Optimized Resource Allocation for Docker Container Based Cloud Environments using Hybrid Adaptive PSO and Decision Tree Model

A Thesis submitted to Gujarat Technological University

for the Award of

Doctor of Philosophy

in

Computer/IT Engineering

by

Manmitsinh Chandrasinh Zala
Enrollment No.: 199999913537

under supervision of

Dr. Jaykumar S. Patel

GUJARAT TECHNOLOGICAL UNIVERSITY
AHMEDABAD

July 2026

Declaration

I declare that the thesis entitled **"Optimized Resource Allocation for Docker Container Based Cloud Environments using Hybrid Adaptive PSO and Decision Tree Model"** submitted by me for the degree of Doctor of Philosophy is the record of research work carried out by me during the period from **Jan 2020** to **April 2026** under the supervision of **Prof. (Dr.) Jaykumar Shantilal Patel**, Professor, Hillwoods Academy of Higher Education (HAHE) and this has not formed the basis for the award of any degree, diploma, associateship, fellowship, titles in this or any other University or other institution of higher learning.

I further declare that the material obtained from other sources has been duly acknowledged in the thesis. I shall be solely responsible for any plagiarism or other irregularities, if noticed in the thesis.

Signature of the Research Scholar: MZala 24/07/26 Date: 24/07/2026

Name of Research Scholar: **Manmitsinh Chandrasinh Zala**

Place: Ahmedabad.

CERTIFICATE

I certify that the work incorporated in the thesis **"OPTIMIZED RESOURCE ALLOCATION FOR DOCKER CONTAINER BASED CLOUD ENVIRONMENTS USING HYBRID ADAPTIVE PSO AND DECISION TREE MODEL"** submitted by **Mr. Manmitsinh Chandrasinh Zala** was carried out by the candidate under my 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, Associateship, Fellowship or other similar titles. (ii) the thesis submitted is a record of original research work done by 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 Research Supervisor: 24/07/2026 Date: 24 / 07 / 2026

Name of Research Supervisor: **Dr. Jaykumar Shantilal Patel**

Place: Ahmedabad.

Course-work Completion Certificate

This is to certify that **Mr. Manmitsinh Chandrasinh Zala** enrolment no. **199999913537** is a PhD scholar enrolled for PhD program in the branch **Computer / IT Engineering** of Gujarat Technological University, Ahmedabad.

(Please tick the relevant option(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)
BC	**AB**

Signature of Research Supervisor: 24/07/2026

Name of Research Supervisor: **Dr. Jaykumar Shantilal Patel**

Originality Report Certificate

It is certified that PhD Thesis titled **"Optimized Resource Allocation for Docker Container Based Cloud Environments using Hybrid Adaptive PSO and Decision Tree Model"** by **Manmitsinh Chandrasinh Zala** has been examined by us. We undertake the following:

a. Thesis has significant new work / knowledge as compared already published or are 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 ≤ 10%).

Signature of the Research Scholar: MZalZ 24/07/2026 Date: 24 / 07/ 2026

Name of Research Scholar: **Manmitsinh Chandrasinh Zala**

Place: Ahmedabad.

Signature of Research Supervisor: 24/07/2026 Date: 24 / 07 / 2026

Name of Research Supervisor: **Dr. Jaykumar Shantilal Patel**

Place: Ahmedabad.

Manmitsinh Zala

199999913537_ManmitsinhZala_Thesis Updated.pdf

Ph D Thesis

Manmitsinh C Zala

Gujarat Technological University

Document Details

Submission ID

trn:oid:::1:3580233115

Submission Date

May 27, 2026, 4:29 PM GMT+5:30

Download Date

May 30, 2026, 10:43 PM GMT+5:30

File Name

199999913537_ManmitsinhZala_Thesis_Updated.pdf

File Size

3.5 MB

126 Pages

23,232 Words

140,779 Characters

7% Overall Similarity

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

Filtered from the Report

- Small Matches (less than 10 words)

Exclusions

- 3 Excluded Sources

Match Groups

87 Not Cited or Quoted 7%
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

4% Internet sources
3% Publications
6% Submitted works (Student Papers)

PhD THESIS Non-Exclusive License to GUJARAT TECHNOLOGICAL UNIVERSITY

In consideration of being a PhD Research Scholar at Gujarat Technological University and in the interests of the facilitation of research at GTU and elsewhere, I, **Manmitsinh Chandrasinh Zala** having Enrollment no. **199999913537**, 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 any of 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: **Manmitsinh Chandrasinh Zala**

Place: Ahmedabad. Date: 24/07/2026

Recommendation of the Research Supervisor:

Signature of Research Supervisor: 24/07/2026

Name of Research Supervisor: **Dr. Jaykumar Shantilal Patel**

Thesis Approval Form

The viva-voce of the PhD Thesis submitted by **Mr. Manmitsinh Chandrasinh Zala** (Enrollment No.:**199999913537**) entitled **"Optimized Resource Allocation for Docker Container Based Cloud Environments using Hybrid Adaptive PSO and Decision Tree Model"** was conducted on Friday,24-July-2026 ,(day and date) at Gujarat Technological University.

(Please tick any one of the following option)

☑ 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.

24-07-2026

Name and Signature
of Supervisor with Seal
Dr. Jaykumar Shantilal Patel

(External Examiner 1)
Name and Signature
Dr.Nishant Doshi

(External Examiner 2)
Name and Signature
Dr.Shruti Oza

Abstract

In Cloud computing containerized applications are light weight virtualization where good performance, energy efficiency and cost depend mainly on how workloads are distributed and balanced across various swarm nodes, Docker swarm is lightweight design for virtualization, it provides static scheduling methods, which lacks real time dynamics often leading to imbalanced CPU usage, This work introduces a hybrid intelligent load-balancing framework that combines a Decision Tree (DT)–based cost model with an Adaptive Particle Swarm Optimization (APSO) algorithm for context-aware, dynamic container placement. The DT component continuously classifies the states of the nodes using live telemetry, including CPU utilization, memory pressure, and task density, while the APSO.adjusts inertia and velocity parameters dynamically to refine the search process and mitigate premature convergence, thereby it improves performance over conventional PSO.

The framework is implemented on a 1-master and 2-worker nodes Docker Swarm cluster running on Ubuntu VirtualBox, instrumented with Prometheus and cAdvisor for detailed monitoring and Grafana for real-time visualization. A mixed workload comprising CPU-intensive factorial computations and controlled memory operations is executed under three strategies: the native Swarm scheduler, a classical PSO-based scheduler, and the proposed DT+APSO model. Analysis of CPU utilization, memory footprint, variance, and 95% confidence intervals shows that DT+APSO lowers overall CPU utilization by 18–20% compared to the default Swarm scheduler and 10-12% relative to standard PSO, while keeping memory usage stable and reducing variance at the node level. These gains translate into energy savings through reduced dynamic CPU power consumption. The modular design supports seamless integration into existing Swarm deployments and provides a scalable foundation for future multi-objective optimization involving latency, energy, and cost trade-offs, which demonstrates the practical benefits of machine-learning-guided metaheuristic scheduling for container orchestration.

Keywords: Load Balancing, Resource Allocation, Heuristic Approach, Docker Container, Cloud Computing, Adaptive PSO, cAdvisor, Decision Tree, Docker Swarm, Grafana, Prometheus.

Acknowledgement

With a heart full of gratitude and humility, I bow before the divine blessings of Goddess Shakti and Lord Krishna, whose grace, strength, and spiritual guidance have remained my constant source of courage throughout this PhD journey. Their blessings gave me patience during difficult times, clarity during confusion, and inner strength to continue moving forward with dedication and faith.

I express my deepest respect and heartfelt gratitude to my beloved mother, Smt. Kanakba Zala, whose unconditional love, blessings, sacrifices, and constant encouragement have always been the foundation of my life. I also pay my humble tribute to my respected father, Late Shri Chandrasinh Zala, whose values, discipline, and blessings continue to inspire me in every step of my personal and academic journey. Whatever I have achieved today is deeply rooted in the "sanskar", guidance, and blessings of my parents.

I am profoundly thankful to my wife, Smt. Hemangiba Zala, for her endless patience, emotional support, understanding, and cooperation during the long and demanding period of my research work. Her silent sacrifices and constant encouragement made this journey smoother and meaningful. I also express my loving gratitude to my dear children, Paridhiba and Yugvirsinh, whose innocent love, smiles, and presence gave me renewed energy during challenging phases of this work. I am also sincerely thankful to my brother, Shri Anirddhasinh Zala, for his support, encouragement, and affection throughout this journey.

I feel highly privileged to express my sincere and respectful gratitude to my research guide, Dr. Jaykumar S. Patel, for his valuable guidance, scholarly supervision, continuous motivation, and constructive suggestions throughout the course of this doctoral research. His academic insight, patience, and timely support have played a significant role in shaping this thesis in a systematic and meaningful manner.

I extend my sincere thanks to my Doctoral Progress Committee members, Dr. Harshad B. Bhadka and Dr. Shivani Trivedi, for their valuable observations, academic suggestions, and critical feedback during various stages of my research work. Their guidance helped me improve the quality, direction, and depth of this thesis.

I am also thankful to Dr. Om Mehta, Dr. Vinod Thummar, Dr. Khushali Raval, Prof. V. G. Patel, Dr. Rashmi Bhatad, Prof. Anil Prajapati, Prof. Pradip Gamit, Prof. Nikunj Gamit, Dr. Jashvant Dave, Dr. Jignesh Vaniya, Dr. Dipak Patel, and Prof. Naimisha Trivedi for their encouragement, academic support, cooperation, and valuable guidance

at various stages of this work. Their words of motivation and professional support have contributed positively to my PhD journey.

I express my sincere gratitude to Dr. Vibha D. Patel, Head of the Department, Information Technology, Vishwakarma Government Engineering College, for her support, encouragement, and cooperation during my research work. I also convey my respectful thanks to Dr. Vinay S. Purani, Principal, Vishwakarma Government Engineering College, for providing a supportive academic environment and constant encouragement for research and professional growth.

I am truly thankful to my friends Shri Priyank Desai, Shri Rajesh Desai, and Shri Deepak Gohel for their moral support, positive words, and constant encouragement during this long academic journey. Their friendship and support helped me remain confident and motivated during difficult phases.

Finally, I express my sincere thanks to all my teachers, colleagues, friends, well-wishers and everyone who directly or indirectly supported me in pursuing and completing this PhD work. Though it may not be possible to mention every name individually, I gratefully acknowledge their blessings, cooperation, and encouragement.

This thesis is not only the outcome of academic effort, but also the result of divine blessings, family sacrifices, guidance of mentors, support of colleagues, and the goodwill of many people who stood with me throughout this journey.

With heartfelt gratitude,
Manmitsinh Chandrasinh Zala
Research Scholar
Gujarat Technological University

Contents

List of Abbreviations

APSO	Adaptive Particle Swarm Optimization
PSO	Particle Swarm Optimization
DT	Decision Tree
DT+APSO	Decision Tree and Adaptive Particle Swarm Optimization
CPU	Central Processing Unit
RAM	Random Access Memory
VM	Virtual Machine
QoS	Quality of Service
SLA	Service Level Agreement
API	Application Programming Interface
ML	Machine Learning
AI	Artificial Intelligence
IoT	Internet of Things
GA	Genetic Algorithm
ACO	Ant Colony Optimization
ABC	Artificial Bee Colony
LB	Load Balancing
RC	Resource Consumption
RA	Resource Allocation
RT	Response Time
ET	Execution Time
TH	Throughput
LA	Load Average
IT	Information Technology
cAdvisor	Container Advisor
CSV	Comma-Separated Values
JSON	JavaScript Object Notation
HTTP	Hypertext Transfer Protocol
REST	Representational State Transfer
YAML	YAML Ain't Markup Language
CLI	Command Line Interface
GUI	Graphical User Interface
SDK	Software Development Kit
MSE	Mean Squared Error
RMSE	Root Mean Squared Error

MAE	Mean Absolute Error
SD	Standard Deviation
Avg.	Average
Max.	Maximum
Min.	Minimum
No.	Number
Req.	Request
Util.	Utilization
Fig.	Figure
Eq.	Equation
Tbl.	Table

List of Figures

List of Tables

CHAPTER 1

Introduction

1.1 Background and Evolution of Cloud Computing

Cloud computing Figure 1.1 has become one of the most important technological foundations of modern digital infrastructure. It has changed the way computing resources are created, accessed, managed , and consumed by individuals, academic institutions, industries, government organizations, and enterprise systems [6]. In traditional computing environments, organizations were required to purchase and maintain physical servers, storage devices, networking equipment, operating systems, database servers, application platforms, and security infrastructure before deploying any digital service. Such infrastructure required large capital investment, continuous maintenance, technical manpower, and periodic hardware upgrades. As user demand increased, the limitations of traditional data centers became more visible, particularly in terms of scalability, resource utilization, cost, and flexibility.

Cloud computing introduced a more flexible and service-oriented model in which computing resources can be provisioned, monitored, scaled, and released according to demand. Instead of permanently owning all hardware and software infrastructure, users can consume computing services through a network-based model. This makes cloud computing suitable for applications where workloads are dynamic and difficult to predict. For example, educational portals, online examination systems, banking applications, healthcare systems, e-commerce platforms, transportation services, and government portals may experience sudden spikes in user demand. A traditional infrastructure model may either remain underutilized during normal periods or become

insufficient during peak periods. Cloud computing addresses this issue by supporting elasticity, shared resource pools, and measured usage.

From centralized computing to distributed cloud infrastructure: The evolution of cloud computing can be understood as a gradual movement from centralized computing toward distributed, virtualized, and service-oriented infrastructure. In early computing environments, applications were usually hosted on dedicated servers. Each application had its own hardware, operating system, and software stack. This model was simple but inefficient because server resources were often underutilized. Later, distributed computing and grid computing introduced resource sharing across multiple systems. Utility computing introduced the idea of paying for computing resources according to usage [7]. Virtualization further improved resource utilization by allowing multiple virtual machines to run on the same physical hardware. Cloud computing combined these ideas and offered them as scalable services through the Internet.

Role of virtualization in cloud growth: Virtualization became a key enabler of cloud computing because it allowed physical resources to be abstracted into multiple logical environments. A single physical server could host several virtual machines, each running a separate operating system and application stack. This improved resource utilization and reduced the need for dedicated hardware. However, virtual machines also introduced overhead because each VM required its own guest operating system. As cloud workloads became more dynamic and microservice-oriented, the need for lighter and faster deployment mechanisms increased. This need created the foundation for containerization.

Emergence of cloud-native applications: Modern cloud applications are no longer limited to monolithic software hosted on a single machine. They are increasingly distributed, modular, service-oriented, and data-driven. Many applications are now developed using microservice architecture, where each functional unit is deployed as an independent service. Such applications require continuous deployment, rapid scaling, fault tolerance, and efficient load balancing. Cloud-native applications therefore depend heavily on automation, orchestration, monitoring, and intelligent scheduling. This makes resource allocation a critical research problem[8].

Relevance to the present thesis: The present thesis, titled *"Optimized Resource Allocation for Docker Container-Based Cloud Environments Using Hybrid Adaptive PSO and Decision Tree"*, focuses on the problem of intelligent resource allocation in containerized cloud environments. The thesis is positioned at the intersection of cloud computing, container-based virtualization, Docker Swarm orchestration, resource monitoring, load balancing, Decision Tree-based node suitability analysis, and Adaptive Particle Swarm Optimization. The central idea is to improve the placement of Docker containers across cloud nodes so that CPU and memory resources are used more efficiently and load imbalance is reduced.

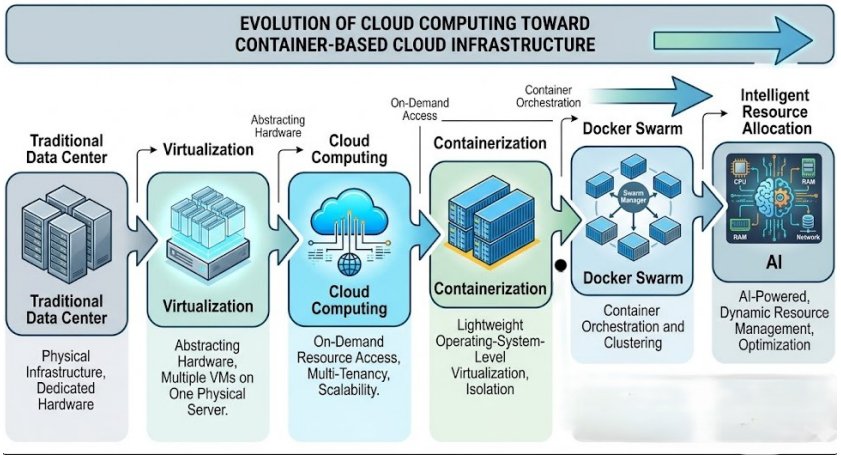


FIGURE 1.1: Evolution of Cloud Computing Toward Container-Based Cloud Infrastructure

1.2 Cloud Computing Models, Characteristics, and Resource Management Requirements

Cloud computing is usually explained through its essential characteristics, service models, and deployment models. These models help define how cloud resources are provided, managed, and consumed. They also provide the foundation for understanding why resource allocation and load balancing are important in cloud-based environments.

Essential characteristics of cloud computing: Cloud computing has five key characteristics: on-demand self-service, broad network access, resource pooling, rapid

elasticity, and measured service. It allows users to access computing resources whenever needed, through standard networks and different devices. Providers share physical and virtual resources among multiple users, scale them quickly according to demand, and monitor usage for control, reporting, and billing.

These characteristics are directly related to the resource allocation problem studied in this thesis. In a cloud environment, resources must be dynamically assigned to workloads. If resources are allocated inefficiently, the benefits of elasticity and resource pooling are reduced. For example, if one node in a cluster becomes overloaded while another remains underutilized, the system fails to use its shared resource pool effectively. Therefore, resource allocation and load balancing are necessary for achieving the practical benefits of cloud computing.

Cloud service models: Cloud services are mainly classified into three models: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). IaaS offers virtual machines, storage, and networking resources. PaaS provides a platform for developing and deploying applications without managing the underlying infrastructure. SaaS delivers ready-to-use software applications to users over the Internet [9].

Containerized environments can be used across these service models. At the IaaS level, Docker Swarm or Kubernetes clusters can be deployed on virtual machines or physical servers. At the PaaS level, containers provide a standardized environment for building and deploying applications. At the SaaS level, user-facing applications may be implemented as microservices running inside containers. Therefore, container scheduling and resource allocation are relevant across multiple cloud service layers.

Cloud deployment models: Cloud deployment models include public cloud, private cloud, community cloud, and hybrid cloud. Public clouds are operated by cloud service providers and made available to multiple users. Private clouds are dedicated to a single organization and provide more control over resources, policies, and data. Community clouds are shared by organizations with common requirements. Hybrid clouds combine two or more cloud deployment models and allow data and application portability across them [10, 11].

The experimental environment used in the present thesis can be considered a controlled private cloud-like environment because the Docker Swarm cluster is configured within a limited infrastructure consisting of one master node and worker nodes. However, the proposed resource allocation model can be conceptually extended to larger private, public, or hybrid containerized environments.

TABLE 1.1: Comparison of Cloud Deployment Models

[3, 6, 12–14]

Deployment Model	**Ownership and Access**	**Major Advantage**	**Major Limitation**	**Relevance to This Thesis**
Public Cloud	Owned by cloud provider and accessed over the Internet	High scalability and lower upfront cost	Less direct control over infrastructure	Docker clusters can be deployed on public cloud VMs
Private Cloud	Dedicated to a single organization	More control, customization, and security	Higher management cost	Similar to controlled experimental Docker Swarm setup
Community Cloud	Shared by organizations with similar needs	Shared cost and common compliance	Governance complexity	Useful for academic or government cloud sharing
Hybrid Cloud	Combination of two or more cloud models	Balance between scalability and control	Integration complexity	Future extension for multi-cloud container scheduling

Need for resource management: The effectiveness of cloud computing depends on how efficiently resources are allocated to workloads. Resource management includes

provisioning, scheduling, monitoring, scaling, migration, and optimization. In containerized environments, resource management becomes more dynamic because containers are lightweight, can be created quickly, and can be scaled frequently. A cloud system must continuously monitor CPU usage, memory consumption, task density, network traffic, and application load. Based on this information, the system should allocate or reallocate containers to maintain balanced utilization.

Resource allocation as a research problem: Resource allocation in cloud computing is not a simple assignment problem. It involves multiple constraints and objectives, including performance, resource utilization, response time, energy consumption, cost, reliability, and service-level agreement requirements. In Docker Swarm environments, the problem becomes: which node should receive a new container or service replica so that the overall cluster remains balanced? This question forms the foundation of the present research.

1.3 Virtualization and the Transition from Virtual Machines to Containers

Virtualization is one of the major technological pillars of cloud computing Table1.2. It allows multiple isolated environments to run on the same physical hardware. The traditional form of virtualization is virtual machine-based virtualization, where a hypervisor creates and manages multiple virtual machines on a physical server. Each virtual machine has its own guest operating system, virtual CPU, virtual memory, virtual disk, and virtual network interface. This architecture provides strong isolation and flexibility, but also consumes significant resources as shown in Table 1.3 and Figure 1.3.

Virtual machine-based virtualization: Virtual machines are useful when applications require strong isolation or different operating systems as explained in Figure 1.3. For example, one virtual machine may run Linux while another runs Windows on the same physical server. This makes VM-based virtualization suitable for heterogeneous

enterprise environments. However, the requirement of a complete guest operating system for each VM increases overhead. VM startup time is also comparatively higher because the full operating system must boot. Storage usage is larger because each VM image contains an operating system and application stack. These limitations become important when applications need rapid scaling.

Container-based virtualization: Containerization provides a lighter alternative to virtual machines. Containers share the kernel of the host operating system and run as isolated processes. A container packages the application code, dependencies, libraries, and runtime environment required to execute the application. Since containers do not require a separate guest operating system, they start faster and consume fewer resources. This makes them suitable for microservices, DevOps pipelines, continuous deployment, and native cloud applications.

The Docker documentation describes Docker as an open platform for developing, shipping, and running applications as shown in Figure 1.3, allowing applications to be separated from the infrastructure so that software can be delivered quickly [1, 15, 16]. Docker Engine is described as an open-source containerization technology that follows a client-server model with a long-running daemon process, APIs, and a command-line client [17]

Why containers are preferred in cloud-native environments: Containers are preferred because they provide portability, faster startup, higher density, and simplified deployment. The same container image can run on a developer's laptop, testing server, private cloud, or public cloud. This reduces the common problem of environment mismatch. Containers also support microservice architecture because each service can be packaged and deployed independently.

Limitations of containers: Although containers provide many advantages, they also introduce new resource-management challenges. Since containers share the host kernel, multiple containers compete for CPU, memory, disk I/O, and network bandwidth. If the scheduler places many resource-intensive containers on the same node, performance degradation may occur. Therefore, the lightweight nature of

containers does not eliminate the need for scheduling; rather, it makes intelligent scheduling more important.

TABLE 1.2: Comparison of Virtual Machines and Docker Containers

Parameter	**Virtual Machine**	**Docker Container**
Virtualization Level	Hardware-level virtualization	Operating-system-level virtualization
Operating System	Each VM has a complete guest OS	Containers share the host OS kernel
Startup Time	Slower because full OS boots	Faster because only container process starts
Resource Overhead	Higher due to guest OS and virtual hardware	Lower due to shared kernel
Isolation	Strong isolation	Process-level isolation
Portability	Depends on VM image and hypervisor	High portability through container images
Resource Density	Lower number of instances per host	Higher number of containers per host
Scaling Speed	Comparatively slower	Faster and suitable for microservices
Best Use Case	Strong isolation and heterogeneous OS needs	Cloud-native applications and microservices

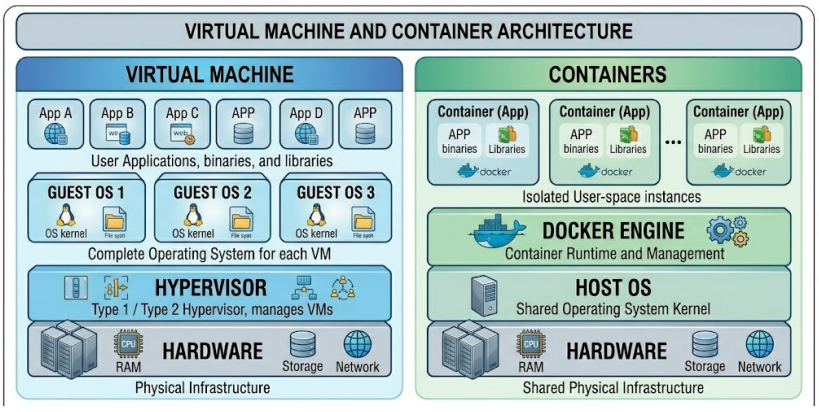


FIGURE 1.2: Virtual Machine and Container Architecture [1–4]

1.4 Docker Ecosystem and Container Lifecycle

As shown in Table 1.3 and Figure 1.3 Docker is one of the most widely used containerization platforms. It provides tools and mechanisms for building, packaging, distributing, running, networking, and monitoring containers. The Docker ecosystem consists of several important components, including Docker Engine, Docker client, Docker daemon, Docker images, Docker containers, Docker registry, Dockerfile, volumes, and networks [6].

Docker Engine: Docker Engine is the core runtime responsible for building and running containers. It follows a client-server architecture. The Docker client sends commands to the Docker daemon. The daemon builds images, starts containers, manages networks, handles volumes, and communicates with registries. This architecture allows users and automation scripts to control containers through command-line instructions or APIs.[7]

Docker image: A Docker image is a read-only template used to create containers. It contains the application code, libraries, dependencies, runtime, environment variables, and configuration instructions. Images are usually built using Dockerfiles. Since images are layered, Docker can reuse existing layers and reduce storage duplication. This improves efficiency in development and deployment.

