamically changing resource situations, selecting the optimal swarm node for deploying incoming microservice containers becomes a complex scheduling decision problem. The proposed scheduling framework considers multiple parameters such as CPU utilization, memory utilization, container count, startup time, and response time simultaneously. A relative closeness-based node ranking approach is incorporated to identify the worker node that provides the best balance between resource availability, Quality of Service (QoS), and energy efficiency.

In the proposed closeness function each worker node is ranked according to its similarity to the ideal optimal node. In this research, the ideal node represents the node having lower CPU utilization, lower memory usage, minimum startup time, minimum response time, and balanced container distribution. Conversely, the negative ideal node represents the least desirable resource condition. The closeness function calculates how close each worker node is to the ideal solution and how far it is from the negative solution. The node having the highest relative closeness score is considered the most suitable node for container deployment.

Initially, resource metrics are constantly collected from all swarm worker nodes using monitoring scripts and the application metrics are collected after deployment of microservices. The collected metrics form the decision matrix used for node ranking. Since different metrics have different measurement scales, normalization is first applied to transform all values into a comparable range. Weighted importance values are then assigned to each parameter according to their contribution to scheduling efficiency and QoS optimization. The weighted normalized matrix is subsequently used for calculating the ideal positive solution and ideal negative solution.

The raw matrix X = [xij] is first subjected to min–max normalization to obtain the normalized matrix R=[rij]. These normalized values are then weighted to produce the weighted decision matrix V=[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, defined within the interval [0,1], aggregates the multi-criteria information into a single scalar score used to rank nodes in descending order of suitability. The scheduler selects the cluster node corresponding to the highest Ci, while the RL agent may incorporate the vector of closeness values St into its state representation. Fig.-9 illustrates the computational workflow of the proposed MCSCM-RL scheduling framework.

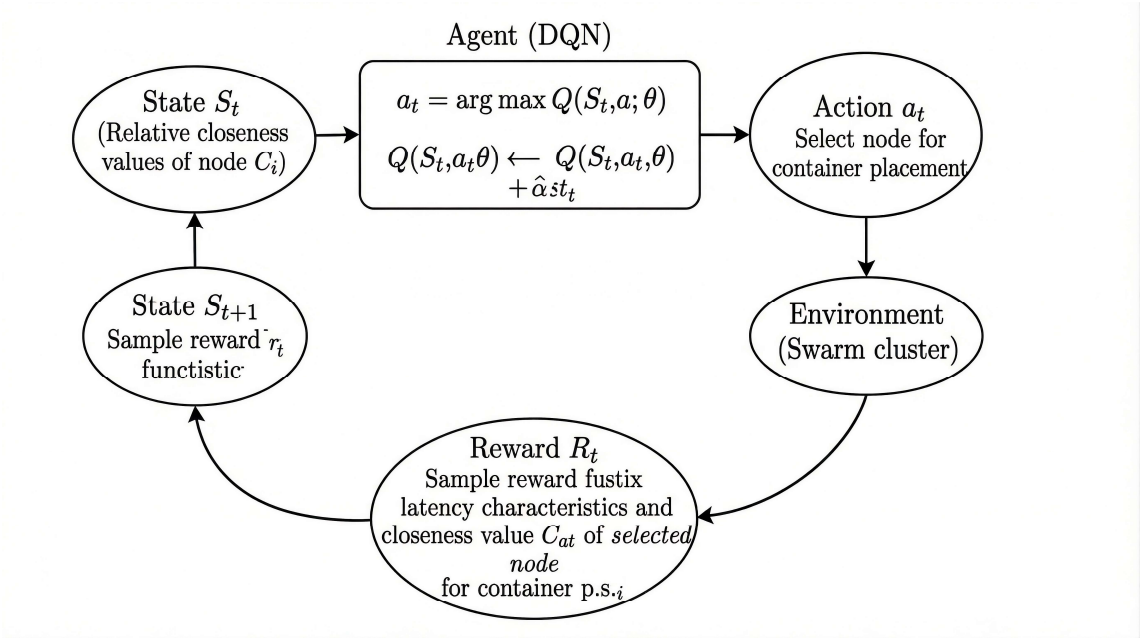


Fig. 9. Computational Workflow

DQN Model Design and Learning Dynamics of MCSCM-RL

The formulation defines the state representation, action space, and reward structure used to guide node selection decisions.

RL State Design Rationale

The state representation of the proposed MCSCM-RL scheduling model is developed to achieve resource availability and QoS of docker swarm cluster nodes. CPU utilization, memory utilization, and container count represent resource usage, while average startup time and average response time gives application-level performance. Combining these performance

metrics within a single state vector permits the DQN agent to develop adaptive and intelligent scheduling mechanism that achieve high system efficiency and application QoS needs. 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 static heuristic container schedulers.

Action Space and Decision-Making

At every stage of the scheduling, the action space of the DQN agent selects one swarm node from the available swarm cluster nodes for the deployment of containerized microservices. The Q-network estimates the expected reward for each candidate node based on the current state, and the node with the highest Q-value will be selected for the deployment, applying an ε-greedy for action selection.

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.

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

Learning and Adaptation to Dynamic Workloads

Adaptation to dynamic workloads is achieved by continuous interaction between the DQN agent and 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. This continuous iteration of assessment enables MCSCM-RL to progressively refine its scheduling policy in response to resource contention, dynamic workloads, and performance fluctuations. In that way MCSCM-RL scheduler offers enhanced robustness compared to static container scheduling approaches.

3.9 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:

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

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

$$S_t = [\, U_{CPU,1},\, U_{Mem,1},\, N_{cont,1},\, T_{startup,avg,1},\, T_{response,avg,1},\, \ldots,\, U_{CPU,m},\, U_{Mem,m},\, N_{cont,m},$$

$$T_{startup,avg,m},\, T_{response,avg,m} \,] \tag{7}$$

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:

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

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:

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

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 container 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:

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

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:

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

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

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

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:

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

9. Container Placement Decision

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

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

3.10 Hyperparameter Details for the DQN Agent

To ensure the reproducibility of the proposed MCSCM-RL scheduler, the Deep Q-Network was configured using the hyperparameters. These parameters are summarized in Table 5. These hyperparameters are used to monitor the learning behaviour of the RL agent, including the neural network optimization process, experience replay mechanism, and exploration strategy. The selected values will be used to achieve stable convergence for efficiently learning optimal scheduling policies in proposed experiment environment.

Table 5: Hyperparameter configuration of the DQN used in MCSCM-RL scheduler

Hyperparameter	Symbol	Value	Description
Learning Rate	α	**0.001**	Controls the step size during neural network weight updates.
Discount Factor	γ	**0.95**	Determines the use of future rewards in Q-value estimation.
Replay Buffer Size	–	**1000**	Maximum number of experiences stored for experience replay.
Mini-batch Size	–	**8**	Number of experiences randomly sampled from the replay buffer during each training iteration.
Initial Exploration Rate	ε	**1.0**	Initial probability of selecting a random action (ε-greedy policy).
Minimum Exploration Rate	εmin	**0.05**	Lower bound of the exploration probability.
Exploration Decay Rate	–	**0.95**	Multiplicative decay factor applied to ε after each replay step.
Optimizer	–	**Adam**	Optimization algorithm used for updating network weights.
Loss Function	–	**Mean Squared Error (MSE)**	Loss function used to minimize the prediction error of Q-values.
Hidden Layer 1	–	**64 neurons**	First fully connected hidden layer with ReLU activation.
Hidden Layer 2	–	**32 neurons**	Second fully connected hidden layer with ReLU activation.
Activation Function	–	**ReLU**	Introduces non-linearity into the neural network.
State Dimension	–	**3**	One relative closeness score for each Docker Swarm node.
Action Dimension	–	**3**	One scheduling action corresponding to each cluster node.
Computing Device	–	**CPU**	Hardware platform used for training and inference.

The DQN agent was implemented using the PyTorch deep learning framework. The Adam optimizer is applied with a learning rate of 0.001 to minimize the Mean Squared Error (MSE) loss function. Experience replay was implemented using a replay memory of 1000 experiences with a mini-batch size of 8. An ε-greedy exploration strategy was adopted, where the exploration probability is initialized to 1.0 and gradually decayed by a factor of 0.95 until reaching a minimum value of 0.05. The neural network comprised two hidden layers containing 64 and 32 neurons with ReLU activation functions. The output layer contained three neurons corresponding to the three available Docker Swarm nodes used for deployment decisions. These hyperparameter settings enabled the proposed MCSCM-RL scheduler to achieve stable learning and efficient container placement.

3.11 Proposed MCSCM-RL Algorithm

Before presenting the pseudocode of the proposed scheduling framework, it is required to describe the logical sequence of operations performed by the Multi-Criteria Scheduling of Containerized Microservices using Reinforcement Learning (MCSCM-RL) algorithm. The proposed algorithm integrates runtime resource monitoring, multi-criteria node ranking, and reinforcement learning-based decision-making to achieve intelligent and adaptive container scheduling in a Docker Swarm environment. The scheduling process begins with the collection of system and application performance metrics from all worker nodes, followed by the computation of node rankings using the relative closeness function. These rankings, together with the collected runtime metrics, form the state representation for the Deep Q-Network (DQN)-based reinforcement learning agent.

The agent then selects the most suitable worker node for deploying incoming microservice containers based on learned scheduling policies. After deployment, Quality of Service (QoS) metrics such as startup time and response time are measured, and a reward value is calculated to evaluate the effectiveness of the scheduling decision. The reinforcement learning model is subsequently updated using the observed rewards, enabling continuous improvement of deployment policies through interaction with the environment.

The pseudocode of the proposed MCSCM-RL scheduling algorithm, which summarizes the flow of the proposed scheduling algorithm, state construction, action selection, reward calculation, and policy update steps.

The complete step-by-step procedure of the proposed MCSCM-RL scheduling algorithm is presented in the following pseudocode.

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

Chapter 4
System Design and Implementation

This chapter presents the experimental platform used to validate the MCSCM-RL scheduling framework described in this thesis in depth. The Docker Swarm cluster made of manager and two worker nodes which are running on three Ubuntu based virtual machines, and hosted on windows based physical machine. The cluster was developed to automatically monitor the application-level and resource-level metrics, such as response time, startup time, memory utilization, CPU utilization and container count.

Two types of deployment scenarios, clean and cumulative, were planned to measure the effectiveness of the existing and proposed scheduler in both situations of clean load and an increasing workload. The proposed MCSCM-RL scheduler was evaluated alongside three existing Docker Swarm scheduling strategies (Spread, Binpack, and Random) and the non-RL multi-criteria custom scheduler (MCSCM). For each scheduling strategy, identical deployment procedures and workload characteristics were employed to ensure methodological integrity and consistency.

4.1 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. The swarm cluster comprising one manager node and two worker nodes, as illustrated in Fig.10. All nodes were connected through a private virtual network to ensure isolated and deterministic communication and formed docker swarm-based container orchestration platform. This configuration provided a controlled, resource-constrained testbed suitable for evaluating container scheduling behaviour. It further enabled consistent collection of CPU, memory, and QoS-related metrics across the swarm cluster throughout all iterations of experiments.

Docker Swarm was selected, as the experimental platform due to the lightweight architecture. 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 container scheduling approaches, where

frequent container deployments and fine-grained metric collection are required with low container orchestration overhead. 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 behaviour and decision-making process of the proposed MCSCM-RL scheduler.

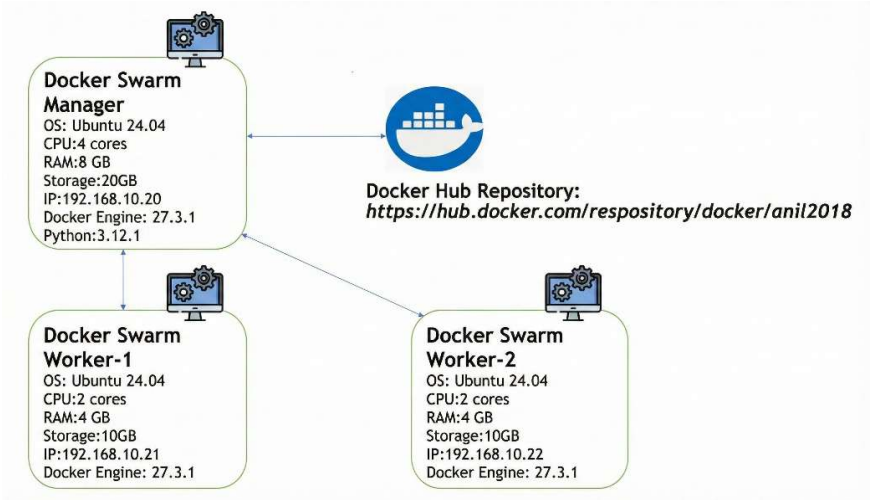


Fig. 10. Docker Swarm three-node cluster used for experiments.

Whereas, the existing literature on Reinforcement Learning-based scheduling in Docker Swarm is very limited and this gap further motivates the present research 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. Future research should focus on how the study's findings can be applied to more sophisticated orchestration platforms, such Kubernetes, using scheduling extenders or custom schedulers. With concentration on Docker Swarm, this study highlights algorithmic clarity and reproducibility, which are needed for isolating the effects of reinforcement learning in container scheduling of containerized microservices.

4.2 Implementation of the Proposed Scheduling Framework

This section presents the practical implementation, deployment procedure of baseline container scheduling strategies (Spread, Binpack, Random) and the proposed Multi-Criteria Scheduling of Containerized Microservices using Reinforcement Learning (MCSCM-RL) framework in a

Docker Swarm environment. The implementation includes the configuration of the Docker Swarm cluster, deployment of containerized microservices, integration of monitoring and metric collection modules, and execution of various scheduling strategies including Spread, Binpack, Random, MCSCM, and the proposed MCSCM-RL scheduling approach. Detailed deployment procedures, configuration steps, execution workflows, and experimental screenshots are provided to demonstrate the practical realization of the proposed scheduling framework. The collected runtime data and execution results serve as the foundation for subsequent performance evaluation and comparative analysis presented in the next chapter.

4.2.1 Creating Docker Swarm Cluster

To create the experimental testbed for the proposed research, three Ubuntu based virtual machines were configured using VMware Workstation Pro on a Windows 11 host system. The virtual machines were allocated appropriate CPU, memory, storage, and network resources to simulate a distributed Docker Swarm cluster environment. One virtual machine was designated as the Manager Node, while the remaining two were configured as Worker Nodes.

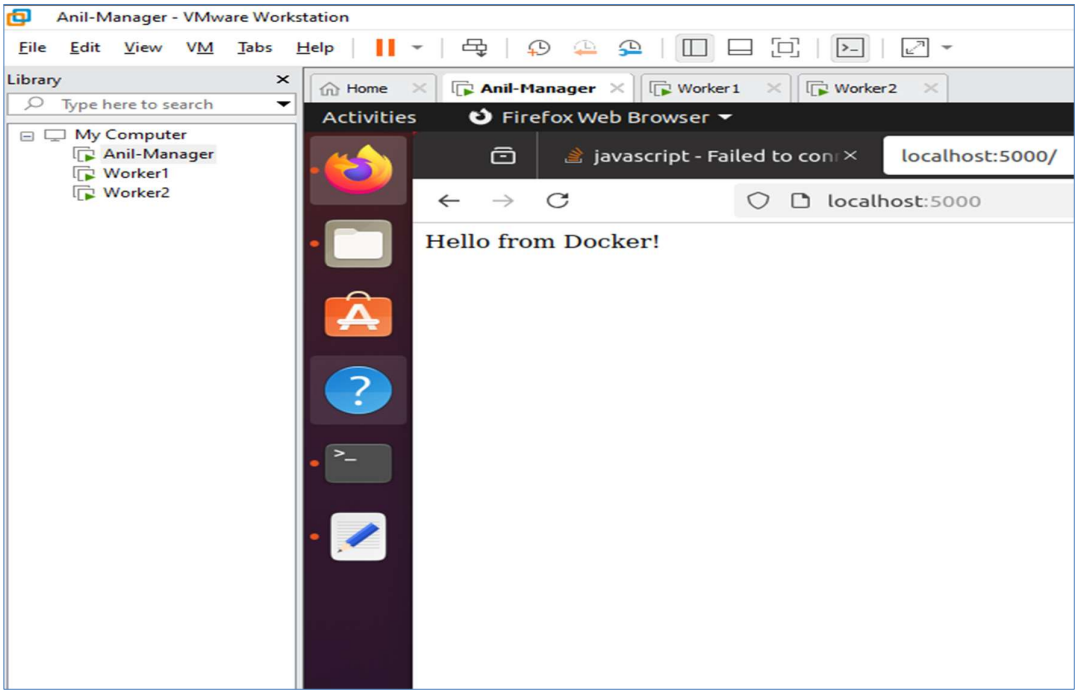


Fig. 11. Creation of Manager and Workers

After creating the virtual machines, Docker Engine was installed and configured on all three Ubuntu nodes to enable container creation and management as shown in Fig.11. Subsequently, Docker Swarm was initialized on the manager node as shown in Fig.12.

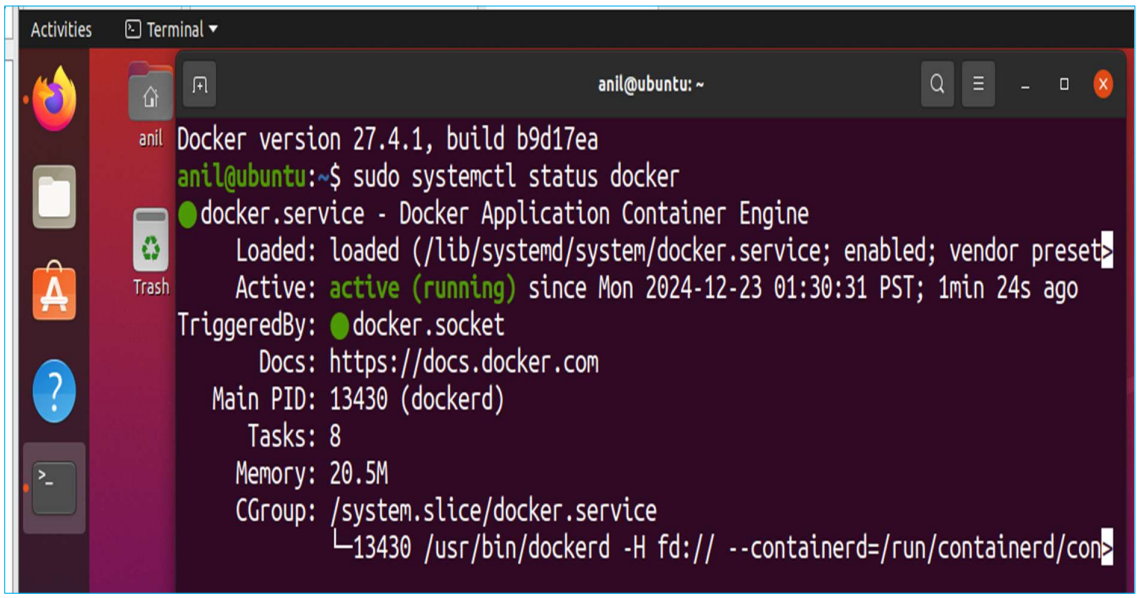


Fig. 12. Installation of docker engine

After the successful installation of Docker Engine and configuration of the Docker Swarm cluster, the operational status of Docker services was verified on all three nodes as shown in Fig.13. Manager node was used to validate the cluster configuration and confirm the active participation of all worker nodes as shown in Fig.14.

