Handling of a Pod Restart to Re-scale Its Resource Requests During Vertical Pod Autoscaler in Kubernetes

Muluken Mannaye Lema

A Thesis Submitted to the Department of Computer Science and Engineering
School of Electrical Engineering and Computing
Presented in Partial Fulfillment of the Requirement for the Degree of Master's in Computer Science and Engineering

Office of Graduate Studies
Adama Science and Technology University

February, 2023
Adama, Ethiopia

Handling of a Pod Restart to Re-scale Its Resource Requests During Vertical Pod Autoscaler in Kubernetes

Muluken Mannaye Lema

Advisor: Dr.-Ing. Frezewd Lemma

A Thesis Submitted to the Department of Computer Science and Engineering
School of Electrical Engineering and Computing
Presented in Partial Fulfillment of the Requirement for the Degree of Master's in Computer Science and Engineering

Office of Graduate Studies
Adama Science and Technology University

Adama, Ethiopia.
February, 2023.

Declaration

I declare that this Master thesis entitled **Handling of a Pod Restart to Re-scale Its Resource Requests During Vertical Pod Autoscaler in Kubernetes** is my original work. That is, it has not been submitted for the award of any academic degree, diploma or certificate in any other university. All sources of materials that are used for this thesis have been duly acknowledged through citation.

_____	_____	_____
Name of student	Signature	Date

Recommendation of Advisors/ Supervisors

I, the major advisor of this Thesis hereby certify that I/we have read the revised version of the thesis entitled "**Handling of a Pod Restart to Re-scale Its Resource Requests During Vertical Pod Autoscaler in Kubernetes"** prepared under my guidance by Muluken Mannaye Lema. submitted in partial fulfillment of the requirements for the degree of Mater's of Science in Computer Science and Engineering. Therefore, I recommend the submission of revised version of the thesis to the department following the applicable procedures.

_____	_____	_____
Major Advisor/Supervisor	Signature	Date
_____	_____	_____
Co-advisor/Co-supervisor	Signature	Date

Approval Page

I, the advisor of the thesis entitled "**Handling of a Pod Restart to Re-scale Its Resource Requests During Vertical Pod Autoscaler in Kubernetes"** and developed by Muluken Mannaye Lema hereby certify that the recommendation and suggestions made by the board of examiners are appropriately incorporated into the final version of the thesis.

_____	_____	_____
Major Advisor/Supervisor	Signature	Date

_____	_____	_____
Co-advisor/ Co-supervisor	Signature	Date

Approval of Board of Reviewers

We, the undersigned, members of the Board of Reviewers of the Thesis by Muluken Mannaye Lema have read and evaluated the thesis entitled "**Handling of a Pod Restart to Re-scale Its Resource Requests During Vertical Pod Autoscaler in Kubernetes**" and examined the candidate during open defense. This is, therefore, to certify that the thesis is accepted for partial fulfillment of the requirement of the degree of Master of Science in Computer Science and Engineering.

_____	_____	_____
Chairperson	Signature	Date
_____	_____	_____
Internal Examiner	Signature	Date
_____	_____	_____
External Examiner	Signature	Date

Final approval and acceptance of the thesis is contingent upon submission of its final copy to the Office of Postgraduate Studies (OPGS) through the Department Graduate Council (DGC) and School Graduate Committee (SGC).

_____	_____	_____
Department Head	Signature	Date
_____	_____	_____
School Dean	Signature	Date
_____	_____	_____
Office of Postgraduate Studies, Dean	Signature	Date

ACKNOWLEDGMENT

I would like to thank the almighty God for providing me everything I need in my life. Next, to my life-coach, my mother S/r Wogayehu Adugna: because I owe it all to you. Many Thanks! I am grateful to my family members and friends, who have provided me through moral and emotional support in my life and who have supported me along the way.

A very special gratitude goes out to all down to my advisor Dr.-Ing Frezwed Lemma for the encouragement and support by giving me good comments and suggestions. I would not forget to remember his encouragement and more over for their timely support and guidance until the completion of the thesis. I am also grateful to the following university staff: Mr. Mesfin, Mr. Ayenew Yifru, and other CSE department staffs for their unfailing support and assistance.

Finally, last but by no means least; also to everyone in the (Cloud) SIG lab, it was great sharing laboratory with all of you during last two years.

Thanks for all your encouragement!

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

List of Acronyms

CA	Cluster Autoscaler
CRIU	Checkpoint Restore in Userspace
HPA	Horizontal Pod Autoscaler
KEDA	Kubernetes Event-Driven Autoscaler
LXC	Linux Containers
OS	Operating System
QoS	Quality of Service
VM	Virtual Machine
SLA	Service Level Agreement
VPA	Vertical Pod Autoscaler

ABSTRACT

The cloud computing paradigm is being transformed by container technology. Running containerized applications creates an abstraction layer that handles cluster management from the perspective of a cloud service provider. The dominant container orchestration platforms for providing container-based infrastructure with automatic deployment, scaling, and operation of containers on the underlying cluster are Docker Swarm and Google Kubernetes. Currently, changing container resource requests in Kubernetes requires restarting the pod. This restart can be undesirable for certain stateful applications, as moving or copying existing state information could be expensive or unfeasible depending on the underlying implementation. Service state information may become temporarily inaccessible, and users trying to access the service may encounter non-trivial delays before getting back to expected performance. Also, excessive scaling might generate excessive load on Kubernetes components that are responsible for handling the pod restart. We implemented parameters that can limit the frequency of re-scaling and avoid unnecessary restarts. We think by tuning these parameters, it might be potentially possible to optimize the algorithms for the constraints on re-scaling frequency. We made this study on kubernetes and docker containers followed by proposing a vertical scaling solution that can add or remove compute resources like cpu and memory to the containerized application. For testing, two container deployment scenarios were taken into consideration, one of which ran a single instance of the container and the other of which ran two replicas of the container. Apache Jmeter generates the http workload from a hardware local work area to the web service running on Kubernetes on EC2. When the web server is undergoing load testing and the performance parameters of the application are being watched, vertical scaling is done. In order to visualize latency, throughput, and the number of active threads over time, Apache Jmeter custom plugins were employed. Latency and connect times were the analyzed performance metrics of the containerized application. From the obtained results, it was concluded that vertical scaling has no significant impact on the performance of a containerized application in terms of latency and connect times.

Keywords: - cloud, Docker, Kubernetes

CHAPTER ONE

1. INTRODUCTION

1.1 Background of the study

Containers, an evolving virtualization solution, have gained remarkable popularity in recent years and are substituting virtual machines (VMs) due to many favorable properties such as shared host OS, quick launch time, portability, scalability, and fast deployment (Wang, 2021). Containers allow applications to encapsulate all necessary dependencies (code, runtime, system tools, and system libraries) into a sandbox in order to create a platform-independent run-time environment that improves productivity and portability (-Lukman Ibrahim Isa, 2021).

Numerous container technologies exist, including Docker, LXC, and Kubernetes, among others. Furthermore, some cloud service providers run containers on top of VMs to improve container isolation, enhance functionality, and simplify system management. Container technologies are gaining popularity among developers and in the deployment of various microservices and applications such as smart vehicles, IoTs, and fog/edge computing. As a result, many cloud service providers have begun to offer container-based cloud services in order to meet the increasing demand. Google Container Engine (cloud.google.com), Amazon Elastic Container Service (aws.amazon.com), and Azure Container Service (azure.micorsoft.com) are some examples.

The cloud computing paradigm is being transformed by container technology (Ahmad et al., 2021). Running containerized applications creates an abstraction layer that handles cluster management from the perspective of a cloud service provider. The dominant container orchestration platforms for providing container-based infrastructure with automatic deployment, scaling, and operation of containers on the underlying cluster are Docker Swarm and Google Kubernetes.

Kubernetes is a popular open-source framework becoming an essential tool for managing container-based clusters and used for tasks such as automatic containerized application deployment, scaling, and management. As Kubernetes systems scale up, elasticity becomes key. Maintaining performance requires sufficient resources to be provisioned to the applications when the workload increases. At the same time, it is better to free idle resources and reduce resource wastage, which is vital in guaranteeing cost-efficient operation.

It has been proved that valid scaling schemes can effectively reduce resource consumption and improve the quality of service. Due to the importance of guaranteeing the system stability, auto scaling schemes of kubernetes are continuously concerned.

Kubernetes Built-in Autoscalers have three components that enable elastic scaling of clusters: Horizontal Pod Autoscaler (HPA), Vertical Pod Autoscaler (VPA) and Cluster Autoscaler (CA). Kubernetes Event-driven Autoscaler (KEDA) is a third-party component for Kubernetes that builds on top of HPA. Kubernetes' Horizontal Pod Autoscaler (HPA) is one of the control loops integrated in the Controller Manager. It allows dynamically adjusting the number of Pod replicas for a particular Deployment by consuming the Metrics API (T. T. Nguyen et al., 2020).

The Vertical Pod Autoscaler (VPA) automatically sets resource requests and limits of a Pod based on historical usage. Thus, it allows each Pod to have the necessary resources available while minimizing excess resources (so-called slack). It can both down-scale Pods when the initially requested resources were too high as well as up-scale Pods when their usage indicates that they under-requested resources (Henschel, 2021). This means that the resources in the cluster are used and shared efficiently, because each Pod is adjusted to the amount it currently requires, instead of operating with static thresholds. Additionally, this improves Pod scheduling on nodes (Rodriguez & Buyya, 2019). VPA can either use the Metrics Server or a Prometheus instance to obtain time-series metrics. Unlike HPA, VPA only supports scaling based on CPU and memory metrics, but not custom (external) metrics (T. T. Nguyen et al., 2020).

VPA and HPA were widely studied and used by the researchers and developers, which also made a good result in the massive production environment (T. T. Nguyen et al., 2020). Whereas, the pod brained schemes did not completely solve all the problems on scaling (Ross et al., 2018). Depending on the problem scenario, generally, the podgrained scaling solutions mainly focus on improving service capabilities by optimizing resource allocation under existing limited resources. By comparison, a node-grained scaling scheme tends to increase computational power through resource expansion, thereby improving system capability and stability.

The current implementation of VPA does not require the user to provide any resource requests, for CPU and memory, for a pod. By default, the initial allocation in Kubernetes is a configurable lower

limit or the minimum guaranteed resources allocated for a pod. The VPA uses a recommender, which tracks the historical usage of the pod and suggests a new limit whenever needed.

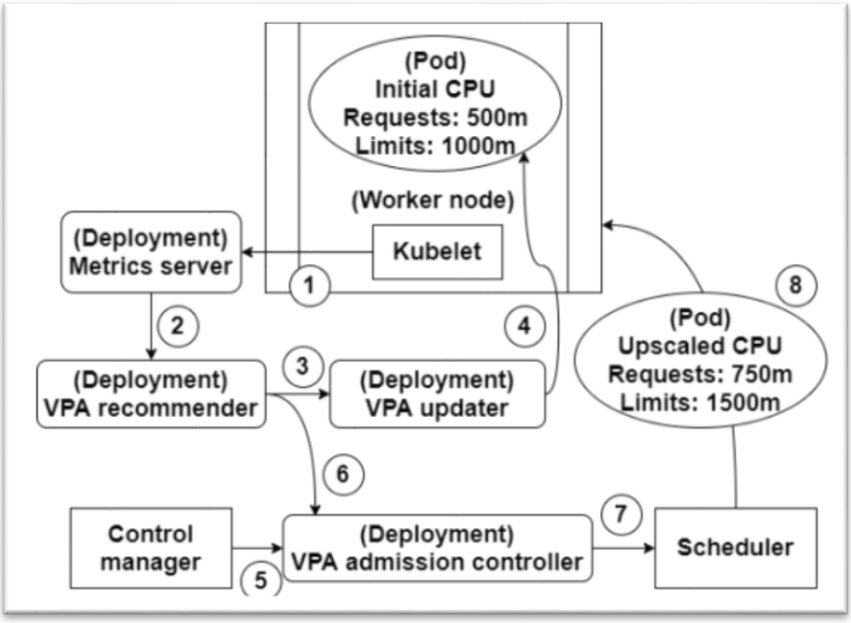

Figure 1. 1 Vertical Pod Autoscaler architecture (Wang, 2021)

The VPA implementation restarts the pod every time the allocation is changed. Such a disruptive process results in a loss of the state of the containers. Though Docker released a Docker update, to allow updation of the resources allocated to the containers, it cannot be directly implemented with large distributed systems like Kubernetes. This is because there are various components such as scheduler, control manager and kubelet that depend on the original allocation to take decisions. Unless all these components are aligned, it is currently difficult to leverage the new Docker update feature for VPA.

1.2 Statement of the problem

Currently, changing container resource requests in Kubernetes requires restarting the pod. This restart can be undesirable for certain stateful applications, as moving or copying existing state information could be expensive or unfeasible depending on the underlying implementation. Service state information may become temporarily inaccessible, and users trying to access the service may encounter non-trivial delays before getting back to expected performance. Also, excessive scaling might generate excessive load on Kubernetes components that are responsible for handling the pod restart. Even though Autoscaling in an extremely valuable feature for software providers,

specifying an autoscaling policy that can guarantee that no performance violations will take place is an extremely hard task, and doomed to fail unless considerable care is taken (Kovács et al., 2018). In addition, an in-depth level of knowledge and a high degree of expertise needs to be in place in order to rule based auto scaling policies to be properly configured, which probably there is a lack of in practice (Evangelidis et al., 2017). While the majority of related work today proposes approaches that let developers optimize scaling decisions by using reinforcement learning and threshold-based policies, they fail to address how different types of metrics affect the performance of auto scaling and consequently the overall performance of the cluster. Some related work let developers set autoscaling rules based on certain SLOs (e.g., Request duration and error rates) however the vast majority base these rules on CPU or memory consumption.

The issue of pod restart is one of the major drawbacks to predictive auto scaling and Kubernetes pod autoscaling in general. If in-place resource updates became possible, many proposed autoscaling and Kubernetes pod autoscaling strategies would also become more viable. On the proposed work we aim to make restarts unnecessary when modifying pod resource requests.

1.3. Motivation

With distributed applications and microservices utilizing containers as the building blocks, proper resource allocation to the containers on demand becomes crucial. Due to the presence of dynamic load to services, containers can be scaled in and out based on the load. Scaling is typically categorized into horizontal and vertical scaling. Horizontal scaling is the addition or removal of container replicas whereas vertical scaling is the addition or removal of computing resources to the running containers. Scaling is crucial in ensuring proper and resource utilization and performance. This feature is also known as elasticity. The motivation of this thesis is to analyze the affects in the performance of a containerized application when vertical scaling is implemented.

1.4. Research Questions

- What performance metrics are affected the most when a pod on Kubernetes is vertically scaled?
- How can continuous delivery of service with vertical scaling be achieved on a Kubernetes-based Docker-containerized application?

1.5. General and specific Objectives

1.5.1 General Objective

The general objective of this research is to design a method that handles automatic pod restart during vertical Pod autoscaling.

1.5.2 Specific Objective

To achieve the general objective, the following specific objectives are drawn:

- To Investigate issues on vertical pod autoscaling in Kubernetes Environment.
- To Analyze the current implementation.
- To design a framework for Kubernetes vertical pod autoscaling helps to handle pod restart.
- To evaluate the new framework.

1.6. Significance of the Study

VPA is the less researched and this thesis will show there exists large room for improvement on VPA. The other significance of this thesis work will be enabling continuous delivery of service which supports on eliminating SLA violations. Many proposed and Kubernetes autoscaling strategies would also become viable.

1.7. Delimitation

In this work, we will concentrate on the vertical Pod autoscaling strategy of Kubernetes to enable it to handle pod restart. The primary purpose followed by this solution is to enable continuous delivery of service which also supports eliminating SLA violation.