Docker container: A Docker container is a running instance of a Docker image. It executes as an isolated process on the host operating system while sharing the host kernel. Containers can be started, stopped, restarted, removed, inspected, and monitored. Each container has its own filesystem, process space, and network configuration, but the underlying resources are shared with other containers [8].

Docker registry: A Docker registry stores and distributes Docker images. Docker Hub is a public registry, while organizations may also maintain private registries for internal images. Registries support image versioning and distribution across development, testing, and production environments.

Dockerfile and image building: A Dockerfile is a text file containing instructions for building a Docker image. It may specify the base image, application files, dependency installation, environment variables, exposed ports, and startup commands. Dockerfiles improve reproducibility because the same image can be rebuilt using the same instructions [2].

Container lifecycle: The container lifecycle begins with application development and image creation. The image is then pushed to a registry and pulled by the target environment. A container is created from the image and then started. During execution, the container consumes CPU, memory, disk, and network resources. It can be monitored, scaled, stopped, restarted, or removed. In orchestration environments, this lifecycle is automated [18].

TABLE 1.3: Docker Components and Their Role

Docker Component	**Function**	**Role in Thesis Context**
Docker Engine	Builds and runs containers	Provides container runtime for the experimental environment
Docker Client	Sends commands to Docker daemon	Used by administrator or Python orchestrator
Docker Daemon	Manages images, containers, networks, and volumes	Executes container operations
Docker Image	Template for container creation	Used to deploy workload containers
Docker Container	Running instance of an image	Represents the workload to be scheduled
Docker Registry	Stores and distributes images	Supports image availability across nodes
Docker Network	Provides communication among containers	Required for Swarm services
Docker Volume	Provides persistent storage	Useful for stateful applications

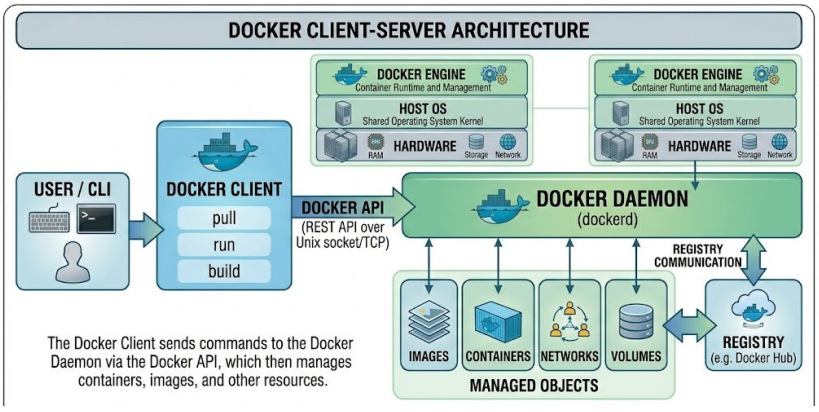


FIGURE 1.3: Docker Client–Server Architecture

1.5 Microservices, Containerized Applications, and Orchestration

Modern cloud applications commonly use microservice architecture, where a large application is divided into small, independent services. Each service handles a specific function and communicates with others through APIs or lightweight protocols. This approach improves modularity, scalability, maintainability, and deployment flexibility. The comparison is shown in Figure 1.4 & Table 1.4.

Microservices and containers: Containers are highly suitable for microservices because each microservice can be packaged as a separate container. This allows different services to be developed in different languages, deployed independently, and scaled according to demand. For example, in an e-commerce application, the user service, product service, payment service, recommendation service, and notification service may all run as separate containers [19].

Benefits of microservice deployment: Microservices improve development speed because teams can work independently on different services. They improve fault isolation because failure in one service does not necessarily stop the entire application. They improve scalability because heavily used services can be scaled independently. They also support continuous integration and continuous deployment because updates can be rolled out service by service.

Challenges of microservice deployment: Microservices introduce new operational challenges. A single user request may pass through multiple services. Some services may be CPU-intensive, while others may be memory-intensive or network-intensive. Poor placement of microservice containers may increase latency, resource contention, and cascading delays. Therefore, orchestration and scheduling become critical[20].

Container orchestration: Container orchestration refers to the automated management of containers across multiple nodes. It includes deployment, scheduling, scaling, networking, service discovery, rolling updates, monitoring, failure recovery,

and load balancing. Without orchestration, managing many containers manually across a distributed infrastructure becomes difficult and unreliable [5, 11].

Popular orchestration tools: Docker Swarm and Kubernetes are two widely known orchestration platforms. Kubernetes is a production-grade open-source system for automating deployment, scaling, and management of containerized applications [21–23]. Docker Swarm is Docker's native clustering and orchestration mechanism. It is si mpler and tightly integrated with Docker, making it suitable for lightweight cluster environments and research experimentation.

TABLE 1.4: Comparison of Docker Swarm and Kubernetes

Parameter	**Docker Swarm**	**Kubernetes**
Integration	Native integration with Docker	Independent orchestration platform
Setup Complexity	Simple and lightweight	More complex but feature-rich
Main Workload Unit	Service and task	Pod
Scheduling	Basic scheduling with Swarm manager	Advanced scheduling with filtering and scoring
Scalability	Suitable for small to medium clusters	Suitable for large-scale production clusters
Learning Curve	Easier	Steeper
Research Usefulness	Good for controlled scheduling experiments	Good for advanced orchestration research
Relevance to Thesis	Experimental platform used for DT+APSO	Future extension possibility

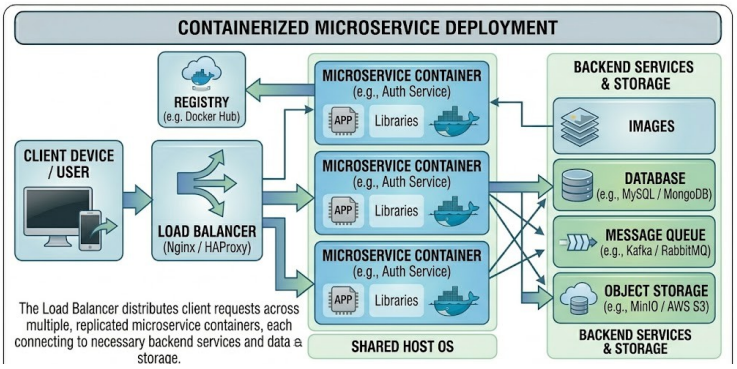


FIGURE 1.4: Containerized Microservice Deployment

1.6 Docker Swarm Architecture and Scheduling Limitations

Docker Swarm is Docker's native orchestration solution also shown in Figure 1.5 and listed in Table 1.5. It enables multiple Docker hosts to be grouped into a single logical cluster. A Docker Swarm cluster consists of manager nodes and worker nodes. The manager node maintains cluster state, accepts service definitions, makes scheduling decisions, and dispatches tasks. Worker nodes receive tasks from the manager and execute them as containers.

Docker documentation explains that a node is an instance of Docker Engine participating in the swarm as shown in Figure 1.5. To deploy an application, a service definition is submitted to a manager node, and the manager dispatches tasks to worker nodes [24, 25]. Docker also explains that a service represents the desired state and a task performs the actual work scheduled on swarm nodes [6].

Manager node: The manager node is responsible for orchestration decisions. It manages cluster membership, schedules tasks, monitors worker nodes, and ensures that the desired number of service replicas is running. If a container fails, the manager can reschedule it on another node.

Worker node: Worker nodes execute the tasks assigned by the manager. They run containers and report their status back to the manager. In a larger cluster, multiple worker nodes may exist, each with different CPU, memory, and workload conditions.

Service and task: A service defines the desired state of an application. For example, a service definition may specify that three replicas of a web application should be running. A task is the actual execution unit assigned to a node. Each task usually corresponds to one container instance.

Default scheduling limitations: Although Docker Swarm provides basic scheduling, it does not fully consider advanced predictive resource behaviour. In practical situations, a node may have fewer containers but higher CPU usage. Another node may have more containers but lower resource pressure. A scheduler that only considers container count may make poor placement decisions. Similarly, if the scheduler does not distinguish between CPU-intensive and memory-intensive containers, it may overload one type of resource.

Resource-awareness limitation: Default scheduling strategies are useful for simple deployment, but they may not sufficiently use real-time CPU utilization, memory pressure, active container count, task density, historical load behaviour, or future overload risk. This creates the possibility of resource imbalance across nodes.

Need for intelligent scheduling: The present thesis addresses this limitation by proposing a hybrid scheduling framework. The Decision Tree component classifies node suitability based on real-time metrics, while APSO searches for an optimized container placement. This approach improves scheduling intelligence without completely replacing the Docker Swarm platform.

Table 1.5: Limitations of Default Docker Swarm Scheduling

Limitation	**Explanation**	**Impact**
Limited resource awareness	Scheduling may not deeply consider CPU and memory pressure	Uneven utilization
Lack of prediction	Future overload risk is not considered	Reactive scheduling
Workload similarity assumption	Different workloads may be treated similarly	Resource contention
Limited adaptivity	Scheduling logic does not learn from workload changes	Reduced performance under dynamic load
Heterogeneity issue	Different node capacities may not be fully reflected	Unfair placement
No metaheuristic optimization	Placement is not optimized through a search algorithm	Suboptimal allocation

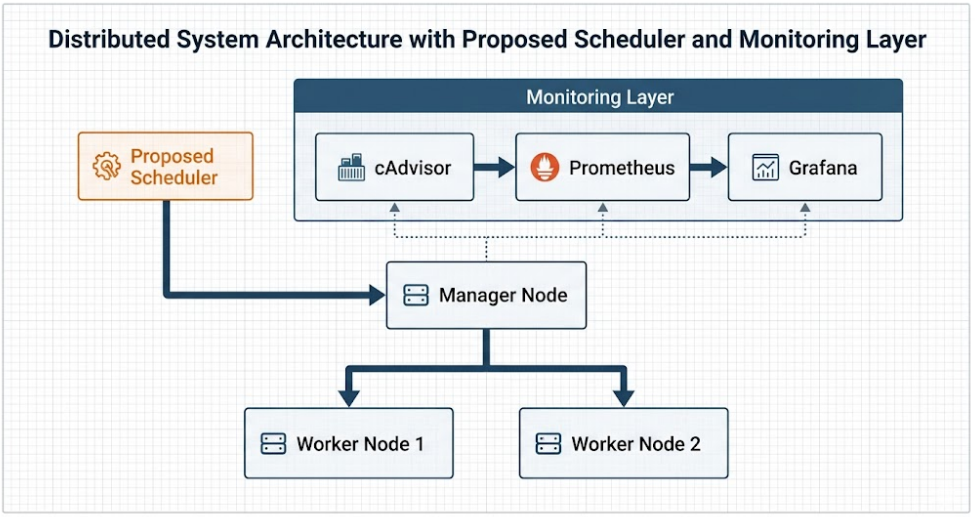


Figure 1.5: Docker Swarm Architecture with Monitoring Layer

1.7 Load Balancing and Resource Allocation Challenges in Container Clouds

Load balancing is one of the most important techniques for maintaining performance and reliability in cloud computing. It distributes workload among available resources so that no single node becomes overloaded while others remain underutilized. In containerized environments, load balancing includes both request-level balancing and resource-level balancing. Request-level balancing distributes incoming user requests among service replicas. Resource-level balancing ensures that containers are placed across nodes in a way that balances CPU, memory, and other resource usage.

Static and dynamic load balancing: Static load balancing methods use predefined rules and do not change according to runtime conditions. They are simple but less effective when workloads are dynamic. Dynamic load balancing methods use real-time system information such as CPU utilization, memory usage, queue length, response time, or node status. Dynamic methods are more suitable for cloud and container environments because workload changes are frequent.

Centralized and distributed load balancing: In centralized approaches, one controller collects system information and makes scheduling decisions. This is simpler but may become a bottleneck or single point of failure. In distributed approaches, multiple nodes participate in decision-making. This improves scalability but increases coordination complexity.

Resource allocation in Docker containers: Resource allocation involves assigning CPU, memory, storage, and network capacity to containers. In Docker Swarm, it also involves deciding which worker node should run a container task. The quality of this decision affects overall performance. Poor placement can lead to CPU saturation, memory pressure, increased response time, and reduced service stability.

CPU utilization: CPU utilization indicates how much processing capacity is being used by a container or node. High CPU utilization may indicate heavy computation, but if it

remains high for a long time, it can cause delays and bottlenecks. In the present thesis, CPU utilization is a major metric for evaluating scheduling quality.

Memory utilization: Memory utilization indicates how much RAM is being consumed. High memory pressure can lead to swapping, degraded performance, or container failure. Memory-aware scheduling is therefore important. A scheduler that focuses only on CPU may still create a memory imbalance.

Resource imbalance: Resource imbalance occurs when one node carries a larger workload than others. For example, one worker node may have high CPU and memory usage while another node remains lightly loaded. Such imbalance reduces the effectiveness of the cloud resource pool.

Interference among containers: Containers running on the same node share the host kernel and hardware resources. If multiple CPU-intensive containers are placed on the same node, they may compete for processor time. If multiple memory-intensive containers are placed together, they may cause memory pressure. Therefore, container placement should consider the resource profile of workloads.

Energy and cost implications: Resource imbalance also affects energy consumption and operational cost. Overloaded nodes may consume more power, while underutilized nodes waste capacity. Efficient resource allocation can reduce dynamic CPU power consumption and improve infrastructure utilization.

1.8 Intelligent Scheduling Using Machine Learning and Decision Tree

Machine learning can improve cloud resource scheduling by learning patterns from historical and real-time data. Instead of relying only on static rules, machine learning models can classify workload states, predict future resource usage, and support better decision-making. In container environments, machine learning can be used to estimate how many replicas are needed, identify overloaded nodes, predict workload growth, or classify node suitability.

Why machine learning is useful for scheduling: Cloud workloads are dynamic and often non-linear. CPU usage, memory consumption, request rate, and response time may change rapidly. Manual rules may not capture these variations effectively. Machine learning models can learn relationships among system parameters and support more adaptive scheduling.

Decision Tree as a scheduling support model: A Decision Tree is a supervised learning algorithm that classifies data through a tree-like structure of decisions. It splits data based on feature values and produces a class label or prediction. Decision Trees are useful in scheduling because they are interpretable and computationally efficient. A Decision Tree can classify a node as underloaded, balanced, or overloaded based on CPU and memory usage.

Node suitability analysis: Node suitability refers to whether a node is appropriate for receiving a new container task. A suitable node should have enough CPU and memory capacity and should not show signs of future overload. In the proposed thesis, the Decision Tree model evaluates node suitability using real-time system parameters.

Input features for Decision Tree: Possible input features include CPU utilization, memory utilization, active container count, historical CPU slope, memory growth trend, task density, and CPU throttling rate. These features represent the current and near-future state of each node.

Decision Tree output: The Decision Tree output can be represented as a load class or cost score. For example, a low-load node may receive a low cost, a medium-load node may receive a moderate cost, and a high-load node may receive a high cost. This cost can then be integrated into the APSO fitness function.

Advantages of Decision Tree: The Decision Tree model is easy to interpret, fast to execute, and suitable for real-time decision support. It does not behave like a black-box model in the same way as many complex machine learning methods. For a PhD thesis based on resource allocation, this interpretability is useful because it helps explain why a node is selected or avoided.

Limitations of Decision Tree: A Decision Tree may become less accurate if workload patterns change significantly and the model is not updated. It may also overfit if not properly controlled. However, in the proposed model, Decision Tree is not used alone. It is combined with APSO, which performs the optimization step.

1.9 Thesis Organization, Key Terminology, and Chapter Summary

Organization of the thesis: The thesis is organized into six chapters. Chapter 1 introduces the research background, cloud computing fundamentals, virtualization, containerization, Docker, Docker Swarm, load balancing, resource allocation, Decision Tree, APSO, proposed framework, problem statement, objectives, scope, and significance. Chapter 2 presents a detailed review of literature related to cloud resource allocation, Docker container scheduling, load balancing, PSO-based optimization, and machine learning-based scheduling. Chapter 3 explains the proposed methodology, system architecture, mathematical model, Decision Tree classifier, APSO algorithm, monitoring tools, and experimental workflow. Chapter 4 presents data analysis and experimental results. Chapter 5 discusses the findings with respect to objectives and literature. Chapter 6 concludes the thesis and provides recommendations and future research directions Table 1.6 Shows Key terminology used in thesis.

TABLE 1.6: Important Terminology Used in the Thesis

Term	**Meaning in This Thesis**
Cloud Computing	On-demand delivery of shared computing resources over a network.
Container	Lightweight isolated environment sharing the host OS kernel.
Docker	Platform for building, running, and managing containers.
Docker Image	Template used to create containers.
Docker Container	Running instance of a Docker image.
Docker Registry	Repository for storing and distributing Docker images.
Docker Swarm	Native Docker orchestration platform for container clusters.
Manager Node	Swarm node responsible for scheduling and cluster management.
Worker Node	Swarm node responsible for executing assigned tasks.
Service	Desired state of an application in Docker Swarm.
Task	Actual unit of work scheduled on a Swarm node.
Load Balancing	Distribution of workload to avoid overload and underutilization.
Resource Allocation	Assignment of CPU, memory, and workload to available nodes.
Decision Tree	Classifier used for node suitability analysis.
PSO	Particle Swarm Optimization, a population-based optimization algorithm.
APSO	Adaptive PSO with dynamically adjusted optimization parameters.
DT+APSO	Proposed hybrid Decision Tree and Adaptive PSO scheduling framework.

Chapter summary: This chapter establishes the conceptual foundation of the thesis by tracing the evolution of cloud computing from traditional service models and virtualization to lightweight containers, microservices, and Docker Swarm architecture. After identifying critical scheduling limitations, load-balancing hurdles, and resource allocation challenges in modern cloud environments, the chapter explores intelligent solutions, including Decision Tree node suitability analysis and Adaptive Particle Swarm Optimization (APSO). It concludes by introducing the proposed DT+APSO framework, a hybrid, machine learning-driven approach designed to optimize container scheduling and resource allocation in Docker Swarm.

The next chapter presents the review of literature, where existing research on Docker container scheduling, load balancing, PSO-based resource allocation, machine learning-based scheduling, and hybrid optimization techniques is critically analyzed.

CHAPTER 2

Literature Survey

2.1 Introduction

The purpose of this chapter is to present a detailed and critical review of the literature related to *Optimized Resource Allocation for Docker Container-Based Cloud Environments Using Hybrid Adaptive PSO and Decision Tree*. The review is organized around the core research themes of the thesis: cloud computing, virtualization, Docker containerization, container orchestration, Docker Swarm scheduling, load balancing, container resource allocation, machine learning-based scheduling, Particle Swarm Optimization-based scheduling, and hybrid intelligent optimization techniques[9, 26, 27].

Container-based cloud computing has become a major research direction because containers provide a lightweight and portable alternative to traditional virtual machines. However, as the number of containers increases across distributed cloud nodes, resource allocation and load balancing become increasingly complex. The major challenge is to place containers or service replicas across available nodes in such a way that CPU utilization, memory usage, service response, and resource fairness remain stable. The approved thesis of this research identifies this problem by stating that existing Docker Swarm schedulers lack adaptability to dynamic cloud workloads, resulting in poor load balancing, node overloading, and inefficient resource utilization. The same thesis defines the objective of developing a hybrid Adaptive Particle Swarm Optimization and Decision Tree-based framework for Docker Swarm resource allocation [27, 28].

The literature shows that earlier work in this area evolved from traditional load-balancing algorithms and virtual machine scheduling toward container scheduling, microservice orchestration, predictive resource allocation, and metaheuristic optimization. Several researchers have used methods such as Round Robin, Weighted Round Robin, First Fit, Bin Packing, Ant Colony Optimization, Particle Swarm Optimization, machine learning, reinforcement learning, and multi-objective optimization for cloud scheduling. However, the problem is still open because no single scheduling method performs best under all types of workload, cluster sizes, resource heterogeneity, and dynamic runtime conditions.

The first research paper related to this thesis emphasizes that cloud applications face several challenges of docker container. It also explains that Docker, Kubernetes, and related container technologies are increasingly used to improve deployment and load balancing, but efficient resource utilization remains a major concern [29]. The second research paper further extends this direction by proposing the combination of PSO and Decision Tree Classification, where PSO searches for optimal resource configurations and Decision Tree Classification identifies patterns in historical resource utilization for predictive modelling [30].

This chapter is therefore written as a state-of-the-art literature review, not only as a summary of existing work. It compares prior research, identifies limitations, and establishes the research gap addressed by the proposed DT+APSO model. The review also connects earlier studies with the specific objectives of this thesis, especially intelligent node suitability analysis, adaptive swarm-based optimization, Docker Swarm-based implementation, and CPU/memory-aware performance evaluation.

A literature review in this area must cover three layers of research. The first layer includes general cloud computing and virtualization studies, which explain why containers emerged as a lightweight alternative to virtual machines. The second layer includes Docker, Docker Swarm, Kubernetes, and container orchestration studies, which explain how containers are deployed and managed across clusters. The third layer includes scheduling algorithms, machine learning methods, metaheuristic approaches, and hybrid optimization models, which are directly related to the thesis

contribution. The proposed DT+APSO method is positioned in the third layer, but it is strongly dependent on the first two layers because scheduling decisions are meaningful only when the underlying cloud and container architecture is clearly understood.

2.2 Evolution of Cloud Resource Management from Virtual Machines to Containers

Cloud computing originally relied heavily on hypervisor-based virtual machines. Virtual machines enabled multiple guest operating systems to run on the same physical server, thereby improving hardware utilization. However, virtual machines require a full guest operating system, virtual hardware abstraction, and significant memory and storage resources. This overhead becomes a limitation when applications must be deployed rapidly or scaled frequently.

Container-based virtualization emerged as a lightweight alternative. Containers share the host operating system kernel and package only the application and its dependencies. Because containers do not require a separate guest operating system, they start faster, consume fewer resources, and provide higher deployment density. In contrast, container-based virtualization uses a container engine such as Docker, shares the host kernel, and supports faster execution and portability [6, 24].

Docker became popular because it simplified the process of building, packaging, and deploying applications. Docker images provide portability, while Docker containers provide isolated execution. In cloud-native environments, containers are especially suitable for microservices because each service can be independently packaged, deployed, scaled, and updated.

However, containerization does not eliminate resource-management problems. In fact, the dynamic nature of containers makes scheduling more complex. A container may be short-lived, lightweight, CPU-intensive, memory-intensive, or network-intensive. When hundreds or thousands of containers run across multiple nodes, poor placement

can lead to CPU bottlenecks, memory pressure, network congestion, and unstable service performance.

The shift from virtual machines to containers can be seen as a shift from coarse-grained resource allocation to fine-grained resource allocation. In virtual machine-based environments, the scheduler usually assigns a relatively large VM instance to a physical host. In a container-based environment, many small containers may run simultaneously on the same host. This increases packing density but also increases the risk of resource interference. Two containers running on the same node may compete for CPU cycles, memory pages, disk I/O, network bandwidth, or cache resources. Therefore, container scheduling cannot be treated as a simple extension of VM placement; it requires its own resource-aware logic detailed comparison shown in Figure 2.1 and Table 2.1.

In traditional virtualized data centers, VM migration was often used to correct imbalance. In container-based systems, container instances are generally recreated or rescheduled rather than migrated in the same manner as full VMs. This makes container orchestration faster, but it also requires accurate scheduling decisions at the time of placement. The lightweight nature of containers allows fast scaling, yet frequent scaling without intelligent placement may lead to oscillation, uneven load, or unnecessary resource consumption.

TABLE 2.1: Virtual Machine and Container-Based Virtualization

Parameter	**Virtual Machine-Based Virtualization**	**Container-Based Virtualization**	**Relevance to Thesis**
Virtualization level	Hardware-level through hypervisor	OS-level through shared kernel	Thesis focuses on container-level scheduling
Operating system	Each VM has separate guest OS	Containers share host OS kernel	Reduces overhead but increases shared-resource contention
Startup time	Slower because complete guest OS must boot	Faster because container process starts directly on host kernel	Important for dynamic service scaling
Resource overhead	High due to guest OS and virtual hardware abstraction	Low due to shared kernel and lightweight packaging	Supports higher density but requires better scheduling
Isolation	Stronger isolation through VM boundary	Comparatively lighter process-level isolation	Requires careful resource and security management
Portability	VM-image and hypervisor dependent	High image portability through Docker images	Useful in Docker-based deployments
Scheduling issue	VM placement and VM migration	Container placement and replica allocation	Thesis addresses Docker container placement
Main challenge	Reducing VM overhead and migration cost	Avoiding container resource imbalance and interference	Supports DT+APSO research direction

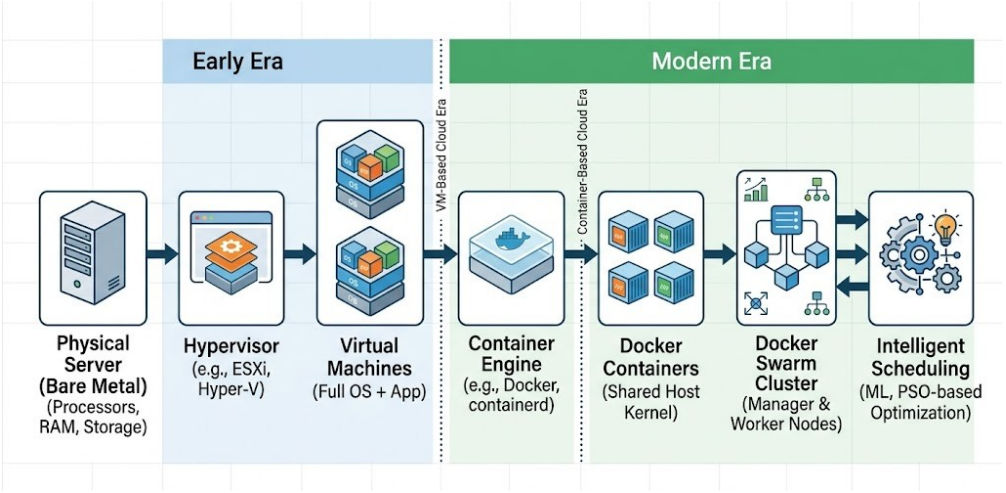


FIGURE 2.1: Evolution from VM-Based Cloud to Container-Based Cloud

The literature indicates that containerization improves speed and resource density, but it creates a new scheduling problem: how to allocate lightweight but highly dynamic workloads across shared nodes. This forms the foundation for the present thesis. The proposed research therefore does not merely compare VMs and containers; it uses the containerization advantage as the basis for developing an intelligent scheduling model that can preserve container benefits while reducing the limitations of default placement.

