Received: March 14, 2023. Revised: April 13, 2023.

International Journal of
Intelligent Engineering & Systems

http://www.inass.org/

# Scheduling of Jobs Allocation for Apache Spark Using Kubernetes for Efficient Execution of Big Data Application

**Jayanthi M[1]*** **K. Ram Mohan Rao[2]**

[1]*Department of Computer Science and Informatics, Mahatma Gandhi University, Nalgonda, India*
[2]*Department of Information Technology, Vasavi College of Engineering, Hyderabad, India*
* Corresponding author's Email: jayanthimgu343@gmail.com

**Abstract:** The use of cloud services is in high demand due to their high storage and computing capacity. Apache spark provides an open deployment framework for data storage and computation using cluster computing. The specified spark core scheduler uses FIFO to manage job execution in batches. However, it may not be suitable for large-scale clusters due to the unevenness in managing resource allocation between different types of applications. Because of this, most of the work executors are still underutilized and resources are wasted in all the pods, leading to cost inefficiencies. Using apache spark on kubernetes to run cloud applications will ensure rapid resource management for workload execution. Incoming workloads vary widely across applications, so it is critical to manage workload allocation to ensure QoS and cost efficiency. This paper proposes a job scheduling mechanism (JSM) for apache spark on kubernetes to dynamically schedule job allocation for the efficient execution of various big data applications. The JSM process predicts the cluster load and suggests relocation of workloads to efficiently distribute the workload to the lower loaded pods in a standard cluster to optimize cost performance. It identifies the upcoming workload of a job and determines the best-fit pod and aims to reduce the usage of CPU and memory which result in enhancing cost efficiency. The JSM's effective management of job allocation and migration among the underload pods preserves the resource and enhances cost efficiency. Experimental settings are configured to evaluate cluster resources with benchmark statistics output for application job execution. The outcome results of cost, job performance, and scheduling overhead show improved cost efficiency for job execution. The comparison with the existing scheduler with varying request load shows an improvisation of 2% in cost efficiency and 3% lower scheduling overhead.

**Keywords:** Scheduling, Job allocation, Big data, Apache spark, Kubernetes, Clustering.

## 1. Introduction

The increased usage of cloud computing over the Internet demands the utilization of cloud resources in terms of the virtual technique model provided by virtual machines (VM) instances. The flexibility to adapt any kind of application makes it an open choice for most organizations to build their private cloud platform as per their needs [1]. Since these private clouds have limited physical resources, most researchers are going on to maximize resource utilization and provide guaranteed services to users.

Big data computing has become important due to the widespread analytical needs in all foremost business and technical areas of medical science, financial management, data processing, streaming, and many other research and developments. The most common frameworks which are being used for data processing in clouds are hadoop and spark [2]. Most organizations typically operate clusters of private computing frameworks on which one or more large data processors are processed. It gains popularity due to its enormous capacity of handling big data processing and providing the best developing support platform and storage. However, scheduling these big data workloads into a cloud-hosted cluster can be challenging because the workload can be CPU-intensive and network-intensive [3].

Network and system heterogeneity, as well as significant network delays between computing nodes, are important factors that adversely affect latency-sensitive applications. In addition, as applications that are exposed to diverse workloads and exhibit quality of service (QoS) requirements, their deployment must also be optimally tuned to runtime. The current trend is to use software for application development and runtime management [4-5]. Containers with operating system-level virtualization have the advantage of reduced overhead compared to VMs. However, the performance of different activities or services can vary significantly over time, making it difficult to adopt appropriate resource allocations and decisions.[6-7].

Kubernetes (k8s) is a powerful tool for managing the deployment and lifecycle of containers, but it does not manage the cloud infrastructure running on top of the containers [8-10]. If there are healthy nodes to work with, it scales containers and containers, but it's up to the user to provision and manage the infrastructure. It allows users to define resource instructions for containers based on their specific CPU and memory needs. Developers will often try to design based on assumptions that can be inaccurate, trial and error, and simulations with test traffic that can be ineffective because test metrics often differ from production usage.

In addition, operations may vary based on equipment requirements to maintain consistent performance, and, the diversity of VM instances located in the cloud makes it difficult to develop cost-effective scheduling schemes [3, 11-12].

Scheduling defines a strategy for managing existing resources and assigning jobs that can be managed effectively. In this paper, we propose an efficient job scheduling (JSM) algorithm that minimizes the cost of using a cloud-hosted apache spark cluster to improve job execution efficiency. As k8s grows in popularity, many companies offering software-as-a-service (SaaS) and platform-as-a-service (PaaS) products are using K8 clusters for their workflow. We use k8s and apache spark clusters to promote cost reduction using JMS through our scheduling and migration processes.

Scheduling job workflows is a good way to optimize data center resource utilization [13-15]. The principle of VM migration is to dynamically distribute VMs periodically according to current resource requirements to take advantage of the dynamic nature of workloads in the cloud and reduce the number of physical servers. However, several essential concerns need to be addressed before moving: (1) the timing of the shifting, (2) the nature of the job to migration, (3) which pods (i.e., where the pods are located) must be selected to support the operations chosen for the migration. This works aims to contribute the following:

- The pod over-load algorithm identifies the upcoming workload of a pod when determining the choice of an overloaded cluster using a pods selection method (PSM). The algorithm determines the likelihood of a Pod being too big to avoid probable desecrations of present SLA instructions and over-loading of the pod.
- Loss of CPU performance resulted in inefficient jobs migration, resulting in additional resource loss and pod performance degradation. In this, a pod-picking algorithm that relies on CPU and memory losses in job relocation is suggested to advance pod enhancement and decrease inappropriate job relocation.
- Jobs are migrated using a pod overload determination algorithm that analyzes pod states and recognizes unloaded pods using job allocation method (JAM). All jobs on the pod are relocated and the pod is terminated. This reduces resource utilization and enhances cost efficiency.

The following sections of the paper are organized into five sections. Section-2 discusses the background and literature study of the relevant work. Section-3 presents the proposed JSM functions and procedures. Section-4 describes the experimental evaluation of the outcomes and section-5 presents the conclusion of the paper.

## 2. Related works

Cloud computing is changing IT engineering by facilitating the flexible allocation or sharing of computing resources comprising of CPU, memory, storage, and networks, to create, customize, and optimize large-scale network and cloud database systems[4, 14]. Since cloud computing serves many users around the world, large data centers host applications from different clients around the world [16]. These platforms use virtualization technology to duplicate the underlying physical resources due to which workloads for different applications vary greatly, workload and resource allocation must be controlled to ensure the quality of service [17].