CHAPTER TWO

2. LITERATURE REVIEW

2.1 Kubernetes

During last years, containers have been the standard technology for building cloud applications. This led to the need of having platforms capable to handle this new kind of workloads and offering tools to ease their deployment and maintenance. One of the most used is Kubernetes (K8s) (Terracciano & Hu, 2021), an open source platform for Container scheduling and management which traces its lineage directly from Borg (Du et al., 2018), the cluster management system used by Google to run its applications. It is designed to completely manage the life cycle of containerized applications and services using methods that provide predictability, scalability, high availability and facilitates both their declarative configuration and automation. The core design principle on which Kubernetes architecture relies is the Controller pattern (Terracciano & Hu, 2021). This makes it possible for users to forget about most of the operations needed to deploy and maintain software. One or more containers can be grouped into a pod, which is a basic deployment unit that can be operated and managed in a Kubernetes cluster. Each pod can be seen as a virtual server and will be assigned a virtual IP address by the master.

2.1.1 Kubernetes Architecture

Kubernetes is a popular open-source platform for automating container-based application deployment, scaling, and management. A pod, Kubernetes' smallest unit, represents a single instance of an application. Each pod contains one or more tightly coupled containers that share the same IP and data storage. Figure 2 depicts the Kubernetes architecture. A Kubernetes cluster typically consists of one or more master nodes and a number of worker nodes. The master node is the control plane in charge of cluster management and control. It is made up of four major parts: etcd, kube-apiserver, kube-scheduler, and kube-controller. (N. D. Nguyen & Kim, 2021) The etcd is a datastore that stores the cluster's configuration and state. The Kubernetes control plane's front end is the kube-apiserver. To interact with the Kubernetes cluster, the user or management request must communicate with the kube-apiserver. The kube-scheduler monitors unscheduled pods and assigns them to a node based on a variety of factors, including resource constraints, affinity, and anti-affinity rules. To keep the cluster in the desired state, the kube-controller constantly monitors

its state. For example, it ensures that an application has the correct number of running replicas based on the desired configuration.

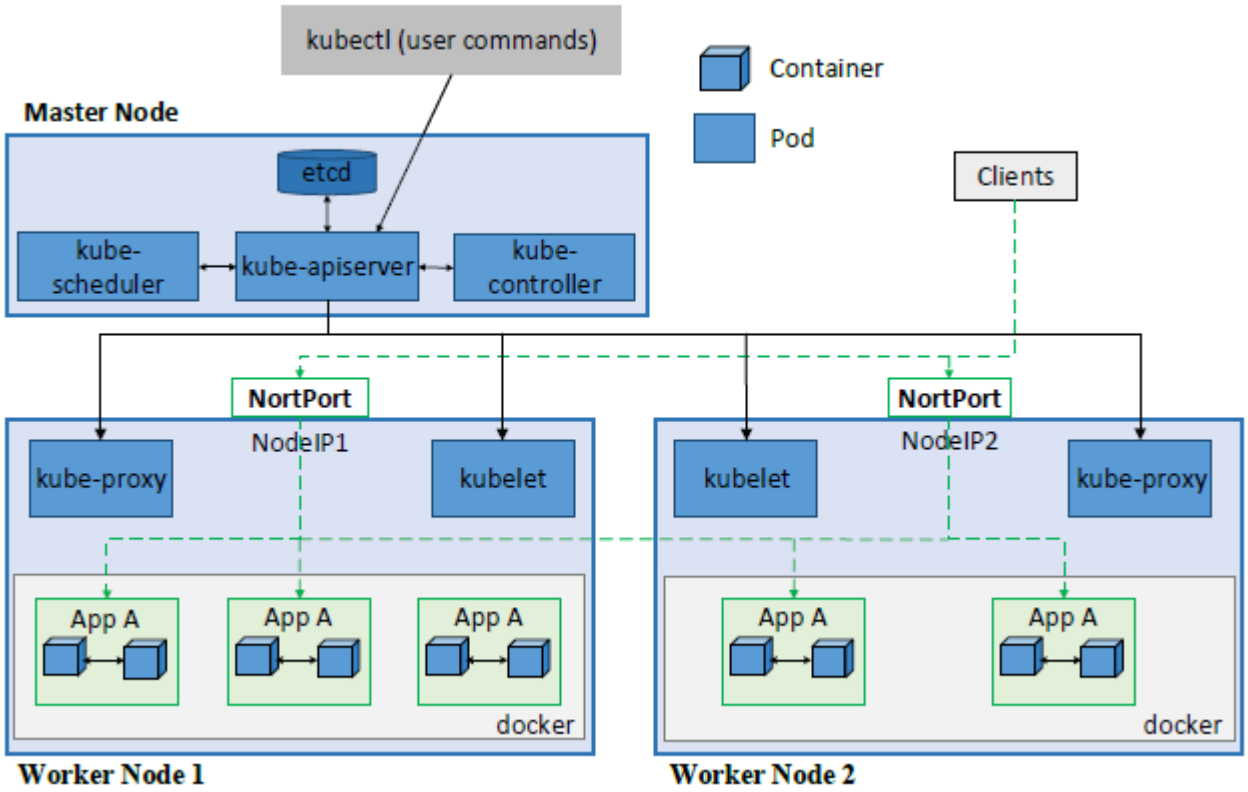

Figure 2. 1 Kubernetes architecture.(N. D. Nguyen & Kim, 2021)

2.2. Containers

A container can be seen as a standard unit of software, packaging up code, and all potential dependencies so that the containerized applications run reliably across different computing environments. Containers provide run-time environments for applications. They are designed to run microservices. One of the most popular types of containers used within Kubernetes is the Docker container (Shah & Dubaria, 2019). Compared with virtual machines that run a complete operating system including the kernel, containers virtualize and run the user-mode portion of an OS, which allows multiple workloads to execute on a single host OS instance. Compared to VMs, containers are lightweight and require fewer system resources such as CPU, memory, and storage (Donca et al., 2020). On the other hand, VMs provide complete isolation between different VMs and the host OS and is capable of running a different OS than the host.

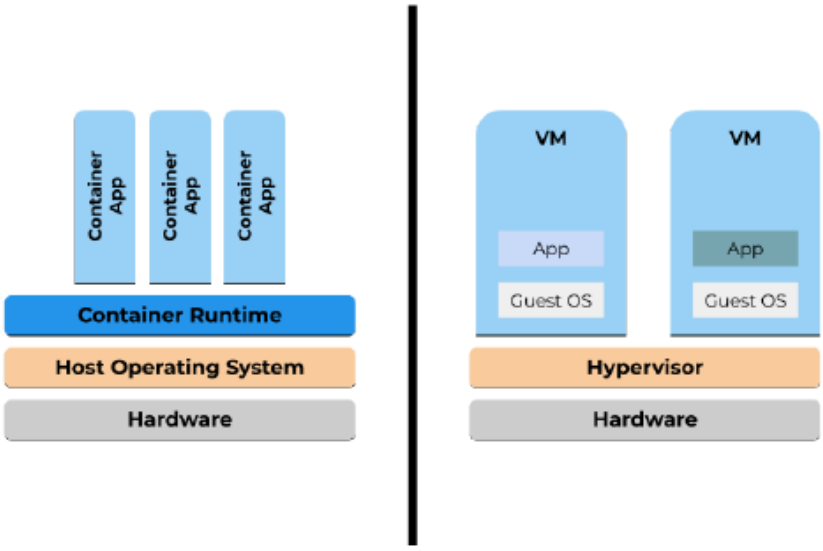

Figure 2. 2 Container vs VM

2.3. Pods

A pod is the most basic scheduling unit within Kubernetes, with each pod containing one or more containers (Wang, 2021). A Kubernetes pod (Kubernetes, 2022) is a group of containers that share the same network. A pod also defines the run-time behavior of the containers. It is also seen as a logical host on which the containers run. This means that containers in a pod can also share the same volumes. (Wang, 2021) Kubernetes automatically provides pods with their own cluster-internal IP addresses at the time of creation, which is used for all communication with the pods. Pods are defined through manifest files, specifying for example container images and resource requests. A set of pods belonging to the same application are identified through labels. In Kubernetes, scaling applications according to load is done at the pod level. As load increases, we can increase the number of pods by creating replicas, and distribute the load evenly among the replicas, which is known as horizontal scaling. The alternative that this research focuses on, is to provide more resources to the containers running in every single pod, which is known as vertical scaling.

2.4. Deployments

A deployment acts as an abstraction layer for the pods with in a kubernetes cluster. Pods are generally deployed using deployments. The deployment object can be used to schedule multiple pod replicas. The fundamental purpose of the deployment object is to maintain resources declared

in the deployment configuration in its desired state. The desired state of a deployment, such as the number of pod replicas, is defined in a configuration file commonly written in YAML format (Kubernetes, n.d.). Such a configuration file also contains pod specifications such as container image, volumes, ports, and resource requests and limits. A Controller manager component, running in the Master node, is then responsible for altering the actual state to the desired state at a controlled rate. According to the specification of deployments, pods are created and destroyed dynamically. A pod from the same deployment can run on multiple machines or nodes. (Wong et al., 2015)

2.5. Nodes

At a high level, a Kubernetes cluster consists of at least one master node and one or more worker nodes, visualized in figure 4. Worker nodes are either virtual machines or physical servers, with each node containing the services required to run pods. Every worker node runs a container run-time such as Docker, which enables pods to run within the node. The Kubelet component contained within each worker node is responsible for inspecting pod specifications given by the Application Programming Interface (API) server contained in the master node, to ensure that those pods are running in a healthy state, and restart pods that have crashed. The other important component inside each worker node is the Kube-proxy, which is responsible for maintaining the distributed network within the cluster and exposes services to the outside world. The master node is responsible for managing the Kubernetes cluster, handling tasks such as scheduling, provisioning, controlling, and exposing API to the clients. As stated above, the master node contains an API server, which is responsible for exposing Hypertext Transfer Protocol (HTTP) APIs that let end-users, cluster components, and external components communicate with one another. Some of the most important APIs are for example those for creating, deleting, modifying, and displaying resource components such as pods in the cluster. The Kubernetes API also allows one to query and manipulate the state of objects such as pods and deployments. The command line tool kubectl can be used to interact with the API server and perform operations in Kubernetes. A master node also contains the Scheduler component, which is responsible for scheduling pods onto appropriate nodes, based on their configurations. The master node further contains a Controller manager component which is responsible for maintaining the health of the cluster, making sure the nodes and pods within the cluster are running correctly. For example, if a worker node goes down, the Controller manager will send commands to the Scheduler to reschedule the pods previously on the node to another one. Configuration files are used for declarative management of objects such as pods and deployments

within Kubernetes. When the controller manager detects differences between the current state of an object within the cluster and the defined state, it makes changes to fix the problem. The last important component in the master node is the etcd key-value database, which is responsible for storing information about the cluster state. (Wang, 2021)

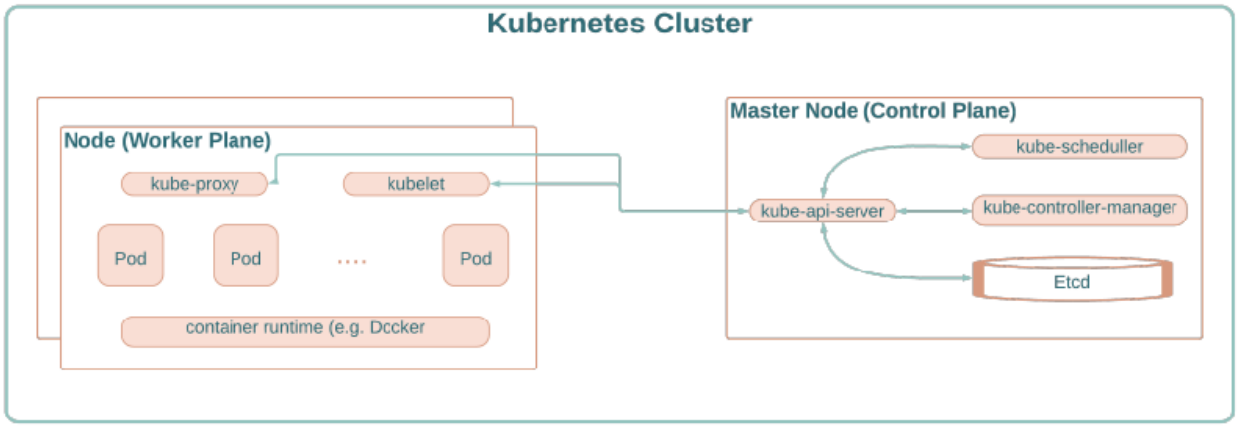

Figure 2. 3 Kubernetes Cluster Architecture.

2.6. Container resource requests and limits

Container CPU and memory requests and limits can be specified in the pod's configuration file, where CPU is given in millicores and memory in bytes. The Kubernetes Scheduler places pods onto nodes based on these definitions, where a pod is scheduled only if there are enough resources. There are two key fields in the pod's configuration file for each container: "request" and "limit." The request field denotes the minimum amount of resources reserved solely for the container., see Figure 5. If there are resources on the node, applications can consume them beyond this specified minimum, however, bounded by the limit field. Decisions made by the Scheduler are based on the resource requests field and do not depend on the limits. The purpose of the resource request field is to guarantee the application enough resources to execute normally even in the case of resource contention. Not defining the resource request field can lead to pods being scheduled onto nodes with insufficient resources to sustain the containers of the pod, which could lead to critical run-time issues. While insufficient CPU leads to CPU throttling, causing poor performance and increased latency, insufficient memory leads to Out-Of-Memory (OOM) errors which cause pods to terminate. (Wang, 2021)

2.7. Metrics server

The Metrics Server component provides the most recent CPU and memory metrics of all pods and nodes on the cluster. This information is collected periodically from the Kubelet running on each node. The default collection rate is once every 60 seconds. This rate can be lowered to a minimum of 15 seconds per collection. The collected metrics are aggregated and stored in memory, ready to be served in Metrics API format. Only the most recent value of each metric is saved. The major drawback is that since metrics are stored in memory if the Metrics Server restarts, all data will be lost (M. Siarkowicz, D. G. Qian Chenglong & Junchen, n.d.).

2.8. Autoscalers

Applications constantly experience fluctuating workloads, so the autoscale feature in cloud computing is important to maintain availability and performance as utilization increases. Moreover, autoscale helps cloud's clients to pay less as consumers only pay for the resources they use. An autoscaler monitors the application's utilization due to the incoming traffic, then it adjusts the amount of resources to provide the expected Quality of Service (Becker et al., 2015). Scaling is the process of providing more resources to respond to application demand increase. (Elamin & Paardekooper, 2021). Kubernetes at its core is a resources management and orchestration tool and for these reasons, one of its most important principles is scalability (Verreydt et al., 2019). Kubernetes aims to handle user applications scalability based on its load and, up to this moment, it offers three controllers to scale users' applications both at Pod and Worker node layers: Vertical Pod Autoscaler (VPA), Horizontal Pod Autoscaler (HPA) and Cluster Autoscaler (CA)(Terracciano & Hu, 2021).

2.8.1. Horizontal pod autoscaling

The Horizontal Pod Autoscaler (HPA) is a Kubernetes feature that scales the number of workload pods in response to resource demand.(Kubernetes.io, n.d.). HPA can automatically scale the number of pods in different Kubernetes controllers, such as a replication controller, deployment, replicaset, or a statefulset. The HPA's scaling action can be based on the Pod's CPU or memory usage. It is also possible to use custom metrics as the Horizontal Pod Autoscaler resource. This is not supported in the HPA's first version, but it is in version 2. (Kubernetes.io, n.d.). For autoscaling using CPU and Memory utilization, the Horizontal Pod Autoscaler collects metrics from the pods

using the metrics API, which requires the metrics server to be deployed as part of the Kubernetes cluster. The metrics gathered will then be compared to the Pod resources request and limit specified in the pod's specification. Using the following equation, the ratio between the desired metric value and the current metric value will be used to calculate the desired number of replicas:

desiredReplicas = currentReplicas * (currentMetricValue / desiredMetricValue) ...eq 1.