2.3 Docker Container Ecosystem, Microservices, and Orchestration Platforms

Docker provides a practical ecosystem for container creation, execution, distribution, and orchestration. Its main components include Docker Engine, Docker images, Docker containers, Docker registries, Docker networks, and Docker volumes. Several studies describe Docker technologies through four core components: Docker client/server, Docker images, Docker registries, and Docker containers. Docker is generally considered superior to traditional VM architecture in speed, portability, rapid delivery, and density, while still facing limitations such as security and scheduling issues [1, 15, 31].

Microservice architecture has increased the importance of containers, In a microservice-based system, an application is divided into several independent services. Each service can be deployed as a separate container. This enables independent scaling and rapid deployment, but it also increases scheduling complexity because service instances may have different workload characteristics.

Container orchestration platforms automate the deployment, scaling, scheduling, networking, and monitoring of containers. The most widely used platforms are Docker Swarm and Kubernetes. Docker Swarm is simpler, Docker-native, and suitable for lightweight orchestration, while Kubernetes provides advanced scheduling, autoscaling, service discovery, and cluster-management functions.

Recent literature confirms that container orchestration remains a central problem in cloud-native systems. Senjab et al. reviewed Kubernetes scheduling algorithms and categorized them into generic scheduling, multi-objective optimization-based scheduling, AI-focused scheduling, and autoscaling-enabled scheduling, while identifying several unresolved issues for future research [14]. Carrión also examined Kubernetes scheduling taxonomy and ongoing issues, showing that scheduling gaps remain relevant even in advanced orchestration platforms [13].

Although Kubernetes dominates large-scale cloud-native orchestration, Docker Swarm remains useful for research because of its lightweight design, simpler configuration, and direct integration with Docker Engine. The present thesis uses Docker Swarm as the experimental platform because it allows controlled evaluation of scheduling strategies such as default Swarm scheduling, standard PSO, and the proposed DT+APSO model.

The Docker ecosystem is not limited to running containers. It also supports image repositories, overlay networks, service definitions, task replication, health checks, and cluster membership. These features allow Docker Swarm to act as a lightweight cloud orchestration platform. However, the scheduling layer of Docker Swarm is less flexible than the scheduling framework available in Kubernetes. This difference is important for the present thesis because it creates an opportunity to improve Docker Swarm through an external intelligent optimization model.

A microservice-based application can have several different execution profiles. Some services may be stateless, some may be stateful, some may be request-intensive, and others may be computation-intensive. If a scheduler simply spreads containers without considering the nature of the services, it may produce an apparently balanced container count but an unbalanced resource state. For example, two CPU-intensive replicas placed on the same worker may create a CPU bottleneck even if the number of containers on the node appears reasonable. This is why the literature increasingly emphasizes resource-aware and prediction-aware scheduling.

TABLE 2.2: Docker Swarm and Kubernetes in Scheduling Research

Feature	**Docker Swarm**	**Kubernetes**	**Research Implication**
Architecture	Manager-worker model	Control plane-worker node model	Both require intelligent placement
Scheduling unit	Service/task	Pod	Both assign workloads to nodes
Configuration complexity	Lower	Higher	Swarm is suitable for controlled PhD experiments
Scheduling flexibility	Basic built-in scheduling	Advanced filtering/scoring framework	Swarm offers scope for external optimization
Monitoring integration	Possible through cAdvisor, Prometheus, Grafana	Strong ecosystem support	Thesis uses Prometheus, cAdvisor, and Grafana
Research gap	Limited predictive/adaptive scheduling	Complex multi-objective scheduling still open	Supports hybrid intelligent scheduling
Practical thesis relevance	Easy to implement and evaluate in a small cluster	Suitable for future large-scale extension	DT+APSO can later be generalized toward Kubernetes

2.4 Load Balancing in Cloud and Container-Based Environments

Load balancing is a fundamental technique in distributed computing and cloud computing. It distributes workloads across available computing nodes so that no node remains overloaded while another remains idle. Effective load balancing improves response time, resource utilization, scalability, availability, and reliability. Figure 2.2 shows classifications of Load balancing approaches.

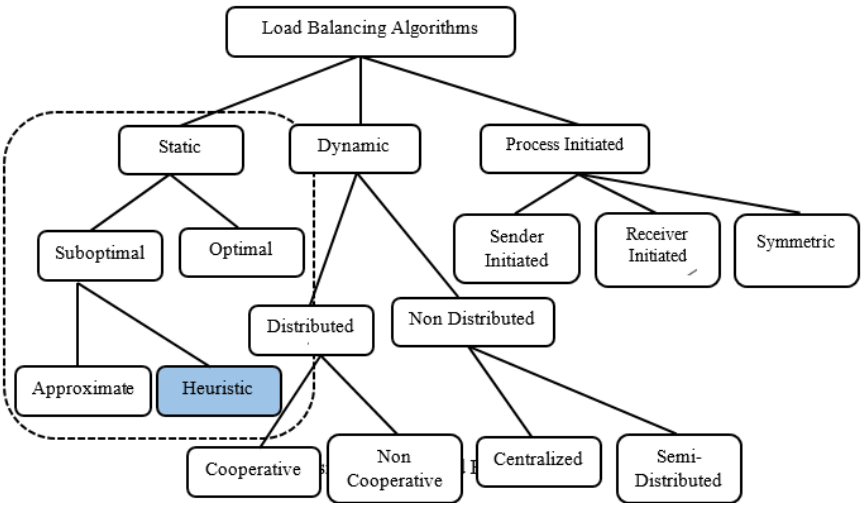


FIGURE 2.2: Classification of Load-Balancing Algorithms

Liu et. al .explain that load balancing maps incoming jobs to available resources and aims to distribute workloads evenly across nodes. They further state that ideal load balancing prevents nodes from being overloaded or underloaded. Load-balancing algorithms can be classified into static and dynamic approaches, and further into centralized, decentralized, cooperative, non-cooperative, adaptive, non-adaptive, sender-initiated, receiver-initiated, and pre-emptive categories [22, 32, 33].

In containerized environments, load balancing operates at two levels. The first is request-level load balancing, where user requests are distributed across service replicas. The second is resource-level load balancing, where containers are placed across nodes to balance CPU, memory, I/O, and network usage. The present thesis focuses mainly on the second type: resource-level load balancing through optimized container placement.

Traditional approaches such as Round Robin, Weighted Round Robin, First Come First Serve, and Least Connection are simple but often fail under dynamic workloads. They generally do not consider future resource pressure, task density, node heterogeneity, or historical workload trends. We also note that traditional methods such as Round Robin,

FCFS, and least connection are not sufficient because they cannot manage dynamic workloads effectively [22, 34].

Cloud load balancing is inherently complex because "load" is a multidimensional metric encompassing factors like CPU, memory, disk I/O, and network traffic; however, relying on only one or two indicators can cause scheduling failures, such as assigning a task to a low-CPU node that is already suffering from critical memory exhaustion. To address this, the present thesis focuses on CPU and memory utilization as the primary performance indicators since both are directly measurable and vital to Docker Swarm stability, where CPU imbalance causes processing delays and memory imbalance leads to severe service instability. Consequently, modern load-balancing literature directly informs the proposed DT+APSO model, which is specifically designed to evaluate these multiple, interconnected indicators simultaneously for optimal container allocation.

The literature suggests that dynamic, adaptive, and predictive load balancing is more suitable for modern container environments than static scheduling. This directly supports the proposed hybrid DT+APSO approach.

2.5 Container Scheduling and Resource Allocation: State of the Art

Container scheduling determines where containers or service replicas should run in a cluster. The scheduler must consider resource availability, workload demand, node health, service constraints, performance requirements, and sometimes energy or cost. The container scheduling problem is challenging because workloads are dynamic, resource requirements are multi-dimensional, and cluster conditions change continuously.

Ahmad et al. presented a major survey and assessment of container scheduling techniques. They observed that despite the increasing use of containers in cloud computing, container scheduling remains an active research area requiring more

systematic study. Their survey classifies existing techniques and identifies research opportunities in scheduling, resource allocation, and optimization [7, 18]. Netaji et al [18]. also reviewed container resource allocation, management, and scheduling, noting that resource allocation in containerized cloud environments must dynamically or statically allocate CPU, memory, disk, and other resources to users [18].

Several container scheduling and load-balancing approaches have been reported in the literature. These include machine learning-based scheduling in microservice architecture, lightweight virtualization for big data applications, chain-oriented load balancing, dynamic priority-based weighted scheduling, decentralized microservice load balancing, Docker Swarm scheduling strategy, Kubernetes resource scheduling, and dynamic web-cluster balancing using Docker containers.

Container scheduling can be understood as a mapping problem. A set of containers or service replicas must be mapped onto a set of available nodes. This mapping must respect resource constraints and performance objectives. In the simplest case, the scheduler may use available CPU or memory as the decision factor. In more advanced cases, the scheduler may consider current load, predicted load, response time, energy cost, service-level agreement, failure probability, and migration overhead.

The scheduling problem becomes more complicated when containers are part of a microservice chain. A microservice chain may include several dependent services, and the performance of one service may affect the response time of the entire chain. A scheduler that optimizes only individual container placement may fail to optimize the full service-chain behaviour. Some literature therefore focuses on chain-oriented or latency-aware scheduling. However, these approaches may not fully address CPU and memory imbalance across Docker Swarm nodes, which remains the focus of the present thesis.

TABLE 2.3: Summary of Container Scheduling Approaches from Literature

Approach	**Main Objective**	**Major Advantage**	**Limitation**	**Metrics Used**
Machine learning-based microservice scheduling [5]	Improve application performance using machine learning models	Maintains load pressure and enhances overall system performance	Inter-service relationships are not fully considered	Response time, load pressure
Lightweight virtualization for big data [35, 36]	Improve the execution of big data workloads	Spark and Docker-based execution may reduce processing time	Limited to large-scale applications and selected workloads	CPU, memory, disk usage, execution time
Chain-oriented load balancing [37]	Balance workload across microservice chains	Reduces latency through priority queue-based processing	Shorter service chains may be affected during scheduling	Latency, processing capacity
Dynamic priority weighted scheduling [22]	Reduce latency in chained microservice execution	Dynamically adjusts request priority during scheduling	Leaf nodes are not fully balanced	Latency, priority queues

Continued on next page

34

Table 2.3 – continued from previous page

Approach	**Main Objective**	**Major Advantage**	**Limitation**	**Metrics Used**
Decentralized microservice load balancing [26]	Support live migration and distributed scheduling decisions	Container migration is faster than traditional VM migration	Microservice management becomes more complex in distributed settings	Migration time, service management
Docker Swarm scheduling strategy [24]	Improve service scheduling using SLA-based service classes	Supports priority-based categorization of services	Requires stronger dynamic consolidation support	Energy consumption, SLA, cost
Kubernetes resource scheduling [13]	Improve pod allocation and resource utilization	Hybrid ACO–PSO scheduling shows promising performance	Pod selection may not be fully dynamic	Efficiency, resource allocation
Dynamic Docker web-cluster strategy [38]	Improve availability and concurrency in Docker-based clusters	Performs better than traditional Round Robin scheduling	Applicability is limited to specific deployment scenarios	Concurrency, availability

Continued on next page

Table 2.3 – continued from previous page

Approach	**Main Objective**	**Major Advantage**	**Limitation**	**Metrics Used**
Container consolidation with prediction [39]	Reduce active physical machines while maintaining QoS	Supports energy-aware container placement	Requires accurate prediction and efficient migration logic	Energy consumption, SLA, CPU usage
Hybrid intelligent scheduling [21]	Combine learning-based prediction with optimization-based scheduling	Supports adaptive and intelligent scheduling decisions	Requires careful model tuning and monitoring integration	CPU usage, memory usage, resource efficiency

36

A major observation from this literature is that many algorithms improve one aspect of scheduling but remain limited in other aspects. For example, latency-focused methods may not optimize memory utilization. SLA-based methods may not address CPU variance. Machine learning methods may classify load but may not perform placement optimization. This motivates the need for a hybrid framework that combines classification and optimization.

2.6 Machine Learning-Based Scheduling and Prediction in Container Clouds

Machine learning has become important in cloud resource management because cloud workloads are dynamic and difficult to predict using static rules. Machine learning models can learn from historical and real-time monitoring data and support proactive decisions. In container scheduling, ML may be used for load prediction, autoscaling, node classification, anomaly detection, or resource-demand estimation.

Previous studies on machine learning-based scheduling in microservice architecture used algorithms such as Artificial Neural Networks and SVM. The CSML approach improved cluster performance and adjusted container load pressure, but future work was needed to consider relationships between services [5]. A recent trend is toward workload prediction for proactive scheduling. An efficient load prediction-driven scheduling strategy proposed in 2023 specifically addresses container-cloud scheduling using load prediction [35, 40]. Similarly, Feng et al. emphasized that workload prediction is critical for proactive resource management of cloud applications because accurate prediction supports better resource provisioning [39].

Decision Tree-based methods are attractive in this context because they are interpretable, fast, and suitable for rule-based node suitability analysis. Our Study explain that a Decision Tree can classify containers as overloaded, underutilized, or balanced using real-time resource parameters such as CPU usage, memory consumption, and request rate. They further note that Decision Trees are useful for

real-time Docker load balancing because they process data quickly, although retraining may be needed when workload patterns change [16, 33].

Machine learning-based resource management has two major advantages. First, it allows the scheduler to move from reactive decision-making to proactive decision-making. A reactive scheduler responds only after a node becomes overloaded. A predictive scheduler can avoid overload by identifying risk before placement. Second, machine learning can help convert raw monitoring metrics into meaningful scheduling information. For example, CPU usage, memory pressure, and container count can be converted into a node suitability class.

However, machine learning models also have limitations. A model trained on one workload may not generalize to another workload. Complex models may require large amounts of data and significant training time. Some models are difficult to interpret. In contrast, Decision Tree models are easier to understand because they produce rule-based decisions. This is why a Decision Tree is suitable for the proposed thesis: it can act as a transparent node-suitability model and can be integrated with APSO without introducing excessive computational overhead.

TABLE 2.4: Machine Learning Methods for Container Scheduling [5]

ML Method	**Possible Use in Container Cloud**	**Strength**	**Limitation**	**Relevance to Thesis**
Decision Tree	Node/load classification	Interpretable and fast	May require retraining	Used in proposed model
Random Forest	Improved classification accuracy	Reduces overfitting	Higher computational cost	Future extension
SVM	Load classification	Works with smaller datasets	Harder to interpret and tune	Reviewed in prior ML scheduling
ANN	Workload prediction	Handles nonlinear patterns	Needs more training data	Useful for autoscaling
LSTM	Time-series workload prediction	Captures temporal behaviour	Computationally heavier	Future predictive scheduling
Reinforcement Learning	Scheduling policy learning	Learns from feedback	Complex training and reward design	Future intelligent scheduler
Gradient Boosting	Node suitability prediction	High predictive performance	Less interpretable than simple tree	Future comparison method

The literature supports as shown in Table 2.4 the use of machine learning for predicting or classifying resource conditions. However, prediction alone does not solve the container placement optimization problem. Therefore, the present thesis integrates Decision Tree with APSO to combine classification with search-based optimization.

2.7 Particle Swarm Optimization-Based Load Balancing and Scheduling

Particle Swarm Optimization is a population-based metaheuristic inspired by the movement of birds and fish. It has been widely applied to scheduling, resource allocation, clustering, routing, and optimization problems. In container scheduling, each particle can represent a possible mapping of containers to nodes. The fitness function evaluates each mapping based on CPU usage, memory usage, load variance, or other scheduling objectives.

Reseracher reviewed PSO variants for Docker container load balancing. They explain that standard PSO treats each particle as a potential solution for distributing Docker containers across nodes. They also discuss variants such as TMPSO [29], Adaptive PSO [41], MOPSO [19], HPSO [38], CPSO, DPSO, QPSO, Hybrid PSO [42], DMS-PSO, OBL-PSO, Chaotic PSO, and CF-PSO . The same paper reports comparative results for PSO, TMPSO, and Adaptive PSO, where average CPU load, memory usage, and execution time were compared across five containers.

Chen and Xiao [19] proposed a multi-objective and parallel PSO algorithm for container-based microservice scheduling. Their MOPPSO-CMS model considers physical-node resources, cluster load balancing, local load balancing, failure rate, and user requirements. This work is important because it shows that PSO can be extended from single-objective scheduling to multi-objective container scheduling.

The literature indicates that PSO is useful because it is simple, fast, and flexible. However, standard PSO may suffer from premature convergence, especially when inertia and acceleration coefficients are fixed. Adaptive PSO addresses this issue by dynamically adjusting parameters to balance exploration and exploitation. Researcher state that Adaptive PSO dynamically adjusts the swarm's behaviour in response to workload fluctuations, making it effective for real-time optimization in containerized environments [29].

The proposed thesis improves the classical PSO-based scheduling idea by introducing two important enhancements. First, it uses Adaptive PSO instead of fixed-parameter PSO, allowing dynamic adjustment of search behaviour. Second, it integrates Decision Tree-based node suitability into the placement evaluation process. This combination allows the algorithm to search efficiently while being guided by resource-state classification.

TABLE 2.5: PSO Variants and Their Relevance to Container Scheduling

PSO Variant	**Main Feature**	**Strength**	**Limitation**	**Relevance to Thesis**
Standard PSO	Uses fixed inertia and learning coefficients	Simple and fast	May converge prematurely	Baseline comparison
TMPSO	Uses additional memory	Balances exploration and exploitation	Increased complexity	Reviewed in comparative study
Adaptive PSO	Dynamically adjusts parameters	Better for dynamic workloads	Needs tuning strategy	Core optimization method
MOPSO	Handles multiple objectives	Useful for CPU, memory, cost, latency	Higher computational cost	Future extension
HPSO	Hierarchical particle structure	Useful for multi-level scheduling	More complex design	Related work
CPSO	Cooperative particle or swarm structure	Improves distributed search	Requires coordination among sub-swarms	Useful for future distributed scheduling
DPSO	Discrete search space	Suitable for placement decisions	May need mapping logic	Relevant to container placement
QPSO	Quantum-behaved search mechanism	Improves global search potential	Requires parameter tuning	Related to advanced PSO research

PSO Variant	Main Feature	Strength	Limitation	Relevance to Thesis
Hybrid PSO	Combines PSO with other methods	Improves robustness	More parameters	Similar direction to DT+APSO
DMS-PSO	Uses multiple swarms	Adapts to dynamic environments	Coordination overhead	Future extension
OBL-PSO	Uses opposition-based learning	Enhances population diversity	Adds computational step	Helps avoid local optima
Chaotic PSO	Uses chaotic sequences	Helps escape premature convergence	May become unstable if not tuned	Relevant to dynamic workload search
Constriction Factor PSO	Uses constriction factor for stability	Improves convergence stability	Requires correct coefficient setting	Related to convergence control

42

The reviewed PSO literature supports the use of Adaptive PSO in this thesis because Docker Swarm workloads are dynamic and require continuous adjustment.

2.8 Container Consolidation, Energy Awareness, and Multi-Resource Scheduling

Container consolidation [6, 34, 39]refers to the process of placing containers on a smaller or more efficient set of nodes while maintaining performance and service-level requirements. Consolidation is closely related to resource allocation, load balancing, and energy efficiency. A good consolidation strategy should avoid both overloading and underutilization.

Liu et al. proposed SLA-driven container consolidation with usage prediction for green cloud computing. Their approach considers current and predicted CPU utilization to detect overutilized and underutilized machines and supports energy-efficient management while maintaining SLA constraints [42]. This work is relevant because it shows that prediction-based resource management can reduce energy usage and improve consolidation decisions.

Recent work has also addressed multi-resource prediction. Awad et al. proposed a multi-resource predictive workload consolidation approach that focuses on resource prediction, overload detection, and underload detection [39]. This direction supports the argument that CPU-only scheduling is insufficient and that memory and other resources must also be considered.

In 2025, Liu et al. proposed a Kubernetes-based containerized workflow resource allocation scheme named CWRA. Their method predicts future workflow tasks during the current task pod's lifecycle and uses dynamic resource scaling to manage high concurrency [12]. The related ARAS work also reported improvements in workflow duration and CPU/memory resource usage by considering future workflow task requests [43].

These studies show that current research is moving from static allocation toward prediction-aware, workflow-aware, and multi-resource-aware scheduling. The present thesis is aligned with this direction because it uses Decision Tree-based suitability analysis and APSO-based optimization to balance CPU and memory utilization in Docker Swarm.

Energy awareness is another important direction in container scheduling. A scheduling strategy that reduces CPU load imbalance can indirectly reduce dynamic power consumption because heavily loaded processors consume more energy than lightly loaded processors. However, the relationship between load balancing and energy consumption is not always straightforward. Consolidating containers onto fewer nodes may save energy by allowing some nodes to become idle, but it may also increase resource contention if too many containers are packed together. Therefore, energy-aware scheduling must balance consolidation with performance. The present thesis focuses primarily on CPU and memory balance, but its framework can be extended to include energy as an additional fitness component.

Multi-resource scheduling is necessary because CPU and memory behave differently. CPU load may fluctuate rapidly, while memory usage may remain stable for longer periods. A node may show low CPU usage but high memory pressure, or high CPU usage with sufficient memory availability. A scheduler that evaluates only one metric may therefore make incorrect decisions. The proposed DT+APSO method uses CPU and memory as core indicators and can be further extended to include network, disk I/O, and energy metrics shown in Table 2.6

TABLE 2.6: Recent Prediction and Consolidation-Oriented Studies

Study	**Year**	**Platform / Scope**	**Main Method**	**Main Contribution**	**Gap for Present Thesis**
Liu et al. [32]	2020	Container cloud	SLA-driven usage prediction	Energy-aware consolidation	Focuses mainly on consolidation, not DT+APSO
Chen and Xiao [19]	2021	Container-based microservices	Multi-objective parallel PSO	Multi-objective container scheduling	Focuses on microservice scheduling rather than Docker Swarm DT+APSO
Ahmad et al. [7]	2022	Container scheduling survey	Taxonomy of scheduling techniques	Broad classification of scheduling methods	Survey only, no implementation
Carrión [13]	2022	Kubernetes scheduling	Taxonomy and issue analysis	Identifies gaps in Kubernetes scheduling	Kubernetes-focused, not Docker Swarm-specific
Awad et al. [39]	2023	Cloud workload consolidation	Multi-resource prediction	Overload/underload detection	VM/cloud focus, less Docker Swarm-specific
Shan et al. / ARAS [43]	2023	Kubernetes workflow	Adaptive resource allocation	Predicts future workflow requests	Workflow-specific, not Swarm-specific

Study	Year	Platform / Scope	Main Method	Main Contribution	Gap for Present Thesis
Xu [22]	2024	Container cloud platform	BP and LSTM resource prediction	ML-based resource scheduling management	Prediction-focused, not APSO-based
Rabieyan et al. [44]	2024	Cloud data centers	Container migration optimization	Optimizes deployment through migration	Migration-focused rather than Swarm placement optimization
Liu et al. [12] / CWRA	2025	Kubernetes workflow engines	Dynamic resource scaling	High-concurrency resource allocation	Kubernetes workflow focus
Present thesis	2025	Docker Swarm	Decision Tree + APSO	Node suitability + adaptive optimization	Focuses on Docker Swarm CPU/memory balancing

46

2.9 Docker Swarm, Kubernetes, and Recent Scheduling Research .

The period from 2022 to 2025 shows rapid growth in container scheduling research. The main trend is toward adaptive, predictive, multi-objective, and AI-assisted scheduling.

Ahmad et al. provided a detailed survey of container scheduling techniques and argued that the container scheduling landscape requires systematic classification and assessment [7]. Carrión presented a taxonomy of Kubernetes scheduling and ongoing issues, showing that even mature orchestration systems still face unresolved scheduling challenges [13]. Senjab et al. reviewed Kubernetes scheduling algorithms and grouped them into generic, multi-objective optimization-based, AI-focused, and autoscaling-enabled categories [14].

Xu proposed a resource scheduling management system for a container cloud platform using resource usage prediction based on the combination of backpropagation and LSTM methods [27]. Rabieyan et al. addressed optimized containerized application deployment through container migration in cloud data centers, emphasizing the importance of software container migration to avoid resource shortages and reduce energy consumption [44].

In 2025, Liu et al. proposed CWRA for Kubernetes workflow engines, combining prediction and dynamic resource scaling [12]. Additional research on container scheduling surveys and future roadmaps highlights that next-generation schedulers should include multi-resource awareness, predictive analytics, reinforcement learning, and cross-platform orchestration [45].

The recent literature also shows that container scheduling is no longer limited to simple node selection. It now includes prediction, autoscaling, multi-objective optimization, energy awareness, carbon awareness, SLA guarantees, and workload-specific scheduling. Kubernetes has received significant research attention because it provides a mature scheduling framework. However, Docker Swarm remains relevant for controlled experiments because it is simpler and allows clearer analysis of

the scheduler behaviour. The gap addressed by this thesis is therefore specific: improving Docker Swarm resource allocation using a hybrid adaptive model, rather than only reviewing scheduling in Kubernetes.