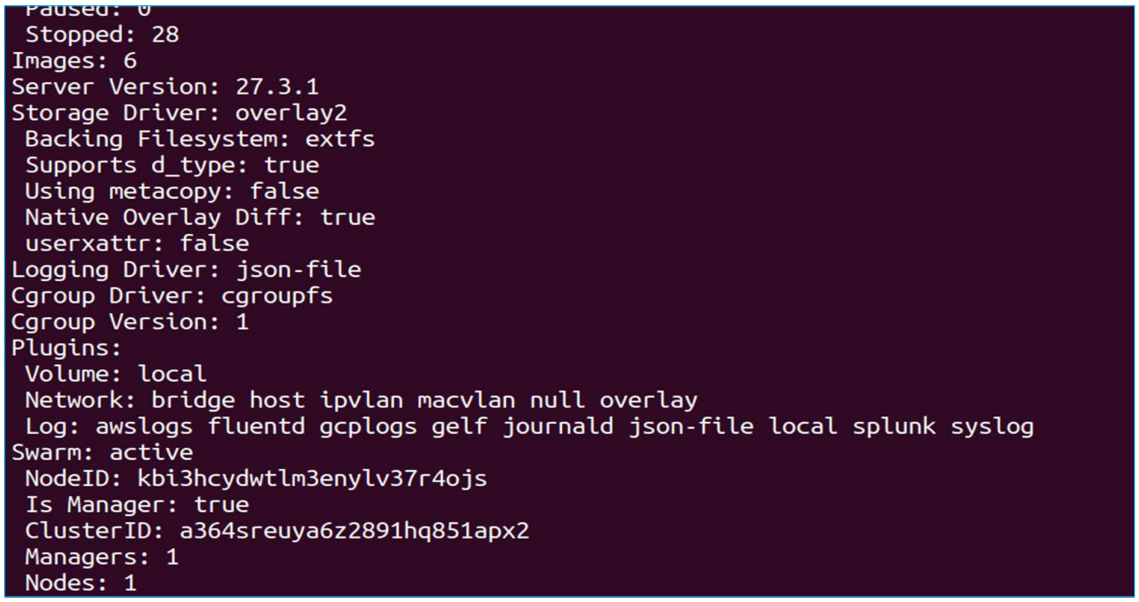


Fig. 13. Checking Docker Swarm on Manager Node

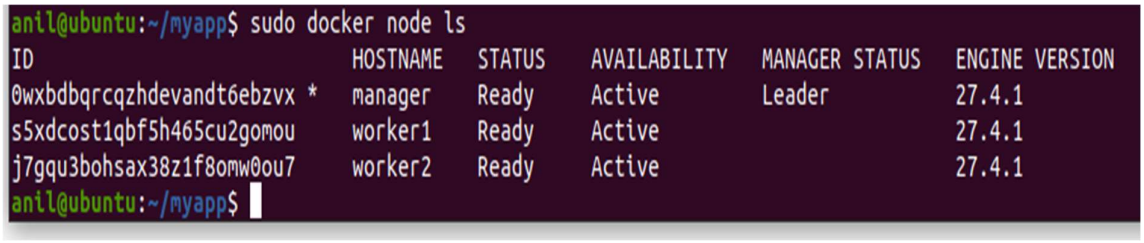
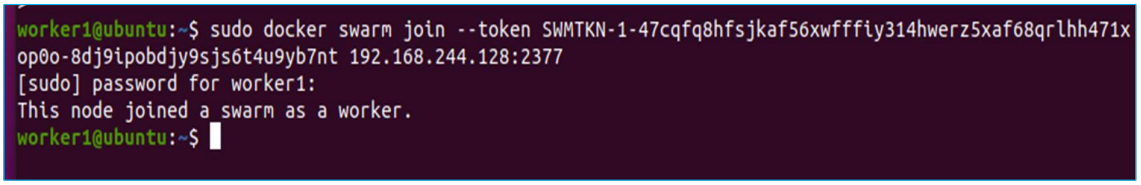


Fig. 14. List of Active Docker Swarm Nodes

The worker nodes were joined to the Swarm cluster using the join token generated by the swarm manager as shown in Fig.15 and Fig.16. This setup established the distributed container orchestration environment required for implementing and evaluating the proposed scheduling framework.

Fig. 15. Token generation by swam manager



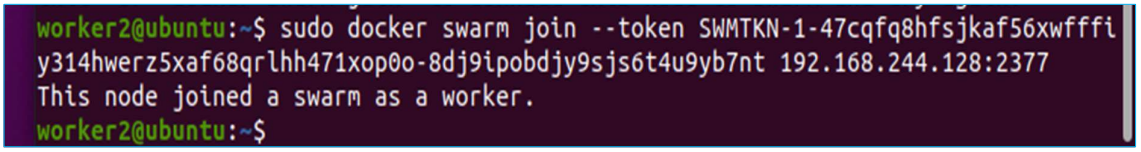


Fig. 16. Workers joined Manager

4.2.2 Microservice Container Development and Deployment

To evaluate the proposed scheduling framework, a containerized microservice was developed and packaged using Docker. A Docker image was created from the application source code and uploaded to Docker Hub, enabling deployment across the Docker Swarm cluster. Building and running docker image of my microservice from Manager Node:

flask-fibonacci-microservice

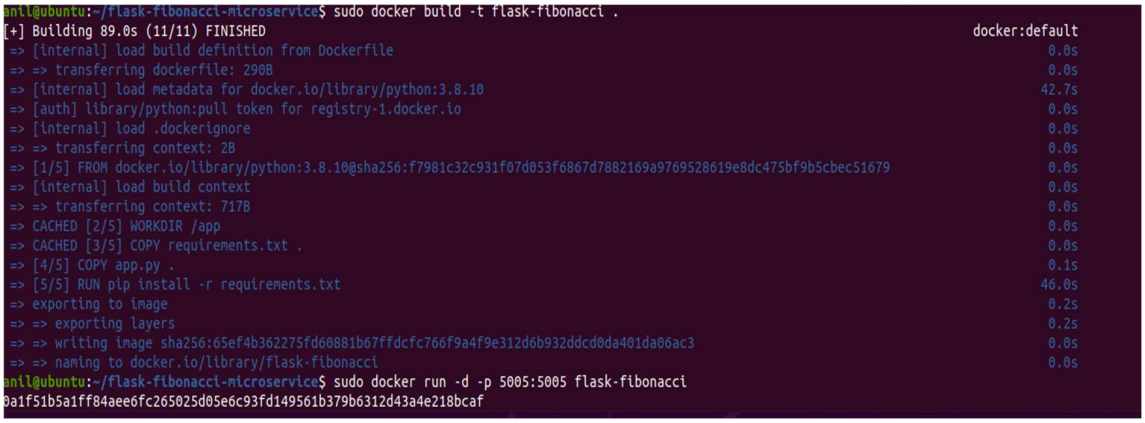


Fig. 17. Building docker image for flask-fibonacci-microservice

Pushing docker image of microservice**:** ***flask-fibonacci-microservice*** to docker hub

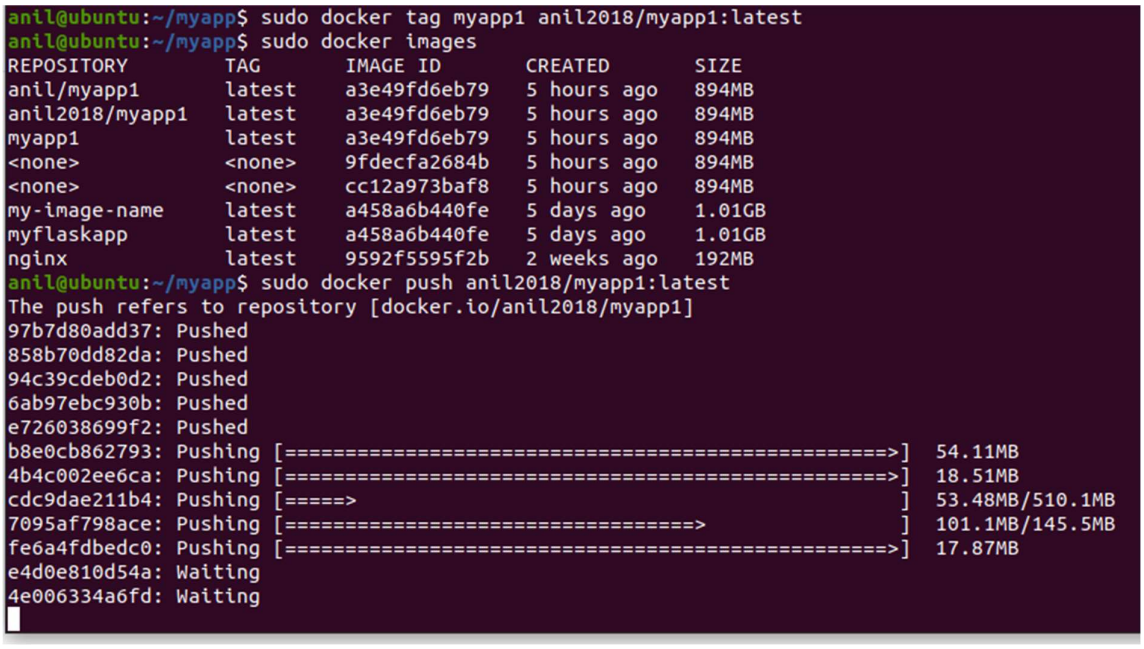


Fig. 18. Pushing flask-fibonacci-microservice to docker hub

To perform the experimental evaluation, the Fibonacci microservice was containerized using Docker and published to the Docker Hub repository **anil2018** as the image **anil2018/myapp1: Fibonacci** as shown in Fig.19. The image was pulled by the Docker Swarm nodes and deployed as a service across the cluster for testing and comparing different container scheduling strategies.

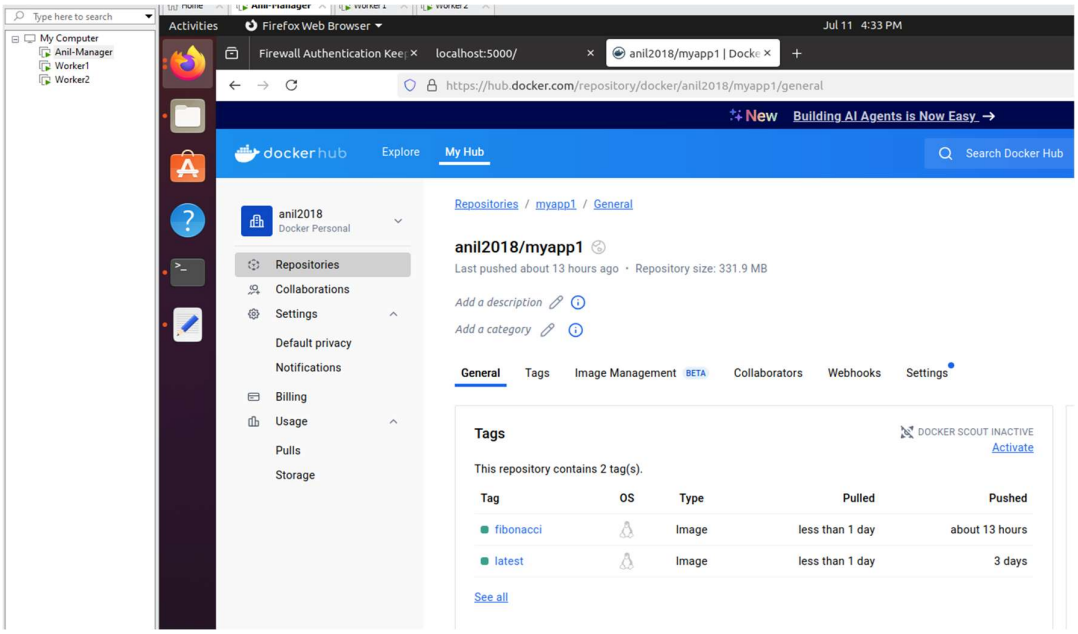


Fig. 19. flask-fibonacci-microservice on docker hub repository

Creating multiple instances of microservice

After publishing the my microservice image to Docker Hub, multiple instances of the Flask-Fibonacci Microservice were created and deployed within the Docker Swarm cluster to simulate realistic workload conditions as shown in Figure-20 and Figure-21. Docker services with multiple replicas were launched, allowing the scheduler to distribute container instances across available worker nodes. These replicated microservices served as the workload for evaluating resource utilization, startup time, response time, load balancing efficiency, and scheduling performance under different deployment strategies.

```
anil@manager:~/flask-fibonacci-microservice$ sudo docker service create  --name fibocopy1 --replicas 2  -p 5014:5005  anil2018/myapp1:fibonacci
x1ayt2xpi9affb8kieye180rp
overall progress: 2 out of 2 tasks
1/2: running   [==================================================>]
2/2: running   [==================================================>]
verify: Service x1ayt2xpi9affb8kieye180rp converged
anil@manager:~/flask-fibonacci-microservice$ sudo docker service ps
```

Fig. 20. Creating flask-fibonacci-microservice instances

Running instances of microservice: ***flask-fibonacci-microservice*** on manager and workers

```
anil@manager:~/flask-fibonacci-microservice$ sudo docker service ps fibocopy1
ID             NAME          IMAGE                      NODE      DESIRED STATE   CURRENT STATE                ERROR     PORTS
9gf1jobxfkof   fibocopy1.1   anil2018/myapp1:fibonacci   worker1   Running         Running about a minute ago
vnyfdorqdzje   fibocopy1.2   anil2018/myapp1:fibonacci   manager   Running         Running about a minute ago
anil@manager:~/flask-fibonacci-microservice$
```

Fig. 21. Running flask-fibonacci-microservice instances on manager and worker

Running instances of microservice on web browser

To verify the successful deployment and accessibility of the containerized microservice, the running instances of the Flask-Fibonacci Microservice were accessed through web browsers using the IP addresses of the Docker Swarm Manager and Worker nodes. The browser-based verification confirmed that the microservice containers were executing correctly across the cluster and responding to client requests. This step also validated the service deployment, network connectivity, and distributed execution of microservice instances within the Docker Swarm environment.

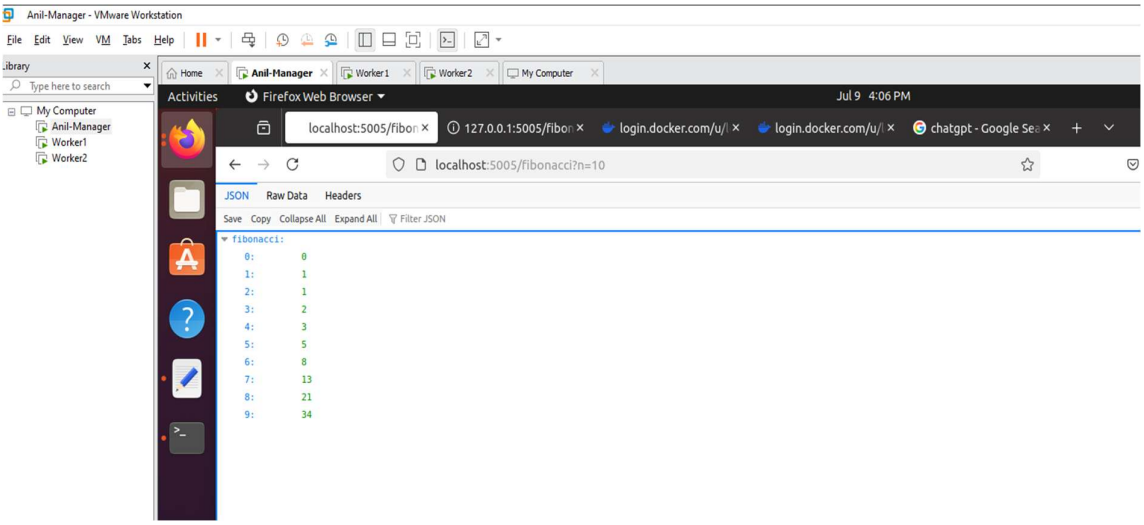


Fig. 22. Running microservice

4.3 Deployment Scenarios

Two experimental deployment scenarios, Clean Deployment and Cumulative Deployment, were created in order to fully evaluate the performance of Random, Spread, Binpack, MCSCM, and the proposed MCSCM-RL scheduling algorithm. To evaluate each scheduling strategy's isolated and progressive behaviour under various system loads, these scenarios were methodically run throughout the Docker Swarm cluster.

Clean deployment: To guarantee an unbiased assessment of performance, every scheduling strategy in this case was tried in a separate environment. Every running container and microservice was deleted from every node in the Swarm cluster before a fresh set of container

deployments was created. The cluster was therefore restored to its initial state of cleanliness. The corresponding scheduling policy was then followed in the sequential deployment of a predetermined number of microservices.

Cumulative deployment: The objective of the scenario was to examine how well each scheduling method could adjust and scale its performance under dynamic workload conditions. Containers and microservices were not deleted in between deployment cycles, in contrast to the clean deployment scenario. Rather, a progressively increasing load on the cluster was simulated by deploying microservices incrementally.

4.4 Experiment Procedure

The experimental procedure was designed to evaluate the performance of the proposed Multi-Criteria Scheduling of Containerized Microservices using Reinforcement Learning (MCSCM-RL) framework in a Docker Swarm environment. Initially, a Docker Swarm cluster consisting of one manager node and two worker nodes was configured using Ubuntu virtual machines hosted on a Windows 11 system. The Flask-Fibonacci microservice was containerized, published to the Docker Hub repository (**anil2018**), and deployed as multiple service replicas across the swarm cluster.

During experimentation, runtime metrics including CPU utilization, memory utilization, container count, startup time, and response time were continuously collected from all worker nodes. These metrics were used for node ranking and scheduling decision-making. The proposed MCSCM-RL scheduler was then executed to dynamically select deployment nodes based on real-time cluster conditions. To validate its effectiveness, the proposed approach was compared with Random, Spread, Binpack, and MCSCM scheduling strategies.