Where:

- desiredReplicas: is the count of replica that will be deployed after the Horizontal Autoscaling event.
- currentReplicas: is the number of pods in the current deployment, replica set, or a statefulset.
- currentMetricValue: is the current pod metric value collected via the metric server or any supported custom metrics APIs, such as Prometheus Adapter.
- desiredMetricValue: is the target metric value needed by the application.

Starting with Kubernetes version 1.18, HPA scaling provides for a variable scaling behavior by specifying scale-down and scale-up scaling rules. The periodSeconds variable specifies a finite time limit during which the scaling policies regulate the number of copies to be scaled. Pods or a proportion of pods can be used to represent the policy definition's goal value. A stabilization window can therefore be used to inhibit scaling, either up or down, for a particular time period determined by the stabilizationWindowSeconds variable in order to prevent undesirable scaling events brought on by metric volatility. Instead, in that window, the intended state from the prior period is applied.

2.8.2. Vertical Pod Autoscaler (VPA)

Vertical Pod Autoscaler tries to find the optimal resource allocation for one pod (Kubernetes, 2021a). You may run into the problem of having too few resources, which will slow down the pod with CPU throttling or kill the pod if an Out-of-memory error occurs when you are just guessing. Also having too many resources will be more expensive and reduces the total amount of resources all pods can use. Kubernetes gives resources to a pod based on requests and limits (Kubernetes, 2021c). Requests are the bare minimum of resources reserved for a specific pod; if fewer than the requested resources are available, the pod is not scheduled. Limits are the maximum number of resources that a pod can use if more resources are required than requested. A pod can obtain

additional system resources only if there are any available. When a pod exceeds that limit, it is killed. If HPA assigns pod replicas for resources allocation, it will automatically delegate more or less CPUs and memory to current pods, VPA will proper scheduling the scale of the nodes in the cluster, this means it will delegate more or less nodes, in order to schedule deployments onto them. This means, the VPA does not change current pod resources but instead tests which of the controlled pods have proper resources in order to allow their controllers to restore them with the modified demands on new nodes. The VPA includes a tool called the VPA Recommender, which tracks current and historical resource consumption and suggests values for CPU and memory demands based on the results. Even if there is a trust issue with how the VPA manages the nodes, the VPA can still be used to make recommendations about the resources that best fit the current load (Donca et al., 2020). Horizontal scaling over vertical scaling is preferred because it is less disruptive, especially for stateless services. That is not the case for stateful services, which may benefit from vertical scaling. Vertical scaling is also useful for tuning the actual resource needs of a service based on actual load patterns. (OReilly-Kubernetes-Patterns-WP) The implementation of VPA is still in beta and very new (Kubernetes, 2021c) and there are not many factors that can be changed The VPA works by the help of three components: Recommender monitors the current and past resource consumptions and from that calculates recommended values for the CPU and memory requests of the container. These previous resources could be checkpoints set up during previous runs. Updater checks whether a running pod has the recommended resources set. If this is not the case, it will recreate it using the suggested request values. Certain arguments, such as when to recreate a pod, how much resource increase it can get per scaling event, how long the pod should run before scaling, and how many of a certain type of pod should be recreated at once, can be set. Admission plugin sets the new calculated recommended resource requests on new pods. The VPA can operate in several modes, each of which affects how the new recommended request values are updated in the pods. Auto mode updates it in place, without stopping the container. This is not yet implemented. Recreate mode updates it by recreating pods. Initial mode Only gives newly recreated pods the new request values. Off mode Recommends for the values are never applied but can be inspected. Due to the fact that VPA is still in beta, the implementation has some limitations. The most important point is that recommendations may exceed available resources, in which case the pods are placed on hold. Furthermore, the algorithm in the recommender is the only one available; changing it would necessitate the implementation of a new recommender. As such the only parameters that are changeable is when to evict certain pods in the updater, creating and testing

those on different network functions requires lots of traffic as the recommender slowly works to the optimal resource values based on historical data. (Elamin & Paardekooper, 2021)

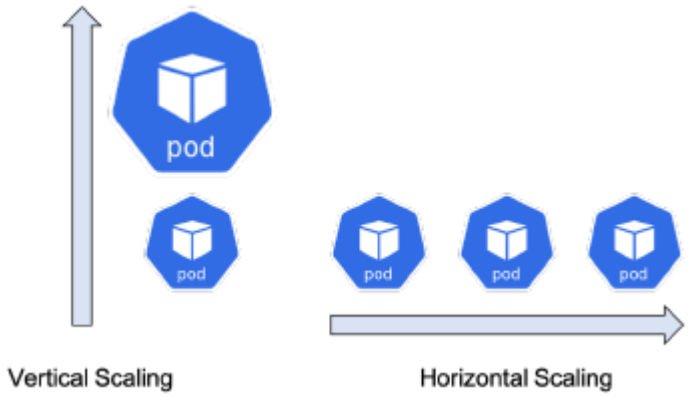

Figure 2. 4 HPA Vs VPA

2.8.3. HPA vs VPA

Horizontal and vertical autoscaling solve two distinct problems. Horizontal scaling addresses the issue of how many replicas of a specific pod spawn. Vertical autoscaling solves the problem of allocating resources to individual pods. With fluctuating demands, horizontal should be preferred over vertical. (Michael Handa, 2020). Vertical is better in stateful services as they cannot use another pod without migrating state and still need a way to reduce resource cost when overusing or add resources when pods are stagnating. Vertical scaling can also be used to fix initial resource demands. These demands can be too lean or too strict and vertical scaling can over time fix them to the right amount based on usage data. Using both vertical and horizontal scaling can be done with multidimensional Pod autoscaling. This a Google Kubernetes Engine (GKE) only solution that lets VPA work on the memory and HPA on CPU utilization (Kubernetes, 2021b). This is consistent with the VPA documentation. (Kubernetes, 2021c). To use VPA and HPA together they need not interfere with each other's calculation by working with other metrics. These other metrics can also be custom metrics.

	Nodes	Pods
Horizontal	# of nodes	# of pods
Vertical	Size of nodes	Size of pods

Table 1. Horizontal vs vertical scaling

2.9. Related work

The review considers only cluster-internal scaling mechanisms (i.e., vertical and horizontal scaling of Pods). There have been interesting papers to improve Kubernetes autoscaling. Nevertheless, there are certainly advances to be made on Kubernetes autoscaling feature. Recent researches into vertical autoscaling in cloud systems have analyzed the autonomic vertical elasticity of Docker containers (Al-Dhuraibi et al., 2017). Another survey analyzes the performance of vertical autoscaling models on stream joins in cloud infrastructures (Rzadca et al., 2020). One significant recent research focuses on vertical scaling of memory resources for jobs running in Google's Borg cloud systems (Rzadca et al., 2020).

In this research, the authors present the autoscaler known as Autopilot, which manages to reduce memory slack from 46% to 23% compared to manually-managed jobs in Google's clusters. At the same time, it also managed to reduce the number of jobs severely impacted by OOM by a factor of 10. Autopilot uses exponentially-smoothed sliding windows over historic usage to generate resource limits. Reinforcement learning techniques are then used to select the best performing window. Autopilot uses a reactive autoscaling strategy that sets resource limits based on past historical usage. For cases like slowly moving window does not manage to react quickly enough, a proactive autoscaling method could potentially help prevent service-level agreement (SLA) violations. Also, by recognizing and taking advantage of repeating patterns, a proactive strategy is potentially able to decrease slack even further.

Title	Objectives	methods	Gaps
RUBAS et al.	Alternative of VPA where it is possible to dynamically change the resources of pods without causing a service downtime by relying on the CRUI functionality.	Vertical scaling via migration	The time of migration is short, it is still significant enough to cause a service interruption. As long as pod is migrating, it cannot process requests and must be compensated elsewhere.
Wang et al	Enable resource sharing between containers and how to reduce container image size.	A new container management system is proposed where containers run on top of a shared layer that contains the essential software and libraries for all containers.	The architecture scores better in terms of image size and startup time but whether is also more memory efficient or guarantees a better performance is not presented.

K. Rzadca et al.	Autoscaler known as Autopilot which manages to reduce memory slack from 46% to 23 % compared to manually managed jobs in Google's cluster	Exponentially -smoothed sliding windows over historic usage to generate resource limits Reinforcement learning techniques to select the best performing window.	For cases where a simple moving window does not manage to react quickly enough.

Table 2. Related work

Previous works directly related to vertical scaling in Kubernetes include the Kubernetes VPA (Najdataei et al., 2020), which sets container resource requests and limits using statistics over a moving window. We will use the default Kubernetes VPA as the baseline when evaluating the performance of our implementation. Another research has studied the disruptive impacts of vertical scaling on the performance of containers running in Kubernetes (Midigudla, 2019). In this research, the authors analyze performance metrics such as application latency and connection time, and the results obtained from experiments concluded that vertical scaling had no significant impact on performance. Other related works have explored the possibility of non-disruptive vertical autoscaling in Kubernetes (Rattihalli et al., 2019). The authors in this research incorporated container migration, to develop a prototype non-disruptive autoscaler called RUBAS, which managed to improve CPU and memory utilization of a test cluster by 10%.

Vertical Pod Autoscaler (VPA) (Wong et al., 2015) is an add-on functionality in Kubernetes that also supports vertical scaling in an automated way to mitigate over-provisioning. RUBAS et al. (Rattihalli et al., 2019) proposes an alternative of VPA where it is possible to dynamically change the resources of Pods without causing a service delay by relying on the Checkpoint Restore in User space (CRUI) functionality. This makes it possible to pause a container, save the complete state of

the container and restart the container at a later time with the correct state. This technique is very effective when it comes to stateful applications. The results show that vertical scaling via a migration with RUBAS can be performed faster than when the resources are restarted with VPA. While the time of a migration is short, it is still significant enough to cause a service interruption. As long as the pod is migrating, it cannot process requests and must be compensated elsewhere.

The effects of dynamic vertical and horizontal scaling of containers has also been studied (Khanh et al., 2015). The idea is enabling vertical scaling for stateful applications i.e. vertical scaling for containers where execution is important without interruption. DoCloud (Arfamaini, 2016), presents a proactive method of performing vertical scaling of Docker containers that run web applications. They pro-actively vary the resource allocation for containers to attain higher utilization. Hadley et. al. (Horne, 2015) have shown the capability of CRIU (Checkpoint Restore In Userspace) across multiple clouds. They show this capability for both stateful and stateless applications. However, we are not aware of any work that combines container migration capability for stateful applications along with vertical scaling. This thesis demonstrates this new capability and explores its effectiveness when used in a Kubernetes environment.

CHAPTER THREE

3. MATERIALS AND METHODS

This chapter's goal is to provide a thorough explanation of the research tools and techniques that will be applied to this project. The study design will be covered in the next chapter after the selection of the research approach. A discussion of their capacity to generate reliable outcomes and fulfill the goals and objectives established by this thesis study will follow. The chapter then moves on to detail the author's sample size and sampling approach as well as the data analysis techniques that will be used. A brief discussion of the study methodology's ethical implications, its limits, and any issues that came up throughout the process is included in the conclusion.

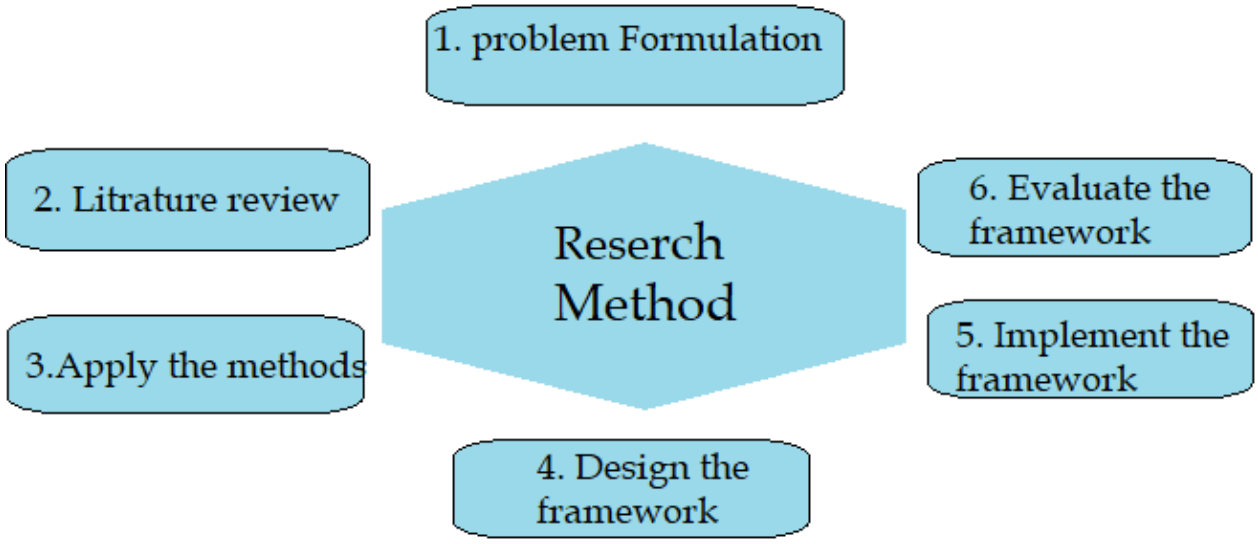

Figure 3. 1 Flow chart of Research Method

3.1. Method

The research technique for this thesis was completed using a methodical manner. To devise a vertical scaling solution, a thorough investigation was carried out. The solution was then implemented under various circumstances, and data was collected and analyzed. Due to workload requirements, vertical pod autoscaling was done manually. The first step was to do a literature review on containers, Docker containers, kubernetes, minikube, and the elasticity of containers, which gave insight into the architectures, tools, and other technologies needed to conduct the tests.

Due to workload requirements, vertical pod autoscaling was done manually. The first step was to do a literature review on containers, Docker containers, kubernetes, minikube, and the elasticity of containers, which gave insight into the architectures, tools, and other technologies needed to conduct the tests. It is possible to specify a deployment specification that takes into account the desired properties and executes the pods appropriately. All instructions and requirements for a pod are transmitted to the containers inside it, and containers operate inside of pods. Users may manage numerous containers via deployments and connect them together to host services by sharing volumes between them. After the deployment is finished, our application must be made available to the internet using a network load balancer.

In Kubernetes, a service may be developed that connects our deployment to a recently constructed load balancer, enabling access to the service through the load balancer's ingress. For a while, load is produced gradually, and the load balancer gradually directs this traffic to the pod. To ascertain the techniques that may be applied to carry out vertical pod autoscaling, Kubernetes architecture and its characteristics were carefully researched. "Rolling update" (Midigudla, 2019), a kubernetes feature that can be used to update an existing deployment, was one such feature that attracted attention.

The deployment controller used by the rolling update functionality (Kubernetes.io, 2021) controls changes to the deployed applications. A deployment may be changed in a variety of ways using rolling updates, for as by altering container images or the number of replica containers. However, the fundamental benefit of rolling updates is that they do not cause deployment problems if a user requests an erroneous update (for example, one that contains faulty container images or uses unavailable resources). The user may keep track of the deployment update request's status and, if necessary, take the appropriate measures. Additionally, rolling updates include a rollback option that cancels the upgrade and returns deployment to the prior configuration.

The following steps have been used to split the thesis work:

Step 1: Analyze the changes in Kubernetes settings following modifying the deployment and look at the options for changing the preset compute resources allocated in a deployment.

Step 2: Apply vertical scaling to the deployment, then examine how the application performs in various circumstances.

3.2. Hosting the containerized application on kubernetes

The containerized application will operate on a single node kubernetes cluster that is set up using Minikube and the Docker driver. The arrangement that we attempted previously with a cluster of two nodes had some issues with ongoing connectivity with many nodes, however this minikube environment made it simple to deal with. The popularity of hosting apps in the cloud causes businesses to place less emphasis on the infrastructure required and more emphasis on the application service.

