International Journal of Engineering Trends and Technology
ISSN: 2231-5381/ https://doi.org/10.14445/22315381/IJETT-V74I3P110

Volume 74 Issue 3, 129-140, March 2026
© 2026 Seventh Sense Research Group®

*Original Article*

# An Optimized Container Scheduling Algorithm for Kubernetes using Maximization of Resource Utilization

Vidhi Sutaria[1], Dharmendra Bhatti[2]

[1]*Asha M Tarsadia Institute of Computer Science and Technology, Uka Tarsadia University, Gujarat, India.*
[2]*IT Head, Uka Tarsadia University, Gujarat, India.*

[1]*Corresponding Author : vidhi.sutaria@utu.ac.in*

Received: 25 August 2025 Revised: 23 January 2026 Accepted: 29 January 2026 Published: 28 March 2026

***Abstract* - *Container scheduling is a key issue in cloud computing, as it determines where and how containers are run to optimize performance and resource utilization. However, a notable gap remains in research, particularly in exploring specific scheduling parameters that could lead to more effective solutions. In this research, discuss the problem of container scheduling and identify underexplored factors that influence scheduling efficiency and costs. This study comprises a comprehensive literature review, a comparison of the proposed method with existing approaches, and experimental validation using a standard data set and a newly introduced approach to achieve an effective container scheduling solution. The results highlight the potential of the proposed approach to improve scheduling strategies in containerized environments.*

***Keywords* - *Cloud Computing, Container Technology, Heuristic Approach, Kubernetes, Machine Learning.*

## 1. Introduction

In today's era, virtualization is crucial because it offers many benefits-storage, computing, resource utilization, cost-effectiveness, scalability, etc. But virtualization has some disadvantages, such as performance overhead, high resource consumption, slower boot times, and inefficiencies for microservices. The newly developed container technology overcomes these disadvantages. Container technology is an emerging technology that helps solve many problems in computing tasks. It helps with fast deployment and startup, practical resource utilization, easy rollback, version control, etc [1]. In container scheduling, the main challenge is scheduling a node upon arrival while maintaining resource utilization and improving efficiency. To solve this problem, the most popular approach identifies four methods, each using a variety of parameters [2]. These four methods are mathematical, heuristic, meta-heuristic, and machine learning [3]. However, none of the previous studies tested all the parameters together, and some were rarely used. In this research, the authors first replicated existing methods and then introduced a new algorithm that integrates a heuristic approach with machine learning techniques to achieve improved results and performance. The findings indicate that the proposed method enhances efficiency and effectiveness more than previously established approaches [4].

## 2. Related Work

The above study identifies the most relevant research papers, review articles, and publications from well-known journals between 2017 and 2025. In those papers, it is found that Kubernetes, an open-source native cloud platform, is now offered as a managed service by almost all major cloud providers [5]. However, these services primarily focus on simplifying the control plane through user-friendly interfaces, rather than enhancing overall system performance. Most optimization efforts are driven by cloud-specific interests, such as replacing default Kubernetes components with proprietary solutions, such as cloud-native load balancers or storage systems. Since most of a cloud provider's revenue comes from computing resources, there is little financial incentive for them to optimize these components, as doing so could ultimately reduce their profits [6].

According to exploration, all major cloud providers (Google Cloud, AWS, Azure) offer only two scheduling strategies: Default (random spread) and bin-packing (an optional strategy). According to the Kubernetes documentation, these scheduler strategies were added very recently, as recently as 2016 [7]. Although Kubernetes provides a mechanism for extending its scheduler, very little research has been done in this area [8]. Some cloud providers, such as Google Cloud, have recently added autopilot offerings in which they manage compute resources and charge users extra for that [9]. If it could optimize the Kubernetes scheduling component with a better and more efficient algorithm, it could easily reduce the cost of computing resources for the actual workload [10]. Based on the literature review, this research paper aims primarily to demonstrate that multiple domains are available for exploration, including application, scheduling, clustering, and infrastructure [11]. However, according to a review of various standard conference and journal papers, the scheduling domain has been less explored [12]. The study aims to identify the behavior of the default scheduler and adjust parameters such as resource utilization and cost to improve scheduling efficiency. The proposed algorithm demonstrates better performance than the existing scheduler in both resource and cost parameters [13].

## 3. Paper Organization

The structure of this research paper is organized as follows. Section 1 provides an introduction to the topic. Section 2 presents the related work, and Section 3 outlines the overall organization of the paper. Section 4 discusses the existing work in this domain. Section 5 introduces the proposed flow diagram for implementation, and Section 6 describes the experimental setup and data set specifications in detail. Section 7 reports on the experiments conducted and the corresponding results, followed by a summary. The concluding section compares existing and proposed approaches and outlines potential directions for future work.

## 4. Existing Work

Based on a review of standard journals, conference papers, and proceedings papers from the last 9 years, significant research gaps have been identified in the scheduling domain compared to other areas of Kubernetes research [14]. The focus of the study is on optimizing container scheduling strategies used to schedule containers in a Kubernetes cluster. To optimize scheduling, four primary methods are identified: Mathematical, Heuristic, Meta-Heuristic, and Machine learning [15].

But different research papers have implemented each methodology individually; none have combined these techniques. Based on the parameters of the existing methods, the study found that optimizing resource comprehensiveness in the cloud -specifically CPU and memory - can reduce the overall cost of cloud utilization [16]. So, this research focuses on optimizing the default Kubernetes algorithm for resources such as CPU and memory, resulting in optimization of cost [2, 17].

### *4.1. Block Diagrams of Existing Work*

Figure 1 shows the container scheduling flow in Kubernetes, highlighting the complete sequence of the existing scheduling cycle. When a new pod requests execution, it first enters the scheduling queue, as shown in Figure 3.

The pod then undergoes a sorting phase, detailed in Figure 4. After sorting, it proceeds to the scheduling cycle, as shown in Figure 2. Finally, the pod enters the binding phase, as shown in Figure 8.

