National College of Ireland

Optimizing Kubernetes Performance by Handling Resource Contention with Custom Scheduler

MSc Research Project
Cloud Computing

Akshatha Mulubagilu Nagaraj
Student ID: 18113575

School of Computing
National College of Ireland

Supervisor: Mr. Vikas Sahni

National College of Ireland
Project Submission Sheet
School of Computing

National College of Ireland

Student Name:	Akshatha Mulubagilu Nagaraj
Student ID:	18113575
Programme:	Cloud Computing
Year:	2020
Module:	Research Project
Supervisor:	Mr. Vikas Sahni
Submission Due Date:	17/08/2020
Project Title:	Optimizing Kubernetes Performance by Handling Resource Contention with Custom Scheduler
Word Count:	5060
Page Count:	19

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

I agree to an electronic copy of my thesis being made publicly available on TRAP the National College of Ireland's Institutional Repository for consultation.

Signature:	
Date:	17th August 2020

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Optimizing Kubernetes Performance by Handling Resource Contention with Custom Scheduler

Akshatha Mulubagilu Nagaraj
18113575

Abstract

In virtualized environment, multiple instances use same resources of the environment,implying that chances of requesting similar resources for finishing jobs at same point in time is critical. Furthermore, resource contention is faced if requested resources exceed the accessibility of resources. Most cases enter a waiting state, when more than one job requests for same set of resources in completion of the task with only few requests being attended to execute. As a result of such delays, there is overall performance degradation. It is a peculiar issue which resurfaces in Kubernetes container management system. Activities like placement location and resource provision is the main aim of Kubernetes scheduler. Container placement, dependent on CPU and memory parameters, is the default and conventional rule in Kubernetes. Although it is very well known that these two factors are not the only resources in a shared environment leading to resource contention issues. Careful and exhaustive resource usage is taken into consideration for placement of the container through the implemented scheduler. To reduce this issue, the implemented solution discusses about how two containers intensely utilizing the same resources could be placed in two distinct and separate Kubernetes pods. So, this proposed Kubernetes scheduler handles resource contention problem resulting into improved performance.

1 Introduction

The growth of lightweight docker containers was initiated by increasing virtualization requirements. Monitoring of the status of the applications along with automatic management of the application is the need post adopting the docker containers. This resulted in creation of the Kubernetes container orchestration tool. Automatic installation coupled with handling of the docker containers is the main aim of Kubernetes. Buyya et al. (2018) states that for improvement in the scheduler, since many issues still limit the performance of the default scheduler, further research can be undertaken. The tendency to reduce conflicts amongst resources and curtail contention of resources can be a reality if resource allocation is in lines with the ratio of the incoming workload.

1.1 Research Question

One or more container pods consisting of multiple containers which share resources is managed by Kubernetes. Diverse workload is supplied by resources viz, CPU, ram, network, storage.

Can Kubernetes Performance be optimized by lowering Resource Contention through implementation of unique and efficient scheduling technique depending on the application resource requirement?

Default Kube-scheduler considers resource demand and in-house resource availability at each pod. The problem of resource management is handled through a resource reserve mechanism that sets the cap on CPU and memory resources. Although such system is inefficient and insufficient to handle resource contention issues. To avoid performance degradation, without reserving resources a run time schedule needs to be in place.

2 Related Work

For an in depth understanding of the resource contention challenge occurring in Kubernetes, it is essential to comprehend previous technologies to Kubernetes. Subsequently, it is essential to come to terms with the knowledge if resource contention, in earlier virtual environments, was an issue or is it unique to Kubernetes. Varied approaches of researchers whose work also aimed at reduction in resource contention issues in default Kubernetes scheduler is also reviewed in this study. This background work is subsequently divided into four subsections as listed below

2.1 Kubernetes Components and Architecture

Discussion related to fundamentals of both the technologies, namely basics of Virtualization and necessity of containerization.

2.2 Limitations of Default Scheduler

This subsection highlights container orchestration structure and findings of various researchers and the limitations of current literature on the same.

2.3 Virtualization Technologies

Post appreciating Kubernetes' architectural components, the above mentioned subsection analyses the issues faced in an earlier virtualized technology, virtual machines to Kubernetes, in relation to resource contention.

2.4 Contribution.

To illustrate this study, resource contention issues from current researches are explained with a focus on Kubernetes.

2.1 Kubernetes Components and Architecture

Division of the above depicted architecture is into two parts

1. Master Components

Medel et al. (2018) A single master component is in control of a cluster in each Kubernetes cluster. Activities like handling the scheduling task along with responding to cluster activities like initiation of a new pod is carried out by the **controller manager**. **Etcd** is a storage component where the system state is accumulated. Scheduling of each pod presented on the node is the job of **kube-scheduler**. Reception of user's command coupled with manipulation of data for Kubernetes objects is carried out by **kube-apiserver**.

2. Node components

Medel et al. (2018) As per Kubernetes structure, a single pod can hold multiple containers. Here each pod get separate unique ID by the master node and that is considered as a virtual server. Medel et al. (2018) Reporting of node events, resource utilization along with status of the pod is functioned through Kubelet node agent.