Spread, Binpack, Random, custom scheduler MCSCM and MCSCM-RL scheduling strategies executed multiple times under identical workload conditions, and the resulting performance metrics were recorded and analysed. Prior to each trial, the cluster was restored to the required baseline state (empty for clean deployments). 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 container 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 such as 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. 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 variations and network fluctuations, all recorded measurements were averaged over multiple experimental repetitions. The experimental results were subsequently used to evaluate resource utilization, load balancing efficiency, Quality of Service (QoS), and energy efficiency of the proposed scheduling framework.

4.5 Container Scheduler Implementation and Scenarios

This section presents the implementation and execution of existing container scheduling strategies within the Docker Swarm cluster environment. To establish a comprehensive performance comparison, four scheduling approaches, namely Random Scheduler, Spread Scheduler, Binpack Scheduler, and the custom Multi-Criteria Scheduling of Containerized Microservices (MCSCM) scheduler, were implemented and evaluated using the Flask-Fibonacci microservice workload. Each scheduler was executed under identical cluster configurations and workload conditions to ensure fair comparison. During experimentation, multiple microservice containers were deployed across the worker nodes of the swarm cluster. The performance metrics were continuously monitored and recorded. Screenshots illustrating scheduler execution, container deployment, node selection, service status, and workload distribution are included to demonstrate the practical implementation of each scheduling strategy.

4.5.1 Binpack Scheduler

The Binpack scheduler was implemented to deploy containerized microservices. The objective of this strategy is to consolidate workloads onto fewer cluster nodes, thereby improving resource utilization and potentially reducing energy consumption. The Fig. 23. and Fig. 24 illustrates the execution of the Binpack scheduler, node selection process, and successful deployment of microservice containers within the Docker Swarm cluster.

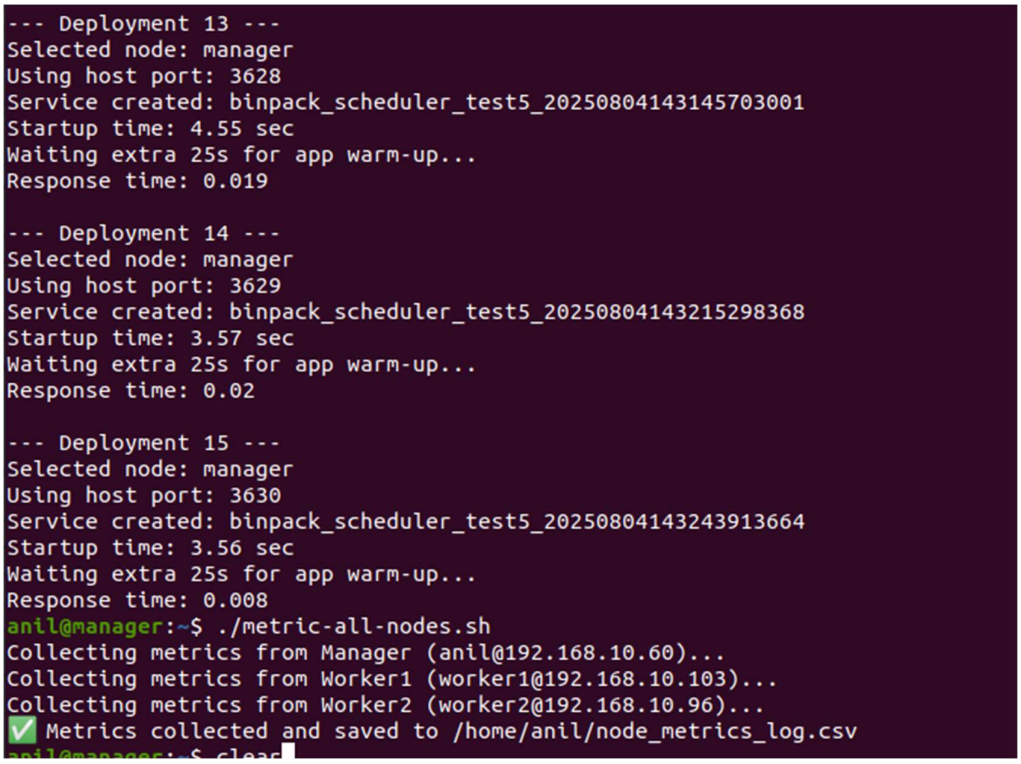


Fig. 23. Deployment of Services

Before executing a new scheduling experiment, all previously deployed services were removed from the Docker Swarm cluster to ensure a clean and consistent deployment environment as shown in Fig. 25. This step eliminated residual workloads and resource utilization from earlier experiments, thereby preventing interference with subsequent scheduling evaluations. The clean deployment process ensured fair comparison among the different scheduling strategies under identical cluster conditions.

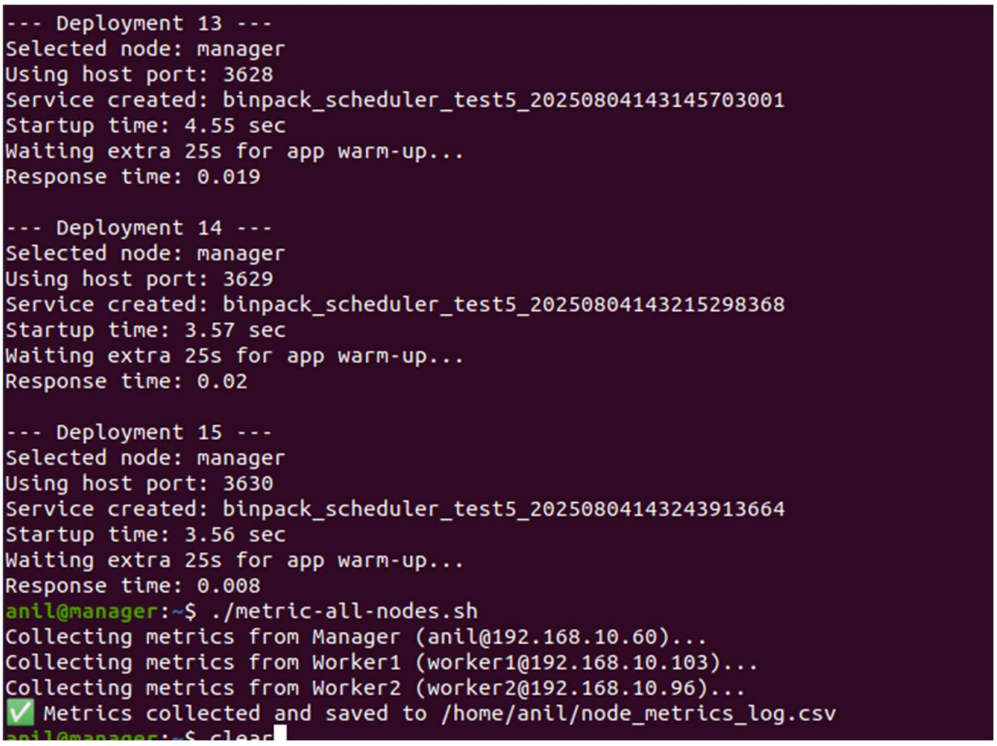


Fig. 24. Deployment of Services and logging metrics for each swarm node

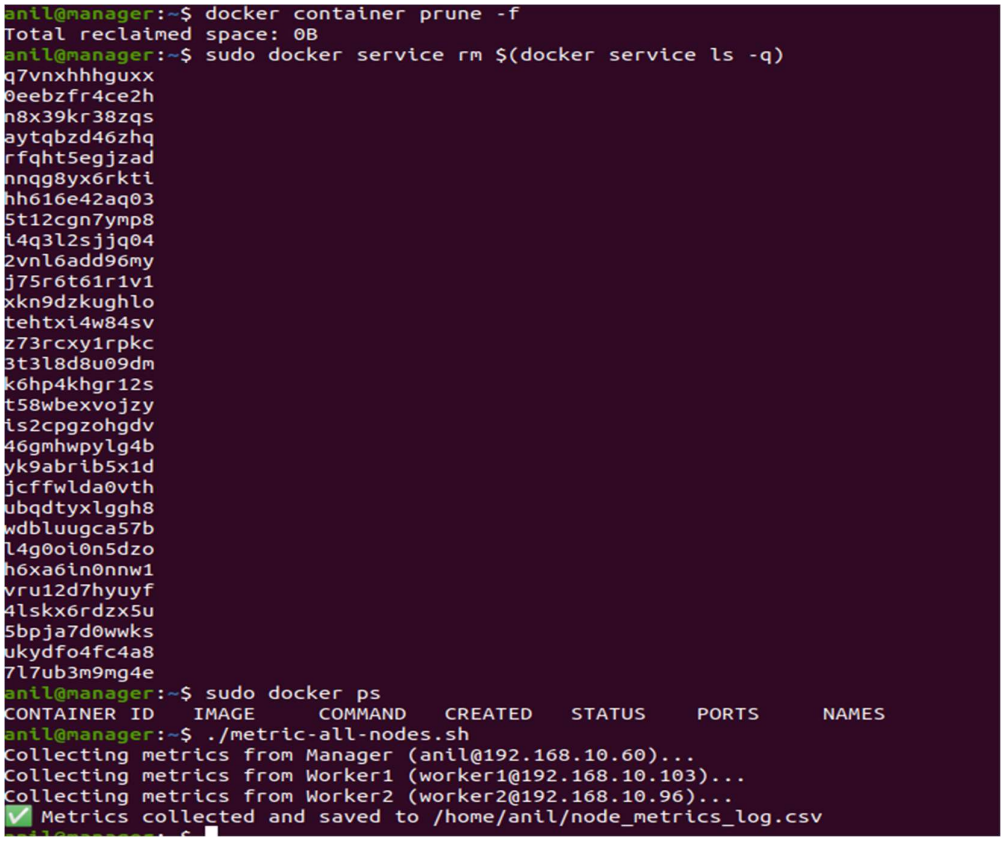


Fig. 25. Removing Services

4.5.2 Spread Scheduler

The Spread scheduler was implemented to distribute microservice containers evenly across the available worker nodes in the Docker Swarm cluster. The primary objective is to achieve balanced resource utilization by selecting nodes with fewer running containers and lower workload concentration. During execution, the scheduler analysed the current cluster state and deployed the Flask-Fibonacci microservice on the most suitable node to maintain an even distribution of workloads. The Figure-26 and Figure-27 illustrate the execution of the Spread scheduler, node selection process, and successful deployment of microservice containers across the cluster.

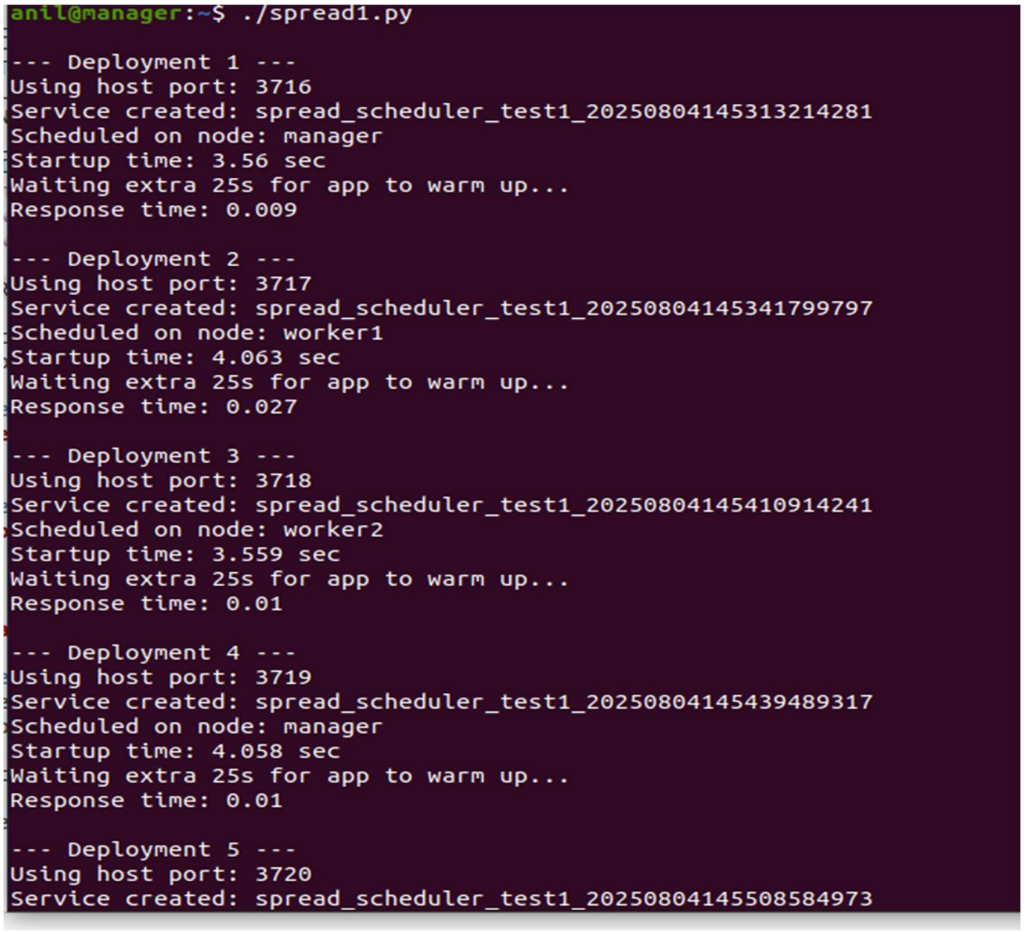


Fig. 26. Deployment of Services

Before initiating the next scheduling experiment, all running services deployed by the Spread scheduler were removed from the Docker Swarm cluster to maintain a consistent and unbiased test environment as shown in Fig. 28. Performing a clean deployment before each experiment allowed fair comparison among the different scheduling strategies.

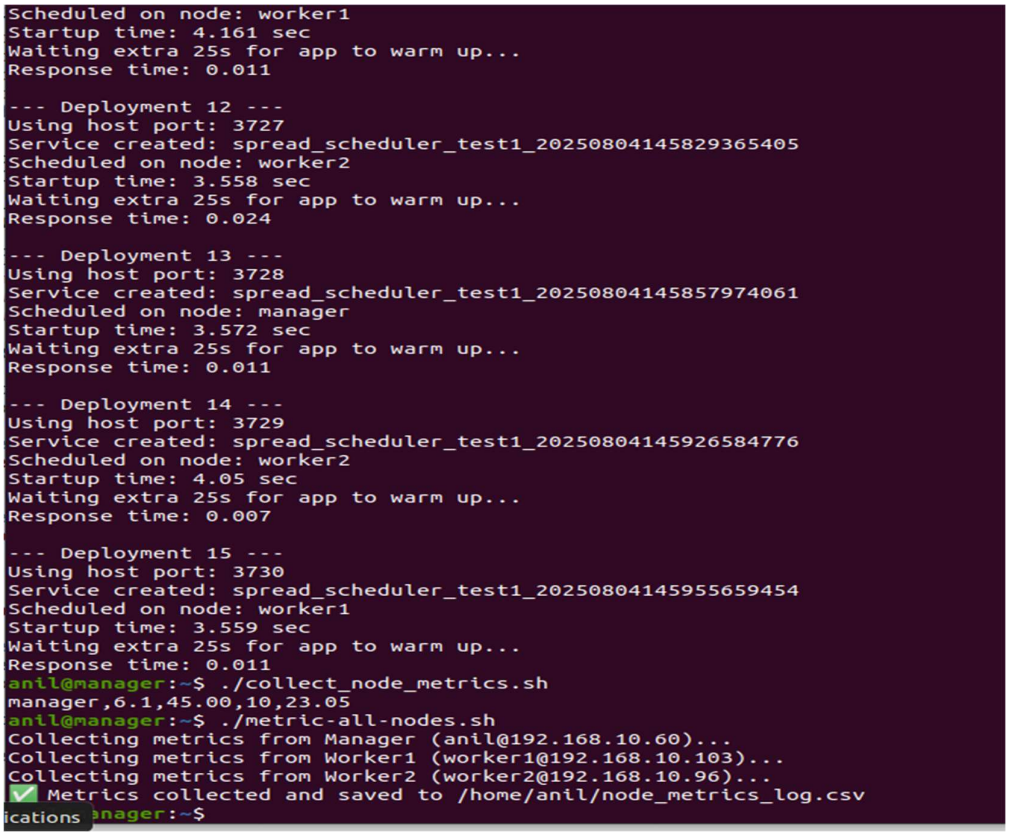


Fig. 27. Deployment of services and logging metrics for each swarm node

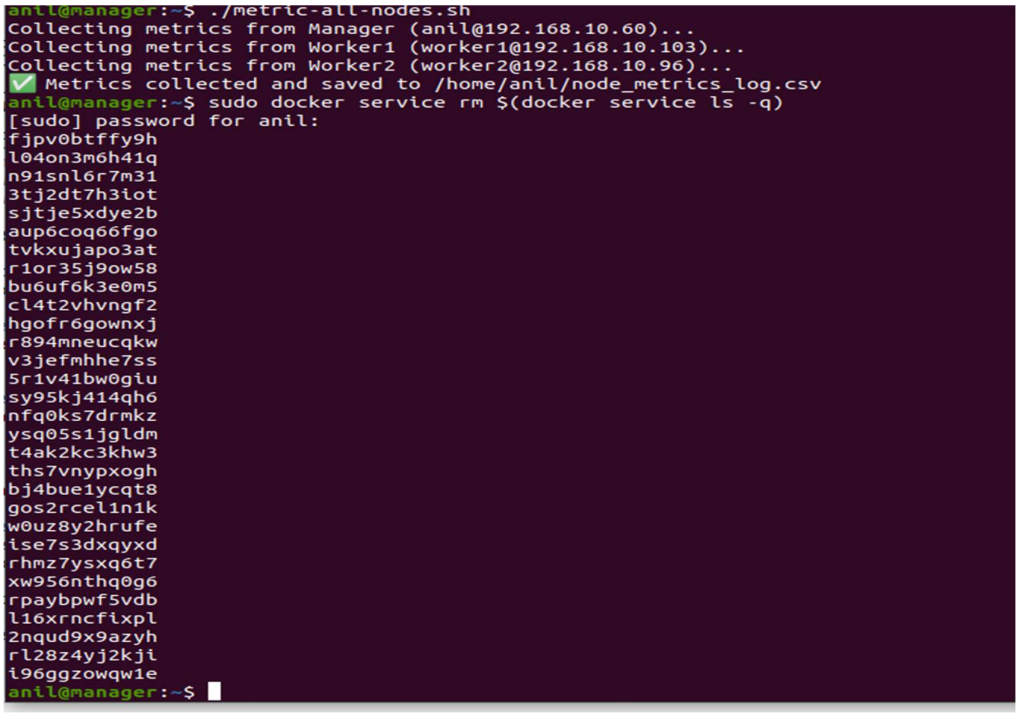


Fig. 28. Removing Services

4.5.3 Random Scheduler

The Random scheduler was implemented to deploy microservice containers by randomly selecting one of the available worker nodes in the Docker Swarm cluster without considering resource utilization, workload distribution, or performance metrics. During execution, the scheduler randomly chose a worker node for deploying the Flask-Fibonacci microservice, and the deployment process was monitored to analyse its impact on resource utilization and application performance.