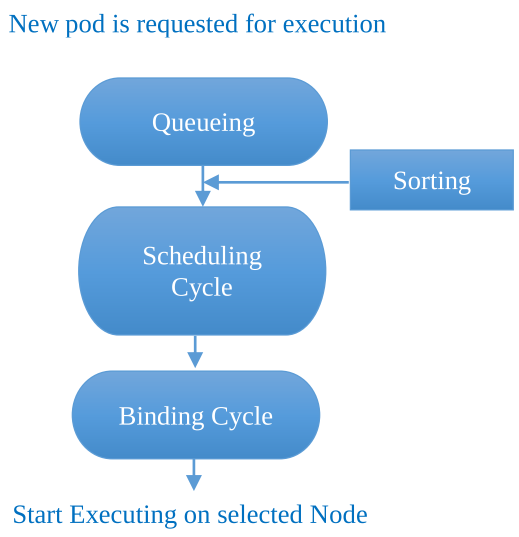
New pod is requested for execution
Queueing
Sorting
Scheduling Cycle
Binding Cycle
Start Executing on selected Node

**Fig. 1 Flow diagram of existing work [18]**

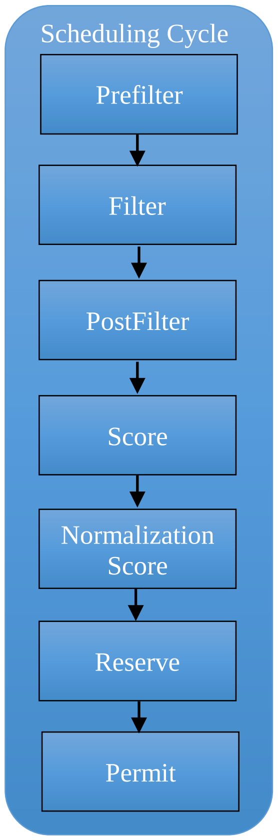

**Fig. 2 Scheduling cycle**

Figure 2 presents a detailed diagram of the scheduling cycle. The process begins with a prefilter stage, where the basic conditions of each node are evaluated. Nodes that pass this check move to the filter stage, where inappropriate nodes are removed based on scheduling requirements [19]. If no nodes qualify, the post-filter stage is triggered, which in turn triggers autoscaling, an internal Kubernetes mechanism that cannot be altered [20]. Once filtering is complete, each eligible node is assigned a score, and the scores are normalized. The highest-scoring node is then reserved, allowing the pod to be scheduled on it [3]. In the figure above, some blocks are highlighted in green, indicating that these parameters can be used to modify the scheduling algorithm to optimize. Figure 3 illustrates the process of entering and scheduling within a queueing system. Initially, a request is generated to get into the queue. Upon activation, the request is organized within the queue for further processing.

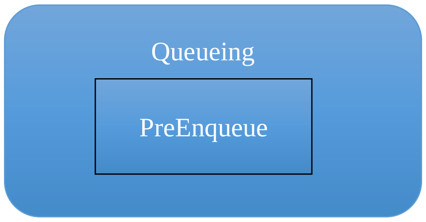

**Fig. 1 Block diagram of the queuing cycle**

Once a pod is queued for scheduling, each queue can be sorted according to various sorting algorithms, such as priority sorting, First Come First Serve, Quick sort, and Bubble sort [21]. As shown in Figure 4, among these, priority and FCFS are the most commonly used queuing algorithms [22]. In the proposed solution, FCFS scheduling will be considered for queuing. Other parameters for the sorting cycle can be regarded as quality of service class and co-scheduling. But these parameters are now not considered in this research.

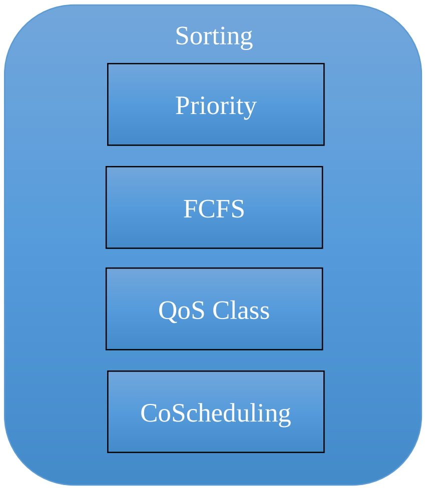

**Fig. 4 Parameters of sorting**

Kubernetes supports creating different filtering algorithms that can be applied during scheduling. The filter can be based on CPU requests, memory requests, node type, specific labels, availability zones, or storage requirements [23]. Figure 5 illustrates the filtering parameters used in the proposed algorithm (indicated by green boxes), while the parameters that could be included but are excluded due to their minimal impact are shown in red boxes [19].

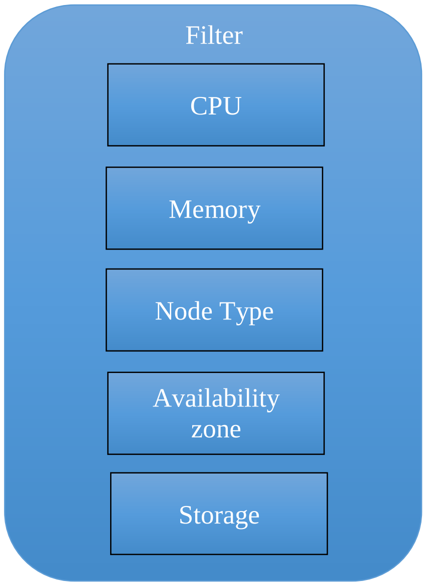

**Fig. 5 Parameters of filtering**

Once the filter stage is completed, the next stage in the scheduling lifecycle will be the scoring stage. In this stage, all nodes that have passed the filter criteria will be scored using a scoring algorithm [24]. Scoring parameters are CPU availability, Memory availability, container image availability, preemptibitity, and storage availability. Among these, all CPU and memory availability is considered.

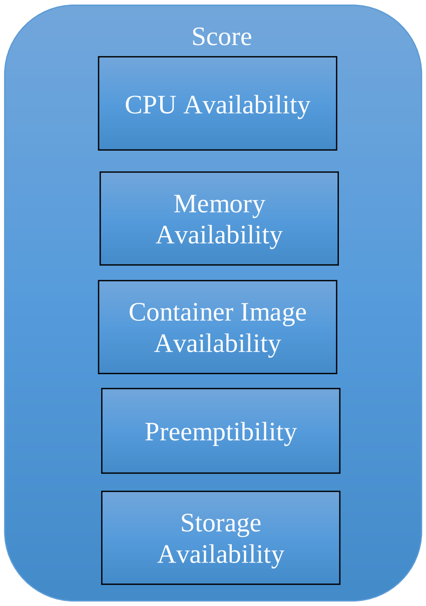

**Fig. 6 Parameters of scoring**

Figure 6 shows the criteria used to determine the score. Any custom algorithm can be added to Kubernetes to define this stage [25]. As the scoring stage can have multiple parameters, the next stage in the lifecycle is the normalization of these scores [26]. It can be done in various ways, as shown in Figure 7. The most common normalization algorithm used is weighted normalization [27]. In addition, custom logic, the simple summation method, and the average method can also be used for weight calculation.

