Dissertation on

"Deploying EOX microservice to Kubernetes Cluster"

Submitted in partial fulfilment of the requirements for the award of degree of

Bachelor of Technology
in
Computer Science & Engineering

UE20CS390B – Capstone Project Phase - 2

Submitted by:

Yuvaraj D C	**PES1UG20CS521**
Adarsh Kumar	**PES2UG20CS016**
Suchit S Kallapur	**PES1UG20CS438**
Veena Garag	**PES1UG20CS492**

Under the guidance of

Prof. Venkatesh Prasad
Professor
Department of Computer Science
PES University

August - December 2023

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
FACULTY OF ENGINEERING
PES UNIVERSITY
(Established under Karnataka Act No. 16 of 2013)
100ft Ring Road, Bengaluru – 560 085, Karnataka, India

PES UNIVERSITY

(Established under Karnataka Act No. 16 of 2013)
100ft Ring Road, Bengaluru – 560 085, Karnataka, India

FACULTY OF ENGINEERING

CERTIFICATE

This is to certify that the dissertation entitled

‘Deploying EOX microservice to Kubernetes cluster’

is a bonafide work carried out by

Yuvaraj D C	**PES1UG20CS521**
Adarsh Kumar	**PES2UG20CS016**
Suchit S Kallapur	**PES1UG20CS438**
Veena Garag	**PES1UG20CS492**

in partial fulfilment for the completion of seventh semester Capstone Project Phase - 2 (UE20CS390B) in the Program of Study - Bachelor of Technology in Computer Science and Engineering under rules and regulations of PES University, Bengaluru during the period August - December 2023. It is certified that all corrections / suggestions indicated for internal assessment have been incorporated in the report. The dissertation has been approved as it satisfies the 7th semester academic requirements in respect of project work.

Venkatesh Prasad	Dr. Mamatha H R	Dr. B K Keshavan
Associate Professor	Chairperson	Dean of Faculty

External Viva

Name of the Examiners	**Signature with Date**
1. _____	_____
2. _____	_____

DECLARATION

We hereby declare that the Capstone Project Phase - 2 entitled **"Deploying EOX microservice to Kubernetes Cluster"** has been carried out by us under the guidance of **Prof Venkatesh Prasad,** Asst. Prof, PES University and submitted in partial fulfilment of the course requirements for the award of degree of **Bachelor of Technology** in **Computer Science and Engineering** of **PES University, Bengaluru** during the academic semester August - December 2023. The matter embodied in this report has not been submitted to any other university or institution for the award of any degree.

PES1UG20CS521	**Yuvaraj D C**
PES2UG20CS016	**Adarsh Kumar**
PES1UG20CS438	**Suchit S Kallapur**
PES1UG20CS492	**Veena Garag**

ACKNOWLEDGEMENT

We would like to express our gratitude to Prof. Venkatesh Prasad, Department of Computer Science and Engineering, PES University, for his continuous guidance, assistance, and encouragement throughout the development of this UE20CS390B - Capstone Project Phase – 2.

We are grateful to the project coordinator, Dr. Priyanka H., all the panel members & the supporting staff for organizing, managing, and helping with the entire process.

We take this opportunity to thank Dr. Mamatha H. R., Chairperson, Department of Computer Science and Engineering, PES University, for all the knowledge and support we have received from the department. We would like to thank Dr. B.K. Keshavan, Dean of Faculty, PES University for his help.

We are deeply grateful to Dr. M. R. Doreswamy, Chancellor, PES University, Prof. Jawahar Doreswamy, Pro Chancellor – PES University, Dr. Suryaprasad J., Vice-Chancellor, PES University, for providing us various opportunities and enlightenment every step of the way.

Finally, this project could not have been completed without the continual support and encouragement we have received from my family and friends.

ABSTRACT

The software engineering industry is rapidly embracing containers to deploy microservice based cloud-native services. When an application has massive workload, Kubernetes can automatically scale the microservices using the default Kubernetes Horizontal Pod Autoscaler. The default Kubernetes approach results in inefficient resource allocation, which affects cloud-native application performance and increases maintenance expenses. This project uses a custom controller algorithm and estimates the appropriate number of instances for containers. The suggested approach maintains cloud-native applications' Quality of Service (QoS) while cutting down on maintenance expenses. This project found that on comparison the default Kubernetes Horizontal Pod Autoscaler is more expensive and scales up and down very inefficiently and is more expensive than the custom controller. Project deals with deploying microservices on cloud and reduce the resource consumption like CPUs, memory, and bring down the cost of creation of pods.

The project is aimed to improve and optimize resource utilization, improve security feature and be able to monitor the performance and visualize the metrics to study the behaviour of the microservices to increased load.

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

Microservices architecture is an approach where an application is divided into smaller, loosely coupled services that can be developed, deployed, and scaled independently. This gives us a lot of advantages like scalability, resilience, agility and coping up with growing technology.

These microservices handle specific business functionality and communicate with other microservices via API to serve the request better.

They are easy to maintain and deploy. They can be containerized with all the necessary runtime environments and could be run on any platform. This feature is very much useful when it comes to distributed architecture.

In this report, we will delve into the implementation of a custom controller auto-scaler within a Kubernetes environment to dynamically scale our microservices-based application. This approach leverages the strengths of both Kubernetes and microservices to achieve seamless scalability and ensure optimal performance under varying load conditions.

2. PROBLEM STATEMENT

Project deals with deploying EOX microservices to Kubernetes clusters. Develop architecture for deploying finished containers on Kubernetes. This has to reduce the life cycle of deployment and management of the containers and improve scalability and performance.

3. LITERATURE REVIEW

3.1. Research Paper - 1

"Autoscaling Cloud-Native Applications using Custom Controller of Kubernetes"

3.1.1 Introduction

The default Kubernetes method results in ineffective resource allocation, declining performance, and increased maintenance costs. As a result, they created the new algorithm, known as Custom Controller. Default Algorithm is more costly than Custom algorithm. Using the Custom Controller nearly lowered the maintenance costs in half. In accordance with workload, Custom Controller manages and scales Pods dynamically. Pods will be scheduled in accordance with the volume of requests made to the server. For this investigation, Amazon Web Services' t3.2xlarge instance was employed. The application based on microservice was developed in PHP to do the computationally demanding tasks.

The following software and hardware were employed in this study's research: Kubernetes, Kubeadm, Kubectl, Docker, and PHP-Apache.

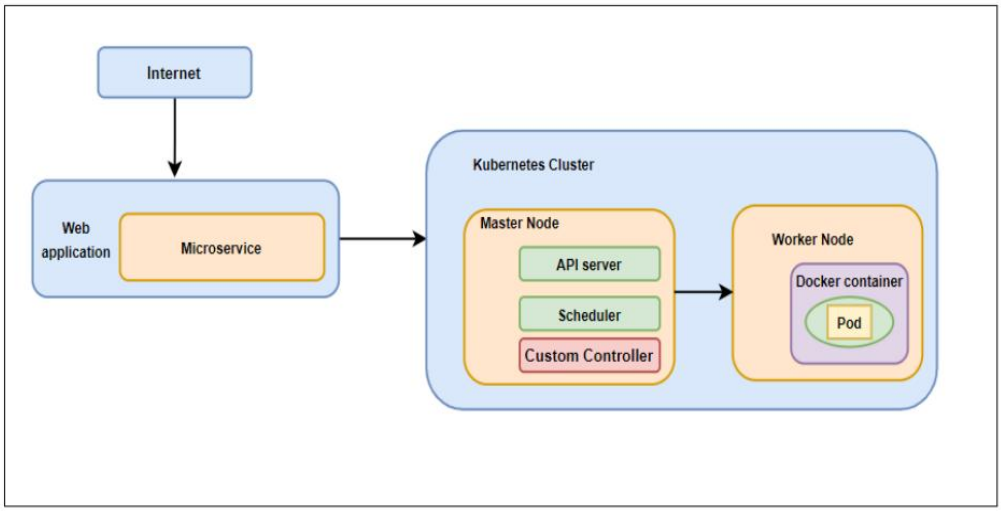

figure 1: Kubernetes Cluster

3.1.2 Methodology

The system consists of one master node and two worker nodes. In this study, an effort is made to create a custom controller to calculate the necessary number of pods to prevent resource waste and website crashes. The Kube-controller manager component is present. The Kubernetes cluster uses YAML. The parameters in the YAML file are sent to the Kube-API server and then to the Kube-controller management to control replicas. The custom controller is then informed to start or create new pods by the Kube-API server. Given below is the algorithm proposed in the research paper:

Algorithm 1 *Autoscaling Algorithm for resource allocation.*

Input: Total_Pods,Total_CPU_Usage_Value,Total_CPU_target_value;
Output: $Total\_Pods_n$ = Total number of pod to be scheduled.
Total_Pods; = sum(pod0,pod1,.....Podn)); // Calculates the total number of pods running in cluster
Size_of_cluster = Total_Pods.length;
Total_CPU_target_value = fetch_target_CPU(); // API call for fetching the target CPU
Total_CPU_Usage_Value = fetch_current_usage(); // API call for fetching the current CPU usage.
if *Size_of_cluster > 0 && Total_CPU_Usage_Value > (Size_of_cluster *Total_CPU_target_value)* **then**
  for *i in Total_Pods* **do**
    $Total\_Pods_n$ = *Total_CPU_Usage_Value / Total_CPU_target_value* // Calculate the total number of pods.
  end
end

Figure 2: custom control algorithm

3.1.3 Experiment

Three trials totalling two minutes, eight minutes, and fifteen minutes were carried out. The performance of the Custom Controller and the default Kubernetes algorithm were compared. The results showed that KHPA allotted 3 pods within the first two minutes of workload while the custom controller allocated 0 pods. In contrast to KHPA, which planned 7 pods after 8 minutes of workload creation, Custom controller only scheduled 2 pods, overprovisioning resources by 4 pods. Performance suffered because KHPA only scheduled 1 pod during the 15 minutes of workload production while Custom Controller scheduled 5.

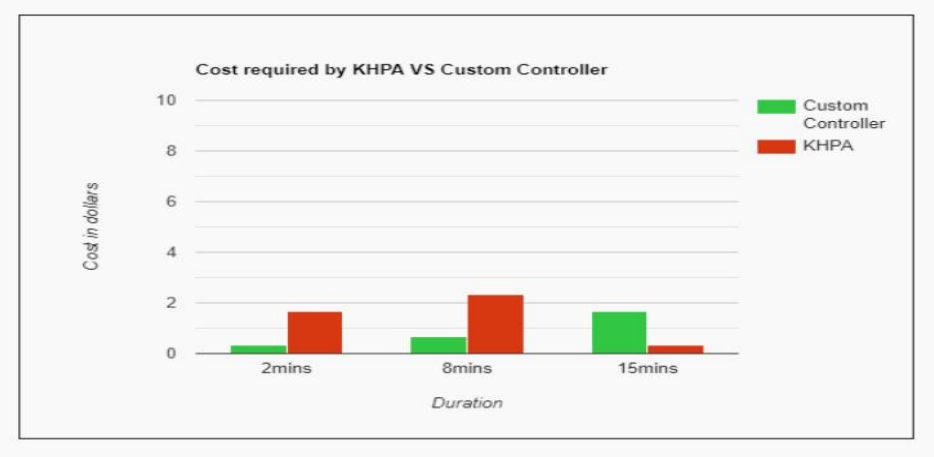

Figure 3: Cost Comparison between custom controller Vs. KHPA

3.1.4 Conclusion

By implementing a custom controller algorithm which calculates the efficient number of pods, this paper attempts to solve the issue of inefficient pod allocation.

