Effective Memory Utilization using Custom Scheduler in Kubernetes.

MSc Research Project
MSc in Cloud Computing

Manisha Chandra
Student ID: X21117284

School of Computing
National College of Ireland

Supervisor: Dr. Aqeel Kazmi

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Manisha Chandra
Student ID:	X21117284
Programme:	MSc in Cloud Computing
Year:	2023
Module:	MSc Research Project
Supervisor:	Dr. Aqeel Kazmi
Submission Due Date:	01/02/2023
Project Title:	Effective Memory Utilization using Custom Scheduler in Kubernetes.
Word Count:	8000
Page Count:	25

Signature:	C Manisha
Date:	1st February 2023

Effective Memory Utilization using Custom Scheduler in Kubernetes.

Manisha Chandra
X21117284

Abstract

Kubernetes had quickly emerged as a popular option for containerized orchestrating workloads on a massive scale. Kubernetes takes advantage of the scheduler which takes into consideration constraints that are defined by the work-load owners as well as the cluster managers in order to identify which node will be the most suitable to host a certain task. In spite of the fact that it may be configured in a wide variety of ways, the default scheduler that comes with Kubernetes is not able to fully fulfill the specifications of revolutionary new applications. Because of this, a number of distinct proposals for custom Kubernetes schedulers have emerged, each of which focuses on meeting the expectations of the applications. As a result of this research, a new custom scheduler has been proposed that satisfies the requirements of the application with regard to effective resource scheduling while taking storage metrics derived from the prometheus tool into consideration, thus taking into account the requirements of both the user and the application. The findings are compared with the default scheduler, which takes into account only the CPU and RAM needs that are supplied by the user, in order to determine the quality of the proposed scheduler.

Keywords: Kubernetes, Custom Scheduler, Storage, Containerization, Cloud.

Contents

1 Introduction **3**
1.1 Research Question . . . 4
1.2 Objective . . . 4
1.3 Paper Structure . . . 4

2 Literature Review **5**
2.1 Kubernetes Survey . . . 5
2.2 Scheduling in Kubernetes . . . 6
2.3 Custom Based Scheduling . . . 7
2.4 Need for the proposed Custom Scheduler . . . 8
2.5 Summary . . . 9

3 Methodology **10**
3.1 Components for the proposed design . . . 11
3.2 Purpose of the custom scheduler . . . 12
3.3 Functioning of the proposed custom scheduler . . . 12

4 Design Specification **13**
4.1 Proposed Kubernetes Scheduler-Architecture . . . 14

5 Implementation **15**
5.1 Kubernetes Cluster Formation . . . 15
5.2 Proposed scheduling algorithm working . . . 16
5.3 Node Exporter and Monitoring tool . . . 17
5.3.1 Monitoring tool-Prometheus tool . . . 17

6 Evaluation **19**
6.1 Setting up multi-node cluster . . . 19
6.2 Configuring Prometheus to Work with Node Exporter on a Kubernetes Cluster . . . 20
6.3 Establishing a Connection With Prometheus . . . 20
6.4 Conducting tests with the Custom Scheduler . . . 20
6.5 Results & Discussion . . . 22

7 Conclusion and Future Work **23**

1 Introduction

Many businesses have embraced virtualization technology to make their Information Technology infrastructure more efficient and less expensive. Infrastructural resources can therefore be rationally allocated and distributed among many applications. As in past, virtual machines were used to do this (VMs). An application can run on a VM. In which an application must be bundled into a single file with all of its dependencies and the operating system that it uses.
However, because the operating system was included in the VMs, this resulted in larger Virtual machine sizes and slower startup times. An alternate lightweight virtualization method utilizing Linux containers can indeed be utilized to overcome these problems. Applications and their dependencies may indeed be packaged together into single deployable units using containers, while the OS kernel can always be accessed by several containers. Containers may be substantially smaller as well as deployed more rapidly as a result Bernstein (2014). An open-sourced project called Docker [1] offers a Linux container implementation.
Organizations are relying on the usage of docker container frameworks to enable the continuous deployment and administration of these containers at a large scale. Prominent and well-known across the IT industry orchestration frameworks include Apache Mesos [2], Docker Swarm , and Kubernetes.
The University of California of Berkeley's Apache Mesos program aims to manage to compute clusters by decoupling their capabilities from the hardware that hosts them. Multiple scheduling systems are used by Mesos, wherein the scheduling algorithm is left to the underlying client's framework (including such Hadoop) while Mesos is still in charge of allocating resources among such client frameworks Rodriguez and Buyya (2019). A job is the fundamental scheduling unit in the context of Mesos, and it can either relate to a unified command or even a container that will be carried out by a Mesos agent. Multiple servers running Docker applications may be combined together into clusters using Docker Swarm Environment.

The term "containerization" refers to a current type of virtualization that makes use of a single operating system kernel to power numerous distributed applications, each of which is built and executed inside its own container. The evolution of containerization is shown in Figure 1

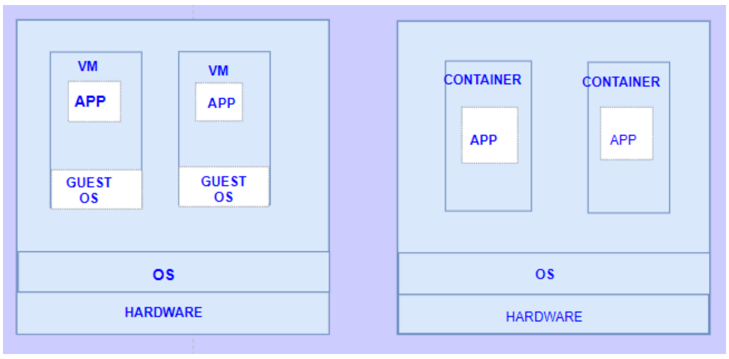


Figure 1: Evolution of Containerization

[1]Docker https://docs.docker.com/get-started/02_our_app/
[2]Mesos https://mesos.apache.org/

It's easy for people who already use Docker to manage because it's built into the Docker Engine Command-Line Interface (CLI). Scaling, load-balancing, as well as rolling updates, are also supported. Google first released Kubernetes available in 2014, and the Cloud Services Foundation (CNCF) oversaw all of its following improvements. In addition to featuring an extensible design, it is delivered with a robust feature set that includes self-healing, horizontal scalability, automatic roll-outs, and roll-backs. A system classification for task scheduling in the above-mentioned frameworks is provided in Rodriguez and Buyya (2019).

1.1 Research Question

Container orchestrators rely heavily on a feature called scheduling. Container allocation to computing nodes inside a cluster at a given time is controlled by a process called scheduling. Effective resource scheduling is driven by storage optimization.

- "How well does the proposed resource scheduler deal with pod allocation to respective nodes with effective storage allocation in light of user and application requirements?"

1.2 Objective

The main goal of this research is to build a custom scheduler in Kubernetes that will use the actual storage metrics of the nodes in the cluster to schedule a pod's resources in the best way possible.

- The suggested scheduler bases its decisions on the amount of memory that is accessible on nodes in order to address the issue of resource scheduling in situations in which the actual consumption is significantly different from the usage that was requested.
- To take advantage of the scalability and faster deployment offered by the cloud services, the Kubernetes cluster is formed using AWS EC2.
- Using the idea of running various schedulers in the same cluster, the study demonstrates the performance of the custom and the proposed scheduler.

1.3 Paper Structure

The paper is organized as follows: in the 1 segment, we'll learn about containerization and virtual machines, and in the second, we'll discuss Kubernetes custom schedule. The research's key objectives are listed. Section 2 describes the default scheduler, the many custom schedulers offered, and the origin of the proposed scheduling system. In section 3, we'll discuss the study's methods and equipment. section 4 details the custom scheduler's design and development. Please see section 6 for evaluation metrics comparing the proposed method to the default scheduler. we will see how the optimized memory consumption will improve resource scheduling in the long run. Here, in section 7, we wrap up the results and comparisons and offer suggestions for future improvements to the Kubernetes scheduler community.

2 Literature Review

In the process of studying various research papers, I came to the conclusion that Apache Mesos cant is not the best option to be considered because of its unique design, which means that some criteria don't apply to it. Despite some commonalities between the scheduling algorithms in Kubernetes as well as Docker Swarm Mode, In this paperRodriguez and Buyya (2019)shows that Kubernetes offers more flexibility in the scheduler's configuration choices.
In specifically, Kubernetes is the subject of this paper's attention among some of the major orchestration frameworks. According to the Clouds Native Survey, 2020 Rejiba and Chamanara (2022), Kubernetes adoption rates really continue to rise. More precisely, the poll reveals that, compared to 2019, 83% of production settings now use Kubernetes, up from 78% in 2019. According to the Sysdig 2022 Cloud Native Security as well as Usage Report [3], 96% of Sysdig's clients (representing a range of organization sizes and sectors) utilize Kubernetes for orchestration. Additional management Kubernetes services are presently offered by major cloud providers, including Amazon Elastic Kubernetes Service, Google Container Kubernetes Engine, Azure Kubernetes Service, and Alibaba Cloud containerization Service for Kubernetes. Last but not least, a performance analysis carried out in Al Jawarneh et al. (2019)shows that Kubernetes beats Mesos as well as Docker Swarm in complicated application situations.