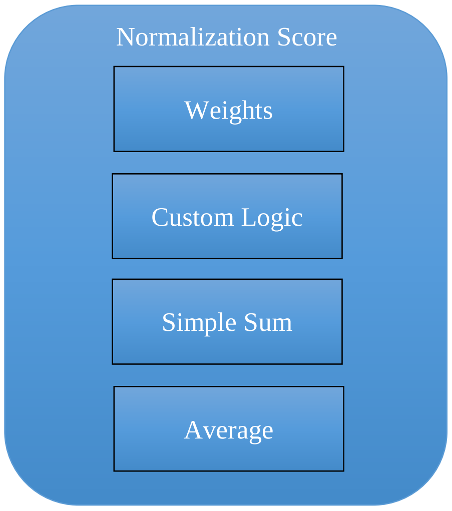

**Fig. 7 Options for normalization of score**

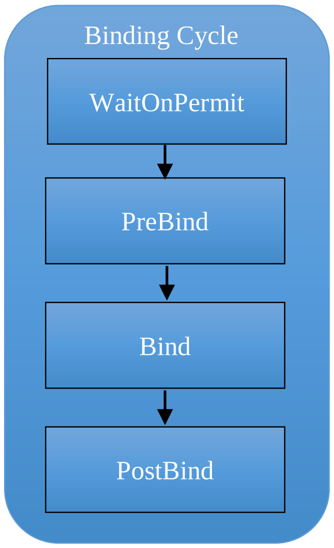

**Fig. 8 Diagram of binding cycle [29]**

After filtering and scoring, the node with the highest score is selected to host the pod, as shown in Figure 8. The binding cycle consists of multiple stages, including WaitOnPermit, PreBind, Bind, and PostBind [12]. Kubernetes defines all these stages, and it will not be possible to extend them using the default Kubernetes lifecycle [28].

The above section outlines Kubernetes operation and execution flow, providing a detailed explanation of how new pod requests are scheduled.

The subsequent section presents the proposed flow diagram and the corresponding experiment design and its results and analysis.

## 5. Proposed Flow Diagram

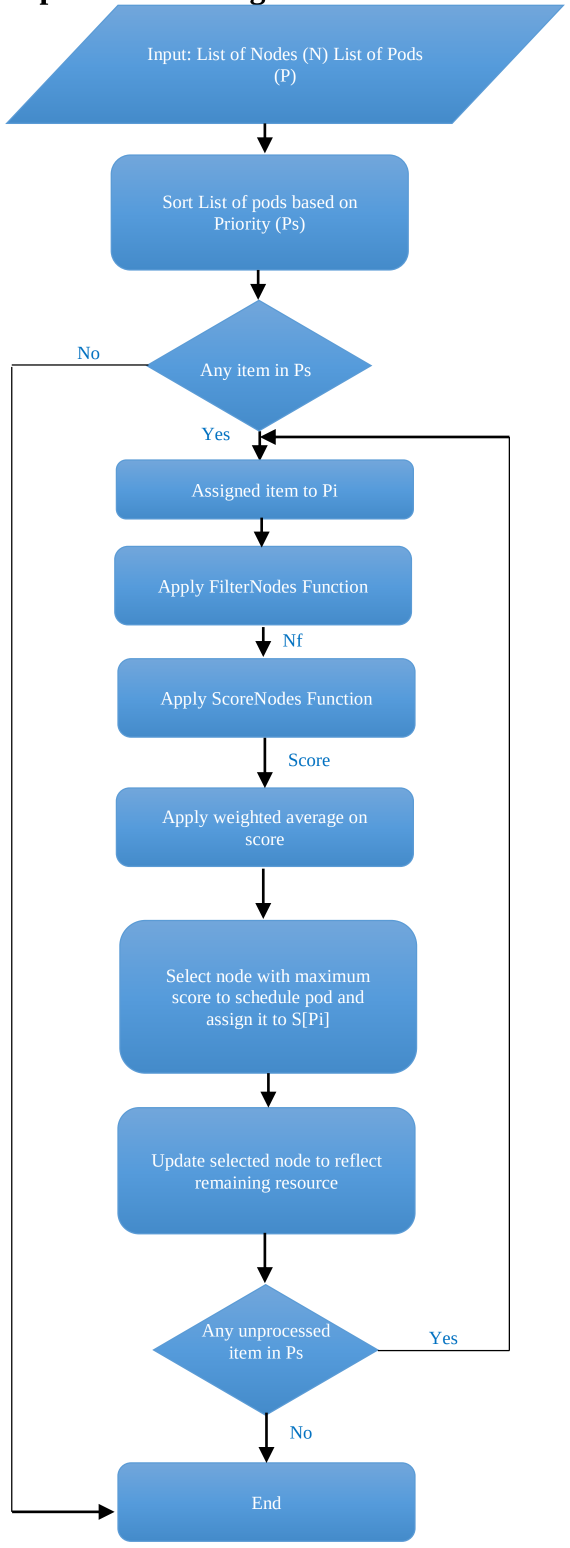
Input: List of Nodes (N) List of Pods (P)
Sort List of pods based on Priority (Ps)
Any item in Ps
No
Yes
Assigned item to Pi
Apply FilterNodes Function
Nf
Apply ScoreNodes Function
Score
Apply weighted average on score
Select node with maximum score to schedule pod and assign it to S[Pi]
Update selected node to reflect remaining resource
Any unprocessed item in Ps
Yes
No
End

**Fig. 9 Diagram of proposed work**

To address the research objective, the authors present the flow chart shown in Figure 9, which outlines the execution of the proposed algorithm. The process begins by identifying the input lists of nodes (N) and pods (P), after which the pods are sorted by priority [30]. Once sorted, the algorithm assigns nodes to each pod and applies a filtering function to eliminate those that do not meet the requirements [31]. After filtering, a scoring mechanism, along with node weights, is used to evaluate each remaining node.

In the final stage, the node with the highest score is selected to schedule the pod, and the system updates the available resources accordingly. If any pods remain unscheduled, the algorithm continues to iterate through the remaining nodes and repeats the cycle [32]. In general, the proposed approach efficiently schedules pods while optimizing resource utilization, with a particular focus on cost and CPU consumption as key resource factors [33].