TABLE 2.7: Recent 2022–2025 State-of-the-Art Studies Relevant to Thesis

Author / Source	**Year**	**Main Focus**	**Technique / Direction**	**Relevance to Thesis**
Ahmad et al. [7]	2022	Container scheduling survey	Taxonomy and assessment	Establishes broad scheduling landscape
Carrión [13]	2022	Kubernetes scheduling taxonomy	Scheduling gaps and issues	Supports need for improved scheduling logic
Senjab et al. [14]	2023	Kubernetes scheduling algorithms	Generic, AI, MOO, autoscaling categories	Shows AI and optimization trends
Awad et al. [39]	2023	Workload consolidation	Multi-resource prediction	Supports CPU/memory-aware scheduling
Load prediction-driven scheduling [39]	2023	Container cloud scheduling	Prediction-based scheduling	Supports proactive scheduling
Shan et al. [43]	2023	Kubernetes workflow scheduling	Adaptive resource allocation	Supports future-aware workload scheduling
Xu [22]	2024	Container cloud resource scheduling	BP + LSTM prediction	Shows ML-based resource prediction trend

Author / Source	Year	Main Focus	Technique / Direction	Relevance to Thesis
Rabieyan et al. [44]	2024	Container deployment optimization	Container migration	Supports dynamic resource management
Liu et al. [12]	2025	Kubernetes workflow allocation	CWRA and dynamic scaling	Shows future-aware allocation
Container scheduling roadmap papers	2024–2025	Future scheduling directions	Predictive analytics, RL, multi-resource scheduling	Supports DT+APSO future scope
Present thesis	2025	Docker Swarm resource allocation	Decision Tree + APSO	Fills the Docker Swarm hybrid scheduling gap

TABLE 2.8: Comparative Analysis of Major Literature

No.	Study / Category	Platform	Technique	Main Objective	Key Limitation	Relevance to Proposed DT+APSO
1	Docker performance studies	Docker	Comparative performance analysis	Compare Docker with VM	Limited scheduling focus	Supports container efficiency argument
2	ML microservice scheduling	Microservices	ANN/SVM/CSML	Predict required containers	Service relationships not fully considered	Supports ML scheduling need
3	Big data on containers	Docker/Spark	Lightweight virtualization	Improve big data execution	Workload-specific results	Supports Docker for distributed workloads
4	Elastic Docker	Docker	Vertical elasticity	Runtime resource adjustment	Threshold-dependent	Supports dynamic resource management

No.	Study / Category	Platform	Technique	Main Objective	Key Limitation	Relevance to Proposed DT+APSO
5	Chain-oriented load balancing	Microservices	Multi-priority queues	Reduce chain latency	Less focus on CPU/memory placement	Supports microservice scheduling importance
6	DPWS	Microservices	Dynamic priority queues	Reduce concurrency latency	Limited leaf-node balancing	Shows scheduling complexity
7	Docker Swarm scheduling	Docker Swarm	SLA classes	Prioritized container scheduling	Needs dynamic consolidation	Supports Swarm improvement gap
8	Kubernetes scheduling	Kubernetes	ACO/PSO hybrids	Improve pod placement	Kubernetes-specific	Supports metaheuristic scheduling
9	PSO comparison paper	Docker	PSO, TMPSO, MOPSO, APSO	Compare PSO variants	Lacks full predictive integration	Forms direct foundation
10	JISEM DT+PSO paper	Docker	Decision Tree + PSO	Adaptive resource optimization	Needs broader cluster-level thesis framing	Direct thesis base

52

No.	Study / Category	Platform	Technique	Main Objective	Key Limitation	Relevance to Proposed DT+APSO
11	Ahmad et al. survey	Containers	Taxonomy	Review scheduling techniques	Survey only	Supports literature landscape
12	Senjab et al. survey	Kubernetes	Scheduling taxonomy	Review algorithms	Kubernetes-focused	Supports AI/MOO trend
13	CWRA 2025	Kubernetes workflow	Prediction + scaling	Handle workflow concurrency	Workflow-specific	Supports predictive scheduling
14	Present thesis	Docker Swarm	Decision Tree + APSO	Optimize CPU/memory placement	Future work: SLA/latency/energy	Final contribution

2.10 Research Gaps Identified from Literature

The review identifies several gaps that justify the proposed research.

Lack of adaptability in default Docker Swarm scheduling: Docker Swarm provides basic scheduling, but it does not deeply consider real-time CPU utilization, memory pressure, historical telemetry, task density, or future overload risk. The approved Thesis directly identifies this issue and states that existing Docker Swarm schedulers lack adaptability, leading to poor load balancing and inefficient resource utilization [37].

Limited predictive intelligence in heuristic methods: Traditional algorithms such as Round Robin, Weighted Round Robin, FCFS, and Least Connection are simple but do not learn from workload behaviour. They may work under stable conditions but fail when workload changes dynamically.

Standard PSO may converge prematurely: Standard PSO is useful for optimization, but fixed parameters may cause premature convergence. Reasearcher note that standard PSO can struggle in complex cloud spaces, while Adaptive PSO is more suitable for changing environments

Machine learning alone does not solve placement optimization: Machine learning methods can classify workload states or predict resource requirements, but classification alone does not necessarily provide an optimized placement of containers across nodes. Therefore, ML must be integrated with an optimization algorithm.

Most studies focus on either CPU, memory, latency, or energy separately: Several studies optimize a specific metric, but real Docker Swarm resource allocation requires balancing CPU, memory, resource variance, and node suitability together.

Limited integration of monitoring, prediction, and optimization: Many studies discuss scheduling algorithms without integrating real-time monitoring tools such as Prometheus, cAdvisor, and Grafana. The proposed framework uses cAdvisor,

Prometheus, and Grafana for live telemetry and visualization, making it more practical for real Docker Swarm environments.

Need for hybrid intelligent scheduling: The literature points toward the need for a hybrid model that combines the interpretability of Decision Tree classification with the adaptive search ability of APSO. This is the central contribution of the present thesis.

Need for Docker Swarm-specific research: A significant amount of recent scheduling literature is focused on Kubernetes, while Docker Swarm receives comparatively less attention. Docker Swarm is still valuable for lightweight deployments and controlled experimentation. Therefore, improving Docker Swarm scheduling through an external intelligent framework contributes to a specific and practical research gap.

Need for explainable optimization: Many AI-based scheduling methods can produce good results but are difficult to interpret. The Decision Tree component of the proposed framework adds explainability because it provides rule-based node suitability analysis. This improves the transparency of scheduling decisions compared with black-box approaches.

2.11 Summary of Literature Review

The literature review shows that containerization has transformed cloud application deployment by offering lightweight, portable, and rapidly scalable execution environments. However, as container adoption grows, resource allocation and load balancing become more difficult. Docker Swarm provides a simple orchestration framework, but its default scheduling strategy does not fully consider real-time resource conditions, predictive node suitability, or dynamic workload changes.

Traditional load-balancing algorithms provide simplicity but lack adaptability. Machine learning-based methods improve prediction and classification but may not solve the placement optimization problem. PSO-based methods provide strong search capability but may suffer from premature convergence when parameters are fixed. Recent studies

from 2022 to 2025 show a movement toward predictive, adaptive, multi-objective, and AI-supported scheduling in Kubernetes and containerized cloud systems.

The review also shows that the proposed thesis is aligned with current research trends but remains distinct in its focus. Recent studies have emphasized Kubernetes scheduling, workload prediction, resource scaling, and container migration. In contrast, this thesis concentrates on Docker Swarm-based resource allocation and develops a hybrid classification-optimization framework. This makes the work practically relevant for lightweight Swarm environments and academically relevant for hybrid intelligent scheduling research.

CHAPTER 3

Proposed Methodology

3.1 Introduction

Cloud computing has become the dominant platform for deploying large-scale digital services, web applications, distributed analytics, and microservice-oriented systems. The shift from traditional monolithic applications to cloud-native applications has increased the need for flexible, scalable, and efficient runtime environments. In this context, containerization has emerged as a practical alternative to conventional virtual machine-based deployment [4, 18, 25, 46]. Docker containers package an application together with its dependencies and execute it as an isolated process over the host operating system kernel. Since containers do not require a separate guest operating system for each application instance, they provide faster startup, lower runtime overhead, better portability, simplified deployment, and improved resource efficiency.

Although containerization provides several advantages over hypervisor-based virtualization [1, 4] efficient resource allocation in containerized cloud environments remains a significant research challenge. In a Docker Swarm cluster, multiple containers are deployed as replicated services across manager and worker nodes. The performance of the cluster depends strongly on how efficiently these containers are placed and how evenly workloads are distributed across available nodes [27]. If the scheduling mechanism allocates too many high-load containers to a particular worker node, that node may experience CPU saturation, memory pressure, delayed task execution, and service degradation. At the same time, other nodes may remain underutilized, resulting in poor resource efficiency. Thus, the problem is not merely to

deploy containers but to deploy them intelligently under continuously changing workload conditions.

Default Docker Swarm scheduling mechanisms provide basic container placement using internal heuristics. These mechanisms are useful for general-purpose deployment, but they do not sufficiently consider historical telemetry, predicted future load, node cost, or adaptive optimization. As a result, static scheduling or simple heuristic placement is often insufficient for dynamic cloud workloads. The limitation becomes more visible in microservice-based environments where containers are created, removed, replicated, and scaled in response to fluctuating user demand. Therefore, an intelligent scheduling framework must be capable of observing the current state of the cluster, learning from historical resource usage, predicting potential overload conditions, and adapting its search strategy as the workload changes.

This chapter presents the proposed methodology for optimized resource allocation in Docker container-based cloud environments using a Hybrid Adaptive Particle Swarm Optimization and Decision Tree model. The methodology combines three major ideas. First, the runtime state of Docker containers and worker nodes is continuously monitored using a telemetry layer consisting of cAdvisor, Prometheus, and Grafana. Second, a Decision Tree classifier is used to classify node or container load into low, medium, and high load states based on CPU usage, memory usage, and resource allocation features. Third, Adaptive Particle Swarm Optimization (APSO) is used to search for an improved container placement configuration by dynamically adjusting inertia weight and using local and global best positions to guide scheduling decisions.

The proposed methodology is designed to overcome three important limitations of conventional approaches. The first limitation is the lack of predictive intelligence in default Swarm scheduling. The second limitation is the tendency of standard PSO to suffer from premature convergence when the search space is dynamic. The third limitation is the inability of non-adaptive algorithms to respond to rapid workload variation. By integrating Decision Tree classification with APSO, the proposed framework introduces both predictive node suitability estimation and adaptive optimization-based placement.

The chapter is organized as follows. Section 3.2 explains the design rationale of the proposed methodology. Section 3.3 formulates the resource allocation problem. Section 3.4 presents the system architecture. Section 3.5 describes the monitoring and telemetry framework. Section 3.6 presents the Decision Tree classification model. Section 3.7 explains the APSO model. Section 3.8 defines the fitness function and constraints. Section 3.9 presents the original proposed algorithm. Section 3.10 explains the algorithm using a worked numerical example. Section 3.11 to Section 3.16 describes the automated orchestration pipeline. Finally, Section 3.17 summarizes the chapter.

3.2 Design Rationale of the Proposed Methodology

The proposed methodology is motivated by the need for an adaptive and intelligent scheduling mechanism for Docker Swarm-based cloud environments. A container scheduling method must not only consider the current availability of resources but also anticipate how a node may behave after additional containers are assigned to it. In a real deployment, resource demand is not constant. CPU-intensive services may create short bursts of processor usage, memory-intensive services may gradually increase memory pressure, and mixed workloads may generate irregular utilization patterns. Therefore, a suitable methodology must combine monitoring, prediction, optimization, and orchestration.

A purely heuristic scheduler is simple and fast, but it cannot learn workload patterns. A purely machine learning-based scheduler can classify load states but may not search the combinatorial placement space effectively. Similarly, a purely metaheuristic optimizer such as standard PSO can search for better placement configurations, but it may become trapped in local optima and may not use historical resource behavior effectively.

The proposed methodology combines the strengths of both machine learning and metaheuristic optimization.The Decision Tree model provides interpretable classification of load states, while APSO provides adaptive optimization for container placement.

The use of a Decision Tree is suitable for this research because the scheduling decision is influenced by threshold-like behavior in resource metrics. For example, a node with CPU utilization above 80% and memory usage above 75% should generally be treated as high load and should not receive additional containers unless no alternative exists. Similarly, a node with CPU below 30% and memory below 25% can be considered a suitable candidate for receiving additional containers. Decision Trees naturally represent such rules using binary splits and are computationally lightweight enough for online or near-real-time decision support.

The use of Adaptive PSO is justified because container placement is an optimization problem. A placement configuration may be represented as a particle, and the quality of that configuration can be evaluated using a fitness function based on CPU usage, memory usage, and resource allocation. Standard PSO uses fixed parameters, whereas adaptive PSO modifies the inertia weight during iterations. At early iterations, a higher inertia weight promotes exploration of the search space. At later iterations, a lower inertia weight encourages exploitation around promising solutions. This balance is important in dynamic Docker environments, where the search process must avoid premature convergence while still reaching a useful placement decision quickly.

The proposed approach emphasizes practical implementation and deployability. It is designed to work externally with Docker Swarm using monitoring APIs and orchestration commands. It does not require modification of the Docker Engine or Swarm manager. This design choice makes the approach suitable for practical cloud environments where modifying the core orchestration engine is difficult or undesirable.

3.3 Problem Formulation

The resource allocation problem in Docker Swarm can be formulated as a constrained optimization problem. Let the Docker Swarm cluster contain a set of worker nodes:

$$N = \{n_1, n_2, \ldots, n_m\} \tag{3.1}$$

where m is the number of available nodes. Let the set of containers or service replicas be represented as:

$$C = \{c_1, c_2, \ldots, c_q\} \tag{3.2}$$

where q is the number of containers to be scheduled. Each node n_j has finite CPU capacity, memory capacity, storage capacity, and network capacity. In this research, the main resource dimensions considered for optimization are CPU and memory, because these are the most direct indicators of container load and were also used as primary metrics in the experimental work.

The allocation of containers to nodes can be represented using a binary decision variable:

$$A_{ij} = \begin{cases} 1, & \text{if container } c_i \text{ is assigned to node } n_j \\ 0, & \text{otherwise} \end{cases} \tag{3.3}$$

Each container must be assigned to exactly one node. Therefore, the allocation constraint is:

$$\sum_{j=1}^{m} A_{ij} = 1, \qquad \forall i \in \{1, 2, \ldots, q\} \tag{3.4}$$

The total CPU utilization of node n_j after allocation is represented as:

$$CPU_j = \sum_{i=1}^{q} A_{ij} \cdot CPU(c_i) \tag{3.5}$$

Similarly, the total memory utilization of node n_j is:

$$MEM_j = \sum_{i=1}^{q} A_{ij} \cdot MEM(c_i) \tag{3.6}$$

where $CPU(c_i)$ and $MEM(c_i)$ denote the CPU and memory demands of container c_i. The node capacities are represented by CPU_j^{max} and MEM_j^{max}. Therefore, the capacity constraints are:

$$CPU_j \leq CPU_j^{max}, \qquad \forall j \in \{1,2,\ldots,m\} \tag{3.7}$$

$$MEM_j \leq MEM_j^{max}, \qquad \forall j \in \{1,2,\ldots,m\} \tag{3.8}$$

The goal of the scheduler is to minimize the imbalance in CPU and memory utilization across nodes. Average CPU utilization is calculated as:

$$\overline{CPU} = \frac{1}{m}\sum_{j=1}^{m} CPU_j \tag{3.9}$$

The CPU imbalance can be measured using standard deviation:

$$CPU_{imbalance} = \sqrt{\frac{1}{m}\sum_{j=1}^{m}(CPU_j - \overline{CPU})^2} \tag{3.10}$$

Similarly, average memory utilization is:

$$\overline{MEM} = \frac{1}{m}\sum_{j=1}^{m} MEM_j \tag{3.11}$$

and memory imbalance is:

$$MEM_{imbalance} = \sqrt{\frac{1}{m}\sum_{j=1}^{m}(MEM_j - \overline{MEM})^2} \tag{3.12}$$

The generalized load balancing optimization objective can therefore be written as:

$$\min F = \alpha \cdot CPU_{imbalance} + \beta \cdot MEM_{imbalance} + \gamma \cdot Cost_{DT} \tag{3.13}$$

where α, β, and γ are weighting factors, and $Cost_{DT}$ is the cost value derived from the Decision Tree load classification. In the implemented algorithm, the fitness function is expressed in normalized resource terms, as shown later in Section 3.8.

3.4 Proposed System Architecture

The proposed architecture Figure 3.1 follows a layered design in which monitoring, prediction, optimization, and orchestration are separated but connected through a feedback-driven workflow. The major architectural layers are:

1. Docker Swarm cluster layer.
2. Application workload layer.
3. Monitoring and telemetry collection layer.
4. Decision Tree classification layer.
5. APSO optimization layer.
6. Orchestration and deployment layer.
7. Result storage and visualization layer.

The Docker Swarm cluster layer consists of one manager node and multiple worker nodes. The manager node maintains cluster state, manages service deployment, and coordinates scheduling decisions. Worker nodes execute containerized workloads. In the experimental setup, the cluster is configured using Ubuntu-based nodes deployed in a virtualized environment. The thesis describes a three-node Docker Swarm testbed with one master and two worker nodes. The testbed is summarized in Table 3.1.

TABLE 3.1: Docker Swarm Experimental Testbed

Node	IP Address	CPU	RAM	Memory/Storage
Master	10.0.4.42	8 Cores	8 GB	20 GB
Worker 1	10.0.4.43	8 Cores	8 GB	20 GB
Worker 2	10.0.4.49	8 Cores	8 GB	20 GB

The application workload layer contains the containerized benchmark application. A CPU-intensive factorial or Fibonacci-type workload is used to generate measurable CPU pressure. Controlled memory operations are also introduced to observe memory-aware balancing. This workload is suitable because it creates repeated computational demand and allows the scheduler to be evaluated under non-trivial load conditions.

The monitoring layer uses cAdvisor to expose per-container metrics and Prometheus to scrape and store time-series metrics. Grafana is used to visualize resource utilization patterns. The Decision Tree classification layer receives processed telemetry and classifies node or container load. The APSO optimization layer uses the classified load state and resource metrics to generate optimized placement. Finally, the orchestration layer applies placement decisions through Docker commands or APIs summarised in Figure 3.1.

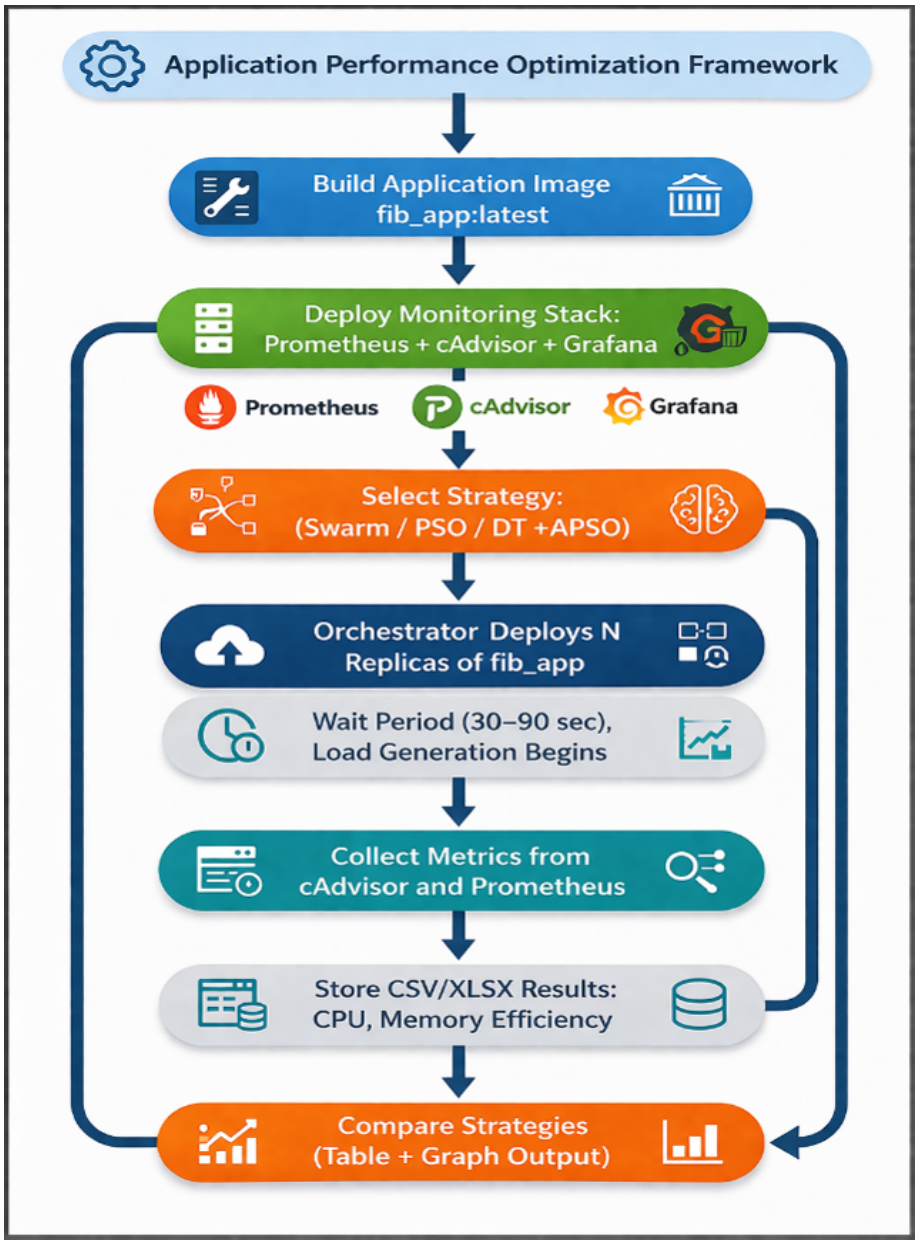


FIGURE 3.1: Proposed Application Performance Optimization Framework for Docker Swarm

3.5 Application Workload Design

The workload used in the proposed methodology is designed to evaluate container scheduling under measurable CPU and memory demand. The primary application container is a CPU-intensive computation service. The workload may be executed as a replicated Docker service, where each replica performs repeated factorial or Fibonacci

computations. Such recursive or iterative numerical workloads generate CPU spikes and sustained processor activity, making them suitable for load balancing evaluation.

The application workload is built into a Docker image, for example:

```
docker build -t fib_app:latest .
```

After image creation, the service can be deployed in Docker Swarm with a defined number of replicas. Each replica is scheduled on one of the available worker nodes. During execution, cAdvisor captures container-specific resource usage, and Prometheus stores the time-series data.

The workload design supports the following types of analysis:

1. CPU saturation analysis.
2. Memory utilization analysis.
3. Node-wise load distribution.
4. Scheduler responsiveness.
5. APSO convergence behavior.
6. Decision Tree node suitability classification.
7. Comparison among default Swarm, standard PSO, and DT+APSO scheduling.

In addition to CPU pressure, memory allocation operations are used to create controlled memory churn. This is important because a scheduler optimized only for CPU may still produce poor memory distribution. Therefore, the proposed methodology considers both CPU and memory as primary decision variables.

3.6 Monitoring and Telemetry Framework

The proposed methodology depends on accurate and timely resource monitoring. Without runtime telemetry, the scheduling model cannot evaluate the current state of the cluster or predict future load conditions. The monitoring stack consists of cAdvisor, Prometheus, and Grafana.

cAdvisor runs on worker nodes and provides detailed container-level statistics. It captures CPU usage, memory usage, filesystem usage, and network activity for running containers. Prometheus scrapes these metrics periodically and stores them as time-series data. Grafana provides dashboards for visualization and supports interpretation of scheduling performance. Table 3.2 shows telemetry collection process is performed in repeated cycles. First, the orchestrator deploys the selected workload. Second, a stabilization period is allowed so that the workload begins generating measurable resource usage. Third, Prometheus queries are executed to extract CPU and memory values. Fourth, the collected metrics are exported to CSV or XLSX files for further analysis. Finally, these metrics are supplied to the Decision Tree and APSO components.

The Decision Tree component provides a predictive and interpretable classification mechanism for identifying node suitability. The purpose of the Decision Tree is not to replace optimization but to guide the optimization process. It converts raw telemetry into meaningful load categories such as low, medium, and high. These categories help APSO avoid overloaded nodes and prefer nodes with lower predicted load.

The key metrics used in this research are listed in Table 3.2.

TABLE 3.2: Monitoring Metrics Used in the Proposed Methodology

Metric	**Purpose in Scheduling**
`container_cpu_usage_seconds_total`	Measures cumulative CPU usage of containers and supports CPU utilization calculation.
`container_memory_usage_bytes`	Measures memory consumed by each container and supports memory pressure detection.
`engine_daemon_container_states_running_total`	Indicates the number of running containers and helps estimate task density.
`node_cpu_seconds_total`	Captures node-level CPU state and supports node utilization analysis.
`node_load1`	Represents short-term system load and assists in detecting overloaded nodes.
Network I/O metrics	Used to observe communication overhead and container traffic behavior.

3.6.1 Feature Representation

Each node or container state is represented using a feature vector:

$$\mathbf{x} = [x_1, x_2, x_3, x_4] \tag{3.14}$$

where:

$$x_1 = \text{CPU utilization } (\%), \tag{3.15}$$

$$x_2 = \text{Memory utilization } (\%), \tag{3.16}$$

$$x_3 = \text{Active container count}, \tag{3.17}$$

$$x_4 = \text{Resource allocation density}. \tag{3.18}$$

For a simplified CPU-memory classification model, the feature vector is represented as:

$$\mathbf{x} = [x_1, x_2] \tag{3.19}$$

where x_1 denotes CPU usage and x_2 denotes memory usage. The target class is:

$$y \in \{Low, Medium, High\} \tag{3.20}$$

3.6.2 CART Splitting Criterion

The Classification and Regression Tree (CART) model uses binary splitting. At each node of the tree, the algorithm selects the feature and threshold that produce the best separation of classes. The Gini impurity of dataset D is computed as:

$$Gini(D) = 1 - \sum_{k=1}^{K} p_k^2 \tag{3.21}$$

where p_k is the probability of class k in dataset D, and K is the number of classes. In this research, $K = 3$ because the load classes are low, medium, and high.