2.1 Kubernetes Survey

Kubernetes is an open source platform service that is adaptable and portable. Its primary function is service & workload management, for which it provides declarative configuration and automatization support. Due to the wide availability of Kubernetes's supporting tools and services, its ecosystem is large and growing rapidly Burns and Beda (2019).
Historically, most businesses have run their app infrastructure on real hardware servers.

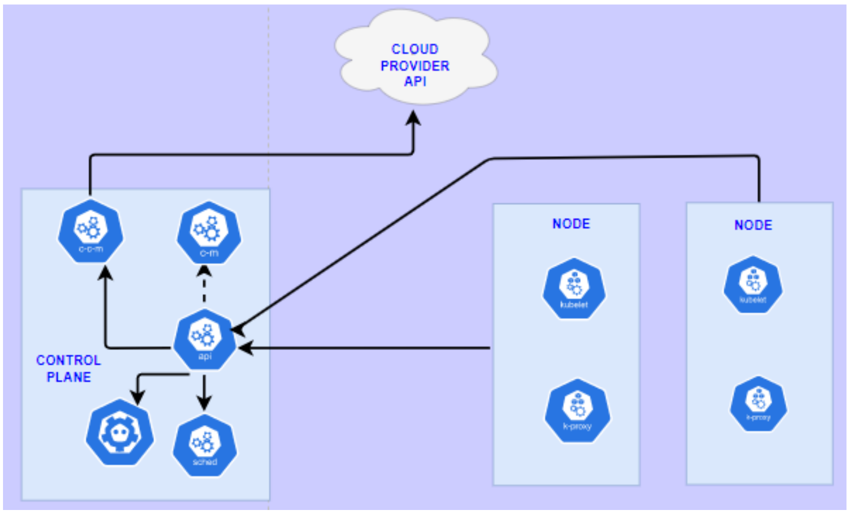


Figure 2: Kubernetes cluster overview

As there was no way to restrict the number of resources that individual apps may use

[3]Sysdig 2022 `https://library.cyentia.com/report/report_010905.html`

inside a given server, this caused allocation issues. The risk always exists, for example, that one program will use the bulk of a server's resources if many applications are executed on the same physical server.
Due to this, the functionality of such programs will degrade. To solve this problem, you may use separate servers for each application. But expansion was impossible because the underutilization of assets made it too expensive for businesses to install the necessary infrastructure. That's when the concept of a "virtual machine" came into play. The success of virtualized hardware has led to the development of smaller, more portable systems known as containers.Cai et al. (2021). Light coupling, efficient resource utilization, and other benefits have contributed to their rise in popularity.

2.2 Scheduling in Kubernetes

Kubernetes (also known as K8s) is "an open-source platform for automating deployment, scaling, and administration of containerized applications."kubernetes. As the unit of deployment, Kubernetes uses pods to encapsulate applications. Pods are the smallest and most fundamental workload objects in Kubernetes. In a pod, one or more containers may run an application, depending on how closely they are connected.
In most cases, the pod requirements as well as the metadata are stated in a YAML2 file. These criteria may include the container image that is going to be used for the creation of the pod, and the needed amount of resources (CPU or RAM) that each container in the pod requires, in addition to the destination port, whereas the metadata might include labels that characterize the pod. After that, the owner of the workload sends the pod (YAML file) to a Kubernetes cluster, where many components collaborate to bring the workload online on one of the cluster nodes.
There are two different kinds of nodes in Kubernetes:

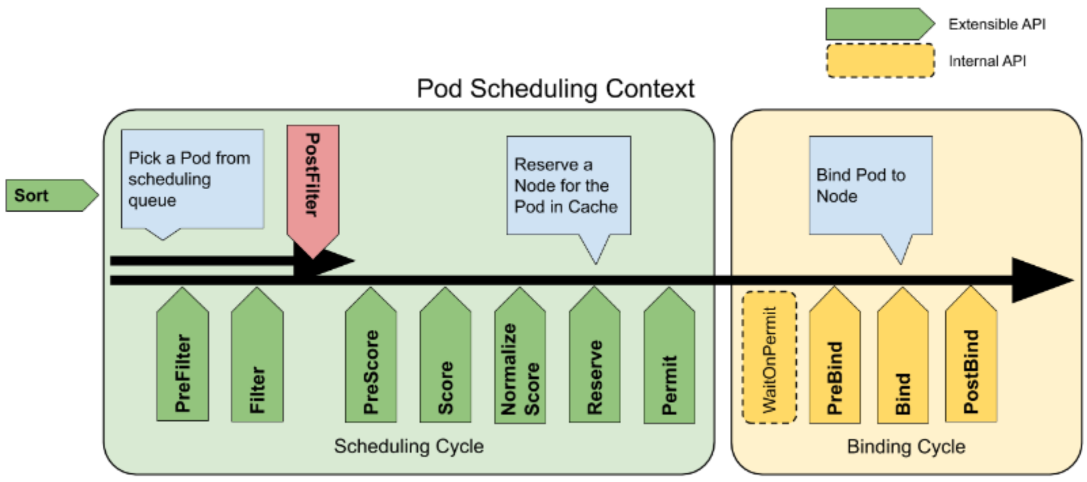


Figure 3: Scheduling in KubernetesScheduling Framework

- nodes that are responsible for the operation of the control plane components, also known as master nodes before.
- worker nodes are machines that are tasked with the operation of pods that hold the workloads of users. They are made up of the kubelet, the kube-proxy, and the container's runtime all rolled into one.

The working of the scheduler will be explained in this section. The scheduler will first filter the available nodes and then score them in order to determine which one will be the most suitable for hosting a pod.

- The procedures that are employed in the filtering stage, which were formerly known as predicates, make up a set of stringent restrictions that a node must satisfy in order to be permitted to execute the pod.
- To be more specific, for a particular pod, an evaluation is performed on each node within the cluster by comparing it to a collection of filters. In the event that a filter is not satisfied, the node in question is removed from the possible node selection for that pod, and the execution of any further filters is skipped.
- Following the completion of the filtering process, the set of viable nodes is sent on to the scoring phase. When scoring, several factors might be taken into consideration. When image proximity scoring is enabled, for example, nodes that already contain the required container images locally get a higher score than nodes that do not have the images locally. After that, the value of each node is assigned a score by using a rank combination of the various score criteria.

After this round of competition, the node that has accumulated the greatest score would be chosen to host a pod. When there are other nodes of this kind, links are severed in a random fashion. In its last step, the binding procedure communicates with the API server about the node that was chosen for the pod. The dashed region provides a visual representation of the whole procedure.

2.3 Custom Based Scheduling

Kubernetes was initially launched in 2014, however, it wasn't until 2017 that the first contributions with custom scheduling methods were suggested. Work tasks that compete for the exact same sort of resource get a high danger of conflicting risk with one another, and this is a problem that is addressed in a variety of ways in the study devoted to this area. As an example, in Medel et al. (2017), The authors advocate for programmers to provide labels to apps based on their predicted resource use. High CPU, Down CPU, Large Disk I/O, and Slow Disk I/O are only a few examples of program labels. The scheduler may account for all these labels by assigning a penalty to each node based on the tags of the apps currently executing on it. The assessment findings derived from a real cluster demonstrate that the suggested scheduler effectively separates apps with heavy use of a particular resource.
The issue of intervention between concurrent ML processes is addressed in detail within work Bao et al. (2019). In particular, they think about worker parameter server (PS)-based distributed ML workloads. In opposition to preexisting systems that depend on specific workload profiles as well as interference models, the authors proposed Harmony as a generalist scheduling approach that relies on deep relevance feedback. The required number of employees and PSs, together with their respective resource needs, the number of resources available, and the positioning matrix for concurrent tasks all go into scheduling choices. To gauge the efficacy of the suggested method, a GPU-based testbed was established. According to the findings, Harmony is faster at finishing jobs than the evaluated baselines.