## 6. Experiment Setup and Dataset Specification

During the implementation phase, the Kubernetes cluster was constructed, consisting of 8 compute nodes, each equipped with 8 CPU cores and 16 GB of memory. For storage requirements, 100GB of persistent volume storage was attached to each node. To simulate real-world dynamic load generation, the load is generated in the same sequence as it is deployed in the actual use case [34]. Each pod definition file contains the maximum number of CPUs it is allowed to use (Limit) and the number of CPUs it requires as dedicated resources (Requests). To maximize scheduling, the dataset has a maximum of 64 (63.75) CPU requests, equal to the total CPU available in the cluster. In this experiment, two of the most commonly used Kubernetes scheduling algorithms [35] were applied. The CPU and Memory requested for each pod definition are given in Table 1. To simulate results, first create nodes in the simulator using the Add New Nodes option. Table 1 describes the data set specification, which represents the workload name, CPU, and memory request and limit. This data set is available on GitHub at the following link: https://github.com/shahparth123/kubernetes-scheduling-dataset.

**Table 1. Data set specification**

| Workload Name | CPU Req | Memory Req | CPU Limit | Memory Limit |
|---|---|---|---|---|
| platform-core-api | 3 | 6 | 4 | 8 |
| platform-data-api | 2 | 4 | 4 | 8 |
| platform-object-api | 4 | 8 | 4 | 8 |
| platform-core-api-kafka | 1 | 1 | 1 | 1 |
| platform-core-api-kafka-zoo | 0.256 | 0.256 | 0.256 | 0.256 |
| platform-data-api-kafka | 1 | 1 | 1 | 1 |
| platform-data-api-kafka-zoo | 0.25 | 0.25 | 0.256 | 0.256 |
| platform-object-api-kafka | 1 | 1 | 1 | 1 |
| platform-object-api-kafka-zoo | 0.25 | 0.25 | 0.256 | 0.256 |
| platform-core-api-mongodb | 2 | 4 | 2 | 4 |
| platform-data-api-mongodb | 2 | 4 | 2 | 4 |
| platform-object-api-mongodb | 6 | 12 | 6 | 12 |
| platform-object-api-postgres | 4 | 8 | 4 | 8 |
| platform-data-api-postgres | 2 | 2.5 | 2 | 2.5 |
| redis-master | 4 | 8 | 6 | 12 |
| redis-replica | 3 | 6 | 3 | 6 |
| keycloak | 2 | 4 | 2 | 4 |
| platform-core-api-r1 | 3 | 6 | 2 | 2.5 |
| platform-data-api-r1 | 2 | 4 | 3 | 6 |
| platform-object-api-r1 | 4 | 8 | 4 | 8 |
| platform-core-api-r2 | 3 | 6 | 4 | 8 |
| platform-data-api-r2 | 2 | 4 | 4 | 8 |
| platform-object-api-r2 | 4 | 8 | 4 | 8 |
| platform-object-api-r3 | 4 | 8 | 4 | 8 |
| platform-object-api-r4 | 4 | 8 | 4 | 8 |

The above section discusses the dataset specifications. This dataset is a real cluster specification published as open source by a well-known cloud-based company.

It includes detailed information on memory and CPU usage, their respective limits, and the corresponding node names.

## 7. Experiment and Results

In this section, the default algorithm and the most and least allocation-based scheduling algorithms are applied to the given data set [36].

The results of these three algorithms were then compared and analysed. In this research, all experiments were conducted on the KubeSim simulator version 0.3.0, which provides the same experimental environment as Kubernetes.

### *7.1. Default Scheduling Algorithm of Kubernetes*

#### *7.1.1. Aim: To schedule a pod on the default scenario of Kubernetes.*

In this experiment, the default Kubernetes scheduling algorithm was used, where all plugins are disabled. As a result, Kubernetes schedules pods based on its basic random scheduling mechanism [37].

The primary limitation of random scheduling is that it often results in inefficient resource use. Since nodes are selected randomly, there is a greater chance that CPU and memory resources are wasted or unevenly utilized [38]. This lack of resource awareness makes random scheduling non-optimal and unsuitable for real-world production systems, where efficiency and balanced resource allocation are crucial.

#### *7.1.2. Observations*

Based on the observation, the default Kubernetes scheduler assigns workloads to nodes in a random, non-optimized manner. It does not check how much CPU or memory a node is already using before assigning new tasks. As a result, some nodes end up carrying more load than they can handle, with their resource usage even exceeding 100\%. The observation table for the default scheduling algorithm is shown in Table 2.

**Table 2. Observation table of the default algorithm**

| Node | CPU | Memory | CPU Used | Memory Used | CPU Util. (%) | Memory Util. (%) |
|---|---|---|---|---|---|---|
| 1 | 8 | 16 | 10.25 | 20.25 | 128.13 | 126.56 |
| 2 | 8 | 16 | 4.25 | 7.25 | 53.13 | 45.31 |
| 3 | 8 | 16 | 9 | 17 | 112.50 | 106.25 |
| 4 | 8 | 16 | 4.25 | 8.25 | 53.13 | 51.56 |
| 5 | 8 | 16 | 20 | 40 | 250.00 | 250.00 |
| 6 | 8 | 16 | 4 | 8 | 50.00 | 50.00 |
| 7 | 8 | 16 | 9 | 15.5 | 112.50 | 96.88 |
| 8 | 8 | 16 | 3 | 6 | 37.50 | 37.50 |
| **Total** | **64** | **128** | **63.75** | **122.25** | - | - |

**Table 3. Result of the default algorithm**

| Workload | CPU Req | Mem Req | Schedule | Node |
|---|---|---|---|---|
| platform-core-api | 3 | 6 | Yes | 8 |
| platform-data-api | 2 | 4 | Yes | 4 |
| platform-object-api | 4 | 8 | Yes | 1 |
| platform-core-api-kafka | 1 | 1 | Yes | 7 |
| platform-core-api-kafka-zoo | 0.25 | 0.25 | Yes | 4 |
| platform-data-api-kafka | 1 | 1 | Yes | 3 |
| platform-data-api-kafka-zoo | 0.25 | 0.25 | Yes | 2 |
| platform-object-api-kafka | 1 | 1 | Yes | 2 |
| platform-object-api-kafka-zoo | 0.25 | 0.25 | Yes | 1 |
| platform-core-api-mongodb | 2 | 4 | Yes | 7 |
| platform-data-api-mongodb | 2 | 4 | Yes | 4 |
| platform-object-api-mongodb | 6 | 12 | Yes | 5 |
| platform-object-api-postgres | 4 | 8 | Yes | 3 |
| platform-data-api-postgres | 2 | 2.5 | Yes | 7 |
| redis-master | 4 | 8 | Yes | 5 |
| redis-replica | 3 | 6 | Yes | 5 |
| keycloak | 2 | 4 | Yes | 5 |
| platform-core-api-r1 | 3 | 6 | Yes | 2 |
| platform-data-api-r1 | 2 | 4 | Yes | 1 |
| platform-object-api-r1 | 4 | 8 | Yes | 1 |
| platform-core-api-r2 | 3 | 6 | Yes | 5 |
| platform-data-api-r2 | 2 | 4 | Yes | 5 |
| platform-object-api-r2 | 4 | 8 | Yes | 6 |
| platform-object-api-r3 | 4 | 8 | Yes | 3 |
| platform-object-api-r4 | 4 | 8 | Yes | 7 |

In the Result table, clearly mention that the default algorithm unscheduled the nodes. It exceeds the limit for CPU and memory, but it cannot schedule the nodes.