The Fig. 29 and Fig. 30 illustrate the execution of the Random scheduler, node selection process, and successful deployment of microservice containers within the cluster.

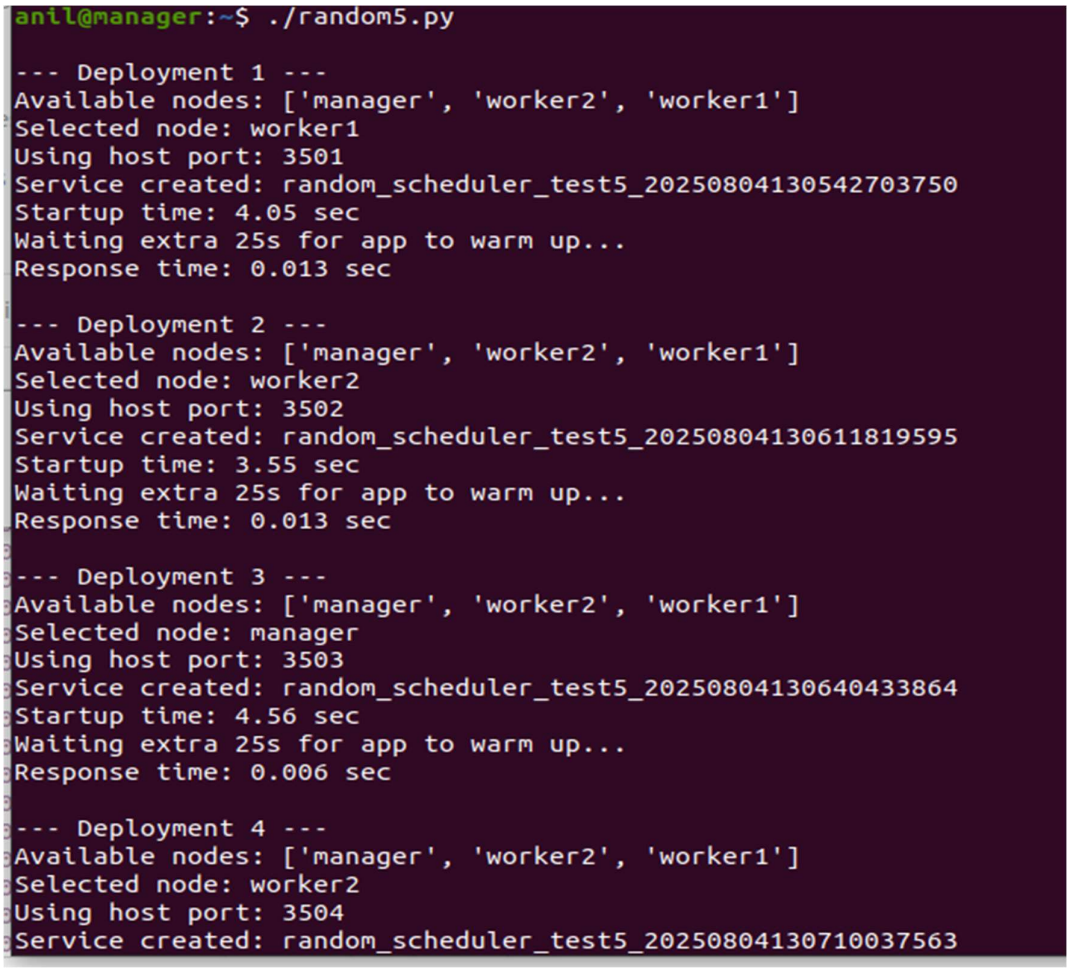


Fig. 29. Deployment of Services

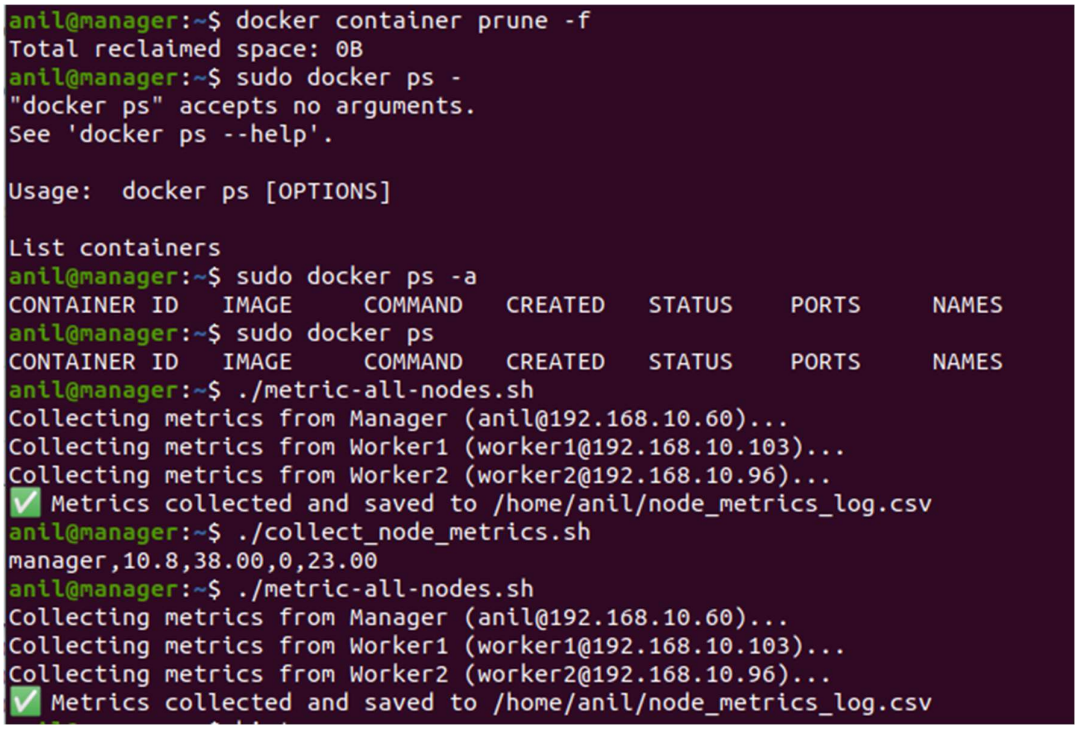


Fig. 30. Deployment of services and logging metrics for each swarm node

After completing the Random scheduling experiment, all deployed services were removed from the Docker Swarm cluster to restore a clean experimental environment to maintain a clean cluster state before each scheduling evaluation enabled accurate and fair comparison among all scheduling strategies as shown in Fig. 31.

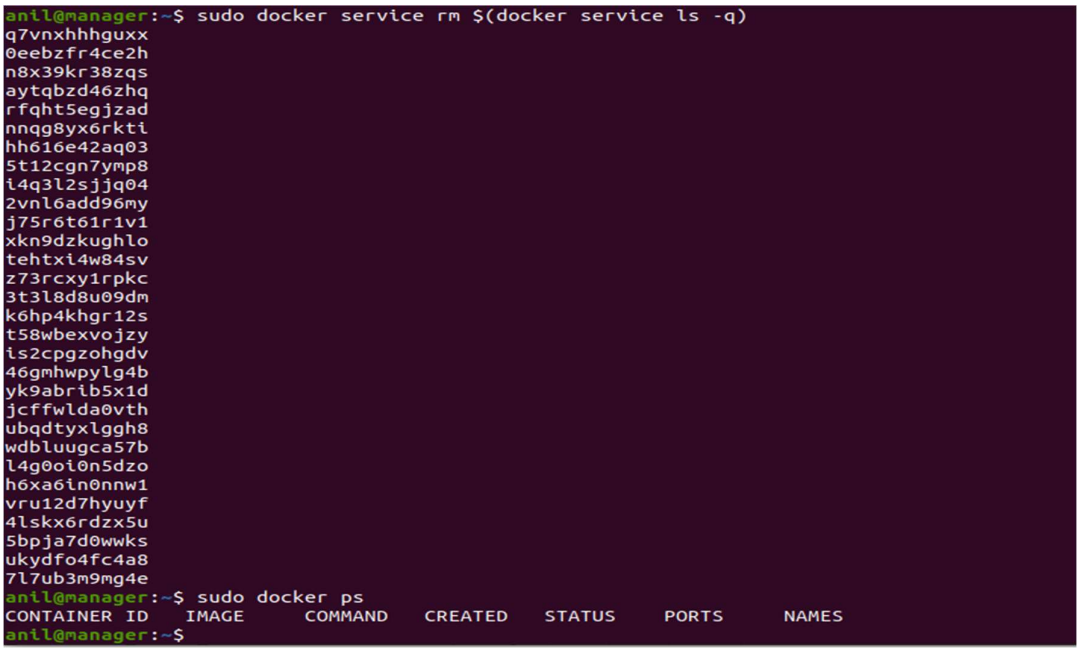


Fig. 31. Removing Services

4.5.4 MCSCM Scheduler

The Multi-Criteria Scheduling of Containerized Microservices (MCSCM) scheduler was implemented to deploy microservice containers based on a comprehensive evaluation of multiple resource-aware and application-aware parameters. Unlike traditional scheduling strategies, the MCSCM scheduler considers metrics such as CPU utilization, memory utilization, container count, startup time, and response time during the node selection process. A multi-criteria node-ranking mechanism based on the relative closeness function was used to identify the most suitable worker node for deployment. The Fig. 32 illustrates the execution of the MCSCM scheduler, node ranking process, and successful deployment of microservice containers within the Docker Swarm cluster.

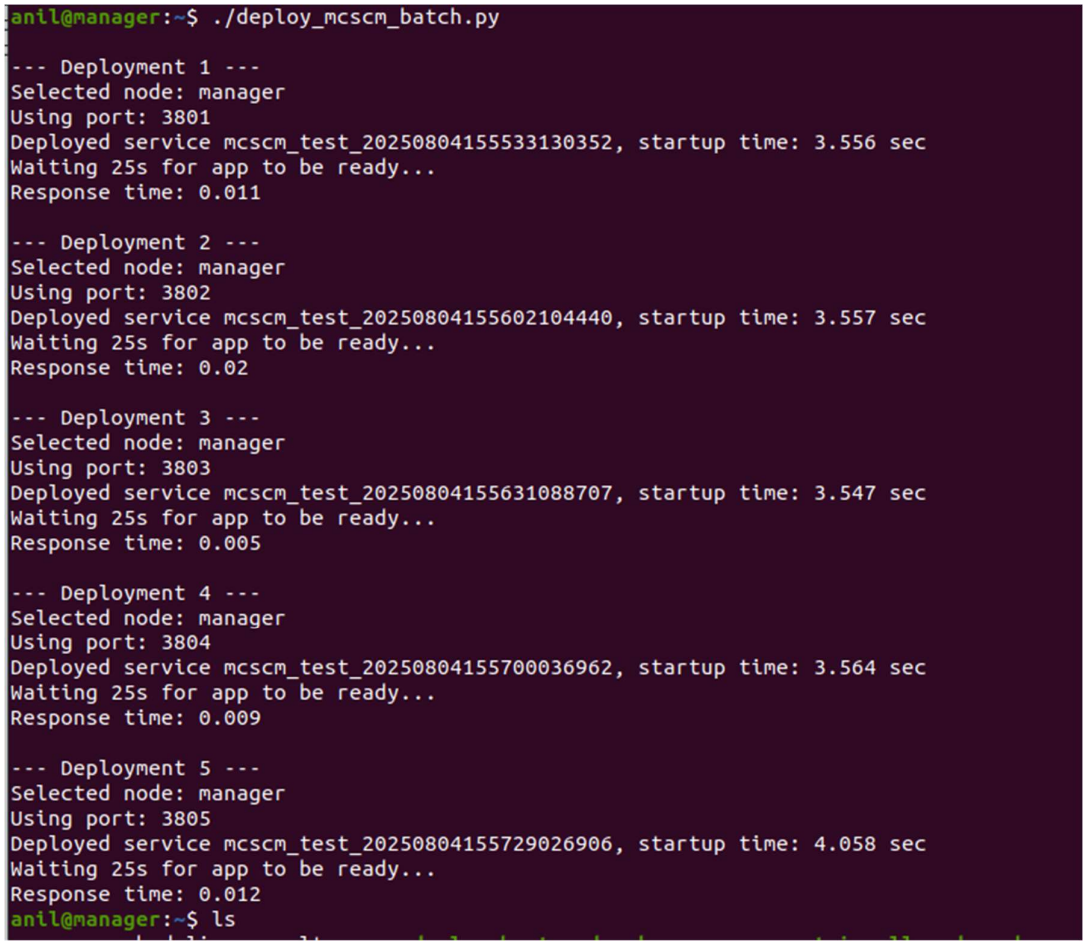


Fig. 32. Deployment of Services

After completing the MCSCM scheduling experiment, all deployed services were removed from the Docker Swarm cluster as shown in Fig. 33.

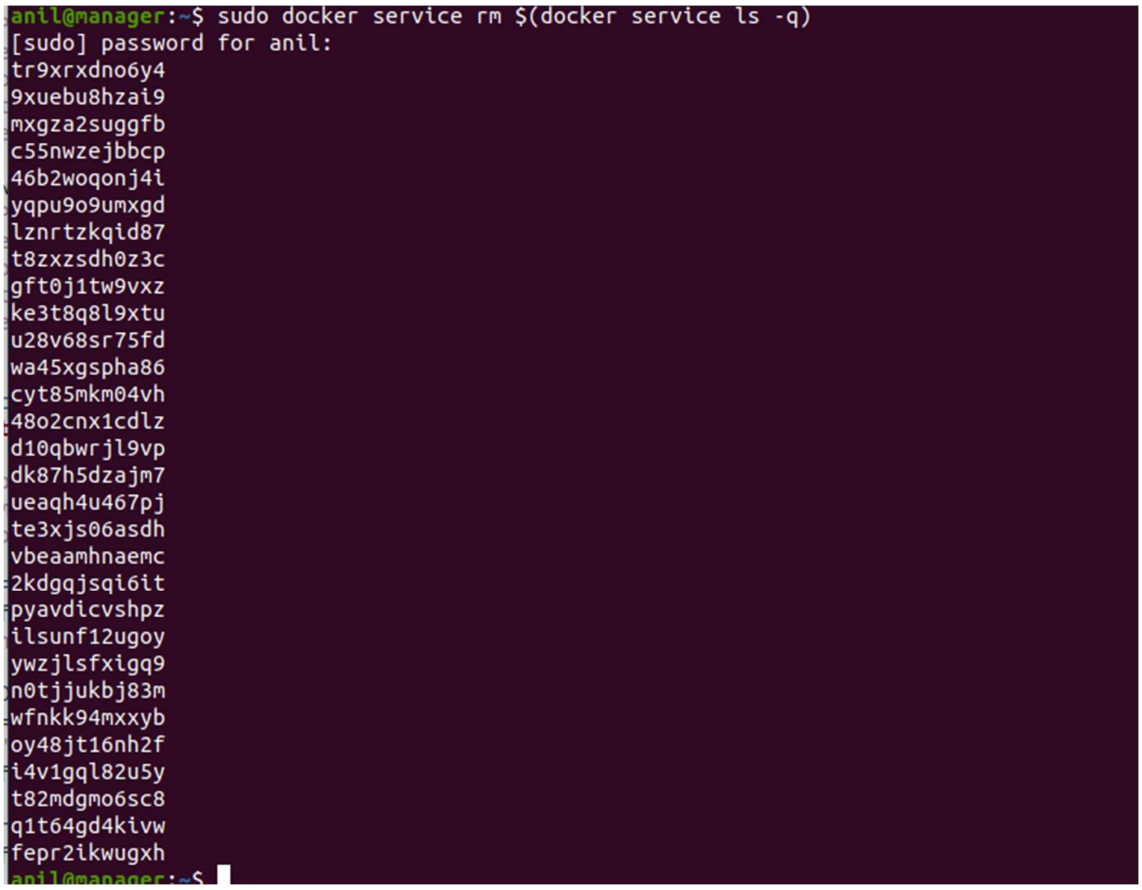


Fig. 33. Removing Services

4.6 MCSCM-RL Scheduler Implementation and Experimental Scenarios

The proposed Multi-Criteria Scheduling of Containerized Microservices using Reinforcement Learning (MCSCM-RL) scheduler was implemented to enable intelligent and adaptive container deployment in the Docker Swarm environment.

This scheduler combines multi-criteria node ranking with a Deep Q-Network (DQN)-based reinforcement learning model to make deployment decisions based on real-time cluster conditions. Fig. 34-35-36 illustrate successful deployment of microservice containers using the proposed MCSCM-RL scheduling framework under clean and cumulative load.