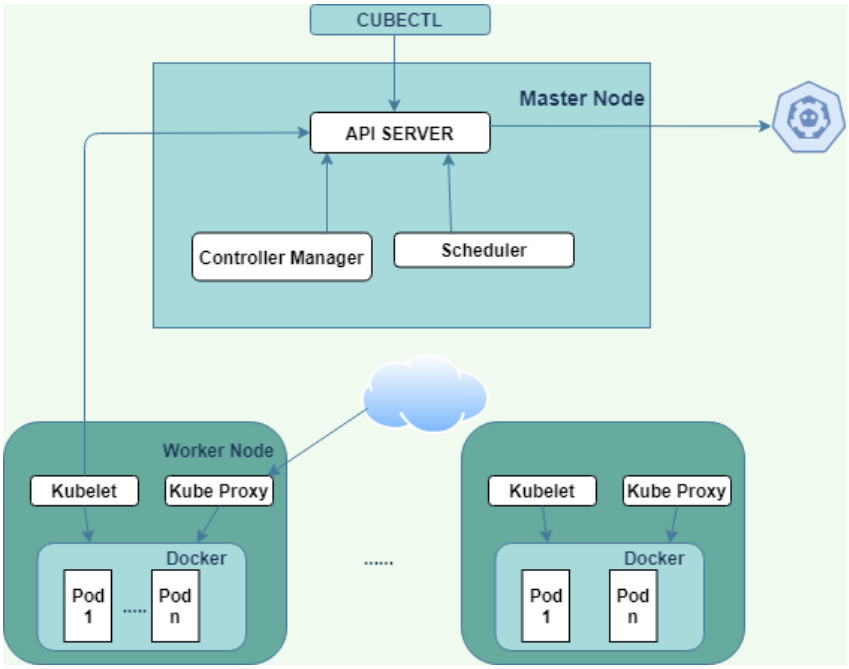


Figure 1: **Kubernetes Architecture**

2.2 Limitations of Default Scheduler

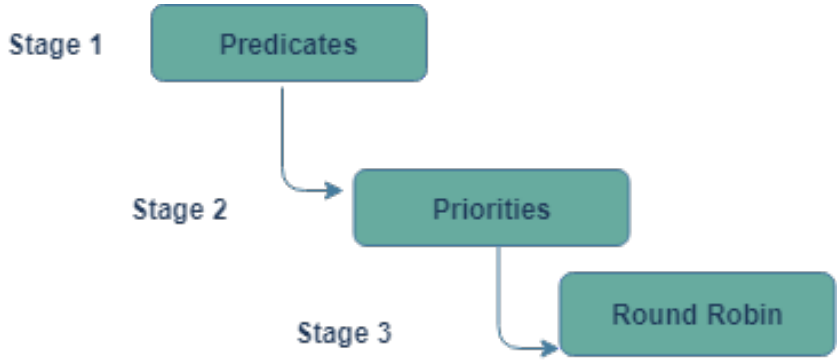


Figure 2: **Traditional Scheduler Architecture**

As illustrated above in Figure 2, there are three main steps in the selection process viz, predicates, properties and round robin. All the accessible pods are provided to Predicates' capacity and it returns valid or fake dependent on acknowledgment or dismissal of every node by pod. For handling and supplying results for node selection by arranging them in the range of 1 to 10 along with similar attributes is functioned after Prioritise. As per priority level sequence, node selection process takes place where nodes from this priority sequence get picked. Hence, nodes at same priority level get elected or served resources on round robin basis.

Round robin technology is used by default scheduler in pods for container positioning. Though, there is no control over the placement of the job. This implies that if similar types of requests are assigned it would lead to demand for similar types of resources. Consequently, many requests do not get served since containers of these jobs contend with each other serving only few of the requests ; leading to resource contention issues.

To solve the issue of performance degradation and resource contention, solution provided by default scheduler specifying limits for CPU and memory are insufficient. Performance is improved at some level through reserve mechanism although it does lead to unused cluster resources. This is why resource reservation is not a full proof solution for resource contention issues. To overcome this difficulty, it would be a better plan to not to put services together which are intensely utilizing same kind of resources.

2.3 Virtualization Technologies

Virtualization technologies are high on radar in the current times since they are majorly utilized in cloud computing. The ability to share multiple computing resources without the interference in execution of other tasks, virtual machine is relied upon to solve the issue of dedicated hardware resources. For controlling resource usage and ensuring that workload remains isolated, virtual machines provide the impression of real physical resources. Felter et al. (2015). Virtual Machine (VM) encapsulates the operating system and application through virtual technology. A single physical machine (PM) helps in the execution of such multiple virtual machines Nasim et al. (2016). Degradation of VM performance can be reduced by increasing the chances of resource contention occurring through physical resource sharing. Consequently, a vital role in maintaining the VM performance is conditional on VM placement decision. Zhao et al. (2018) Nasim et al. (2016). Buyya et al. (2018) state that the process of overcoming the wastage of dedicated resources through sharing of multiple nodes is containerization. Mavridis and Karatza (2019) performed an analysis of how an application's output and performance is influenced by configuration of virtual machines and container technologies. The conclusion of this study was the fact that configuration of is dependent on application's sensitivity. Recent introduction of Unikernels technology, according to a study of Benedictis and Lioy (2019), consists of all the elements needed for auto booting. Futuristically, this can be used as an alternative for currently existing Linux containers. Containers always cannot be alternative for VMs. This is isolated from the fact that there can be minimization of resource cost along with good utilization of resources through containers. In the end, selection of system generated best-fit match is important.Kubernetes, a Container Management System, Chang et al. (2017)open source container orchestration framework which offers resolution for bundle of nodes, VM and container solutions are restricted to single nodes only. Scenarios of under-provision along with over provision is prevented through this tool. Work by Medel et al. (2018) analyse container and pods performance in Kubernetes through deployment, maintenance and termination phases. Improved resource management and overall capacity planning can be executed through Petri net-based performance model. The application's design including circumstantial structure of the containers and pods is achieved through it. Since the chances of resource contention issues are not considered, the utility is limited only to performance management of the Docker containers.

Resource Contention in various virtualization technologies

To study above research question, this section does analysis of resource contention problem in earlier technologies such as virtual machine, docker containers as well as Kubernetes. Additionally, it dwells and seeks an explanation into the issues of performance degradation due to challenges in resource contention.

1. **Virtual Machine**

Operating multiple VMs on identical PM enhances resource utilization, according to Zhao

et al. (2018). Since resources are shared, there is high probability that resource contention issues would crop up since similar resources are being employed in turn heading towards degradation of performance. Isolation of resources is achieved through hypervisor of VMs but that is not enough. Isolation, again, is challenging for bandwidth, cache, and memory. According to Nasim et al. (2016a), migration of VMs to a smaller number of hosts is carried out to save energy from unutilized hosts. Due to such migration, there are variations in characteristics which leads to performance degradation. An optimized VM placement solution is provided through this research of performing quantitative methods of assessments of requested resources like disk, memory and CPU. In a study by Blagodurov et al. (2010), multiple researches focus on analysis of the performance intrusion for detection of inter-VM contention. Hence, Blagodurov et al. (2010) proposes mitigation of the issue by interference and detection interference at PM level metrics.

