Journal of Intelligent Management Decision
https://www.acadlore.com/journals/JIMD

Security-Enhanced QoS-Aware Autoscaling of Kubernetes Pods Using Horizontal Pod Autoscaler (HPA)

Vani Rajasekar[1], Muzafer Saračević[2], Darjan Karabašević[3*], Dragiša Stanujkić[4], Amor Hasić[2], Melisa Azizović[2], Srivarshan Thirumalai[1]

[1] Kongu Engineering College, Thoppupalayam, Kumaran Nagar, 638060 Tamil Nadu, India
[2] Department of Computer Sciences, University of Novi Pazar, 36300 Novi Pazar, Serbia
[3] Faculty of Applied Management, Economics and Finance, University Business Academy in Novi Sad, 11000 Belgrade, Serbia
[4] Technical Faculty in Bor, University of Belgrade, 19210 Bor, Serbia

* Correspondence: Darjan Karabašević (darjan.karabasevic@mef.edu.rs)

Received: 08-02-2024 **Revised:** 09-10-2024 **Accepted:** 09-20-2024

Citation: V. Rajasekar, M. Saračević, D. Karabašević, D. Stanujkić, A. Hasić, M. Azizović, and S. Thirumalai, "Security-enhanced QoS-aware autoscaling of Kubernetes pods using Horizontal Pod Autoscaler (HPA)," *J. Intell Manag. Decis.*, vol. 3, no. 3, pp. 175–186, 2024. https://doi.org/10.56578/jimd030304.

Abstract: Container-based virtualization has emerged as a leading alternative to traditional cloud-based architectures due to its lower overhead, enhanced scalability, and adaptability. Kubernetes, one of the most widely adopted open-source container orchestration platforms, facilitates dynamic resource allocation through the Horizontal Pod Autoscaler (HPA). This auto-scaling mechanism enables efficient deployment and management of microservices, allowing for rapid development of complex SaaS applications. However, recent studies have identified several vulnerabilities in auto-scaling systems, including brute force attacks, Denial-of-Service (DoS) attacks, and YOYO attacks, which have led to significant performance degradation and unexpected downtimes. In response to these challenges, a novel approach is proposed to ensure uninterrupted deployment and enhanced resilience against such attacks. By leveraging Helm for deployment automation, Prometheus for metrics collection, and Grafana for real-time monitoring and visualisation, this framework improves the Quality of Service (QoS) in Kubernetes clusters. A primary focus is placed on achieving optimal resource utilisation while meeting Service Level Objectives (SLOs). The proposed architecture dynamically scales workloads in response to fluctuating demands and strengthens security against autoscaling-specific attacks. An on-premises implementation using Kubernetes and Docker containers demonstrates the feasibility of this approach by mitigating performance bottlenecks and preventing downtime. The contribution of this research lies in the ability to enhance system robustness and maintain service reliability under malicious conditions without compromising resource efficiency. This methodology ensures seamless scalability and secure operations, making it suitable for enterprise-level microservices and cloud-native applications.

Keywords: Kubernetes; Quality of Service (QoS); Horizontal Pod Autoscaler (HPA); Attacks; Data protection

1 Introduction

Cloud computing has become incredibly popular in recent years. Companies are deploying premises networks and public clouds in greater numbers. Online services are used for most real-time applications, including banking, education, and health care. These services are crucial for the same models that are set up in clouds [1]. The resource provision feature of cloud computing, which provides quick elasticity and dynamic scalability, is one of its key benefits. Scaling pods is by default based solely on CPU utilization, a single performance parameter. The proposed methodology for achieving the scenario involves the use of several tools that work together seamlessly. Docker is a popular tool that is used for containerization, which refers to the process of packaging an application along with its dependencies and configuration files into a single unit known as a container. With Docker, it is possible to create a Docker image that contains all the necessary components for running an application. These images can then be used to create Docker containers that can be run on any machine that has Docker installed. Kubernetes, on the other hand, provides a platform for managing and deploying containerized applications. It enables users to automate the deployment, scaling, and management of containerized applications. By using Kubernetes, it is possible to

deploy a containerized application across multiple nodes, making it easier to manage and scale [2, 3]. Helm is a package manager for Kubernetes that enables users to easily deploy and manage applications on a Kubernetes cluster. It provides a simple way to define, install, and upgrade Kubernetes applications. Prometheus is an open-source monitoring system that is used to collect metrics from various sources. It is widely used for monitoring Kubernetes clusters, and it is capable of collecting metrics from various components of the cluster, including nodes, pods, and containers. IaaS clouds have on-demand scaling capabilities. Scalability, however, is a challenge in SaaS setups. To increase service quality, existing programs must reside together with new apps and services. Businesses today need a mobile and interoperable environment that can scale and adapt to user needs to incorporate all these constraints into the systems. A pod is guaranteed to receive a minimum amount of CPU time for each request in Kubernetes [4]. HPA automatically scales copies of pods based on observed CPU utilization by default.

