Deploying Microservices on Kubernetes Clusters and Improving Scalability Using a Custom Scheduler

Adarsh Kumar
Department of computer science and engineering
PES university
Bengaluru, India
pesug20cs016@pesu.pes.edu

Yuvaraj D C
Department of computer science and engineering
PES university
Bengaluru, India
dcyuvaraj@gmail.com

Veena Garag
Department of computer science and engineering
PES university
Bengaluru, India
veenagarag929@gmail.com

Suchit S Kallapur
Department of computer science and engineering
PES university
Bengaluru, India
such4869@gmail.com

***Abstract*— Project deals with deploying microservices on cloud and reducing the resource consumption like CPUs memory, and bringing down the cost of creation of pods. The project is aimed to improve and optimize resource utilization, improve security features and be able to monitor the performance and visualize the metrics in order to study the behavior of the microservices to increase load.**

Keywords — Kubernetes, Custom Controller, Microservices, Load testing, Monitoring, External access, Custom Controller (key words)

I. INTRODUCTION

In the dynamic realm of software development, the ability to effectively manage and scale applications has become increasingly crucial. With the ever-growing demands of modern applications, traditional monolithic architectures often struggle to adapt to fluctuating workloads, leading to performance bottlenecks and potential downtime. To address these challenges, the combination of Kubernetes and microservices architecture has emerged as a powerful solution, providing a robust and scalable foundation for building modern applications.

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.

A. Background

Kubernetes, an open-source container orchestration platform, has revolutionized the way applications are deployed, managed, and scaled in cloud-native environments. Its ability to automate many of the manual processes involved in managing containerized applications has made it an invaluable tool for developers and operations teams alike.

Microservices architecture, on the other hand, promotes the decomposition of complex applications into smaller, independently deployable services. This modular approach enhances development efficiency, simplifies maintenance, and enables finer-grained scalability, making it ideally suited for modern applications with dynamic workloads.

Traditionally, scaling applications has been a manual and often reactive process, requiring operators to monitor resource utilization and manually adjust the number of application instances. This approach is not only time-consuming and error-prone but also fails to adapt to the rapidly changing demands of modern applications.

B. Proposed Solution

To address these challenges, we propose the implementation of a custom controller auto-scaler within the Kubernetes environment. This auto-scaler will continuously monitor application metrics, such as CPU usage and request latency, and dynamically adjust the number of microservice instances based on the current workload. This proactive approach ensures that our application can handle fluctuating traffic without compromising performance or user experience.

II. RELATED WORK

The survey on several research papers related to Kubernetes and microservices lead to insights on the improvements possible in resource utilization and autoscaling.

[1] suggests the use of custom controllers to optimize the pod usage and bring down the total pods as compared to default Kubernetes HPA. 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 KHPA is more expensive than Custom scheduler algorithms. Using the Custom Controller nearly lowered the maintenance costs in half. In accordance with workload, Custom Controller manages and scales Pods dynamically. 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.

[2] suggests a technique that applies service-specific routing across the Istio control plane to inject sidecar proxies onto every microservice 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.

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 utilizing a control plane

In monolithic systems, scaling often requires replicating the entire application, leading to potential resource inefficiencies. Microservices, with their independent scalability, enable more efficient resource utilization by allocating resources to specific services based on demand. This flexibility is particularly advantageous in dynamic and evolving environments.

[3] Because the microservices operating on the containers are isolated from one another, managing the containers is a constant responsibility for any organization. As a distinct entity, they operate on their own. Here's where Kubernetes comes into play. All Kubernetes is a container management platform. Install and configure the Kubernetes cluster on each of these three nodes using a master and two worker/slave architectures. This will provide enhanced container failover in the event that a virtual machine fails or reboots. They distribute the load evenly among all application containers by using the Round Robin algorithm.

The six parts that make up the suggested system provide a complete solution. The setup phase, the initial module, starts the system configuration. The system creates an infrastructure for Kubernetes Cloud and Containers in Module 2, setting the stage for an adaptable and scalable deployment. Module 3 concentrates on application deployment, guaranteeing a smooth software integration. The deployment of backend scripts, which carry out the suggested algorithm, is introduced in Module 4. In order to maintain traffic balance, these scripts dynamically scale up or down web server containers based on predetermined limitations while concurrently inserting them behind a load balancer. They achieve this by continually monitoring real-time traffic trends. In order to maximize resource efficiency and system stability, Module 5 places a strong emphasis on dynamic scaling and high availability. Ultimately, Module 6 produces comprehensive analysis reports that offer insightful information about the functionality and traffic patterns of the system. When combined, these modules create a stable and flexible architecture that can effectively handle containerized apps in a demanding and changing environment.

For large traffic platforms and apps, the study developed a dynamic scaling algorithm that takes backend latency into account. The system will dynamically scale up and down in response to user requests and traffic demand.

