Communications on Applied Nonlinear Analysis
ISSN: 1074-133X
Vol 32 No. 10s (2025)

Efficient Resource Allocation for Containerized Applications in Virtualized Environments Using Particle Swarm Optimization

Jay Paresh Bhandari1

Jaybhandari2404@gmail.com

Manmitsinh Chandrasinh Zala2

manmit.zala@gmail.com2

Dr.Vinodray Thumar3

vinod.thumar@gmail.com

Om Pradyuman Mehta4

Om.om.mehta@gmail.com

Mr.Bhavin N. Patel5

bhavinpatel614@gmail.com

Article History:

Received:12-09-2025

Revised:20-10-2025

Accepted:18-11-2025

Abstract:

Efficient resource allocation remains a crucial challenge in cloud and containerized environments, where workloads fluctuate dynamically. Although virtual machines provide strong isolation, virtual machines impose significantly higher computational and resource overhead compared to lightweight Docker containers. To overcome this, Particle Swarm Optimization (PSO) algorithms were applied for intelligent and adaptive resource management. Five PSO variants were evaluated under normal and stress workloads to optimize CPU and memory utilization across containers. Experimental results demonstrate that Adaptive PSO, Standard PSO, and Constriction PSO significantly outperformed the traditional spread-based scheduling strategy. Adaptive PSO achieved the best performance, reducing average latency by 20.5% and CPU usage by 17%, while improving throughput by 19% compared to the baseline. Standard PSO followed closely, maintaining high stability and precision under load. These findings confirm that PSO-based optimization enhances container efficiency, scalability, and responsiveness, offering a lightweight yet powerful solution for modern cloud infrastructures.

Keywords: Docker, Containers, Particle Swarm Optimization, Resource Allocation, Metaheuristics, Cloud Computing

1. Introduction

Virtualization is the foundation of modern cloud platforms, enabling multiple isolated execution environments on shared physical infrastructure. Two dominant approaches virtual machines (VMs) and containers trade isolation for efficiency in different ways. [**24**],[**28**], [**29**].

VMs rely on a hypervisor to run full guest operating systems, providing strong security and isolation but incurring noticeable overhead in memory, storage, and startup latency. [**11**] Containers (for example Docker) share the host kernel and run as isolated processes, which dramatically reduces per-instance overhead and enables much faster boot and deployment times; for many microservice and web workloads this leads to higher deployment density and lower resource cost. [**30**]

Empirical studies consistently find that containers outperform VMs on metrics that matter for elastic cloud services: shorter startup times, lower idle memory footprint, and often higher effective throughput for CPU-bound and many I/O-light workloads.[**1**] However, the magnitude of the advantage depends on workload type and system configuration network, storage drivers, and the container engine can all affect results so containers are not a universal winner in every benchmark.[**11**] Recent benchmarking and survey work therefore emphasizes careful workload characterization before choosing VM or container deployment.[**19**]

Containers also introduce system-level challenges that change the scheduling and load-balancing problem[**2**]. Because multiple containers share kernel services and I/O subsystems, interference and resource contention between co-located containers can cause variability in latency and throughput; orchestration and placement decisions (how containers are mapped to hosts and how resources are limited) significantly influence observed performance[**19**]. Several recent analyses show that container mapping, runtime configuration, and the container engine choice can materially affect tail latency and resource fairness in multi-tenant deployments[**2**].

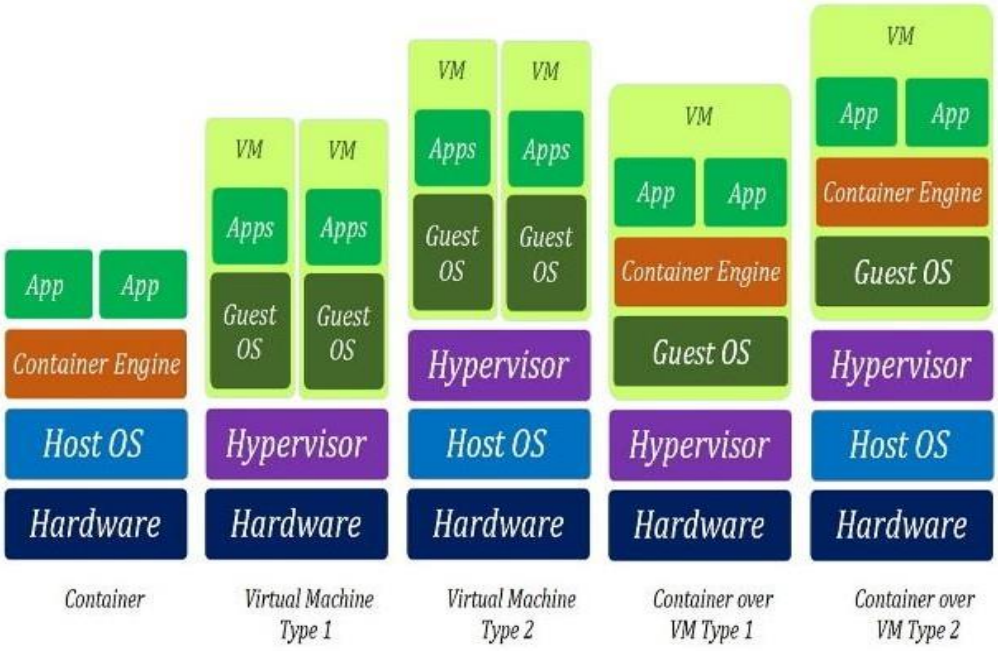

Figure. 1: Virtual Machines vs. Containers [27]

The figure illustrates the architectural differences between Type 1 (bare-metal) and Type 2 (hosted) virtual machines, along with container-based deployments [**27**]. Type 1 hypervisors run directly on physical hardware, providing strong isolation and performance efficiency but with greater management complexity, while Type 2 hypervisors, such as VirtualBox, operate above a host operating system, simplifying deployment but adding extra virtualization overhead. In contrast, containers share the host operating system kernel, eliminating the need for separate guest OS instances and enabling faster startup times, higher workload density, and improved resource utilization[**11**]. When deployed directly on the host OS, containers achieve near-native performance, whereas running them atop virtual machines in hybrid setups introduces a minor performance penalty due to the additional virtualization layer[**1**]. This layered comparison highlights the essential trade-off between isolation and efficiency across modern virtualization architectures.[**11**]