2. **Containers** Containers were developed to conquer constraints of virtual machine. Oh et al. (2018) states that due to resource contention issues container based clusters are facing performance loss. GPU sharing between containers was the focused area of this research where they succeeded sharing with reduced memory loss. By employing adaptive fair-share technique, we can look at minimization of resource contention issues. On basis of their importance, each active container is provided with different amount of memory. This is done instead of storing equal quantity of memory for each container. But the limitations of this research is the fact that it is based upon GPU parameters resulting into improved performance. In addition to that, Kim et al. (2018) does research on similar problem but that is in containerized scientific workflows where he provides solution of Hierarchical Recursive Resource Sharing. In this case, along with dynamic resource allocation technique they also maintain multi-tier applications. This provides maximum number of resources to request in the top most tier and the minimal for the rest of the tiers.

3. **Kubernetes**

Resource management according to Blagodurov et al. (2010) states that it is the process of assigning resources like computing processes, virtual machines, storage, network, nodes to an application. Further Madni et al. (2017) mentions that, resource congestion occurs when more than one user request for similar sets of resources of a particular instance. Additionally, he also states that resource sharing amongst containers is similar to resource sharing of virtual machines. This implies that requesting nodes compete amongst themselves when there is a request for more than one single node. Blagodurov et al. (2010) also states that based on resource availability, Kubernetes scheduler is liable to align container placement. Considering parameters like CPU and memory, Kubernetes scheduler takes the decision of alignment. Furthermore, in a study by Fazio et al. (2016) it is concluded that Container contention is still an topic for analysis. Consumption of resource and application performance are dependent on each other. These two are interrelated factors, as when number of microservices executes on a host, few of them need more storage, few need more network likewise, requests are bandwidth or memory intensive. But since there is improper management of resources and incorrect container placement decision it leads to resource contention and performance degradation. Recognition of factors which Kubernetes scheduler contemplates on while determining on container placement is the main aim of this research.

2.4 Contribution

With minimal overhead, programmed resource management envisages rapid instance life-cycle implementation according to Medel et al. (2018). To come to that conclusion, Medel et al. (2018) work analyses launching and termination of Kubernetes overhead for distinct configurations using Petri Net based model. By monitoring network and characters of CPU, rules have been created for assigning the number of containers per pod. Although his work, for deciding the overheads, only considers CPU and network characters. For efficient resource scheduling, one has to understand characters of the application according to Figueiredo (2006). In such scenario, CPU load along with other resources such as I/O, memory, network needs to be considered for better resource scheduling. However, the constrain of this research is limited to manual selection of performance metric. This needs to be changed to automatic for handling online classification. Also, a generic program for better resource provisioning has been built by Chang et al. (2017) since dynamic resource provisioning in Kubernetes default is based on only CPU utilization which in turn is insufficient for taking a call on utilization. This generic platform considers QOS metrics along with system resource utilization along with additionally employing CPU utilization. Not only is over provisioning handled but scenarios of under provisioning are also looked after through this modularized process. And in case it falls below 40 percent, it is considered to be light load period ; it is considered to be a heavy load period when 70 percent is exceeded. This is taken into consideration if pod is to be added or released. The deployment process is simplified and streamlined of resource provisioning algorithm without affecting other modules. Concurrently considering I/O contention issues at either node level or cluster level projects the resource management problems. To take care at the node level or cluster level, McDaniel et al. (2015) has built a two-tier approach by combining the qualities of docker container and docker swarm, both of which are responsible for it. Depending upon the priority level- high, medium or low, from the node level I/O resources are supplied for each container. Unlike at cluster level which assigns the properties of docker swarm. Here, the work is focused on handling contention problems of containers with I/O shares. Wei-guo et al. (2018) achieved kube-scheduler performance improvement by blending properties of two different algorithms such as ant colony and swarm optimization. Through this model, load balancing is enhanced and cost reduction for resources is achieved since it only considers parameters of memory and CPU.

2.5 Literature Work Summary

Author	Research	Benefits	Limitations
I. Figueiredo(2006) Jian Zhang and Figueiredo (2006)	Classification on patterns of resource consumption	Overall enhancement in system throughput and report scheduling	Selection of resources is done manually.
Wei-guo et al. (2018) Wei-guo et al. (2018)	Combining features and properties of existing algorithms for achieving collective results such as serve node with a minimal objectives.	Performance improvement, reduction in load balancing management, leads to improved resource utilization	This work considers limited resources such as CPU and memory.
Medel et al.(2018)Medel et al. (2018)	Resource provisioning by tracking performance of an application with various configurations .	Reduction in wastage of resources, better capacity planning resulting in improved resource management in turn overall improved structure.	Research is limited to monitoring performance at different configurations without considering any other resource management policies.
McDaniel et al.(2015) McDaniel et al. (2015)	Provide solution on I/O contention problem at different levels by implementing scheduler at node and cluster level.	Reduction in I/O resource contention	Only useful for applications need more I/O.

Table 1: Literature work summary

3 Methodology

While the purpose of this analysis is to define the resource contention problem in the Kubernetes pods and, depending on the form of application scheduler, to place and container in such a way that the resource contention problem does not occur. With this implementation, scheduler positions the container as mentioned in subsequent algorithm.

3.1 Custom Scheduler Design Overview

Below described 3 scenarios works for each requested job from workload file as given in figure 3.

Scenario 1: there are 4 different types of already existing requests in the first pod then the occurrence of fifth request irrespective of the type would lead to the launch of a new pod

Scenario 2: There is a storage intensive request and in case there is already existing request in the earlier pod, even if the container count is less than 4, there is a launch of new pod to handle the storage intensive application.

Scenario 3: In case there is vacancy for a particular type of container and there are more resources available for execution, even if there are less than 4 containers in a pod, a new pod would be launched.