Several initiatives have surfaced to do is provide assistance for guaranteed bandwidth in the Kubernetes scheduler, which is particularly useful for workloads that entail the transmission of huge volumes of data across the network. For instance, a "network bandwidth management system" (NBWguard) is proposed in Xu et al. (2018). The authors argue that Kubernetes would benefit by recognizing networking as a resource, much like CPUs and memory. In this way, users may set QoS restrictions and demands based on the amount of network bandwidth they have access to. Through the use of Linux os based network administration tools, NBWguard is able to limit traffic.
The majority of the articles discussed herein focus on fog/edge computing setups in which nodes of a cluster are dispersed over many physical locations. Therefore, it is important to consider the architecture of nodes in order to reduce unnecessary transmission costs. A multi-container pod might not always run effectively on a fog node with insufficient resources, as discussed in Kayal (2020). Because of this, they suggest decomposing such pods onto their component containers and distributing them over numerous fog nodes thus accounting for the expenses associated with cross-communication. The authors offer a method to achieve this by ranking pod queue items based on how much interaction there is between containers. In the event that no suitable node is located, the pods will indeed be partitioned such that certain nodes may host other containers inside it. Next, the proximity between the various fog nodes is taken into consideration as part of a topology-aware network scoring. The authors suggest adding these various procedures as scheduling plugins. On the other hand, no details about the actual implementation were given.

2.4 Need for the proposed Custom Scheduler

At the heart of this section are proposals for custom schedulers which account for the real load of the nodes in the cluster (i.e. their actual utilization level) while allocating resources. Example contributions include [Chima Ogbuachi et al. (2020), Ogbuachi et al. (2019)], where The authors take into account an edge and fog computing environment and stress the need for scheduling based on real node resource use. Node values are determined using data gathered from real-time observation of devices, including load, temperatures, and safeness. Scheduling times were reduced when compared to the default scheduler, and nodes ran at comparable temperatures, demonstrating that the suggested scheduler doesn't really jeopardize node health.
In Medel et al. (2017), The authors suggest that app makers mark their software with labels indicating how much space and CPU time it will need. Utilizing the tags of the apps currently executing on a node, the scheduling is able to determine a penalty for employing that node. The assessment findings based on a real cluster demonstrate that the suggested scheduler effectively separates apps that use a lot of a particular resource.
The authors in Li et al. (2019) propose a tweaked version of the particle swarm optimization method to make the scheduling calls. The method considers the memory and CPU utilization of nodes, the peculiarities of the payloads' consumption, and any quantities that could be refusing to allow towards particular nodes. The suggested technique improved the nodes' resource use by 20% when compared to the standard Kubernetes scheduler.
authors of Li et al. (2020) pay particular attention to the need of knowing the actual I/O load at any given time. In particular, Prometheus Rejiba and Chamanara (2022) is used to gather information on the system's I/O and CPU utilization. A scheduler expander

technique makes use of these measurements to allow for individualized behavior based on a given score. In fact, we offer two new scoring functions—BalancedDiskIOPriority(BDI) and BalancedCpuDiskIOPriority(BCDP) (BCDI). The findings of the assessment show that the cluster's disk I/O is better balanced when BCDI is used, while CPU use is also more evenly distributed.
We also highlight the IBM huge pile scheduler plugins, which are known as Trimaran 2021-08-0 (2021), in addition to the above-mentioned scholarly achievements. The intention behind these plugins was to raise the cluster's resource consumption by alerting the scheduler to any gaps between the allocated and used resources. Specifically, TargetLoadPacking as well as LoadVariationRiskBalancing are the two rating plugins that makeup Trimaran. TargetLoadPacking assigns points to nodes depending on their actual resource consumption, with the goal of keeping all nodes at a constant resource utilization level. LoadVariationRiskBalancing assigns points to nodes depending on how consistently they use resources relative to the mean. Trimaran's default metric source [4] is Kubernetes Metric- server, however, alternative options like Prometheus as well as SignalFx are available. Intel further suggests a telemetry aware scheduling (TAS) [5]to facilitate scheduling in light of timely telemetry data. The scheduler makes use of the extension mechanism, considering factors like the node's energy consumption, free RAM, as well as CPU temperature. In the project proposal, we will take into account not only the CPU and real memory utilization but also the other ancillary metrics, by scraping the necessary data with the Prometheus tool[6], to ensure that nodes are scheduled to pods in the most effective way possible. .

2.5 Summary

In this analysis, I identified the key improvements which have been done in the scheduling area, by also emphasizing how the major drivers towards Kubernetes custom scheduling have evolved over time. This was done alongside Kubernetes driver changes. A method for categorizing the examined works based on their top objectives, target surroundings, and workloads, a customized scheduling operation, and assessment and implementation methods was discussed. We described their distinctive scheduling contribution according to their goals and analyzed the trends for each target.

Our poll suggests that customizing a scheduler may not be enough depending on the goal's difficulty. Actually, specialized resources may be needed to support anticipated scheduling behavior. Predictive scheduling may demand a finer-grained and smaller dataset. Kubernetes is constantly improving the system, notably the scheduler. This indicates that issues that are now being solved by custom schedules may, in future schedulers, by default available to be used in scheduling operations. It also implies that it is essential to keep a close eye on the most recent developments that have been done to a scheduler in to ensure that any adaptations are made in accordance with the most up-to-date standards of practice. In Table 2.5 gives us a brief summary of various proposed custom scheduler the environment they were targeted to and the limitations.

[4]Metric-server `https://github.com/kubernetes-sigs/metrics-server`

[5]Trimaran `https://github.com/kubernetes-sigs/scheduler-plugins/blob/master/kep/61-Trimaran-real-load-aware-scheduling/`

[6]Prometheus `https://prometheus.io/docs/introduction/design-doc/`

Overview of Kubernetes Custom-Schedulers			
References	workloads target	Main-Idea	Limitations
Ghag et al. (n.d.)	Heterogeneous	The volume of traffic that is created by containers	network latency.
Ungureanu et al. (2019)	cloud scheduling	Resource management Job objectives Runtime predicted task	with different algorithms are being involved, it varies the result.
Santos et al. (2019)	Fog	Time required to go full circle to a specific location Bandwidth	Limited to the certain network related applications
Kaur et al. (2019)	edge	An Interference from Human-Caused Carbon Emissions Use of resources	Problems with many dimensions and complicated structures are the only ones for which this method is appropriate.
Townend et al. (2019)	cloud	An exact simulation of the hardware's internals Processor speed Behavior-modeling workload metrics	not much effective for resource scheduling
Li et al. (2020)	Cloud	Node's disk I/O stress Utilization of the Node's Central Processing Unit	Similar to the default scheduler in kubernetes.
Beltre et al. (2019)	Cloud	possession of the greater part of the resources Time spent waitingProject resource requirements	a lot of downtime and a focus on CPU rather than I/O operations..

3 Methodology

Three primary duties must be completed by the Kubernetes scheduler: In the beginning, keep an eye out for any unplanned pods. Second, identify the "best" node for every unscheduled pod. The term "best" refers to a suitable node with one of the most RAM available in the scope of our unique scheduler. Bind that pod to the chosen nodes third. The second objective, node selection, is everything we've tailored for our needs in order to prefer the node with the most RAM available. These include the subsequent actions:

- Locate all nodes that 'fit' a pod, such that, which meet the memory and CPU demands of the pod.
- For each suitable node, extract the pertinent metric data (memory).
- Choose the nodes only with the best (highest) metric value.