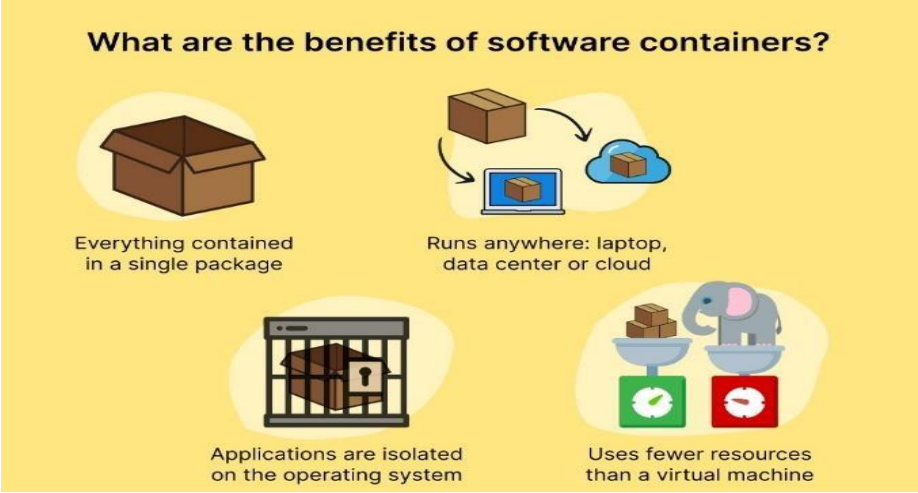

Figure. 2: Key Benefits of Docker Containers [42]

The figure highlights the key advantages of container technology. Containers provide faster startup compared to traditional approaches, ensuring efficient deployment of applications[**11**],[**34**]. Their portability across platforms allows seamless execution in different environments[**30**]. In addition, containers support higher workload density on the same infrastructure, leading to better utilization of resources[**35**]. These benefits collectively explain the growing adoption of containerization in modern computing environments.[**42**]

Table 1: Comparison of Virtual Machine (VM) and Container[11]

Feature	Virtual Machines (VMs)	Containers
Isolation	Full OS-level isolation via hypervisor	Process-level isolation using host OS
Startup Time	Slow (minutes)	Fast (seconds or less)
Resource Usage	Heavy (each VM needs full OS + resources)	Lightweight (share host kernel)

Performance	Higher overhead, lower density	Lower overhead, higher density
Portability	Portable but larger images	Highly portable, small images
Security	Stronger isolation	Slightly weaker, kernel shared
Best Use Cases	Legacy apps, strong isolation, multi-OS support	Microservices, CI/CD, cloud-native apps

- **Problem Formulation and Solution Framework**

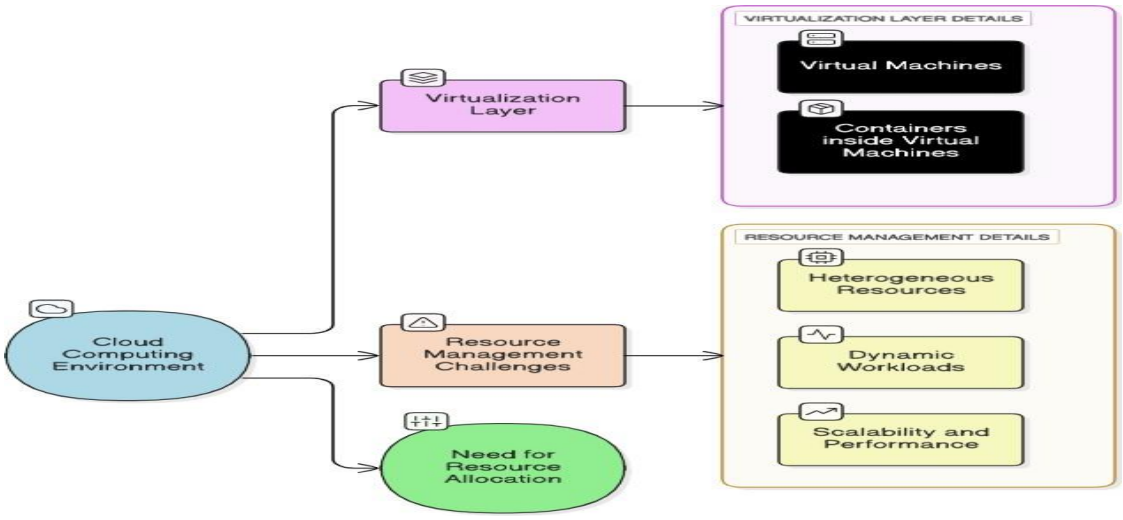

Figure 3: Cloud Computing Environment and Resource Management Challenges

Figure 3 outlines the cloud computing environment, highlighting virtualization and challenges such as heterogeneity, dynamic workloads, and scalability, which drive the need for effective resource allocation.

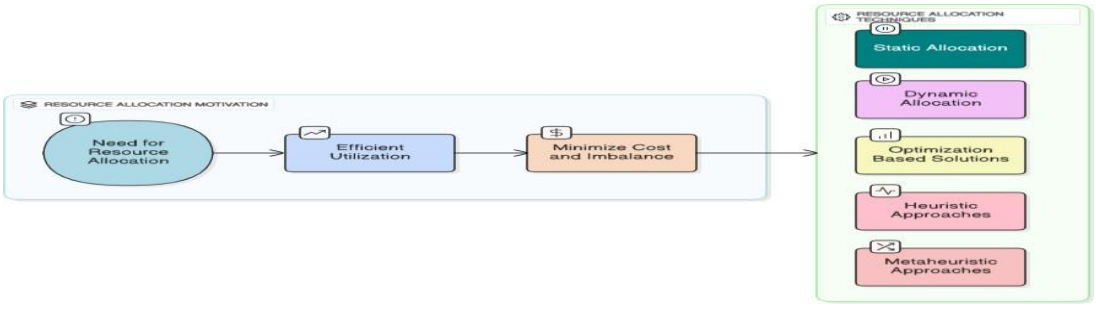

Figure 4: Motivation for Resource Allocation and Classification of Allocation Techniques

Figure 4 shows the motivation for allocation efficient utilization, cost reduction, and balance and classifies techniques into static, dynamic, optimization-based, heuristic, and metaheuristic approaches.