In this work, we utilize a framework of apache spark with kubernetes to process big data. Kubernetes acts as a cluster manager for the nodes executing sparks jobs along with the proposed job scheduling method. A brief concept and applicability of Apache sparks and Kubernetes in real time are discussed below.

## 2.1 Apache sparks

Apache spark is a data processing framework [18] that can execute fast processing jobs on very big data and can be distributed across multiple computers either alone or in combination with other distributed computing tools [19]. These two key attributes are relevant to the fields of big data and machine learning, which require the integration of enormous computing resources to process huge data. It also takes some of the logistical burdens of performing these jobs off developers' shoulders through an easy-to-use API that eliminates most of the burden of distributed computing and big data processing [20-21].

Essentially, an apache spark program consists of two key modules: a driver that translates user code into multiple jobs that can be shared on worker nodes, and an executor that executes jobs that run on those nodes [22]. It will need some form of cluster manager to coordinate between the two. Spark can run autonomously in a cluster utilizing only the framework of spark and VM on every instance in the cluster. However, it is possible to use a more reliable sourcing or cluster-controlling system to control the distribution of VM based on job burdens. Enterprise applications typically run these on hadoop YARN, apache mesos, kubernetes, and dockers.

The framework of Spark is designed explicitly to collaborate and run big data analysis for real-time applications [16, 18, 21]. It became a solid establishment for data learning and took the field of big data analytics at high speed. The software helps research engineers build and share scalable, high-performance data analysis pipelines. Ultimately, it hides the particulars of distributed processing behind an appropriate API. Behind the scenes, spark partitions the raw data and distributes partitions among computer groups to optimize user calculations and use parallelism to reduce data traffic. Due to its appropriate API and raw implementation speed, Spark has led the world of big data immensely since its beginning. Many different applications run by multiple users compete for the same resources. Users are aware of the problems caused by the conflict: companies spend a lot of money on cloud appliances or VMs and don't get the results they need when they need them.

Let's take a look at the remaining residuals with distributed processing problems. When developers submit a distributed processing application, they must specify the amount of CPU required for the application and other required parameters. But the hardware requirements (CPU, network, memory, etc.) may change after the job is finished. It was noted that most workplaces require reservations, while a few do not. Thus, most distributed systems that perform these jobs are said to consume twice as many resources as they need.

Almost every business and every researcher needs big data analytics and competes with others on their clusters for resources. The emergence of real-time analytics – serving relevant content to website visitors, retail offers based on recent purchases, etc. performing such jobs makes source contention an even more pressing issue. A QoS certification effort will allow programmers to prioritize jobs and assure that nodes performing those jobs will prioritize the resources needed to complete them within a given time frame. In such a system, Spark jobs have real-time requirements, and QoS ensures that these jobs are sufficiently responsive to big data to improve analytics.

Hagar et al. [7] proposed a deep learning model using Apache Spark for network intrusion detection to achieve high performance. In comparison, Apache Spark represents an improvisation of accurate prediction. But its management is based on the incoming requests, which shows an increase in resource consumption with increasing requests and also a high wait period. Gousios [23] suggests using Apache Spark for big data analysis. It describes the computational capabilities of Apache Spark in software engineering, with high-performance analytical mechanisms. It shows a high delay in scheduling and execution with high requests in the queue. Zaharia et al. [2] describe spark scalability to support programming models and big data applications. It demonstrates effective support for workloads, highlights the importance of computability in big data programming libraries, and encourages the development of more easily interoperable libraries. Zaharia et al. [24] introduce MapReduce for large data-intensive applications. It covers machine learning algorithms for interactive data analysis using the Spark framework. It shows Spark's improvement in query response time handling.

## 2.2 Kubernetes

To have a solution for the problem of executing multiple services on millions of servers around the world, Google built its internal container infrastructure called kubernetes (k8s). As the demand for cloud services grows, k8s supports auto-scaling to accommodate applications [25]. If container CPU usage reaches a definite level, then k8s will keep creating new container replicas until usage falls underneath the threshold, and when the request decreases, k8s will again reduce replicas to free up cluster resources to execute other workloads which use pod [26]. Kubernetes natively provides a pod scaling service, although it will schedule pods to run on any node that meets its requirements, it will not automatically scale the infrastructure, *i.e.* allocate more or less CPU and memory to a single pod [27].

Kubernetes users can configure the open-source cluster autoscaler feature to automatically resize the cluster and add more resources when pods are waiting to run [28]. However, there are some limitations to using this tool, especially for users who wish to take a more hands-off approach to their infrastructure, but it can result in a severe resource penalty since k8s does not care about instance type or size and will schedule pods on any healthy available node. Since a pod can only run on a single node, it will wait to be scheduled until a node of sufficient capacity becomes available. This delay can translate into service disruption to customers and wasted resources. Pods on these smaller instances can be rescheduled on another node, and the cluster will remain efficient, running only as much as it needs, with all pods scheduled [29-30].

This limits the ability to utilize different instance types and sizes when using cluster autoscaler and autoscaling groups. Instances must have the same capacity (CPU and memory) if they are in the same node group. Managing multiple node pools is complex, and ASGs must be managed independently by the user. This is the concept of container-based autoscaling, which uses real-time container requirements when provisioning infrastructure instead of fitting containers to predetermined or existing instances. Improper configuration can result in idle resources and increased operational costs or application performance issues because the cluster does not have enough capacity to run. Certain features and configurations must be set up or tuned correctly to achieve low cost and application reliability. Another important consideration is the workload type, as different configurations may need to be applied to further reduce costs, based on the job category and application necessity.

Its popularity has grown since initial support for running spark on kubernetes was added to apache spark [22, 31]. The reasons are due to better isolation and resource distribution for simultaneous Spark applications in Kubernetes and the benefits of using a homogeneous and cloud-native setup for the entire technology stack of the enterprise. However, executing spark on kubernetes presents some challenges in a reliable, cost-effective, and secure manner.

Hu et al. [26] proposed improvements to scaling Kubernetes resources based on pod replica prediction. It suggests enhancements to Kubernetes that automatically scale in response to dynamic load changes. The scaling of resources completely depends on the incoming resource request for a job. Its prediction does not estimate the leftover resource which makes the loss of the resources and also increasing in pod replica creates managing overhead, which depletes the cost-effectiveness.

Liu et al. [28] describe the load-balancing component capabilities of kubernetes to support modest static capacity balancing guidelines and define a scheme to accommodate difficult commercial requirements. An added sophisticated dynamic load assessment policy is employed, which can more compliantly tailor the capacity balancing scheme to real commercial necessities. It defines a static capacity model which reserves the resource in prior which makes unavailability of pods for incoming variable jobs need. Even though it manages the resource load effectively but it shows high scheduling overhead with increasing job requests.