The pod life cycle is shown in the Figure 4

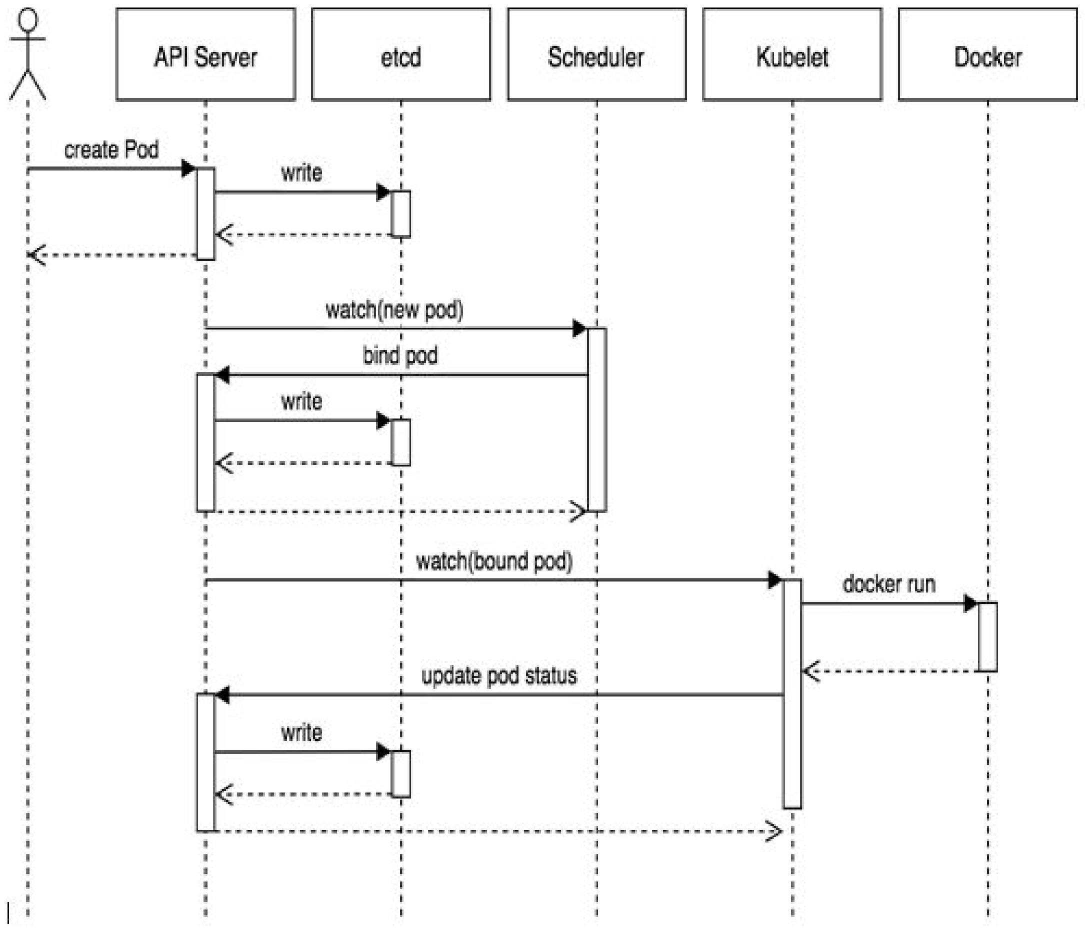


Figure 4: Kubernetes Pod creation-state diagram

3.1 Components for the proposed design

API server, Controller, and the Kubernetes standard scheduler are the components that makeup the master. In order to install Kubernetes-master and slave, we need to allot specialized vms that has the capacity to access IP addressesChang et al. (2017).

- **etcd**: a storage component that is utilized to record the current status of the system, which enables its other master elements to sync them to the intended state by observing etc;
- **Scheduler**: The scheduler is in charge of allocating time slots for every pod that resides on a certain node of the system;
- **API Server**: is in charge of accepting instructions and altering the information for Kubernetes components (like pods) within the system in a manner that is appropriate for user commands.
- **Kubelet** Kubelet is indeed the node interface that is deployed on each worker node. It is responsible for communication between the nodes. It is responsible

for monitoring the POD requirement via all the slaves and masters in the system. Kubelet provides visibility into resource use, the state of pods, and events occurring on nodes. It makes the data on specific port 10255 available to the public.

- **namespaces** Process seclusion is a crucial consideration during the deployment of a service using Kubernetes namespaces. Within a Kubernetes cluster, it's the role of the namespaces to compartmentalize the pods/containers. Pods that share the same namespace are able to interact with one another.

- **Monitoring Module**: we are using **Prometheus** and **Kubenetes dashboard** for scraping in the data and knowing the status of the nodes and pods respectively. More information will be provided in the subsequent sections.

3.2 Purpose of the custom scheduler

The purpose of the proposed custom scheduler that was developed is to maximize the use of Memory across all of the nodes. Although the normal scheduler takes into account CPU as well as memory requests—which are not required to be provided in the yaml files that describe pods and deployments of pods—the proposed scheduler is more concerned with the amount of memory that is being used by the nodes.
It is conceivable for pods to utilize significantly more storage than what is asked, significantly less storage than what is requested, or otherwise not provide a storage request at all. Because it's possible that the pods won't utilize the predicted amount of storage after they've been scheduled, concentrating on storage requests could not be the best way to get an accurate picture of what storage will be utilized.
Scenario being considered by comparing default and custom scheduler behavior
In order to account for the possibility that the real consumption would be substantially different from the utilization that was requested, the custom scheduler takes choices depending on the amount of memory that is present in the nodes. For the purpose of determining which nodes provides the most memory accessible, we make use of a statistic known as node memory memAvailable.
If a node has been operating a pod that requested an amount of storage but was instead actually using some of the memory, and then another node has been operating a pod that didn't get a memory proposal but was certainly using just a large amount of memory, then Kubernetes will most probably schedule the latest pod just on a node that is absolutely using a large amount of memory. This is because the default Kubernetes scheduling method prioritizes the pod that's already using the most memory. According to the queries, it seems to be the node that is the least loaded. Rather than having memory-intensive apps all run on a single node, our custom scheduler has the ability to distribute memory utilization among all of the nodes.

3.3 Functioning of the proposed custom scheduler

The scheduler begins by continuously monitoring the newly formed pods, which is the first and most crucial step in the process. On pods, we may utilize the Watch method provided by the Kubernetes client-go SDK. This method provides us with a stream of events over which we can search over.

A **Field Selector** is required in order for the Watch function to operate properly. That choice now has two components thanks to our additions. If **spec.nodeName** is present, it indicates that we're exclusively interested in a pod in which there is no nodeName parameter set. If they already have a nodeName assigned to them, this indicates that they're on the schedule.
It becomes more intriguing when we get to the spec.schedulerName=random-scheduler portion. Kubernetes is capable of running many schedulers inside a cluster at the same time, and a scheduler name may be included in a pod specification.
When going through all the events that are supplied by the watch function, it is also necessary to filter again for ADDED events. This is because we just want to be concerned with pods that have been recently added.
The following step is to locate a node that is suitable. In this really basic demonstration, we are going to choose a node at random first from the list of nodes. Notice that for each and every scheduled event, we are accessing the API server in order to get the list of the nodes.
After we have located a node of the pod, the single most significant thing that is still left to do is to inform the API server of the discovery.
Even if this is sufficient for scheduling the pod, we are going to add one further item. The scheduler produces a Kubernetes Schedule event. Because the standard scheduler already accomplishes this, we will be able to follow the occurrences even when we're testing out the custom scheduler.

4 Design Specification

The following resources were utilized to create and test the custom scheduler: I have decided to employ AWS cloud services in order to set up a Kubernetes cluster formation so that I may reap the benefits of cloud services like EC2.

For the purpose of this research experiment, I formed a cluster of Kubernetes by utilizing the Kubeadm-dind-cluster command (KDC) shell script. This included downloading both Kubernetes and Docker in order to construct the cluster. For the development of the master node, which will have 2 CPUs and the minimum storage need, we will be utilizing t2.medium, while the slave nodes will be produced using t2.micro, each of which will have only one CPU. GoLanguage was chosen to develop the scheduler because it has a short startup time, minimal runtime overhead, and the ability to function without the need for a virtual machine. It is by far the most common language used for putting microservices into action. We are using a node exporter, which has to be installed in the slave nodes, in order to collect the node metrics while installing the pod. After that, we will get the promotheseus in the master node and attempt to connect with the slave nodes. This will offer the necessary visual help so that the live values that are being created in the nodes can be checked. The application programming interface (API) of this tool is coupled to the filtering mechanism of the scheduler so that the compatible node may be deployed to the appropriate pod.

The table6.5 lists the many tools and technologies, along with the versions of each tool and technology.

Design Specification	
Tools and Technologies	Description/Version
Cluster Creation Platform	AWS EC2
Operating System	Ubuntu Server 20.04 LTS
Application Container	Ngnix
Containerization orchestrator Software	Kubernetes 1.20.0
Software for Containerization	Docker version 20.10.12
Monitoring tools	Prometheus and node exporter
Number of CPUs for Slave and Master	1 and 2 respectively
Storage	minimum 4GiB memory
Coding language used	GoLang 1.18
files used for communication between pods and nodes	YAML

The following technologies will each have their explanations and working descriptions provided in the subsequent section.

4.1 Proposed Kubernetes Scheduler-Architecture

The proposed custom Kubernetes scheduler is developed to place containerized apps to nodes in accordance with their real memory use rather than the amount of memory that was requested by the user. It is necessary to pick a suitable node that satisfies the pod's RAM and CPU requirements that has the greatest amount of free memory for each and every unscheduled-pod .