3.1.5 Future Work

The future work suggested by this paper is the above algorithm can be improved for memory and storage intensive microservices. It is found that replicas take time to get stabilized this issue can be considered.

3.2 Research Paper – 2

“Dynamic Load Balancing of Microservices in Kubernetes Clusters using Service Mesh (2022)”

3.2.1 Introduction

This paper suggests a technique that applies service-specific routing across the Istio control plane to inject sidecar proxies onto every micro-service using service-mesh Istio and dynamically balancing the load among services. Due to its static nature, the default Kubernetes load balancing strategy performs poorly as the workload on the application grows and is unable to handle the fluctuating traffic. Encrypting the communications between services with mTLS ensures the security of inter-service communication.

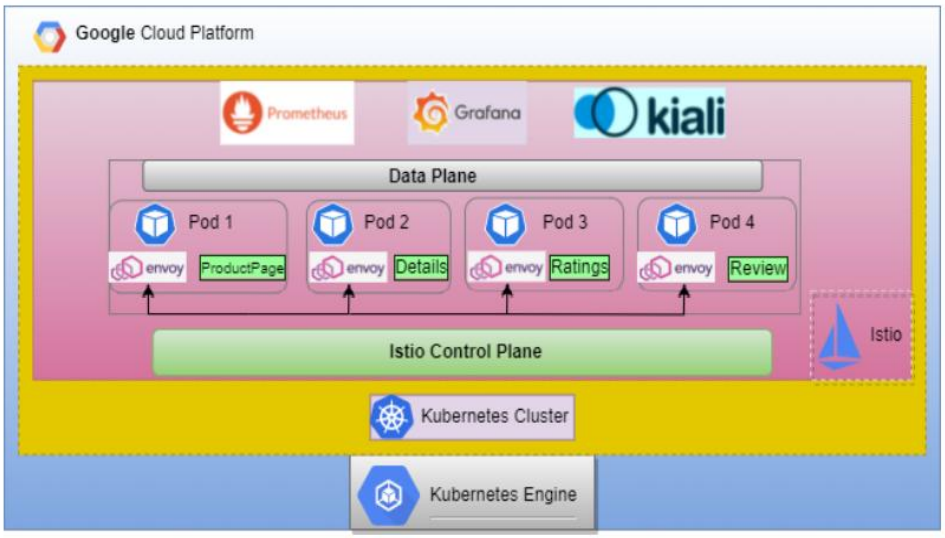

Figure 4: An example of a High-Level Design of proposed model

3.2.2 Methodology

The accompanying diagram shows the order in which the planned system was implemented. Istio configurations are kept apart using the Istio-system namespace. On the ingress-gateway for this namespace, the network configurations (YAML file) are applied. The Istio-ingress gateway is configured using the ingress host, ingress port, and secure ingress port. By dispersing the incoming load among the many application services, this gateway serves as a crucial load balancer. In Google Cloud Platform, firewall rules are created to permit the ingress port and secure ingress port that the Istio-ingress gateway will use. Envoy proxies are installed as sidecars to each pod that runs a service. The entire set of security policies and dynamic routing must be applied to each service is included in a YAML file with the titles Virtual Service and Enable mTLS, respectively. These policies are then applied to each sidecar found in each pod utilising a control plane. The locust tool causes load on the bookseller application.

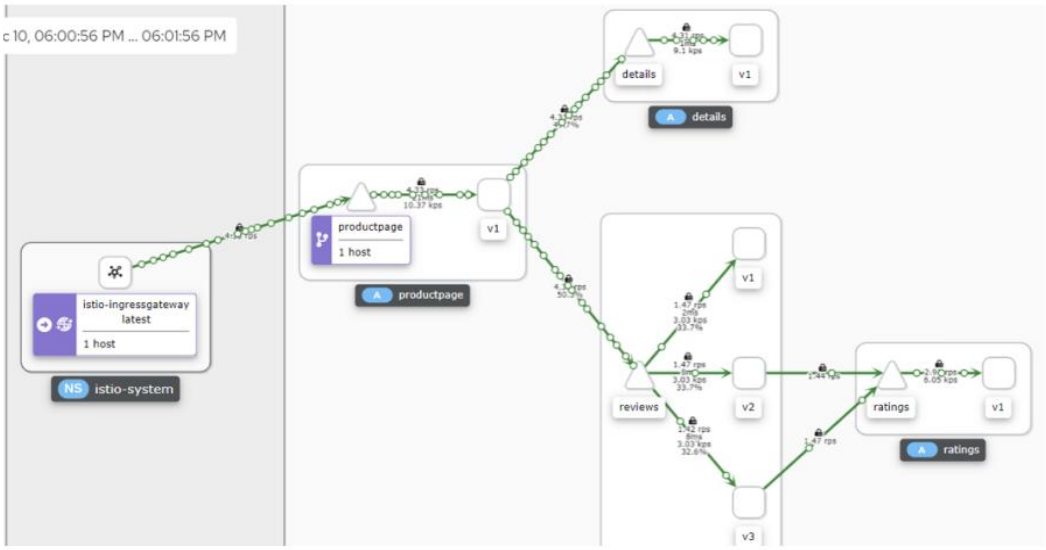

Figure 5: Traffic Visualization through Kiali Software

3.2.3 Experiment

The studies involved producing load for five minutes under three distinct circumstances (as stated in table below). They simulated user traffic and the number of incoming traffic load per second on the microservices using the load testing tool locust. With the aid of Kiali and Grafana, metrics including response time, CPU usage, and memory usage were monitored and noted.

Scenario	Total Users	Requests per second
1	1000	100
2	3000	300
3	5000	500

Figure 6: Scenario of Users and Requests per second

According to the aforementioned experiments, the response time between the default Kubernetes-based system and the proposed Istio-based system is almost identical. when there are fewer users and incoming requests per second on the application (case 1) for both models. When the traffic load gradually increases (case 2 3), the default Kubernetes system struggles to perform well.

But the proposed Istio-based system performs greatly by maintaining stability compared to the Kubernetes default system.

3.2.4 Conclusion

This experiment suggests a service-mesh Istio-based system that effectively manages the various dynamic load balancing on a web application built with a microservices architecture and deployed on a Kubernetes cluster by speeding up response times and using fewer resources than a default Kubernetes system.

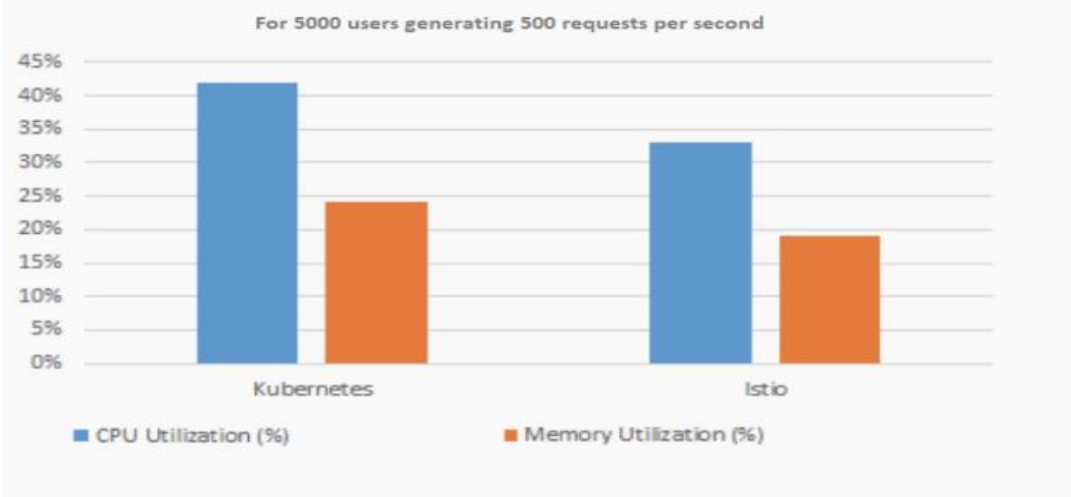

Figure 7: CPU and Memory Utilization in Kubernetes vs Istio

3.2.5 Future Work

Further research to reduce the latency and complexity can be done

3.3 Research Paper-3

“An Efficient and Scalable Traffic Load Balancing Based on Web Server Container Resource Utilization using Kubernetes Cluster (2022)”

3.3.1 Introduction

Because the microservices that are running on the containers don't communicate with one another, managing the containers is a constant challenge for any business. They operate separately and independently. Here, Kubernetes enters the picture. All that Kubernetes is a platform for managing containers.

3.3.2 Methodology

Horizontal Container Scaling Algorithm:

Desired App Replicas = ceil [current App Replicas* (current Scaling Metric Value / desired Scaling Metric Value)]

3.3.3 Experiment

In this experiment was conducted with a given target CPU usage value of 70%, and a minimum pod count of 4 with a maximum pod count of 20. Current status of deployment is six pods averaging

95% usage.

Desired Replicas calculated = ceil [6*(85/70)]

= ceil (8.14) = 9

Scaling down example

The next scenario is with a given target CPU usage value of about 70%, minimum pod count was given as 4, and maximum pod count was given of value 20, current status of application pod deployment is that here are 10 total pods. Given that 10 pods averaging 45% usages. When we use the same algorithm to scale down the pods we get:

Desired Replicas count = ceil [10*(45/70)] = ceil (6.42) = 7

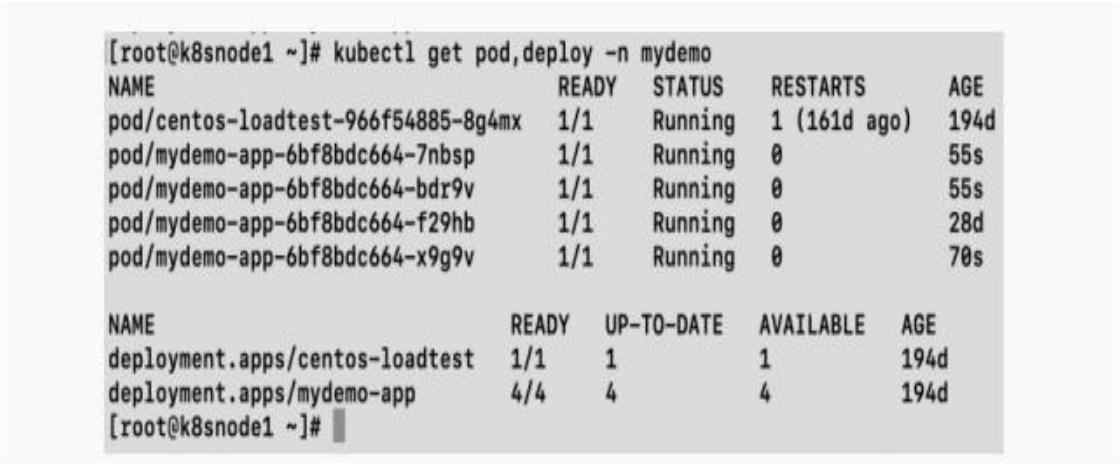

Figure 8: Terminal view for Number of pod scale after web traffic load increased

Figure 9: Terminal view for Number of pods that get auto deployed after load increased

3.3.4 Conclusion

According to user demand and traffic demand, the article proposed a dynamic scaling technique that takes backend delay into account for platforms and apps with high traffic levels.