Figure 3. 2 Minikube starting on docker driver

Minikube offers the same requirements as Docker and easy instructions for a speedy installation of a single-node Kubernetes cluster. For developing and testing containerized apps and services without using a production environment, the setup's simplicity and design make it ideal. It feels like we can access a tiny "Kubernetes Development Kit" with only a few keystrokes. Because of this, we've continued the transformation process to make our software a containerized application, and the architecture resulting from this scenario served as the basis for our web app's Kubernetes design. We can be certain that employing this technology is the best method to increase our deployment capacity in larger settings after evaluating the Kubernetes resources that were established. However, Minikube is just a straightforward Kubernetes package.

The inability to create and deploy to a multi-node cluster is a serious drawback. A deployment is referred to as a pod, which houses the containerized application. A deployment is written in a yaml

file, which has a series of instructions that specify the pod parameters, such as the container image, port, and computing resources, as well as the deployment behavior, such as the number of replicas and namespace it operates on. In Kubernetes, a new namespace is established in order to separate the application from other services. The new namespace produced by using the kubectl api is shown in Figure 9.

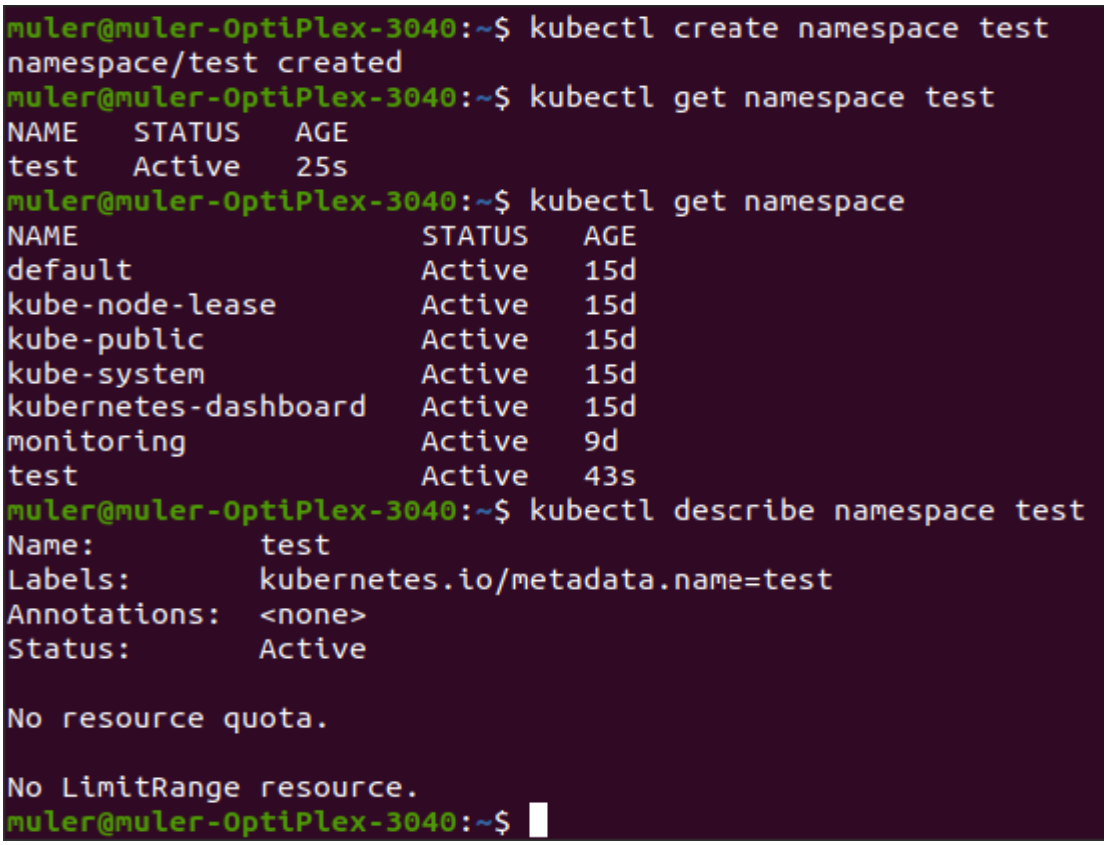

Figure 3. 3 Creating a namespace

We deploy our containerized application as a service using this namespace. Figure 10 displays a sample kubernetes deployment file. Users now have simple access to utilize Docker containers on Kubernetes thanks to the integration of Kubernetes and Docker. When a string matching a Docker image is included in the deployment file's "-image" specification, Kubernetes will automatically download the image from Docker Hub and execute the container within a pod. It is possible to confirm the cpu shares assigned to the container in the deployment file. When a deployment is successful, the produced pods and the cpu shares assigned to them may be examined and confirmed.

```
#Deployment file for a containerized application
kind: Deployment
apiVersion: apps/v1
metadata:
  name: webserver
  namespace: test
spec:
  replicas: 1
  selector:
    matchLabels:
      app: my_app

   template:
    metadata:
      labels:
        app: webserver
    spec:
      containers:
      - name: dhmi17
        image: <docker container image>
        ports:
        - containerPort: 80
```

Figure 3. 4 A sample deployment file

A kubernetes service that provides a load balancer to our deployment has to be defined in order to expose the containerized application to the internet. Users may launch an AWS load balancer service directly from the kubernetes service configuration file thanks to the interaction between Kubernetes and AWS. The load balancer distributes incoming traffic to the pod hosting the containerized application, as seen in figure 11.

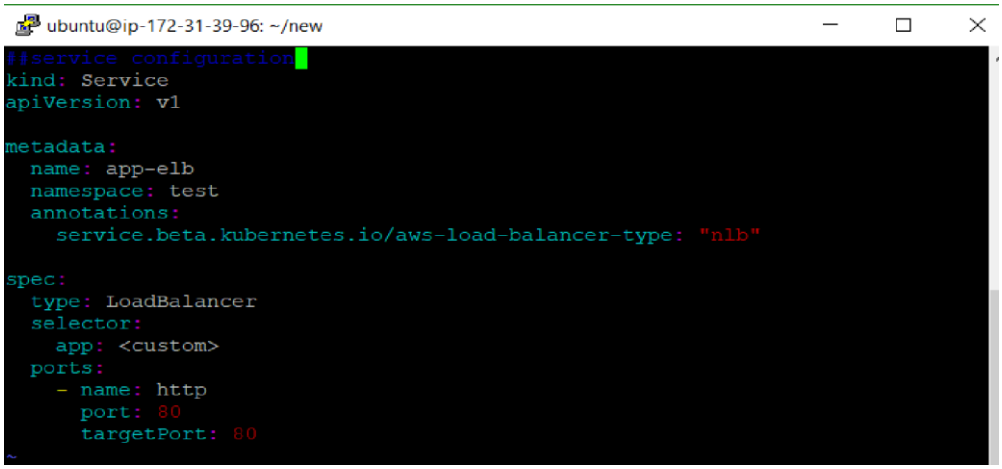

Figure 3. 5 A sample service configuration

Figure 11 illustrates how a yaml file, which also allows for the specification of the pod's incoming and target ports, is used to describe a Kubernetes service. Traffic is redirected from the entering port to the target port of the pod. The pod port to which the load balancer directs incoming traffic is specified by the "-target port" argument.

3.3. Resource allocation to the containers

The deployment configuration file might include a specification addressing the computing resource (cpu and memory) range of the containers (Kubernetes, 2021a). The specification takes the "-limits" and "-requests" parameters. The specification "-limits" refers to the maximum CPU or memory that is made available to the container prior to startup, while the specification "-requests" refers to the maximum CPU and memory that the container may request after starting up. By utilizing this functionality, Kubernetes is able to optimize resource use by ensuring that the pods on the nodes are scheduled correctly. The pods housing the containers won't start, and the deployment will fail, if the value of the parameter "-requests" exceeds the value of the specification "-limits."

The container's inability to request more resources than are available is the cause of this. The container will utilize all of the limitations to start up if no requests are made and just the limits are given. The CPU resource requirements for the container are shown in Figure 12. The input to the specifications "-limits" and "-requests" can be either a decimal, such as "0.5m," which is the value's one hundredth millicore, or an integer, which corresponds to the entire number of CPUs indicated by the integer value. Think about a Kubernetes cluster that has two CPUs and one worker node. One container will start up and request 500m (millicore) of CPU if a deployment is configured with spec "-limit" set to "1" and spec "-request" set to "0.5m".

```
#Deployment file for a containerized application
kind: Deployment
apiVersion: apps/v1
metadata:
  name: webserver
  namespace: test
spec:
  replicas: 1
  selector:
    matchLabels:
      app: my_app

  template:
    metadata:
      labels:
        app: webserver
    spec:
      containers:
      - name: dhmi17
        image: <docker container image>
        ports:
        - containerPort: 80
        resources:
          limits:
            cpu: "1000m"
          requests:
            cpu: "500m"
```

Figure 3. 6 A deployment file which describe compute resources

3.4. Vertical Scaling Method

A containerized application may be vertically scaled by adding or deleting computational resources. When using a Kubernetes platform, containerized applications are orchestrated using Kubernetes pods. In a kubernetes architecture, pods serve as the building blocks and manage the containers. The only way to communicate with containers is to address the pods that are hosting them. The pods are set up when a deployment is detailed such that they precisely adhere to the instructions provided in the deployment configuration. When a pod is attempted to be modified, such as by terminating it, it dies but is instantly recreated in accordance with the deployment settings. This feature makes sure that the underlying pod cannot be changed without a deployment adjustment.

3.5. Updating a Deployment

Kubectl api may be used from the terminal to modify the current deployment settings in order to boost the computing resources of containers in a kubernetes deployment. This makes it possible to change the deployment's current configuration. The deployment may be updated by changing the container specifications.

Figure 3. 7 Updating a deployment

Figure 13 shows a kubernetes deployment in the namespace "test" that has been changed to reflect changes to the container's CPU spec ("-requests"). As a result, the old pod is terminated and a new one is created with the modified deployment settings.

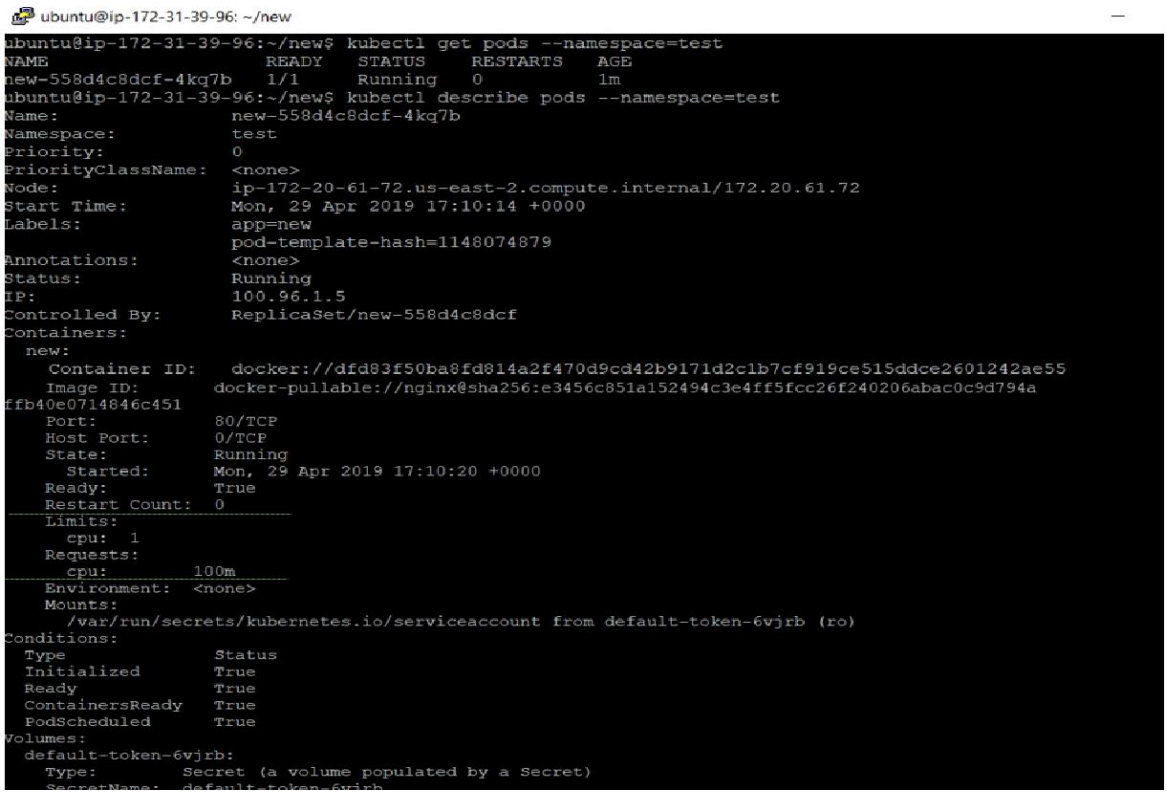

Figure 3. 8 Initial pod description

The initial pod description of the deployment is shown in Figure 14. You may look up the container's original CPU specifications. Figure 15 shows how the new pod produced in the namespace "test" is described, and how the modified CPU requirements may be checked.

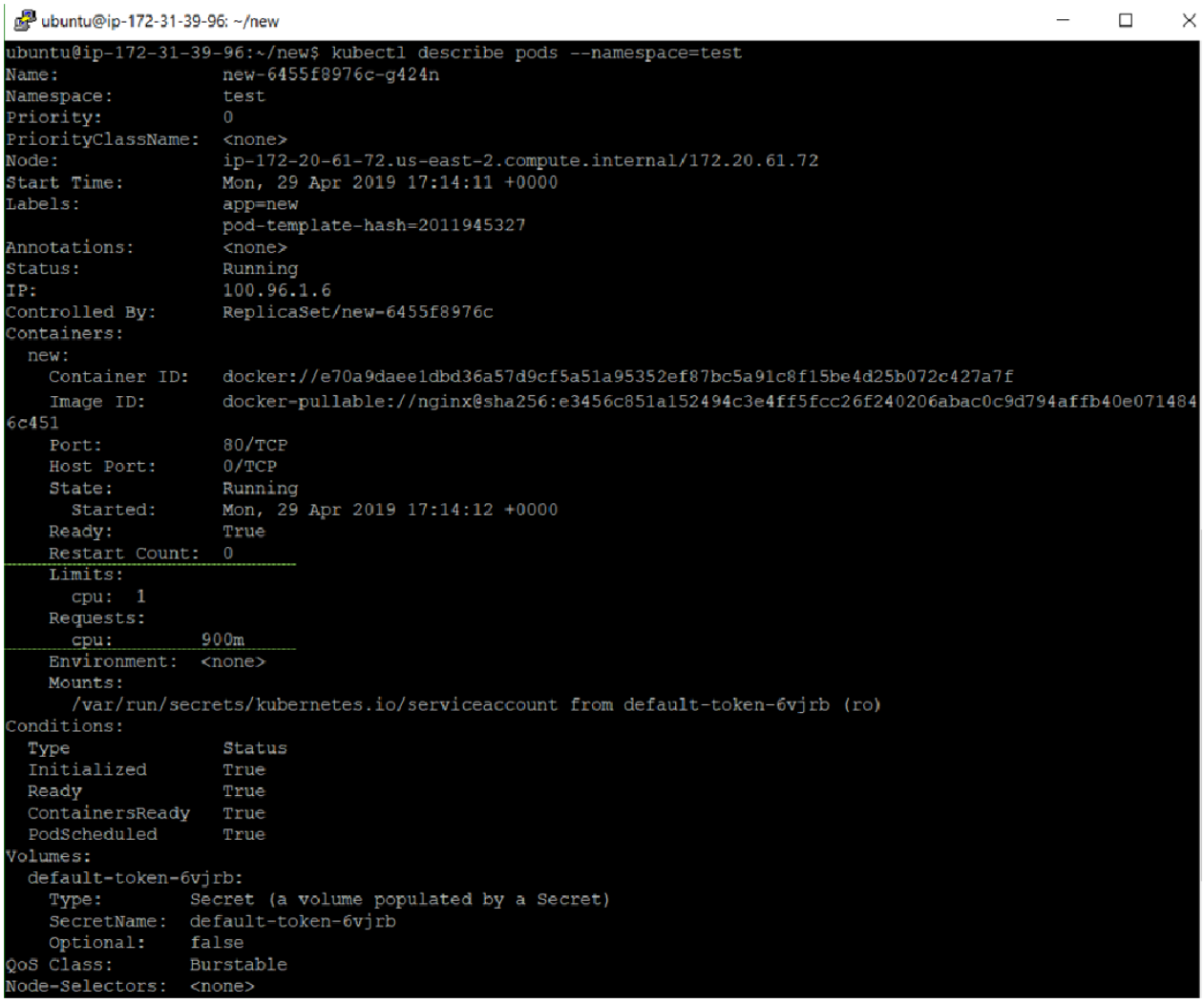

Figure 3. 9 The newly created pod description

3.6. Environment Setup

3.6.1. Tools and Technologies Used in Research

This investigation was carried out in a single node cluster setting using Minikube (V 1.26.1). (Ubuntu). For deployment and autoscaling, Kubernetes VPA (Vertical Pod Autoscaler) was utilized. The microservice-based application is created in PHP to handle the computationally demanding tasks. The following technologies and tools were employed in this study's research:

- The most recent version of Kubernetes (1.18) and the Ubuntu 20.04.4 LTS operating system are used in this study.
- Minikube (V 1.26.1) with Docker driver: Tool to quickly run a Kubernetes cluster locally on one node.