Z. He [5] describes a container technology-based cluster management system that manages large-scale containers in kubernetes. He provides predictive models and request control mechanisms for incoming application requests, and creates service models to demonstrate cluster manageability and query response time improvements in Kubernetes. It shows an effective model for managing incoming job requests, but in case of high application requests it attains a high delay of scheduling, and also increasing the container size impacts the cost of the resources which reduces the cost efficiency of this model at high job requests.

Huaxin et al. [11] propose a better-quality scheduling algorithm concerned with a multi-occupant model where cluster consumers are sculpted as virtual clusters and a load of the cluster is regularly supervised in kubernetes. Rossi et al. [10] demonstrated the geographically spread and flexible placement of kubernetes containers with a planning tool that depends on kubernetes and prolongs it with adaptive and network-aware scheduling competencies. It offers an elastic and decentralized control loop that will be effortlessly adapted to dissimilar utilization schemes. Experimental evaluations demonstrate the benefits of combining stability and deployment strategies and the significance of using network-aware deployment strategies when deploying kubernetes in geographically spread environments.

Islam et al. [12] propose efficient scheduling algorithms that minimize resource consumption on a cloud-hosted apache spark cluster. It suggests prioritizing work within a given time frame, as well as a scheduling algorithm to monitor and adapt to cluster changes. It executed spark with apache mesos to demonstrate the application's performance. As a result, resource utilization reduction of up to 34% is shown under various workload conditions. But here two constraints are observed concerning time and resources. If the resource is managed for optimizing the cost then scheduling delay is increasing with increasing job requests, in vice-versa if scheduling time is met to avoid the deadline violation then high resources are utilized which impacts the cost-effectiveness. So, to have a balance model for both time and resources an accurate prediction mechanism is needed which can able to manage the resource and reduce the pod replication and time effectively with the increasing demand of jobs with a minimum scheduling overhead and high-cost efficiency. In this paper, we propose an effective resource management and scheduling mechanism using Kubernetes cluster management to determine the improvement in the job scheduling mechanism.

## 3. Proposed job scheduling mechanism

Jobs are used to creating one or more pods and utilizing resources to meet the growing demands of an application. Pod replication utilizes additional resources and reduces the cost-effectiveness of the system. To improve cost efficiency, we propose a job scheduling mechanism (JSM) in the created pods. JSM identifies over-loaded or low-loaded pods and relocates jobs from over-loaded pods to low-loaded pods.

The execution of JSM considers an IaaS configuration model where the service provider manages the system with the support of cloud, local, and application controllers. In such an IaaS model application controller leverages various built-in software technologies such as apache spark to control incoming user requests, and the cloud manager manages cloud facilities at the API level and ensures that user requests are directed to the desired application. Local administrators are responsible for managing the internal resources of the node and assigning jobs for the execution of requests. Fig. 1 shows the workflow model of JSM

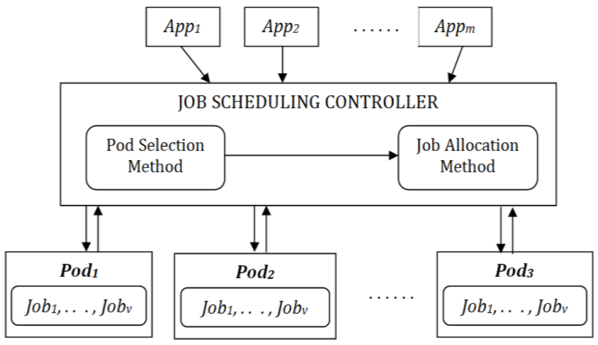

Figure. 1 Workflow model of JSM functions

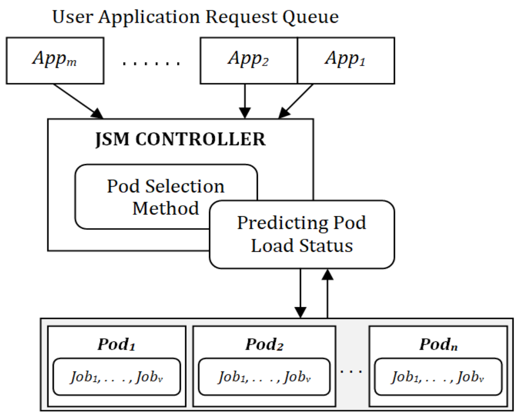

Figure. 2 Pod selections

functional modules.

These pods run different application jobs, which can be very dissimilar from each other but run simultaneously in the same node container. Based on receiving workloads, multiple pods on a node can be dynamically created or share the resources of containers. Since inbound workloads vary greatly and the resource requirements of each activity are different, effective resource and time management must be planned. To optimize the job workloads it can place dynamically in the pods to utilize the unused pod resources. Hence, the node pods which are idle or have no jobs to execute can be terminated to make it cost-effective.

### 3.1 Job scheduling methodology

The goal of JSM is to reduce the use of resources and increase cost efficiency. Methods of dynamic planning of operations in pods are based on two methods. The first method performs the pod's sections and the second method performs the jobs allocation.

### A. Pods selection method (PSM)

To decide on pod selection, the prediction of load on each pod needs to be gathered first. In traditional strategies, simply the most key resource

Table 1. Notations list

| Notations | Description |
|---|---|
| $L$ | Integral load value at each pod resources |
| $D$ | Set of different values of the resource of a pod |
| $w_r$ | Weight co-efficient of a pod |
| $f_{d_r}$ | Pod values for memory and CPU |
| $f_{cpu}$ | CPU co-efficient |
| $f_{mem}$ | Memory co-efficient |
| $C_{use}$ | Current usage of CPU |
| $C_{resv}$ | Current reserve of CPU |
| $C_{total}$ | Total CPU capacity |
| $M_{use}$ | Current usage of Memory |
| $M_{resv}$ | Current Reserve of Memory |
| $M_{total}$ | Total Memory size |
| $w_{cpu}$ | weighted coefficient of the CPU |
| $w_{mem}$ | weighted coefficient of the Memory |
| $W$ | A vector of the weighted value of Pods |

elements are utilized for the load prediction, which usually focuses on the CPU performance.

In this regard, the cluster configuration and accurate prediction in cloud hosting servers are challenging due to their dynamic variability. The workflow of the mechanism of pod selection in PSM is shown in Fig. 2. Here, the PSM method interrelates with a set of pods and computes load conditions to identify the most appropriate pods for allocating inbound requests by the applications. The description of the utilized notations for computation is shown in Table 1.