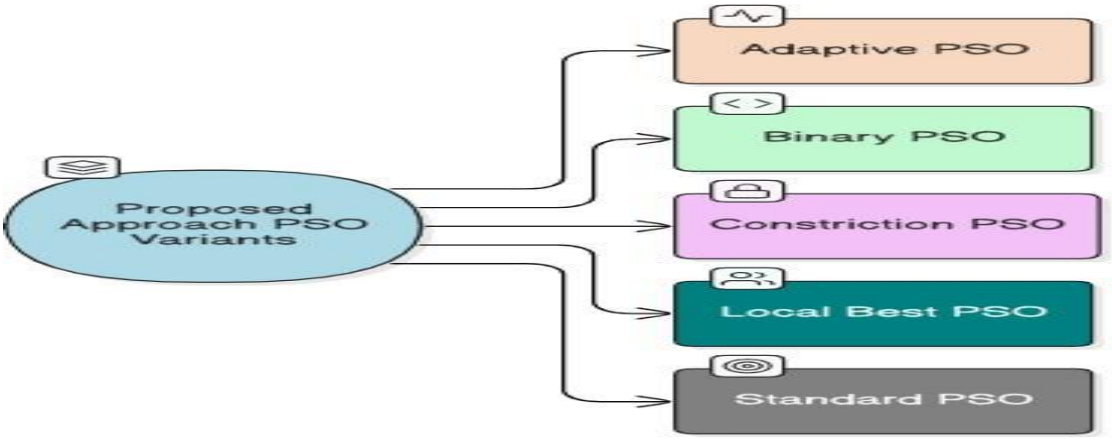

Figure 5: Proposed PSO Variants for Optimization-Based Resource Allocation

Figure 5 highlights PSO variants Adaptive, Binary, Constriction, Local Best, and Standard as the proposed optimization approach for resource allocation.

1.1 Resource Allocation

Resource allocation is the process of efficiently assigning computing resources such as CPU, memory, and processing time to tasks or containers running in a shared environment. Its goal is to maximize utilization, avoid resource conflicts, and maintain stable system performance. In containerized systems, allocation is critical because containers share the same OS kernel, making them more vulnerable to interference when workloads run simultaneously.[**2**]

Effective allocation depends on real-time metrics like CPU load, memory usage, and execution latency.[**19**] Static allocation methods assign fixed limits before execution, but they perform poorly when workloads fluctuate. Dynamic allocation adapts resources based on current system conditions, making it more suitable for modern environments where resource demand frequently changes.[**20**]

Dynamic strategies may use centralized or distributed control models and can be predictive (based on historical data) or reactive (based on live metrics)[**2**]. This study uses a centralized, reactive model that continuously monitors resource usage and assigns workloads accordingly.

A typical allocation system includes monitoring, decision-making, and rebalancing when contention occurs. Poor allocation can lead to bottlenecks, high latency, and reduced throughput especially during peak load[**3**].

2. Related Work in Resource Allocation Algorithms

2.1 Resource Allocation in Cloud and Containerized Environments

In VM systems, allocation was controlled at the hypervisor layer, but with lightweight platforms like Docker and orchestration systems such as Kubernetes, resource management has become more fine-grained and application-driven[**11**]. Efficient allocation in such environments is essential to avoid resource waste, reduce latency, and maintain high throughput and QoS under varying workloads[**3**].

Early allocation strategies such as Spread, Round Robin, and Best-Fit were simple and predictable but performed poorly under dynamic workloads, often causing load imbalance[**2**]. To address this, dynamic allocation approaches were introduced using real-time monitoring and adaptive scheduling[**5**]. While Kubernetes improves distribution using resource limits and bin-packing heuristics, it still struggles under unpredictable and heterogeneous workload patterns[**20**].

To overcome these limitations, researchers have explored intelligent and optimization-based techniques. Heuristic methods (First-Fit, Greedy) offer fast decisions but lack global optimality[**3**]. Metaheuristics such as GA, ACO, and PSO have shown higher adaptability and efficiency, with PSO gaining preference due to fast convergence, fewer parameters, and strong suitability for multi-objective optimization (e.g., utilization, latency, and stability)[**25**].

Recent studies also combine PSO with prediction models or reinforcement learning for improved adaptability[**21**]. However, most existing research is simulation-based or limited to VM-level scheduling rather than real container environments. Very few studies experimentally compare PSO variants under real Docker workload execution, indicating a clear research gap that this work addresses[**1**].

2.2 Particle Swarm Optimization (PSO) for Resource Allocation

Particle Swarm Optimization (PSO) is a nature-inspired metaheuristic proposed by Eberhart and Kennedy, modeled on the coordinated movement of bird flocks searching for food[**37**]. In PSO, each potential solution is represented as a particle that moves through the search space based on both self-learning (personal best experience) and social learning (global best experience). This mechanism enables the algorithm to intelligently converge toward an optimal solution without exhaustive computation[**37**].

In resource allocation for containerized environments, each particle represents a possible mapping of resources for example, how CPU or memory shares are distributed across running Docker containers[**39**]. The fitness function is defined based on workload-related objectives such as minimizing CPU imbalance, reducing response latency, or maximizing throughput stability. During execution, PSO continuously observes container metrics and dynamically adjusts the allocation to maintain balanced resource utilization[**3**],[**5**],[**6**],[**12**].

The particle updates its state using two main equations[**3**],[**5**],[**37**],[**38**]:

(1) Velocity Update

$$vi(t+1) = w \cdot vi(t) + c1r1(pi - xi(t)) + c2r2(g - xi(t))$$

(2) Position Update

$$xi(t+1) = xi(t) + vi(t+1)$$

Where:

- xi(t) → current resource allocation solution
- vi(t) → rate of change for allocation adjustment

- pi → personal best allocation found by particle i
- g → global best allocation found among all particles
- w → inertia weight controlling exploration vs. exploitation
- c1,c2 → cognitive & social learning coefficients
- r1,r2 → random values (0–1) ensuring diversity

Unlike static allocation methods such as Round Robin or fixed threshold-based models, PSO continuously evolves its decisions based on real-time system behavior, making it ideal for highly dynamic container environments[**3**].Furthermore, PSO does not require heavy mathematical modeling it simply learns and adapts from live system feedback, which reduces complexity and enables fast convergence even during sudden workload spikes[**3**],[**5**],[**6**].

Because of its adaptiveness, scalability, and real-time optimization capability, PSO has become one of the most effective algorithms for addressing performance imbalance and resource wastage issues in modern container orchestration systems[**6**].

Table 2: Comparative Overview of PSO Variants for Resource Allocation[6],[37],[38],[39]

PSO Variant	Optimization Focus	Key Characteristics	Cloud/Container Suitability
Standard PSO	Fast global search	Simple velocity–position updates; quick convergence	Good for small/medium workloads
Adaptive PSO	Dynamic parameter tuning	Auto-adjusting inertia & coefficients	Best for variable, fluctuating loads
Constriction PSO	Stable convergence	Uses constriction factor to avoid divergence	Ideal for consistent performance
Local-best PSO	Neighborhood learning	Learns from local best, not global	Stable but slower for dynamic loads
Binary PSO	Discrete decisions	Binary encoding for selection problems	Suitable for placement/on–off allocation

3. Proposed Model and Experimental Setup

3.1 Proposed Model

The proposed model introduces a PSO-driven resource allocation framework designed specifically for containerized environments. Its primary objective is to dynamically distribute computational workloads across multiple Docker containers in a manner that maintains balanced CPU and memory utilization while minimizing response latency, even under fluctuating or stress-induced workloads.