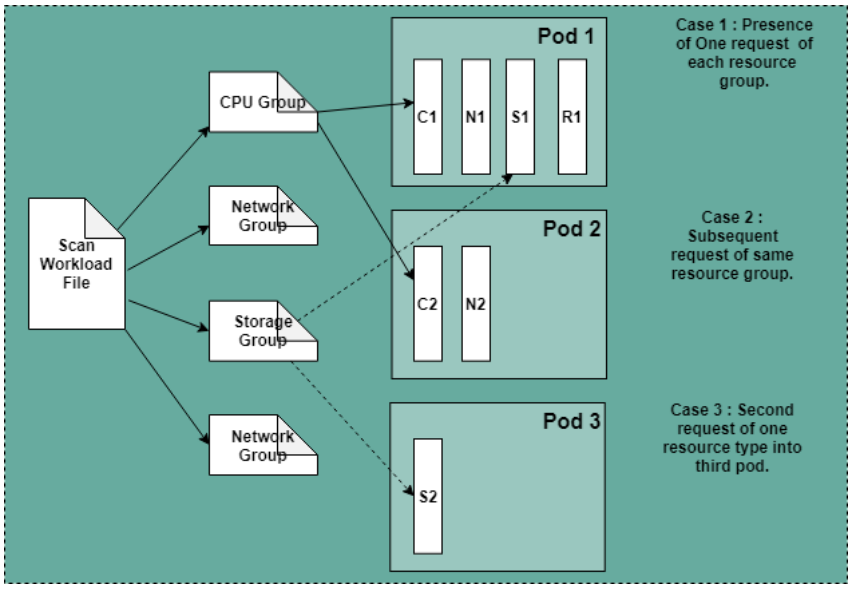


Figure 3: **Custom Scheduler Design Overview**

3.2 Working of Proposed Custom Scheduler

3.2.1 Application Classification

Algorithm 1 : Application Categorization Algorithm

Input: Application Workload, An

Result: Type of an Application.

For each application, do

1. Deploy Application on one VM.
2. Monitor application resource utilization.
3. Compare between resource utilization.
4. Categorized resource in a group of maximum utilized resource (as per Step 3 result).
5. end

This section underlines two different algorithms, in the attempt to experiment scheduling, viz, application categorization and application placement algorithms that aid in scheduling applications. Through python simulation and for testing purpose by generating fabricated workload, this research gives a unique approach in formation of an environment that aids in implementation. Requests are categorized into 4 sections using Algorithm 1 viz, storage-group, network-group, CPU-group or memory-group. Each request's utilization is calculated and post the calculation, they are classified depending upon the comparison amongst as many as possible resources. Using Application Scheduling Algorithm, custom scheduler decides the placement of the application post the category of request is identified.

3.2.2 Scheduling Algorithm

Data : Application and group of an application

Result : Application placement in respective container
For each request
Switch
Case 1 : Group == CPU
Check if (container in pod ¡=4)
Check if (container of CPU group)
Sufficient resource check()
Check existence of CPU group request ()
Launch New pod ()
else
Launch Container in same pod()
else
Launch New pod ()
Launch New pod ()
Case 2: Group == Memory
Case 3: Group == Network
Case 4: Group == Storage

The scheduling algorithm scheduler post the categorization of the application validates if an application has high mips rate than others resources. If so then it is put under CPU-group. Such a request needs to be placed in a pod. An individual pod here holds maximum of 4 containers where every single container is a representation of a different group. Before execution, it checks the count of containers in each pod is not more than 4. If that is not the case then availability of resources is confirmed. Post that types of requests are compared and validated if such a request is already existing and finally a new pod is launched. Likewise, similar request are not aligned with to the pod where there is already a request of the existing kind. There is no interference of the pods amongst each other since they function independently in a well-mannered function where each container in a pod belongs to different group.

3.3 Application Baseline and Benchmarking

Success of this implementation is validated on the basis of CPU, Network, Storage and Ram like parameters. These are considered as baseline for proposed algorithm. Further, to do comparative analysis, user has given choice to execute either using original scheduler or with proposed scheduler. Then, comparison of both outcomes can be compared to comment on implementation.

3.4 Application Execution Environment

This study executes python simulator and provides an input of fabricated workload for validation. This is done to simulate proposed scheduling technique.

3.4.1 Creation of Workload

This implementation makes use of fabricated workload. This workload is generated using random number generator. For simulation purpose it considers CPU, Storage, Ram and network parameters. These parameters can be changed as per application need. For real implementation actual data set can be passed as input for this application.

3.4.2 Simulation using Python

Python scripting language is executed for simulating this Kubernetes architecture. Previously, existing cloudsim simulator was supposed to be utilized for this research work. Since this Kubernetes service is new and any simulator is not yet fully featured. As with existing versions of cloudsim, it is possible to calculate utilization of VM and host but it is not possible to monitor resource utilization of each request. However, for real implementation GoogleCloud can be used.

4 Design Specification

Main goal of new design is to overcome drawbacks of default round-robin based algorithm and handle that by redirecting it to new custom scheduling technique. Initially, the algorithm would distinguish the arrival of new request into one of the four resources group. This is based on the level of intensive usage of resources. Consequently, proposed scheduler facilitates the decision of which container placement is to be applied. This is shown in Figure 4.