3.4 Research Paper-4

“Managing Multi-Cloud Deployments on Kubernetes with Istio, Prometheus and Grafana (2022)”

3.4.1 Introduction

deploying and managing clusters across multi public clouds has a great advantage of harnessing the prominent features provided by each cloud service provider. It gives us the negotiating power and reduces single vendor dependency.

when the project requirement is not met by the features provided by a single vendor, in such cases we go for a multi-vendor system or multi public cloud platform. but implementing such architecture becomes difficult.

3.4.2 Implementation

Anthos which is implemented using Istio open-source project implemented by google is a milestone that has made this possible.

We can have many clusters on various public cloud providers be it an AWS, google cloud platform and Istio control plane using which we can inject envoy proxies and form a service mesh of interconnected clusters.

These clusters can communicate with each other using a VPN tunnel. here Kubernetes plays a major role in scaling, pod allocation and deployment.

Istio is going to take care of traffic management and observability and security issues the implementation makes use of various technologies such as Kubernetes, Istio, Grafana, Prometheus Grafana is a metric analysis and visualization tool along with-it Prometheus is also a very useful tool to monitor and alerting

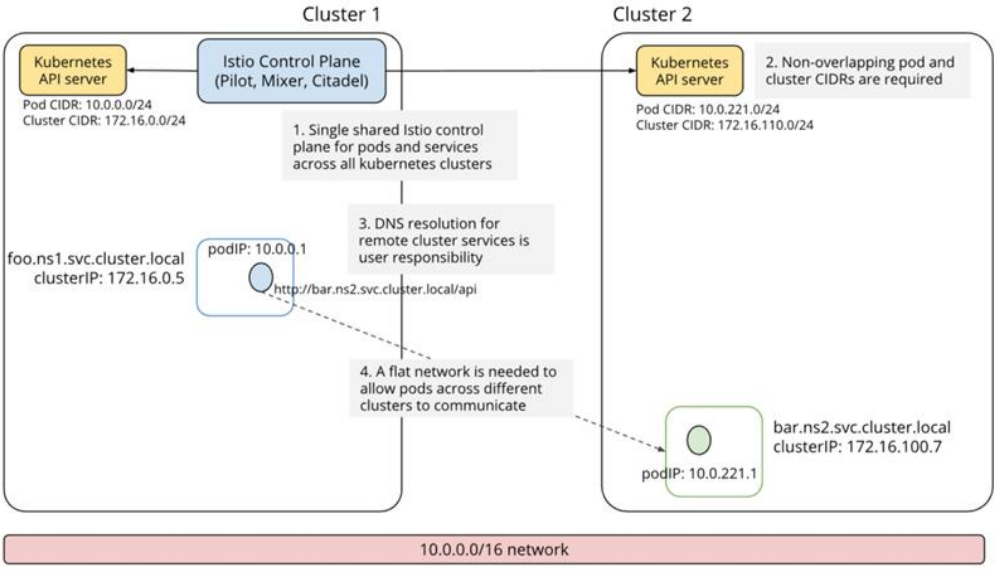

Figure 10: Istio mesh spanning multiple Kubernetes clusters with direct network access to remote pods over VPN

3.5 Summary of Literature Survey

After analysing all these papers, we get some insights on how to deploy and manage microservices in multiple public cloud platforms, by making use of various tools like Istio, Grafana, Prometheus etc.

We came across various algorithms that talk about efficient load balancing techniques - horizontal container scaling algorithm, scaling up and down algorithms, and dynamic scaling algorithms.

Also, we could see that the default Kubernetes algorithm can lead to inefficient usage of resources when the number of requests is high, hence the paper proposes a custom controlled algorithm which actually gave 50% cost reduction.

3.6 Conclusion

Default Kubernetes algorithm can lead to inefficient usage of resources when the number of requests is high, hence the paper proposes a custom controlled algorithm which actually gave 50% cost reduction.

Hence, this algorithm can be used for deploying microservice in our project.

4. PROJECT REQUIREMENT SPECIFICATION

4.1. Implementation Perspective

This algorithm aims to decrease the costs of bearing the cloud infrastructure due to unnecessary traffic loads being applied on one node. Instead, the custom scaling algorithm is scaling the number of pods horizontally, up, and down and thus reducing the load on each node specifically. The algorithm is focused on improving upon the default horizontal pod allocator which consumes needlessly more costs to scale up and down the pods by calculating only the necessary required pods and creating them. The goal is to scale up and down on the needs of the CPU utilisation of each node created for the microservices that can be easily integrated into a Kubernetes cluster by any of the companies that wish to use Kubernetes. The infrastructure must be able to handle a very large user base at a time due to the highly fluctuating requests per second by a user. It is also to be able to deploy the custom scheduler such that cost saving is done effectively and efficiently.

4.2. Operating Environment

The server that is running the Kubernetes cluster can either be of Windows server or a Linux Server that has Docker, Kubernetes and Istio available to be installed or present in the server. The application will require certain dependencies such as Python or Java or other dependencies depending on the application that is being run on the Kubernetes cluster which will be installed automatically when installing the image. The server will require a good internet connection to have the server up and running to receive and server external user requests to the website.

4.3. DEPENDENCIES, ASSUMPTIONS AND RISKS

4.3.1. Dependencies

1. Kubernetes resources such as pods, deployments, services, and ingress controllers are used to manage the lifecycle and networking of microservices.

2. The container images are stored in a container registry, such as Docker Hub or Google Container Registry and pulled by Kubernetes nodes when needed.

3. The service discovery and load balancing features which are provided by Kubernetes services, which can route traffic to the appropriate microservice instance.

4. Persistent storage for microservices, such as databases or file systems, are provisioned using Kubernetes Persistent Volumes and Claims. Hence it is important to ensure that storage is properly configured and accessible by the microservices.

5. A well configured network to enable communication between microservices.

4.3.2. Assumptions

1. The microservices are designed to be stateless and scalable, with multiple replicas running in parallel to handle incoming traffic.

2. It is assumed that each microservice can be packaged as a container image that can be deployed on the platform. This is due to microservices being deployed on Kubernetes typically being containerized.

3. It is assumed that each microservice can be discovered by other services in the cluster, allowing them to communicate with each other.

4. The microservices are independent and loosely coupled with minimal dependencies on other microservices or infrastructure components.

4.4. Functional Requirements

1. Ability to scale up and down the pods based on the CPU utilisation of the nodes and the traffic load being generated by external user requests.
2. Ability to reduce costs of maintenance of pods by scaling down as soon as possible once the load on the server has decreased.

4.5. Non - Functional Requirements

1. Ability to enable users to access services without restriction from the server's end.
2. The system should have backups ready due to data persistence being an issue in Kubernetes clusters in case of data loss or data corruption.

4.6. Other Requirements

1. **Performance:** The application and system should be designed in such a way that the server will run efficiently when there is a huge varying load present on the server.
2. **Testing:** The custom scheduler will be tested using load generator tools like locust and compared with other schedulers like the default Kubernetes horizontal pod allocator and another Custom Scheduling Algorithm. These testing tools will be used to ensure that the server does not unintentionally crash by putting it through rigorous testing.
3. **Evaluation:** There is testing that the application is put through when the custom scheduler is running and when KHPA is running. These are run on specific time frames and are then compared with one another. Based on metrics such as how much CPU utilisation is taken and the number of pods created and deleted in the time frames.

5. SYSTEM DESIGN

5.1. Kubernetes Structure

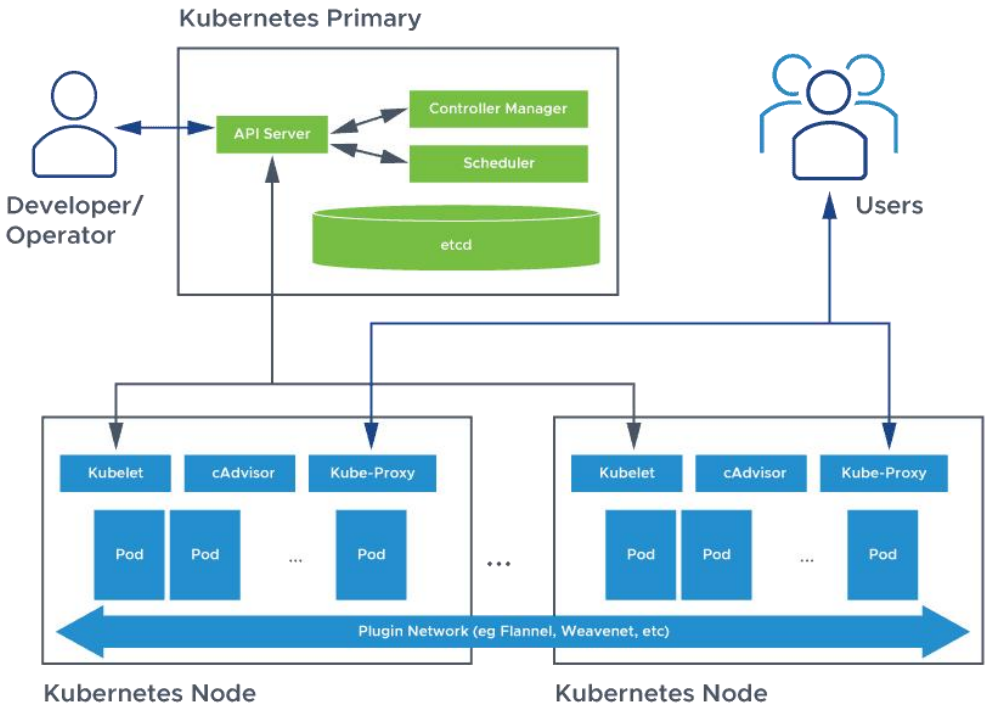

Figure 11: Structure of the Kubernetes Master Node and the Kubernetes slave node

The above figure is the architecture which depicts how the master nodes and worker nodes work in the Kubernetes cluster where the Developer sends requests to the Kubernetes API Server which is in the master node. The master node which consists of the Scheduler and the Controller Manager calculates the number of worker nodes as well as which of the worker nodes needs to do which specific work as in a server if there is already some CPU utilization in the worker nodes, the master node cannot allot more CPU load onto the worker node than it can handle. Thus, the worker nodes receive the instructions from the Master Node through Kubelet and after processing sends requested information to the Users through the Kube-Proxy, which is a network proxy in the Kubernetes cluster that is present in each node.

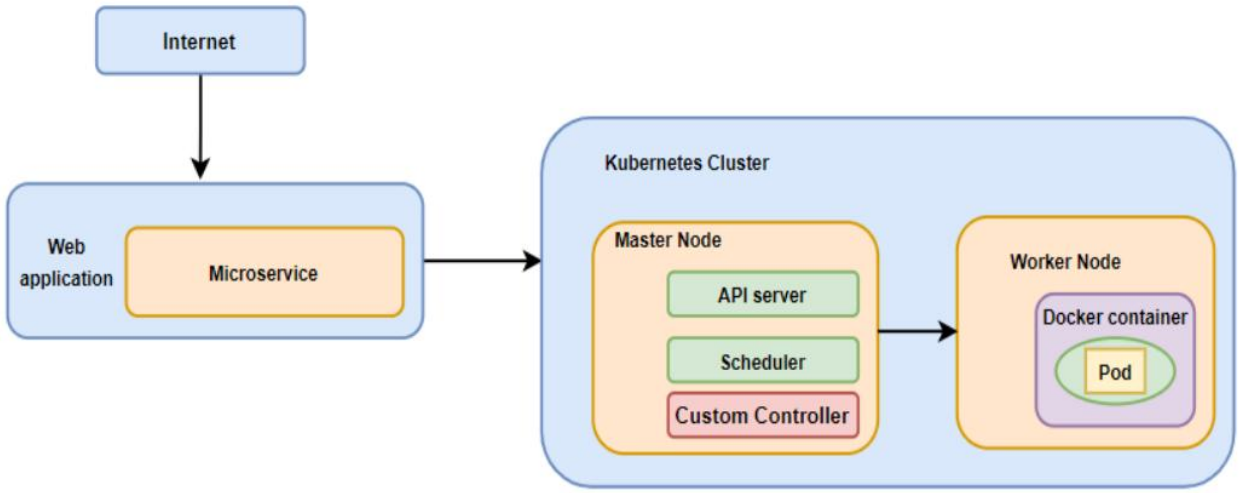

Figure 12: Kubernetes Control Plane with introduction of custom scheduler

In the figure above, the Kubernetes Control Plane is introduced with another component, namely the Custom Controller/Scheduler to determine the number of pods to be scheduled for the microservice. The custom controller will work specifically using the CPU utilisation by taking the metrics from the Kubernetes Metrics Server which is a tool that maintains data on resource usage for pods and nodes in the Kubernetes Cluster. The custom controller communicates with the other components of control plane via API calls to change the state of the data plane.