[4] A multi-cloud environment is used by Cloud-Architects to eliminate vendor dependence. Utilizing numerous cloud service providers and providers within a single cloud network infrastructure is known as multi-clouding. Utilizing multi-cloud settings allows for the distribution of computing resources, low downtime, and high data availability—all advantages offered by various suppliers.

This multi-cloud will be a hybrid environment that leverages various infrastructure settings, including public and private clouds. Selecting a cloud solution from a single vendor has benefits and drawbacks. Positively, dealing with a single party streamlines negotiations and agreements by simplifying the process. When there are only one vendor handling all of the work, it makes it less likely for people to point the finger at one another when something goes wrong. Dealing with a single source makes support and services easier to understand and allows for seamless service integration. There is less of a requirement for in-depth user training, and compatibility problems are less common. This strategy, nevertheless, is not without its problems. Price talks might not go as well if you have less negotiating power. Reliance on the policies of a single vendor may restrict options and freedom of choice, thus resulting in vendor lock-in problems.

Cloud with Multiple Vendors: Anthos: - Google's take on the robust Istio open-source project is called Anthos Service Mesh, which lets you monitor, control, and secure your services without requiring any changes to your application code. Multi-public cloud architecture offers many benefits, including low latency, negotiation power, risk reduction, and the ability to leverage the distinctive qualities of many vendors' platforms to meet specific business needs and promote innovation. A variety of technologies, including Prometheus, Istio, Grafana, and Kubernetes, are employed to make this concept a reality. Two clusters running on the infrastructure of cloud suppliers are connected via a VPN connection. Here, Kubernetes is crucial to the management of workloads across cloud platforms and to the automation of several manual processes, including rollouts, rollbacks, and horizontal scaling.

III. TOOLS AND TECHNOLOGY

Kubernetes (Minikube): Kubernetes cluster with minimum of two nodes master and slave

Prometheus, Grafana (monitoring tools) and Kubernetes metric server.

Kiali, graphical visualizer of the microservice application

Locust: load generating tool for testing the application

WinSCP and PuTTY: tools to remote login to live server instance and deploy application

IV. CUSTOM SCHEDULER

Kubernetes, a container orchestration platform, provides various auto-scaling mechanisms to efficiently manage resource utilization and application performance. The Horizontal Pod Auto-scaler (HPA), Vertical Pod Auto-scaler (VPA), and Cluster Auto-scaler are the primary built-in auto-scalers, each addressing specific scaling needs.

The HPA dynamically adjusts the number of replicas (pod instances) based on resource metrics, primarily CPU or memory utilization, ensuring optimal resource allocation and performance for applications with predictable workloads.

The Custom Autoscaler, operating at the cluster level, scales the number of nodes based on the overall resource demands of pods within the cluster. This mechanism is particularly useful for dynamic workloads or scenarios requiring high availability.

Kubernetes custom controllers offer tailored functionalities that can be more specific and targeted compared to the default controllers. While default controllers cover general use cases, custom controllers are built to address specific needs or behaviors within a Kubernetes cluster.

Custom controllers extend Kubernetes' capabilities by allowing users to define and implement their own control loops to manage custom resources. These controllers watch for specific resource types (Custom Resources) and implement logic to reconcile the desired state with the current state of those resources.

A. Advantages of a Custom Auto Scaling Controller over Default Controller:

Tailored Logic: Custom controllers allow for personalized logic and decision-making in auto scaling based on unique metrics or requirements specific to your application.

Specialized Scaling Policies: They enable the creation of scaling policies that might not be achievable with the default controller, like more advanced scaling algorithms or integrating with external systems for metrics.

Resource Optimization: Custom controllers can optimize resource allocation precisely for your workload patterns, potentially leading to cost savings and better performance.

B. Implementation of Custom Controller Algorithm

The Nginx application was used as a microservice and installed on a Kubernetes cluster during the project's initial phase. The application pod was successfully launched after a seamless deployment procedure, and a service was set up to facilitate simple browser access. The Locust tool was used to inject simulated traffic and measure the application's scalability and responsiveness in order to evaluate the system's performance under various loads. The foundation for the later integration of a unique controller algorithm was established by this first configuration.

The project's core is a proprietary controller algorithm written in Python that is carefully crafted to improve Nginx pod orchestration in a Kubernetes context. The algorithm was contained within a Docker image, creating a bespoke scheduler, to be seamlessly integrated into the deployment routine. In order to do this, a Dockerfile defining the Python code execution had to be created. The intricate details of the specific scheduling procedure were then handled by creating a separate pod. When demand was generated via Locust, the Nginx pods dynamically changed the number of deployments they had based on calculations made by the proprietary algorithm. This demonstrated how flexible and responsive the developed solution was to changing workloads. This thorough method shows how a custom controller algorithm may be successfully integrated into the Kubernetes environment.