### *7.2. Least Allocated Node Scheduling Algorithm*

*7.2.1. Aim: To schedule a pod on the least allocated node of Kubernetes.*

In this experiment, a least-allocated node scheduling algorithm will be used to schedule pods. In which a pod is allocated to a node that has the least resource utilization, resulting in the minimum use of memory and CPU resources [4]. This algorithm also balances resource utilization between nodes [39].

*7.2.2. Observations*

According to the observation, every node is scheduled based on the least memory and resource usage. First, it applies filters to determine which nodes can have resources beyond what the pod requests for scheduling. After that, it will use the scoring to find the best-fitted node among all.

The node’s scoring will be based on the resources available on the node. The node with the maximum score will be chosen for scheduling. Then the binding cycle will actually schedule a pod on the selected node for execution [40].

**Table 4. Observation table of the least allocated node scheduling algorithm**

| Node | CPU | Memory | CPU Used | Memory Used | CPU Util. (%) | Memory Util. (%) |
|---|---|---|---|---|---|---|
| 1 | 8 | 16 | 5.00 | 10.00 | 62.50 | 62.50 |
| 2 | 8 | 16 | 6.00 | 12.00 | 75.00 | 75.00 |
| 3 | 8 | 16 | 5.25 | 10.25 | 65.63 | 64.06 |
| 4 | 8 | 16 | 7.00 | 13.00 | 87.50 | 81.25 |
| 5 | 8 | 16 | 8.00 | 13.50 | 100.00 | 84.38 |
| 6 | 8 | 16 | 8.00 | 16.00 | 100.00 | 100.00 |
| 7 | 8 | 16 | 5.50 | 10.50 | 68.75 | 65.63 |
| 8 | 8 | 16 | 7.00 | 13.00 | 87.50 | 81.25 |
| Total | 64 | 128 | 51.75 | 98.25 | - | - |
| Un-scheduled | - | - | 12.00 | 24.00 | - | - |

**Table 5. Result of the least allocated node scheduling algorithm**

| Workload Name | CPU Request | Memory Request | Scheduled | Allocated on Node |
|---|---|---|---|---|
| platform-core-api | 3 | 6 | Yes | 1 |
| platform-data-api | 2 | 4 | Yes | 2 |
| platform-object-api | 4 | 8 | Yes | 6 |
| platform-core-api-kafka | 1 | 1 | Yes | 4 |
| core-api-kafka-zoo | 0.26 | 0.26 | Yes | 7 |
| data-api-kafka | 1 | 1 | Yes | 8 |
| data-api-kafka-zoo | 0.25 | 0.25 | Yes | 3 |
| object-api-kafka | 1 | 1 | Yes | 5 |
| object-api-kafka-zoo | 0.25 | 0.25 | Yes | 7 |
| core-api-mongodb | 2 | 4 | Yes | 3 |
| data-api-mongodb | 2 | 4 | Yes | 7 |
| object-api-mongodb | 6 | 12 | Yes | 8 |
| object-api-postgres | 4 | 8 | Yes | 4 |
| data-api-postgres | 2 | 2.5 | Yes | 5 |
| redis-master | 4 | 8 | Yes | 2 |
| redis-replica | 3 | 6 | Yes | 3 |
| keycloak | 2 | 4 | Yes | 5 |
| core-api-r1 | 3 | 6 | Yes | 7 |
| data-api-r1 | 2 | 4 | Yes | 1 |
| object-api-r1 | 4 | 8 | Yes | 6 |
| core-api-r2 | 3 | 6 | Yes | 5 |
| data-api-r2 | 2 | 4 | Yes | 4 |
| object-api-r2 | 4 | 8 | No | - |
| object-api-r3 | 4 | 8 | No | - |
| object-api-r4 | 4 | 8 | No | - |

As a result of the least allocation, unscheduled CPU was 12 and memory was 24, which was clearly shown in the result table and was better than the default algorithm.

### *7.3. Most Allocated Node Scheduling Algorithm*

#### *7.3.1. Aim: To Schedule a Pod on the Most Allocated Node of Kubernetes*

In this experiment, the most-allocated-node scheduling is used to schedule pods. In which pod will the node be that has the maximum utilization of memory and CPU [41]. It will first filter nodes that meet the requested resources.

After that, it will assign each node a score based on its utilization. The Highest utilized node will get a higher score. Then select the node with the highest score assigned to the pod for execution [42]. As this algorithm tries to focus on maximizing utilization on any available node first before selecting the next node, this algorithm is also known as the bin-packing algorithm.

#### *7.3.2. Observations*

As a result of most allocations, unscheduled CPU was 4 and memory was 8, which was clearly shown in the result table and was better than the default algorithm and the least allocation algorithm. According to the observation, every node is scheduled based on the highest memory and resource usage. First, it applies filters to determine which nodes will serve as the go-to for scheduling. After that, it will use the scoring to find the worst-fitting node. Then, it goes for the binding cycle [43].

**Table 6. Observation table of the most allocated node scheduling algorithm**

| Node | CPU | Memory | CPU Used | Memory Used | CPU Util. (%) | Memory Util. (%) |
|---|---|---|---|---|---|---|
| 1 | 8 | 16 | 7.75 | 12.75 | 96.88 | 79.69 |
| 2 | 8 | 16 | 8.00 | 16.00 | 100.00 | 100.00 |
| 3 | 8 | 16 | 7.00 | 14.00 | 87.50 | 87.50 |
| 4 | 8 | 16 | 8.00 | 16.00 | 100.00 | 100.00 |
| 5 | 8 | 16 | 8.00 | 14.50 | 100.00 | 90.63 |
| 6 | 8 | 16 | 7.00 | 13.00 | 87.50 | 81.25 |
| 7 | 8 | 16 | 6.00 | 12.00 | 75.00 | 75.00 |
| 8 | 8 | 16 | 8.00 | 16.00 | 100.00 | 100.00 |
| Total | 64 | 128 | 59.75 | 114.25 | - | - |
| Unscheduled | - | - | 4.00 | 8.00 | - | - |