5.2. Architecture

5.2.1. Design pattern used:

Consider an enterprise operating system, the application might consist of so many microservices which need to communicate with each other for better serving of requests, it could be business logic, or consistency, scalability, security, handling partial failures etc communication is important.

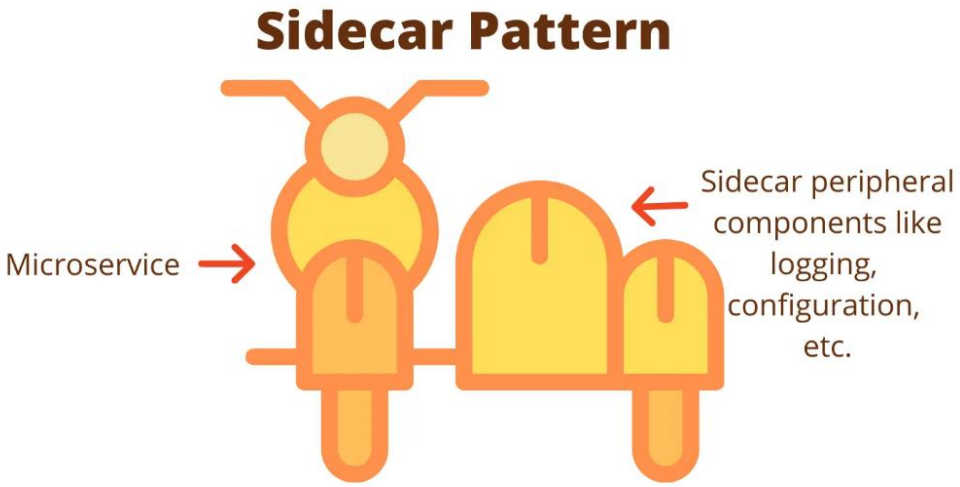

Figure 13: Sidecar Pattern

We can use a sidecar design pattern, the sidecar which is also called a sidecar/envoy proxy which facilitates the various functionalities like service discovery, traffic management, load balancing, circuit breaking, etc.

It is a container that will be deployed along with microservice. The microservice knows nothing about the outside world. Any interaction with the outside will happen through the sidecar proxy

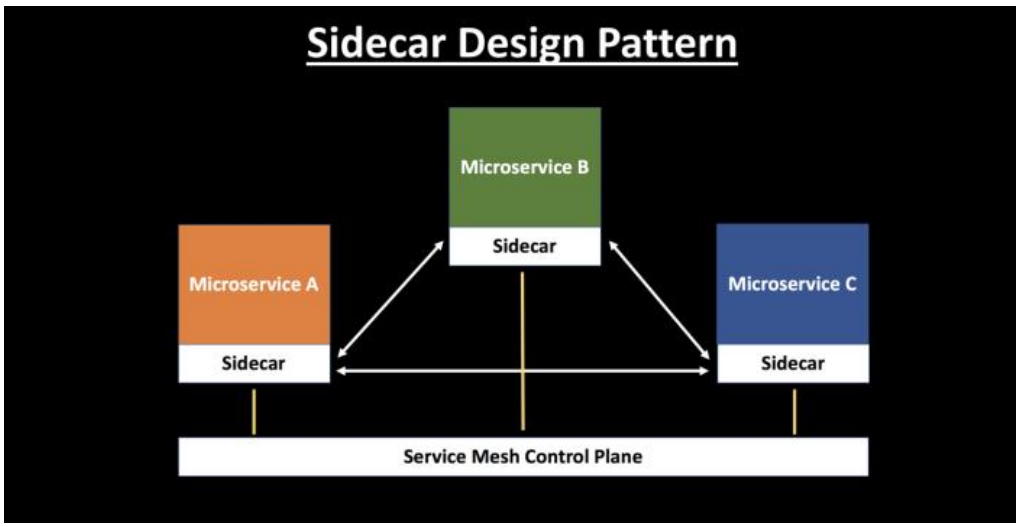

Figure 14: Sidecar Pattern with Microservices

These sidecars are independent of the microservices hence their manipulation and configuring becomes easy and they can be controlled by the service mesh control plane.

So, each of the microservice will be containerized and deployed along with a sidecar/envoy proxy.

5.2.2. Istio service mesh architecture:

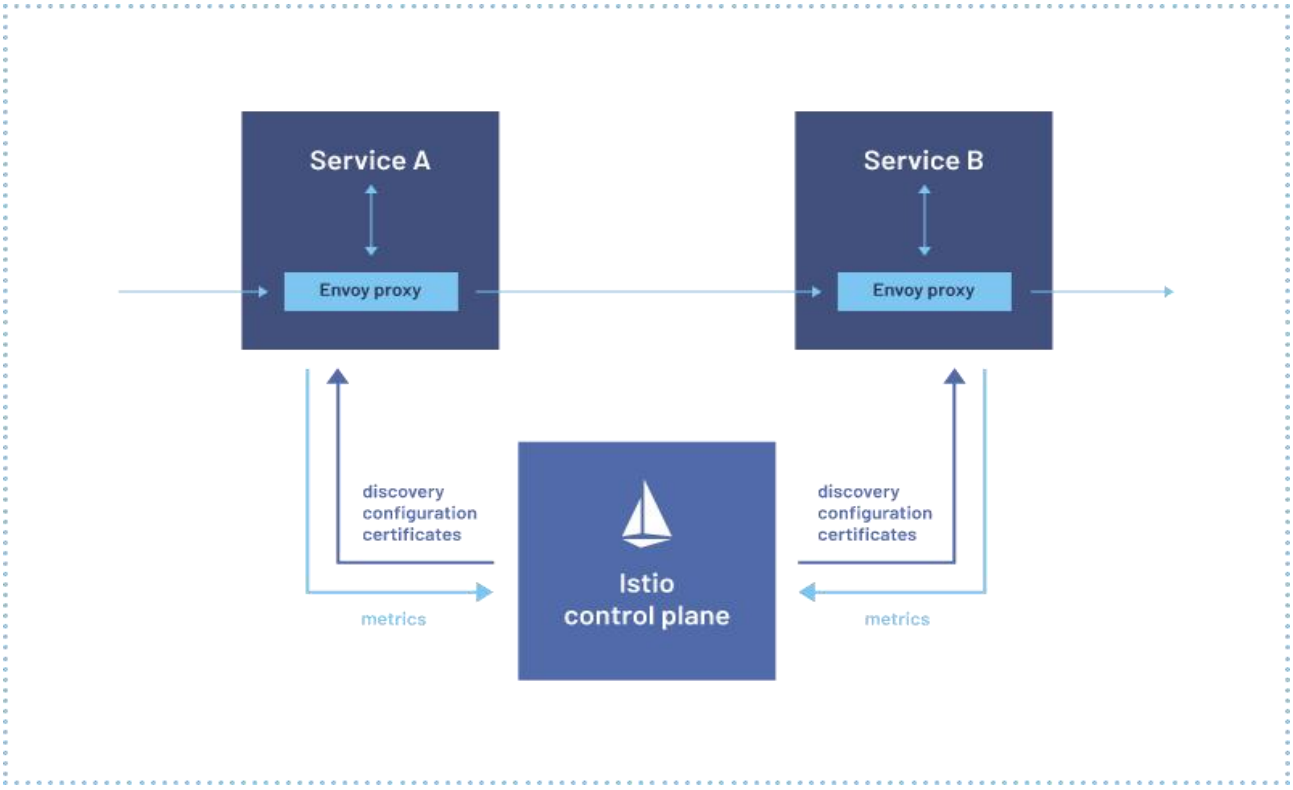

Figure 15: Istio service mesh

The design describes the usage of Istio service mesh to inject the sidecar and load balance the requests between the containers from the control plane

Deploying Istio on Kubernetes will involve running a cluster on a dedicated namespace and will depend on Kubernetes for networking and service discovery. Kubernetes does not implement service to service communication therefore Istio service mesh will take care of this including load balancing, traffic management, security and fault tolerance.

5.2.3. Advantages of Istio service mesh:

Istio is an open-source service mesh platform. It helps manage and secure microservices in a Kubernetes cluster

1. Envoy proxy: Istio deploys a sidecar proxy, the Envoy proxy, alongside each service instance in the cluster. The proxy intercepts all inbound and outbound traffic to the service instance and sends it through the Istio data plane.

2. Istio data plane: The Istio data plane consists of a set of Envoy proxies and a control plane. The Envoy proxies forward requests to other services and implements the various Istio features such as traffic management, security, and observability.

3. Istio control plane: The Istio control plane manages the Envoy proxies and their configuration. It provides a central place to configure and manage the various Istio features, such as routing rules, security policies, and telemetry.

4. Traffic management: Istio can route traffic between services based on various criteria, such as HTTP headers, source IP address, or user identity. It can also implement features like load balancing, circuit breaking, and retries.

5. Security: Istio can enforce mutual TLS authentication between services, implement access control policies, and provide encryption for service-to-service communication.

6. Observability: Istio provides various telemetry features such as request tracing, metrics, and logging. This allows operators to understand how traffic is flowing through the system and to diagnose issues

There is scope for implementing custom controller algorithms to reduce the maintenance cost as the default Kubernetes algorithm leads to improper resource allocation, performance degradation and increase the maintenance cost.

Implementing this at the master node which is responsible for scaling up and down will impact by reducing maintenance cost by 50% as per the literature survey.