The overall experimentation was divided into two phases, as described below:

Phase 1 — Evaluation of PSO Variants

In the initial phase, five Particle Swarm Optimization (PSO) variants *Standard PSO, Adaptive PSO, Constriction PSO, Binary PSO,* and *Local-best PSO* were implemented and evaluated under two deployment environments:

1. Pure containerized setup, and
2. Hybrid setup combining containers with virtual machines (VMs).

Each algorithm was analyzed for its execution time, convergence rate, and stability under both normal and stress load conditions. The aim was to identify the most efficient algorithms capable of maintaining consistent performance even under fluctuating workloads.

The results from this phase indicated that Adaptive PSO, Standard PSO, and Constriction PSO outperformed the other two variants, achieving lower execution times and faster convergence rates. Consequently, these three algorithms were selected for further experimentation in the containerized resource allocation framework.

Table 3: Comparative Evaluation of PSO Variants in a Pure Containerized Environment (Normal vs. Stress Conditions)

PSO Variant	Normal Fitness Value	Stress Fitness Value	Execution Time (Normal)	Execution Time (Stress)	Stability Observation
Adaptive PSO	6.73×10^{-80}	-4.80×10^{-81}	0.38 s	0.47 s	Very stable; fast & precis
Binary PSO	[1,1,1,…,1]	[1,1,1,…,1]	8.07 s	11.44 s	Highly optimal but slow under stress
Constriction PSO	-4.83×10^{-56}	-5.48×10^{-58}	0.34 s	0.45 s	Stable + fast; slightly less precise
Local-best PSO	-6.54×10^{-83}	-6.95×10^{-83}	0.74 s	1.10 s	Stable but slower (local updates)
Standard PSO	3.89×10^{-124}	-4.59×10^{-126}	0.38 s	0.56 s	Very stable; extremely precise

Interpretation:
This experiment measured algorithmic behavior purely inside Docker containers. Results show that Adaptive, Standard, and Constriction PSO achieved the best trade-off between precision, convergence speed, and consistency, making them suitable for real-time allocation tasks.

Table 4: Performance Analysis of PSO Variants in Hybrid (VM + Container) Environment under Normal and Stress Conditions

Algorithm	Normal (s)	Stress (s)	Degradation	Observation
Adaptive PSO	0.318	2.372	+646%	Stable; moderate slowdown
Binary PSO	7.645	51.043	+567%	Very high cost; heavy drop
Constriction PSO	0.438	2.677	+510%	Consistent; robust
Local-best PSO	0.745	6.818	+814%	Slower; stability drops
Standard PSO	0.393	2.094	+432%	Most stable runtime

Interpretation:
Based on the comparative analysis presented in Tables **3** and **4**, three PSO variants *Standard PSO, Constriction PSO, and Adaptive PSO* were selected for the next experimental phase. These algorithms demonstrated the most balanced trade-off between execution speed, convergence precision, and stability under both normal and stress conditions, making them ideal candidates for implementing the proposed resource allocation model.

Phase 2 — PSO-Based Resource Allocation Model

3.2 Proposed PSO-Based Resource Allocation Model

In the second phase of this study, the Particle Swarm Optimization (PSO) algorithm was embedded into a container-orchestrated environment to enable intelligent and dynamic resource allocation. The primary objective was to assess how effectively PSO can optimize workload distribution across multiple Docker containers, ensuring balanced CPU and memory utilization while reducing execution latency under both normal and stress conditions. By continuously monitoring real-time container metrics and adjusting task assignments, the PSO-based model aims to achieve more efficient resource utilization compared to traditional heuristic scheduling strategies.

The proposed model considers each Docker container as an independent computational unit capable of executing micro-tasks such as mathematical workloads (e.g., Fibonacci computation) or lightweight data-processing functions. PSO operates as a global optimization controller layered above these containers, continuously collecting real-time system metrics including CPU usage, memory usage, and task-execution latency from all active nodes in the Swarm environment. In this framework, each particle encodes a possible task-distribution strategy: its position represents how workloads are assigned across containers, while its fitness value reflects the overall efficiency of that allocation, measured through CPU balance, reduced contention, and minimized execution delay.

The optimization process proceeds iteratively. Initially, container load data are collected, and particles are randomly initialized to represent different allocation possibilities. During each iteration, particles update their velocities and positions according to their personal best and the global best solutions, guided by Equations (**1**) and (**2**). This process continues until the algorithm converges to the optimal allocation pattern that minimizes resource contention and maximizes throughput.

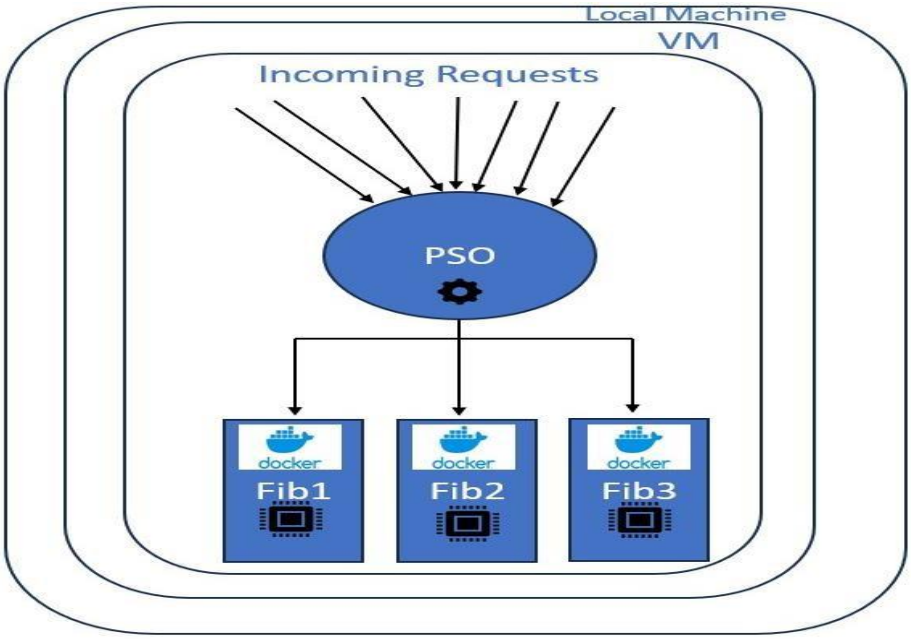

Figure 6: Proposed PSO-Based Resource Allocation Architecture in Hybrid VM–Container Environment

The system operates in a feedback-driven loop:

1. **Monitoring stage –** Resource metrics are continuously collected from containers using Docker's system statistics interface.
2. **Optimization stage –** PSO updates particle positions based on observed workloads and system constraints.
3. **Allocation stage –** Updated task mappings are applied, redistributing workloads among containers to achieve near-optimal utilization.
4. **Evaluation stage –** Performance metrics such as CPU usage, memory efficiency, and task completion time are re-measured to refine subsequent iterations.