For a candidate split that divides dataset D into left subset D_L and right subset D_R, the weighted Gini impurity is:

$$Gini_{split} = \frac{|D_L|}{|D|} Gini(D_L) + \frac{|D_R|}{|D|} Gini(D_R) \tag{3.22}$$

The optimal split is selected by minimizing the weighted Gini impurity:

$$(j^*, s^*) = \arg\min_{j,s} Gini_{split}(j, s) \tag{3.23}$$

where j denotes the selected feature and s denotes the splitting threshold.

3.6.3 Decision Rules for Load Classification

The load classification rules used in this research are represented as:

$$f(x_1, x_2) = \begin{cases} High, & x_1 > 80 \wedge x_2 > 75, \\ Low, & x_1 < 30 \wedge x_2 < 25, \\ Medium, & \text{otherwise.} \end{cases} \tag{3.24}$$

where x_1 is CPU utilization and x_2 is memory utilization. These thresholds are used to support interpretable node classification and to convert resource observations into scheduling cost values.

The corresponding decision regions are:

$$R_1 = \{x : x_1 > 80 \land x_2 > 75\} \Rightarrow High, \tag{3.25}$$

$$R_2 = \{x : x_1 < 30 \land x_2 < 25\} \Rightarrow Low, \tag{3.26}$$

$$R_3 = \text{Remaining region} \Rightarrow Medium. \tag{3.27}$$

For a leaf region R_m, the class probability is:

$$P(y = k \mid x \in R_m) = \frac{N_k}{N_m} \tag{3.28}$$

where N_k is the number of samples of class k in region R_m, and N_m is the total number of samples in that region. The final predicted class is:

$$\hat{y} = \arg\max_k P(y = k \mid x) \tag{3.29}$$

3.6.4 Decision Tree Cost Mapping

The output of the Decision Tree is converted into a numeric cost so that it can be included in the APSO fitness function. The cost mapping is:

$$Cost = \begin{cases} 3, & High, \\ 2, & Medium, \\ 1, & Low. \end{cases} \tag{3.30}$$

A higher cost indicates a less suitable node for additional container placement. A lower cost indicates a more suitable node.

3.7 Adaptive Particle Swarm Optimization Model

Particle Swarm Optimization is a population-based optimization method inspired by collective intelligence. In the context of Docker container scheduling, each particle represents a candidate placement solution. The particle position describes how containers or replicas are distributed across nodes, and the particle velocity determines how the placement solution changes during the optimization process.

Let P be the number of particles and I_{max} be the maximum number of iterations. Let $x_i(t)$ denote the position of particle i at iteration t, and let $v_i(t)$ denote its velocity. The personal best position of particle i is denoted by p_i, and the global best position of the swarm is denoted by g.

3.7.1 Velocity Update

The velocity update equation used in the proposed algorithm is:

$$v_{ij}(t+1) = w \cdot v_{ij}(t) + c_1 \cdot r_1 \cdot (p_{ij} - x_{ij}) + c_2 \cdot r_2 \cdot (g_j - x_{ij}) \tag{3.31}$$

where $v_{ij}(t+1)$ is the updated velocity of particle i in dimension j, w is the inertia weight, c_1 is the cognitive coefficient, c_2 is the social coefficient, r_1 and r_2 are random values in the range $[0, 1]$, p_{ij} is the local best value, and g_j is the global best value.

The first term, $w \cdot v_{ij}(t)$, controls the influence of the previous velocity. The second term, $c_1 \cdot r_1 \cdot (p_{ij} - x_{ij})$, pulls the particle toward its own best-known solution. The third term, $c_2 \cdot r_2 \cdot (g_j - x_{ij})$, pulls the particle toward the best solution found by the swarm. Together, these components enable the search process to combine individual learning and social learning.

3.7.2 Position Update

After velocity update, the particle position is updated as:

$$x_{ij}(t+1) = x_{ij}(t) + v_{ij}(t+1) \tag{3.32}$$

Since container allocation is discrete in practice, the updated position is corrected to an integer value and constrained within the permissible node range. If a particle position exceeds the node boundary, it is adjusted using boundary conditions.

3.7.3 Adaptive Inertia Weight

The adaptive inertia weight used in the original algorithm is:

$$w = w_{max} - \frac{(w_{max} - w_{min})}{I_{max}} \times iteration \tag{3.33}$$

where w_{max} is the maximum inertia weight, w_{min} is the minimum inertia weight, and *iteration* is the current iteration number. In the experimental configuration, inertia weight is initialized near 0.9 and decreases toward 0.4.

This adaptation allows the swarm to explore widely during early iterations and gradually focus on promising solutions in later iterations. Larger inertia values improve exploration, while smaller inertia values improve exploitation.

3.8 Fitness Function and Optimization Objective

The fitness function evaluates the quality of each particle or placement solution. The original algorithm defines the fitness function as:

$$f = \sum_{i=1}^{N} \left(\frac{CPU_i}{Total\ CPU} + \frac{MEMORY_i}{Total\ MEMORY} + \frac{RESOURCE_i}{Total\ RESOURCE} \right) \tag{3.34}$$

The goal is to minimize the fitness value. A lower fitness value indicates that the selected placement produces lower relative CPU usage, lower memory usage, and better normalized resource consumption.

To integrate Decision Tree load classification, the enhanced fitness function can be represented as:

$$Fitness = \sum_{i=1}^{N} \left(\frac{CPU_i}{CPU_{total}} + \frac{MEM_i}{MEM_{total}} + \lambda \cdot Cost_i \right) \tag{3.35}$$

where $Cost_i$ is the Decision Tree-derived load cost for node i, and λ is a weighting factor controlling the influence of predicted load.

The optimization objective is min Fitness subject to:

$$CPU_i \leq CPU_i^{max} \tag{3.36}$$

$$MEM_i \leq MEM_i^{max} \tag{3.37}$$

optimisation

$$\sum_{j=1}^{m} A_{ij} = 1 \tag{3.38}$$

These constraints ensure that container placement does not exceed node capacity and that each container is assigned to one node.

3.9 Proposed Algorithm

Following is the proposed algorithm using Decesion Tree and Adaptive PSO Algorithm.

Algorithm 1 Hybrid Adaptive PSO and Decision Tree-Based Load Distribution Algorithm

Step 1: Initialization

1: Define the number of particles P, maximum iterations $I_{\max}$, inertia weight w, cognitive coefficient C_1, social coefficient C_2, and population size for classification C_{pop}.

2: Initialize the positions and velocities of particles randomly within the permissible range.

3: Initialize the personal best position of each particle.

4: Identify and set the global best position.

Step 2: Classification for Load Prediction

5: Train a Decision Tree classifier.

6: Collect historical data with features such as CPU usage, memory usage, and resource allocation.

7: Use load labels such as low, medium, and high for training the classifier.

8: Predict the load condition of each container using the trained classifier based on current resource features.

Step 3: Fitness Evaluation

9: Define the fitness function as:

$$f = \sum_{i=1}^{N} \left(\frac{CPU_i}{\text{Total CPU}} + \frac{MEMORY_i}{\text{Total MEMORY}} + \frac{RESOURCE_i}{\text{Total RESOURCE}} \right)$$

10: Calculate the fitness value for each particle.

Step 4: Particle Update

11: Adaptively update the inertia weight as:

$$w = w_{\max} - \frac{(w_{\max} - w_{\min})}{I_{\max}} \times t$$

12: Update the velocity of each particle as:

$$v_{ij}(t+1) = w \cdot v_{ij}(t) + C_1 \cdot r_1 \cdot (p_{ij} - x_{ij}) + C_2 \cdot r_2 \cdot (g_j - x_{ij})$$

13: Update the position of each particle as:

$$x_{ij}(t+1) = x_{ij}(t) + v_{ij}(t+1)$$

14: Apply boundary conditions to ensure that particle positions remain within the permissible range.

Step 5: Update Personal and Global Best Positions

15: Evaluate the fitness value of each particle's new position.
16: Update the personal best position if the new position provides a better fitness value.
17: Update the global best position if the new position provides the best overall fitness value.
Step 6: Termination
18: **if** maximum iterations $I_{\max}$ are reached or the fitness value converges **then**
19: Terminate the algorithm.
20: **end if**
21: Output the global best position as the optimal load distribution.

3.9.1 Stepwise Explanation of the Proposed Algorithm

The proposed algorithm consists of six major phases: initialization, Decision Tree classification, fitness evaluation, particle update, local and global best update, and termination.

3.9.2 Initialization Phase

During initialization, the algorithm defines the basic PSO parameters. The number of particles P determines how many candidate scheduling solutions are explored in parallel. The maximum number of iterations I_{max} defines the stopping limit of the optimization process. The inertia weight w controls the influence of previous velocity on the current particle movement. The cognitive coefficient C_1 controls the influence of the particle's own best solution, while the social coefficient C_2 controls the influence of the global best solution. The classification population size C_{pop} defines the data population used by the Decision Tree classifier.

Each particle is initialized randomly within the permissible range. In the context of Docker Swarm scheduling, a particle position represents a possible mapping of containers to available nodes. For example, if there are three worker nodes, a particle dimension may take values 1, 2, or 3, representing assignment to Worker 1, Worker 2, or Worker 3 respectively.

3.9.3 Classification for Load Prediction

After initialization, a Decision Tree classifier is trained using historical resource data. The input features include CPU usage, memory usage, and resource allocation. The output labels represent load states such as low, medium, and high. Once trained, the classifier predicts the current load condition of each node or container.

The purpose of this classification phase is to introduce predictive intelligence into the scheduling mechanism. Instead of selecting nodes only from current resource values, the scheduler uses classified load state to estimate suitability. A high-load node receives a higher cost, while a low-load node receives a lower cost.

3.9.4 Fitness Evaluation

Each particle is evaluated using the defined fitness function. The fitness function combines normalized CPU usage, normalized memory usage, and normalized resource usage. A lower fitness value indicates a more suitable allocation. Fitness evaluation allows the optimizer to compare multiple candidate placements and identify better solutions.

3.9.5 Particle Update Phase

The algorithm updates inertia weight adaptively using the iteration count. At the beginning of the optimization, inertia is high, which allows particles to explore different placement configurations. As iterations progress, inertia decreases, encouraging particles to exploit promising regions of the search space.

Velocity and position are then updated using PSO equations. After position update, boundary conditions ensure that particle positions remain within valid node assignment ranges.

3.9.6 Local and Global Best Update

Each particle compares its new fitness value with its previous local best. If the new solution is better, the local best is updated. The algorithm also compares all local best values and updates the global best if a better overall solution is found. The global best represents the best container placement discovered so far.

3.9.7 Termination Phase

The algorithm terminates when the maximum number of iterations is reached or when the fitness value converges. The final global best position is returned as the optimal load distribution.

3.10 Worked Example of the Proposed Algorithm

This section explains the working of the proposed algorithm using a simplified example. Consider a Docker Swarm cluster with three worker nodes and three container replicas to be scheduled.

TABLE 3.3: Initial Node Resource Utilization for Worked Example

Node	CPU Usage (%)	Memory Usage (%)	Resource Usage (%)
Node 1	85	80	90
Node 2	40	45	50
Node 3	20	18	25

As illustrated in Table 3.3, according to the Decision Tree rules, Node 1 is classified as high load because CPU usage is greater than 80% and memory usage is greater than 75%. Node 3 is classified as low load because CPU usage is below 30% and memory usage is below 25%. Node 2 is classified as medium load because it does not satisfy either high-load or low-load conditions.

The cost mapping is therefore:

$$Cost(Node1) = 3, \tag{3.39}$$

$$Cost(Node2) = 2, \tag{3.40}$$

$$Cost(Node3) = 1. \tag{3.41}$$

Assume the total CPU, memory, and resource reference values are each normalized to 100. The original fitness calculation for each node is as follows.

For Node 1:

$$f_1 = \frac{85}{100} + \frac{80}{100} + \frac{90}{100} = 2.55 \tag{3.42}$$

For Node 2:

$$f_2 = \frac{40}{100} + \frac{45}{100} + \frac{50}{100} = 1.35 \tag{3.43}$$

For Node 3:

$$f_3 = \frac{20}{100} + \frac{18}{100} + \frac{25}{100} = 0.63 \tag{3.44}$$

Since Node 3 has the minimum fitness value, it is the most suitable node for receiving a new container replica. This decision is also consistent with the Decision Tree classification because Node 3 is classified as low load.

To illustrate the APSO update, assume a particle represents a container placement decision. Let the current position of a particle in one dimension be:

$$x_{ij}(t) = 2 \tag{3.45}$$

This means the current particle assigns the container to Node 2. Assume:

$$v_{ij}(t) = 0.5, \tag{3.46}$$

$$w = 0.7, \tag{3.47}$$

$$c_1 = 2.0, \tag{3.48}$$

$$c_2 = 2.0, \tag{3.49}$$

$$r_1 = 0.4, \tag{3.50}$$

$$r_2 = 0.6, \tag{3.51}$$

$$p_{ij} = 3, \tag{3.52}$$

$$g_j = 3. \tag{3.53}$$

As per Equation 3.31 the velocity update becomes as:

$$v_{ij}(t+1) = 0.7(0.5) + 2.0(0.4)(3-2) + 2.0(0.6)(3-2) \tag{3.54}$$

$$v_{ij}(t+1) = 0.35 + 0.8 + 1.2 = 2.35 \tag{3.55}$$

The position update as per Equation 3.32 is:

$$x_{ij}(t+1) = 2 + 2.35 = 4.35 \tag{3.56}$$

Since only three nodes are available, the boundary condition is applied. The corrected node assignment becomes:

$$x_{ij}(t+1) = 3 \tag{3.57}$$

Thus, after APSO update and boundary correction, the particle moves from Node 2 toward Node 3, which is the global best node according to the current fitness evaluation. This example demonstrates how the proposed algorithm uses Decision Tree load prediction and APSO movement rules to shift placement decisions away from overloaded nodes and toward underutilized nodes.

3.11 Automated Orchestrator Pipeline

The proposed methodology is implemented using an automated orchestrator pipeline. The orchestrator is responsible for executing experiments in a repeatable and controlled manner. It performs deployment, waiting, data collection, removal, and comparison steps across all scheduling strategies.

The main functions of the orchestrator are:

1. Build the Docker application image.
2. Deploy the monitoring stack using Docker Swarm.
3. Select the scheduling strategy.
4. Deploy a specific number of replicas.
5. Wait for workload stabilization.
6. Query Prometheus and cAdvisor metrics.
7. Store CPU and memory results in CSV/XLSX format.
8. Execute APSO and Decision Tree scheduling.
9. Compare default Swarm, PSO, and DT+APSO strategies.
10. Generate result tables and visualizations.

The orchestrator ensures that all strategies are evaluated under the same workload and infrastructure conditions. This is important for fair comparison because changes in workload duration, monitoring interval, or cluster state may otherwise affect the results.

3.12 Scheduling Strategies Used for Comparison

The proposed methodology compares three scheduling strategies.

3.12.1 Default Docker Swarm Scheduler

The default Docker Swarm scheduler is used as the baseline. It deploys containers using built-in placement mechanisms. However, it does not use historical telemetry or predictive classification. Therefore, it may produce uneven workload distribution when the workload changes dynamically.

3.12.2 Standard PSO Scheduler

The standard PSO scheduler uses particle-based optimization to search for improved replica placement. Each particle represents a candidate allocation. The fitness function evaluates CPU and memory utilization. However, standard PSO uses fixed parameters and lacks predictive node classification. As a result, it may converge prematurely in dynamic environments.

3.12.3 Proposed DT+APSO Scheduler

The proposed scheduler as shown in Table 3.4 combines Decision Tree classification with Adaptive PSO. The Decision Tree component predicts node suitability, and APSO performs adaptive optimization. This hybrid strategy improves scheduling intelligence, avoids overloaded nodes, and adapts to changing workloads.

TABLE 3.4: Comparison of Scheduling Strategies Used in the Methodology

Strategy	**Main Principle**	**Limitation / Strength**
Default Docker Swarm	Uses built-in scheduling heuristics	Simple and easy to deploy but lacks predictive and adaptive intelligence.
Standard PSO	Searches for optimized placement using particles	Better than static scheduling but may suffer from premature convergence due to fixed parameters.
Proposed DT+APSO	Combines Decision Tree load prediction with adaptive swarm optimization	Provides predictive, adaptive, and telemetry-driven scheduling for dynamic container workloads.

3.13 Performance Evaluation Metrics

The proposed methodology evaluates performance using the following metrics.

3.13.1 CPU Utilization

CPU utilization measures the percentage of processor capacity used by containers and nodes. Lower and more balanced CPU utilization indicates better scheduling efficiency.

3.13.2 Memory Utilization

Memory utilization measures the amount of memory consumed by containers. Stable memory utilization indicates that the scheduler avoids memory pressure and prevents excessive concentration of memory-intensive containers on a single node.

3.13.3 Node-Level Variance

Variance measures the degree of imbalance among nodes. Lower variance indicates better load balancing.

$$Variance = \frac{1}{m}\sum_{j=1}^{m}(L_j - \overline{L})^2 \tag{3.58}$$

where L_j is the load of node j and $\overline{L}$ is the average load.

3.13.4 Execution Time

Execution time measures the time required to complete the scheduling and optimization process.

3.13.5 Convergence Speed

Convergence speed indicates how quickly the optimizer reaches a stable or near-optimal solution.

$$ConvergenceRate = |F(t+1) - F(t)| \tag{3.59}$$

where $F(t)$ is the fitness value at iteration t.

3.13.6 Load Distribution Fairness

Fairness measures whether workload is evenly distributed across worker nodes. A scheduler with high fairness prevents both overload and underutilization.

3.14 Methodology Flowchart

The complete methodology can be summarized as a closed-loop workflow as shown in Figure 3.2 First, workload containers are deployed in Docker Swarm. Second, cAdvisor and Prometheus collect metrics. Third, the Decision Tree model classifies node load. Fourth, APSO computes optimized placement. Fifth, the orchestrator applies scheduling decisions. Finally, monitoring continues and the optimization cycle repeats.

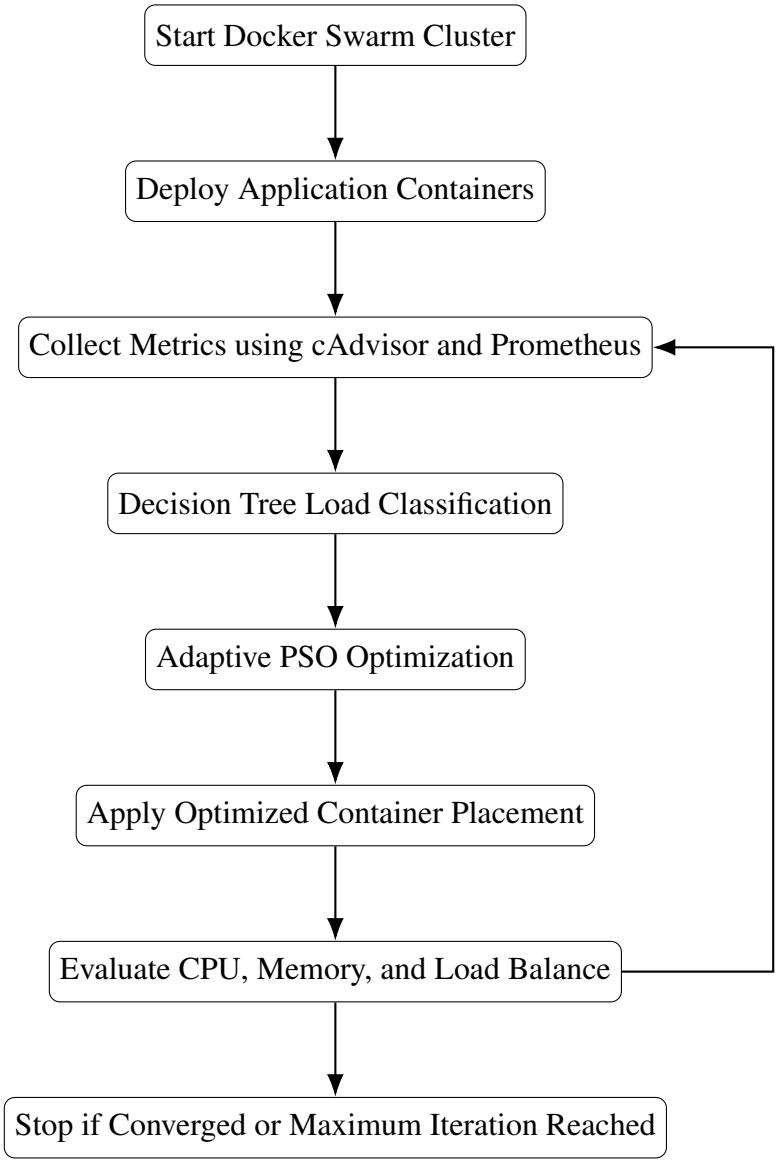


FIGURE 3.2: Flowchart of the Proposed DT+APSO Scheduling Methodology

3.15 Complexity Analysis

The computational complexity of the proposed method depends on the number of particles, number of iterations, number of nodes, and Decision Tree classification cost. Let P denote the number of particles, I_{max} denote the number of iterations, and m denote the number of nodes. For each iteration, the algorithm evaluates particles across nodes. Therefore, the APSO complexity is approximately:

$$O(P \times I_{max} \times m) \tag{3.60}$$

The Decision Tree classification cost for a trained tree is proportional to the tree depth d:

$$O(d) \tag{3.61}$$

Thus, the combined scheduling complexity can be represented as:

$$O(P \times I_{max} \times m + d) \tag{3.62}$$

Because Decision Tree prediction is fast and the number of Swarm nodes in the experimental setup is limited, the main computational cost comes from APSO fitness evaluation. However, the overhead is acceptable because the scheduler operates externally and does not interfere with Docker Engine internals.

3.16 Expected Benefits of the Proposed Methodology

The proposed methodology provides several technical benefits.

First, the use of Decision Tree classification improves node suitability analysis. Instead of relying only on raw CPU or memory values, the scheduler receives a classified load state that reflects resource condition more clearly.

Second, Adaptive PSO improves the search process by dynamically adjusting inertia weight. This reduces the risk of premature convergence and improves the balance between exploration and exploitation.

Third, the monitoring layer ensures that scheduling decisions are based on live telemetry. This makes the methodology suitable for dynamic cloud environments where workload changes continuously.

Fourth, the orchestrator enables repeatable experiments and fair comparison across default Swarm, standard PSO, and proposed DT+APSO scheduling.

Fifth, the proposed methodology is practical because it integrates externally with Docker Swarm and does not require changes to the Docker Engine or Swarm manager.

3.17 Chapter Summary

This chapter presented the proposed methodology for optimized resource allocation in Docker container-based cloud environments using a Hybrid Adaptive Particle Swarm Optimization (APSO) and Decision Tree model. It explained the need for intelligent scheduling in Docker Swarm environments, where dynamic workloads may lead to CPU saturation, memory pressure, node overloading, and inefficient resource utilization. The resource allocation problem was formulated as a constrained optimization problem, followed by the description of the proposed architecture consisting of the Docker Swarm cluster, application workload, monitoring framework, Decision Tree classifier, APSO optimizer, and automated orchestration pipeline. The monitoring layer uses cAdvisor, Prometheus, and Grafana to collect real-time CPU, memory, container, and node-level metrics for scheduling decisions. The Decision Tree model was described using Gini impurity, class probability, decision regions, and cost mapping to classify nodes into low, medium, and high load states, while the

APSO model was explained through velocity update, position update, and adaptive inertia weight equations.

The original algorithm from the thesis was preserved and explained with its major stages, including initialization, load prediction, fitness evaluation, particle update, local and global best update, and termination. A numerical example demonstrated how the algorithm identifies overloaded, medium-load, and low-load nodes and selects the most suitable node using the fitness function. The chapter also discussed the automated orchestrator pipeline, comparative scheduling strategies, performance metrics, methodology flowchart, complexity analysis, and expected benefits. Overall, the proposed methodology provides a practical, adaptive, and predictive framework that combines real-time telemetry, Decision Tree-based node suitability analysis, and APSO-based optimization to improve resource utilization, reduce CPU and memory imbalance, avoid overloaded nodes, and enhance Docker Swarm cluster stability.

CHAPTER 4

Experimental Setup and Implementation

4.1 Introduction

The previous chapter presented the proposed Hybrid Adaptive Particle Swarm Optimization and Decision Tree (DT+APSO) methodology for optimized resource allocation in Docker container-based cloud environments. This chapter describes the experimental setup used to validate the proposed methodology. The focus of this chapter is on the implementation environment, Docker Swarm cluster configuration, workload design, monitoring framework, Decision Tree configuration, Adaptive PSO parameter setting, data collection procedure, and performance evaluation metrics.

The experimental setup is designed to evaluate the proposed scheduling model under controlled and measurable workload conditions. Since containerized applications may generate different resource usage patterns depending on CPU demand, memory allocation, request rate, and replica count, the experimental design must support continuous monitoring and fair comparison. Therefore, the implementation combines Docker container execution, Docker Swarm orchestration, runtime metric collection, Decision Tree-based load classification, Adaptive PSO-based optimization, and result visualization.

The validation is carried out in two phases. The first phase uses a Docker Desktop-based local environment with five containers to observe container-level resource optimization. The second phase uses a Docker Swarm-based distributed cluster consisting of one master node and two worker nodes to evaluate node-wise scheduling and load balancing.