4.1 Proposed Kubernetes Scheduler Architecture Overview

In this segment, we examine and emphasize the overview of the architecture of the proposal to explore our research objective. The general use of the application is improved significantly through this strategy. This implementation works in two stages

- Application Classification
- Application Scheduling

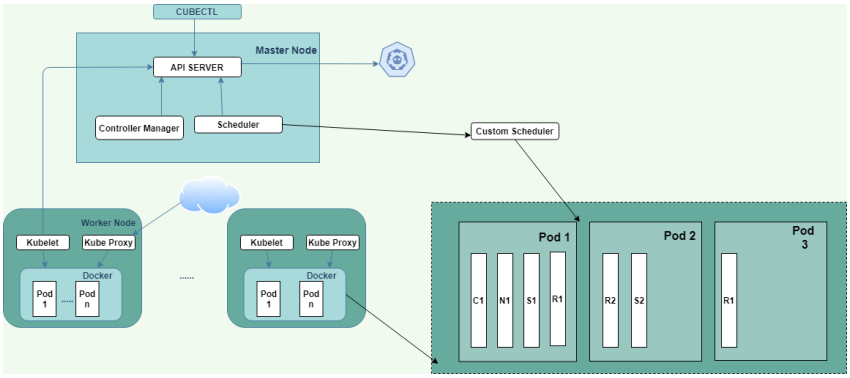


Figure 4: **Proposed Kubernetes Scheduler Architecture Overview**

In the first stage, categorisation of the application is done through characterization model using Application Categorization Algorithm which is based upon resource usage of the workload being provided. The application is distinguished into Storage group, CPU group, Ram group and Network group by comparing resource need. Then after, in second stage scheduling algorithm makes decision on to add new pod or use existing one by validating against earlier discussed cases. The proposed model ensures that different requests belonging to same resource group are not billed in the same pod. Additionally, it also ensures that accessibility and availability of the resources of the pod is not surpassed by the resources requested. Similarly, chances of same job requests intending to use similar resources are minimised when identical types of requests are placed in various pods. Thus,

reduction of resource contention, to some degree, and enhancement of the application's performance by averting over-utilization and under-utilization can be achieved through improved resource utilization is the realization and proof of the principle.

4.2 Kube-Scheduler Architectural Flow

Architectural workflow involving different stages such as scanning of generated workload, creation of pods and then launching of containers in existing pods or in new pod depending upon workload execution is illustrated in Figure 5. Briefly, the file reader pursues the information and data from the source workload file in the workload reading phase. Also, initialization of datacentre takes place with required configuration depending upon the configuration of the system consisting of amount of hosts for the datacentre. Later, to host development, the algorithm performs the workload file and determines the resource consumption for each request. Once resource utilization is identified, the type of the application is identified for execution by assessing resource usage of every single request. Workload placement is decided through custom scheduler following resource utilization. Suppose the request belonging to memory group gets served by first pod. After that if second request is of CPU group then depending on resource availability it will be assigned to the same pod of memory. But if a request is CPU intensive is followed after the first one then it gets assigned into the next pod. Network, storage and any additional resources are handled through this algorithm. Accommodation of maximum four containers, each of different types a single pod can contain.

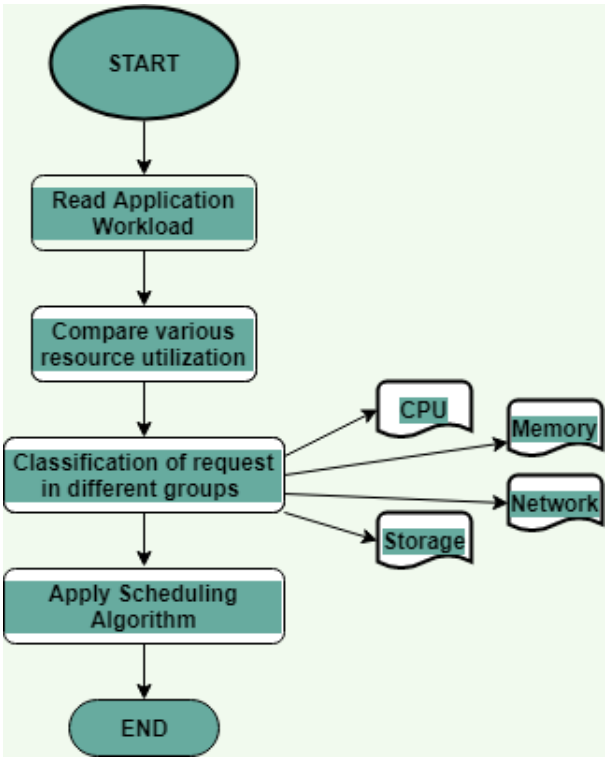


Figure 5: **Kube-Scheduler Architectural Flow**

4.3 Implementation and Deployment

To show the proposed work for genuine and real usage application needs to be deployed straightforwardly on Google cloud or simulation can be used. The proposed scheduler can be better appreciated when the work would be done through simulation utilizing python. Steps illustrated in Figure 6 need to be performed for real implementation whereas in figure 7, there are details of steps which can be carried out for simulation of this work.

4.3.1 Simulation

As for real implementation we need to monitor actual input application. In a same way or simulation, either existing dataset file like NASA or fabricated workload file needs to be presented. Simulation is done by following below mentioned steps illustrated in figure 7. First datacenter structure needs to be created and post that depending on requests, Host machines and containers get initiated. Once execution is over, resources will be made available for further execution.

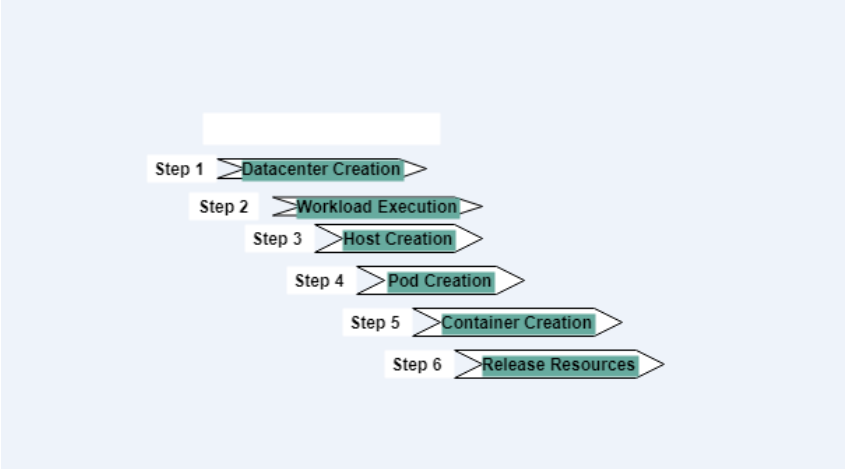


Figure 6: **Simulation**

4.3.2 Real Implementation on Google Cloud

Here the process talks about real implementation of this proposal on Google Cloud. For this, different monitoring tools are available in the market. Tools such as Grafana, Prometheus can be used and after that Google cloud container cluster get implemented by uploading container docker image. Ultimately, the application gets presented to the web.