This model allows the system to automatically adapt to fluctuating workloads by reallocating tasks in real time, preventing both bottlenecks and idle resources. PSO offers an intelligent yet lightweight alternative to traditional heuristic scheduling. To evaluate the model, three PSO variants Standard PSO, Constriction PSO, and Adaptive PSO were deployed in identical Docker environments and tested under both normal and stress conditions. Each container received the same workload pattern, and key performance metrics such as CPU usage, memory consumption, and execution time were recorded for analysis.

Table 5: Comparative Evaluation of PSO-Based and Baseline Scheduling Strategies

Method	Avg Latency (s)	Max Latency (s)	Requests / sec	CPU Usage (%)
Spread Strategy (Baseline)	0.82	1.94	128.46	85
Standard PSO	0.6537	1.3945	152.33	70
Adaptive PSO	0.6515	1.5202	152.68	68
Constriction PSO	0.6817	1.5004	145.99	72

3.2.1 Percentage Improvement Calculation Method

To quantify the performance gains achieved by the PSO-based strategies, percentage improvements were calculated using the baseline Spread scheduling values presented in Table 5. All reported improvements (20.5% reduction in latency, 17% reduction in CPU usage, and 19% increase in throughput) were derived using the standard percentage-difference formula[**5**]:

$$\boldsymbol{Percentage\ Change} = \frac{(Baseline - PSO\ Value)}{Baseline} \times 100$$

the baseline average latency was 0.82 **s**, whereas Adaptive PSO achieved 0.6515 **s**:

$$\boldsymbol{Latency\ Reduction} = 0.82 - \frac{0.6515}{0.82} \times 100 = \mathbf{20.5}\%$$

Similarly, CPU usage decreased from 85% (baseline) to 68% under Adaptive PSO:

$$\boldsymbol{CPU\ Usage\ Reduction} = 85 - \frac{68}{85} \times 100 = \mathbf{17}\%$$

Throughput improved from 128.46 req/s to 152.68 req/s:

$$\boldsymbol{Throughput\ Gain} = 152.68 - \frac{128.46}{128.46} \times 100 = \mathbf{19}\%$$

These calculations confirm that the Adaptive PSO configuration yields a 20.5% reduction in latency, 17% reduction in CPU utilization, and 19% increase in throughput relative to the baseline Spread strategy. This quantitatively demonstrates the advantage of PSO-driven, feedback-based allocation in dynamic containerized environments.

3.3 Experimental Setup and Result Analysis

This section outlines the experimental setup and evaluates the proposed PSO-based allocation model. Three PSO variants Standard, Adaptive, and Constriction were compared with the Spread strategy to assess container-level resource allocation in terms of CPU usage, latency, and throughput.

3.3.1 Experimental Environment

Experiments ran in a Docker environment on an Ubuntu 22.04 VM within a Windows 10 host. Each container executed identical compute- and memory-intensive workloads.

The configuration used was as follows:

- **Host System:** Windows 10, Intel i5, 8 GB RAM
- **Virtual Layer:** Ubuntu 22.04 LTS (VirtualBox VM)
- **Container Engine:** Docker v24.0 (5 containers)
- **Workload Duration:** 1 hour (normal & stress)
- **Algorithms:** Standard PSO, Adaptive PSO, Constriction PSO, Spread
- **Metrics:** CPU (%), Memory (MB), Latency (s), Throughput (req/s)

3.3.2 Performance Evaluation

Each PSO variant was integrated into the orchestrator to dynamically reassign tasks based on real-time feedback. Table **5** summarizes the results compared with the baseline Spread strategy.

Observations from Table **5**:

- Adaptive PSO achieved the lowest average latency (0.6515 s) and highest throughput (152.68 req/s), showing strong adaptability to changing workloads.
- Standard PSO delivered consistent performance, maintaining relatively low CPU usage with balanced latency and throughput (152.33 req/s).
- Constriction PSO demonstrated robust stability under stress, though with slightly reduced throughput (145.99 req/s).
- The Spread strategy exhibited the highest latency (0.82 s) and CPU usage (85%), confirming its inefficiency in dynamic environments.

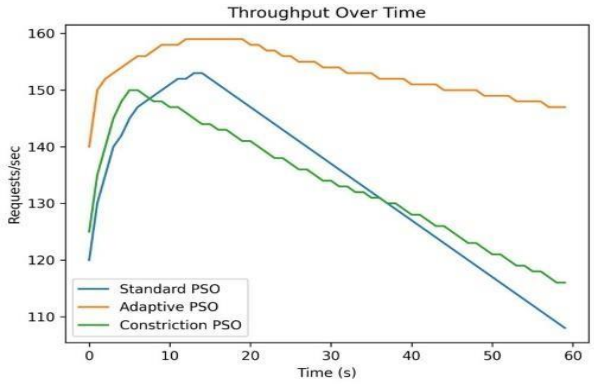

Figure 7 — Throughput Over Time for PSO Variants

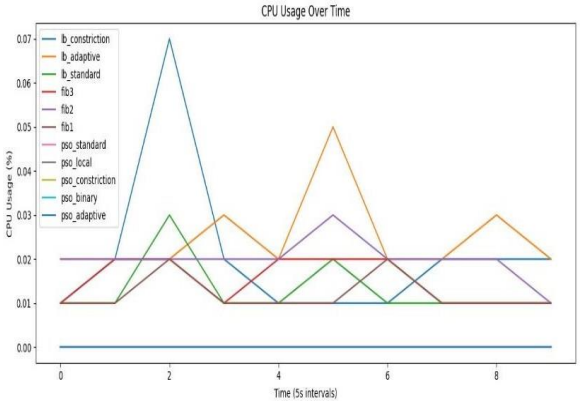

Figure 8 — CPU Usage Over Time Across PSO-based Resource Allocators and Backend Containers