This two-phase evaluation provides both container-level and cluster-level understanding of the proposed DT+APSO framework.

4.2 Experimental Design

The experimental design follows a controlled evaluation approach. The same workload pattern is executed under different scheduling or optimization strategies so that the performance difference can be attributed mainly to the scheduling mechanism. The experiments are designed to answer the following research questions:

1. Whether Adaptive PSO can reduce CPU usage and improve container-level resource distribution.
2. Whether Decision Tree classification can identify overloaded, medium-load, and underutilized containers or nodes.
3. Whether the hybrid DT+APSO model improves CPU and memory utilization compared with standard PSO.
4. Whether the proposed strategy provides better node-wise balancing in Docker Swarm compared with default Swarm scheduling.

The complete experimental workflow is shown in Figure 4.1.

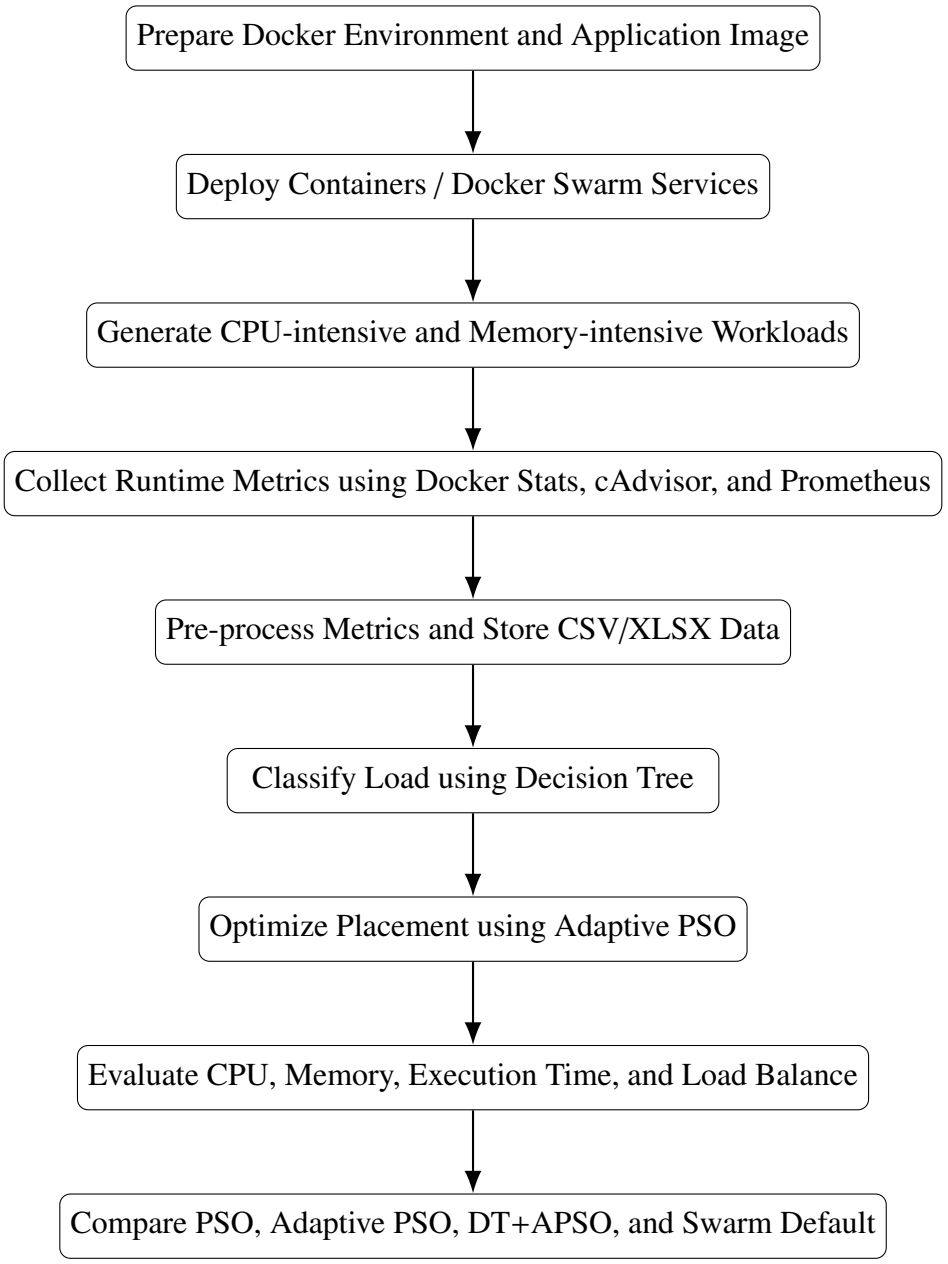


FIGURE 4.1: Experimental Workflow For Validation Of The Proposed DT+APSO Framework

As illustrated in Figure 4.1, The workflow begins with Docker image preparation and service deployment. After deployment, CPU-intensive, memory-intensive, and mixed workloads are executed. Runtime metrics are collected using Docker Stats API, cAdvisor, and Prometheus. These values are processed and supplied to the Decision Tree classifier and APSO optimizer. Finally, the generated results are compared with baseline and conventional scheduling strategies.

4.3 Experimental Infrastructure

The experiments are conducted using two complementary implementation environments. The first environment is a Docker Desktop-based local container testbed, while the second is a Docker Swarm-based distributed cluster.

4.3.1 Phase-I Docker Desktop-Based Experimental Environment

The first phase evaluates the internal behavior of the APSO and Decision Tree model using five Docker containers. The configuration of this environment is shown in Table 4.1.

TABLE 4.1: Phase-I Docker Desktop Experimental Configuration

Component	**Configuration**
Processor	Ryzen / Intel i7 class processor
RAM	16 GB
Operating System	Windows OS with Docker Desktop
Docker Version	Docker 24.0.2
Programming Language	Python 3.10
Python Libraries	Docker SDK, Scikit-learn, NumPy, Pandas, Matplotlib
Number of Containers	5
Workload Type	CPU-intensive, memory-intensive, and mixed workloads
Monitoring Method	Docker Stats API
Data Storage Format	CSV file for algorithm input and result analysis

This setup is mainly used to collect container-wise CPU and memory usage before and after optimization. It helps verify whether the proposed optimization framework can reduce computational load, improve utilization balance, and avoid inefficient resource concentration.

4.3.2 Phase-II Docker Swarm Cluster-Based Environment

The second phase uses a distributed Docker Swarm cluster with one master node and two worker nodes. This setup represents a practical container orchestration environment for node-wise load balancing evaluation. The configuration is shown in Table 4.2.

TABLE 4.2: Phase-II Docker Swarm Cluster Testbed

Node	IP Address	CPU	RAM	Memory/Storage
Master	10.0.4.42	8 Cores	8 GB	20 GB
Worker 1	10.0.4.43	8 Cores	8 GB	20 GB
Worker 2	10.0.4.49	8 Cores	8 GB	20 GB

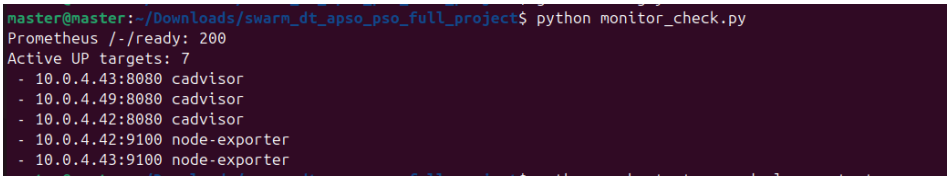


FIGURE 4.2: Monitoring Stack Docker Swarm

In this phase, Docker Swarm is initialized on the master node, and worker nodes join the cluster using the Swarm join token. Application replicas are deployed as Docker services. The monitoring stack is deployed using Prometheus, cAdvisor, and Grafana. This setup is used to evaluate the performance of default Docker Swarm scheduling, standard PSO scheduling, and the proposed DT+APSO scheduling strategy.

4.4 Software Stack and Tools

The implementation uses a software stack capable of supporting deployment, monitoring, optimization, classification, and visualization. The tools used in the experiments are summarized in Table 4.3.

Python is used as the integration layer for data collection, Decision Tree prediction, APSO optimization, and result generation. Docker Stats API is used in the local environment, whereas Prometheus and cAdvisor are used in the Swarm-based cluster environment.

TABLE 4.3: Software Stack Used in Experimental Implementation

Tool / Library	**Purpose**
Docker Engine	Container runtime for building and executing containerized applications.
Docker Desktop	Local execution platform for Phase-I container experiments.
Docker Swarm	Cluster orchestration platform for distributed container scheduling.
Python 3.10	Main programming language for orchestration, classification, optimization, and result processing.
Docker SDK for Python	Programmatic interaction with Docker containers and services.
Scikit-learn	Implementation of Decision Tree classifier using CART and Gini criterion.
NumPy	Numerical computation for APSO and statistical calculations.
Pandas	Data processing, CSV handling, and tabular analysis.
Matplotlib	Plot generation for CPU, memory, efficiency, and comparison graphs.
Docker Stats API	Real-time resource monitoring of containers in Phase-I experiments.
cAdvisor	Low-level container metrics collection in Docker Swarm.
Prometheus	Time-series metric storage and query engine.
Grafana	Visualization of CPU, memory, node load, and monitoring dashboards.
Apache JMeter	Synthetic workload generation for request-based experiments.

4.5 Workload Design

The workload design is important because a scheduling algorithm can be evaluated properly only when containers generate measurable resource pressure. In this study, four workload categories are considered, as shown in Table 4.4.

TABLE 4.4: Workload Categories Used for Experimental Evaluation

Workload Category	**Description**
CPU-intensive workload	Generates computational stress using factorial, recursive, or loop-based numerical operations.
Memory-intensive workload	Generates memory pressure using controlled memory allocation and processing.
Mixed workload	Combines CPU and memory operations to represent dynamic containerized application behavior.
Request-based workload	Uses Apache JMeter to generate concurrent requests and evaluate scheduling response.

CPU-intensive workloads help evaluate whether the scheduler can reduce CPU concentration on overloaded containers or nodes. Memory-intensive workloads help evaluate memory stability. Mixed workloads provide a more realistic representation of cloud-native application behavior.

4.6 Adaptive PSO Configuration

The Adaptive PSO component optimizes the resource allocation decision. In this experiment, each particle represents a candidate container allocation state. The particle updates its position according to local best and global best experience. The inertia

weight is adaptively reduced to balance exploration and exploitation. The APSO configuration is shown in Table 4.5.

Table 4.5: Adaptive PSO Parameter Configuration

Parameter	**Value**
Swarm Size	50 particles
Maximum Iterations	100
Initial Inertia Weight w_{max}	0.9
Final Inertia Weight w_{min}	0.4
Cognitive Coefficient c_1	2.0
Social Coefficient c_2	2.0
Particle Representation	Candidate container allocation state
Optimization Goal	Minimize CPU-memory-resource fitness value

The adaptive inertia weight is computed as:

$$w = w_{max} - \frac{(w_{max} - w_{min})}{I_{max}} \times iteration \tag{4.1}$$

The velocity update equation is:

$$v_{ij}(t+1) = w \cdot v_{ij}(t) + c_1 \cdot r_1 \cdot (p_{ij} - x_{ij}) + c_2 \cdot r_2 \cdot (g_j - x_{ij}) \tag{4.2}$$

The position update equation is:

$$x_{ij}(t+1) = x_{ij}(t) + v_{ij}(t+1) \tag{4.3}$$

The adaptive design allows the algorithm to explore a wider search space during early iterations and focus on local improvement during later iterations. This behavior is suitable for Docker environments because workload demand changes dynamically.

4.7 Decision Tree Classifier Configuration

The Decision Tree classifier identifies patterns in CPU and memory usage and classifies containers or nodes into load categories. The classifier follows the CART model and uses the Gini index as the splitting criterion. The configuration is shown in Table 4.6.

Table 4.6: Decision Tree Classifier Configuration

Parameter	**Configuration**
Classifier Type	CART (Classification and Regression Tree)
Splitting Criterion	Gini Index
Maximum Depth	10
Minimum Samples Split	2
Input Features	CPU usage, memory usage, resource allocation
Output Classes	Low, Medium, High
Purpose	Predictive workload classification for APSO guidance

The Gini impurity is calculated as:

$$Gini(D) = 1 - \sum_{k=1}^{K} p_k^2 \tag{4.4}$$

The Decision Tree classifies the load using the following decision rule:

$$f(x_1, x_2) = \begin{cases} High, & x_1 > 80 \wedge x_2 > 75 \\ Low, & x_1 < 30 \wedge x_2 < 25 \\ Medium, & otherwise \end{cases} \tag{4.5}$$

The classified load state is converted into a numerical cost:

$$Cost = \begin{cases} 3, & High \\ 2, & Medium \\ 1, & Low \end{cases} \tag{4.6}$$

This cost is used to guide APSO so that high-load nodes are avoided during placement.

4.8 Data Collection and Pre-processing

Runtime data collection is performed using Docker Stats API in the local environment and Prometheus-cAdvisor in the Swarm environment. CPU and memory readings are collected at five-second intervals and aggregated over one-minute periods. This aggregation reduces the effect of short-term fluctuations and produces stable input for the optimization model shown in Figure 4.3.

The raw values may contain sudden spikes caused by transient system activity. Therefore, pre-processing is applied before feeding the data into the Decision Tree and APSO algorithm. The pre-processing steps include abnormal outlier removal, moving average smoothing, normalization of resource values, and storage in CSV or XLSX format Figure 4.4 shows monitoring framework as node exporter installed at Manager and Worker node, Figure 4.5 and Figure 4.6 shows Menu driven orchestrator program and The Fibonacci Mystack application screenshot for experiments performed..

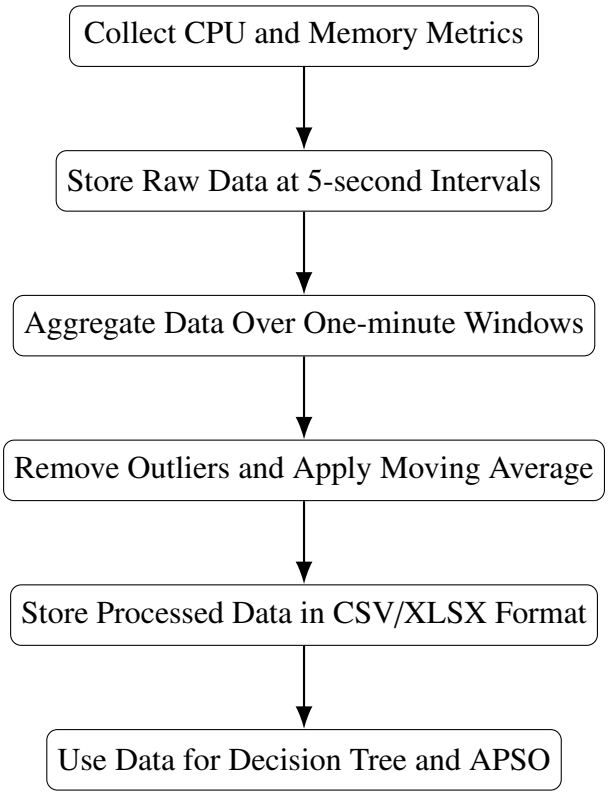


FIGURE 4.3: Data Collection And Pre-Processing Pipeline

4.9 Evaluation Metrics

The proposed framework is evaluated using CPU utilization, memory utilization, efficiency improvement, execution time, node-wise load balance, and algorithm-wise comparison. These metrics directly represent the effectiveness of resource scheduling in Docker container environment as shown in Equations 4.7- 4.11 below.

CPU efficiency improvement is calculated as:

$$CPU_{Improvement}(\%) = \frac{CPU_{Before} - CPU_{After}}{CPU_{Before}} \times 100 \tag{4.7}$$

Memory efficiency improvement is calculated as:

$$MEM_{Improvement}(\%) = \frac{MEM_{Before} - MEM_{After}}{MEM_{Before}} \times 100 \tag{4.8}$$

Average CPU load is calculated as:

$$\overline{CPU} = \frac{1}{n}\sum_{i=1}^{n} CPU_i \tag{4.9}$$

Average memory usage is calculated as:

$$\overline{MEM} = \frac{1}{n}\sum_{i=1}^{n} MEM_i \tag{4.10}$$

Node-wise variance is calculated as:

$$Variance = \frac{1}{m}\sum_{j=1}^{m} (L_j - \overline{L})^2 \tag{4.11}$$

where L_j is the load of node j and $\overline{L}$ is the average cluster load.

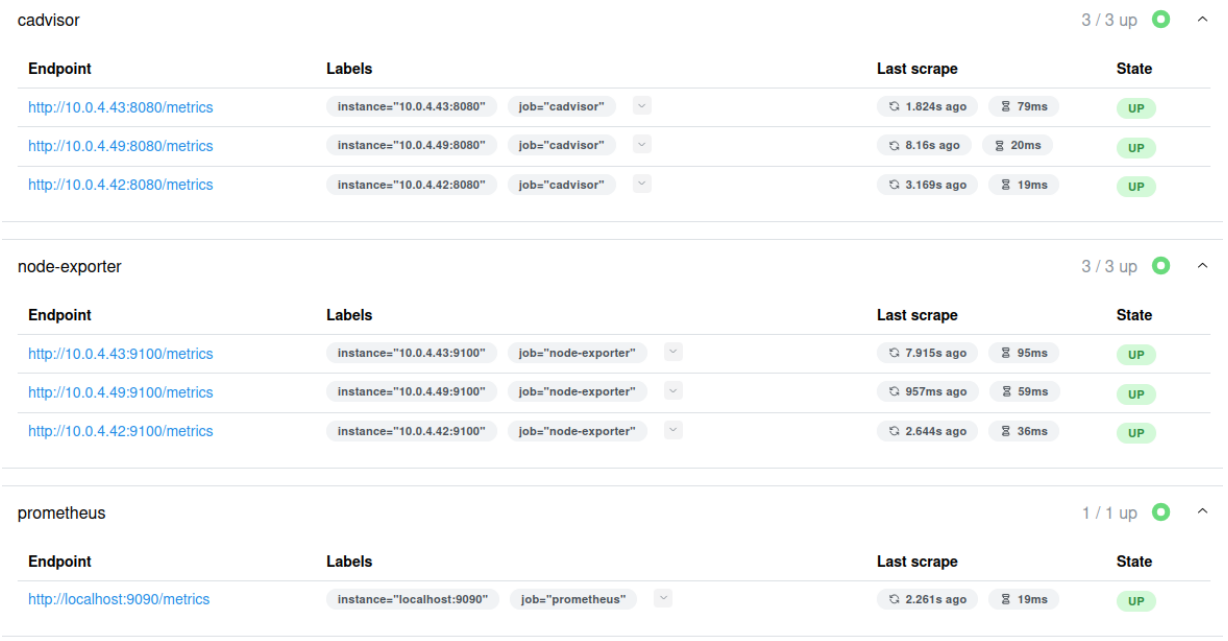


FIGURE 4.4: Metric Collection Via CAdvisor And Node Exporter

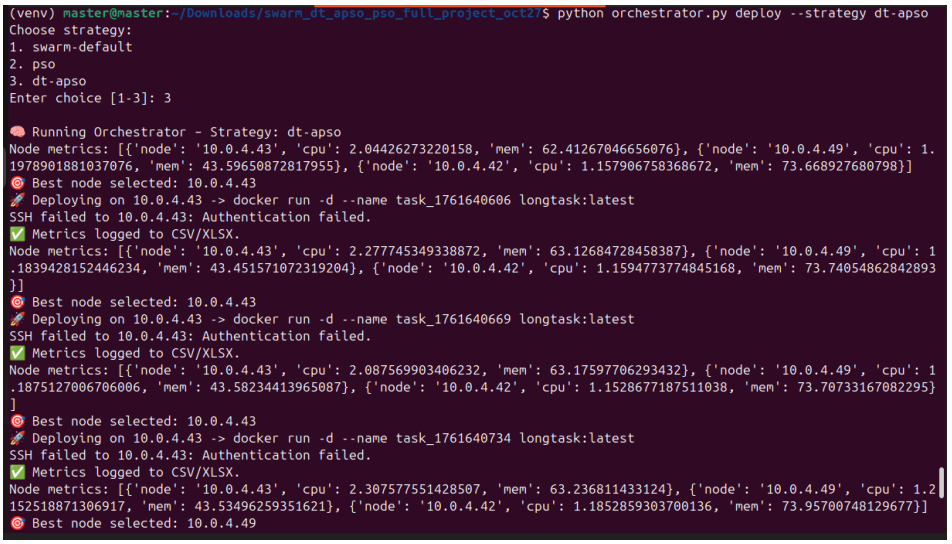


FIGURE 4.5: Menu Driven Orchastration for All Strategy

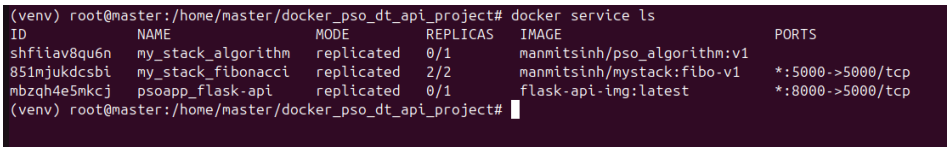


FIGURE 4.6: Fibonacci MyStack Application

4.10 Chapter Summary

This chapter described the experimental setup and implementation framework used for validating the proposed DT+APSO model. It presented the Docker Desktop-based container environment, the Docker Swarm cluster environment, the software stack, workload design, Adaptive PSO configuration, Decision Tree classifier configuration, data collection process, pre-processing pipeline, and evaluation metrics. The next chapter presents the results obtained from these experiments and provides comparative analysis against PSO variants and default Docker Swarm scheduling.

CHAPTER 5

Results Analysis and Comparative Evaluation

5.1 Introduction

This chapter presents the experimental results and comparative analysis of the proposed Hybrid Adaptive PSO and Decision Tree model. The analysis is organized around CPU-centric workload results, memory-centric workload results, execution time, efficiency improvement, comparison with PSO and its PSO variants, and Docker Swarm node-wise resource utilization.

The purpose of this chapter is to determine whether the proposed DT+APSO framework improves resource allocation and load balancing in Docker container-based cloud environments. The results are interpreted in terms of CPU utilization, memory usage, improvement percentage, execution overhead, and node-wise scheduling behavior.

5.2 Result Analysis of CPU-Centric Workload

The first experiment evaluates CPU-centric workload behavior before and after optimization. Table 5.1 shows the CPU and memory values for five containers.

TABLE 5.1: Before-And-After Resource Usage For CPU-Centric Processes

Container	CPU Before (%)	Memory Before (%)	CPU After (%)	Memory After (%)
Container 1	13.8	93.0	12.2	93.7
Container 2	16.6	93.8	15.3	93.7
Container 3	20.5	92.8	15.6	93.9
Container 4	11.7	93.4	14.0	94.4
Container 5	15.2	93.5	16.5	93.7

The results in Table 5.1 show that Container 3 experiences a meaningful reduction in CPU usage from 20.5% to 15.6%. Container 1 also shows a reduction from 13.8% to 12.2%. Container 4 and Container 5 show slightly increased CPU after optimization, which indicates that the optimizer redistributed part of the workload from heavier containers to comparatively lighter containers. This is acceptable in load balancing because the objective is not to reduce every individual value but to reduce imbalance and improve overall distribution As shown in Figure 5.1.

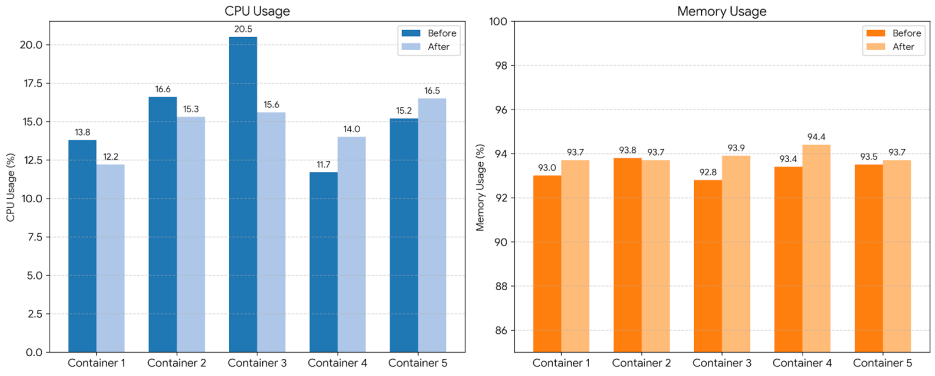


FIGURE 5.1: CPU And Memory Usage Before And After Optimization

5.3 Execution Time Analysis

Execution time is important because a scheduling algorithm must not introduce excessive overhead. The Decision Tree classifier is lightweight and produces fast

classification results, whereas PSO optimization takes more time because it performs iterative search. The measured execution times are shown in Table 5.2.

TABLE 5.2: Execution Time Of Decision Tree And PSO Optimization

Process	**Execution Time (seconds)**
Decision Tree	0.014994
PSO Optimization	0.217299

The Decision Tree classifier completes execution in 0.014994 seconds, confirming its suitability for real-time or near-real-time load classification. PSO optimization requires 0.217299 seconds, which is higher but still acceptable for scheduling decisions. The difference is expected because PSO performs iterative optimization, while Decision Tree prediction only traverses a trained tree.

5.4 CPU and Memory Efficiency Improvement

Table 5.3 presents CPU and memory efficiency improvement after optimization.