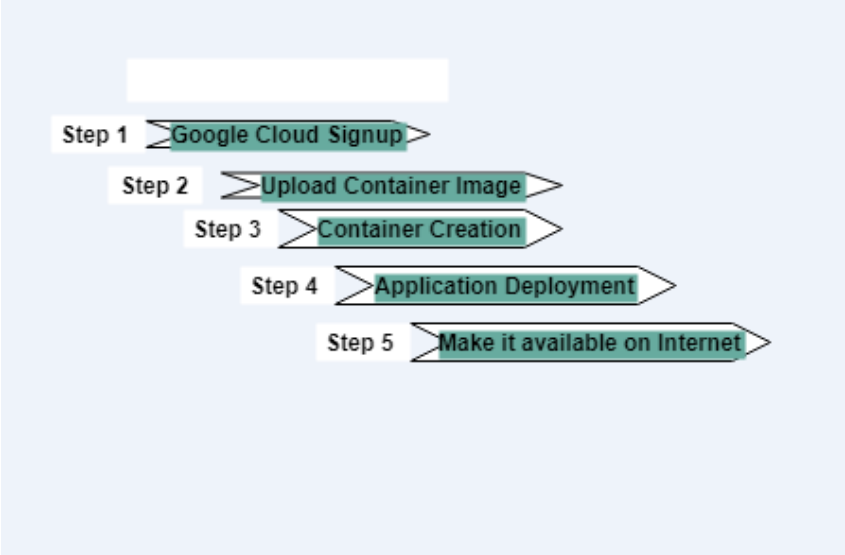


Figure 7: **Implementation on Google Cloud steps**

5 Code Implementation

To incorporate the Kubernetes Scheduler, this section explores the model's implementation framework in depth where it retains a different class for each component.

5.1 Development Structure

Using python, Kubernetes model has been simulated. This can be seen in figure 8 where aforementioned algorithms for simulation are worked upon through developed classes and methods.

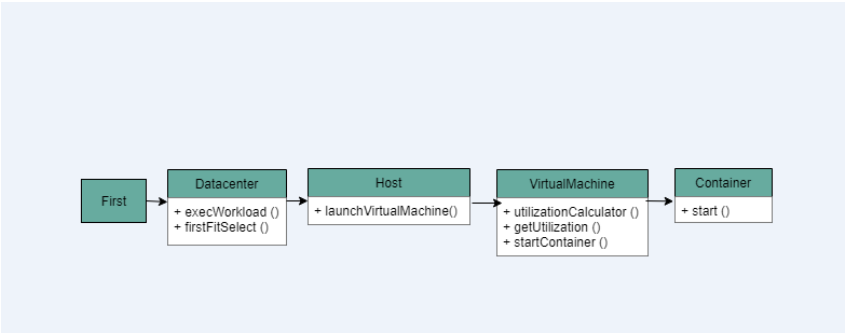


Figure 8: **Development Structure**

As indicated above, separate class is maintained for creation of datacentre, Virtual Machine, Container and Host. First class will be called initially which makes call to datacentre class where datacentre initialization methods are defined. Further for execution of workload, user has given choice to execute an application by using default kube-scheduler or using proposed scheduler. After that PrintUtilization function prints all utilization outcomes into output file. Next to that it applies check of over-utilization for all resource types, where it checks for 30 limits. If it crosses 30 then new pod or here

VM will get launch. Here for now 4 resources are targeted so that, maximum container capacity in a host will be 4 only. Each request looks for best-fit host.

6 Evaluation

This part of the study analyses and compares results of original and new Kubernetes scheduler to carryout the main aim of the research. To increase the validation and sureness of the implemented solution, examination flow was repeated for around five times. The first section of the study evaluates the utilization of resources of default scheduler implementation for evaluating the experimental results. Similarly, in second part, implemented scheduler outcome is determined.

6.1 Output of Default Scheduler Implementation

For validating the implementation, results of both the original system results and implemented solution results are to be compared. Below is the result of analytical analysis for all the targeted resources.

(a) RAM Utilization (b) Storage Utilization

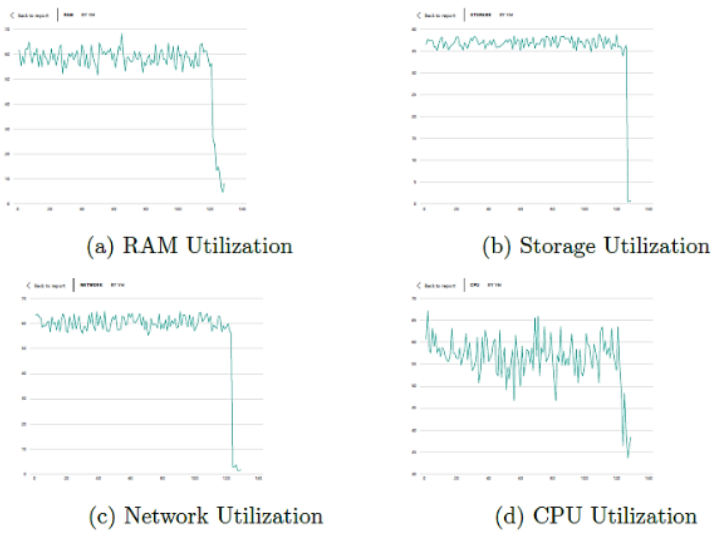
(c) Network Utilization (d) CPU Utilization

Figure 9: **Output with Default Scheduler**

It is clear that in 9a, utilization of the RAM goes above 90 in nearly all cases. However it is also plotted that is exceeding up to 70 in almost all cases. However, good amount of variation is seen here that is 30-90. In the next 9b figure, utilization of Storage is varying from 0 to 70. Also in some cases it has even crossed 70. However in 9c, even though utilization is not varying, it lies around. It has also been observed in 9c network utilization consistently lies near 60. But at some positions there are sudden falls. Again, in same pattern, there are fluctuations for storage type. It is roughly varying between 30 and 90. Overall, results are not only fluctuating but are also seen to produce non-linear results. Interestingly, few plots are over 70 and a note should be made on this occurrence. It is safe to conclude that due to uncertain results, resource contention seems to take place highly.