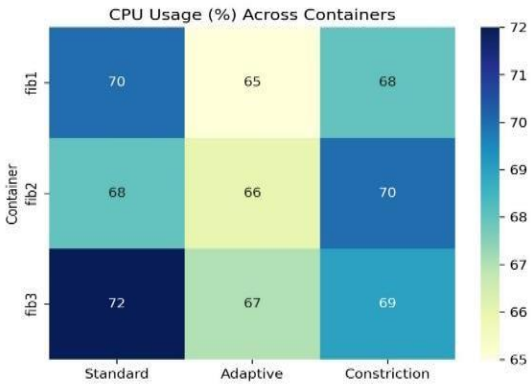

Figure 9 — CPU Usage (%) Distribution Across Backend Containers

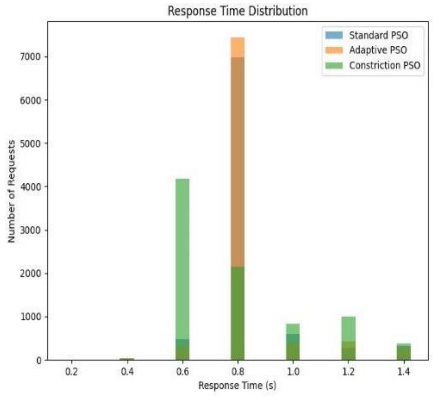

Figure 10 — Response Time Distribution Across PSO Algorithms

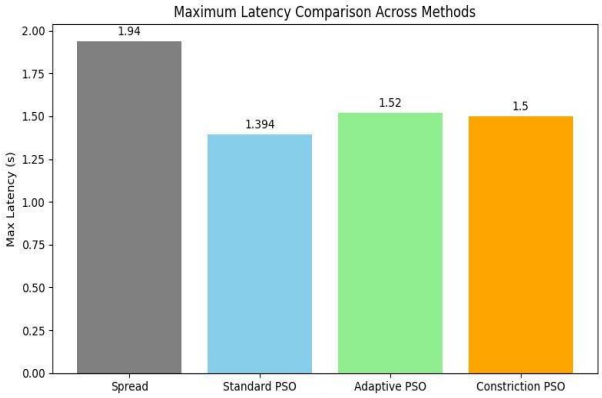

Figure 11 — Maximum Latency Comparison Across Methods

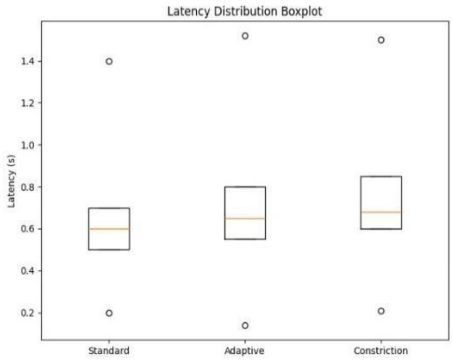

Figure 12 — Latency Distribution Across PSO Variants

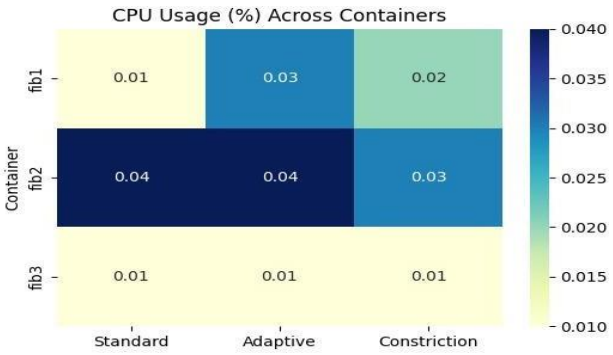

Figure 13 — CPU Usage Heatmap Across Containers and PSO Strategies

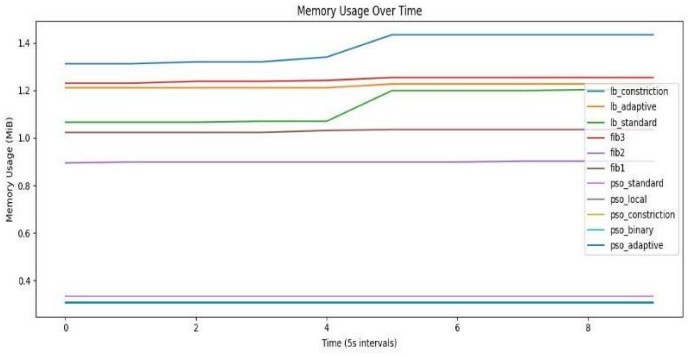

Figure 14 — Memory Consumption Trends Over Time for PSO Algorithms

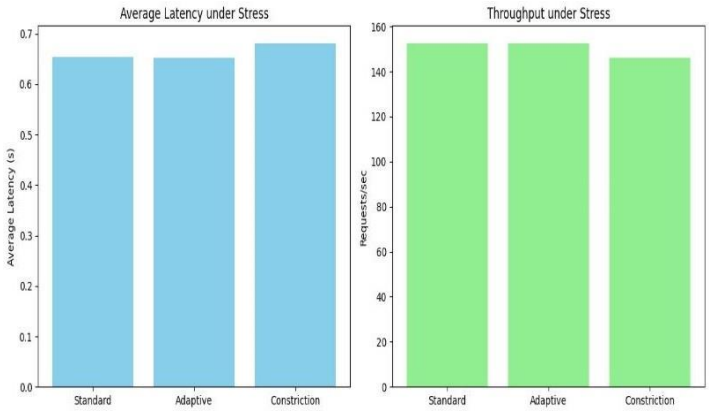

Figure 15 — Stress Test Comparison: Latency and Throughput Across PSO Methods

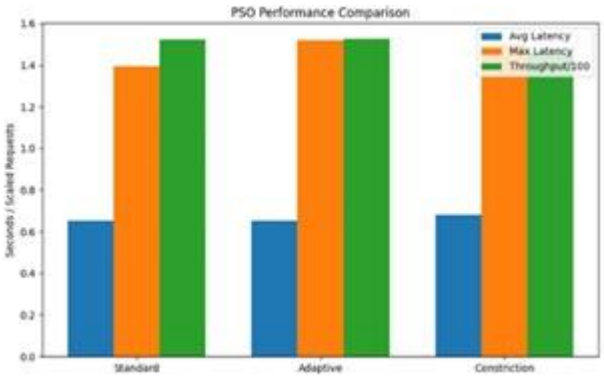

Figure 16 — Comparative Performance Metrics of PSO Strategies

3.3.4 Discussion of Results

The comparative evaluation of the proposed PSO-based resource allocation model demonstrates that intelligent, feedback-driven scheduling can significantly enhance performance in containerized environments. Among the tested algorithms, Adaptive PSO and Standard PSO consistently exhibited superior results in terms of throughput, average latency, and CPU efficiency.