TABLE 5.3: CPU And Memory Efficiency Improvement

Container	**CPU Before (%)**	**Memory Before (%)**	**CPU After (%)**	**Memory After (%)**	**CPU Improvement (%)**	**Memory Improvement (%)**
Container 1	22.1	95.7	11.7	95.2	47.06	0.522
Container 2	12.8	95.7	11.7	95.3	8.59	0.418
Container 3	20.2	95.4	8.2	95.2	59.41	0.210
Container 4	11.5	95.4	10.5	95.2	8.70	0.210
Container 5	25.0	94.9	11.2	95.3	55.20	-0.421

Container 3 achieves the highest CPU improvement of 59.41%, followed by Container 5 with 55.20% and Container 1 with 47.06%. These values indicate that the optimization framework is more effective for containers with higher initial CPU demand. Memory improvement is comparatively smaller because memory usage remains more stable during the workload. The slight negative memory improvement for Container 5 indicates a marginal increase, but it does not affect the overall

conclusion because the major gain is observed in CPU balancing result graph is illustrated in Figure 5.2.

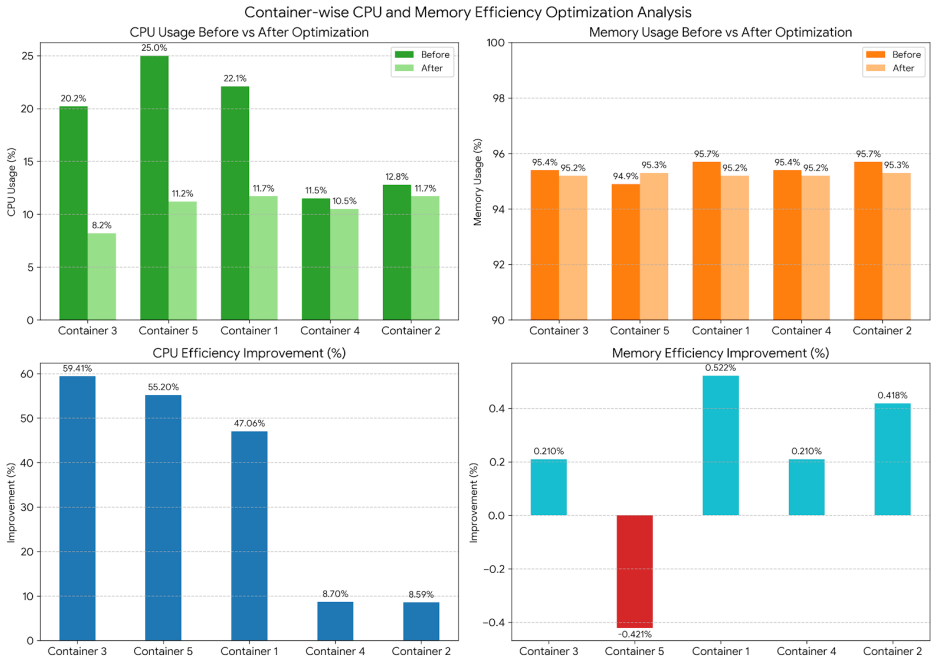


FIGURE 5.2: Container-Wise CPU And Memory Efficiency Improvement

5.5 Memory-Centric Workload Results

The memory-centric workload experiment evaluates the behavior of the optimization framework when memory usage is the dominant factor. Table 5.4 shows the before-after values.

TABLE 5.4: Before-And-After Resource Usage For Memory-Centric Processes

Container	CPU Before (%)	Memory Before (%)	CPU After (%)	Memory After (%)
Container 1	2.0	89.1	9.4	86.5
Container 2	19.8	89.1	6.2	86.5
Container 3	7.6	89.1	4.2	86.5
Container 4	17.4	89.1	0.4	86.5
Container 5	8.1	89.1	2.6	86.5

The results Table 5.4 shows that memory usage is reduced from 89.1% to 86.5% for all containers. This indicates that the optimization process contributes to a more stable memory profile. CPU usage decreases for most containers except Container 1, where CPU increases from 2.0% to 9.4%. This increase may result from workload redistribution toward an initially underutilized container.

5.6 Before and After Resource Usage after 1000 Iterations

To observe longer optimization behavior, the experiment also evaluates resource usage after 1000 iterations. Table 5.5 presents the experimental result.

TABLE 5.5: Before-And-After Resource Usage After 1000 Iterations

Container	**CPU Before (%)**	**Memory Before (%)**	**CPU After (%)**	**Memory After (%)**
Container 1	8.5	93.0	1.5	92.8
Container 2	5.2	93.0	3.8	92.8
Container 3	6.2	93.0	10.0	92.8
Container 4	8.2	93.0	1.2	92.8
Container 5	9.7	93.0	3.5	92.8

After 1000 iterations, memory values become stable around 92.8%, while CPU values are reduced for most containers. Container 3 increases from 6.2% to 10.0%, indicating that it receives additional work during balancing. The result confirms that the optimizer redistributes workload to avoid concentration on selected containers.

5.7 Comparison of PSO and APSO+DT

The hybrid APSO+DT approach is compared with standard PSO using CPU and memory values. Table 5.6 presents the results.

TABLE 5.6: Comparison Of PSO And APSO+DT For CPU And Memory Improvement

Container	CPU PSO (%)	CPU APSO+DT (%)	Memory PSO (%)	Memory APSO+DT (%)	CPU Improvement (%)	Memory Improvement (%)
Container 1	71.82	57.46	84.04	67.24	14.36	16.81
Container 2	60.11	48.09	75.73	60.59	12.02	15.15
Container 3	69.64	55.71	83.53	66.82	13.93	16.71
Container 4	83.11	66.49	97.93	78.35	16.62	19.59
Container 5	66.49	40.01	77.78	62.22	10.00	15.56

The APSO+DT method reduces CPU usage for all five containers compared with PSO. Container 4 shows the highest CPU improvement of 16.62% and the highest memory improvement of 19.59%. Container 5 shows CPU reduction from 66.49% to 40.01%, which is a significant practical improvement. These results indicate that Decision Tree-based workload classification improves the scheduling guidance of APSO by identifying more suitable allocation patterns as shown in Figure 5.3.

```
master@master:~/Downloads/swarm_dt_apso_pso_full_project$ python orchestrator.py collect --label baseline
[collect:baseline] rows=0
[saved] /home/master/Downloads/swarm_dt_apso_pso_full_project/results/baseline_metrics.csv
[saved] /home/master/Downloads/swarm_dt_apso_pso_full_project/results/baseline_metrics.xlsx
[saved] /home/master/Downloads/swarm_dt_apso_pso_full_project/results/baseline_metrics.txt
master@master:~/Downloads/swarm_dt_apso_pso_full_project$ python orchestrator.py remove --strategy swarm-default
[remove] removed exp_default
master@master:~/Downloads/swarm_dt_apso_pso_full_project$ python orchestrator.py deploy --strategy pso
[pso] alloc: {'swarm_master': 2, 'worker1-VirtualBox': 2, 'worker2-VirtualBox': 2}
[deploy] exp_pso-swarm_master replicas=2 on swarm_master
[deploy] exp_pso-worker1-VirtualBox replicas=2 on worker1-VirtualBox
[deploy] exp_pso-worker2-VirtualBox replicas=2 on worker2-VirtualBox
master@master:~/Downloads/swarm_dt_apso_pso_full_project$ python orchestrator.py wait --seconds 90
[wait] 90s ...
master@master:~/Downloads/swarm_dt_apso_pso_full_project$ python orchestrator.py collect --label pso
[collect:pso] rows=0
[saved] /home/master/Downloads/swarm_dt_apso_pso_full_project/results/pso_metrics.csv
[saved] /home/master/Downloads/swarm_dt_apso_pso_full_project/results/pso_metrics.xlsx
[saved] /home/master/Downloads/swarm_dt_apso_pso_full_project/results/pso_metrics.txt
master@master:~/Downloads/swarm_dt_apso_pso_full_project$ python orchestrator.py remove --strategy pso
[remove] removed exp_pso-swarm_master
[remove] removed exp_pso-worker1-VirtualBox
[remove] removed exp_pso-worker2-VirtualBox
master@master:~/Downloads/swarm_dt_apso_pso_full_project$ python orchestrator.py deploy --strategy dt-apso
[dt-apso] alloc: {'swarm_master': 1, 'worker1-VirtualBox': 1, 'worker2-VirtualBox': 4}
[deploy] exp_dtapso-swarm_master replicas=1 on swarm_master
[deploy] exp_dtapso-worker1-VirtualBox replicas=1 on worker1-VirtualBox
[deploy] exp_dtapso-worker2-VirtualBox replicas=4 on worker2-VirtualBox
```

FIGURE 5.3: Comparison Of PSO And APSO+DT For CPU And Memory Utilization

5.8 Comparison of PSO Variants

The comparison of PSO, TMPSO, and Adaptive PSO is important because it establishes the background for selecting Adaptive PSO as the optimization foundation. Table 5.7 presents the comparative values.

TABLE 5.7: Comparative Analysis Of PSO, TMPSO, And Adaptive PSO

Container	**Algorithm**	**CPU Load**	**Memory Usage (MB)**	**Execution Time (Sec.)**
1	PSO	23.5655	61.78275	77.06965
2	PSO	23.40568	61.70284	77.02170
3	PSO	23.39910	61.69955	77.01973
4	PSO	23.45902	61.72951	77.03771
5	PSO	23.46109	61.73055	77.03833
Average	PSO	23.46	61.73	77.04
1	TMPSO	50.18283	75.09142	85.05485
2	TMPSO	49.44382	74.72191	84.83315
3	TMPSO	49.27857	74.63929	84.78357
4	TMPSO	49.75902	74.87951	84.92771
5	TMPSO	49.61538	74.80769	84.88462
Average	TMPSO	49.66	74.83	84.90
1	Adaptive PSO	25.99327	62.99663	77.79798
2	Adaptive PSO	26.30020	63.15010	77.89006
3	Adaptive PSO	26.21970	63.10985	77.86591
4	Adaptive PSO	26.78923	63.39462	78.03677
5	Adaptive PSO	26.87113	63.43557	78.06134
Average	Adaptive PSO	26.43	63.22	77.93

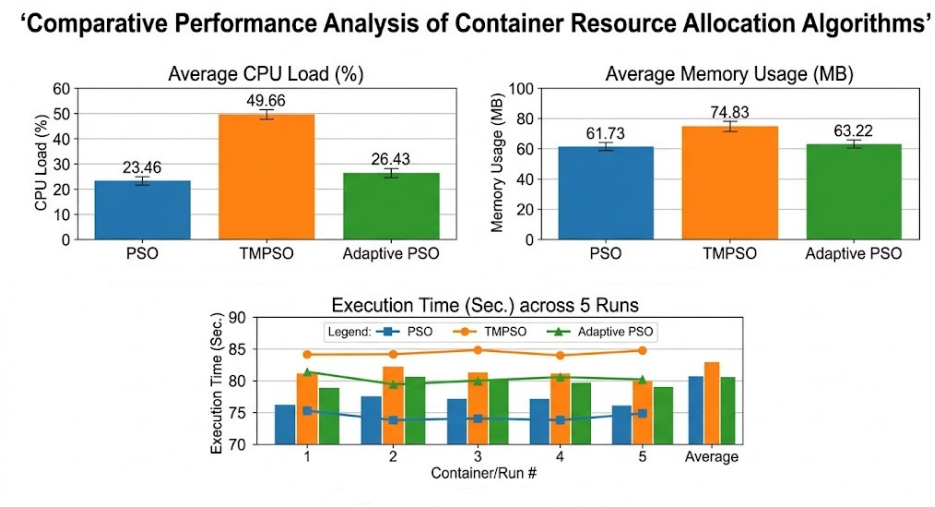


FIGURE 5.4: Comparison of all PSO Variants

The Results shown in Table 5.7 Standard PSO shows lower average CPU load and execution time in the static single-objective setting. However, Adaptive PSO is more suitable for dynamic workload conditions because it adjusts parameters during execution. TMPSO shows higher CPU load, memory usage, and execution time, indicating greater computational overhead. This comparison supports the use of Adaptive PSO as a robust optimization base for dynamic Docker container scheduling Figure 5.4 is comparison bar chart for all three variants of PSO.

5.9 Docker Swarm Node-Wise CPU Utilization

The Swarm-based experiment compares DT+APSO, standard PSO, and default Docker Swarm scheduling using node-wise CPU utilization. Table 5.8 presents total CPU utilization per node.

Table 5.8: Total CPU Utilization per Node

Node	DT+APSO	PSO	Swarm-Default
Total	41.5333	45.8479	39.1625
10.0.4.42	10.5146	11.6747	11.4539
10.0.4.43	20.3312	22.4355	15.9302
10.0.4.49	10.6875	11.7377	11.7784

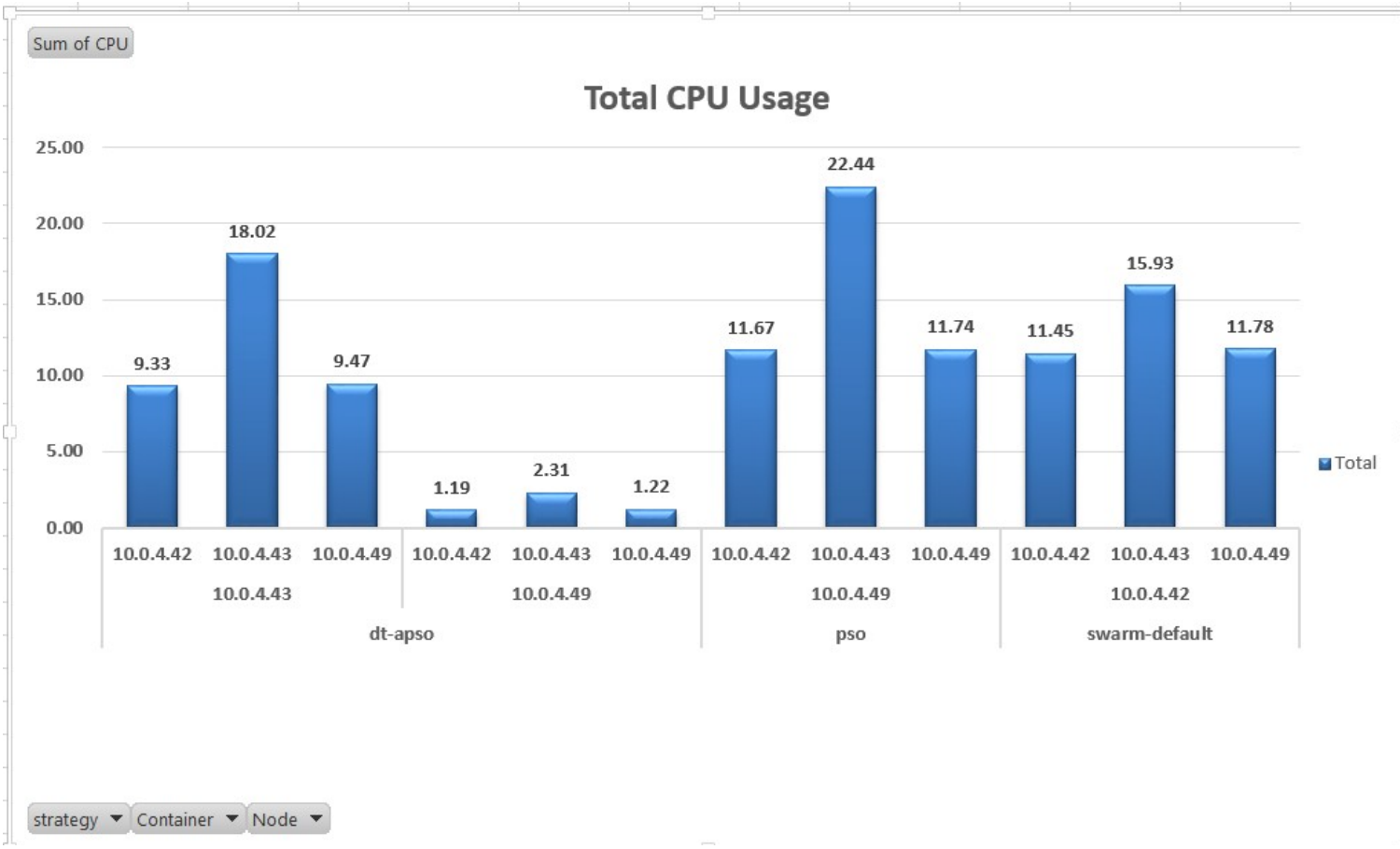


Figure 5.5: CPU Usage Comparison of PSO, DT+APSO, and Swarm-Default

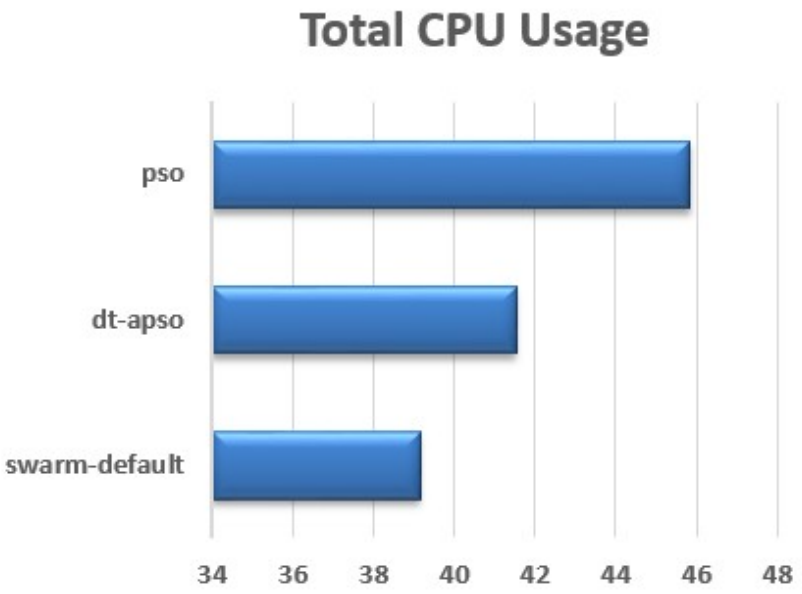


FIGURE 5.6: Total CPU Utilization

The CPU results in Figure 5.6 show that DT+APSO produces lower total CPU utilization than PSO. The total CPU value for DT+APSO is 41.5333, whereas PSO records 45.8479. This indicates that the proposed hybrid strategy reduces CPU demand compared with standard PSO. Although the default Swarm scheduler shows a lower total CPU value in this specific run, it does not include predictive optimization. Therefore, CPU total must be interpreted along with memory usage, node variance, scheduling adaptability, and placement fairness.

5.10 Docker Swarm Node-Wise Memory Utilization

The Figure 5.8 shows Memory utilization is another important metric because memory pressure can degrade container performance and cause instability. Table 5.9 presents total memory utilization per node.

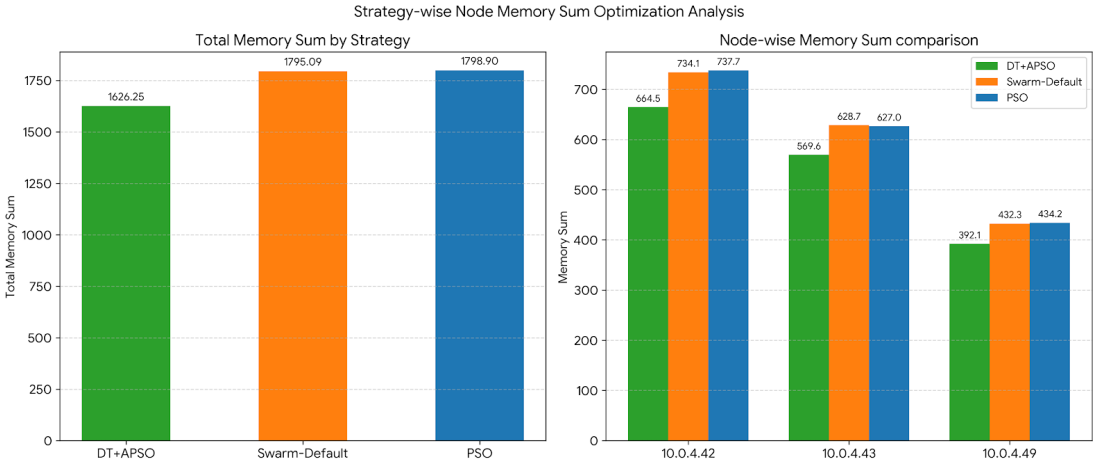


FIGURE 5.7: Container wise Memory usage Comparison

TABLE 5.9: Total Memory Utilization per Node

Node	DT+APSO	PSO	Swarm-Default
Total	1626.2452	1798.8996	1795.0890
10.0.4.42	664.5458	737.7496	734.0947
10.0.4.43	569.5854	626.9549	628.6942
10.0.4.49	392.1139	434.1951	432.3000

5.11 Node-Wise CPU Strategy Comparison

The memory results strongly support the effectiveness of DT+APSO. The total memory utilization under DT+APSO is 1626.2452, while PSO and Swarm-Default record 1798.8996 and 1795.0890, respectively. This shows that the proposed method reduces memory usage compared with both baseline strategies. The result also indicates that Decision Tree-guided node suitability analysis helps APSO avoid memory-heavy placement decisions.

Table 5.10 presents CPU utilization strategy-wise on each node.

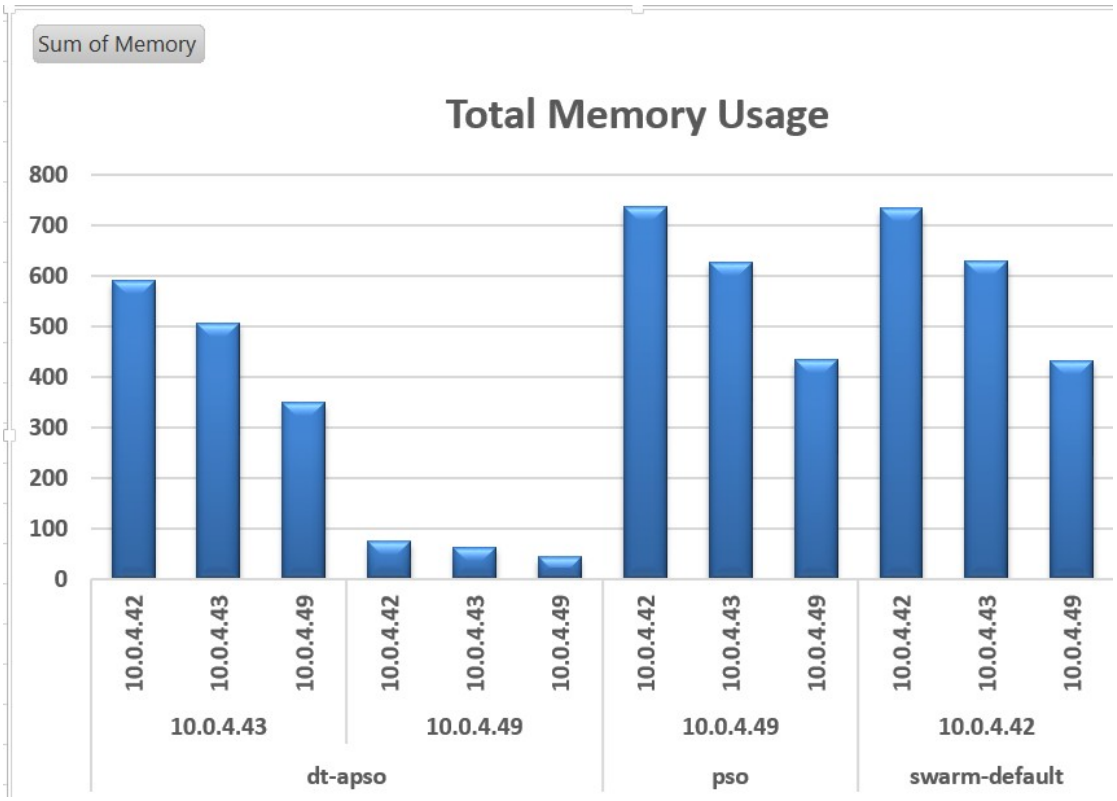


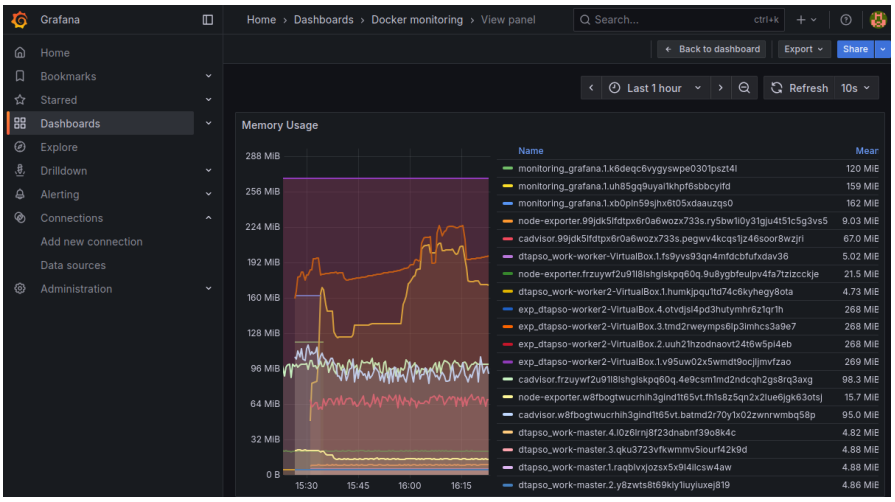
FIGURE 5.8: Container wise Memory usage Comparison

TABLE 5.10: CPU Utilization Strategy-wise on Each Node

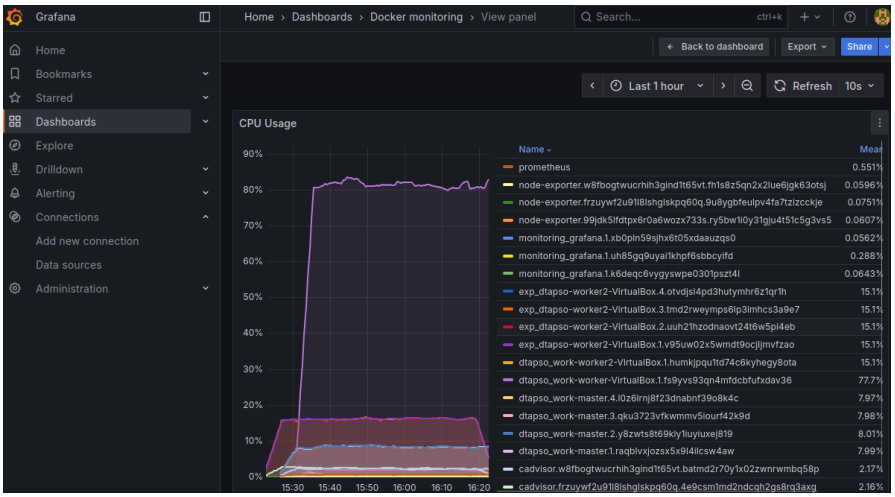
Strategy	**10.0.4.42**	**10.0.4.43**	**10.0.4.49**
Total	33.6432	58.6969	34.2036
DT+APSO	10.5146	20.3312	10.6875
PSO	11.6747	22.4355	11.7377
Swarm-Default	11.4539	15.9302	11.7784

For nodes 10.0.4.42 and 10.0.4.49, DT+APSO produces the lowest CPU usage compared with PSO and Swarm-Default. On node 10.0.4.43, Swarm-Default has lower CPU usage, but the total behavior across CPU and memory must be interpreted together. In distributed scheduling, the best strategy is the one that maintains resource stability across multiple dimensions rather than optimizing one isolated node value.