6.2 Output with Proposed Scheduler

To understand the success of this research, it is necessary to compare results with existing and with new implementation and on same workload. Experimental results of implemented solution are illustrated in Figure 10



(a) RAM Utilization (b) Storage Utilization

(c) Network Utilization (d) CPU Utilization

Figure 10: **Output with Proposed Scheduler**

This execution is also done with same dataset and for same resource set as earlier. Here it has been clearly visible that these utilization's are not that varying. These are fluctuating around certain point but there are no spikes or downs. As per 10.a. figure RAM utilization is in the range of 50-65. This is unlike what we did with default implementation results. Consequently, all observations are mapped near 60. Unlike default implementations, observations are not differing. Hence storage utilization, as illustrated in 10b, is within 35-38 range. But in existing scheduler, the fluctuation of results is seen in the range of 65-70s. In figure 10.c, results lies in the same range of 60. But for this resource type drop rate has been improved. One can observe that there is certain improvement as per Figure 10d's fluctuations for utilization of CPU. The observed range improved is from 45-65 to 25-95.

6.3 Statistical Analysis

For conducting statistical analysis, normality test is to be executed along with a histogram method. Normality test is performed on summarized data. Data is not normal and this can be observed through the test results. After performing Statistical Analysis Test, as the data is not normal, Mann-whitneyWijnand and van de Velde (2000) test is selected. This is one tail non-programmatic test. The outcome for all the monitored resources is observed as p ¡ 0.05. Consequently, as the p value is less than 0.05 null hypothesis is rejected. This leads to the fact that an alternate hypothesis has be to accept indicating that there is substantial difference amongst the two groups with 95

```
data:  mydata$Oram and mydata$Pram
W = 10294, p-value = 0.04659
```

Figure 11:

6.4 Comparison between Original and Proposed results

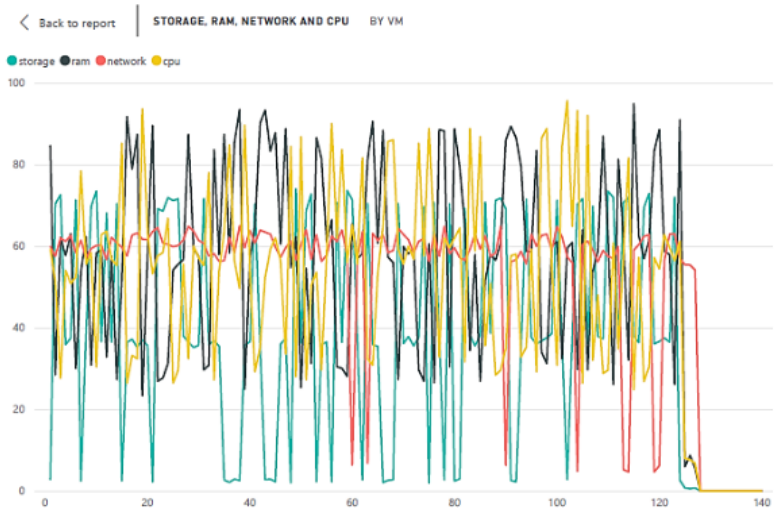


Figure 12: **Original**

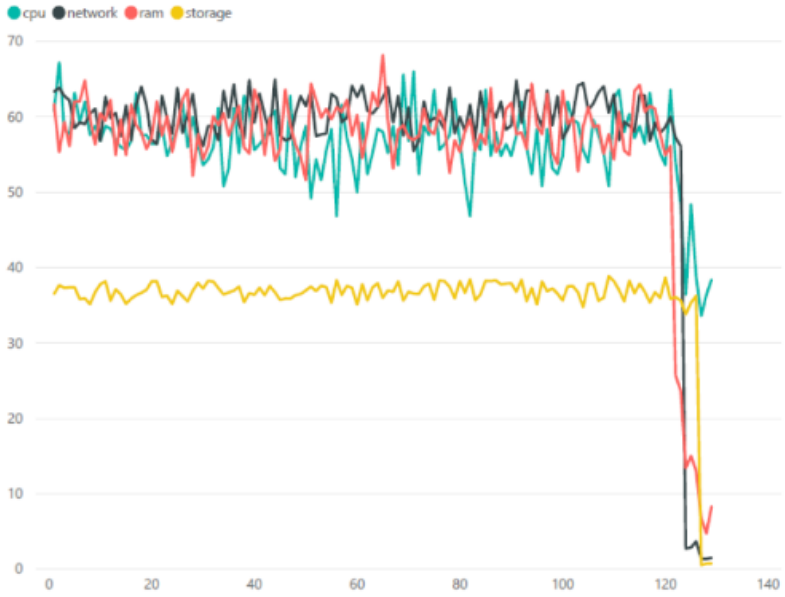


Figure 13: **Proposed**

Here figure 12 depicts the combined result of all monitored resources with original implementation and figure 13 depicts for implemented scheduler. It can be clearly seen that

results of all types of resource utilization is approximately varying in the range of 3- 95 whereas that is not the case with new scheduler. Proposed scheduler plots utilization around 40 and 60. This is good improvement and consistent results have been observed. Results are not varying as earlier. As per Figure 11 plotting's utilization is going above 60 at multiple points but that with proposed scheduler it resides in the range of 60. It varies by +5 and -5 which is quite good improvement for CPU, Network and Ram. Additionally, in same way storage results are also plotted near 40 with minimal variations.

This brings us to a conclusion that the proposed implementation outcome does improve resource management and moderates the prospects of resource contention to arise.