According to the decentralized scheme, each node maintains a vector to store the load factor values obtained from pods as load keys. Thus, L is used to measure the integral load value at each pod resource, and it describes the actual load condition in the pod using Eq. (1).

$$L = \frac{\sum_{r=0}^{D} w_r f_{d_r}}{D} \tag{1}$$

where D is a set of different values of the resource of a pod, wr symbolizes the weight co-efficient of the rth pod and $f_{d_r}$ symbolizes pod values for memory and CPU, which are utilized to classify a pod for selection. The value of CPU and Memory is the most influential feature in the pod selection, so we compute CPU and Memory co-efficient using Eqs. (2) and (3),

$$f_{cpu} = 1 - \frac{C_{use}}{C_{total} - C_{resv}} \tag{2}$$

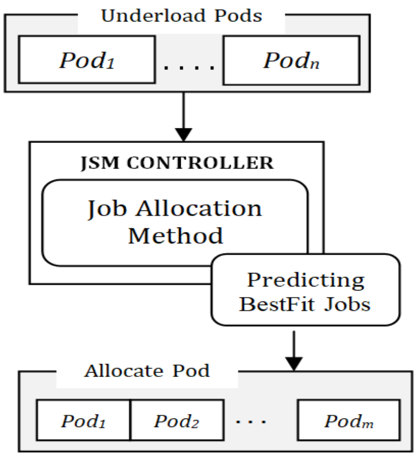

Figure. 3 Job allocations

$$f_{mem} = 1 - \frac{M_{use}}{M_{total} - M_{resv}} \tag{3}$$

Utilizing the computed fcpu and fmem we compute the required weighted coefficient of the pod node using Eqs. (4) and (5).

$$w_{cpu} = \frac{f_{cpu}}{f_{cpu} + f_{mem}} \tag{4}$$

$$w_{mem} = \frac{f_{mem}}{f_{cpu} + f_{mem}} \tag{5}$$

With computing the value of L for each pod, we define the weight vector, $W = (L_{pod_1}, L_{pod_2}, \ldots, L_{pod_n})$. Now, with the obtained value of W, a sequence of pod array from over-loaded to low-loaded is created. Based on this obtained sequence array the incoming job requests are prioritized for allocation. The process of job allocation we discuss in the next section.

## B. Job allocation method (JAM)

Allocation of job workloads is a major problem with the use of critical resources. Therefore, it is very important to predict the best herd before the work is done. The resources needed to run the distributed pod must also be guaranteed. Various activities must be performed to ensure that the pods identified in the above process do not overload new pods hosting these activities and reduce resource utilization. After recognizing the pod to which a job can be allocated we have to identify the job which can best fit the selected pod as shown in Fig. 3. To do so we implement the function of the Best-Fit method [34] to predict the most appropriate pods suitable for allocation.

Here the resources required for the job need to be considered primarily. The job is compared with the computed weighted load vector to identify the

Table 2. Load with varying resource use

| CPU/Mem Use | $f_{cpu}$ | $f_{mem}$ | $w_{cpu}$ | $w_{mem}$ | Load (L) |
|---|---|---|---|---|---|
| 0.5 | 0.44 | 0.41 | 0.49 | 0.48 | 0.79 |
| 0.6 | 0.33 | 0.29 | 0.37 | 0.35 | 0.89 |
| 0.7 | 0.22 | 0.18 | 0.25 | 0.21 | 0.95 |
| 0.8 | 0.11 | 0.06 | 0.12 | 0.07 | 0.99 |

pod for utilization based on a defined level of minimum weight load (MWL) as β1 and Highest weight load (HWL) as β2 to find the pod for allocation. The process of JAM tries to select the pod which has the least load or is idle currently. In a case, if no pod is found for allocation based on the threshold configured then a new pod is started for the allocation.

To choose the best pod for the job, we use the BestFit algorithm, which organizes the collection of jobs based on the increase in processor capacity required by the collection of pods.

Let's consider a pod having Ctotal =1, Cresv=0.10 and Mtotal =1, Mresv=0.15, then its fcpu , fmem, and wcpu , wmem will be as given in Table 2.

The computed *L* values are stored in a *WVector*, and it is utilized as input for evaluating the incoming job resource need and its allocation. Let's assume an incoming job demands a load of *JOBUtilz = 0.42*, then we select the best-fit pod within the range of minimum weighted load (MWL) as $\beta_1=0.2$ and highest weighted load (HWL) as $\beta_2=0.95$. Here, with iteration based on the number of pods count and comparing the *JOBUtilz* with the value of *WVector* we get three pods having load {*0.79, 0.89, 0.95*}, and among these the least loaded pod is considered for allocating the incoming job.

This methodology of allocation is implemented by the JAM method to discover a suitable pod for the job as described in Algorithm. 1,

**Algorithm-1: JAM method**

***Input:*** Selected pod collection with weighted load value, *WVector.*

```
1.  JOBUtilz = calculate the utilization of jobToAllocate.
2.  MWL →$\beta_1$;HWL →$\beta_2$;
3.  minUtilz=$\beta_2$;bestAllocation=-1;
4.  podcnt=sizeOf (WVector);

5.  for (p=0; p<podcnt; p++)
6.  {
7.     if (JOBUtilz+ WVector[p]>= $\beta_1$)&&(JOBUtilz+ WVector[p]<= $\beta_2$))
8.     {
9.       if (WVector[p]<minUtilz)
10.      {
11.         minUtilz=WVector[p];
12.         bestAllocation =Pod[p] ;
13.      }
14.    }
15. }
16. if (bestAllocation>0) {
17.      JobToAllocate=bestAllocation;
18. }
19. else {
20.      createNewPod(JobToAllocate);
21.  }
```

Algorithm-1 takes the selected collection from the PSM method as a *WVector* to determine the distribution of jobs. Here, for each pod in the *WVector*, iterative estimation is performed compared to every performance requirement to identify the best allocation. It calculates the sum of *JOBUtilz* of the present job with the pod weight value and then compares it with the threshold value. If it is>= $\beta_1$ and <= $\beta_2$, and the present value of the *JOBUtilz* pod is lower than the relative to a minimum utility constant (*minUtilz*) which is configured to 0.2, then it is mapped to the best pod. If no pod is found, then a new pod will be created to perform the job.

## 4. Experiment evaluation

To evaluate the effectiveness of the proposed JSM, we configure the spark application on a kubernetes cluster in a self-contained environment with identical nodes and implement the JSM method to predict low-loaded pods to distribute the incoming Spark application requests. For evaluation, we set up a comparison program with baseline schedulers over benchmark application data [33]. Evaluation measures are used to measure costs, performance, and cost planning. The VMs are configured on the GCP platform with 4 CPU cores, 16 GB of memory, and 20 GB of storage for a cost of $0.24/hour.