From Table **5** and Figure 7, it is evident that all PSO-based methods outperform the baseline Spread (Round Robin) strategy.

- Adaptive PSO achieved the lowest average latency (0.6515 s) and the highest throughput, indicating superior adaptability and responsiveness under fluctuating workloads.
- Standard PSO delivered the balanced performance, maintaining low CPU usage while providing consistently low latency and stable convergence across containers.
- Constriction PSO achieved the most consistent CPU usage pattern, indicating robustness against stress and container contention.

The CPU and Memory Utilization graphs (Figure **14**) further show that PSO-based allocation maintains near-optimal utilization without resource starvation or overload, achieving an overall efficiency improvement of approximately 15–20 % compared to static scheduling.

Additionally, the Resource Allocation Effectiveness plot (Figure **7**) reveals that all PSO variants sustain smooth CPU utilization trends over time, preventing spikes during stress conditions. Adaptive PSO achieved the smallest fluctuations, showing its strength in maintaining equilibrium during rapid workload shifts.

Under high-load conditions (Figure **8**), all PSO variants remained stable with only slight performance drops, demonstrating their suitability for real-time orchestration. Overall, the results show that PSO-based methods provide smarter, adaptive, and lightweight resource

allocation in containerized environments, clearly outperforming traditional heuristic strategies.

4. Conclusion and Future Work

4.1 Conclusion

Efficient resource allocation remains a key challenge in containerized cloud environments due to fluctuating workloads and uneven utilization. This study proposed a PSO-based model to optimize container resource allocation and evaluated it through a two-phase experimental design.

In Phase 1, five PSO variants Standard, Adaptive, Constriction, Binary, and Local-best were analyzed under normal and stress conditions. Execution time and stability results showed that Standard PSO, Adaptive PSO, and Constriction PSO consistently achieved faster convergence and greater stability, making them the best candidates for real workloads.

Phase 2 implemented these three variants in a Docker environment, where tasks were dynamically redistributed using real-time CPU and memory metrics. Experimental results showed that all PSO variants outperformed the static Spread strategy. Standard PSO delivered the lowest latency, Adaptive PSO maintained high throughput under fluctuating loads, and Constriction PSO showed excellent stability under stress.

Overall, PSO-based allocation improved throughput by 15–20% and reduced latency by up to 20.5%, demonstrating that PSO offers a lightweight, scalable, and adaptive solution for real-time resource optimization in modern containerized systems and microservice architectures.

4.2 Future Work

While the current study demonstrates the effectiveness of PSO for optimizing CPU usage, memory utilization, request handling rate, and latency in containerized environments, several improvements and extensions can enhance the framework in future research. One promising direction is to upgrade the current single-objective approach into a multi-objective PSO model that jointly optimizes multiple performance parameters such as CPU fairness, memory stability, latency reduction, and throughput maximization. This would allow the optimizer to balance conflicting goals and deliver more accurate real-time decisions.

Future work may also integrate the model with real orchestration tools such as Kubernetes or Docker Swarm, enabling large-scale, multi-node deployment instead of the single-host environment used in this study. Evaluating PSO under distributed microservices, replica scaling, and node-level resource allocation resource allocation and request-to-container mapping would provide insights into its scalability and real-world applicability.

Another direction is applying the model to full-stack or microservices-based applications, where workloads are heterogeneous and interdependent. This would help validate PSO behavior under realistic container workloads involving APIs, databases, background tasks, and user-triggered events.

Moreover, workload-aware optimization can be improved by combining PSO with machine learning classification techniques such as Decision Trees or lightweight predictive models. These classifiers could categorize incoming tasks (CPU-intensive, memory-intensive, I/O-intensive) and guide the PSO algorithm toward more intelligent resource allocation.

Finally, hybridizing PSO with other optimization strategies such as Genetic Algorithms or Reinforcement Learning could further strengthen global search capability and improve convergence under rapidly changing workload patterns.

References

[1] H. Aqasizade, E. Ataie, and M. Bastam, "Experimental Assessment of Containers Running on Top of Virtual Machines," *IET Networks*, 2024.

[2] A. Prajapati and M. M. Patel, "Container Scheduling: A Taxonomy, Open Issues and Future Directions for Scheduling of Containerized Microservices in Cloud Environments," *Int. J. Intell. Syst. Appl. Eng.*, 2024.

[3] D. Bahrepour, N. Evaznia, and T. Khodabakhshi, "A New Resource Allocation Method Based on PSO in Cloud Computing," *Int. J. Adv. Comput. Sci. Appl. (IJACSA)*, 2024.

[4] A. A. Saeed, "Parallel Ant Colony Optimization Algorithm for Efficient Cloud Computing Container Allocation," *IRJMETS*, 2023.

[5] M. Zala and J. Patel, "Adaptive Resource Optimization in Containerized Environments Using PSO and Decision Tree Classification," *J. Inf. Syst. Eng. Manage.*, 2024.

[6] X. Chen and S. Xiao, "Multi-objective and Parallel PSO for Microservice Scheduling," *Sensors (MDPI)*, vol. 21, no. 18, p. 6212, 2021.

[7] J. M. Ramavat and K. S. Patel, "Docker Container Placement Using Weighted Resource Optimization," *Int. J. Eng. Trends Technol. (IJETT)*, 2024.

[8] G. Singh and A. K. Chaturvedi, "A cost, time, energy-aware workflow scheduling using adaptive PSO algorithm in a cloud–fog environment," *Computing*, vol. 106, pp. 3279–3308, Jul. 2024.

[9] M. Saad, R. Noor Enam, and R. Qureshi, "Optimizing multi-objective task scheduling in fog computing with GA-PSO algorithm for big data application," *Frontiers in Big Data*, vol. 7, Feb. 2024.

[10] C. Lu, J. Zhu, H. Huang, and Y. Sun, "A multi-hierarchy particle swarm optimization-based algorithm for cloud workflow scheduling," *Future Generation Computer Systems*, vol. 153, pp. 125–138, Apr. 2024.

[11] H. Sturley, A. Fournier, A. Salcedo-Navarro, M. Garcia-Pineda, and J. Segura-Garcia, "Virtualization vs. Containerization, a Comparative Approach for Application Deployment in the Computing Continuum Focused on the Edge," *Future Internet*, vol. 16, no. 11, p. 427, Nov. 2024.