7 Conclusion

To sum up, on this proposed implementation, it can be concluded that, doing small improvement of application positioning helps to achieve performance improvement by lowering the contention problem and applying scheduling algorithms. A novel methodology is used which plans an application by arranging the application on the level of intensity it preserves during resource usage. There are fluctuations in the usage of resources in the default scheduler for the placement of container technique using round-robin even after the procedure of reserving resources such as memory and CPU which causes either over or underutilization of these resources. However, that is not the case with the custom scheduler implemented in this work. It can be seen in the results that it minimizes fluctuations at great extent. Even though this implementation takes care of resource utilization with reduced wastage it applies restriction to not go beyond 70 to maintain application performance. Application performance should not be constrained while handling resources. In comparison to the default scheduler, results vary a great extent, and also there is no control over maximum resource utilization without impacting application performance because there is no control over the level of maximum resource utilization.

8 Future Work

Current implementation works well in handling contention problems by managing the placement of application. Also, it lowers fluctuations and improves application performance. But still, it can be observed that resources are not fully- utilized. It is expected that live-migration can handle this as well. So that container live-migration can be studied as a part of future work for this research work.

References

Beltre, A., Saha, P. and Govindaraju, M. (2019). Kubesphere: An approach to multi-tenant fair scheduling for kubernetes clusters, *2019 IEEE Cloud Summit*, pp. 14–20. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9045748&isnumber=9045505.

Chang, C., Yang, S., Yeh, E., Lin, P. and Jeng, J. (2017). A kubernetes-based monitoring platform for dynamic cloud resource provisioning, *GLOBECOM 2017 - 2017*

IEEE Global Communications Conference, (Conference Rank : B), Singapore, Singapore, pp. 1–6. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8254046&isnumber=8253909.

Cloud, G. (n.d.). Kubernetes engine. reliable, efficient, and secured way to run kubernetes clusters. https://cloud.google.com/kubernetes-engine/,.

Google Cloud (n.d.). https://cloud.google.com/.

Hightower, K., Burns, B. and Beda, J. (2017). Kubernetes: Up and running: Dive into the future of infrastructure.

Jian Zhang and Figueiredo, R. J. (2006). Application classification through monitoring and learning of resource consumption patterns, *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*, pp. 10 pp.–. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1639378&isnumber=34366.

Kozhirbayev, Z. and Sinnott, R. O. (2017). A performance comparison of container-based technologies for the cloud, Vol. 68, p. 175–182.

Kube-Scheduler (n.d.). https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/.

Li, Z., O'Brien, L., Zhang, H. and Cai, R. (2012). A factor framework for experimental design for performance evaluation of commercial cloud services, *4th IEEE International Conference on Cloud Computing Technology and Science Proceedings*, pp. 169–176. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6427525&isnumber=6427477.

Mavridis, I. and Karatza, H. (2017). Performance and overhead study of containers running on top of virtual machines, *2017 IEEE 19th Conference on Business Informatics (CBI)*, Vol. 02, pp. 32–38. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8012937&isnumber=8012382.

McDaniel, S., Herbein, S. and Taufer, M. (2015). A two-tiered approach to i/o quality of service in docker containers, *2015 IEEE International Conference on Cluster Computing*, pp. 490–491. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7307624&isnumber=7307539.

Medel, V., Tolosana-Calasanz, R., Ángel Bañares, J., Arronategui, U. and Rana, O. F. (2018). Computers electrical engineering, *Characterising resource management performance in Kubernetes*, Vol. 68, pp. 286–297. http://www.sciencedirect.com/science/article/pii/S0045790617315240.

Medel, V., Tolón, C., Arronategui, U., Tolosana-Calasanz, R., Bañares, J. and Rana, O. (2017). Client-side scheduling based on application characterization on kubernetes, pp. 162–176. https://www.researchgate.net/publication/320248964_Client-Side_Scheduling_Based_on_Application_Characterization_on_Kubernetes/citations.

Pereira Ferreira, A. and Sinnott, R. (2019). A performance evaluation of containers running on managed kubernetes services, *2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 199–208. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8968907&isnumber=8968820.

Resource Quota (n.d.). https://kubernetes.io/docs/concepts/policy/resource-quotas/.

Rightscale (n.d.). Cloud computing trends: 2019 state of the cloud survey. https://www.rightscale.com/blog/cloud-industry-insights/cloud-computing-trends-2019-state-cloud-survey,.

Song, S., Deng, L., Gong, J. and Luo, H. (2018). Gaia scheduler: A kubernetes-based scheduler framework, *2018 IEEE Intl Conf on Parallel Distributed Processing with Applications, Ubiquitous Computing Communications, Big Data Cloud Computing, Social Computing Networking, Sustainable Computing Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom)*, pp. 252–259. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8672301&isnumber=8672218.

Wei-guo, Z., Xi-lin, M. and Jin-zhong, Z. (2018). Research on kubernetes' resource scheduling scheme, *Proceedings of the 8th International Conference on Communication and Network Security, (Conference Rank : B)*, ICCNS 2018, ACM, Qingdao, China, pp. 144–148. http://doi.acm.org/10.1145/3290480.3290507.

Wijnand, H. P. and van de Velde, R. (2000). Mann–whitney/wilcoxon's nonparametric cumulative probability distribution, *Computer Methods and Programs in Biomedicine* **63**(1): 21 – 28. "http://www.sciencedirect.com/science/article/pii/S0169260700000584".

Wu, Q., Yu, J., Lu, L., Qian, S. and Xue, G. (2019). Dynamically adjusting scale of a kubernetes cluster under qos guarantee, *2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS)*, pp. 193–200. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8975761&isnumber=8975714.

Xing, S., Qian, S., Cheng, B., Cao, J., Xue, G., Yu, J., Zhu, Y. and Li, M. (2019). A qos-oriented scheduling and autoscaling framework for deep learning, *2019 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8852319&isnumber=8851681.

Zhang, M., Ren, H. and Xia, C. (2017). A dynamic placement policy of virtual machine based on moga in cloud environment, *2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC), (Conference Rank :B)*, uangzhou, China, pp. 885–891. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8367365&isnumber=8366890.