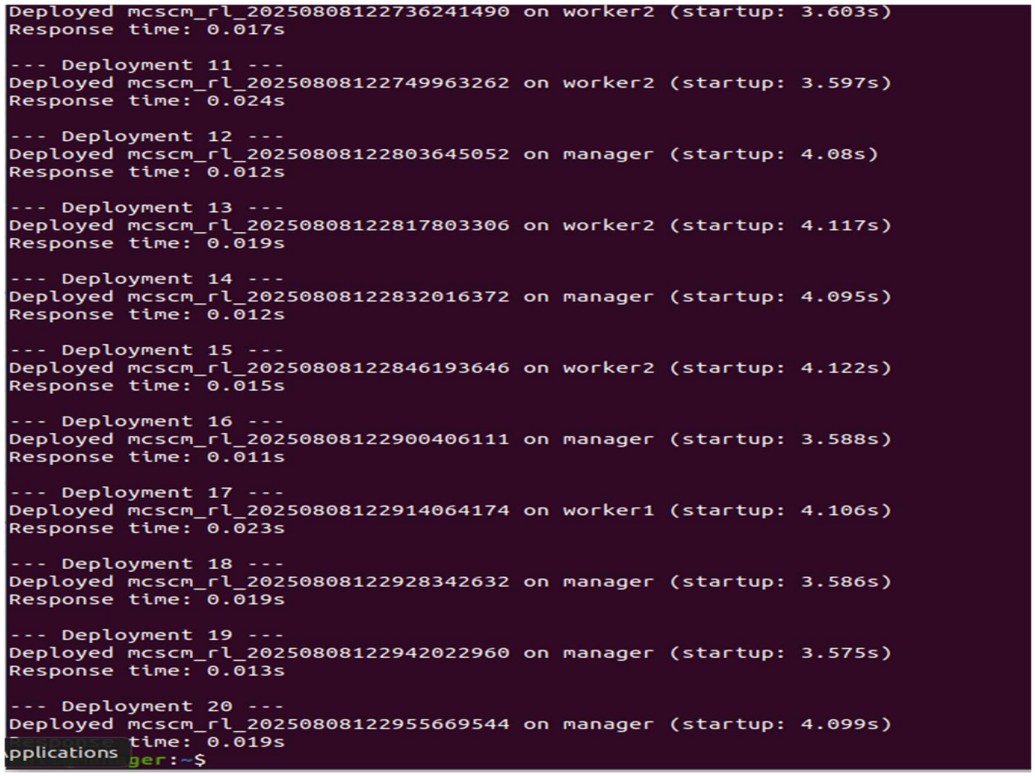


Fig. 34. Deployment of Services

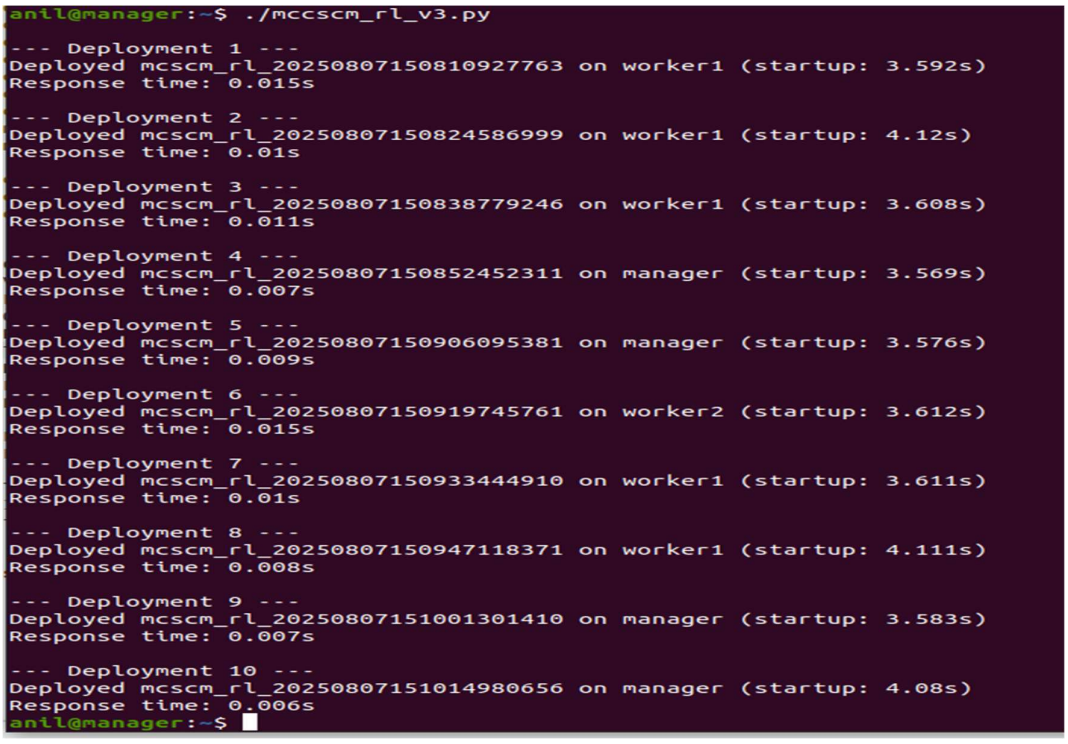


Fig. 35. Clean Deployment

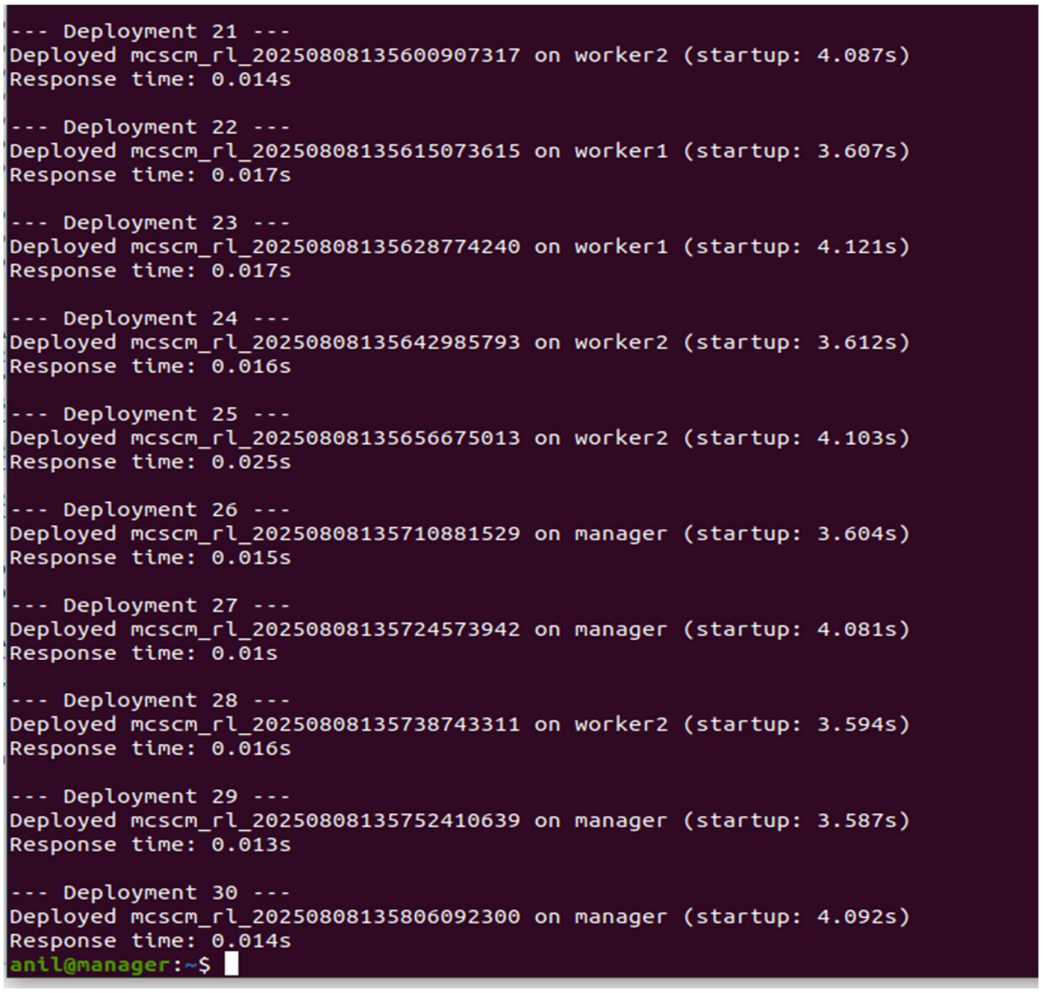


Fig. 36. Cumulative Deployment

Fig. 37 demonstrates the status of microservice services deployed across the Docker Swarm cluster nodes using the proposed MCSCM-RL scheduling strategy. The figure shows the distribution of containerized microservice instances among the manager and worker nodes based on the scheduling decisions made by the reinforcement learning agent. It confirms the successful deployment of services and illustrates how the MCSCM-RL scheduler dynamically allocates workloads according to real-time resource conditions, node rankings, and QoS requirements within the cluster environment.

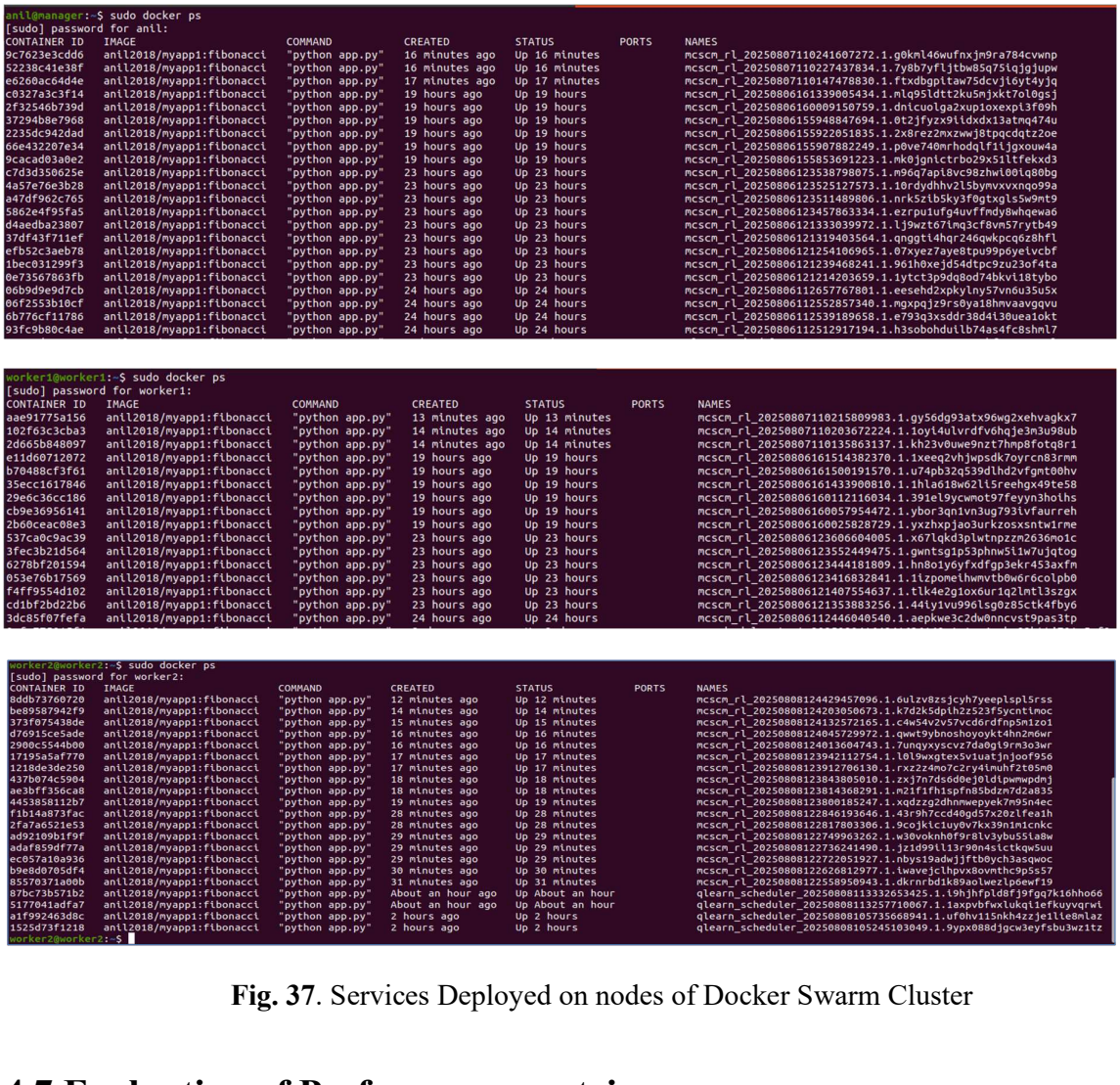
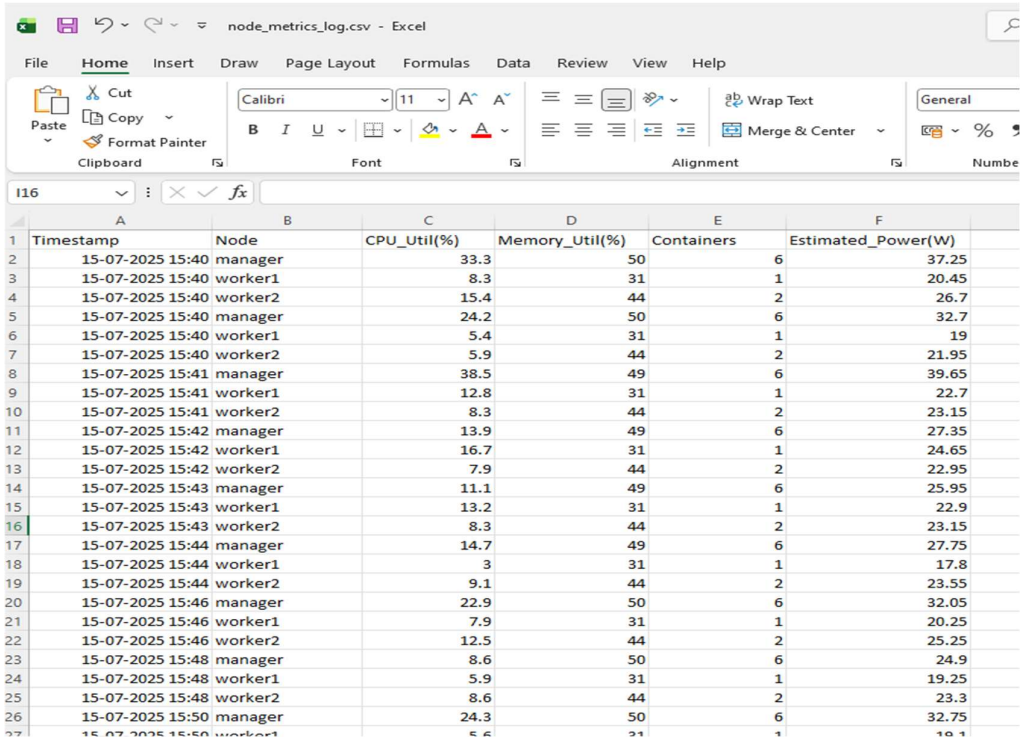


Fig. 37. Services Deployed on nodes of Docker Swarm Cluster

4.7 Evaluation of Performance metrics

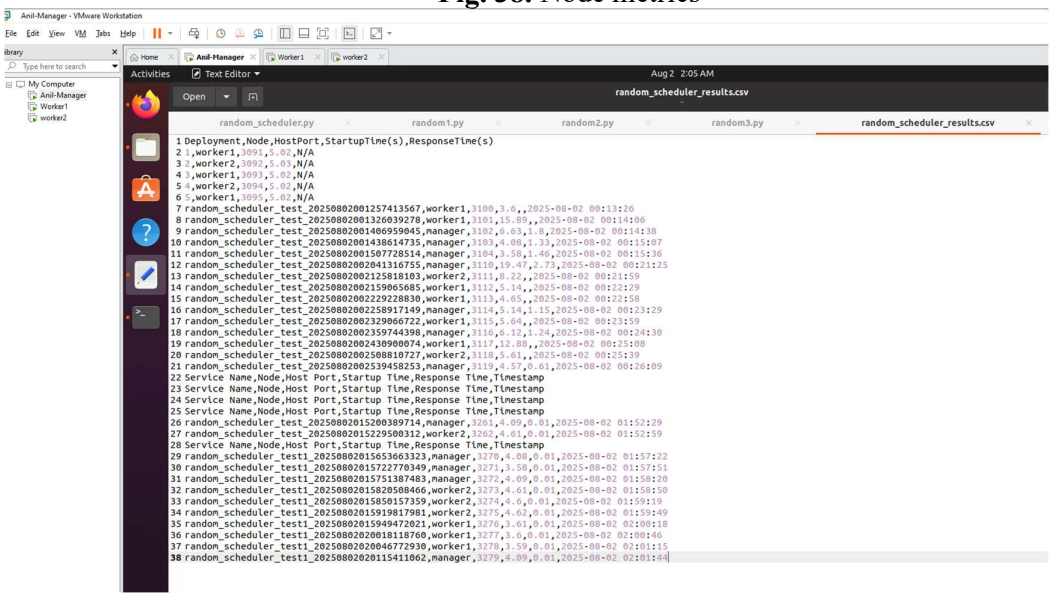
To support intelligent scheduling decisions and performance evaluation, the proposed framework continuously collects and logs both node-level resource metrics and application-level QoS metrics during each microservice deployment. The collected node metrics include CPU utilization, memory utilization, and container count, and the QoS metrics include startup time and response time of deployed microservices. These metrics are periodically recorded into log files using the monitoring and performance measurement modules implemented in the Docker Swarm environment. The logged data serves as the input for multi-criteria node ranking, reinforcement learning-based scheduling decisions, and comparative performance analysis of different scheduling strategies. The Figure-38 and Figure-39 illustrate a log file containing the recorded node metrics and QoS metrics collected during the execution of the proposed MCSCM-RL scheduling framework.



node_metrics_log.csv - Excel

	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

Fig. 38. Node metrics



random_scheduler_results.csv

```
Deployment,Node,HostPort,StartupTime(s),ResponseTime(s)
1,worker1,3091,5.02,N/A
2,worker2,3092,5.03,N/A
3,worker1,3093,5.02,N/A
4,worker2,3094,5.02,N/A
5,worker1,3095,5.02,N/A
random_scheduler_test_20250802001257413567,worker1,3100,3.6,,2025-08-02 00:13:26
random_scheduler_test_20250802001326039278,worker1,3101,15.89,,2025-08-02 00:14:06
random_scheduler_test_20250802001406959045,manager,3102,6.63,1.8,2025-08-02 00:14:38
random_scheduler_test_20250802001438614735,manager,3103,4.08,1.33,2025-08-02 00:15:07
random_scheduler_test_20250802001507728514,manager,3104,3.58,1.46,2025-08-02 00:15:36
random_scheduler_test_20250802002041316755,manager,3110,19.47,2.73,2025-08-02 00:21:25
random_scheduler_test_20250802002125818103,worker2,3111,8.22,,2025-08-02 00:21:59
random_scheduler_test_20250802002159065685,worker1,3112,5.14,,2025-08-02 00:22:29
random_scheduler_test_20250802002229228830,worker1,3113,4.65,,2025-08-02 00:22:58
random_scheduler_test_20250802002258917149,manager,3114,5.14,1.15,2025-08-02 00:23:29
random_scheduler_test_20250802002329066722,worker1,3115,5.64,,2025-08-02 00:23:59
random_scheduler_test_20250802002359744398,manager,3116,6.12,1.24,2025-08-02 00:24:30
random_scheduler_test_20250802002430900074,worker1,3117,12.88,,2025-08-02 00:25:08
random_scheduler_test_20250802002508810727,worker2,3118,5.61,,2025-08-02 00:25:39
random_scheduler_test_20250802002539458253,manager,3119,4.57,0.01,2025-08-02 00:26:09
Service Name,Node,Host Port,Startup Time,Response Time,Timestamp
Service Name,Node,Host Port,Startup Time,Response Time,Timestamp
Service Name,Node,Host Port,Startup Time,Response Time,Timestamp
Service Name,Node,Host Port,Startup Time,Response Time,Timestamp
random_scheduler_test_20250802015200389714,manager,3261,4.09,0.01,2025-08-02 01:52:29
random_scheduler_test_20250802015229500312,worker2,3262,4.61,0.01,2025-08-02 01:52:59
Service Name,Node,Host Port,Startup Time,Response Time,Timestamp
random_scheduler_test1_20250802015653663323,manager,3270,4.08,0.01,2025-08-02 01:57:22
random_scheduler_test1_20250802015722770349,manager,3271,3.58,0.01,2025-08-02 01:57:51
random_scheduler_test1_20250802015751387483,manager,3272,4.09,0.01,2025-08-02 01:58:20
random_scheduler_test1_20250802015820508466,worker2,3273,4.61,0.01,2025-08-02 01:58:50
random_scheduler_test1_20250802015850157359,worker2,3274,4.6,0.01,2025-08-02 01:59:19
random_scheduler_test1_20250802015919817981,worker2,3275,4.62,0.01,2025-08-02 01:59:49
random_scheduler_test1_20250802015949472021,worker1,3276,3.61,0.01,2025-08-02 02:00:18
random_scheduler_test1_20250802020018118760,worker1,3277,3.6,0.01,2025-08-02 02:00:46
random_scheduler_test1_20250802020046772930,worker1,3278,3.59,0.01,2025-08-02 02:01:15
random_scheduler_test1_20250802020115411062,manager,3279,4.09,0.01,2025-08-02 02:01:44
```

Fig. 39. QoS metrics

Experimental Files and Scripts Used in the Proposed Scheduling Framework

The implementation of the proposed scheduling framework involves several shell scripts, Python programs, and data files for cluster management, metric collection, node ranking, scheduling, logging, and performance evaluation. Table-5 summarizes the purpose and functionality of the major files used throughout the experimental setup and execution process.