- Kubectl (V 1.24.3): The Kubectl utility allows users to provide commands to the Kubernetes cluster. Kubectl has allowed for the application's deployment. Additionally, Kubectl offers Kubernetes logs.
- Docker Driver: Microservices are containerized using Docker. The study's implementation of Docker is (v20.10.6.) Microservices are kept in containers. Pods will be booked for containers.
- PHP-Apache: Apache is the web server responsible for accepting and processing HTTP requests. PHP-Apache makes it possible to create dynamic content.
- Terminal: Used to configure setups and manage clusters.

3.7. Assessment Carried Out in This Research

To ensure consistency in the evaluation of the custom controller findings, a total of three trials are run. After the workload is produced, specific time slots of two minutes, eight minutes, and fifteen minutes are utilized for assessing the three studies. Here, it has been investigated whether or not the Kubernetes autoscaler is scheduling the required amount of pods. The Kubernetes Vertical Pod Autoscaler default algorithm is used for validation. On the basis of the goal and existing CPU use, observations were made.

3.8. Design Specification

This section talks about the specifications that were utilized to create the project. The suggested architecture of the system is defined in 3.8.1 of the proposed Specification of the System, and the proposed architecture of the custom controller is presented in 3.8.2.

3.8.1. Proposed Specification of the System

Lightweight Docker containers are employed as part of this thesis. When microservices are containerized, they may be upgraded or restarted after failing. This study manages the deployment and operation of containers using the orchestration technology Kubernetes.
Kubernetes is used to manage the containers inside the cluster. A Kubernetes single node cluster is run using Minikube and the Docker driver. The hardware and software requirements for a kubernetes single node cluster are shown in Table 3 below.

Table 3: Hardware and software tools used for this research implementation

sno	Tools used	Description
1.	Desktop pc	-Dell Optiplex 3040 - Processor: intel core i5 cpu 3.2GHz * 4 - Memory: $ GiB
2.	Operating system	- Ubuntu 20.04.4 LTS
3	Working environment and command line tools	- Minikube: Version: 1.26.1 using docker driver - kubectl: Version: 1.24.3 - Docker: Version 20.10.17
4	Visualization Software	Prometheus: Grafana:

The Kubernetes group that houses the web application was configured up to operate on the Amazon Web Services EC2 environment.The minimal need for kubernetes is 2 cores, although the minikube with docker driver operates on an Ubuntu 20.04 machine with a 4 core CPU.

The designation for the instance type is t2.micro. Only the requests from the remote host ip are now permitted thanks to modifications made to the incoming and outbound traffic rules. Additionally, the virtual machine is set up to get a private IP address that will be used to access its terminal. Once the instance is running, a remote computer can use ssh to connect to it using its public dns. S3 storage service is utilized to store the cluster state. The bucket has an internationally distinctive name, as was indicated in section 2.5.3. Other AWS services can now utilize the bucket after configuring the name.

The kubernetes cluster's hosted zone needed to be created next. Users using Route53 in AWS can set up a hosted zone and sub-domains. If the provided zone's suffix is "k8s.local," the kubernetes cluster nodes will automatically setup the sub-domains. The kops api is used at the terminal to construct the kubernetes cluster. A single node in a Kubernetes cluster is set up for testing. Kubernetes nodes that pop up are shown in Figure 16 along with the information that goes with

them. For testing purposes, a bespoke CPU-intensive containerized web application is made available as a service. HTTP requests are made to the web service's load balancer ingress using Apache Jmeter, which has been configured locally on the host computer. Figure 4.15 shows an illustration of the experimental architecture.

.

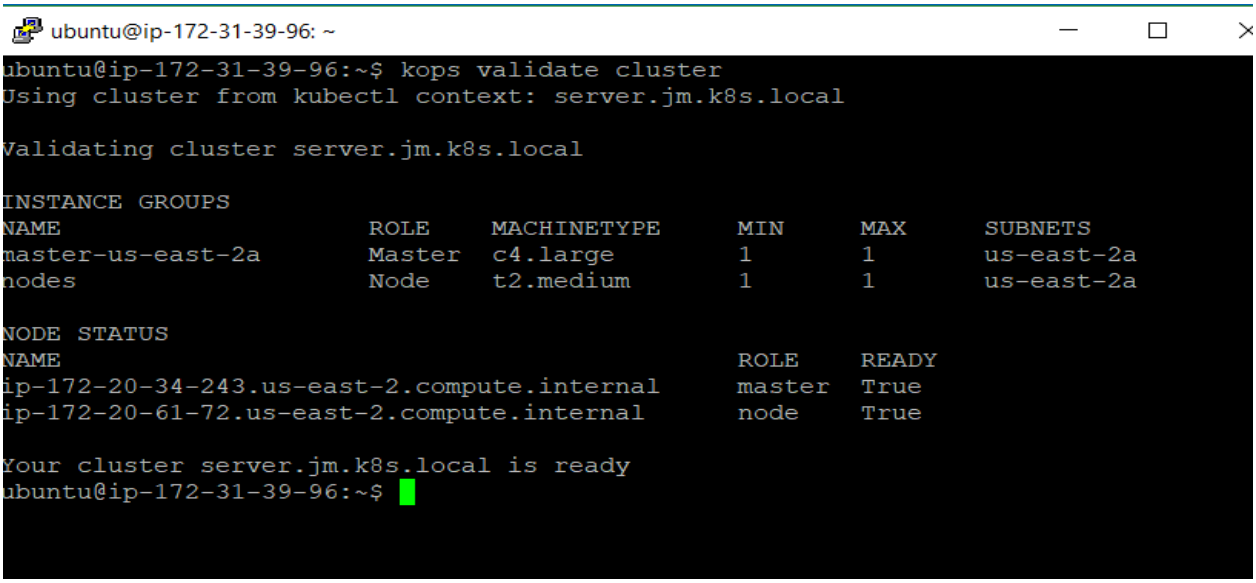

Figure 3. 10 Asingle node cluster description

3.9. Experimentation Methodology

For testing, two container deployment scenarios were taken into consideration, one of which ran a single instance of the container and the other of which ran two replicas of the container. Apache Jmeter generates the http workload from a hardware local work area to the web service running on Kubernetes on EC2. When the web server is undergoing load testing and the performance parameters of the application are being watched, vertical scaling is done. In order to visualize latency, throughput, and the number of active threads over time, Apache Jmeter custom plugins were employed. The following procedures were followed during the experiments:

- Deploy the web service using just one container and two replicas.
- Create a workload for the web service's DNS using Apache Jmeter.
- While the web server is undergoing load testing, do vertical scaling and use Jmeter plug-ins to record the performance metric data.

3.9.1. Experiment 1: Single Clone (Replica)

In this case, a single replica of the container—which is executing our web application—is operating under a kubernetes deployment, with the container receiving a "100m" CPU share.This creates a kubernetes pod for our container, which runs on a node with 1 vCPU in the kubernetes system. During the loads, vertical scaling is done when HTTP load is supplied using Jmeter from a local physical computer with a thread group setup. The CPU resources allocated to the container are raised during vertical scaling from "100m" to "900m". The vertical scaling range is detailed in Table 3.2.

For dynamic loading, two test scenarios with various request types were employed. Jmeter is set up with the following settings for the two test cases:

1. 2000 samples over 60 seconds
    a. Number of users: 500
    b. Ramp-up (seconds): 50
    c. Loop count: 4
    d. http request defaults: GET
2. 2000 samples over 30 seconds
    a. Number of users: 500
    b. Ramp-up (seconds): 50
    c. Loop count: 4
    d. http request defaults: GET

The aforementioned Jmeter setups will send 2000 http GET queries to the web service over a period of 60 and 30 seconds, respectively. Vertical scaling is done during the load testing as the load is created over the course of 60 and 30 seconds, and the performance data are gathered from Jmeter plugins after the load test.

3.9.2. Experiment 2: Two clone (replicas)

A kubernetes deploy is set up to operate two replicas of the capsule in this case (running our web application). This creates two Kubernetes pods, each operating a single container, and utilizing a single vCPU on a Kubernetes node. The deployment is set up to have a "50m" CPU share again for container and "50m" CPU shares for each of the two clones that are created, for a total of "100m" CPU shares for the deployment that will operate as a service. Even during dynamic loading, vertical

scaling is done when HTTP load is supplied using Jmeter from a local physical computer with a thread group setup. By increasing the CPU shares allocated to the container from "50m" to "450m," or from "100m" to "900m," it is possible to scale the deployment vertically. The vertical scaling range and its features are shown in Figure 4.16.

For load testing, two test scenarios with various request samples were employed. Jmeter is set up with the following settings for the two test cases:

1. 2000 samples over 60 seconds
    a. Number of users: 500
    b. Ramp-up (seconds): 50
    c. Loop count: 4
    d. http request defaults: GET
2. 2000 samples over 30 seconds
    a. Number of users: 500
    b. Ramp-up (seconds): 30
    c. Loop count: 4
    d. http request defaults: GET

CHAPTER FOUR

4. RESULT AND DISCUSSIONS

In this chapter, we assess the suggested algorithms that were provided in section 3.4. First, we discuss the workload and experimental settings that were used to assess the algorithms' performance in Sections 4.1 and 4.2. Then, in Section 4.3, we first go into specifics of experiments to clarify the procedures and difficulties we encountered while putting the suggested algorithms into practice, and then we give our opinions. The parameters that are analyzed are latency and connect times. The amount of delay between initiating a request and immediately receiving a response is known as latency. The amount of time needed to establish a TCP connection between the client (the host system running Apache Jmeter) and the server is known as the "connect time" (Docker containerized application).

4.1. Experiment 1 One Replica

The environment described in the earlier section serves as the setting for the tests. Jmeter is used to track the throughput and average bytes per second for the transmitted requests, as well as the lowest response time in milliseconds (ms), maximum response time in milliseconds (ms), average response time in milliseconds (ms), and throughput.

4.1.1 Test Case 1

The first test had 500 threads (users), a loop count of 4, and a duration of 60 seconds. Scaling vertically from "100m" to "900m" During the 60 second period, CPU sharing to the container was carried out at the 13th second of a load test. The metric error% in table 4.1 indicates the proportion of unsuccessful requests. In 2000 samples, 4 samples (0.2%) were unsuccessful.

Label	Samples	Min response Time(ms)	Max. Response Time(ms)	Avg. Response Time(ms)	Standard Deviation	Error %	Throughput	Avg. Bytes
HTTP Requests	2000	246	469	231	19.53	0.2	32.8/sec	852.3