### 4.1 Benchmark applications data

We evaluate the enhancement of the proposed JSM using benchmark data information provided in BigDataBench [33]. We selected three types of application outputs for comparison as discussed by Islam et al. [12] WordCount, Sort, and PageRank.

It creates a varying job workload input ranging from 1 Gb to 20 Gb for evaluating the execution performance. These jobs are extracted from the Facebook and Hadoop response traces for these varying jobs. Incoming jobs vary according to high and low load for the different intervals of period. There are more than 100 job requests when the load is high and nearly 50 job requests when the load is low. Therefore, during periods of high load, maximum pod resources are over-loaded, whereas in low-load pods they are underutilized.

## 4.2 Baseline schedulers performance

The problem with most cluster scheduling methods for spark jobs is that they don't take facilitator-level job assignments into account. These methods primarily focus on choosing the resources or several nodes required by the respective job when it performing a decision for scheduling. However, the proposed JSM executes at the level of pod selectivity by incorporating resource utilization predictions to efficiently schedule for assigning jobs. The following scheduler is compared to identify the improvement of the proposal.

- *FIFO*: Apache spark's default FIFO scheduler is installed on apache mesos. Here, jobs are scheduled based on a first-come first-service. Instead of using the scheduler's merge preference, it distributes executors in a round-robin manner. Most existing scheduling algorithms use this preferred method of placing jobs and choose this scheduler as one of their baselines as it is also a common choice for users using spark jobs.
- *Morpheus* [32]: In this strategy, low-cost packaging is utilized for actuator assignment. Based on the present load in a cluster it employs the strategy to detect the job's resource requirements (such as memory or CPU cores). Later, the jobs are sorted in ascending order as per the limited resource requirements. As a result, the cluster's resource is well-proportioned during the scheduling and allows for running maximum jobs.
- *BFD* [12]**:** This proposal is a greedy procedure inherited from the best-fit decreasing (BFD) heuristic to optimize the cost of apache spark clusters deployed using apache mesos as cluster scheduler for the applications.

## 4.3 Result evaluation

### A. Cost efficiency

This evaluation shows that the proposed scheduling algorithm can be applied to diverse kinds of applications while decreasing the cost of employing big data clusters. It stores the state of several pods used by the node to compute the sum of the cost acquired by the scheduler. Here, over period T, we summed up the number of pods used for the various jobs. The number of pods utilized in a given period is directly proportional to the running cost. So, the total cost we computed using the given Eq. (6).

$$Total\_Cost = \sum_{t \in T} NumberofPods(t) \qquad (6)$$

It calculates the cost of each scheduling algorithm under heavy and light workloads to evaluate cost-effectiveness.

Tables 3 and 4 shows the sum of cost acquired by diverse scheduling procedures during low and high job load period respectively. The proposed JSM utilizes pod resources efficiently, significantly reducing the cost of running jobs compared to other schedulers. Assigning work with effective scheduling allows JSM to achieve better cost efficiencies compared to BFD, Morpheus, and FIFO. Besides, Morpheus achieves somewhat improved than FIFO to keep costs down due to it jobs orders in a queue that balances the cluster resources to run additional jobs in the whole scheduling technique.

Table 3. Schedulers cost having low-load

| **Workload Type** | **FIFO** | **MORPHEUS** | **BFD** | **JSM** |
|---|---|---|---|---|
| WordCount | 3.1 | 2.85 | 2.2 | 1.58 |
| Sort | 3.4 | 3.2 | 2.4 | 1.61 |
| PageRank | 6.82 | 6.54 | 6.12 | 5.14 |

Table 4. Schedulers cost having high-load

| **Workload Type** | **FIFO** | **MORPHEUS** | **BFD** | **JSM** |
|---|---|---|---|---|
| WordCount | 4.5 | 4.35 | 3.76 | 3.45 |
| Sort | 4.31 | 4.2 | 4.15 | 3.81 |
| PageRank | 7.52 | 7.32 | 7.21 | 6.85 |

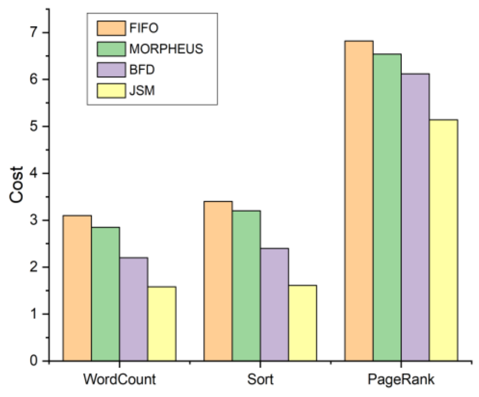

Figure. 4 Cost of schedulars at low-load

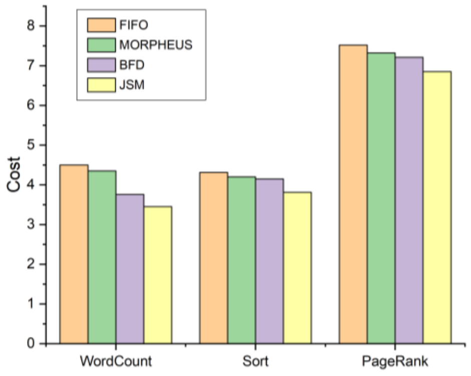

Figure. 5 Cost of schedulars at high-load

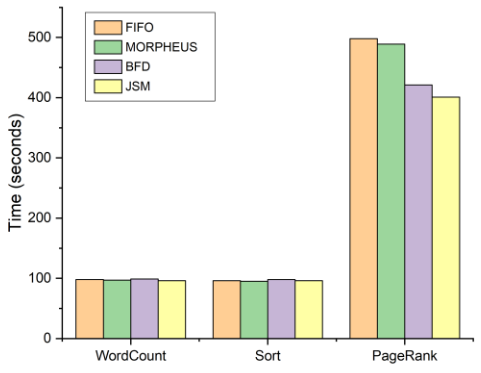

Figure. 6 Job execution performance at low load

Table 5. Job performance having low-load

| Workload Type | FIFO | MORPHEUS | BFD | JSM |
|---|---|---|---|---|
| WordCount | 98 | 97 | 99 | 96 |
| Sort | 96 | 95 | 98 | 96 |
| PageRank | 498 | 489 | 421 | 401 |

Table 6. Job performance having high-load

| Workload Type | FIFO | MORPHEUS | BFD | JSM |
|---|---|---|---|---|
| WordCount | 97 | 99 | 100 | 95 |
| Sort | 125 | 128 | 124 | 120 |
| PageRank | 302 | 303 | 295 | 292 |