Table 6. Experimental Files and Scripts

On Manager Node:	**Purpose**
collect_node_metric.sh	Collect metrics on each cluster node
metric_all.node.sh	Combine all node metrics into a central CSV
node_metrics.csv	Contains node metrics for all nodes
rank_node.py	Apply ranking/ RL model to pick best node
binpack_scheduler_results.csv	Store application metrics (Response time and Start-up time) for Binpack scheduler
binpack1.py	Implements the Binpack scheduling strategy
spread_scheduler_results.csv	Store application metrics (Response time and Start-up time) for spread scheduler
spread1.py	Implements the Spread scheduling strategy
random_scheduler_results.csv	Store application metrics (Response time and Start-up time) for Random scheduler
random.py	Implements the Random scheduling strategy
mcscm_scheduler_results.csv	Store application metrics (Response time and Start-up time) for MCSCM scheduler
mcscm.py	Implements the MCSCM scheduling strategy
mcscm_rl_scheduler_results.csv	Store application metrics (Response time and Start-up time) for MCSCM-RL scheduler
mcscm_rl.py	Implements the MCSCM-RL scheduling strategy
Microservices: *myapp1: fibonacci*	Microservice developed and deployed in container

.

Chapter 5
Performance Evaluation and Result Analysis

The performance comparison of the proposed MCSCM-RL scheduler with the default Docker Swarm container schedulers Binpack, Spread, Random, and the non-RL multi-criteria scheduler (MCSCM) is presented in this chapter. In order to record the performance under low load and gradually growing cluster usage, the assessment considers both clean and cumulative load deployment scenarios. One manager node and two worker nodes made up the three-node Docker Swarm cluster configuration used for all deployments of the experiment. Each container scheduling strategy was tested on comparable deployment iterations.

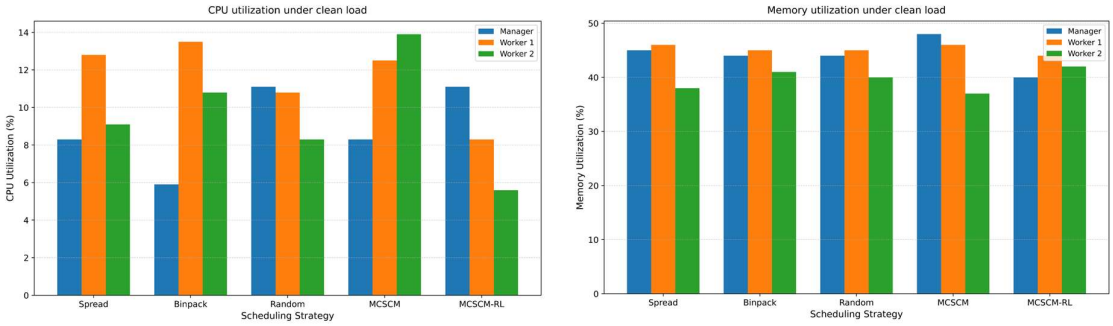
Multiple container deployments of microservices were used to evaluate the performance of the MCSCM-RL scheduler versus Docker Swarm's default schedulers in order to provide a fair and comprehensive comparison. Performance metrics are taken into account, such as average startup time, average response time across nodes, CPU utilization, memory utilization, and number of containers deployed on the node. Together, these metrics capture the application-level performance of the microservices and system-level efficiency.

5.1 Performance Evaluation under Clean Deployment Conditions

To ensure fair and unbiased comparison among the all existing and proposed container scheduling strategies, all experiments were carried out under clean deployment scenario. Before executing each scheduling strategy, all previously deployed microservices and containers were removed from the Docker Swarm cluster, thereby resetting the cluster to its initial state. Subsequently, identical containerized microservices were deployed using Spread, Binpack, Random, MCSCM, and MCSCM-RL scheduling strategies. The results obtained under clean deployment scenarios provide a consistent basis for evaluating the effectiveness of the proposed scheduling framework and comparing its performance with existing scheduling approaches.

Under this scenario, performance differences among all container scheduling strategies are relatively small due to uniform resource availability. Nonetheless, the MCSCM-RL scheduler consistently maintains balanced CPU and memory consumption while achieving marginally faster startup and response times. The resource utilization for all scheduling strategies is shown

in Fig. 40. To enable accurate evaluation under clean load, Fig. 40(a) shows CPU utilization and Fig. 40(b) shows Memory utilization for three swarm nodes under all scheduling strategies.



(a) CPU Utilization (b) Memory Utilization

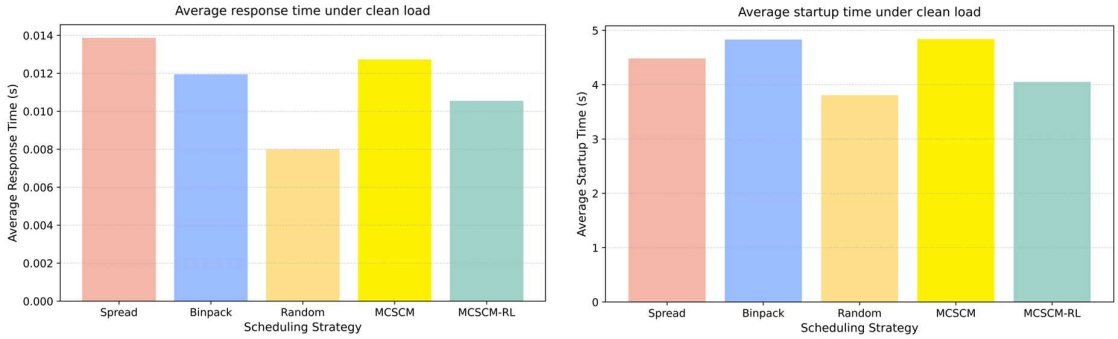
Fig. 40. Resource Utilization under clean deployment

Fig. 40(a) demonstrates the CPU utilization across the all 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 shows comparatively high CPU utilization (approximately 12–13%), which reflects behaviour of default Binpack and Spread scheduling strategies. Random scheduler gives imbalanced CPU utilization across the cluster, with worker1 reaching 10.8% utilization and worker2 8.3%. MCSCM exhibits the non-deterministic nature of scheduling strategy, which increases CPU utilization on worker nodes only, 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 cluster nodes more evenly. Its adaptive decision-making avoids repeated selection of a single cluster node, making it more efficient even under clean load scenario. These results indicate that MCSCM-RL reduces unnecessary CPU overhead by dynamically selecting nodes based on learned system states rather than static container placements.

Fig. 40(b) presents memory utilization for each scheduling strategy across the three nodes of the cluster. The results show 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 favours already stronger cluster 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 on the cluster. The memory balancing behaviour of MCSCM-RL shows its ability to prevent early memory accumulation on specific nodes compared to deterministic container schedulers.

The application performance for all scheduling strategies is shown in Figure-41. Figure 41(a) shows average response time and Figure 41(b) shows average startup time for all scheduling strategies.



(a) Average Response time

(b) Average Startup time

Fig. 41. Application Performance under clean deployment

Each scheduling strategy was evaluated on a Docker Swarm cluster fully reset to an empty state before the deployment of containerised microservices. 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 container deployments.

Fig. 41(a) shows the average response time of the containerized microservice under each container scheduling strategy across the cluster. The response-time analysis shows clear distinctions in service latency. Random scheduler achieves the lowest response time (0.008 s), but this performance is inconsistent. Spread scheduler achieves (0.0139 s) and MCSCM scheduler achieves (0.0127 s) which exhibits higher latencies, as both distribute containerized microservices without adaptation to real-time cluster node conditions. Binpack scheduler performs slightly better (0.01195 s) than others, but still lacks adaptive balancing. MCSCM-RL achieves 0.01055 s, which exhibits improved response time over Spread scheduler and MCSCM scheduler. It maintains stable and predictable performance. The RL-based scheduler

learns to avoid nodes with even slightly higher CPU or Memory loads, yielding more consistent and lower latency compared to deterministic scheduling strategies.

Fig. 41(b) demonstrates the average startup time under clean load deployment scenario. Random scheduler achieves the lowest startup time, that is 3.808 s. Due to non-deterministic behaviour of Random scheduler, this result is not stable or repeatable. Spread scheduler achieves 4.48 s and Binpack scheduler achieves 4.83 s, that shows moderate startup delay. Which reflects their respective default load allocation behaviours.

MCSCM scheduler achieves 4.84 s, because of static multi-criteria bias toward specific nodes, which increases container initialization overhead. MCSCM-RL scheduler achieves significantly improves startup time, it is 4.05 s, and outperforms Spread, Binpack, and MCSCM schedulers. The improved startup time in MCSCM-RL results from its dynamic action that avoids nodes with higher current microservice counts or continuing container creation operations, that enables 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 shown in Fig. 42.

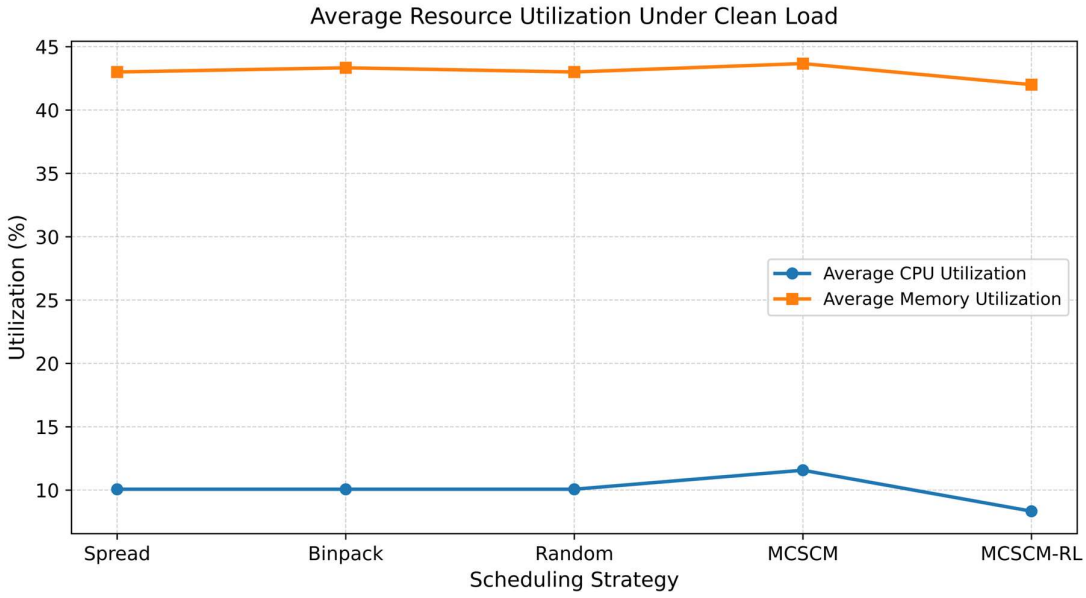


Fig. 42. Average Resource utilization under clean load

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 container

scheduling strategies. Docker swarm strategies (Spread, Binpack, Random) are exhibiting similar utilization patterns as they are static and non-adaptive by nature. The RL-based method recognizes the nature of the cluster conditions and learn from it. Thus, it gives the early benefits that eventually result in measurable improvements in efficiency at cluster level.

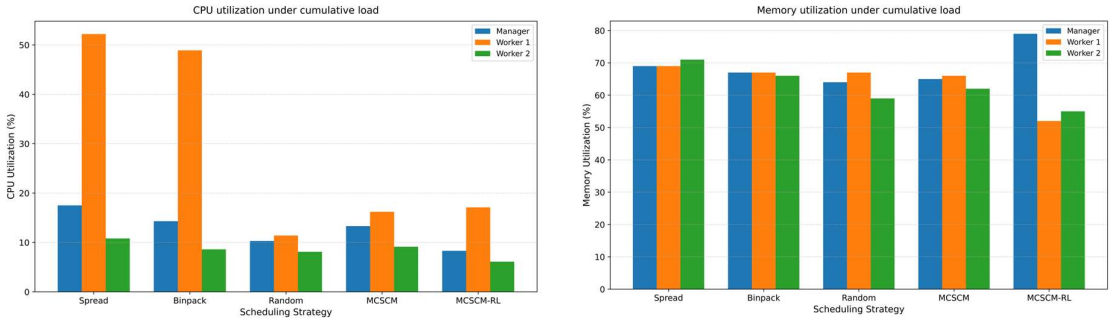
Random scheduling occasionally demonstrates competitive or superior startup and response time performance in the clean deployment scenario. These results, where all swarm nodes initially have the same resource availability, are caused by the lack of resource contention. Under such conditions, random placement unconditionally deploys containers across swarm cluster nodes, rather than considering workloads and possibility of resource saturation. Because of this, random scheduling may benefit temporarily from balanced resource utilization without requiring explicit decision logic.

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

5.2 Performance Evaluation under Cumulative Deployment conditions

Under cumulative deployment scenarios, each new microservice is deployed without removing the previously deployed containers. In this resource balancing scenario, the container scheduler must respond to an increase in workload while simultaneously minimizing delay, preventing resource saturation, and maintaining a good resource balance. With the lowest average CPU utilization, balanced memory utilization, stable response time, and the lowest startup time, MCSCM-RL continuously outperforms all other scheduling techniques in this cumulative deployment scenario. The results across all performance metrics and cluster are summarized in Fig. 43. and Fig. 44.

To enable accurate evaluation under cumulative load, Fig. 43(a) shows CPU utilization and Fig. 43(b) shows Memory utilization for three swarm nodes under all scheduling strategies.



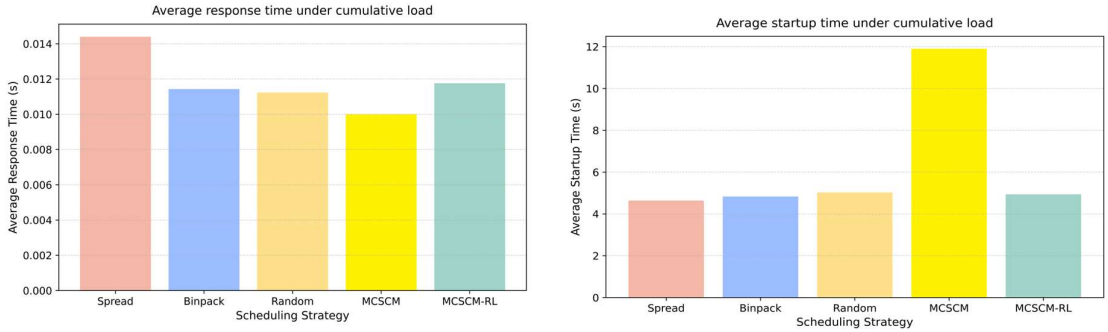
(a) CPU Utilization (b) Memory Utilization

Fig. 43. Resource Utilization under cumulative deployment

Fig. 43(a) shows CPU utilization all swarm cluster nodes for all container scheduling strategies. Binpack scheduler and Spread scheduler generate high CPU utilization on worker1, that are 49% and 52% respectively. It indicates heavy unbalancing in load. Random scheduler performs better than that, it is 11% on worker1, it still shows unbalanced CPU utilization across swarm nodes due to random patterns. MCSCM achieves 14% CPU load on worker2, that shows static bias in scoring of cluster nodes. MCSCM-RL maintains the most balanced CPU utilization. MCSCM-RL prevents extreme hotspots. In contrast, Spread scheduler and Binpack scheduler show five times higher CPU utilization on worker1 compared to other cluster nodes.

Fig. 43(b) shows memory utilization under cumulative load. Spread scheduler shows the highest memory utilization, with worker2 it reaches to 71% and all swarm nodes staying near or above 69%. Binpack scheduler exhibits similar memory utilization, that is 66–67%, it is because of its consolidation-oriented behaviour. Random scheduler slightly reduces memory utilization is 59% on worker2, still the container distribution remains inconsistent.

MCSCM keeps memory usage moderately balanced but inflation persists due to repeated selection of static nodes. MCSCM-RL exhibits the most balanced memory utilization with worker2 and worker1, and maintains significantly lower utilization that is 55% and 52% 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 sustainable container deployments without degrading cluster performance.



(a) Average Response time

(b) Average Startup time

Fig. 44. Application performance under cumulative deployment

Fig. 45(a) shows the average response time for each container scheduler. Binpack scheduler and Random scheduler give moderate performance; they give 0.01143 s and 0.01123 s average response time respectively. But both exhibit fluctuations due to unstable cluster node selection for the deployment. Spread scheduler gives average response time of 0.0144s and performs the worst due to growing contention on worker1 and memory exhaustion across cluster nodes. MCSCM gives the lowest response time, that is 0.0100 s. MCSCM-RL gives average response time of 0.01244s and exhibits the competitive performance with deviations arising from adaptive exploration. Even as deployments increase, MCSCM-RL provides more reliable response times with significantly fewer high-latency spikes.