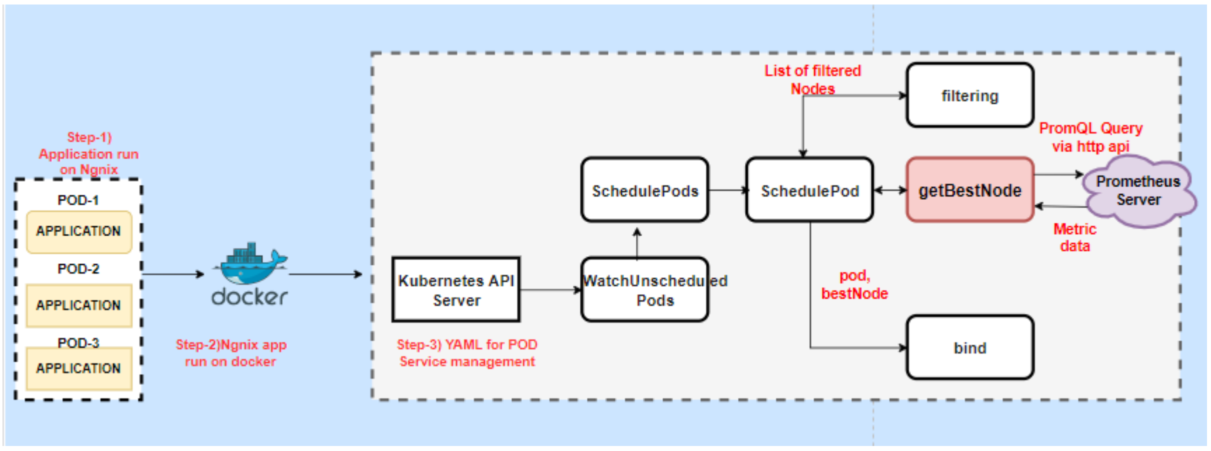


Figure 5: Internal architecture of custom scheduler working

Figure 5 shows the detailed architecture diagram of the custom scheduler and thereby explaining the steps which are involved in this process.
Step-1: During this stage of the procedure, the Kubernetes cluster will be created. In order to accomplish this goal, I made use of an EC2 instance. One worker node and two slave nodes make up the whole network. In addition to this, we require unique Yaml files

for each and every subsequent assessment.
Step-2: At this stage, it is necessary for us to download Docker, Kubernetes, and Kubeadm by employing a shell script that contains all of the necessary steps to successfully download the technologies. This action needs to be taken on each of the worker nodes as well as the master nodes.
Step-3: At the stage of **getBestNode**, we will need to make a prometheus server connection and call the respective node exposer in order to scrape the metric details and feed them to it in order to get the nodes so that we can schedule the pods to run on them.
Step-4 Now, when the custom scheduler is being executed. We will be able to get the best node by going through the process of filtering and binding methods that have been written in the scheduling process using **GO** language. This will allow us to acquire the best node. In order to compare the results of the suggested scheduler with those of the default scheduler, we need to first run the default scheduler.

5 Implementation

The concept Kelsey Hightower provided for creating a Kubernetes custom scheduler served as the basis for the solution. In his demonstration, Hightower shows how to create a prototype custom scheduler that creates schedules depending on some manually inserted randomized annotations in which every node is annotated along with cost randomly; in which the scheduler picks the one which has a low-cost- node. The modules for monitoring and identifying unscheduled pods, performing the default predicate verifications to discover nodes that meet the requested memory and CPU requirements of pods, and governing the pods to the chosen nodes are all based on parts of his code base that we used as the foundation for the scheduler.
On top of this foundation, we developed components that allow the rescheduling decision to take the nodes metric data from Prometheus into account. The HTTP API in Prometheus is used to make a PromQL query to obtain the node memory mem Available model parameters for the suitable nodes after the default predicate tests are completed to identify the list of suitable nodes that meet the pod's desired Memory and cpu requirements:
The node with the highest value would then be found and given to the pod. The main elements of a custom scheduler are shown in the following figure6.

5.1 Kubernetes Cluster Formation

This section will talk about how the kubernetes cluster has been implemented on AWS by using EC2.
Creating Nodes: The initial step is to create the master and slave nodes, for our process we have created one master and two slave nodes. As the master nodes do more computation work as it contains the API, Control Panel with the default, and the custom scheduler. I have chosen **T2.MEDIUM** as we require 2 CPUs and the operating system we have chosen **Ubuntu**.
For creating the slave nodes, two have been created with **T2.MICRO** which has only one CPU. We need to first create a shell script file in all the nodes which contains the commands to run. The shell script contains the commands to install **Docker** as we need a container solution to run the kubernetes. The next step will be to download kubernetes, and the last this is to get the kubeadm which is used to generate tokens in the master

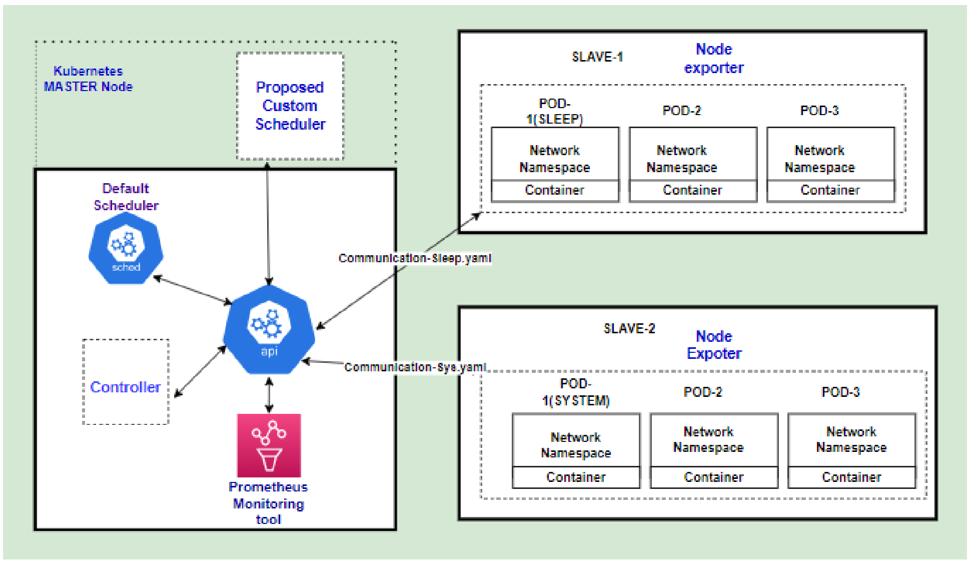


Figure 6: high-level architecture of the components present in the kubernetes cluster

node. This shell script is run by using the **Root** Privileges.
Then we need to run the following command which initializes kubeadm and generated a token in the master node which is then copied and run in the slave nodes in order to form the cluster. By installing Kubernetes nodes via Docker containers instead of virtual machines (VMs) or distinct bare-metal computers, you may quickly build a multi-node Virtual machine cluster on a single system with the help of the customizable script known as Kubeadm-dind-cluster, or KDC. It even makes it simple to construct many clusters within the same system as compared to other solutions.
deploying nodes via Docker containers to set up a multi-node cluster on such a single system.
sudo./dind-cluster-v1.13.sh up
chmod +x dind-cluster

5.2 Proposed scheduling algorithm working

In figure 8 we can see the overview of the working of the custom scheduler.
1.**monitorUnscheduledpods()**: This procedure is located in the main.go file on our scheduler code artifacts. This will establish a pod and designate a node for it after continuously monitoring for any requests that were made by the user.
2.**SchedulePods()**: The list of unscheduled pods will be obtained from the get method using this method, and then an attempt will be made to transmit each pod for the scheduling of the node process.
3.**SchedulePod()** The code portions for the fit() and bind() methods are included inside this method. The **fit()** method is composed of two functional components: first, it will take the input, which is a pod sent by the schedulePod() method, do the necessary filtering and choose the optimal set of nodes.
This list of nodes is then provided to the method known as **getBestNode()**, which is the implementation of how to get the metric data of the nodes from the prometheus server API. Based on the metric data, this method chooses the node that is considered to be

the best.
4.**Bind()** At long last, the pod that has been chosen by the schedulerPod() method and the best node that has been chosen by the getBestNode() method are chosen as inputs to this method, and the method then binds the pod to the best node.

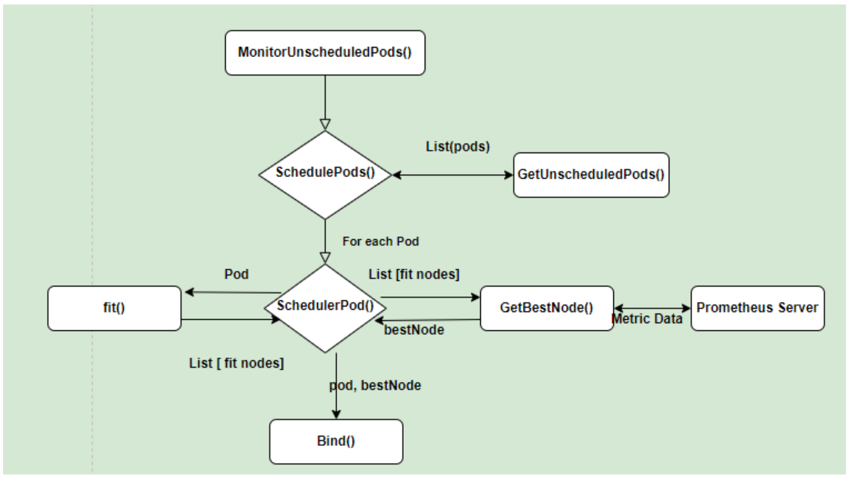


Figure 7: Working flow chart of the proposed custom scheduler

5.3 Node Exporter and Monitoring tool

A very scalable open-source monitoring platform is known as Prometheus. It gives the Kubernetes orchestration container platform built-in monitoring features. It is also becoming quite popular inside the observability arena because it aids in metrics as well as alerts.
Node Exporter This is a part of prometheus exporter which is used to get various metrics related to the nodes. In our research, we are using node exporter to get the memory available in the working nodes.