**Table 7. Result of the most allocated scheduling algorithm**

| Workload Name | CPU Request | Memory Request | Scheduled | Allocated on Node |
|---|---|---|---|---|
| platform-core-api | 3 | 6 | Yes | 1 |
| platform-data-api | 2 | 4 | Yes | 1 |
| platform-object-api | 4 | 8 | Yes | 6 |
| platform-core-api-kafka | 1 | 1 | Yes | 1 |
| platform-core-api-kafka-zoo | 0.256 | 0.256 | Yes | 1 |
| platform-data-api-kafka | 1 | 1 | Yes | 1 |
| platform-data-api-kafka-zoo | 0.25 | 0.25 | Yes | 1 |
| platform-object-api-kafka | 1 | 1 | Yes | 6 |
| platform-object-api-kafka-zoo | 0.25 | 0.25 | Yes | 1 |
| platform-core-api-mongodb | 2 | 4 | Yes | 6 |
| platform-data-api-mongodb | 2 | 4 | Yes | 8 |
| platform-object-api-mongodb | 6 | 12 | Yes | 8 |
| platform-object-api-postgres | 4 | 8 | Yes | 5 |
| platform-data-api-postgres | 2 | 2.5 | Yes | 5 |
| redis-master | 4 | 8 | Yes | 3 |
| redis-replica | 3 | 6 | Yes | 3 |
| keycloak | 2 | 4 | Yes | 5 |
| platform-core-api-r1 | 3 | 6 | Yes | 2 |
| platform-data-api-r1 | 2 | 4 | Yes | 2 |
| platform-object-api-r1 | 4 | 8 | Yes | 7 |
| platform-core-api-r2 | 3 | 6 | Yes | 2 |
| platform-data-api-r2 | 2 | 4 | Yes | 7 |
| platform-object-api-r2 | 4 | 8 | No | 4 |
| platform-object-api-r3 | 4 | 8 | No | 4 |
| platform-object-api-r4 | 4 | 8 | No | - |

## 8. Summary of Results

In all three algorithms, the default algorithm, which turns off all plugins and schedules the node at random, resulted in allocating more than 100% of the available resources, which is not suitable for real-world scenarios [44]. In the least-allocated and most-allocated algorithm scenarios, it finds the node with the most resources for the given request before actually scheduling, preventing resource over-allocation. In the least-fitted scenario, there will be 12 CPUs and 24 GB of Memory additionally required, while in the most-fitted scenario, only 4 CPUs and 8 GB of Memory will be additionally needed [45].

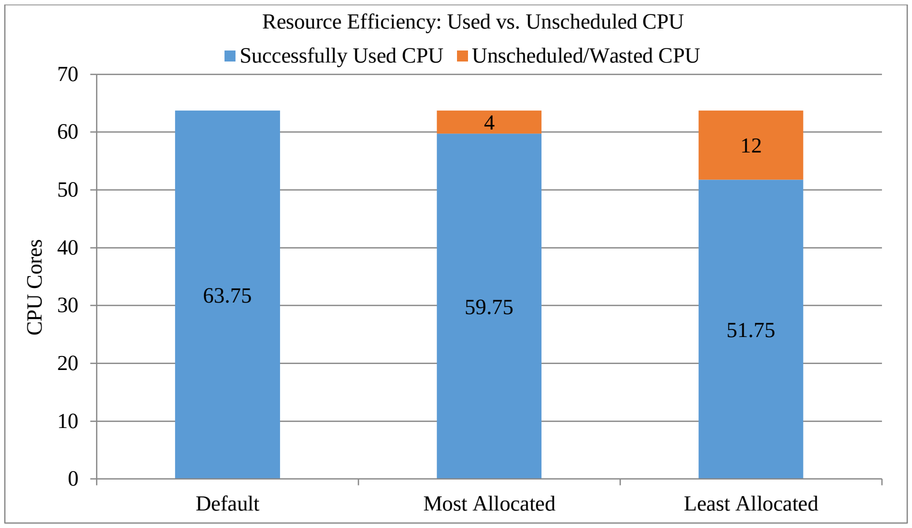

**Fig. 10 Resource efficiency by algorithms**

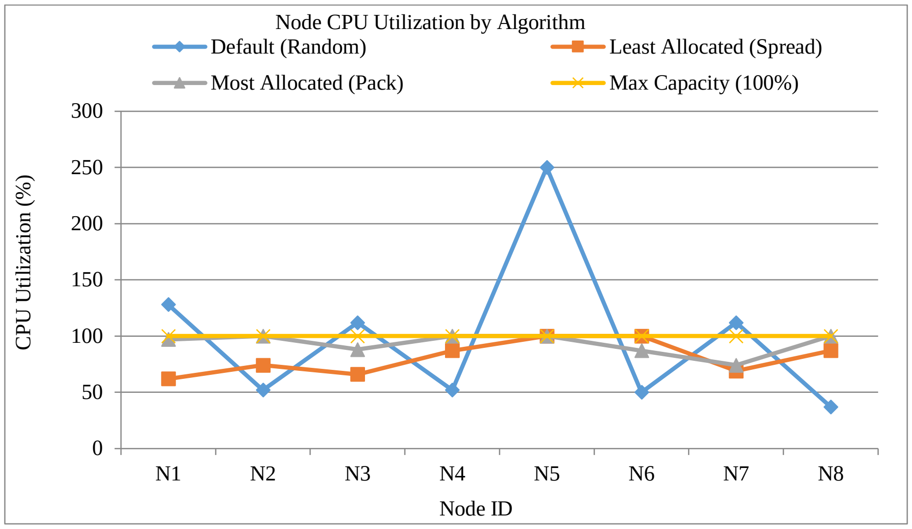

**Fig. 11 Resource utilization by algorithms**

So, according to the analysis above, the results of the most allocated node provide the most significant benefits by improving resource efficiency while preventing over-allocation. In the next section, describe the conclusion and future scope based on the analysis and observations of the experimental results.

## 9. Conclusion and Future Work

This paper presents an efficient scheduling algorithm designed for node-level resource management. This proposed algorithm has been compared with different algorithms using a publicly available data set. This algorithm has been evaluated on the basis of various parameters, including RAM and CPU requests and usage. According to the results, the algorithm that allocates the most nodes outperforms both the default algorithm and the algorithm that assigns the fewest nodes. In this study, only CPU and RAM are considered main parameters. However, other resources, such as the network and I/O, can also be added in future research for additional efficiency. In future studies, the authors plan to examine these factors and formulate an optimal algorithm by introducing multiple machine learning techniques into the scheduling algorithm. According to the result, the proposed scheduling algorithm gives a better result than the default algorithm. This algorithm has been considered by various parameters, such as the RAM request, CPU request, RAM used, and CPU used. In this study, only CPU and RAM are considered. However, there are still other resources that must be considered, such as the network, IO, etc. In our future work, we will consider these factors. We will also develop a suitable algorithm by merging a single technique as a machine learning technique.