Table 4.1: Data for the HTTP requests for case 1

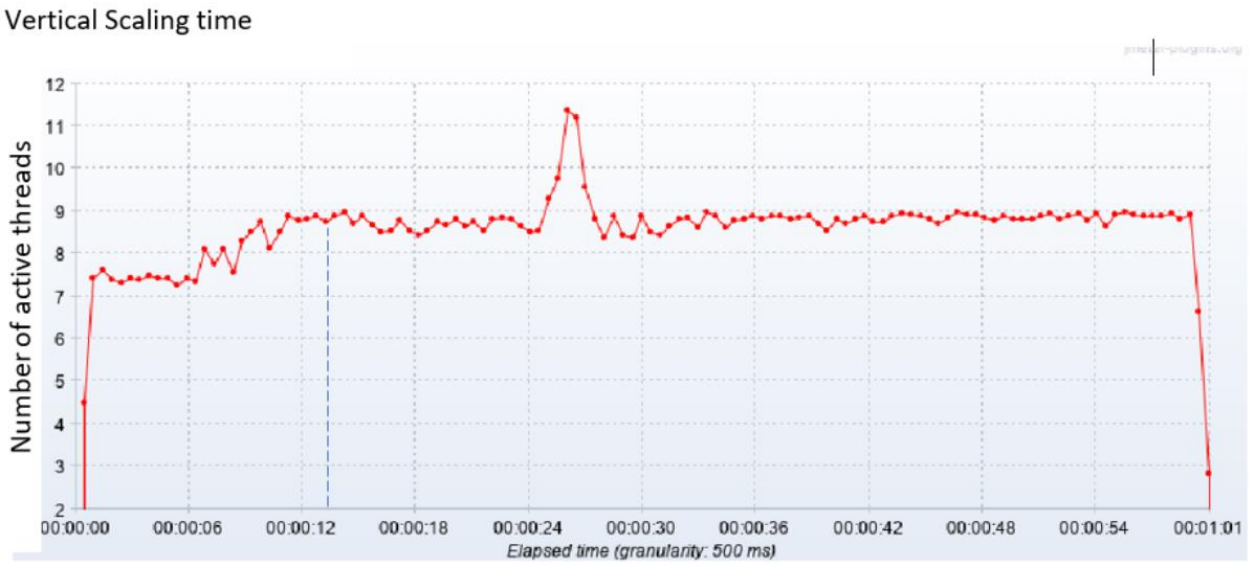

Figure 4. 1 number of active nodes vs time

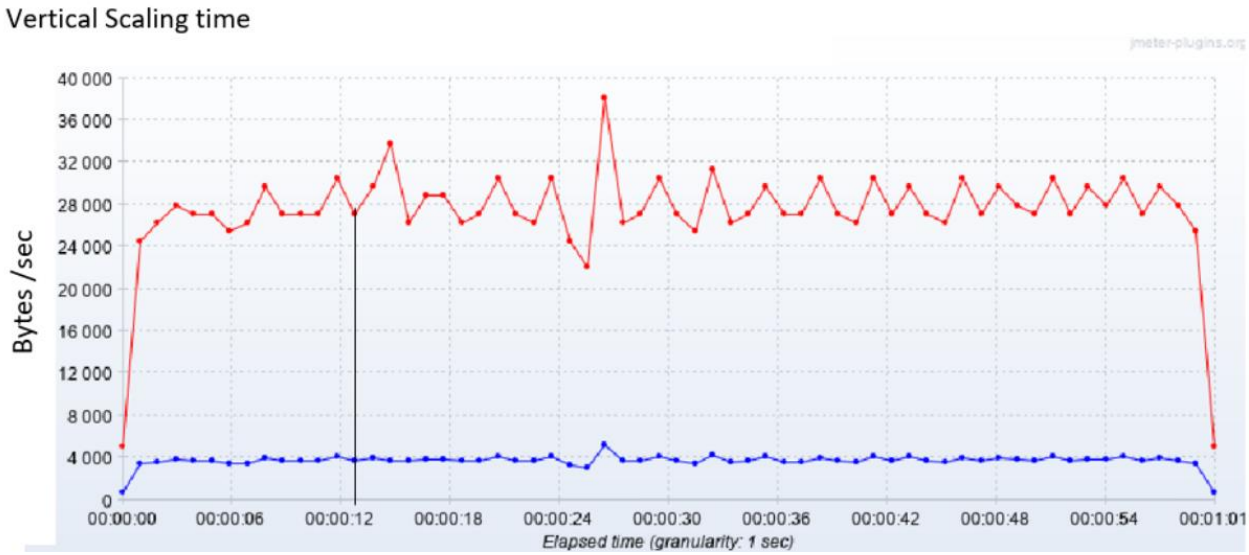

Figure 4. 2 bytes vs time

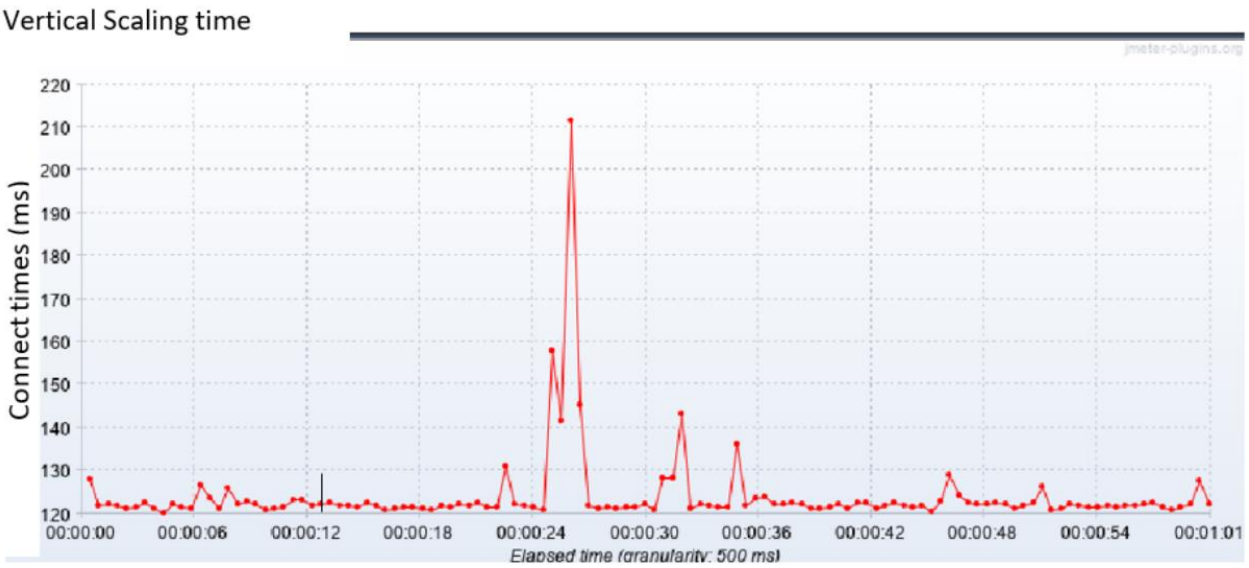

Figure 4. 3 connect times over time

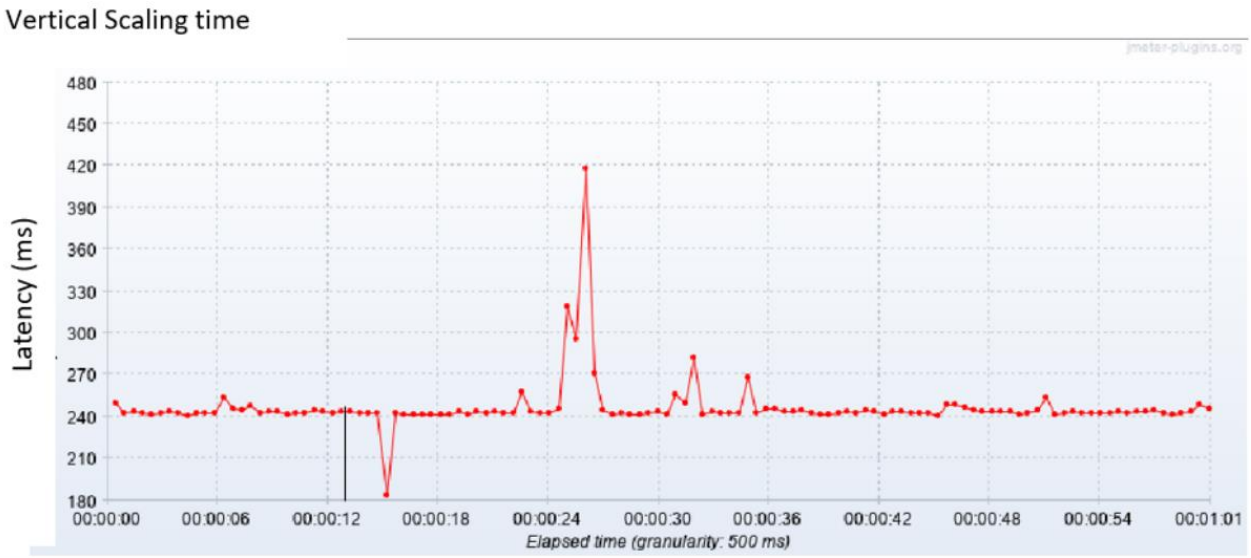

Figure 4. 4 latency vs time

from figure 4.4 At the moment of vertical scaling, or the 13th second, there is zero latency (or a rejected request). After that, the latency is not much affected except from the 24th to the 30th second. The rise in active threads over time throughout the (24th–30th) second period, as seen in figure 4.1, is the source of this increasing latency. A Jmeter plug-in "server hits per second" was set in order to confirm that the rise in latency is caused by the increase in the number of threads throughout the interval. This feature estimates how many requests are made of the service every second, and its use aims to determine if the volume of requests rose between the 24th and 30th seconds.

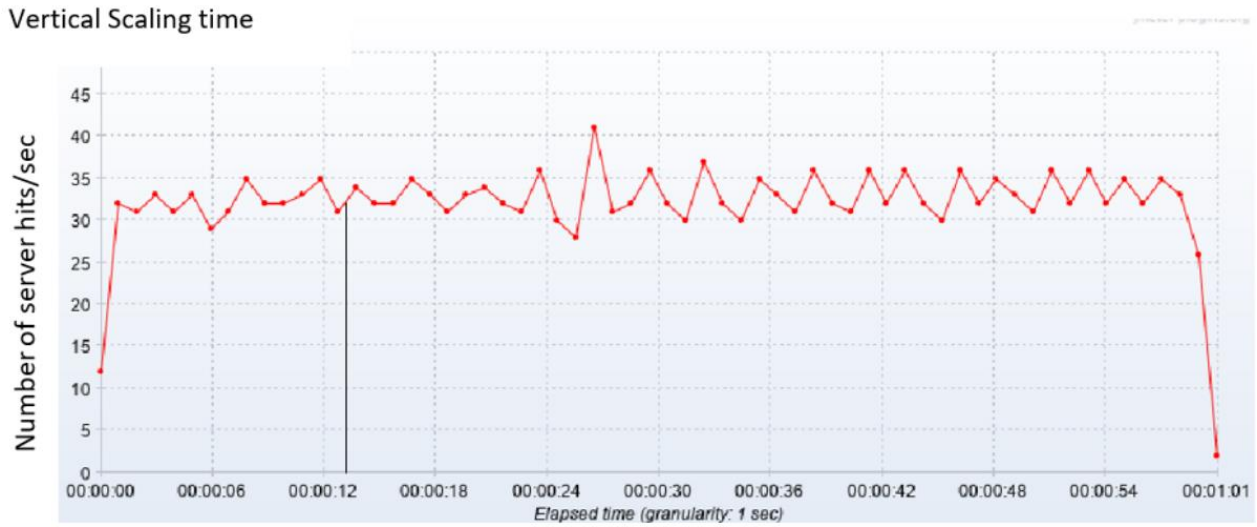

Figure 4. 5 number of servers hits vs time

No discernible difference in latency or connect time can be seen during the 13th second, which is the moment of vertical scaling.

4.1.2 Test Case 2

The second test had 500 threads (users), a loop count of 4, and a duration of 30 seconds. This test case simulates a heavier workload over a shorter period of time—30 seconds. At every sixth second during the 30 seconds of the stress test, the CPU share was vertically scaled from "100m" to "900m" to the container. The metric error% in table 5.2 indicates the proportion of unsuccessful requests. 8 samples (0.4%) out of 2000 samples were unsuccessful.

Label	Samples	Min response Time(ms)	Max. Response Time(ms)	Avg. Response Time(ms)	Standard Deviation	Error %	Throughput	Avg. Bytes
HTTP Requests	2000	231	23431	279	726.01	0.4	61.3/sec	855.7