Fig. 45(b) shows the average startup time for each container scheduler. Spread, Binpack, and Random exhibits the average startup times in the range of 4.6–5.0 s, and gradually increasing as cluster resource load. MCSCM suffers heavily, with average startup time of 11.904 s, that shows static ranking leads to overloading of node. MCSCM-RL offers the best startup performance, it gives average response time of 4.57 s. It significantly outperforming all other container scheduling strategies except Random scheduler. Its adaptive node selection helps avoid overloaded nodes, reducing container initialization delays. This exhibits MCSCM-RL's ability to adaptively maintain QoS even under heavy cumulative deployment on the cluster.

The average resource utilization (CPU and memory) across the three-node Docker Swarm cluster for all container scheduling strategies under cumulative load deployment is presented in Fig. 45.

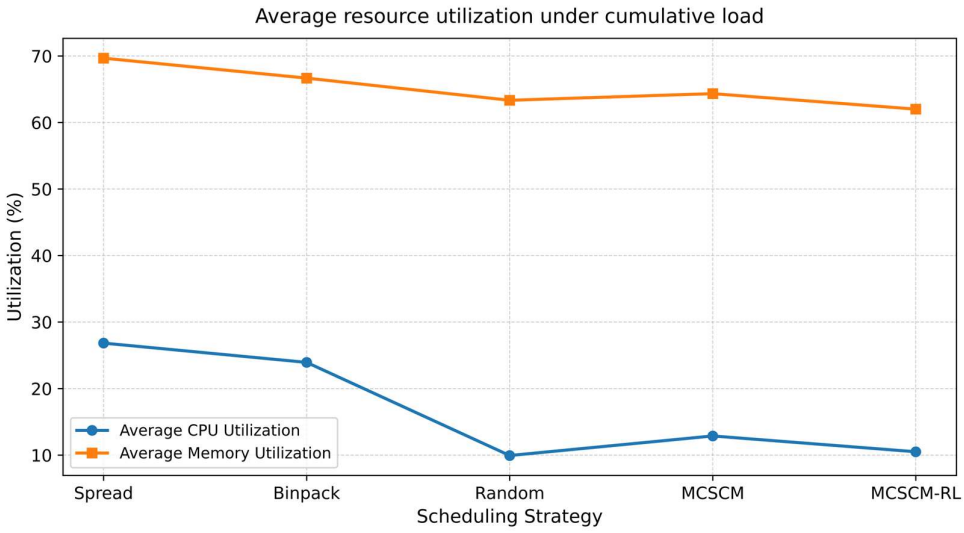


Fig. 45. Average Resource utilization under cumulative load

The results show a distinct behavioural difference. Spread scheduler exhibits the highest resource consumption, with average CPU utilization of 27% and average memory utilization of 70%. As microservices deployments increases, Spread continues distributing replicas uniformly, causing worker1 to become heavily loaded due to its higher baseline capability. Binpack scheduler shows slightly lower memory utilization than Spread scheduler. But CPU utilization is nearly 24%, as it continues deploying microservices onto nodes that initially appear least loaded. By the time, this creates a burdened resource load on worker1. Random scheduler retains lower CPU utilization nearly 10% but memory utilization increases to 66%, due to random selection frequently targets the same cluster nodes. MCSCM exhibits balancing but suffers from static ranking. MCSCM-RL demonstrates the lowest average CPU utilization that is approximately 10–11% and lowest memory utilization that is approximately 62%. The results show that MCSCM-RL consistently outperforming all other container schedulers. The real-time learning of MCSCM-RL scheduler enables it to avoid node overloading and mitigate the cumulative buildup of containerized microservices.

Overall, MCSCM-RL scheduler offers the most resource-efficient behaviour. Under cumulative deployments MCSCM-RL consistently outperforms all container scheduling strategies, delivering lowest average CPU utilization, balanced memory utilization, stable response time and lowest startup time across the cluster. The only container scheduling technique that can maintain lengthy cumulative deployment sequences without causing cluster instability is MCSCM-RL.

5.3 Comparative Analysis of Clean and Cumulative Deployment Scenarios

The combined comparison of node-level and application-level performance metrics across all five container scheduling strategies under clean and cumulative deployment conditions is presented in Figure-46. Results exhibit that Binpack scheduler and Spread scheduler gets CPU and memory inflation on worker nodes and are remarkable in response time and startup performance due to their deterministic deployment mechanism. Random scheduler avoids such deterministic bias but it does not offer meaningful optimization. The MCSCM exhibits improved decision-making under clean load; however, its reliance on fixed ranking leads to progressively unbalanced placements under cumulative deployment. As services accumulate, it selects the same high-scoring nodes, causing resource contention and increase in startup time. Hence, it limits the applicability of MCSCM in dynamic, real-world microservice environments.

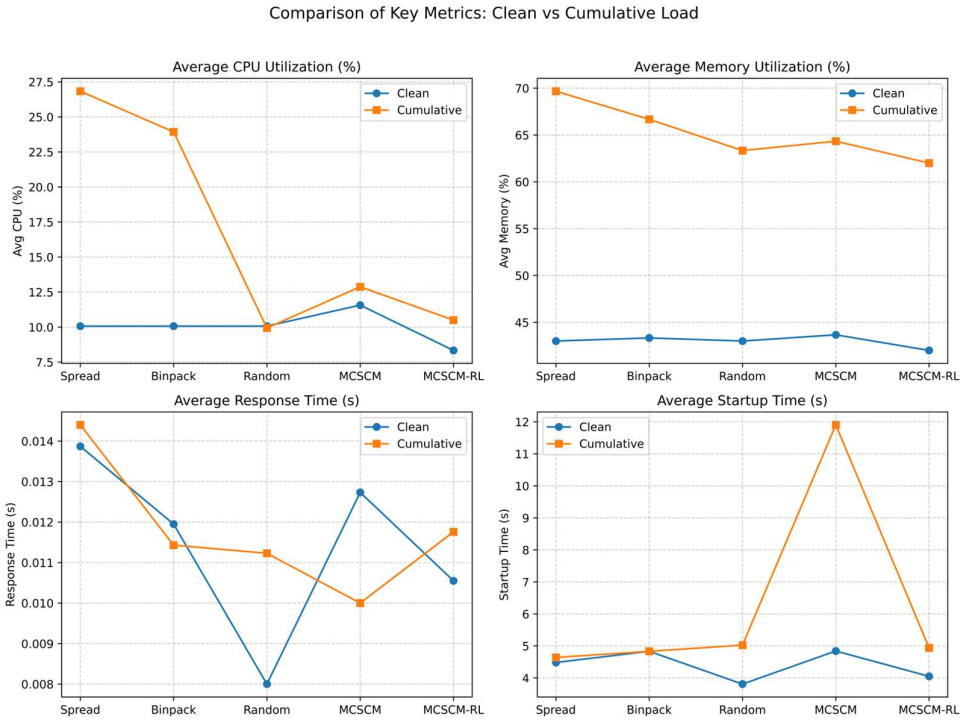


Fig. 46. Comparative Analysis of Clean and Cumulative Deployment Scenarios

In the clean deployment, Random scheduler sometimes gives better performance in startup time and response time. The reason behind this is the absence of resource contention, where at the beginning, all cluster 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. On the other hand, as the container deployment of substantially increases and the load on the cluster grows, the weakness of Random scheduler gets revealed. Random scheduler increasingly deploys containers to already congested nodes, because deployment decisions are taken without considering current resource utilization or application performance history. It leads to higher startup times and worse response times. In contrast, container 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 under dynamic load conditions. This transition from short-term effectiveness to long-term inefficiency shows the importance of adaptive container scheduling. MCSCM-RL scheduler demonstrates the consistent performance improvements through adaptive container deployment. In contrast, the proposed MCSCM-RL scheduler, consistently offers excellent scalability and robustness across all metrics. By continuously learning from cluster conditions, it maintains the lowest CPU and memory utilization and maintains stable response time under changing workload. Additionally, it achieves the better startup-time behaviour among all container scheduling strategies. Altogether, these results support MCSCM-RL as the best container scheduling approach that can maintain high quality of service (QoS), energy efficiency, and even resource utilization under both clean and cumulative deployment conditions. It proves that it is most suitable for large-scale orchestration of containerized microservices.

A more thorough statistical analysis could enhance the evaluation even more. The findings show that MCSCM-RL performs better in dynamic workload and variable executions than default Docker swarm schedulers. More thorough testing across large clusters, public cloud platforms and diverse workload scenarios would be the future development.

5.4 Discussion on Cluster Scalability

The experimental evaluation of the proposed framework was conducted on Docker Swarm cluster consisting of one manager node and two worker nodes. This experiment environment was selected to provide a controlled environment for evaluating the effectiveness of the proposed MCSCM-RL scheduler. It ensures reproducibility of the experimental results. Although the swarm cluster size is relatively small, the proposed framework is designed to be scalable to larger cloud-native environments.

As the cluster size increases to large-scale deployments containing 50–100 or more swarm nodes, the complexity of the scheduling problem will also increase. In the proposed scheduler, each swarm worker node represents a potential container deployment action. Consequently, the size of the action space grows linearly with the number of swarm worker nodes. For example, the current implementation considers three possible scheduling actions corresponding to the three swarm cluster nodes, whereas a 50-node cluster would require the DQN agent to select from 50 possible actions. A larger action space generally requires additional exploration and training iterations before the reinforcement learning agent converges to an effective scheduling decision. Traditional Q-learning approaches explicitly maintain a Q-table, but the proposed DQN approximates the action-value function using a neural network. This function approximation enables the scheduler to generalize across similar system states and avoids the exponential growth in memory requirements associated with Q-table-based reinforcement learning.

The state representation applied in this research remains independent of the cluster size because it is derived from node-level and application-level performance metrics, including CPU utilization, memory utilization, startup time, and response time. The dimensionality of the action space increases with the number of swarm worker nodes. The DQN architecture can be extended by modifying the output layer to represent additional scheduling actions without fundamentally changing the learning framework. Large clusters are expected to require long training periods, large and more diverse experience replay memories, and additional computational resources to achieve convergence. Training efficiency may also be affected by increasingly dynamic workload patterns and heterogeneous resource configurations present in large-scale cloud environment.

Overall, the proposed MCSCM-RL framework provides a scalable foundation for intelligent container scheduling. Future work will focus on evaluating the scheduler on large-scale Docker Swarm and Kubernetes clusters containing 50–100 or more worker nodes to analyse convergence time, scheduling overhead, computational complexity, and QoS performance. Furthermore, advanced reinforcement learning techniques, such as Double DQN, Dueling DQN, Proximal Policy Optimization (PPO), and hierarchical or multi-agent reinforcement learning, may be applied to improve convergence speed and scheduling efficiency in large-scale cloud environments.

Chapter 6
Conclusion and Future Work

The rapid adoption of cloud-based applications, microservices architecture, and containerization technologies have significantly increased the demand for efficient resource management and intelligent container deployment in container-based cloud environments. Container orchestration platforms such as Docker Swarm and Kubernetes have simplified the container deployment and management of containerized microservices. However, efficient container scheduling remains a crucial challenge due to dynamic workload conditions, heterogeneous resource availability, Quality of Service (QoS) needs, and increasing energy consumption concerns. Traditional container scheduling strategies such as Spread, Binpack, and Random are basically depends on static heuristic-based scheduling approaches and not able to adapt effectively to dynamic conditions. These limitations motivated the development of an intelligent scheduling framework capable of making adaptive deployment decisions based on both resource-aware and application-aware parameters.

This research proposed a Multi-Criteria Scheduling of Containerized Microservices using Reinforcement Learning (MCSCM-RL) scheduling framework for Docker Swarm environments. The proposed framework integrates multi-criteria decision-making and Deep Reinforcement Learning techniques for intelligent and adaptive scheduling of containerized microservices. An advanced scheduling framework was developed with considerations of resource performance indicators like CPU utilization, memory utilization, container count, and application performance indicators like startup time, response time. A multi-criteria node-ranking mechanism based on the relative closeness was implemented to evaluate the suitability of worker nodes for the deployment of the containerized microservice. Additionally, a Deep Q-Network based reinforcement learning agent was integrated into the scheduling framework to enable adaptive node selection based on dynamic cluster conditions.

A complete experimental environment was established using a Docker Swarm cluster consisting of one manager node and two worker nodes deployed on Ubuntu virtual machines. A CPU-Intensive microservices were developed, deployed, pushed on docker hub repository and used as workloads for evaluating performance of container scheduling mechanisms.

Resource metrics and application metrics were continuously monitored and logged using custom monitoring scripts and performance measurement modules.

The experimental results demonstrated that the proposed MCSCM-RL framework effectively improved container scheduling performance by dynamically adapting deployment decisions according to real-time resource conditions and application needs. Compared to default Docker swarm container scheduling strategies, the proposed scheduler achieved better workload distribution, improved resource utilization, minimum startup time, lower response time. It offers enhanced QoS and more balanced utilization of cluster resources. The research demonstrates that intelligent scheduling approach, which can effectively balance performance optimization and energy efficiency, which is increasingly important for modern cloud datacentres and large-scale systems. Overall, the proposed MCSCM-RL scheduling framework offers a promising step toward the realization of self-adaptive and intelligent cloud orchestration systems capable of meeting the performance, scalability, and sustainability requirements of next-generation cloud-based applications.

The current research work has been evaluated on a relatively small-scale Docker swarm cluster and focused on a specific set of resource and application-level metrics. Future research can extend the proposed framework to large-scale cloud environments and container orchestration platforms such as Kubernetes to evaluate its scalability, robustness, and adaptability under diverse workload situations. In addition, the proposed scheduling framework can be enhanced by integrating additional scheduling parameters such as network latency, bandwidth utilization, service dependencies, and workload prediction. Future work may also investigate the applicability of this scheduling framework in edge computing or multi-cloud environments. The integration of security-oriented scheduling mechanisms could further enhance the reliability of cloud-based infrastructures.

References

[1] X. Wan, X. Guan, T. Wang, G. Bai, and B. Y. Choi, (2018) "Application deployment using Microservice and Docker containers: Framework and optimization," Journal of Network and Computer Applications, vol. 119, Pages 97-109, doi: 10.1016/j.jnca.2018.07.003.

[2] L. De Lauretis, (2019), "From Monolithic Architecture to Microservices Architecture," IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), IEEE, Oct. 2019, page no: 93–96. doi: 10.1109/ISSREW.2019.00050.

[3] O. Bentaleb, A. S. Z. Belloum, A. Sebaa, and A. El-Maouhab, (2022), "Containerization technologies: taxonomies, applications and challenges," J. Supercomput., vol. 78, no. 1, page no: 1144–1181, doi: 10.1007/s11227-021-03914-1.

[4] Y. Hu, H. Zhou, C. de Laat, and Z. Zhao, (2020), "Concurrent container scheduling on heterogeneous clusters with multi-resource constraints," Future Generation Computer Systems, vol. 102, page no:. 562–573, doi: 10.1016/J.FUTURE.2019.08.025.

[5] O. Oleghe,(2021) "Container Placement and Migration in Edge Computing: Concept and Scheduling Models," IEEE Access, vol. 9, page no: 68028–68043, doi: 10.1109/ACCESS.2021.3077550.

[6] G. Fan, L. Chen, H. Yu, and W. Qi, (2020), "Multi-objective optimization of container-based microservice scheduling in edge computing," Computer Science and Information Systems, vol. 18, no. 1, page no: 23–42, Jan. 2020, doi: 10.2298/CSIS200229041F.

[7] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi, (2019), "Cloud Container Technologies: A State-of-the-Art Review," IEEE Transactions on Cloud Computing, vol. 7, no. 3, page no: 677–692, Jul. 2019, doi: 10.1109/TCC.2017.2702586.

[8] M. A. Rodriguez and R. Buyya,(2019), "Container-based cluster orchestration systems: A taxonomy and future directions," Softw. Pract. Exp., vol. 49, no. 5, page no: 698–719, May 2019, doi: 10.1002/spe.2660.

[9] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi, (2019), "Cloud Container Technologies: A State-of-the-Art Review," IEEE Transactions on Cloud Computing, vol. 7, no. 3, page no: 677–692, Jul. 2019, doi: 10.1109/TCC.2017.2702586.

[10] N. Zhou, H. Zhou, and D. Hoppe, (2023), "Containerization for High-Performance Computing Systems: Survey and Prospects," IEEE Transactions on Software Engineering, vol. 49, no. 4, page no: 2722–2740, Apr. 2023, doi: 10.1109/TSE.2022.3229221.

[11] X. Chen and S. Xiao, (2021), "Multi-Objective and Parallel Particle Swarm Optimization Algorithm for Container-Based Microservice Scheduling," Sensors, vol. 21, no. 18, page no: 6212-6220, doi: 10.3390/s21186212.

[12] E. Truyen, D. Van Landuyt, D. Preuveneers, B. Lagaisse, and W. Joosen, (2019) "A Comprehensive Feature Comparison Study of Open-Source Container Orchestration Frameworks," Applied Sciences, vol. 9, no. 5, page no: 931-942, doi: 10.3390/app9050931.

[13] P. Sharma, L. Chaufournier, P. Shenoy, and Y. C. Tay, (2016), "Containers and Virtual Machines at Scale," in Proceedings of the 17th International Middleware Conference, New York, NY, USA: ACM, Nov. 2016, page no: 1–13. doi: 10.1145/2988336.2988337.

[14] T. Salah, M. J. Zemerly, C. Y. Yeun, M. Al-Qutayri, and Y. Al-Hammadi, (2017) "Performance comparison between container-based and VM-based services," in 2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN), IEEE, Mar. 2017, page no: 185–190. doi: 10.1109/ICIN.2017.7899408.

[15] T. Lorido-Botran and M. K. Bhatti, (2021), "Adaptive Container Scheduling in Cloud Data Centers: A Deep Reinforcement Learning Approach," page no: 572–581. doi: 10.1007/978-3-030-75078-7_57.

[16] L. M. Al Qassem, T. Stouraitis, E. Damiani, and I. M. Elfadel,(2024), "Containerized Microservices: A Survey of Resource Management Frameworks," IEEE Transactions on Network and Service Management, vol. 21, no. 4, page no: 3775–3796, Aug. 2024, doi: 10.1109/TNSM.2024.3388633.

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

[18] J. W. Joab Jackson. (2022) APPLICATIONS & MICROSERVICES WITH DOCKER & CONTAINERS, 1st ed., page no: 81-90,vol. 2. CloudpulseStretegies.

[19]Chris Richardson. (2019) "Building Microservices," https://www.f5.com/company/blog/nginx/building-microservices-using-an-api-gateway.

[20] T. Menouer and P. Darmon, (2019), "New Scheduling Strategy Based on Multi-Criteria Decision Algorithm," in 2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), IEEE, Feb., page no: 101–107. doi: 10.1109/EMPDP.2019.8671594.