5.3.1 Monitoring tool-Prometheus tool

Metric Gathering: Prometheus retrieves metrics through HTTP using the pull approach. For use instances when Prometheus doesn't scrape the data, it is possible to send metrics to Prometheus using Push gateway. One such instance is gathering unique data from temporary Kubernetes tasks and Cronjobs.
Metric Endpoint: The platforms you want Prometheus to watch should disclose their metrics on to an endpoint called /metrics. This API is used by Prometheus to retrieve the statistics on a regular basis.
PromQL: The query language PromQL, which is included with Prometheus, can be utilized to search the data on the Prometheus dashboards. To begin, a Kubernetes namespace will be created for all of the monitoring components to reside under. All of the Prometheus-kubernetes deployments objects will be put into the default namespace if you do not build a dedicated namespace beforehand.

we need to run the script to set up a brand-new namespace that you can refer to as monitoring.

- kubectl create namespace monitoring

Kubernetes APIs are used by Prometheus in order to get all of the obtainable metrics through Nodes, Pods, Implementation, and so on.

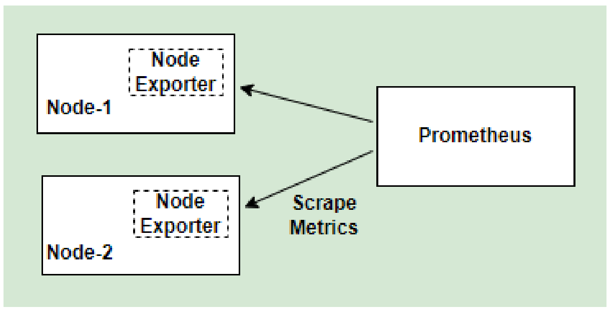


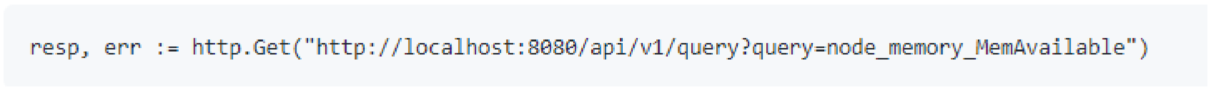
Figure 8: Monitoring tools used in the custom scheduler

created a new file with the name clusterRole.yaml,

One will see that I have added the get, list, as well as watch rights to the role that is provided below. These permissions apply to nodes, service endpoints, pods, as well as ingresses. The monitoring namespace serves as the location for the role binding. In the event that has a use case in which it is necessary to get metrics from another object, have need to be included in this cluster role. The Kubernetes nodes' Linux system-level statistics would be made available to the user using the Node Exporter service.
A portion of a Prometheus configuration map is devoted to the scraping config for the node-exporter. After we have deployed the node exporter, we ought to be able to see the targets as well as metrics associated with the node exporter within Prometheus.

- cd node-exporter/
- kubectl create -f node-exporter-daemonset.yml

We then constructed components to incorporate the nodes metric measured value by Prometheus into the scheduling choice we were making based on this foundation. After the standard predicate tests have been conducted to determine the list of suitable nodes that fulfill the pod's specified CPU and memory needs, a Prom QL query is made using Prometheus Http API to acquire the node memory mem Available model parameters for the suitable nodes, which are as follows: After that, the node that has the highest value



```
resp, err := http.Get("http://localhost:8080/api/v1/query?query=node_memory_MemAvailable")
```

Figure 9: Prometheus API query which is used to get the node metrics in order to determine the best node

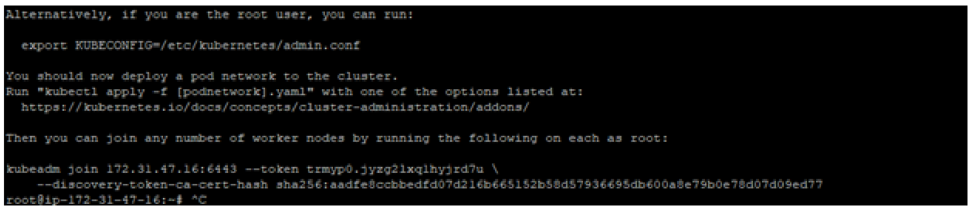
is located, which is the one that is allocated to the pod.

6 Evaluation

Experiment has been conducted by using AWS EC2 services and the server used to create the kubernetes cluster is **Ubuntu Server 20.04 LTS**. For evaluation, we install five pods in order to compare the performance of the custom scheduler to that of the default scheduler on Kubernetes. These pods are named as follows: Sleep (two replicas), Sys (1 replica) on both the default scheduler as well as the custom scheduler.

6.1 Setting up multi-node cluster

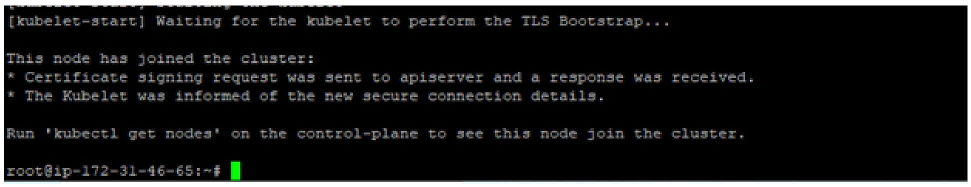
Kubeadmdind-cluster, often known as KDC, is a script that may be configured to meet a variety of needs. Its primary function is to simplify the process of installing several Kubernetes node on a single computer by using Docker containers.
This figure 10 shows the master node where the token is being generated in order to provide in the slave nodes to form kubeadm cluster



```
Alternatively, if you are the root user, you can run:

  export KUBECONFIG=/etc/kubernetes/admin.conf

You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
  https://kubernetes.io/docs/concepts/cluster-administration/addons/

Then you can join any number of worker nodes by running the following on each as root:

kubeadm join 172.31.47.16:6443 --token trmyp0.jyzg2lxqlhyjrd7u \
    --discovery-token-ca-cert-hash sha256:aadfe8ccbbedfd07d216b665152b58d57936695db600a8e79b0e78d07d09ed77
root@ip-172-31-47-16:~# ^C
```

Figure 10: kubeadm token in order to form cluster

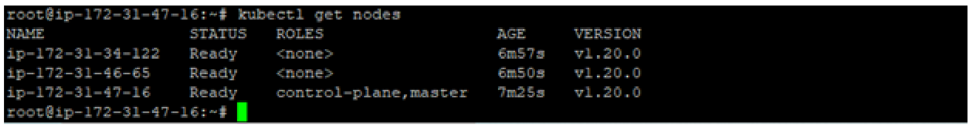
The token is generated in the master node and then provided to the slave node in order for it to join the cluster. In the 11 we can see that slave node has been added. Even before doing this, we need to run a few commands in the master node in order to form kubeadm cluster. Clear instructions are provided in the configuration manual. If the link expires for the creation of the token we need to use "**Kubectl token create**".



```
[kubelet-start] Waiting for the kubelet to perform the TLS Bootstrap...

This node has joined the cluster:
* Certificate signing request was sent to apiserver and a response was received.
* The Kubelet was informed of the new secure connection details.

Run 'kubectl get nodes' on the control-plane to see this node join the cluster.

root@ip-172-31-46-65:~#
```

Figure 11: After we run the token in Slave node

In order to check if the cluster is formed, we need to use the command **Kubectl get nodes**. This can be seen the in the below figure 12