The service mesh architecture also provides a path for observability and trace the origin of the problem

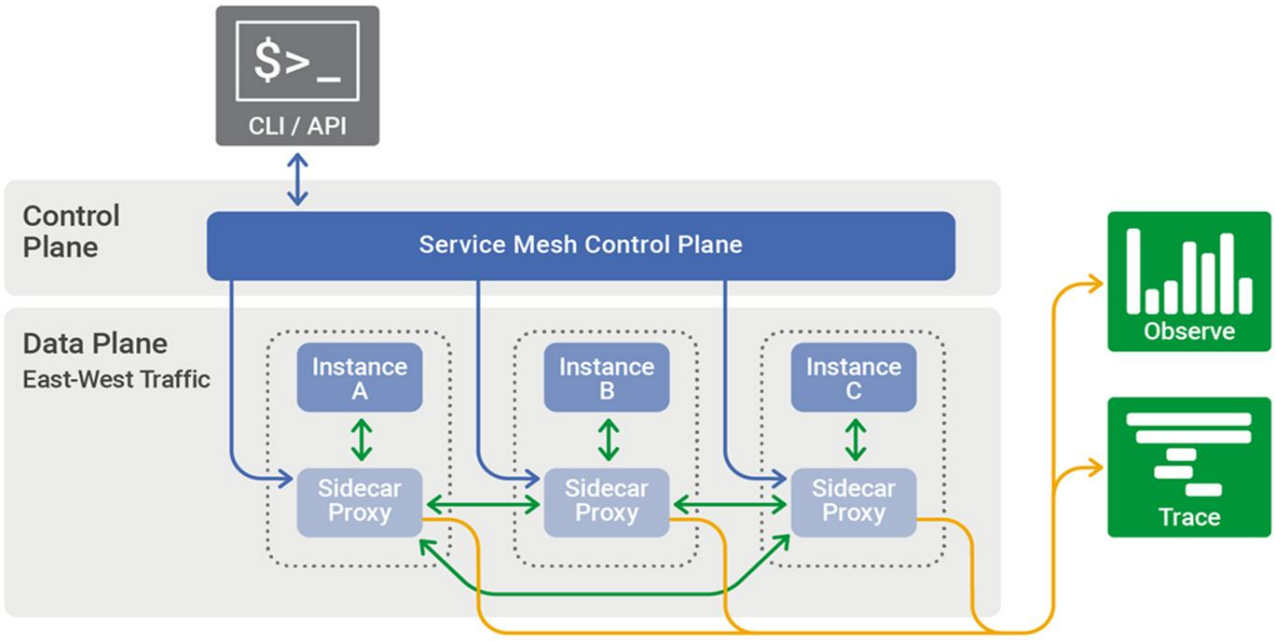

Figure 16: Istio Service Mesh Architecture

In the above figure we can see the working of Istio Service Mesh Architecture. The client sends requests through the API calls which is sent to the Service Mesh Control Plane which forwards it to the nodes after deciding which node to forward the process to. The forwarded process is then sent through the envoy proxies to the actual node which sends the value after finishing the process back to the envoy proxy which collects the metrics and is used to observe.

5.3. Prototype Architecture

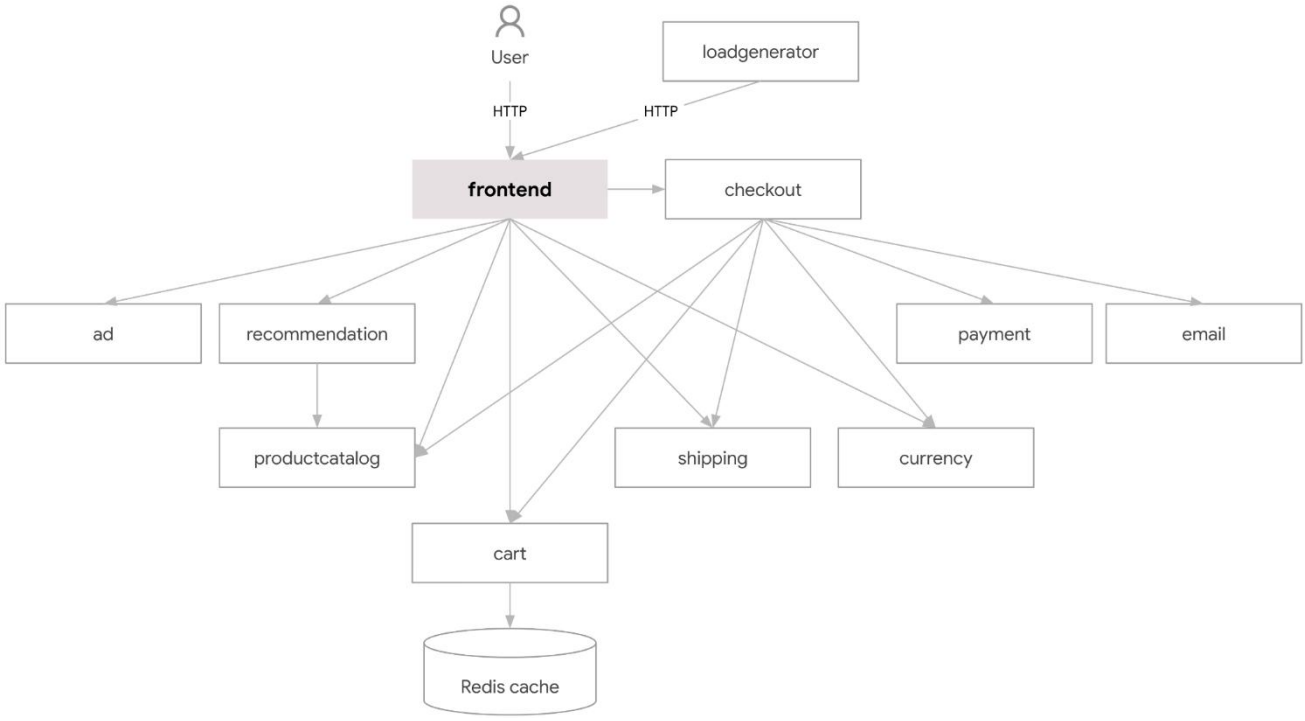

Figure 17: prototype/ product-based approach

Here the above figure consists of a prototype class diagram that would be the case for a typical e-commerce application running on Kubernetes cluster. The user sends requests through the frontend which is then sending information to the system which will send to the specific microservice page being used currently and the frontend will update itself. The checkout service forwards the requests to the cart service which the frontend will display. The cart microservice consists of the Redis Cache or other types of storage of data which will consist of the items added to the cart. The cache is accessed and shown in the frontend to the user. The load generator is sending HTTP requests as multiple users which will then increase the load on the system.

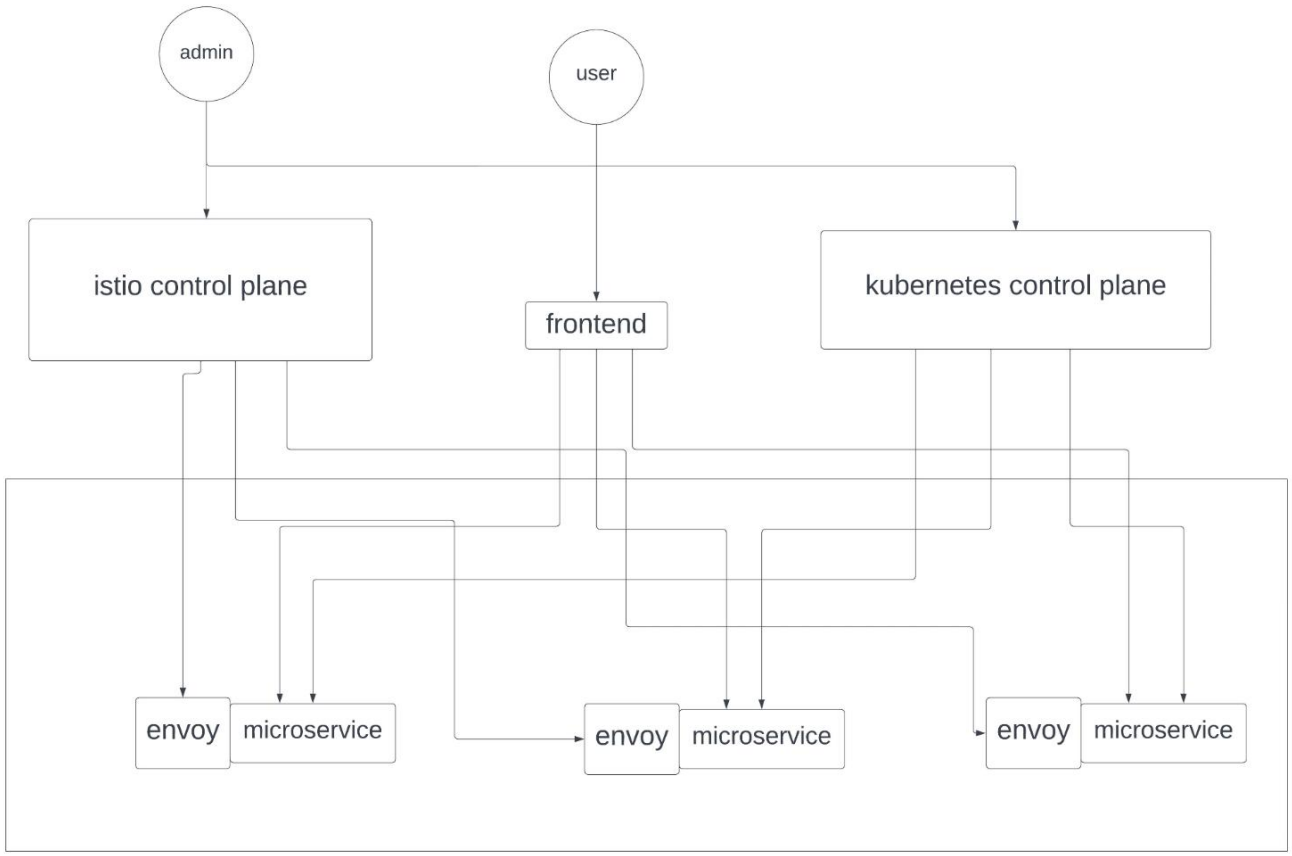

Figure 16: final deployment architecture

Here is a basic architecture of the final deployment which consists of the user sending the requests to the server only through the frontend, while the admin can configure the Istio control plane and Kubernetes control plane. The requests sent to the frontend are then sent to their respective microservices. The microservice only communicates with its envoy proxy. The envoy proxy sends the necessary requests to all the other ends.