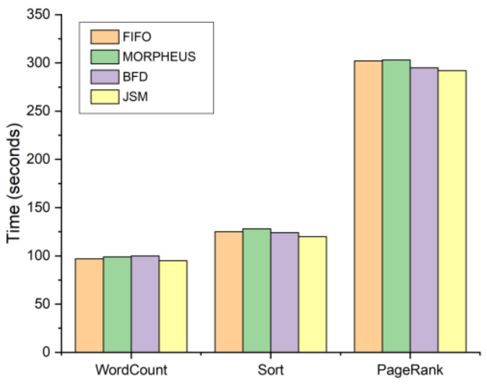

Figure. 7 Job execution performance at high load

Fig. 4 shows the proposed JSM exhibits significant cost reduction during the low-load period. In comparison to the baseline scheduling algorithm, JSM reduces the cost of cluster usage for WordCount and sort applications by at least 35% and 40%, respectively. For PageRank applications, JSM reduces resource consumption costs by at least 15% compared to FIFO. JSM also decreases resource utilization costs by 8% related to morpheus. The proposed JSM attempts to allocate the jobs of the similar assignment in fewer pods, so most of the random work happens within the pods, improving job performance and thus reducing the overall cost of applications. As shown in Fig. 5, during high-load the decrease in cost is lesser than low-load due to the overused of the cluster. In this situation, JSM shows cost efficiency near 6% to 24% with variation in the value of workloads.

## B. Job Performance

Tables 5 and 6 shows the average job finishing time for the scheduling procedures at low and high-load variation.

It shows similar or marginally improved by FIFO, morpheus, and BFD in comparison to the JSM with WordCount and sort. The enhancement of JSM is due to the use of fewer pods to accommodate the workloads and utilizing the available resources to their maximum efficiency as shown in Figs. 6 and 7.

In contrast, network-based applications such as PageRank degrade the result of FIFO, morpheus, and BFD because of unwarranted communication during the job allocation. However, JSM outperforms the comparing procedure in both load situations with PageRank applications. During high-load periods, the cluster gets over-loaded, so it is not likely to combine a few pods shaving the same kind's job. Hence, the outcome shows a better for PageRank applications at low load hours than at high load hours. During low-load hours, JSM improves runtime by at least 15% with WordCount, Sort, and PageRank by 5%.

## C. Scheduling overhead

Here we evaluate the scheduling costs of several scheduling procedures in terms of deadline. The deadline is defined as proportional to the time it takes the scheduler to allocate jobs that are waiting in the queue for execution. The obtain values of each schedulers are given in Table 7.

Table 7. Scheduling overhead comparison

| Schedulers | Deadline Violation (%) |
|---|---|
| FIFO | 43 |
| MORPHEUS | 35 |
| BFD | 8 |
| JSM | 12 |

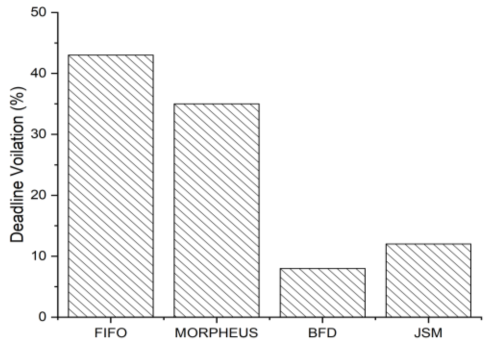

Figure.8 Scheduling overload for deadline violation

In FIFO mode, the important job having the first deadline must have to wait in the scheduling queue, if it is requested after a few non-importance jobs. It needs to wait till other jobs have been completed which are in the queue ahead of it. Morpheus independently determines the priority of work, where the work that leads is well-adjusted for the important job through sharing of resources in the cluster. But, in reality, time-constrained importance may not provide a balanced distribution of resources during deployment. Therefore, other important jobs are performed before these jobs. BFD uses a modest scheme known as "Earliest Deadline First", where entire jobs are organized as per their deadlines, and the job has the first deadline is scheduled first. The proposed JSM follows a similar allocation scheme to schedule jobs and allocate the best jobs using the BestFit method.

Fig. 8 depicts scheduling overload in the case of deadline violation percentages for different schedulers. Here the higher the violation, the more overload in scheduling. In such cases, FIFO show 41% and Morpheus shows 35% of jobs had deadline violations. Here, BFD shows a minimum deadline will of 8%. Compared with BFD, the proposed JSM has a 3% higher deadline violation rate. It has a marginally more time violation than BFD because it occasionally acquires a long period to identify the best cost-efficient allocation with this method, which arise more time violations compared to BFD. This makes JSM slightly higher scheduling overhead than BFD. However, this overhead can be ignored in comparison to baseline scheduling procedures.

Table 8. Cost efficiency performance comparison

| No. of Requests | [5] | [12] | [26] | [28] | JSM |
|---|---|---|---|---|---|
| 2000 | 3.89 | 3.75 | 5.29 | 3.79 | 3.45 |
| 4000 | 4.66 | 4.15 | 5.95 | 4.5 | 3.81 |
| 6000 | 6.88 | 7.21 | 7.25 | 5.88 | 5.85 |
| 8000 | 7.85 | 7.81 | 9.54 | 7.59 | 6.54 |
| 10000 | 9.81 | 8.33 | 11.28 | 8.9 | 7.52 |

## D. Comparison with schedulers

In this section, we compare the proposed JSM with a few schedulers for evaluating the effectiveness of our proposal by measuring the cost efficiency and scheduling overhead measure. We compare with the scheduler proposed by Z. He [5] for managing large-scale job requests in Kubernetes using predictive models and request control mechanisms, Islam et al. [12] suggest a scheduler through job prioritizing and resource management in spark with apache mesos, Hu et al. [26] suggest a scheduler for auto-scaling of resources to dynamic managing load in kubernetes, and Liu et al. [28] presents a load-balancing scheduler based defined balancing guidelines to accommodate the incoming jobs in Kubernetes.

All these schedulers are well managed and schedule the incoming job at low load, but at high load, they show variability in the results of cost efficiency and scheduling overhead as given in Tables 8 and 9.

Fig. 9 shows the cost efficiency performance of the proposed JAM in comparing the schedulers. All the methods show an increase in cost with the increasing number of requests. It is due to the increasing job queue demands more resources which inflame the usage and reduces the cost efficiency (higher the cost lower the efficiency). The proposal given in [26] shows a high-cost value because it frequently does pod replication to balance the load of the incoming requests, whereas [5, 12, 28] and the proposed JSM shows nearby similar cost efficiency at lower request numbers, but with increasing requests they show an average growth in their cost it is because all these schedule methods are based on prediction and resource management. The proposed JSM shows a lower cost at a higher number of costs due to efficiently allocating the job within the best-fit pod with the least pod creation. It shows an average of 2% better cost efficiency in comparison.