Table 4.2 Data for the HTTP requests for test case 2

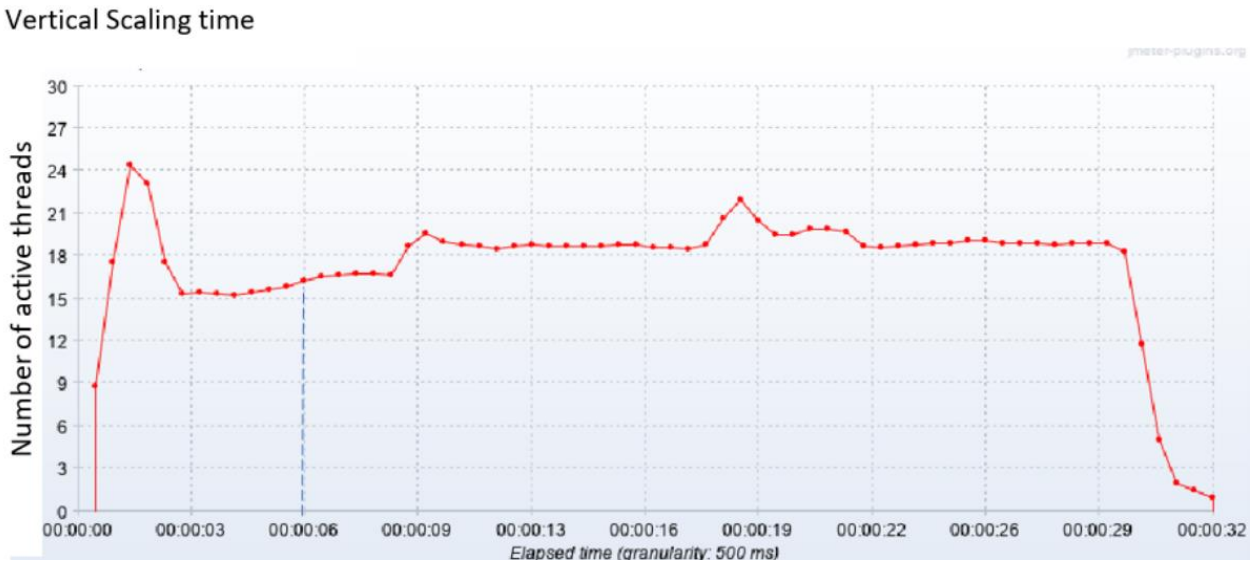

Figure 4. 6 number of active users vs time

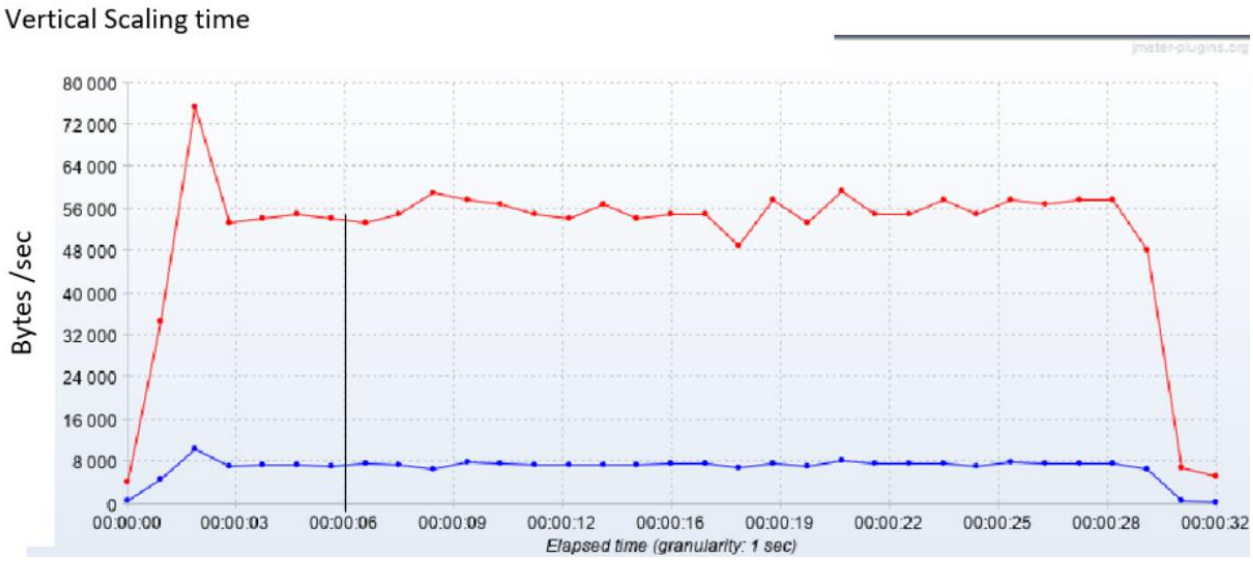

Figure 4. 7 bytes vs time

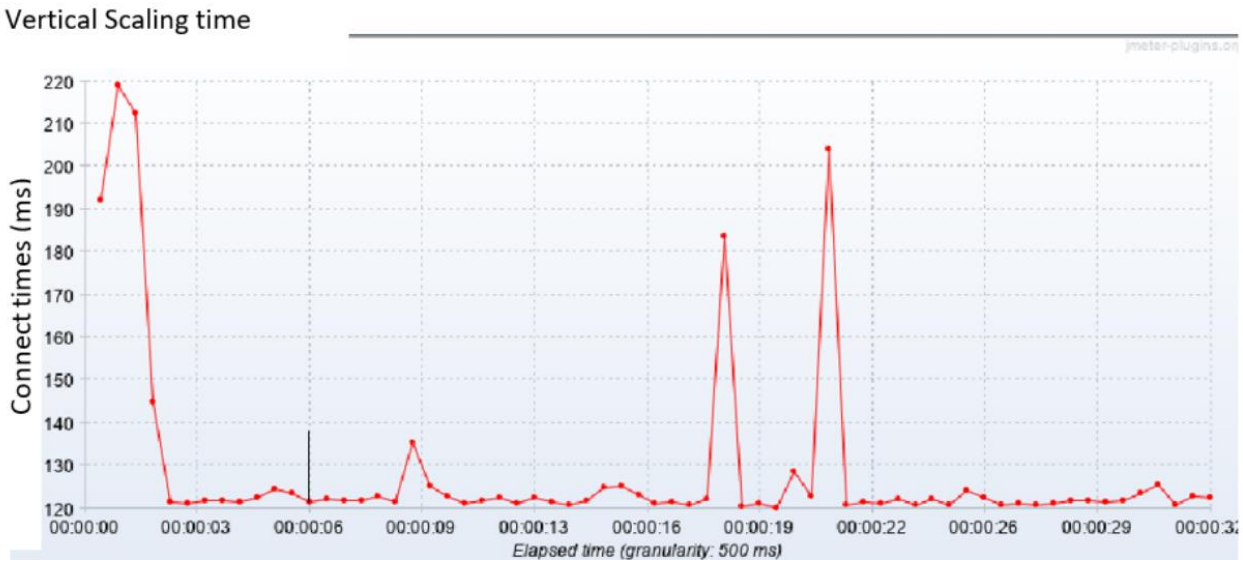

Figure 4. 8 connect times vs time

Figure 4. 9 latency(ms) vs time

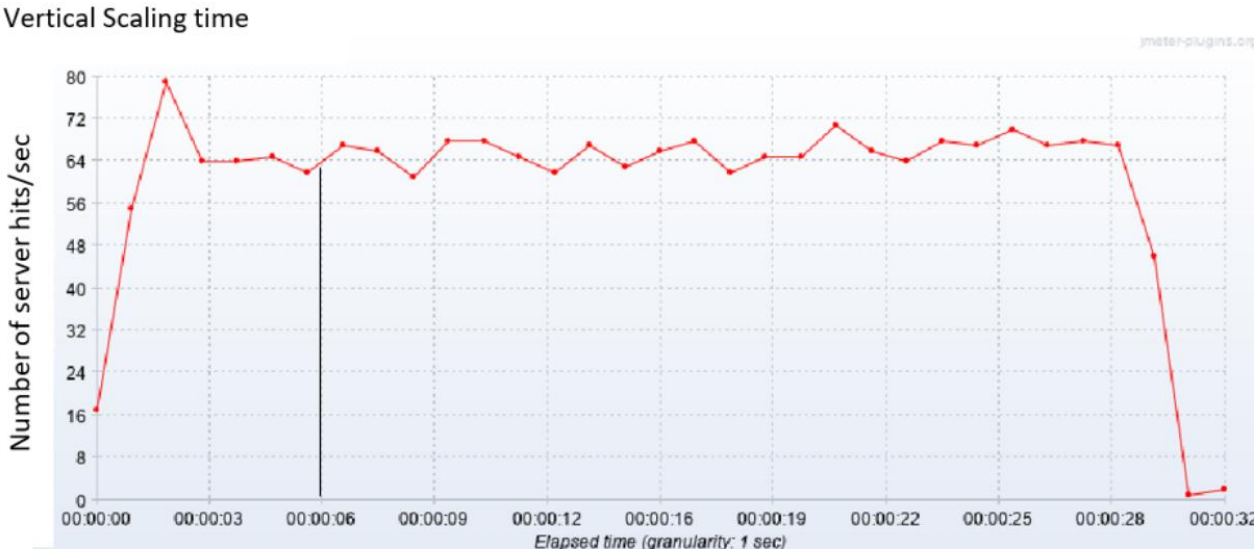

Figure 4. 10 number of server hits/sec vs time

as shown in the above figure 4.6 a spike in the number of threads to 24 before the third second and another increase to 21 threads between the 18th and 20th seconds. Figure 4.9 shows that there is also an increase in latency for the same intervals. No substantial changes in latency or connect time can be detected after the vertical scaling point, which is the sixth second. Later spikes in latency are addressed by an increase in the number of active threads, and this can also be confirmed by the number of server hits per second from figure 4.10.

4.2. Experiment 2 One Replica

The experiment was conducted in the same environment as previously explained in the early chapter. Jmeter is used as experiment 1 to measure the performance of the experiment.

4.2.1 Test Case 1

The first test had 500 threads (users), a loop count of 4, and a duration of 60 seconds. During the 60 second load test, the CPU share was vertically scaled from "100m" to "900m" at the 13th second. The metric error% in table 5.1 indicates the proportion of unsuccessful requests. 4 samples (0.2%) out of 2000 samples were unsuccessful.

Label	Samples	Min response Time(ms)	Max. Response Time(ms)	Avg. Response Time(ms)	Standard Deviation	Error %	Throughput	Avg. Bytes
HTTP Requests	2000	246	469	231	19.53	0.2	32.8/sec	852.3

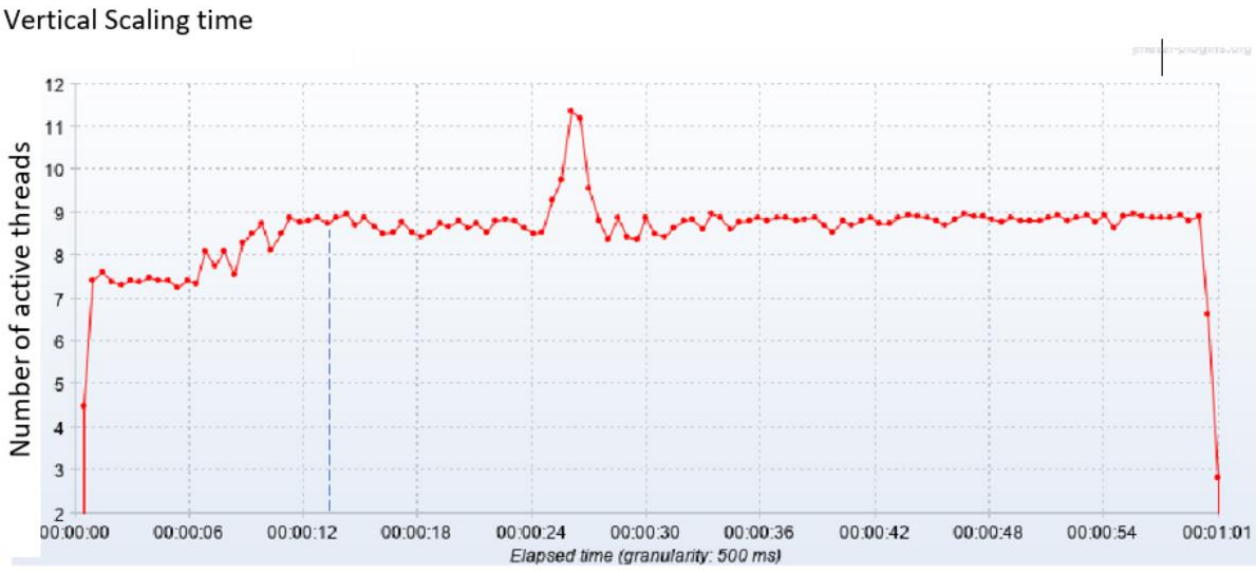

Figure 4. 11 Number of active nodes vs time

Figure 4. 12 byets vs time

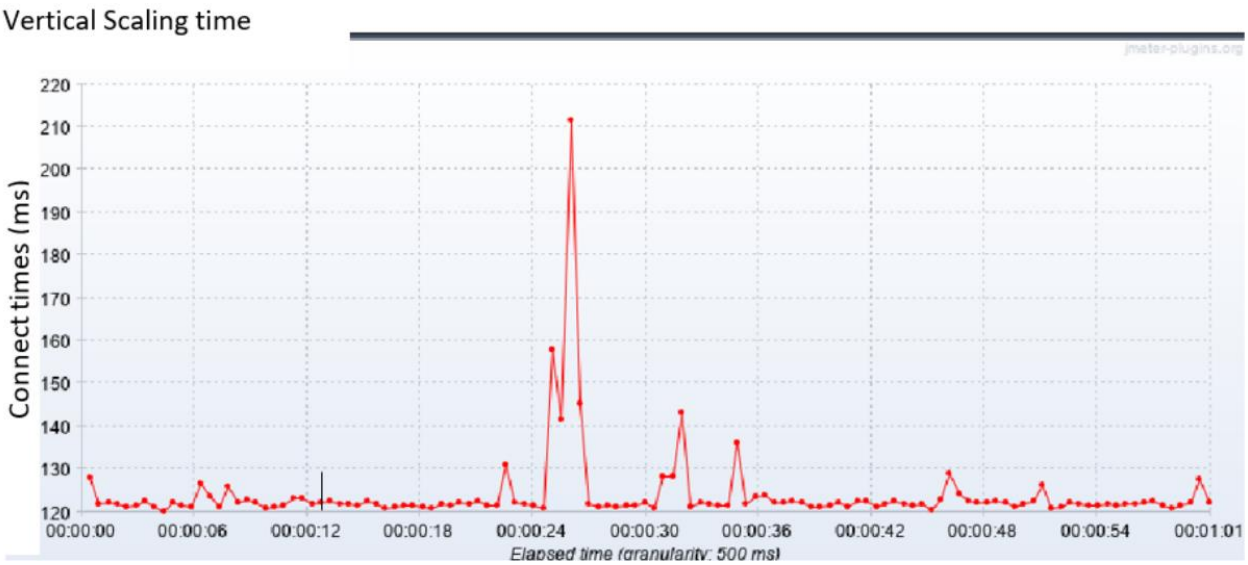

Figure 4. 13 connect times (ms) vs times

Figure 4. 14 latency (ms) vs time

In figure 4.14, the 13th second, or the time of vertical scaling, has a zero latency value (or a rejected request). After that, the latency is not much affected except from the 24th to the 30th second. The rise in active threads over time throughout the (24th-30th) second period, as seen in figure 5.1, is what causes this increased latency. A Jmeter plug-in "server hits per second" was set in order to confirm that the rise in latency is caused by the increase in the number of threads throughout the interval. This function estimates how many requests are made of the service each second, and its use attempts to determine if the volume of requests grew between the seconds (24th and 30th).

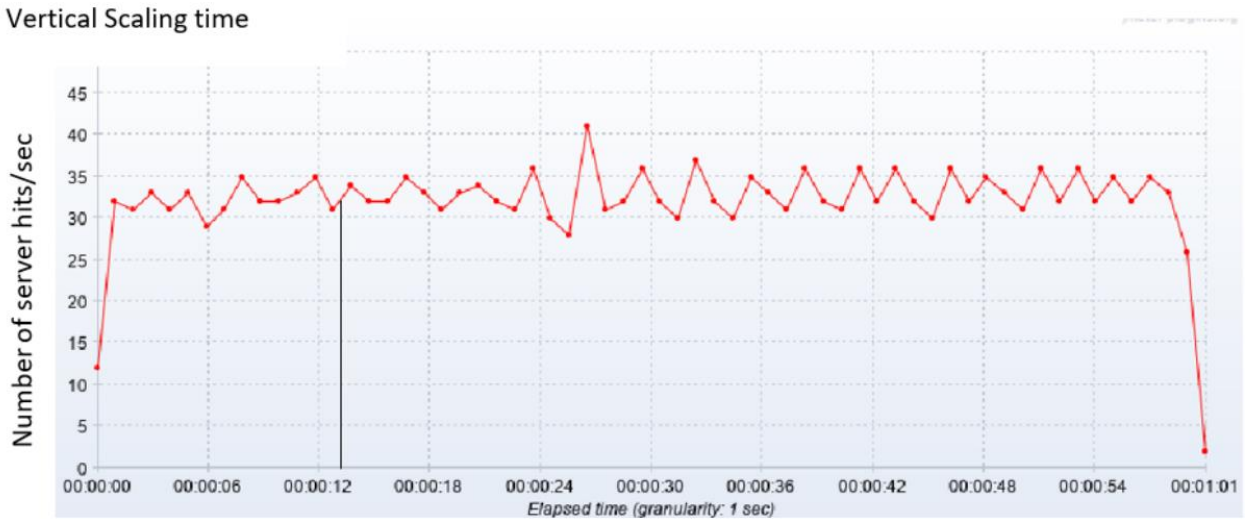

Figure 4. 15 number of server hits/sec vs time

No discernible difference in latency or connect time can be seen during the 13th second, which is the moment of vertical scaling.

4.2.2 Test Case 2

The second test had 500 threads (users), a loop count of 4, and a duration of 30 seconds. This test case simulates a heavier workload for a shorter period of time—30 seconds.. VDuring the 30 seconds of load testing, the CPU share to the container was erratically scaled from "100m" to "900m" every 6 seconds. The metric error% in table 5.2 indicates the proportion of unsuccessful requests. 8 samples (0.4%) out of 2000 samples were unsuccessful.

Label	Samples	Min response Time(ms)	Max. Response Time(ms)	Avg. Response Time(ms)	Standard Deviation	Error %	Throughput	Avg. Bytes
HTTP Requests	2000	231	26163	294	1118.41	0.45	56/sec	856.7