5.4. Flowchart

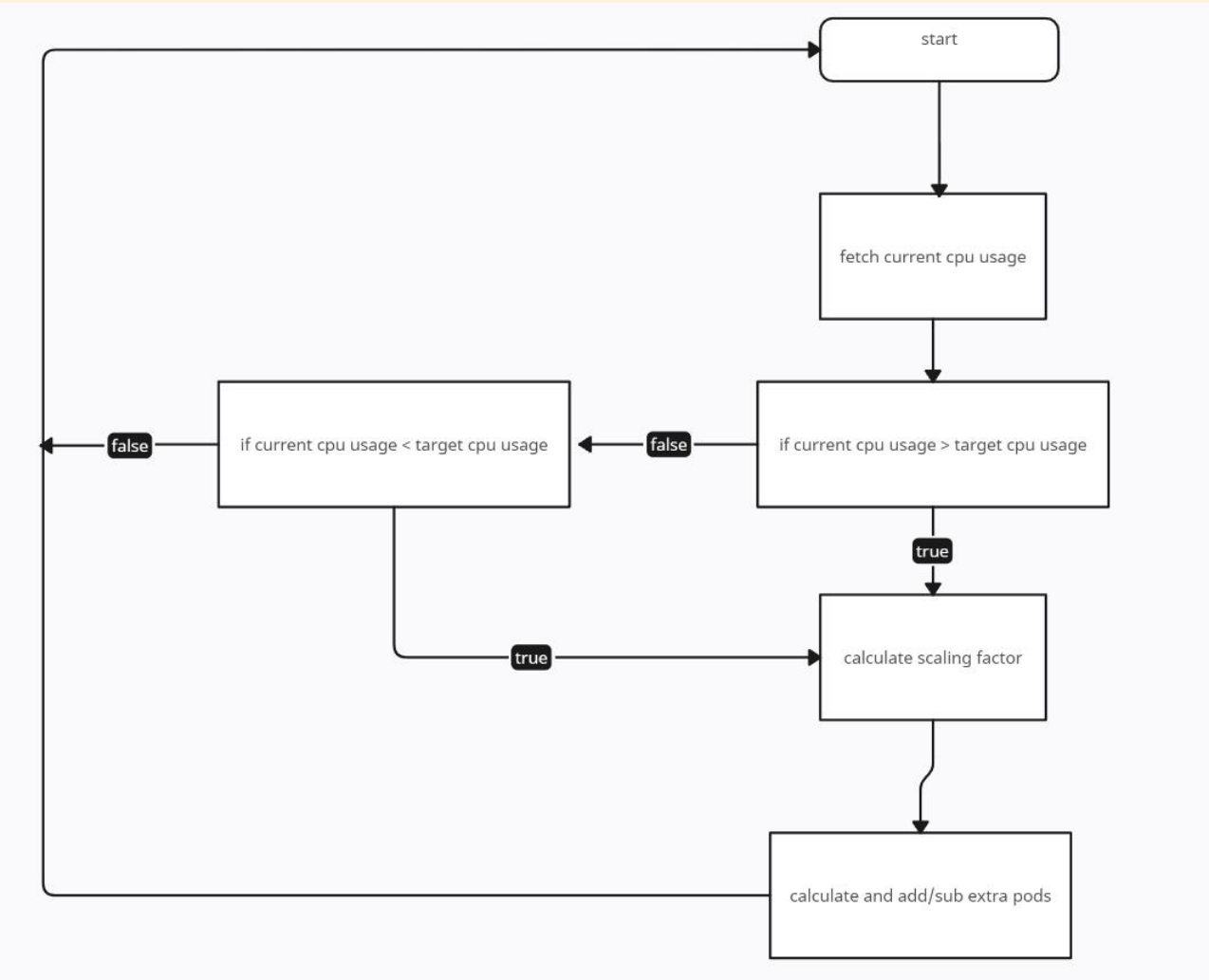

Figure 17: Flowchart working of the algorithm

Here the flowchart describes the working of the custom scheduler which takes in current CPU usage from Kubernetes Metrics Server and then compares if it is greater than or lesser than the target CPU usage value which is set in the configuration of the custom scheduler. After which it either is repeating again or calculating the scaling factor which is a value which helps to decide whether to add or delete extra pods(replicas) to the already existing pod(s).

6. PROPOSED METHODOLOGY

6.1 Methodology for microservice deployment on Kubernetes:

To address the challenges of microservice deployment on Kubernetes, we propose a methodology that includes the following steps: design, build, test, deploy, and monitor.

During the design phase, the microservices are identified and their dependencies are mapped out. In the build phase, the services are developed and containerized. Testing ensures that each service works correctly and communicates with other services. Deployment involves setting up the Kubernetes cluster and deploying the services. Finally, monitoring ensures that the services are running correctly, and alerts are raised if any issues arise.

6.2 The methodology for load balancing and scalability:

Custom controller algorithm**:**

Algorithm 1 *Autoscaling Algorithm for resource allocation.*

Input: Total_Pods,Total_CPU_Usage_Value,Total_CPU_target_value;
Output: $Total\_Pods_n$ = Total number of pod to be scheduled.
Total_Pods; = sum(pod0,pod1,.....Podn)); // Calculates the total number of pods running in cluster
Size_of_cluster = Total_Pods.length;
Total_CPU_target_value = fetch_target_CPU(); // API call for fetching the target CPU
Total_CPU_Usage_Value = fetch_current_usage(); // API call for fetching the current CPU usage.
if *Size_of_cluster > 0 && Total_CPU_Usage_Value > (Size_of_cluster *Total_CPU_target_value)* **then**
  for *i in Total_Pods* **do**
    $Total\_Pods_n$ = *Total_CPU_Usage_Value / Total_CPU_target_value* // Calculate the total number of pods.
  end
end

Deshpande, Neha (2021) “Autoscaling Cloud-Native Applications using Custom Controller of Kubernetes.” Master’s thesis, Dublin, National College of Ireland. https://norma.ncirl.ie/5089/

6.2.1 Methodology

To prevent resource waste and website crashes, the architecture consists of one master node and two worker nodes. In this paper, an attempt was made to create a custom algorithm to calculate the required number of pods. Kube-controller manager is one of the components. Kubernetes cluster using YAML. The YAML file's parameters are sent to the Kube-API server and then to the Kube-controller manager, where they are used to control replicas. The custom controller is then instructed to start existing pods or generate new ones via the Kube-API server.

6.2.2 Conclusion

This methodology schedules the efficient number of pods and preserves the resources along with maintaining stability.

6.3 Benefits and Drawbacks of the proposed methodology:

- Decrease in cost for maintaining Quality of Service.
- Provides support to only CPU intensive microservices.
- Data Persistence is complex in projects related to Kubernetes clusters when it comes to maintaining the data after it has scaled up and is going to scale down.

7. IMPLEMENTATION AND PSEUDOCODE

```
# Run the command and capture the output
    process = subprocess.Popen(command, shell=True, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
    stdout, stderr = process.communicate()

    # Check for any errors
    if process.returncode != 0:
        print("Error:", stderr.decode('utf-8'))
    else:
        # Process the output to access CPU values
        output_lines = stdout.decode('utf-8').split('\n')
        # The output contains columns for NAME, CPU(cores), and MEMORY(bytes)
        # Extract the CPU values from the output
        for line in output_lines[1:]:  # Skip the header line
            if line:
                #print(line)
                pod, pod_name, cpu_usage_m, memory_usage_b= line.split()

                #print(f"Pod: {pod_name}, CPU Usage: {cpu_usage}")
                cpu_usage_m = int(cpu_usage_m.rstrip("m"))
                break
                #current_usage_value = (cpu_usage_m / total_cpu_requests)*100
                #print(current_usage_value)

    print("fetched current cpu usage")
    print(cpu_usage_m)
```

Figure 18: Fetching Metrics Values

The above code snippet consists of a python script to retrieve metrics from the Kubernetes Metrics Server such as CPU Utilisation and total number of pods being currently available. The python script uses the subprocess library to run shell commands which receives the response from the Kubernetes Metrics Server which is then going to be stripped into smaller bits so that the custom scheduler can take in only the required values, being the total CPU utilisation. The output values from the shell command are given in pod name, CPU (in cores) and MEMORY (in Bytes).

```
scaling_factor = (cpu_usage_m - target_cpu_value) / 100.0
if cpu_usage_m > target_cpu_value:
    # Increase pods proportionally to the difference between current and target CPU

    predicted_pods = round(min_pods + (max_pods - min_pods) * scaling_factor)
    predicted_pods = max(min_pods, min(predicted_pods,max_pods))
```

Figure 19: Scaling Factor of Algorithm

The code snippet above is a Python script designed for dynamic scaling in computing environments. Calculates the scaling factor based on the difference between the current CPU usage (cpu_usage_m) and the target CPU value (target_cpu_value). If the current CPU usage exceeds the target, the predicted number of pods is calculated using a proportional scaling factor within the specified range (min_pods to max_pods). The results are rounded, and constraints are applied to ensure that the predicted number of pods is within the defined limits. This script is a basic implementation of autoscaling logic that is often used in systems to dynamically adjust resource allocation based on observed workloads. The last line in the code snippet being max(function) is used when we are to calculate how many pods to create and is replaced with min(function) when needed to calculate how many pods to delete.

```
pod = client.V1Pod( metadata=client.V1ObjectMeta(name="my-pod-{}".format(unique_id),labels={"app": "my-nginx"}),
    spec=client.V1PodSpec(containers=[client.V1Container(name="my-nginx",image="veenagarag/my-nginx-sample",)]),)

try:
    # Create the pod in the specified namespace
    api.create_namespaced_pod(namespace, pod)

    print(f"Pod {pod.metadata.name} created successfully.")
except Exception as e:
    print(f"Error creating pod: {e}")
```

Figure 20: Creation of pods when scaling up

The code snippet is designed to interact with a Kubernetes cluster through the Kubernetes Python client. This segment defines a Kubernetes pod with a unique name that includes a dynamically generated unique identifier and a specified label that identifies it as part of the my-nginx application. The pod specification contains a single container configured to use the veenagarag/my-nginx-sample Docker image. The script then attempts to create this pod within her specified Kubernetes namespace using the Kubernetes Python client's ‘create_namespaced_pod’ method. A try-Except block is used to handle exceptions that may occur during the pod creation process. If an exception occurs, an error message is printed detailing the problem that occurred. Conversely, if the pod creation is successful, a confirmation message indicating that the pod creation was successful is printed along with its name. This code segment serves as a basic building block for automating the deployment of Kubernetes pods within a given namespace, and can potentially be further integrated into larger automation or orchestration workflows within a Kubernetes environment.

```
if predicted_pods < total_podsn and total_podsn > min_pods:
    # Find the indices of elements that start with "my-pod-"
    indices_to_deleten = [i for i, element in enumerate(pods_Arrn) if element.startswith("my-pod-")]
    n = total_podsn - predicted_pods
    # Extract the elements to be deleted
    elements_to_deleten = [pods_Arrn[i] for i in indices_to_deleten[:n]]
    for i in elements_to_deleten:
        try:
            # Delete the pod
            api.delete_namespaced_pod(i, namespace)
            print(f"Pod {i} deleted successfully.")
        except Exception as e:
            print(f"Error deleting pod: {e}")
```

Figure 21: Deletion of pods when scaling down

The provided code snippet removes excess pods if the predicted number of pods ('predicted_pods') is less than the total number of pods ('total_pods') and the specified minimum threshold ('min_pods'). If these conditions are met, the script identifies and prints the pod to remove by searching for the index of the element starting with the prefix "my-pod-" in the pods_Arr array. Calculates the number of pods to

remove (n) based on the difference between the total number and the predicted number of pods. The script then extracts and prints the relevant pod items and attempts to delete each identified pod using Kubernetes API calls. All successful deletions are output as notifications, and any errors that occur during the deletion process are captured and output. Overall, this script segment helps you dynamically adjust the number of pods by removing extra pods as needed.