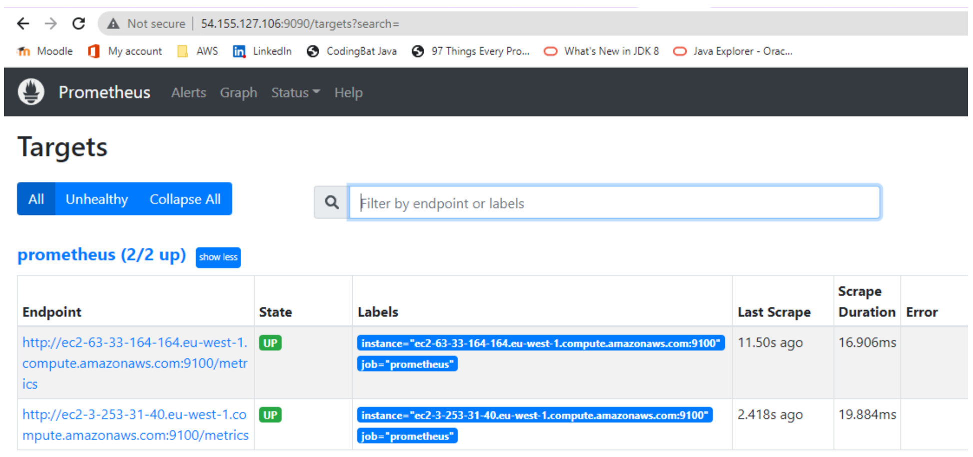


```
root@ip-172-31-47-16:~# kubectl get nodes
NAME               STATUS   ROLES                  AGE    VERSION
ip-172-31-34-122   Ready    <none>                 6m57s  v1.20.0
ip-172-31-46-65    Ready    <none>                 6m50s  v1.20.0
ip-172-31-47-16    Ready    control-plane,master   7m25s  v1.20.0
root@ip-172-31-47-16:~#
```

Figure 12: Cluster Formation in the master node

6.2 Configuring Prometheus to Work with Node Exporter on a Kubernetes Cluster

This is achieved by first Creating a namespace for monitoring using the kubectl command.
kubectl create namespace monitoring
Configure a DaemonSet on each of the nodes, then set up the node exporter:
In this step, we need to cd to the node expoter folder and use the following command
kubectl create -f node-exporter-daemonset.yml
The next step is to set up Prometheus, here we need to do monitoring by using yaml file
.



Targets

prometheus (2/2 up)

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://ec2-63-33-164-164.eu-west-1.compute.amazonaws.com:9100/metrics	UP	instance="ec2-63-33-164-164.eu-west-1.compute.amazonaws.com:9100" job="prometheus"	11.50s ago	16.906ms	
http://ec2-3-253-31-40.eu-west-1.compute.amazonaws.com:9100/metrics	UP	instance="ec2-3-253-31-40.eu-west-1.compute.amazonaws.com:9100" job="prometheus"	2.418s ago	19.884ms	

Figure 13: Prometheus dashboard showing slave nodes state

In figure 13, we can see the nodes which we created and connected with the node exporter. When we click on the endpoint of any of the slave nodes, we can check the node metrics regarding it.

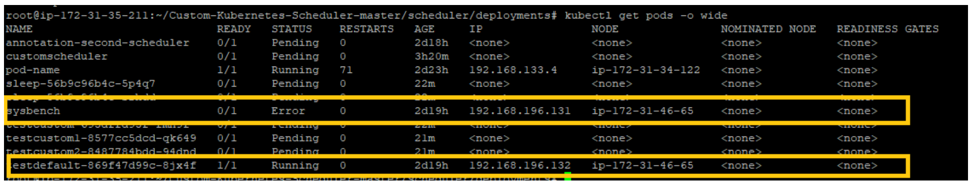
6.3 Establishing a Connection With Prometheus

In order to connect UI of prometheus, by using command **kubectl port-forward [your-prometheus-pod-name] 8080:9090 -n monitoring**. In the 9 we can see the kubernetes cluster with the following nodes.

6.4 Conducting tests with the Custom Scheduler

As it was said at the introduction of this section we will deploy 4 pods in order to test the performance of the proposed scheduler.
Step-1:
The sleep. yaml is just a pod that demands 1800Mi in memory as well as an unlimited capacity for sleeping. In all, I deployed two sleep pods and make a request for 3,600 megabytes of RAM. The default for the test. yaml triggers the execution of the Nginx application using the standard scheduler.
Step-2:
The workload for the sysbench memory tasks is executed by using the sysbench.yaml file. The benchmark program begins by allocating a buffer with the size specified by the

memory block size. After that, it reads and writes data from the buffer till the volume specified by the memory total size is achieved. The user is able to make selections about the thread count as well as the kind of operation. The buffer size that has been allotted for the present project is 2 gigabytes, while the overall volume that must be read is 5 terabytes. The reading operation is carried out in a random manner by ten different threads.

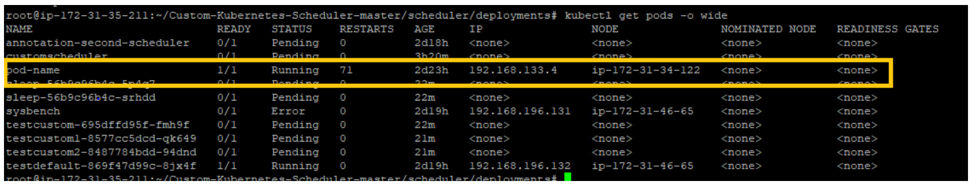


```
root@ip-172-31-35-211:~/Custom-Kubernetes-Scheduler-master/scheduler/deployments# kubectl get pods -o wide
NAME                          READY   STATUS    RESTARTS   AGE     IP                NODE              NOMINATED NODE   READINESS GATES
annotation-second-scheduler   0/1     Pending   0          2d18h   <none>            <none>            <none>           <none>
customscheduler               0/1     Pending   0          3h20m   <none>            <none>            <none>           <none>
pod-name                      1/1     Running   71         2d23h   192.168.133.4     ip-172-31-34-122  <none>           <none>
sleep-56b9c96b4c-5p4q7        0/1     Pending   0          22m     <none>            <none>            <none>           <none>
sleep-56b9c96b4c-srhdd        0/1     Pending   0          22m     <none>            <none>            <none>           <none>
sysbench                      0/1     Error     0          2d19h   192.168.196.131   ip-172-31-46-65   <none>           <none>
testcustom-695dffd95f-fmh9f   0/1     Pending   0          22m     <none>            <none>            <none>           <none>
testcustom1-8577cc5dcd-qk649  0/1     Pending   0          21m     <none>            <none>            <none>           <none>
testcustom2-8487784bdd-94dnd  0/1     Pending   0          21m     <none>            <none>            <none>           <none>
testdefault-869f47d99c-8jx4f  1/1     Running   0          2d19h   192.168.196.132   ip-172-31-46-65   <none>           <none>
```

Figure 14: Deployment using default scheduler

Step-3

The last step involves the deployment of three pods: two pods that will run a sleep application, and one pod that will run a sys-bench workload. The deployment of such three pods takes place just on three distinct nodes. Both the custom scheduler as well as the default scheduler are used in the deployment of a fourth pod. Both the custom scheduler as well as the default scheduler provide a set of metrics that are used to determine how pods should be distributed to nodes.



```
root@ip-172-31-35-211:~/Custom-Kubernetes-Scheduler-master/scheduler/deployments# kubectl get pods -o wide
NAME                          READY   STATUS    RESTARTS   AGE     IP                NODE              NOMINATED NODE   READINESS GATES
annotation-second-scheduler   0/1     Pending   0          2d18h   <none>            <none>            <none>           <none>
customscheduler               0/1     Pending   0          3h20m   <none>            <none>            <none>           <none>
pod-name                      1/1     Running   71         2d23h   192.168.133.4     ip-172-31-34-122  <none>           <none>
sleep-56b9c96b4c-5p4q7        0/1     Pending   0          22m     <none>            <none>            <none>           <none>
sleep-56b9c96b4c-srhdd        0/1     Pending   0          22m     <none>            <none>            <none>           <none>
sysbench                      0/1     Error     0          2d19h   192.168.196.131   ip-172-31-46-65   <none>           <none>
testcustom-695dffd95f-fmh9f   0/1     Pending   0          22m     <none>            <none>            <none>           <none>
testcustom1-8577cc5dcd-qk649  0/1     Pending   0          21m     <none>            <none>            <none>           <none>
testcustom2-8487784bdd-94dnd  0/1     Pending   0          21m     <none>            <none>            <none>           <none>
testdefault-869f47d99c-8jx4f  1/1     Running   0          2d19h   192.168.196.132   ip-172-31-46-65   <none>           <none>
root@ip-172-31-35-211:~/Custom-Kubernetes-Scheduler-master/scheduler/deployments#
```

Figure 15: Deployment using custom scheduler

In the next part, we will go through the findings of the tests that were carried out to determine how best to schedule all pods using both the custom scheduler and the default scheduler. As we can see the CPU utilization is constant while using the default

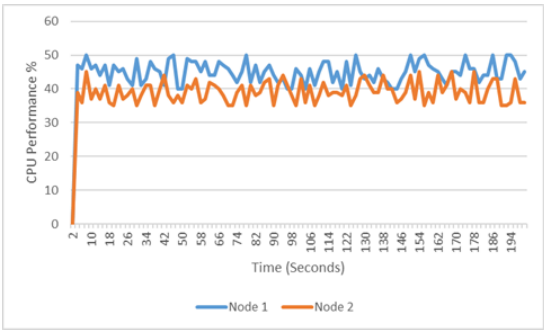


Figure 16: CPU Performance of nodes - Default Scheduler

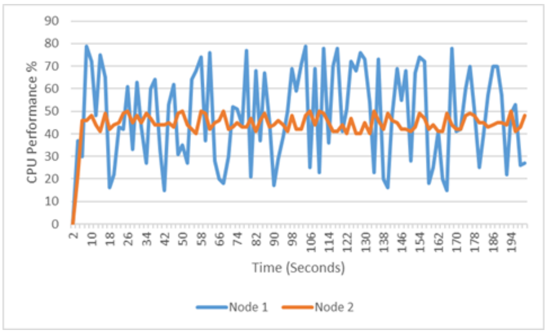


Figure 17: CPU Performance of nodes -Proposed Scheduler

scheduler and increases to 80% while using the custom scheduler for the initial 10 minutes the scheduler keeps the pod in the pending stage before running due to the additional task of working on getting the metric details from the node exporter tool.