## Acknowledgments

The authors express their sincere gratitude to Maverick Quantum Inc. for granting access to the cluster usage dataset used in this study.

## References

[1] Imtiaz Ahmad et al., “Container Scheduling Techniques: A Survey and Assessment,” *Journal of King Saud University-Computer and Information Sciences*, vol. 34, no. 7, pp. 3934-3947, 2022. [CrossRef] [Google Scholar] [Publisher Link]

[2] Tarek Menouer, “KCSS: Kubernetes Container Scheduling Strategy,” *The Journal of Supercomputing*, vol. 77, no. 5, pp. 4267-4293, 2020. [CrossRef] [Google Scholar] [Publisher Link]

[3] David Balla, Csaba Simon, and Markosz Maliosz, “Adaptive Scaling of Kubernetes Pods,” *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, Budapest, Hungary, pp. 1-5, 2020. [CrossRef] [Google Scholar] [Publisher Link]

[4] Carmen Carrión, “Kubernetes Scheduling: Taxonomy, Ongoing Issues and Challenges,” *ACM Computing Surveys*, vol. 55, no. 7, pp. 1-37, 2022. [CrossRef] [Google Scholar] [Publisher Link]

[5] A. Arunarani, Dhanabalachandran Manjula, and Vijayan Sugumaran, “Task Scheduling Techniques in Cloud Computing: A Literature Survey,” *Future Generation Computer Systems*, vol. 91, pp. 407-415, 2019. [CrossRef] [Google Scholar] [Publisher Link]

[6] Ye Wu, and Haopeng Chen, “ABP Scheduler: Speeding Up Service Spread in Docker Swarm,” *2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC)*, Guangzhou, China, pp. 691-698, 2017. [CrossRef] [Google Scholar] [Publisher Link]

[7] Ying Mao et al., “Draps: Dynamic and Resource-Aware Placement Scheme for Docker Containers in a Heterogeneous Cluster,” *2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC)*, San Diego, CA, USA, pp. 1-8, 2017. [CrossRef] [Google Scholar] [Publisher Link]

[8] Piotr Dziurzanski, and Leandro Soares Indrusiak, “Value-based Allocation of Docker Containers,” *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, Cambridge, UK, pp. 358-362, 2018. [CrossRef] [Google Scholar] [Publisher Link]

[9] Weiwen Zhang et al., “Cost-Efficient and Latency-Aware Workflow Scheduling Policy for Container-based Systems,” *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*, Singapore, pp. 763-770, 2018. [CrossRef] [Google Scholar] [Publisher Link]

[10] Shengbo Song et al., “Gaia Scheduler: A Kubernetes-based Scheduler Framework,” *2018 IEEE Intl Conf on Parallel and Distributed Processing with Applications, Ubiquitous Computing and Communications, Big Data and Cloud Computing, Social Computing and Networking, Sustainable Computing and Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom)*, Melbourne, VIC, Australia, pp. 252-259, 2018. [CrossRef] [Google Scholar] [Publisher Link]

[11] Han Dong et al., “Towards Performance and Energy Aware Kubernetes Scheduler,” *ACM SIGEnergy Energy Informatics Review*, vol. 5, no. 2, pp. 69-75, 2025. [CrossRef] [Google Scholar] [Publisher Link]

[12] Jingze Lv, Mingchang Wei, and Yang Yu, “A Container Scheduling Strategy based on Machine Learning in Microservice Architecture,” *2019 IEEE International Conference on Services Computing (SCC)*, Milan, Italy, pp. 65-71, 2019. [CrossRef] [Google Scholar] [Publisher Link]

[13] Yang Hu, Cees De Laat, and Zhiming Zhao, “Multi-Objective Container Deployment on Heterogeneous Clusters,” *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, Larnaca, Cyprus, pp. 592-599, 2019. [CrossRef] [Google Scholar] [Publisher Link]

[14] Tarek Menouer, and Patrice Darmon, “Containers Scheduling Consolidation Approach for Cloud Computing,” *Pervasive Systems, Algorithms and Networks: 16th International Symposium, I-SPAN 2019*, Naples, Italy, pp. 178-192, 2019. [CrossRef] [Google Scholar] [Publisher Link]

[15] Minxian Xu, and Rajkumar Buyya, “Brownoutcon: A Software System based on Brownout and Containers for Energy-Efficient Cloud Computing,” *Journal of Systems and Software*, vol. 155, pp. 91-103, 2019. [CrossRef] [Google Scholar] [Publisher Link]

[16] Alfred Daimari, Matthijs Jansen, and Daniele Bonetta, “*Energy Consumption of Heuristic Kubernetes Schedulers*,” Technical Report, pp. 1-7, 2025. [Google Scholar]

[17] Yang Hu et al., “Concurrent Container Scheduling on Heterogeneous Clusters with Multi-Resource Constraints,” *Future Generation Computer Systems*, vol. 102, pp. 562-573, 2020. [CrossRef] [Google Scholar] [Publisher Link]

[18] Jialin Yang et al., “A Survey on Task Scheduling in Carbon-Aware Container Orchestration Systems,” *arXiv Preprint*, pp. 1-35, 2025. [CrossRef] [Google Scholar] [Publisher Link]

[19] Leonardo R. Rodrigues et al., “Network-Aware Container Scheduling in Multi-Tenant Data Center,” *2019 IEEE Global Communications Conference (GLOBECOM)*, Waikoloa, HI, USA, pp. 1-6, 2019. [CrossRef] [Google Scholar] [Publisher Link]

[20] László Toka et al., “Machine Learning-based Scaling Management for Kubernetes Edge Clusters,” *IEEE Transactions on Network and Service Management*, vol. 18, no. 1, pp. 958-972, 2021. [CrossRef] [Google Scholar] [Publisher Link]

[21] Laszlo Toka et al., “Adaptive AI-based Auto-Scaling for Kubernetes,” *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, Melbourne, VIC, Australia, pp. 599-608, 2020. [CrossRef] [Google Scholar] [Publisher Link]

[22] Mingming Wang, Dongmei Zhang, and Bin Wu, “A Cluster Autoscaler based on Multiple Node Types in Kubernetes,” *2020 IEEE 4th Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*, Chongqing, China, pp. 575-579, 2020. [CrossRef] [Google Scholar] [Publisher Link]