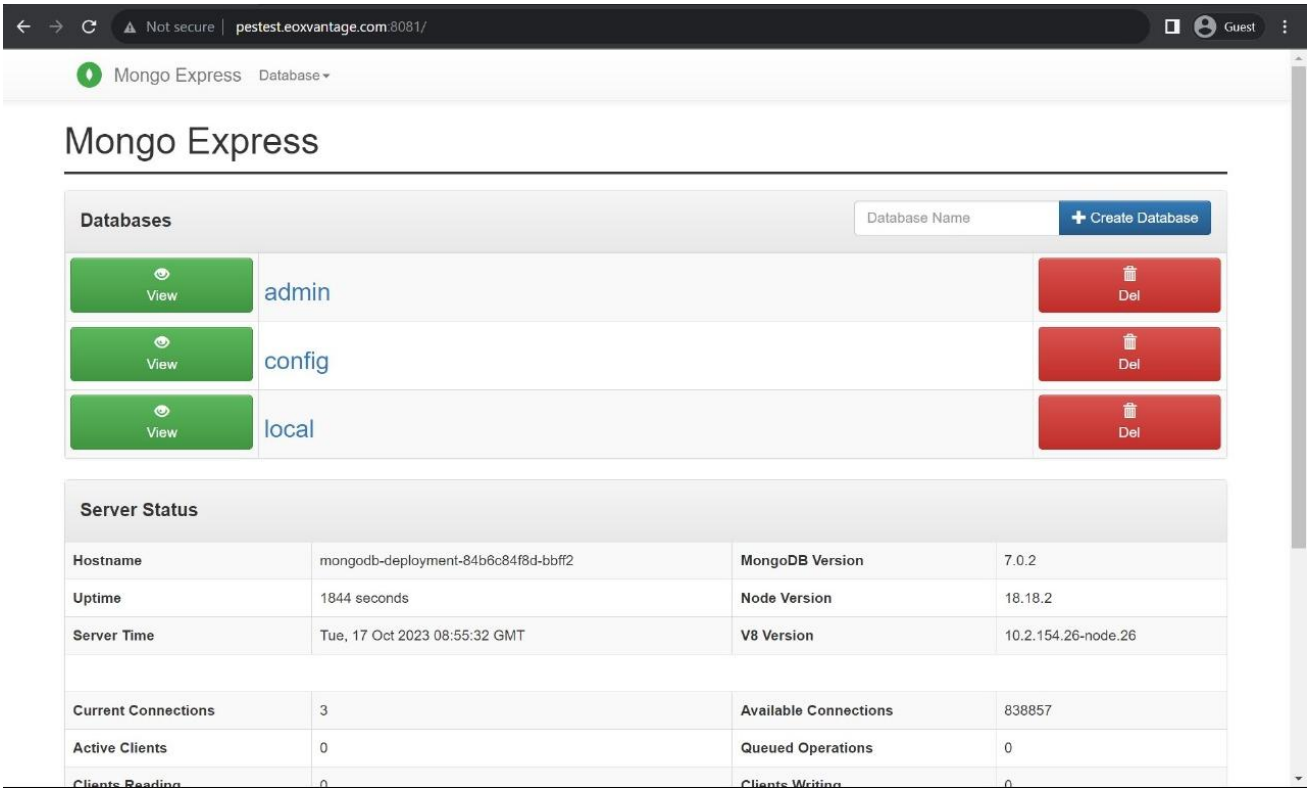

Figure 22: Demo Application running on live server

Successfully deployed MongoDB Application on the live server and the services were able to accept and serve the external requests.

8. RESULTS AND DISCUSSION

8.1. Custom Scheduler

The implementation of custom scheduler algorithm into the Kubernetes plane is the crucial aspect of the project. The algorithm calculates the number of pods to be scheduled based on the varying load on the application pods. The algorithm takes care of both increasing the pods when there is increase of load and decrease of pods when there is decrease in the load on the application pods. The algorithm takes care of efficient number of pods creation and deletion leading to efficient resource consumption and decrease in maintenance cost.

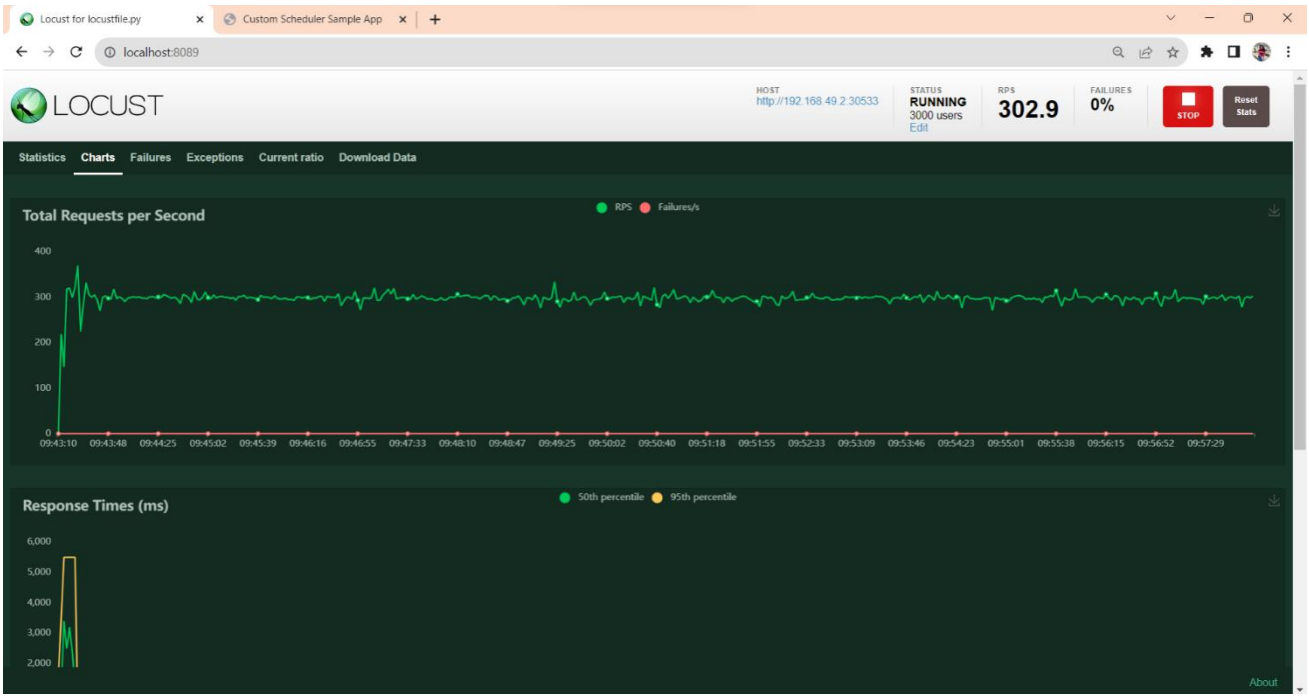

Figure 23: Load Generation using Locust

The above image shows the varying user load on the application running the custom scheduler that has been generated from the beginning till the 15th minute. The load generator has been set to 3000 users and a total of 600 RPS (Requests per Second) by the users.

```
C:\Users\veena garag\Desktop\actual custom implementation\custom in python\custom algorithm demo>kubectl top pods
NAME                                      CPU(cores)   MEMORY(bytes)
custom-scheduler-deployment-567df875b6-6kk4m   1m           2Mi
custom-scheduler-pod                      1m           5Mi
locust-deployment-75cd59c768-5tpcf        549m         172Mi
my-nginx-deployment-58fc9f94bd-gp5s6      79m          16Mi
my-pod-dff792                             0m           7Mi
my-pod-ec3cad                             0m           6Mi

C:\Users\veena garag\Desktop\actual custom implementation\custom in python\custom algorithm demo>
```

Figure 24: Pods at 2nd min frame

The above image shows the pods created at the 2nd minute after the load generator started generating load.

```
C:\Users\veena garag\Desktop\actual custom implementation\custom in python\custom algorithm demo>kubectl top pods
NAME                                      CPU(cores)   MEMORY(bytes)
custom-scheduler-deployment-567df875b6-6kk4m   1m           2Mi
custom-scheduler-pod                      1m           6Mi
locust-deployment-75cd59c768-5tpcf        654m         170Mi
my-nginx-deployment-58fc9f94bd-gp5s6      93m          16Mi
my-pod-bac772                             0m           8Mi
my-pod-dff792                             0m           7Mi
my-pod-ec3cad                             0m           6Mi

C:\Users\veena garag\Desktop\actual custom implementation\custom in python\custom algorithm demo>
```

Figure 25: Pods at 8th min frame

The above image shows the pods created during the 8th minute after the load generator started generating load.

```
C:\Users\veena garag\Desktop\actual custom implementation\custom in python\custom algorithm demo>kubectl top pods
NAME                                      CPU(cores)   MEMORY(bytes)
custom-scheduler-deployment-567df875b6-6kk4m   1m           2Mi
custom-scheduler-pod                      1m           5Mi
locust-deployment-75cd59c768-5tpcf        613m         173Mi
my-nginx-deployment-58fc9f94bd-gp5s6      82m          17Mi
my-pod-31bc20                             0m           7Mi
my-pod-ec3cad                             0m           6Mi

C:\Users\veena garag\Desktop\actual custom implementation\custom in python\custom algorithm demo>
```

Figure 26: Pods at 15th min frame

The above image shows the pods created during the 15th minute after the load generator started generating load.

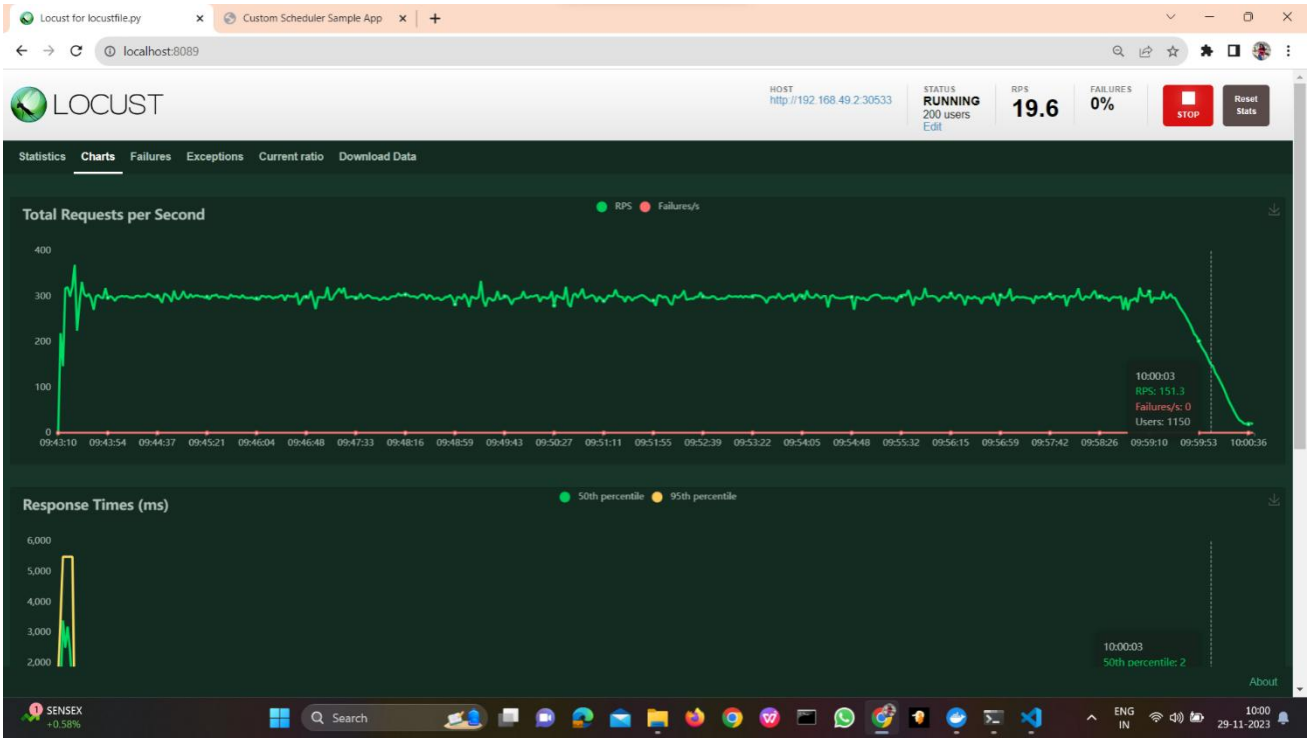

Figure 27: Decreasing Load

In the above image we are decreasing the load from 3000 users to 200 users and the RPS from 600 to 50.

```
C:\Users\veena garag\Desktop\actual custom implementation\custom in python\custom algorithm demo>kubectl top pods
NAME                                        CPU(cores)   MEMORY(bytes)
custom-scheduler-deployment-567df875b6-6kk4m   1m           2Mi
custom-scheduler-pod                        1m           6Mi
locust-deployment-75cd59c768-5tpcf          58m          139Mi
my-nginx-deployment-58fc9f94bd-gp5s6        9m           12Mi

C:\Users\veena garag\Desktop\actual custom implementation\custom in python\custom algorithm demo>
```

Figure 28: Pods after decreasing load

In the above image we can see that the additional replicas that had been created using the custom scheduler have now been removed and is back to one pod only in a very short period.