Fig. 10 shows the scheduling overhead among the schedulers. Here, [26] shows the least scheduling overhead due to its high scale pod creation with load variation due to which it attains

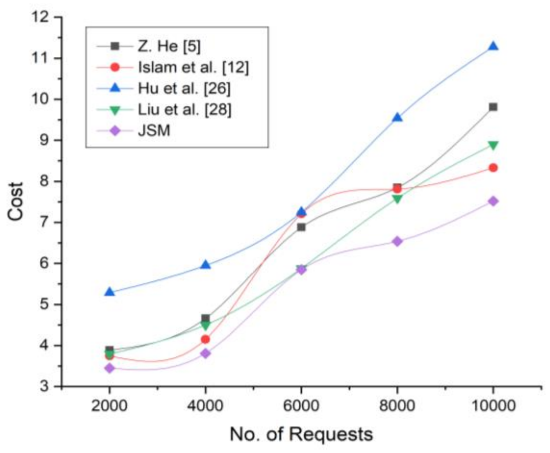

Figure. 9 Cost efficiency performance

Table 9. Scheduling overload comparison

| No. of Requests | [5] | [12] | [26] | [28] | JSM |
|---|---|---|---|---|---|
| 2000 | 1 | 1 | 1 | 1 | 1 |
| 4000 | 3 | 2 | 1 | 3 | 2 |
| 6000 | 12 | 8 | 3 | 10 | 8 |
| 8000 | 14 | 12 | 6 | 12 | 10 |
| 10000 | 18 | 15 | 10 | 16 | 12 |

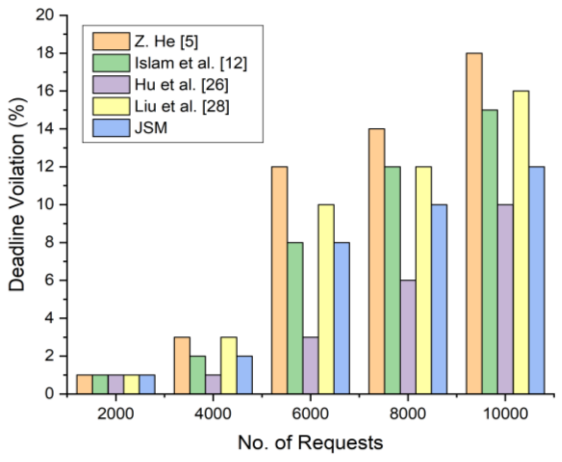

Figure. 10 Scheduling overload performance

high cost as a limitation. The other schedulers also show similar scheduling overhead in case of low request numbers, but at high numbers, an 3-4 % of deadline violation is observed. It is due to the effective management of resources by employing load-balancing and employing the assessment policy by [28], whereas [5] employ the predictive models and control mechanisms for incoming application requests, and the [12] employ efficient scheduling algorithms that minimize resource consumption which causes the deadline violation in high request. The proposed JSM shows a 3% lower scheduling overhead in comparison to [5, 12, 28] due to its allocation scheme to schedule jobs and allocate the best jobs using the BestFit method. As a result, resource utilization reduction and average scheduling overhead make JSM a cost-efficient scheduler for application systems.

## 5. Conclusion

Job scheduling seems to be a challenge for big data processing in distributed cloud computing. This paper describes a job scheduling method (JSM) for apache spark on kubernetes to improve cost efficiency. The JSM defines two methods for predicting the overload and underload of pods running jobs and assigning jobs to underload pods to conserve resources on nodes. The PSM is defined for selecting pods, and the JSM is to find the best running pod for a specific job run.

It contributes a mechanism to identify the upcoming workload of a pod and determine the best fit using PSM. It also reduces the usage of CPU and memory which result in enhancing cost efficiency, and the JSM contributes to the effective management of job allocation and migration among the underload pods to shut down the pod to preserve the resources and enhances cost efficiency. The comparison with the existing scheduler with varying requests shows an improvisation of 2% of cost efficiency and 3% lower scheduling overhead.

We demonstrated extensive experimental results on benchmark application datasets to demonstrate the efficiency of the proposed JSM in different types of workloads. We also compared the algorithm to the existing default scheduler. The results recommend that the JSM scheduling method reduces the cost of resource usage by up to 35-40% in Apache Spark clusters deployed in the cloud. Evaluation measures of cost, job performance, and scheduling overhead show improved cost efficiency for job execution. The proposed work will evaluate the performance of heterogeneous work applications with different workload types to further improve work scheduling and cost efficiency.

## Conflicts of interest

The authors declare no conflict of interest.

## Author contributions

Conceptualization, methodology, software, validation, Jayanthi; formal analysis, investigation, resources, data curation, Jayanthi; writing—original draft preparation, writing—review and editing, Jayanthi; visualization, supervision, Dr. K. Ram Mohan Rao.

Received: March 14, 2023. Revised: April 13, 2023.

## References

[1] R. Busetti, N. E. Ioini, H. R. Barzegar, C. Pahl, "Distributed Synchronous Particle Swarm Optimization for Edge Computing", In: *Proc. of International Conf. on Future Internet of Things and Cloud (FiCloud)*, Rome, Italy, pp. 153-160, 2022.

[2] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, I. Stoica, "Apache spark: a unified engine for big data processing", *Communications of the ACM*, Vol. 59, No. 11, pp 56–65, 2016.

[3] Y. Mao, Y. Fu, W. Zheng, L. Cheng, Q. Liu, and D. Tao, "Speculative Container Scheduling for Deep Learning Applications in a Kubernetes Cluster", *IEEE Systems Journal*, Vol. 16, No. 3, pp. 3770-3781, 2022,

[4] A. Marchese and O. Tomarchio, "Network-Aware Container Placement in Cloud-Edge Kubernetes Clusters", In: *Proc. of IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, Taormina, Italy, pp. 859-865, *2022*

[5] Z. He, "Novel Container Cloud Elastic Scaling Strategy based on Kubernetes", In: *Proc. of International Conf. of Information Technology and Mechatronics Engineering Conference (ITOEC)*, Chongqing, China, pp. 1400-1404, 2020.

[6] M. A. A. Fattah, N. A. Othman, and N. Goher, "Predicting Chronic Kidney Disease Using Hybrid Machine Learning Based on Apache Spark", *Computational Intelligence and Neuroscience*, Vol. 2022, Article ID 9898831, p. 12, 2022.

[7] A. A. Hagar and B. W. Gawali, "Apache Spark and Deep Learning Models for High-Performance Network Intrusion Detection Using CSE-CIC-IDS2018", *Computational Intelligence and Neuroscience*, Vol. 2022, Article ID 3131153, pp. 11, 2022.