[21] R. Buyya, M. A. Rodriguez, A. N. Toosi, and J. Park. (2018), "Cost-Efficient Orchestration of Containers in Clouds: A Vision, Architectural Elements, and Future Directions," J. Phys. Conf. Ser., vol. 1108, page no: 012001-012011, Nov. 2018, doi: 10.1088/1742-6596/1108/1/012001.

[22] I. Ahmad, M. Gh. AlFailakawi, A. AlMutawa, and L. Alsalman. (2022), "Container scheduling techniques: A Survey and assessment," Journal of King Saud University - Computer and Information Sciences, vol. 34, no. 7, page no: 3934–3947, doi: 10.1016/j.jksuci.2021.03.002.

[23] Pan, Y., Chen, I., Brasileiro, F., Jayaputera, G., & Sinnott, R. (2019). A Performance Comparison of Cloud-Based Container Orchestration Tools. 2019 IEEE International Conference on Big Knowledge (ICBK), page no: 191–198. https://doi.org/10.1109/ICBK.2019.00033

[24] Lin, M., Xi, J., Bai, W., & Wu, J. (2021). Ant Colony Algorithm for Container-based Microservice Scheduling in Hybrid Cloud. Journal of Physics: Conference Series, 1994(1), page no: 012028. https://doi.org/10.1088/1742-6596/1994/1/012028

[25] Fard, H. M., Prodan, R., & Wolf, F. (2020). Dynamic Multi-objective Scheduling of Microservices in the Cloud. 2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC), page no: 386–393. https://doi.org/10.1109/UCC48980.2020.00061

[26] Zhao, X., & Huang, C. (2020). Microservice Based Computational Offloading Framework and Cost-Efficient Task Scheduling Algorithm in Heterogeneous Fog Cloud Network. IEEE Access, 8, page no: 56680–56694. https://doi.org/10.1109/ACCESS.2020.2981860

[27] Kaewkasi, C., & Chuenmuneewong, K. (2017). Improvement of container scheduling for Docker using Ant Colony Optimization. 2017 9th International Conference on Knowledge and Smart Technology (KST), page no: 254–259. https://doi.org/10.1109/KST.2017.788611

[28] Söylemez, M., Tekinerdogan, B., & Kolukısa Tarhan, A. (2022). Challenges and Solution Directions of Microservice Architectures: A Systematic Literature Review. Applied Sciences, 12(11), page no: 5507. https://doi.org/10.3390/app12115507

[29] Zhong, Z., Xu, M., Rodriguez, M. A., Xu, C., & Buyya, R. (2022). Machine Learning-based Orchestration of Containers: A Taxonomy and Future Directions. ACM Computing Surveys, 54(10s), page no: 1–35. https://doi.org/10.1145/3510415

[30] Muniswamy, S., & Vignesh, R. (2022). DSTS: A hybrid optimal and deep learning for dynamic scalable task scheduling on container cloud environment. Journal of Cloud Computing, 11(1), page no: 33. https://doi.org/10.1186/s13677-022-00304-7

[31] Rao, W., & Li, H. (2025). Energy-aware scheduling algorithm for microservices in Kubernetes clouds. Journal of Grid Computing, 23(2), Article 2. https://doi.org/10.1007/s10723-024-09788

[32] Liu, R., Lv, H., Yang, P., & Bie, R. (2024). A multi-task genetic programming approach for online multi-objective container placement in heterogeneous cluster. Complex & Intelligent Systems. Advance online publication. page no: 1-20,https://doi.org/10.1007/s40747-024-01605

[33] Nkenyereye, L., & Lee, S. (2025). Functionality-aware offloading technique for scheduling container applications in edge infrastructures. Journal of Cloud Computing: Advances, Systems and Applications, 14, page no:1-28, Article 28. https://doi.org/10.1186/s13677-025-00737

[34] Moreschini, S., Pour, S., Lanese, I., Balouek, D., Bogner, J., Li, X., & Pecorelli, F. (2025). AI techniques in the microservices life-cycle: A systematic mapping study. Computing, 107, page no:1235–1268. https://doi.org/10.1007/s00607-025-01432

[35] Al Qassem, L. M., & others. (2025). Predictive container orchestration in the cloud using artificial intelligence techniques. Computing. HTTPs://doi.org/10.1007/s00607-025-01505

[36] Zhang, Y., & others. (2024). Pattern learning for scheduling microservice workflow to cloud containers. International Journal of Machine Learning and Cybernetics. page no:3701-3714 https://doi.org/10.1007/s13042-024-02115-5

[37] Shi, K., Huang, Y., & Xu, H. (2025). Resource-Constrained Scheduling in Containerized Edge Computing Using Graph Transformer-Enhanced DQN,page no: 210–220. https://doi.org/10.1007/978-981-95-0006-2_18

[38] J. Wang et al. (2025), "Adaptive resource allocation for cloud-native microservice via meta-learning and hyper-heuristic algorithms," Journal of Grid Computing, doi: 10.1007/s10723-025-09814-5.

[39] Sanjalawe, Y., Al-E'mari, S., Fraihat, S., & Makhadmeh, S. (2025). AI-driven job scheduling in cloud computing: a comprehensive review. Artificial Intelligence Review, 58(7), page no:197. https://doi.org/10.1007/s10462-025-11208-8

[40] Kallel, A., Rekik, M., & Khemakhem, M. (2025). A deep reinforcement learning-based optimization approach for containerized microservice scheduling in Hybrid Fog/Cloud environments. Engineering Applications of Artificial Intelligence, page no:141, 109745. https://doi.org/10.1016/j.engappai.2024.109745

[41] Turin, G., Borgarelli, A., Donetti, S., Damiani, F., Johnsen, E. B., & Tapia Tarifa, S. L. (2023). Predicting resource consumption of Kubernetes container systems using resource models. Journal of Systems and Software, page no:203, 111750. https://doi.org/10.1016/j.jss.2023.111750

[42] Marchese, A., & Tomarchio, O. (2025). Enhancing the Kubernetes platform with a load-aware orchestration strategy. SN Computer Science, 6, Article 224. https://doi.org/10.1007/s42979-025-03712

[43] Bai, X., Islam, M. T., Buyya, R., & Toosi, A. N. (2025). REACH: Reinforcement Learning for Adaptive Microservice Rescheduling in the Cloud-Edge Continuum. arXiv:2510.06675, 2025

[44] Cai, J., Zeng, H., Liu, F., & Chen, J. (2025). Intelligent Dynamic Multi-Dimensional Heterogeneous Resource Scheduling Optimization Strategy Based on Kubernetes. Mathematics, 13(8), page no:1342. https://doi.org/10.3390/math13081342

[45] Jian, Z., Xie, X., Fang, Y., Jiang, Y., Lu, Y., Dash, A., Li, T., & Wang, G. (2024). DRS: A deep reinforcement learning enhanced Kubernetes scheduler for microservice-based system. Software: Practice and Experience, 54(10), page no:2102–2126. https://doi.org/10.1002/spe.3284

[46] Senjab, K., Abbas, S., Ahmed, N., & Khan, A. ur R. (2023). A survey of Kubernetes scheduling algorithms. Journal of Cloud Computing, 12(1), page no :87. https://doi.org/10.1186/s13677-023-00471-1

[47] Santos, J., Zaccarini, M., Poltronieri, F., Tortonesi, M., Stefanelli, C., di Cicco, N., & de Turck, F. (2025). HephaestusForge: Optimal microservice deployment across the Compute Continuum via Reinforcement Learning. Future Generation Computer Systems, page no:166, 107680. https://doi.org/10.1016/j.future.2024.107680

[48] Lorido-Botran, T., & Bhatti, M. K. (2021). Adaptive Container Scheduling in Cloud Data Centers: A Deep Reinforcement Learning Approach (page no: 572–581). https://doi.org/10.1007/978-3-030-75078-7_57

[49] Sutton, R. S., & Barto, A. G. (2018). Reinforcement learning: An introduction (2nd ed.). MIT Press. https://incompleteideas.net/book/the-book-2nd.html

[50] Auer, P., Cesa-Bianchi, N., & Fischer, P. (2002). Finite-time Analysis of the Multiarmed Bandit Problem. Machine Learning, 47(2–3), page no: 235–256. https://doi.org/10.1023/A:1013689704352

[51] Osband, I., Russo, D., & Van Roy, B. (2013). Efficient reinforcement learning via posterior sampling. In Advances in Neural Information Processing Systems 26. https://proceedings.neurips.cc/paper_files/paper/2013/file/eb160de1de89d9058fcb0b968dbbbd68-Paper.pdf

[52] J. Xu, J. A. B. Fortes, and T. Carpenter. (2012). On the use of random scheduling for load balancing in distributed systems. IEEE Transactions on Parallel and Distributed Systems, 23(12), page no:c4. https://doi.org/10.1109/TPDS.2012.296

[53] Rejiba, Z., and J. Chamanara, 2022, "Custom Scheduling in Kubernetes: A Survey on Common Problems and Solution Approaches," *ACM Computing Surveys* 55(7), page no:1–37. https://doi.org/10.1145/3544788.

[54] Thota, S., 2018, "Docker and Google Kubernetes," *International Journal of Advanced Research* 6(7), page no:984–998. DOI: 10.21474/IJAR01/7449.

[55] Alam, M., Rufino, J., Ferreira, J., Ahmed, S. H., Shah, N. and Chen, Y. (2018). Orches tration of Microservices for IoT Using Docker and Edge Computing, IEEE Commu nications Magazine 56(9): page no:118–123.

[56] Medel, V., Tolosana-Calasanz, R., Bañares, J. Á., Arronategui, U. and Rana, O. F. (2018). Characterising resource management performance in Kubernetes, Computers and Electrical Engineering 68(May 2017): page no:286–297.https://doi.org/10.1016/j.compeleceng.2018.03.041

[57] Li, Z., M. Kihl, Q. Lu, and J. A. Andersson, 2017, "Performance Overhead Comparison Between Hypervisor and Container Based Virtualization," in *Proceedings of the IEEE International Conference on Advanced Information Networking and Applications (AINA)*, page no: 955–962.

[58] Acevedo, C. A. J., J. P. G. Jorge, and I. R. Patiño, 2017, "Methodology to Transform a Monolithic Software into a Microservice Architecture," in *Proceedings of the 2017 IEEE 6th International Conference on Software Process Improvement (CIMPS)*, Zacatecas, Mexico, page no: 1–6.

[59] M. Bestari, A. Kistijantoro, and A. Sasmita. 2020. Dynamic Resource Scheduler for Distributed Deep Learning Training in Kubernetes. In 2020 7th Int. Conf. on Advance Informatics: Concepts, Theory and Applications (ICAICTA).

[60] E. Casalicchio and S. Iannucci. 2020. The state-of-the-art in container technologies: Application, orchestration and security. Concurrency and Computation: Practice and Experience 32, 17 (2020), page no:56-68.

[61] Islam, M. T., S. Karunasekera, and R. Buyya, 2022, "Performance and Cost-Efficient Spark Job Scheduling Based on Deep Reinforcement Learning in Cloud Computing Environments," *IEEE Transactions on Parallel and Distributed Systems* 33(7), page no:1695–1710. https://doi.org/10.1109/TPDS.2021.3124670

[62] Fu, K., W. Zhang, Q. Chen, D. Zeng, and M. Guo, 2021, "Adaptive Resource Efficient Microservice Deployment in Cloud-Edge Continuum," *IEEE Transactions on Parallel and Distributed Systems* 33(8), page no:1825–1840. https://doi.org/10.1109/TPDS.2021.3131996

[63] Baarzi, A. F., and G. Kesidis, 2021, "Showar: Right-Sizing and Efficient Scheduling of Microservices," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC 2021)*, page no:427–441

[64] Menouer T, Manad O, Cérin C, Darmon P (2019) Power efficiency containers scheduling approach based on machine learning technique for cloud computing environment. In: Esposito C, Hong J, Choo KKR(eds) Pervasive systems. Algorithms and networks. Springer, Cham, page no:193–206

[65] Al-Dhuraibi, Y., F. Paraiso, N. Djarallah, and P. Merle, 2021, "Reinforcement Learning-Based Application Autoscaling in the Cloud: A Survey," *ACM Computing Surveys* 54(11s), Article 236, page no:1–35

[66] Joseph, C.T.; Martin, J.P.; Chandrasekaran, K.; Kandasamy, A. Fuzzy Reinforcement Learning based Microservice Allocation in Cloud Computing Environments. In Proceedings of the TENCON 2019-2019 IEEE Region 10 Conference (TENCON), Kochi, India, 17–20 October 2019; page no:1559–1563.

[67] Choi, Y., J. Yi, and S. Hong, 2021, "A Scheduling Scheme in the Cloud Computing Environment Using Deep Q-Learning," *IEEE Access* 9, page no:4275–4284.

[68] Mao, H., M. Alizadeh, I. Menache, and S. Kandula, 2016, "Resource Management with Deep Reinforcement Learning," in *Proceedings of the 15th ACM Workshop on Hot Topics in Networks (HotNets '16)*, Atlanta, GA, USA, page no:50–56. https://doi.org/10.1145/3005745.3005750

[69] Chen, Z., 2022, "RIFLING: A Reinforcement Learning-Based GPU Scheduler for Deep Learning Research and Development Platforms," *Software: Practice and Experience* 52(6), page no:1319–1336, https://doi.org/10.1002/spe.3066

[70] Huang, J., C. Xiao, and W. Wu, 2020, "RLSK: A Job Scheduler for Federated Kubernetes Clusters Based on Reinforcement Learning," in *Proceedings of the 2020 IEEE International Conference on Cloud Engineering (IC2E)*, page no:116–123, 10.1109/IC2E48712.2020.00019

[71] Casalicchio, E., and S. Iannucci, 2020, "The State-of-the-Art in Container Technologies: Application, Orchestration and Security," *Concurrency and Computation: Practice and Experience* 32(17), page no:56-68. https://doi.org/10.1002/cpe.5668

[72] Li, C., Bai, J., Ge, Y., & Luo, Y. (2020). Heterogeneity-aware elastic provisioning in cloud-assisted edge computing systems. *Future Generation Computer Systems*, *112*, page no:1106–1121. https://doi.org/10.1016/j.future.2020.06.022

[73] Chung, Andrew & Park, Jun & Ganger, Gregory. (2018). Stratus: cost-aware container scheduling in the public cloud. page no:121-134. 10.1145/3267809.3267819.

[74] Mugeraya, S., and K. Devadkar, 2022, "Dynamic Task Scheduling and Resource Allocation for Microservices in Cloud," *Journal of Physics: Conference Series* 2325(1), 012052.page no:1-11 https://doi.org/10.1088/1742-6596/2325/1/012052

[75] Joseph, C. T., and K. Chandrasekaran, 2020, "Dynamic Interaction Aware Resource Allocation for Microservices in Cloud Computing," *Future Generation Computer Systems* 107, Volume:111,ISSN 1383-7621, https://doi.org/10.1016/j.sysarc.2020.101785

[76] Xu, M., and R. Buyya, 2019, "BrownoutCon: A Software System Based on Brownout and Containers for Energy-Efficient Cloud Computing," *Journal of Systems and Software* 155, page no:91–103. https://doi.org/10.1016/j.jss.2019.05.031

[77] Hassan, M. F., Akbar, R., Shah, S. N. M., Hassan, F., Magsi, S. A., & Siddiqui, M. A. (2022). Containerized Microservices Orchestration and Provisioning in Cloud Computing: A Conceptual Framework and Future Perspectives. *Applied Sciences*, *12*(12), page no:5793. https://doi.org/10.3390/app12125793

[78] Malik, S., Tahir, M., Sardaraz, M., & Alourani, A. (2022). A Resource Utilization Prediction Model for Cloud Data Centers Using Evolutionary Algorithms and Machine Learning Techniques. *Applied Sciences*, *12*(4), 2160. https://doi.org/10.3390/app12042160

[79] Prajapati, A., & Patel, M. M. (2024). Container Scheduling: A Taxonomy, Open Issues and Future Directions for Scheduling of Containerized Microservices in Cloud Environments. International Journal of Intelligent Systems and Applications in Engineering, 12(22s), page no:1108–1122.

[80] G. H. M. Matos, M. Carvalho and D. F. Macedo, "Container-Based Microservice Scheduling Using Reinforcement Learning in Distributed Cloud Computing," 2024 IEEE Latin-American Conference on Communications (LATINCOM), Medellin, Colombia, 2024, page no: 1-6, doi: 10.1109/LATINCOM62985.2024.10770680.

[81] O. Baker, W. Li, H. Ma, K. Subaramaniam, D. Lopez and S. Palaniappan, "Machine Learning-Driven Container Scheduling for Edge-Empowered Microservices," *2025 IEEE International Conference on Robotics and Technologies for Industrial Automation (ROBOTHIA)*, Kuala Lumpur, Malaysia, 2025, page no: 1-6, doi: 10.1109/ROBOTHIA63806.2025.10986316.

[82] Grzesik, P., & Mrozek, D. (2024). Combining Machine Learning and Edge Computing: Opportunities, Challenges, Platforms, Frameworks, and Use Cases. *Electronics*, *13*(3), page no:640. https://doi.org/10.3390/electronics13030640

[83] Pratham Jangra, Anuttam Anand, Dr. Amit Kumar Tyagi. "Survey On Container Systems And Their Efficient Orchestration Algorithms".International Journal Of Creative Research Thoughts Issn:2320-2882, Vol.9, Issue 5, page no:.i53-i61,https://www.ijcrt.org/papers/IJCRT2105852.pdf

[84] Rao, V. et al. (2021). Scheduling Microservice Containers on Large Core Machines Through Placement and Coalescing. In: Klusáček, D., Cirne, W., Rodrigo, G.P. (eds) Job Scheduling Strategies for Parallel Processing. JSSPP 2021. page no:560-572,https://doi.org/10.1007/978-3-030-88224-2_5

List of Publications

1. Anil Prajapati, Dr. Manish M Patel, Container Scheduling: A Taxonomy, Open Issues and Future Directions for Scheduling of Containerized Microservices in Cloud Environments, International Journal of INTELLIGENT SYSTEMS AND APPLICATIONS IN ENGINEERING, ISSN:2147-6799214, 2024, 12(22s), 1108–1122, DOI: 10.18201/ijisae.20240663439

2. Anil A. Prajapati, Manish M. Patel, Multi-Criteria Deep Reinforcement Learning Energy-Aware and QoS-Driven Scheduling for Containerized Microservices in Docker Swarm Clusters, Journal of Advances in Engineering and Intelligence Systems e-ISSN:2821-0263 Volume 05, Issue 01, 2026, DOI:https://doi.org/10.22034/aeis.2026.568553.1405