When it comes to dynamic pod scheduling in the Kubernetes environment, a flexible and responsive custom controller algorithm is critical to the system's performance. The Nginx application in our case study is an example of a microservice that is deployed and scaled with the help of our Python technique. The main goal is to ensure responsiveness and optimal resource utilization by dynamically adjusting the number of pods assigned to the application in response to changing loads.

The figure 1 shows the code snippet which calculates the number of pods to be scheduled when the current CPU usage value is greater than the target CPU usage value of the pod. However, when the current CPU usage is less than the target CPU usage, there is minor change in the logic. The figure 2 shows the logic to remove or discord pods when the load has drastically decreased.

```
scaling_factor = (cpu_usage_m - target_cpu_value) / 100.0
predicted_pods = round(max_pods - (max_pods - min_pods) * scaling_factor)
predicted_pods = max(min_pods, min(predicted_pods,max_pods))

//cpu_usage_m - current cpu usage value
//target_cpu_value - target cpu usage value
//max_pods - maximum number of pods
//min_pods - minimum number of pods
```

Fig 1. Algorithm to calculate the pods to be scheduled

```
scaling_factor = (cpu_usage_m - target_cpu_value) / 100.0
predicted_pods = round(max_pods - (max_pods - min_pods) * scaling_factor)
predicted_pods = min(min_pods, min(predicted_pods,max_pods))

//cpu_usage_m - current cpu usage value
//target_cpu_value - target cpu usage value
//max_pods - maximum number of pods
//min_pods - minimum number of pods
```

Fig 2. Logic to calculate the pods to be discarded

In order to obtain real-time CPU utilization values from the Nginx pods that are now executing, the custom controller algorithm cleverly makes use of the Kubernetes Metric Server. Following that, these variables play a crucial role in the algorithm's computations, which enable it to dynamically decide how many pods to schedule or reduce in order to successfully adapt to variations in system load. This flexible

strategy, when combined with Kubernetes orchestration, shows how dedicated we are to effective resource management and long-term performance in microservices implementation.

C. Experiment and Testing

Using the Locust tool, the dynamic pod scheduling system was experimentally evaluated under two different workloads. A load test with 3000 users producing 600 requests per second (RPS) was performed first. A second test with 200 users producing 50 RPS was then carried out. An observation of the dynamics of pod creation and deletion was conducted during the first load test involving 3000 users in order to measure the volume and speed of these events. The collected information was methodically arranged in a tabular style, registering the pods generated at three different time intervals: two minutes, eight minutes, and fifteen minutes. The experiment sought to provide important insights into the system's dynamic scaling capabilities by thoroughly understanding the system's adaptability and efficiency under various workloads.

The same kind of observational method was used in the next workload scenario with 200 users and 50 RPS. Tracking pod formation and deletion metrics across the same three time intervals two minutes, eight minutes, and fifteen minutes was the main objective of the experiment. This methodology made it easier to compare things and gave researchers a more complex picture of how the dynamic pod scheduling system handled different workloads. The captured observations offer a comprehensive viewpoint on the system's capacity for dynamic resource scaling, offering important information for microservices deployment optimization in practical contexts.

During the testing phase, the performance of the custom scheduler was compared to that of the Kubernetes Horizontal Pod Autoscaler (HPA) using the same workloads. Using the Locust tool, tests were carried out with 200 users producing 50 requests per second (RPS) and 3000 users producing 600 RPS.The load was decreased drastically from 3000 users to 200 users and the result and behavior is recorded in the tables. The subsequent observations were centered on monitoring the quantity of pods generated by the default HPA inside three distinct time intervals: two minutes, eight minutes, and fifteen minutes. The comparative analysis yielded significant information regarding the relative performance and adaptability of the dynamic pod scheduling system over the default Kubernetes HPA.

TABLE 1. NUMBER OF PODS SCHEDULED WHEN LOAD IS 3000 USERS AND 600 RPS.

Algorithm Used (for 3000 users, 600 RPS)	***KHPA***	***Custom Scheduler***
2 minutes	9	2
8 minutes	10	3
15 minutes	10	5

TABLE 2. NUMBER OF PODS SCHEDULED WHEN LOAD IS 200 USERS AND 50 RPS

Algorithm Used (for 200 users, 50 RPS)	***KHPA***	***Custom Scheduler***
2 minutes	10	2
8 minutes	5	1
15 minutes	1	1

V. RESULTS AND DISCUSSION

The study of the collected data revealed unique patterns in the scheduling of pods between the Kubernetes Horizontal Pod Autoscaler (KHPA) and the custom scheduler at different workloads. KHPA tended to schedule more pods than the custom scheduler, according to the tabulated numbers of pods scheduled for various loads and algorithms. Interestingly, this difference was most noticeable when the system load was high, highlighting KHPA's tendency towards more aggressive scaling.