[23] Rui Kang et al., “Design of Scheduler Plugins for Reliable Function Allocation in Kubernetes,” *2021 17th International Conference on the Design of Reliable Communication Networks (DRCN)*, Milano, Italy, pp. 1-3, 2021. [CrossRef] [Google Scholar] [Publisher Link]

[24] Dinh-Dai Vu, Minh-Ngoc Tran, and Younghan Kim, “Predictive Hybrid Autoscaling for Containerized Applications,” *IEEE Access*, vol. 10, pp. 109768-109778, 2022. [CrossRef] [Google Scholar] [Publisher Link]

[25] Gianluca Turin et al., “Predicting Resource Consumption of Kubernetes Container Systems using Resource Models,” *Journal of Systems and Software*, vol. 203, pp. 1-19, 2023. [CrossRef] [Google Scholar] [Publisher Link]

[26] Zhaolong Jian et al., “DRS: A Deep Reinforcement Learning Enhanced Kubernetes Scheduler for Microservice-based System,” *Software: Practice and Experience*, vol. 54, no. 10, pp. 2102-2126, 2024. [CrossRef] [Google Scholar] [Publisher Link]

[27] Saurav Nanda, and Thomas J. Hacker, “RACC: Resource-Aware Container Consolidation using a Deep Learning Approach,” *Proceedings of the First Workshop on Machine Learning for Computing Systems*, pp. 1-5, 2018. [CrossRef] [Google Scholar] [Publisher Link]

[28] Hemant Kumar Mehta et al., “WattsApp: Power-Aware Container Scheduling,” *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*, Leicester, UK, pp. 79-90, 2020. [CrossRef] [Google Scholar] [Publisher Link]

[29] Donglei Xiao et al., “Load-Balanced Scheduling Optimization Strategy for High-Communication Tasks in Kubernetes with RDMA,” *Computer Communications*, vol. 241, 2025. [CrossRef] [Google Scholar] [Publisher Link]

[30] Shubha Brata Nath et al., “Green Containerized Service Consolidation in Cloud,” *ICC 2020 - 2020 IEEE International Conference on Communications (ICC)*, Dublin, Ireland, pp. 1-6, 2020. [CrossRef] [Google Scholar] [Publisher Link]

[31] Yanghua Peng et al., “DL2: A Deep Learning-Driven Scheduler for Deep Learning Clusters,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 8, pp. 1947-1960, 2021. [CrossRef] [Google Scholar] [Publisher Link]

[32] Ying Mao et al., “Speculative Container Scheduling for Deep Learning Applications in a Kubernetes Cluster,” *IEEE Systems Journal*, vol. 16, no. 3, pp. 3770-3781, 2022. [CrossRef] [Google Scholar] [Publisher Link]

[33] Jiaming Huang, Chuming Xiao, and Weigang Wu, “RLSK: A Job Scheduler for Federated Kubernetes Clusters based on Reinforcement Learning,” *2020 IEEE International Conference on Cloud Engineering (IC2E)*, Sydney, NSW, Australia, pp. 116-123, 2020. [CrossRef] [Google Scholar] [Publisher Link]

[34] Haoyu Wang, Zetian Liu, and Haiying Shen, “Job Scheduling for Large-Scale Machine Learning Clusters,” *Proceedings of the 16th International Conference on Emerging Networking Experiments and Technologies*, pp. 108-120, 2020. [CrossRef] [Google Scholar] [Publisher Link]

[35] Zijie Liu et al., “KubFBS: A Fine-Grained and Balance-Aware Scheduling System for Deep Learning Tasks based on Kubernetes,” *Concurrency and Computation: Practice and Experience*, vol. 34, no. 11, 2022. [CrossRef] [Google Scholar] [Publisher Link]

[36] Mohamed Rahali, Cao-Thanh Phan, and Gerardo Rubino, “KRS: Kubernetes Resource Scheduler for Resilient NFV Networks,” *2021 IEEE Global Communications Conference (GLOBECOM)*, Madrid, Spain, pp. 1-6, 2021. [CrossRef] [Google Scholar] [Publisher Link]

[37] M. Chandra, “*Effective Memory Utilization using Custom Scheduler in Kubernetes*,” Master’s Thesis, Dublin, National College of Ireland, pp. 1-25, 2023. [Google Scholar] [Publisher Link]

[38] Zeineb Rejiba, and Javad Chamanara, “Custom Scheduling in Kubernetes: A Survey on Common Problems and Solution Approaches,” *ACM Computing Surveys*, vol. 55, no. 7, pp. 1-37, 2022. [CrossRef] [Google Scholar] [Publisher Link]

[39] Khaldoun Senjab et al., “A Survey of Kubernetes Scheduling Algorithms,” *Journal of Cloud Computing*, vol. 12, no. 1, pp. 1-26, 2023. [CrossRef] [Google Scholar] [Publisher Link]

[40] Henrik Daniel Christensen, Saverio Giallorenzo, and Jacopo Mauro, “Priority Matters: Optimising Kubernetes Clusters Usage with Constraint-based Pod Packing,” *arXiv Preprint*, pp. 1-8, 2025. [CrossRef] [Google Scholar] [Publisher Link]

[41] Saeid Ghafouri, Sina Abdipoor, and Joseph Doyle, “Smart-Kube: Energy-Aware and Fair Kubernetes Job Scheduler using Deep Reinforcement Learning,” *2023 IEEE 8th International Conference on Smart Cloud (SmartCloud)*, Tokyo, Japan, pp. 154-163, 2023. [CrossRef] [Google Scholar] [Publisher Link]

[42] Zheng Xu et al., "Enhancing Kubernetes Automated Scheduling with Deep Learning and Reinforcement Techniques for Large-Scale Cloud Computing Optimization," *Proceedings of the Ninth International Symposium on Advances in Electrical, Electronics, and Computer Engineering (ISAEECE 2024)*, Changchun, China, 2024. [CrossRef] [Google Scholar] [Publisher Link]

[43] Wei Rao, and Hongjian Li, "Energy-Aware Scheduling Algorithm for Microservices in Kubernetes Clouds," *Journal of Grid Computing*, vol. 23, no. 1, 2025. [CrossRef] [Google Scholar] [Publisher Link]

[44] Vedran Dakić et al., "Optimizing Kubernetes Scheduling for Web Applications using Machine Learning," *Electronics*, vol. 14, no. 5, pp. 1-17, 2025. [CrossRef] [Google Scholar] [Publisher Link]

[45] Mazen Farid et al., "Optimizing Kubernetes with Multi-Objective Scheduling Algorithms: A 5G Perspective," *Computers*, vol. 14, no. 9, pp. 1-39, 2025. [CrossRef] [Google Scholar] [Publisher Link]