Figure 5.9 (a) and (b) shows memory utilisation and CPU Utilization in Grafana dashboard.



(a) Grafana memory utilization



(b) Grafana CPU utilization

FIGURE 5.9: Grafana Monitoring Dashboard for CPU and Memory Utilization

5.12 Comparative Discussion

The experimental results show that the proposed DT+APSO framework improves resource optimization in Docker container-based environments. In the CPU-centric experiment, high-load containers experience clear CPU reduction after optimization. The efficiency improvement results show that Container 3 achieves 59.41% CPU improvement and Container 5 achieves 55.20% improvement. These results indicate that the optimizer is effective when containers show high initial computational demand.

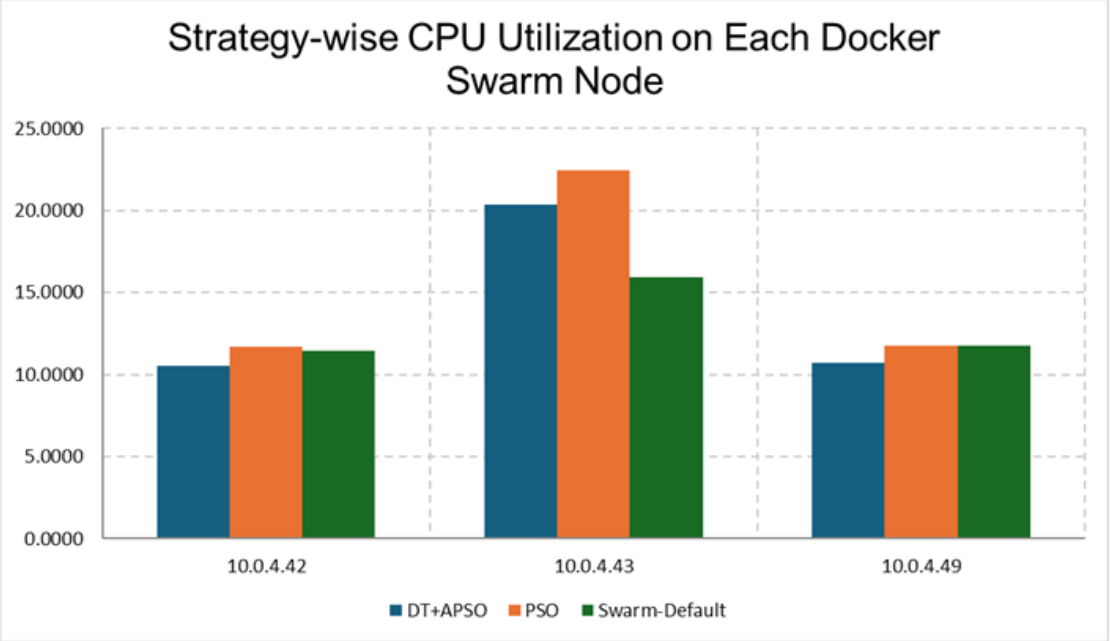


FIGURE 5.10: Node-wise CPU Utilization

The Figure 5.10 memory-centric experiment shows stable memory reduction from 89.1% to 86.5% across all containers. This confirms that the proposed method is not limited to CPU optimization but also contributes to memory stability. The comparison between PSO and APSO+DT shows that the hybrid method consistently reduces CPU and memory values compared with standard PSO. This confirms the benefit of combining Decision Tree classification with Adaptive PSO.

The Docker Swarm node-wise results further demonstrate the value of the proposed model in distributed environments. DT+APSO reduces total memory usage compared with both PSO and Swarm-Default and reduces total CPU usage compared with PSO. The result confirms that the proposed framework is suitable for Docker Swarm scheduling where multiple nodes and replicas must be managed simultaneously.

Compared with default Docker Swarm scheduling, the proposed DT+APSO framework provides more intelligent placement because it uses live telemetry and predictive classification. Default Docker Swarm scheduling is simple and useful for general deployment, but it does not consider historical behavior or predictive load classification. Therefore, DT+APSO is more suitable for dynamic workloads with changing CPU and memory behavior.

5.13 Chapter Summary

This chapter presented the results analysis and comparative evaluation of the proposed Hybrid Adaptive PSO and Decision Tree model. The analysis included CPU-centric results, memory-centric results, execution time, CPU and memory efficiency improvement, comparison with PSO, comparison with PSO variants, and Docker Swarm node-wise resource utilization.

The results showed that the proposed framework improves CPU efficiency, maintains stable memory usage, and reduces resource imbalance. CPU efficiency improvement reached 59.41% for Container 3 and 55.20% for Container 5. The Decision Tree classifier executed in 0.014994 seconds, while PSO optimization required 0.217299 seconds. The comparison of PSO and APSO+DT showed consistent reductions in CPU and memory utilization. The Docker Swarm results also showed that DT+APSO reduces total memory usage compared with PSO and Swarm-Default and reduces total CPU usage compared with standard PSO.

Overall, the results validate that the proposed DT+APSO model is effective for intelligent resource allocation and load balancing in Docker container environments. The integration of predictive Decision Tree classification with Adaptive PSO optimization improves scheduling adaptability, reduces resource contention, and supports practical deployment in Docker Swarm-based cloud infrastructures.

CHAPTER 6

Conclusion and Future Work

6.1 Conclusion

This chapter presented the design, implementation, and evaluation of a hybrid Decision Tree and Adaptive Particle Swarm Optimization (DT+APSO) framework for optimized resource allocation in a Docker container-based cloud environment. The study was motivated by the limitations of conventional scheduling methods in Docker Swarm, which often rely on static or rule-based placement strategies and are therefore unable to respond efficiently to rapidly changing workload conditions. To address this issue, the proposed approach combined the predictive capability of a Decision Tree model with the adaptive search strength of APSO, enabling more intelligent and dynamic container placement decisions.

The experimental results demonstrated that the DT+APSO framework achieved better load distribution across the cluster when compared with the default Docker Swarm scheduler and standard PSO-based allocation. In particular, the proposed method reduced imbalance in CPU utilization, stabilized memory consumption, and improved the overall efficiency of replica placement. The Decision Tree component helped identify the suitability of worker nodes based on real-time system metrics, while the APSO component adaptively refined the allocation process by balancing exploration and exploitation during optimization. As a result, the proposed scheduler was able to avoid overloading specific nodes and make better use of available resources across the Docker Swarm cluster.

Another important outcome of this research is its practical applicability. The framework was implemented in a real Docker Swarm environment using live

monitoring data collected through Prometheus and cAdvisor, with visualization support through Grafana. This ensured that resource allocation decisions were based on actual system behaviour rather than only simulated conditions. Moreover, the proposed solution was integrated externally through orchestration mechanisms, without requiring modifications to the Docker engine itself. This makes the framework flexible, scalable, and suitable for real-world deployment in containerized cloud infrastructures.

Overall, the work confirms that the integration of machine learning and adaptive optimization can significantly enhance resource allocation in container-based cloud environments. The DT+APSO model provides a robust, intelligent, and efficient solution for handling dynamic workloads in Docker clusters. The findings of this thesis therefore establish a strong foundation for future intelligent schedulers that can improve performance, resource utilization, and stability in modern cloud computing platforms.

6.2 Future Work

Although the proposed DT+APSO framework achieved promising results, several opportunities remain for further enhancement. In the present work, the optimization process mainly focused on CPU and memory utilization. In future, the framework can be extended into a multi-objective optimization model that simultaneously considers additional parameters such as network bandwidth, task completion time, response latency, throughput, and storage usage. Such an extension would allow the scheduler to make more balanced allocation decisions in highly dynamic and heterogeneous cloud environments.

Another important direction for future research is energy-aware resource allocation. By integrating power consumption models or node energy profiles into the fitness function, the scheduler could reduce unnecessary energy usage by consolidating workloads onto fewer active nodes during low-demand periods. This would be especially useful in

green cloud computing environments, where reducing power consumption and carbon footprint has become an important design objective.

The proposed framework can also be extended beyond Docker Swarm and adapted for Kubernetes, which is currently the most widely adopted container orchestration platform. Implementing the DT+APSO strategy through Kubernetes scheduler extensions or custom scheduling APIs would help evaluate the generality and portability of the proposed method. Successful deployment in Kubernetes would further validate the practical relevance of the approach across different container orchestration ecosystems.

In addition, future work can incorporate Service Level Agreement (SLA)-aware scheduling into the optimization model. In such a framework, the scheduler would consider not only resource utilization but also application-level quality-of-service requirements such as response time, availability, and throughput guarantees. This would allow the system to prioritize critical services, support bursty workloads more effectively, and improve user satisfaction in production cloud platforms.

Further improvements may also include the use of more advanced learning and optimization techniques, such as ensemble learning, reinforcement learning, or hybrid deep learning-based workload prediction models. These methods may help the system capture more complex workload patterns and improve decision-making under uncertain or rapidly changing operating conditions. Testing the framework on larger clusters, real cloud traces, and edge-cloud environments would also strengthen its applicability and validate its performance at scale.

In summary, the future scope of this research lies in making the proposed scheduler more comprehensive, intelligent,and adaptable.By extending it toward multi-objective, energy-aware, SLA-conscious, and cross-platform scheduling, the DT+APSO framework can evolve into a more powerful and practical solution for next-generation containerized cloud and edge computing systems

List of References

[1] V. G. da Silva, M. Kirikova, and G. Alksnis, "Containers for virtualization: An overview," *Applied Computer Systems*, vol. 23, no. 1, pp. 21–27, 2018.

[2] A. Anand, "A survey on docker container and its use cases," *Journal/venue not provided in source*, vol. 7, no. 3, pp. 1999–2002, 2021.

[3] D. Reis, B. Piedade, F. F. Correia, J. P. Dias, and A. Aguiar, "Developing Docker and Docker-Compose Specifications : A Developers ' Survey," vol. 10, 2022.

[4] R. Dua, A. R. Raja, and D. Kakadia, "Virtualization vs containerization to support PaaS," *Proceedings - 2014 IEEE International Conference on Cloud Engineering, IC2E 2014*, pp. 610–614, 2014.

[5] J. Lv, M. Wei, and Y. Yu, "A container scheduling strategy based on machine learning in microservice architecture," *Proceedings - 2019 IEEE International Conference on Services Computing, SCC 2019 - Part of the 2019 IEEE World Congress on Services*, pp. 65–71, 2019.

[6] R. Peinl and F. Holzschuher, "The docker ecosystem needs consolidation," in *Proceedings of the 5th International Conference on Cloud Computing and Services Science (CLOSER 2015)*. Lisbon, Portugal: SCITEPRESS – Science and Technology Publications, Lda., May 2015, pp. 535–542.

[7] I. Ahmad, M. G. AlFailakawi, A. AlMutawa, and L. Alsalman, "Container scheduling techniques: A survey and assessment," *Journal of King Saud University - Computer and Information Sciences*, vol. 34, no. 7, pp. 3934–3947, 2022.

[8] H. Zeng, B. Wang, W. Deng, and W. Zhang, "Measurement and evaluation for Docker container networking," in *2017 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*, 2017, pp. 105–108.

[9] K. S. Shams, M. W. Powell., T. M. Crockett, J. S. Norris, R. Rossi, and T. Soderstrom, "Polyphony: A workflow orchestration framework for cloud computing," *CCGrid 2010 - 10th IEEE/ACM International Conference on Cluster, Cloud, and Grid Computing*, pp. 606–611, 2010.

[10] L. Wu and H. Xia, "Particle Swarm Optimization Algorithm for Container Deployment," *Journal of Physics: Conference Series*, vol. 1544, no. 1, 2020.

[11] H. Rajavaram, V. Rajula, and B. Thangaraju, "Automation of Microservices Application Deployment Made Easy By Rundeck and Kubernetes," *2019 IEEE International Conference on Electronics, Computing and Communication Technologies, CONECCT 2019*, pp. 3–5, 2019.

[12] D. Liu, Y. Xia, C. Shan, K. Tian, and Y. Zhan, "A kubernetes-based scheme for efficient resource allocation in containerized workflow," *Future Generation Computer Systems*, vol. 166, p. 107699, 2025.

[13] M. del Carmen Carrión, "Kubernetes scheduling: Taxonomy, ongoing issues and challenges," *ACM Computing Surveys*, vol. 55, no. 7, pp. 1–37, 2022.

[14] K. Senjab, S. Abbas, N. Ahmed, and A. ur Rehman Khan, "A survey of kubernetes scheduling algorithms," *Journal of Cloud Computing*, vol. 12, no. 1, 2023.

[15] B. Bashari Rad, H. John Bhatti, and M. Ahmadi, "An Introduction to Docker and Analysis of its Performance," *IJCSNS International Journal of Computer Science and Network Security*, vol. 17, no. 3, pp. 228–235, 2017.

[16] P. Mohan, T. Jambhale, L. Sharma, S. Koul, and S. Koul, "Load Balancing using Docker and Kubernetes : A Comparative Study," vol. 3878, no. 2, pp. 782–792, 2020.

[17] H. Zhu, Y. Hu, and W. Zhu, "A dynamic adaptive particle swarm optimization and genetic algorithm for different constrained engineering design optimization problems," *Advances in Mechanical Engineering*, vol. 11, no. 3, pp. 1–27, 2019.

[18] V. K. Netaji and G. P. Bhole, "A comprehensive survey on container resource allocation approaches in cloud computing: State-of-the-art and research challenges," *Web Intelligence*, vol. 19, no. 4, pp. 295–316, 2021.

[19] X. Chen and S. Xiao, "Multi-objective and parallel particle swarm optimization algorithm for container-based microservice scheduling," *Sensors*, vol. 21, no. 18, p. 6212, 2021.

[20] D. N. Jha, S. Garg, P. P. Jayaraman, R. Buyya, Z. Li, and R. Ranjan, "A holistic evaluation of docker containers for interfering microservices," *Proceedings - 2018 IEEE International Conference on Services Computing, SCC 2018 - Part of the 2018 IEEE World Congress on Services*, no. VM, pp. 33–40, 2018.

[21] Z. Wei-guo, M. Xi-lin, and Z. Jin-zhong, "Research on kubernetes' resource scheduling scheme," *ACM International Conference Proceeding Series*, pp. 144–148, 2018.

[22] Y. Xu and Y. Shang, "Dynamic Priority based Weighted Scheduling Algorithm in Microservice System," *IOP Conference Series: Materials Science and Engineering*, vol. 490, no. 4, 2019.

[23] C. Singh, N. S. Gaba, M. Kaur, and B. Kaur, "Comparison of different CI/CD Tools integrated with cloud platform," *Proceedings of the 9th International Conference On Cloud Computing, Data Science and Engineering, Confluence 2019*, pp. 7–12, 2019.

[24] C. Cérin, T. Menouer, W. Saad, and W. B. Abdallah, "A New Docker Swarm Scheduling Strategy," *Proceedings - 2017 IEEE 7th International Symposium on Cloud and Service Computing, SC2 2017*, vol. 2018-Janua, pp. 112–117, 2018.

[25] M. Beranek, V. Kovar, and G. Feuerlicht, *Framework for Management of Multi-tenant Cloud Environments*. Springer International Publishing, 2018, vol. 10967 LNCS. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-94295-7_21

[26] M. Rusek, D. Rzegorz, and A. Orłowski, "A decentralized system for load balancing of containerized microservices in the cloud," *International Conference on Systems Science (ICSS)*, vol. 539, no. November, pp. 142–152, 2016. [Online]. Available: http://link.springer.com/10.1007/978-3-319-48944-5

[27] K. Xu, Y. Wang, P. Duan, and M. Li, "Resource scheduling management system of container cloud platform based on virtualization technology," in *Proceedings of the 2024 9th International Conference on Cyber Security and Information Engineering (ICCSIE '24)*, 2024, pp. 144–149.

[28] Z. Kozhirbayev and R. O. Sinnott, "A performance comparison of container-based technologies for the Cloud," *Future Generation Computer Systems*, vol. 68, pp. 175–182, 2017. [Online]. Available: http://dx.doi.org/10.1016/j.future.2016.08.025

[29] T. Shi, H. Ma, and G. Chen, "Energy-Aware Container Consolidation Based on PSO in Cloud Data Centers," *2018 IEEE Congress on Evolutionary Computation, CEC 2018 - Proceedings*, pp. 1–8, 2018.

[30] M. Zakarya, "Energy and performance aware resource management in heterogeneous cloud datacenters," no. August 2017.

[31] K. Ye and Y. Ji, "Performance Tuning and Modeling for Big Data Applications in Docker Containers," *2017 IEEE International Conference on Networking, Architecture, and Storage, NAS 2017 - Proceedings*, 2017.

[32] J. Liu, S. Wang, A. Zhou, J. Xu, and F. Yang, "Sla-driven container consolidation with usage prediction for green cloud computing," *Frontiers of Computer Science*, vol. 14, pp. 42–52, 2020.

[33] E. Jafarnejad Ghomi, A. Masoud Rahmani, and N. Nasih Qader, "Load-balancing algorithms in cloud computing: A survey," *Journal of Network and Computer Applications*, vol. 88, pp. 50–71, 2017. [Online]. Available: http://dx.doi.org/10.1016/j.jnca.2017.04.007

[34] A. Tsertou, A. Amditis, E. Latsa, and I. Kanellopoulos, "Dynamic and synchromodal container consolidation : the cloud computing enabler," *Transportation Research Procedia*, vol. 14, pp. 2805–2813, 2016. [Online]. Available: http://dx.doi.org/10.1016/j.trpro.2016.05.345

[35] J. Bhimani, Z. Yang, M. Leeser, and N. Mi, "Accelerating big data applications using lightweight virtualization framework on enterprise cloud," *2017 IEEE High Performance Extreme Computing Conference, HPEC 2017*, 2017.

[36] N. NaiK, "Docker Container Based Big Data Processing System In Multiple Clouds for Everyone," vol. 29, no. 3, pp. 712–717, 2017.

[37] F. Wan, X. Wu, and Q. Zhang, "Chain-Oriented Load Balancing in Microservice System," *2020 World Conference on Computing and Communication Technologies, WCCCT 2020*, pp. 10–14, 2020.

[38] W. Ren, W. Chen, and Y. Cui, "Dynamic Balance Strategy of High Concurrent Web Cluster Based on Docker Container," *IOP Conference Series: Materials Science and Engineering*, vol. 466, no. 1, 2018.

[39] M. Awad, A. Leivadeas, and A. Awad, "Multi-resource predictive workload consolidation approach in virtualized environments," *Computer Networks*, vol. 237, p. 110088, 2023.

[40] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, "Autonomic Vertical Elasticity of Docker Containers with ELASTICDOCKER," *IEEE International Conference on Cloud Computing, CLOUD*, vol. 2017-June, pp. 472–479, 2017.

[41] Z.-h. Zhan and J. Zhang, "Adaptive particle swarm optimization," in *Ant Colony Optimization and Swarm Intelligence*, ser. Lecture Notes in Computer Science, vol. 5217. Berlin, Heidelberg: Springer, 2008, pp. 227–234.

[42] B. Liu, J. Li, W. Lin, W. Bai, and P. Li, "K-PSO : An improved PSO-based container scheduling algorithm for big data applications," no. October 2019, pp. 1–16, 2020.

[43] C. Shan, C. Wu, Y. Xia, Z. Guo, D. Liu, and J. Zhang, "Adaptive resource allocation for workflow containerization on kubernetes," *Journal of Systems Engineering and Electronics*, vol. 34, no. 3, pp. 723–743, 2023.

[44] R. Rabieyan, R. Yahyapour, and P. Jahnke, "Optimization of containerized application deployment in virtualized environments: a novel mathematical framework for resource-efficient and energy-aware server infrastructure," *The Journal of Supercomputing*, vol. 80, pp. 22 598–22 630, 2024.

[45] S. Sharma and R. Vishwakarma, "An efficient container scheduling framework for resource allocation in a cloud computing environments," *Communications on Applied Electronics*, vol. 7, no. 40, pp. 39–48, 2025.

[46] L. Wu and H. Xia, "Particle Swarm Optimization Algorithm for Container Deployment Particle Swarm Optimization Algorithm for Container Deployment," 2020.

[47] Y. Xue, B. Xue, and M. Zhang, "Self-adaptive particle swarm optimization for large-scale feature selection in classification," *ACM Transactions on Knowledge Discovery from Data*, vol. 13, no. 5, pp. 1–27, 2019.

[48] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. F. D. Rose, "Performance Evaluation of Container-based Virtualization for High Performance Computing Environments," 2013.

[49] A. Open, A. Journal, L. Xu, B. Song, and M. Cao, "An improved particle swarm optimization algorithm with adaptive weighted delay velocity," 2021. [Online]. Available: https://doi.org/10.1080/21642583.2021.1891153

[50] N. Nguyen and D. Bein, "Distributed MPI cluster with Docker Swarm mode," *2017 IEEE 7th Annual Computing and Communication Workshop and Conference, CCWC 2017*, 2017.

[51] P. Mohan, T. Jambhale, L. Sharma, S. Koul, and S. Koul, "Load Balancing using Docker and Kubernetes: A Comparative Study," *International Journal of Recent Technology and Engineering*, vol. 9, no. 2, pp. 782–792, 2020.

[52] J. Liu and S. Wang, "SLA-driven container consolidation with usage prediction for green cloud computing," pp. 1–11, 2019.

[53] Q. Li and Y. Fang, "Multi-algorithm collaboration scheduling strategy for docker container," *2017 International Conference on Computer Systems, Electronics and Control, ICCSEC 2017*, pp. 1367–1371, 2018.

[54] L. Li, J. Chen, and W. Yan, "A particle swarm optimization-based container scheduling algorithm of docker platform," *ACM International Conference Proceeding Series*, pp. 12–17, 2018.

[55] A. Khan, "Key Characteristics of a Container Orchestration Platform to Enable a Modern Application," *IEEE Cloud Computing*, vol. 4, no. 5, pp. 42–48, 2017.

[56] Y. Kang and R. Y. C. Kim, "Twister Platform for MapReduce Applications on a Docker Container," *2016 International Conference on Platform Technology and Service, PlatCon 2016 - Proceedings*, no. i, pp. 16–18, 2016.

[57] G. Goos, *Cyberspace Safety and Security*. Cham, Switzerland: Springer, 2019.

[58] G. P. Geethu and S. K. Vasudevan, "An in-depth analysis and study of Load balancing techniques in the cloud computing environment," *Procedia Computer Science*, vol. 50, pp. 427–432, 2015. [Online]. Available: http://dx.doi.org/10.1016/j.procs.2015.04.009

[59] J. Cito, G. Schermann, J. E. Wittern, P. Leitner, S. Zumberi, and H. C. Gall, "An Empirical Analysis of the Docker Container Ecosystem on GitHub," *IEEE International Working Conference on Mining Software Repositories*, pp. 323–333, 2017.

[60] J. Cito and H. C. Gall, "Using docker containers to improve reproducibility in software engineering research," *Proceedings - International Conference on Software Engineering*, vol. 1, pp. 906–907, 2016.

[61] M. Cerqueira De Abranches and P. Solis, "An algorithm based on response time and traffic demands to scale containers on a Cloud Computing system," pp. 343–350, 2016.

[62] G. Ambrosino, G. B. Fioccola, R. Canonico, and G. Ventre, "Container mapping and its impact on performance in containerized cloud environments," *Proceedings - 14th IEEE International Conference on Service-Oriented System Engineering, SOSE 2020*, pp. 57–64, 2020.

[63] Kubernetes Documentation, "Dynamic resource allocation," https://kubernetes.io/docs/concepts/scheduling-eviction/dynamic-resource-allocation/, 2026, accessed: 2026-05-24.

List of Publications

List of Publications derived from the Thesis

1. Zala, Manmitsinh Chandrasinh, and Patel, Jaykumar Shantilal, "Load Balancing using Particle Swarm Optimization based Algorithms in Docker Container Cloud Environment: A Comparative Analysis," International Journal of Intelligent Systems and Applications in Engineering, Vol. 12, No. 4, 2024, DOI: 10.17762/ijisae.v12i4.6533
2. Zala, Manmitsinh Chandrasinh, and Patel, Jaykumar Shantilal, "Adaptive Resource Optimization in Containerized Environments Using Particle Swarm Optimization and Decision Tree Classification," Journal of Information Systems Engineering and Management, Vol. 9, No. 4s, 2024, Published on December 30, 2024, DOI: 10.52783/jisem.v9i4s.11666.