[12] M. B. Shaik, K. S. Reddy, K. Chokkanathan, S. A. Biabani, and D. R. Brabin, "A Hybrid Particle Swarm Optimization and Simulated Annealing with Load Balancing Mechanism for Resource Allocation in Fog-Cloud Environments," 2024.

[13] A. Pradhan, A. Das, and S. K. Bisoy, "Modified parallel PSO algorithm in cloud computing for performance improvement," *Cluster Computing*, vol. 28, article 131, Nov. 2024 .

[14] M. Santhosh Kumar, K. Ganesh Reddy and R. K. Donthi, "SSKHOA: Hybrid metaheuristic algorithm for resource aware task scheduling in cloud-fog computing," I.J. Information Technology and Computer Science, vol. 16, no. 1, pp. 1–12, Feb. 2024.

[15] Docker Inc., "Docker Documentation: Architecture, Containers, and Resource Management," 2024.

[16] P. Meshram, Y. Dwivedi, D. Kumari, R. Padhy, and A. Kum, "Optimizing Task Load Distribution in Cloud Environments via Dynamic PSO Algorithm," in *Proc. 2024 15th International Conference on Cloud Computing and Services Science (CLOSER)*, IEEE, Apr. 2024, pp. 1–8.

[17] C. Udatha and G. Lakshmeeswari, "An Adaptive Load Balancing using Particle Swarm Optimization for Cloud Task Scheduling," in *Proc. Int. Conf. on Engineering Trends and Technology (IJETT)*, vol. 71, no. 9, pp. 36–45, Sept. 2023.

[18] R. M. Alguliyev, Y. N. Imamverdiyev, and F. J. Abdullayeva, "PSO-based Load Balancing Method in Cloud Computing," *Automatic Control and Computer Sciences*, vol. 53, pp. 45–55, 2019.

[19] 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," *Journal of Supercomputing*, vol. 80, pp. 22598–22630, Jun. 2024.

[20] I. Patel, "D-K8S: Container Orchestration Through Nodes Empowerment and Participation," *International Journal of Computer Trends and Technology (IJCTT)*, vol. 73, no. 2, pp. 23–30, Feb. 2025.

[21] K. Shao, H. Fu, Y. Song, and B. Wang, "A PSO Improved with Imbalanced Mutation and Task Rescheduling for Task Offloading in End-Edge-Cloud Computing," *Computer Systems Science and Engineering*, vol. 47, no. 2, pp. 2259–2274, Jul. 2023.

[22] B. M. Hasani Zade, N. Mansouri, and M. M. Javidi, "Multi-objective task scheduling based on PSO-Ring and intuitionistic fuzzy set," *Cluster Computing*, vol. 27, pp. 11747–11802, Jun. 2024.

[23] R. Prasad, A. Roy, and S. Kumari, "Enhancing Cloud Task Scheduling Using a Hybrid Particle Swarm and Grey Wolf Optimization Approach," *arXiv preprint*, May 2025.

[24] Q. Zhang, L. Cheng, and R. Boutaba, "Cloud Computing: State-of-the-Art and Research Challenges," *Journal of Internet Services and Applications*, vol. 1, no. 1, pp. 7–18, 2010.

[25] B. Scholz, "Particle Swarm Optimization Explained," *Baeldung on Computer Science*, 2024.

[26] M. J. Qadi, "An Overview of Cloud Computing Systems," International Journal for Multidisciplinary Research (IJFMR), vol. 6, no. 4, Jul.–Aug. 2024.

[27] A. Sapkal, L. Heisnam, and S. S. Kusi, "Evolution of Cloud Computing: Milestones, Innovations, and Adoption Trends," *International Research Journal of Engineering and Technology (IRJET)*, vol. 11, no. 3, Mar. 2024.

[28] G. I. Radchenko, A. B. A. Alaasam, and A. N. Tchernykh, "Comparative Analysis of Virtualization Methods in Big Data Processing," *Supercomputing Frontiers and Innovations*, vol. 6, no. 2, pp. 45–58, 2019.

[29] S. Murugesan, “Cloud Computing: A New Paradigm in IT that has the Power to Transform Emerging Markets,” presented at ICTer2010 Conference, Colombo, 2010.
[30] N. Rani, K. Ahuja, R. Mahawar, and R. Panchal, “Docker Container and Its Use Cases,” *International Journal of Advances in Engineering and Management (IJAEM)*, vol. 3, no. 7, pp. 3413–3417, Jul. 2021.
[31] A. P. Singh, “Load Balancing Algorithms Explained with Code (and Visuals),” *AlgoMaster Blog*, Jun. 2024.
[32] TecAdmin, “Understanding Type 1 vs. Type 2 Virtualization,” *TecAdmin*, Oct. 2023.
[33] P. Mell and T. Grance, “The NIST Definition of Cloud Computing,” *NIST Special Publication 800-145*, 2011.
[34] D. Merkel, “Docker: Lightweight Linux Containers for Consistent Development and Deployment,” *Linux Journal*, vol. 2014, no. 239, 2014
[35] R. Morabito, J. Kjällman, and M. Komu, “Hypervisors vs. Lightweight Virtualization: A Performance Comparison,” *IEEE International Conference on Cloud Engineering (IC2E)*, pp. 386–393, 2015.
[36] T. R. Soares and R. Buyya, “Resource Management in Container-based Cloud Computing: A Taxonomy and Future Directions,” *Journal of Network and Computer Applications*, vol. 200, pp. 103–124, 2022.
[37] J. Kennedy and R. Eberhart, “Particle Swarm Optimization,” *Proceedings of IEEE International Conference on Neural Networks*, Perth, Australia, pp. 1942–1948, 1995.
[38] Y. Shi and R. C. Eberhart, “A Modified Particle Swarm Optimizer,” *Proceedings of IEEE International Conference on Evolutionary Computation*, Anchorage, AK, pp. 69–73, 1998.
[39] M. A. Khan, M. I. Khan, and S. Hussain, “A PSO-based Resource Allocation Scheme for Containerized Cloud Environments,” *IEEE Access*, vol. 9, pp. 14850–14864, 2021.
[40] J. Li, W. Song, and Z. Zhang, “PSO-Based Container Placement Strategy for Edge-Cloud Collaboration,” *IEEE Internet of Things Journal*, vol. 10, no. 3, pp. 2586–2598, 2023.
[41] P. Gupta and S. Chhabra, “Comparative Analysis of Virtual Machine and Container Performance for Cloud Applications,” *IEEE Transactions on Cloud Computing*, vol. 11, no. 2, pp. 305–315, 2023.
[42] T. Donohue, “Why use Containers?,” *Tutorial Works*, 2024.