Analyzing the two algorithm's response times under varying loads revealed a perceptive finding. The custom scheduler adjusted more quickly when the system load dropped, effectively reducing the number of pods. KHPA, on the other hand, responded more slowly, taking a longer period to lower the number of pods as the demand dropped. This disparity in response demonstrates the custom scheduler's dynamic adaptability, which showed a quicker and more effective approach to resource usage amid load variations.

The investigation also clarified the effectiveness of both algorithms' resource usage. KHPA tended to schedule more pods in cases when the initial load was higher, which could result in higher resource use. By scheduling fewer pods at first, the custom scheduler, in comparison, showed a more restricted approach and a more effective resource utilization technique. In a dynamic microservices deployment scenario, this observation highlights the subtle trade-offs between the two algorithms with respect to responsiveness, resource consumption, and scalability.

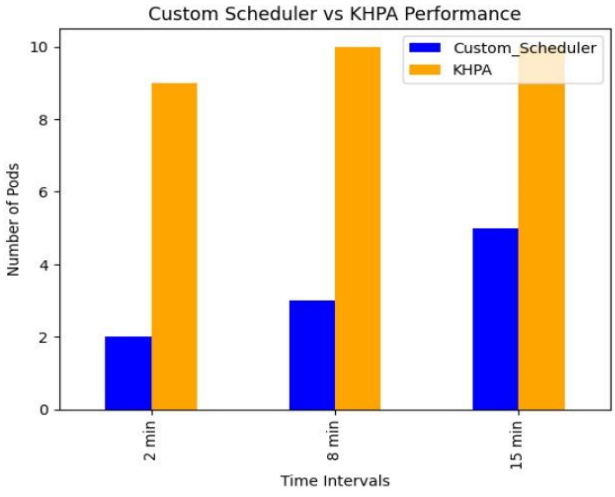

Fig 3. Comparison between Custom Scheduler and KHPA during 3000 users and 600 RPS

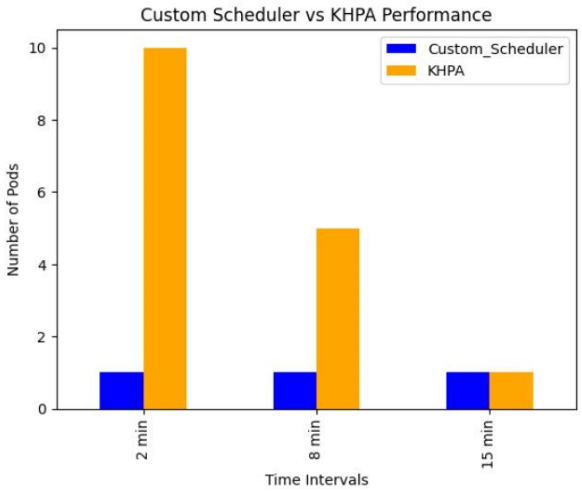

Fig 4. Comparison between Custom Scheduler and KHPA during 200 users and 50 RPS

VI. CONCLUSION AND FUTURE WORK

When it came to dynamic scaling scenarios, the custom controller auto-scaler outperformed the Kubernetes Horizontal Pod Autoscaler (KHPA). Above all, the custom scheduler was remarkably sensitive, quickly modifying the number of pods in response to varying system loads. It demonstrated a more responsive behavior during load reductions, effectively lowering pod counts in contrast to the comparatively sluggish adaptation shown with KHPA.

In addition, the customized scheduler demonstrated a clever approach to resource management under high workloads. The custom scheduler started off with fewer pods than KHPA since it took a more conservative approach. This could have resulted in increased resource use. This demonstrated its ability to strike a balance between efficiency and performance, which makes it a reliable option for microservices deployment in Kubernetes environments.

A number of directions might be investigated for future work to improve the system even more. First off, adding extra metrics like memory utilization or response time beyond CPU usage can offer more sophisticated management over microservices scaling. Proactive resource allocation can also be improved by adding predictive scaling, which uses machine learning on past data to predict future loads. Finally, deployment procedures can be automated and continuous

Performance feedback can be obtained for continual enhancements by linking the custom scheduler with CI/CD pipelines. The aforementioned future directions are intended to enhance the custom controller auto-scaler's scalability, efficiency, and adaptability in dynamic microservices contexts.

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.

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

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

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

REFERENCES

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

2. Shitole, Abishek Sanjay (2022) Dynamic Load Balancing of Microservices in Kubernetes Clusters using Service Mesh. https://norma.ncirl.ie/5943/ Masters thesis, Dublin, National College of Ireland.

3. An Efficient and Scalable Traffic Load Balancing Based on Web Server Container Resource Utilization using Kubernetes Cluster. (May – 2022) Ashok L Pomnar, 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