8.2. Kubernetes Horizontal Pod Autoscaler (KHPA)

The demonstration of KHPA on the same load to analyses the performance of custom scheduler is vital. The comparison of performance of custom scheduler and KHPA will prove the performance superiority of custom scheduler. The experiment is conducted on different time frames to fetch the number of pods KHPA schedules.

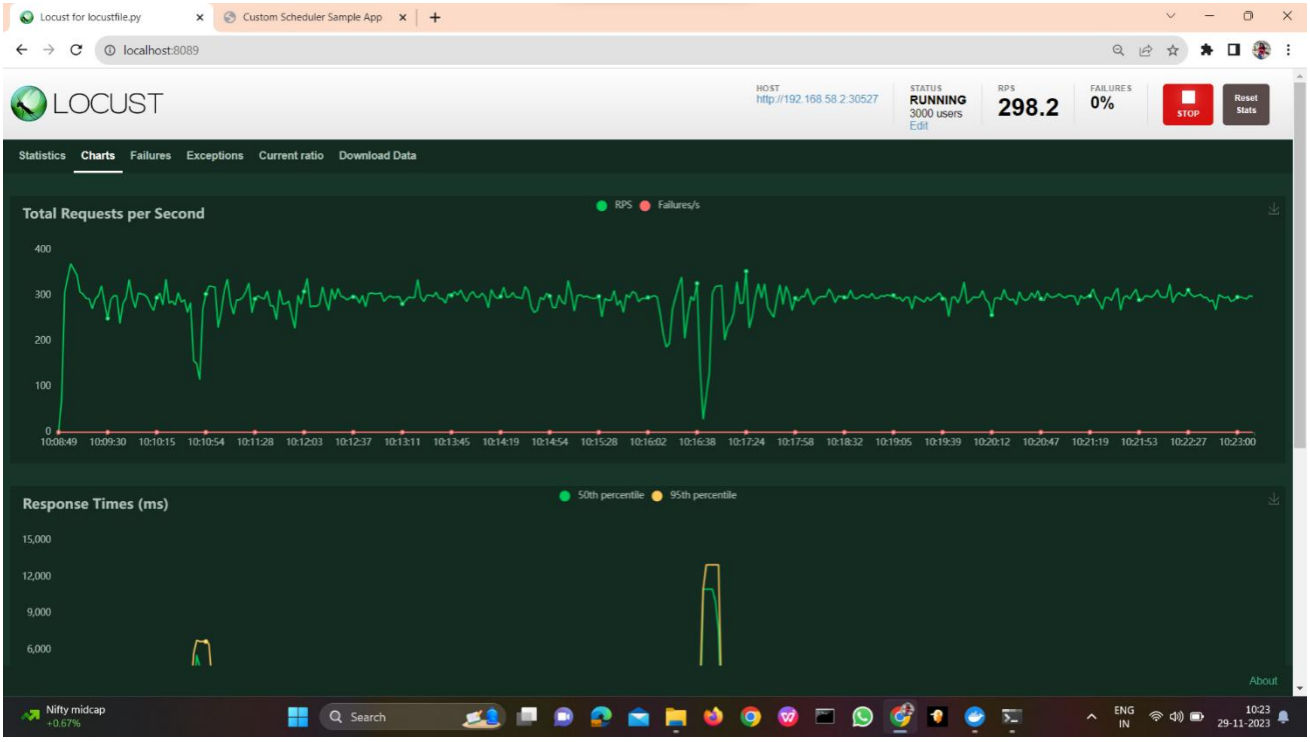

Figure 29: Load generation using Locust

The above image shows the varying user load on the application running the KHPA (Kubernetes Horizontal Pod Autoscaler) that has been generated from the beginning till the 15th minute. The load generator has been set to 3000 users and a total of 600 RPS (Requests per Second) by the users.

```
C:\Users\veena garag\Desktop\khpa demo>kubectl get hpa
NAME                REFERENCE                      TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
my-nginx-deployment   Deployment/my-nginx-deployment   90%/50%   1         10        9          22h
```

Figure 30: Pods at 2nd min frame

The above image shows the pods created at the 2nd minute after the load generator started generating load.

```
C:\Users\veena garag\Desktop\khpa demo>kubectl get hpa
NAME                  REFERENCE                        TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
my-nginx-deployment   Deployment/my-nginx-deployment   46%/50%   1         10        10         22h
```

Figure 31: Pods at 8th min frame

The above image shows the pods created during the 8th minute after the load generator started generating load.

```
C:\Users\veena garag\Desktop\khpa demo>kubectl get hpa
NAME                  REFERENCE                        TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
my-nginx-deployment   Deployment/my-nginx-deployment   46%/50%   1         10        10         23h
```

Figure 32: Pods at 15 min frame

The above image shows the pods created during the 15th minute after the load generator started generating load.

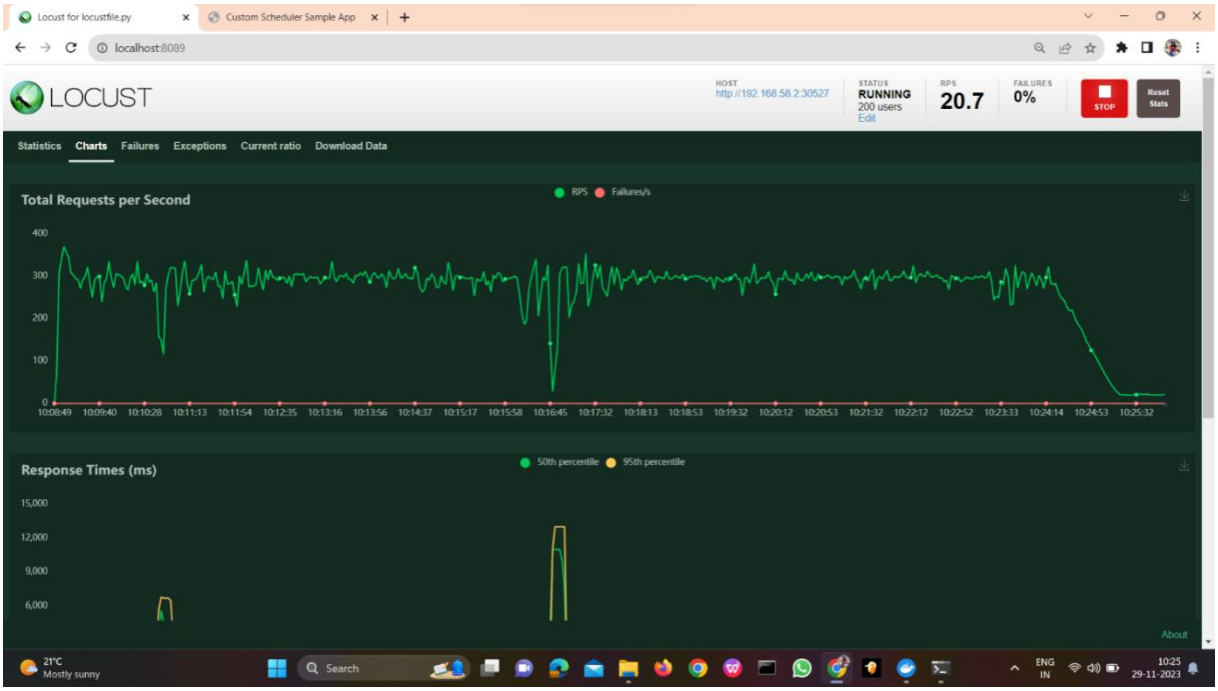

Figure 33: Load Fluctuation in the Locust Testing

In the above image we are decreasing the load from 3000 users to 200 users and the RPS from 600 to 50.

```
C:\Users\veena garag\Desktop\khpa demo>kubectl get hpa
NAME                  REFERENCE                        TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
my-nginx-deployment   Deployment/my-nginx-deployment   16%/50%   1         10        10         23h
```

Figure 34: Pods after decreasing load

In the above image we can see that the additional replicas that had been created using KHPA takes a much longer period of time to remove the pods even though the load on the pods is lesser than the target CPU utilisation.

9. CONCLUSION AND FUTURE WORK

We can thus conclude that the custom controller is creating replicas only, when necessary, which when used by KHPA is very random and excessive. The custom controller is also scaling down the system by deleting the replicas soon when they are not required, thus making it much more efficient and cost saving to KHPA.

For the future, Support for additional metrics: dealing with the memory utilisation aspect of this project would be very useful as the custom controller should not only be focused on the CPU utilisation of each of the pods when being created but also on the memory utilisation of each of those pods as they could be very high even if the CPU utilisation is low.

Predictive scaling: We can use predictive scaling to forecast future load and scale our microservices appropriately, as opposed to responding to the CPU utilization that is occurring right now. This could enhance overall scalability and prevent performance bottlenecks. This could be accomplished by training a model using historical CPU consumption data through machine learning.

Integrate with CI/CD pipelines: This would allow us to automatically deploy our custom scheduler to your Kubernetes cluster. Additionally, we may gather input on your scheduler's performance via CI/CD pipelines and use that information to gradually enhance the program.

10. REFERENCES/BIBLIOGRAPHY

[1] Deshpande and Neha (2021) "*Autoscaling Cloud-Native Applications using Custom Controller of Kubernetes.*" Master's thesis, Dublin, National College of Ireland. https://norma.ncirl.ie/5089/

[2] Shitole and Abishek Sanjay (2022) "*Dynamic Load Balancing of Microservices in Kubernetes Clusters using Service Mesh.*" Master's thesis, Dublin, National College of Ireland. https://norma.ncirl.ie/5943/

[3] Ashok L Pomnar, "An Efficient and Scalable Traffic Load Balancing Based on Web Server Container Resource Utilization using Kubernetes Cluster." (May – 2022) AVCOE Sangamner 422 605, Maharashtra, India. https://ijisrt.com/assets/upload/files/IJISRT22MAY1644_(1)_(1).pdf

[4] V. Sharma, "Managing Multi-Cloud Deployments on Kubernetes with Istio, Prometheus and Grafana," 2022 8th International Conference on Advanced Computing and Communication Systems (ICACCS), Coimbatore, India, https://doi.org/10.1109/ICACCS54159.2022.9785124

[5] Nandan Adhikari, (April 19, 2021) "Horizontal Pod Autoscaler in Kubernetes" https://around25.com/blog/horizontal-pod-autoscaler-in-kubernetes/

[6] Nana Janashia, (March 26, 2023) "Kubernetes ConfigMap and Secret explained" https://www.techworld-with-nana.com/post/kubernetes-configmap-and-secret-explained

11. APPENDIX A: ACRONYMS AND ABBREVIATIONS

KHPA: Kubernetes Horizontal Pod Autoscaler.

min: Minutes

RPS: Requests per Second

Word Count: 5533

Plagiarism Percentage 4%

Matches

1 **World Wide Web Match**
View Link

2 **World Wide Web Match**
View Link

3 **World Wide Web Match**
View Link

4 **World Wide Web Match**
View Link

5 **World Wide Web Match**
View Link

6 **World Wide Web Match**
View Link

7 **World Wide Web Match**
View Link

8 **World Wide Web Match**
View Link

9 **World Wide Web Match**
View Link

Suspected Content

1. INTRODUCTION Microservices architecture is an approach where an application is divided

into smaller, loosely coupled services that can be developed, deployed, and scaled 3