6.5 Results & Discussion

The following table provides a summary of the results that were obtained for your perusal. In order to evaluate both the performance of the default scheduler as well as the performance of the proposed scheduler, a total of five pods were deployed across two distinct nodes. The sysbench pods have asked for 24 percent of memory to be deployed to one of the nodes, however, if the application utilizes more memory than the user requested, then another pod that needs 84 percent of memory will fall to sleep once the deployment takes place. When making the comparison, these severe and extreme examples were taken into consideration as appropriate. When determining the bestNode, the proposed custom scheduler will make use of the node exporter tool in conjunction with the Prometheus tool. Both of these tools will take into account the storage metrics and send the exact amount of memory being used as well as the memory that is available to the scheduler. Because of this benefit, the scheduler will be in a position to make the correct decision in order to choose the node for the correct node. When compared to the default scheduler, this method assigns the newly created pod to the sysbech node. This node has asked for less memory, but it uses more of it, therefore it is susceptible to unavailability for the pods that are being deployed in this node.

Result		
PODS	**NODE-1**	**NODE-2**
Sys application(memory request of 24 %)	Running	
Sleep application(memory request of 84 %)		Running
Default Scheduler	Running	
Custom Scheduler		Running

The suggested scheduler has been evaluated in Amazon Web Services EC2 cloud services, where scalability and security were taken into consideration during the process. Amazon Elastic Container Service is a managed service that allows users to host Kubernetes on the cloud as well as onpremises datacentres. Amazon EKS is capable of automatically handling the scalability and availability of a Kubernetes control-plane module in the cloud. These nodes are responsible for a variety of important tasks, including the scheduling of containers, the management of application availability, the storage of cluster data, and more. You will be able to make the most of the performance, scalability, stability, and availability offered by Aws platform, in addition to integrations of AWS network and security services.

7 Conclusion and Future Work

We identified the significant contributions which have been made in kubernetes resource scheduling, while also highlighting the ways in which the primary drivers behind custom schedules in Kubernetes have developed over course of the years. Researchers also proposed a technique for categorizing the works that were assessed based on characteristics such as the top-level aims, the target environments as well as workloads, the specific handling of the case of the scheduler that was changed, in addition to the implementation and assessment methods. In addition to this, we offered an in-depth description of the evaluated custom scheduling inputs based on their specific aims, while also doing an analysis of the primary trends that were observed for each purpose.
Thus, customizing the scheduler by itself might not be the best course of action. In point of fact, it's possible that specialized resources will have to be produced in order to offer additional support for the planned scheduling behavior. Because of this, it is possible that more fine-grained and reduced datasets will be required to offer anticipatory scheduling capabilities. It is also important to point out that the Kubernetes organization is continuously working to enhance the system as a whole, including the scheduling in specific. This indicates that problems that are now being solved by custom schedules may, in future releases of a scheduler, be made available by default to be used in scheduling operations. It also implies it is essential to keep a close eye on the most recent improvements that have been made towards the scheduling to ensure that every adjustment that is made in it is in accordance with the most up-to-date best practices.

References

2021-08-0, G. (2021). scheduler-plugins/kep/61-trimaran-real-load-aware-scheduling at master · kubernetes-sigs/scheduler-plugins.

Al Jawarneh, I. M., Bellavista, P., Bosi, F., Foschini, L., Martuscelli, G., Montanari, R. and Palopoli, A. (2019). Container orchestration engines: A thorough functional and performance comparison, *ICC 2019-2019 IEEE International Conference on Communications (ICC)*, IEEE, pp. 1–6.

Bao, Y., Peng, Y. and Wu, C. (2019). Deep learning-based job placement in distributed machine learning clusters, *IEEE INFOCOM 2019-IEEE conference on computer communications*, IEEE, pp. 505–513.

Beltre, A., Saha, P. and Govindaraju, M. (2019). Kubesphere: An approach to multi-tenant fair scheduling for kubernetes clusters, *2019 IEEE Cloud Summit*, IEEE, pp. 14–20.

Bernstein, D. (2014). Containers and cloud: From lxc to docker to kubernetes, *IEEE cloud computing* **1**(3): 81–84.

Burns, B. and Beda, J. (2019). ja hightower, k. kubernetes: Dive into the future of infrastructure.

Cai, H., Wang, C. and Zhou, X. (2021). Deployment and verification of machine learning tool-chain based on kubernetes distributed clusters, *CCF Transactions on High Performance Computing* **3**(2): 157–170.

Chang, C.-C., Yang, S.-R., Yeh, E.-H., Lin, P. and Jeng, J.-Y. (2017). A kubernetes-based monitoring platform for dynamic cloud resource provisioning, *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*, pp. 1–6.

Chima Ogbuachi, M., Reale, A., Suskovics, P. and Kovács, B. (2020). Context-aware kubernetes scheduler for edge-native applications on 5g, *Journal of communications software and systems* **16**(1): 85–94.

Ghag, A., Ben-Itzhak, Y., Pettit, J. and Pfaff, B. (n.d.). Traffic footprint characterization of workloads using bpf.

Kaur, K., Garg, S., Kaddoum, G., Ahmed, S. H. and Atiquzzaman, M. (2019). Keids: Kubernetes-based energy and interference driven scheduler for industrial iot in edge-cloud ecosystem, *IEEE Internet of Things Journal* **7**(5): 4228–4237.

Kayal, P. (2020). Kubernetes in fog computing: Feasibility demonstration, limitations and improvement scope, *2020 IEEE 6th World Forum on Internet of Things (WF-IoT)*, IEEE, pp. 1–6.

Li, D., Wei, Y. and Zeng, B. (2020). A dynamic i/o sensing scheduling scheme in kubernetes, *Proceedings of the 2020 4th International Conference on High Performance Compilation, Computing and Communications*, pp. 14–19.

Li, J., Liu, B., Lin, W., Li, P. and Gao, Q. (2019). An improved container scheduling algorithm based on pso for big data applications, *International Symposium on Cyberspace Safety and Security*, Springer, pp. 516–530.

Medel, V., Tolón, C., Arronategui, U., Tolosana-Calasanz, R., Bañares, J. Á. and Rana, O. F. (2017). Client-side scheduling based on application characterization on kubernetes, *International Conference on the Economics of Grids, Clouds, Systems, and Services*, Springer, pp. 162–176.

Ogbuachi, M. C., Gore, C., Reale, A., Suskovics, P. and Kovács, B. (2019). Context-aware k8s scheduler for real time distributed 5g edge computing applications, *2019 International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, IEEE, pp. 1–6.

Rejiba, Z. and Chamanara, J. (2022). Custom scheduling in kubernetes: A survey on common problems and solution approaches, *ACM Journal of the ACM (JACM)* .

Rodriguez, M. A. and Buyya, R. (2019). Container-based cluster orchestration systems: A taxonomy and future directions, *Software: Practice and Experience* **49**(5): 698–719.

Santos, J., Wauters, T., Volckaert, B. and De Turck, F. (2019). Towards network-aware resource provisioning in kubernetes for fog computing applications, *2019 IEEE Conference on Network Softwarization (NetSoft)*, IEEE, pp. 351–359.

Townend, P., Clement, S., Burdett, D., Yang, R., Shaw, J., Slater, B. and Xu, J. (2019). Improving data center efficiency through holistic scheduling in kubernetes, *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, IEEE, pp. 156–15610.

Ungureanu, O.-M., Vlădeanu, C. and Kooij, R. (2019). Kubernetes cluster optimization using hybrid shared-state scheduling framework, *Proceedings of the 3rd International Conference on Future Networks and Distributed Systems*, pp. 1–12.

Xu, C., Rajamani, K. and Felter, W. (2018). Nbwguard: Realizing network qos for kubernetes, *Proceedings of the 19th International Middleware Conference Industry*, pp. 32–38.