Figure 4. 16 number of active nodes vs time

Figure 4. 17 bytes vs times

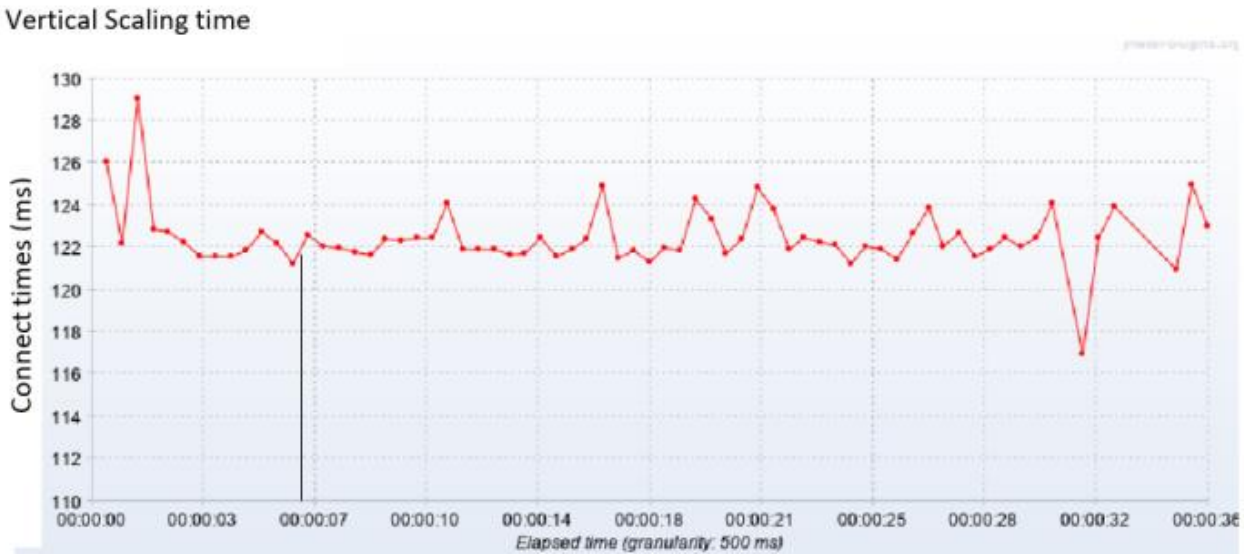

Figure 4. 18 connect times(ms) vs times

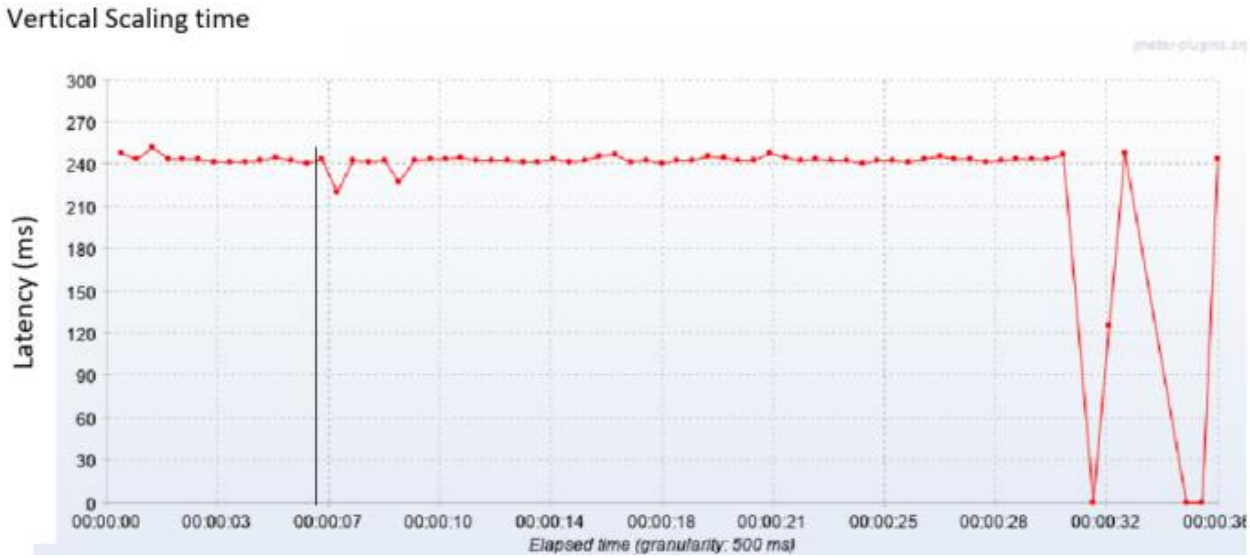

Figure 4. 19 latency (ms) vs time

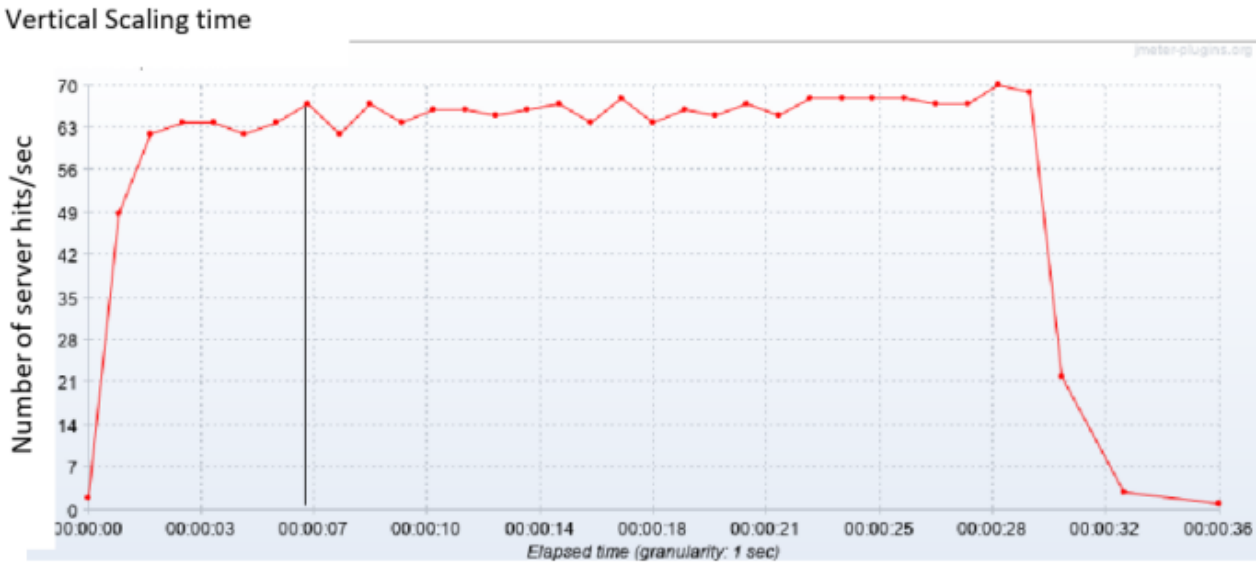

Figure 4. 20 number of servers hits vs time

4.3 Discussion on results

The buffer range of CPU shares when lowering a container's CPU shares is an intriguing finding during vertical scaling. A deployment is updated by Kubernetes by first producing a new pod and then deleting the previous pod. From an initial value of "900m," the CPU shares of the container in a single replica deployment were lowered to various levels. Only when the new CPU share number was less than or equal to "460m" could Kubernetes update the deployment.

Repeating this experiment with various beginning CPU shares. A new pod cannot be started by kubernetes to update the deployment and accommodate the modified container. The workload was gradually produced, and for the test scenarios with and without conducting vertical scaling, the measure "error%" was also analyzed. Additionally, progressive is the traffic rerouting from the load balancer to the pods. The percentage of unsuccessful HTTP requests is known as error%. The HTTP requests error% for single replica and two replica workload conditions are shown in Table 5.5. When vertical scaling is carried out, the data reveals a drop in error%, indicating that there is little downtime and that traffic is diverted to the newly produced pod. from the above experiments that we analyzed as follow

Experiment one: - After conducting vertical scaling, no appreciable impact on latency and connect times was seen. By analyzing the number of active threads throughout the timeframe, a spike in latency and connect times during that time period was addressed.

Experiment two: - After executing vertical scaling, there was no discernible impact on latency and connect times. In the initial test instance, a little increase in latency and connect times were seen over a period of time, but because so many threads were active, it was impossible to determine what caused the spike. During the 30 seconds of load testing for the second test scenario, no appreciable improvements in latency and connect times were seen.

4.4 Research questions and answers

RQ1: What performance metrics are affected the most when a pod on Kubernetes is vertically scaled?

Performance of the containerized application was examined under various workloads for various deployment circumstances. The performance indicators that were examined were latency and connect times. The application's latency and connect times were unaffected by vertical scaling. Thus, vertical scaling had no impact on any of the performance measures that were taken into consideration. Thus, it can be said that performing vertical scaling has no adverse effects on the performance of the containerized application running on Kubernetes in terms of latency and connect times.

RQ2: How can continuous delivery of service with vertical scaling be achieved on a Kubernetes-based Docker-containerized application?

As part of the literature review for this project, the architectures of kubernetes and docker containers were examined. A deployment in Kubernetes may be set to specify the resource limitations and requests for each container. Vertical scaling was carried out using this deployment capability as a baseline. A new pod with updated container resources is created as a result of updating this deployment by adding new resource specifications to it.

CHAPTER FIVE

5. CONCLUSIONS AND RECOMMENDATIONS

This thesis work provides an effective correction for resource allocation for applications running inside a Kubernetes cluster. We compare our design with the default Kubernetes Vertical Pod Autoscaler (VPA), which also aims at providing accurate resource allocation. This thesis provides many benefits over Kubernetes VPA by providing a non-disruptive method of auto-scaling, a more accurate estimation methodology which together improve the cluster CPU and memory utilization by 10%, and reduced runtime by 15%. We see an overhead for each application runtime in the range of 5% to 20%, but the applications have near accurate allocations which allows the cluster to be utilized by other applications. Our work also decreases the number of restarts over Kubernetes VPA. We also, quantify the overhead of container migration in our work at various stages of migration, conduct experiments to determine the best interval size for monitoring the container for the purpose of estimations, compare the allocation pattern of our work with Kubernetes VPA, and evaluate the overhead of In-Place scaling. Overall, even with grossly inaccurate allocations, our work is able to correct the allocations with much less number of corrections than Kubernetes default VPA while providing much improved cluster utilization.

At the end of the last Chapter, we only briefly mentioned the effects of restarting a pod to rescale its resource requests. To effectively utilize our proposed auto scaling strategy or any other vertical autoscaler in Kubernetes, we have to make sure the running application can handle restarts with minimal disruption. Currently, changing container resource requests in Kubernetes requires.
restarting the pod. This restart can be undesirable for certain stateful applications, as moving or copying existing state information could be expensive or unfeasible depending on the underlying implementation. Service state information may become temporarily inaccessible, and users trying to access the service may encounter non-trivial delays before getting back to expected performance. This was an issue that was also identified during our real-time tests. Also, excessive scaling might generate excessive load on Kubernetes components that are responsible for handling the pod restart. As we have seen from the test results, this autoscaling strategy manages to achieve similar or superior performance in terms of slack and insufficient CPU when compared to VPA. However, more frequent re-scaling is inevitable as we try to adapt to the workload in a fine-grained manner. We have, however, implemented parameters that can limit the frequency of re-scaling and avoid

unnecessary restarts, by tuning these parameters, it is potentially possible to optimize the algorithms for the constraints on re-scaling frequency. As such, to effectively apply predictive autoscaling in real Kubernetes
applications, further research will be needed to understand how the frequency of restarts can affect pods.

Special acknowledgment

This research project is funded by Adama Science and Technology University under the grant number:

ASTU/SM-R/507/22

Adama, Ethiopia

REFERENCES

-Lukman Ibrahim Isa, S. A. R. J. (2021). IRJET- optimizing Multimedia Processing over Cloud Cluster using Modern Container and Container Orchestration Technology. *Irjet*, *8*(1), 298–303.

Ahmad, I., AlFailakawi, M. G., AlMutawa, A., & Alsalman, L. (2021). Container scheduling techniques: A Survey and assessment. *Journal of King Saud University - Computer and Information Sciences*, *xxxx*. https://doi.org/10.1016/j.jksuci.2021.03.002

Al-Dhuraibi, Y., Paraiso, F., Djarallah, N., & Merle, P. (2017). Autonomic Vertical Elasticity of Docker Containers with ELASTICDOCKER. *IEEE International Conference on Cloud Computing, CLOUD*, *2017-June*, 472–479. https://doi.org/10.1109/CLOUD.2017.67

Arfamaini, R. (2016). No 主観的健康感を中心とした在宅高齢者における 健康関連指標に関する共分散構造分析Title. *Applied Microbiology and Biotechnology*, *85*(1), 2071–2079.

Becker, F. G., Cleary, M., Team, R. M., Holtermann, H., The, D., Agenda, N., Science, P., Sk, S. K., Hinnebusch, R., Hinnebusch A, R., Rabinovich, I., Olmert, Y., Uld, D. Q. G. L. Q., Ri, W. K. H. U., Lq, V., Frxqwu, W. K. H., Zklfk, E., Edvhg, L. V, Wkh, R. Q., … ح ,فاطمی. (2015). CPU Autoscaling in Cloud Computing Environments. *Syria Studies*, *7*(1), 37–72. https://www.researchgate.net/publication/269107473_What_is_governance/link/548173090cf22525dcb61443/download%0Ahttp://www.econ.upf.edu/~reynal/Civil wars_12December2010.pdf%0Ahttps://think-asia.org/handle/11540/8282%0Ahttps://www.jstor.org/stable/41857625

Donca, I. C., Corches, C., Stan, O., & Miclea, L. (2020). Autoscaled RabbitMQ Kubernetes Cluster on single-board computers. *2020 22nd IEEE International Conference on Automation, Quality and Testing, Robotics - THETA, AQTR 2020 - Proceedings*. https://doi.org/10.1109/AQTR49680.2020.9129886

Du, L., Wo, T., Yang, R., & Hu, C. (2018). Cider: A Rapid Docker Container Deployment System through Sharing Network Storage. *Proceedings - 2017 IEEE 19th Intl Conference on High Performance Computing and Communications, HPCC 2017, 2017 IEEE 15th Intl Conference on Smart City, SmartCity 2017 and 2017 IEEE 3rd Intl Conference on Data Science and Systems, DSS 2017*, *2018-Janua*, 332–339. https://doi.org/10.1109/HPCC-SmartCity-DSS.2017.44

Elamin, M., & Paardekooper, P. (2021). *Scaling of containerized network functions*. 1–24.

Evangelidis, A., Parker, D., & Bahsoon, R. (2017). Performance Modelling and Verification of Cloud-Based Auto-Scaling Policies. *Proceedings - 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2017*, 355–364. https://doi.org/10.1109/CCGRID.2017.39

Henschel, J. (2021). *Dimensioning , Performance and Optimization of Cloud-native Applications*.

Horne, R. (2015). Embracing global computing in emerging economies: First workshop, EGC 2015 Almaty, Kazakhstan, february 26–28, 2015 proceedings. In *Communications in Computer and Information Science* (Vol. 514).

Khanh, H., Eds, D., & Hutchison, D. (2015). *Service-Oriented Computing*.

Kovács, J., Kacsuk, P., & Emődi, M. (2018). Deploying Docker Swarm cluster on hybrid clouds using Occopus. *Advances in Engineering Software*, *125*(September 2017), 136–145. https://doi.org/10.1016/j.advengsoft.2018.08.001

Kubernetes.io. (n.d.). *Horizontal Pod Autoscaler*. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

Kubernetes.io. (2021). *"Deployments."* https://kubernetes.io/docs/concepts/%0Aworkloads/controllers/deployment/%0A

Kubernetes. (2021a). Managing Compute Resources for Containers. In *Kuberentes Documentation*. https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container/

Kubernetes, E. (n.d.). Deployment. In *Online*. https://kubernetes.io/docs/concepts/workloads/pods/

Kubernetes, E. (2021b). *Configuring multidimensional Pod autoscaling*. https://cloud.google.com/kubernetes-engine/docs/how-%0Ato/multidimensional-pod-autoscaling

Kubernetes, E. (2021c). *Kubernetes. Vertical Pod Autoscaler*. https:%0A//github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler%0A

Kubernetes, E. (2022). *Pods*. https://cloud.google.com/%0Akubernetes-engine/docs/concepts/deployment

M. Siarkowicz, D. G. Qian Chenglong, A., & Junchen, T. (n.d.). "Metrics server FAQ,." In *online*. https://github.com/kubernetes-sigs/metrics-server/blob/%0A4fdff44db13551696f2ddcd5f6ed9c88cb859ecd/FAQ.md

Michael Handa. (2020). *How to Scale Using Kubernetes: From Startup to Superstar*. https://bluesentryit.com/how-to-scale-using-kubernetes-%0Afrom-startup-to-superstar/

Midigudla, D. (2019). *Master of Science in Telecommunication Systems Performance Analysis of the Impact of Vertical Scaling on Application Containerized with Docker Kubernetes on Amazon Web Services EC2* (Issue June). www.bth.se

Najdataei, H., Gulisano, V., Papadopoulos, A. V, Walulya, I., Papatriantafilou, M., & Tsigas, P. (2020). Performance Modeling and Vertical Autoscaling of Stream Joins. *ArXiv Preprint ArXiv:2005.04935*.

Nguyen, N. D., & Kim, T. (2021). Balanced leader distribution algorithm in kubernetes clusters. *Sensors (Switzerland)*, *21*(3), 1–15. https://doi.org/10.3390/s21030869

Nguyen, T. T., Yeom, Y. J., Kim, T., Park, D. H., & Kim, S. (2020). Horizontal pod autoscaling in kubernetes for elastic container orchestration. *Sensors (Switzerland)*, *20*(16), 1–18. https://doi.org/10.3390/s20164621

Rattihalli, G., Govindaraju, M., Lu, H., & Tiwari, D. (2019). Exploring potential for non-disruptive vertical auto scaling and resource estimation in kubernetes. *IEEE International Conference on Cloud Computing, CLOUD*, *2019-July*, 33–40. https://doi.org/10.1109/CLOUD.2019.00018

Rodriguez, M. A., & Buyya, R. (2019). Container-based cluster orchestration systems: A taxonomy and future directions. *Software - Practice and Experience*, *49*(5), 698–719. https://doi.org/10.1002/spe.2660

Ross, I. M., Gong, Q., Karpenko, M., & Proulx, R. J. (2018). Scaling and balancing for high-performance computation of optimal controls. *Journal of Guidance, Control, and Dynamics*, *41*(10), 2086–2097. https://doi.org/10.2514/1.G003382

Rzadca, K., Findeisen, P., Swiderski, J., Zych, P., Broniek, P., Kusmierek, J., Nowak, P., Strack, B., Witusowski, P., Hand, S., & Wilkes, J. (2020). Autopilot: Workload autoscaling at Google. *Proceedings of the 15th European Conference on Computer Systems, EuroSys 2020*. https://doi.org/10.1145/3342195.3387524

Shah, J., & Dubaria, D. (2019). Building modern clouds: Using docker, kubernetes google cloud platform. *2019 IEEE 9th Annual Computing and Communication Workshop and Conference, CCWC 2019*, 184–189. https://doi.org/10.1109/CCWC.2019.8666479

Terracciano, L., & Hu, D. Y. X. (2021). *Fast and fine-grained resource allocation for cloud-native applications on Kubernetes*. https://www.politesi.polimi.it/handle/10589/174028

Verreydt, S., Beni, E. H., Truyen, E., Lagaisse, B., & Joosen, W. (2019). Leveraging Kubernetes for adaptive and cost-efficient resource management. *WOC 2019 - Proceedings of the 2019*

5th International Workshop on Container Technologies and Container Clouds, Part of Middleware 2019, 37–42. https://doi.org/10.1145/3366615.3368357

Wang, T. (2021). *Predictive vertical CPU autoscaling in Kubernetes based on time-series forecasting with Holt-Winters exponential smoothing and long short-term memory*.

Wong, A., Rexachs, D., & Luque, E. (2015). for Performance Analysis and Prediction. *IEEE Transactions on Parallel and Distributed Systems*, *26*(7), 2009–2019.