[8] M. N. Tran, D. D. Vu, and Y. Kim, "A Survey of Autoscaling in Kubernetes", In: *Proc. of International Conf. on Ubiquitous and Future Networks (ICUFN)*, Barcelona, Spain, pp. 263-265, 2022.

[9] N. T. Nguyen and Y. Kim, "A Design of Resource Allocation Structure for Multi-Tenant Services in Kubernetes Cluster", In: *Proc. of International Conf. on Communications (APCC)*, Jeju Island, Korea, pp. 651-654, 2022.

[10] F. Rossi, V. Cardellini, F. L. Presti, and M. Nardelli, "Geo-distributed efficient deployment of containers with Kubernetes", *Computer Communications*, Vol. 159, pp. 161–174, 2020.

[11] S. Huaxin, X. Gu, K. Ping, and H. Hongyu, "An Improved Kubernetes Scheduling Algorithm for Deep Learning Platform", In: *Proc. of International Computer Conf. on Wavelet Active Media Technology and Information Processing*, pp. 113 – 116, 2020.

[12] T. Islam, S. N. Sriramaa, S. Karunasekeraa, and R. Buyya, "Cost-efficient dynamic scheduling of big data applications in Apache spark on cloud", *Journal of Systems and Software*, Vol. 162, p. 110515, 2020.

[13] C. Li, L. Huo, and H. Chen, "Real-time workflow oriented hybrid scheduling approach with balancing host weighted square frequencies in clouds", *IEEE Access*, Vol. 8, pp. 40828-40837, 2020.

[14] S. Jangiti, S. E, R. Jayaraman, H. Ramprasad, and V. S. S. Sriram, "Resource ratio based virtual machine placement in heterogeneous cloud data centres", *Sadhana*, Vol. 44, Article: 236, 2019.

[15] Z. Li, S. Guo, L. Yu, and V. Chang, "Evidence-Efficient Affinity Propagation Scheme for Virtual Machine Placement in Data Center", *IEEE Access*, Vol. 8, pp. 158356-158368, 2020.

[16] Z. Liu, "Research on Public Management Application Innovation Based on Spark Big Data Framework", *Mathematical Problems in Engineering*, Vol. 2022, ArticleID: 3797050, p. 9, 2022.

[17] Z. Wan, Z. Zhang, R. Yin, and G. Yu, "KFIML: Kubernetes-Based Fog Computing IoT Platform for Online Machine Learning", *IEEE Internet of Things Journal*, Vol. 9, No. 19, pp. 19463-19476, 2022.

[18] J. Yan and X. Ma, "Microblog Emotion Analysis Method Using Deep Learning in Spark Big Data Environment", *Mobile Information Systems*, ArticleID: 1909312, p. 9, 2022.

[19] W. Zhouhuo, "Parallel Classification Algorithm Design of Human Resource Big Data Based on Spark Platform", *Security and Communication Networks*, Vol. 2021, ArticleID: 5811918, p. 10, 2021.

[20] J. Wang, "Clustering Algorithm for Big Datasets with Mixed Attribute Features under Spark", *Mathematical Problems in Engineering*, Vol. 2022, ArticleID: 6156799, p. 11, 2022.

[21] S. Ilbeigipour, A. Albadvi, and E. A. Noughabi, "Real-Time Heart Arrhythmia Detection Using Apache Spark Structured Streaming", *Journal*

*of Healthcare Engineering*, Vol. 2021, ArticleID: 6624829, pp. 13, 2021.

[22] M. A. Mohamed, I. M. E. Henawy, and A. Salah, "Usages of Spark Framework with Different Machine Learning", *Algorithms Computational Intelligence and Neuroscience*, Vol. 2021, ArticleID: 1896953, p. 7, 2021.

[23] G. Gousios, "Big Data Software Analytics with Apache Spark", In: *Proc. of International Conf. on Software Engineering: Companion*, Gothenburg, Sweden, pp. 542-543, 2018.

[24] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets", In: *Proc. of International USENIX Conf. on Hot Topics in Cloud Computing*, p. 10, 2010.

[25] A. P. Ferreira and R. Sinnott, "A Performance Evaluation of Containers Running on Managed Kubernetes Services", In: *Proc. of International Conf. on Cloud Computing Technology and Science (CloudCom)*, Sydney, NSW, Australia, pp. 199-208, 2019.

[26] T. Hu and Y. Wang, "A Kubernetes Autoscaler Based on Pod Replicas Prediction", In: *Proc. of International Asia-Pacific Conf. on Communications Technology and Computer Science (ACCTCS)*, Shenyang, China, pp. 238-241, 2021.

[27] J. Sithiyopasakul, T. Archevapanich, B. Purahong, P. Sithiyopasakul, and C. Benjangkaprasert, "Automated Resource Management System based on Kubernetes Technology", In: *Proc. of International Conf. on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON)*, Chiang Mai, Thailand, pp. 1146-1149, 2021.

[28] Q. Liu, E. Haihong, and M. Song, "The Design of Multi-Metric Load Balancer for Kubernetes", In: *Proc. of International Conf. on Inventive Computation Technologies*, Coimbatore, India, pp. 1114-1117, 2020.

[29] J. Park, U. Choi, S. Kum, J. Moon, K. Lee, "Accelerator-Aware Kubernetes Scheduler for DNN Tasks on Edge Computing Environment", *IEEE/ACM Symposium on Edge Computing (SEC)*, San Jose, CA, USA, pp. 438-440, 2021.

[30] R. Kang, M. Zhu, F. He, T. Sato, and E. Oki, "Design of Scheduler Plugins for Reliable Function Allocation in Kubernetes", In: *Proc. of International Conf. on the Design of Reliable Communication Networks*, Milano, Italy, pp. 1-3, 2021.

[31] Y. Huimin, "Research on Parallel Support Vector Machine Based on Spark Big Data Platform", *Scientific Programming*, Vol. 2021, ArticleID: 7998417, pp. 9, 2021.

[32] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, I. N. Goiri, S. Krishnan, J. Kulkarni, and S. Rao, "Morpheus: Towards automated SLOs for enterprise clusters", In: *Proc. of International USENIX Conf. on Operating Systems Design and Implementation*, pp. 117–134, 2016.

[33] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu, "Bigdatabench: A big data benchmark suite from internet services", *IEEE International Symposium on High Performance Computer Architecture*, Orlando, FL, USA, pp. 488-499, 2014.

[34] J. Sgall, "A new analysis of Best Fit bin packing", In: *Proc. of International Conf. on Fun with Algorithms*, Springer, Berlin, Heidelberg, pp 315–321, 2012.