The application may crash if it experiences stress and the number of incoming requests exceeds the pod's capacity. To prevent this, Kubernetes implements pod auto-scaling, which can be monitored by Grafana and managed through alerts from Prometheus. Before deploying the application in the EKS cluster, the proposed approach first sets up all essential services to establish the required infrastructure. Helm coordinates the deployment, service management, and horizontal auto-scaling within the Kubernetes cluster. Prometheus is installed in the cluster to collect data about the environment where the application runs. This data is then visualized on an interactive dashboard in Grafana, which displays alerts and other relevant information.

The Elastic Kubernetes Service (EKS) is provisioned in Amazon Web Services (AWS), and the Docker image is deployed into the EKS cluster using Helm. HPA in Kubernetes facilitates pod auto-scaling during periods of high demand or upgrades. Figure 1 illustrates the application deployment along with the necessary AWS infrastructure, which includes services such as AWS EC2, EKS, Relational Database Service (RDS), node groups, auto-scaling groups, security groups, and Virtual Private Cloud (VPC).

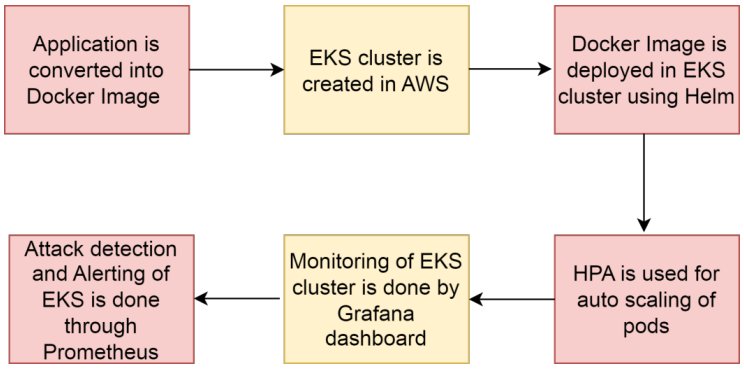


Figure 1. The flow of the proposed methodology

Note: This figure was prepared by the authors

The cloud VMs' autoscaling system is the target of the attacks. The auto-scaling mechanism swings between scale-up and scale-down phases as a result of the recurring spikes in traffic volumes. Along with causing financial harm, the attacks significantly reduce performance. The cloud service experiences a significant performance hit throughout the recurrent scale-up process, which typically lasts up to a few minutes. The attacker stops transmitting traffic when the scale-up procedure is complete and waits for the scale-down process to begin. The attacker launches another attack after the scale-down phase is finished, and so on [5–8]. Additionally, there are more machines after the scale-up process but fewer users. As a result, the victim unknowingly pays for extra, unused equipment.

While resource saturation may lead to SLO breaches, the precise association has not yet been formally established. This is an important aspect to keep in mind. A perfect autoscaling system should respond immediately to changes in application-level metrics, such as rising SLO violation rates. Designing and implementing a monitoring system that keeps track of and combines monitoring information from resources in data centers and microservices is a difficult task. Using this data, the QoS service implements rules and takes action to ensure that the SLOs for resource utilization are met. To overcome the aforementioned issue, the proposed method suggests enhancing the Kubernetes cluster autoscaler with unique rules based on the state-of-the-art existing methodology. This study provides a multi-tier architecture with a monitoring system and QoS services to measure the workload and usage of resources across all tiers and layers and to constantly adjust to satisfy QoS and SLO requirements.

The major contributions of the proposed approach are:

- Implementation of Kubernetes clusters over on-premise AWS cloud infrastructure.
- Dynamic autoscaling of pods using HPA to meet the SLA requirement on scale-up and scale-down.
- Detection of attack and its mitigation through continuous monitoring using Helm and Grafana.
- Ensuring the better SLO requirement for improved resource usage and guaranteed QoS.

2 Literature Review

In software architecture, virtual machines are a crucial intermediate layer. The advantages of containers over virtual machines have been the subject of extensive research due to the growing popularity of container-based virtualization solutions. Containers provide advantages for Platform-as-a-Service (PaaS) clouds because of their simplicity in application management, deployment, and setup. Zhu et al. [9] suggest a bi-metric strategy for scaling pods that accounts for both CPU usage and thread pool usage, which is a type of significant soft resource in HTTPD and Tomcat. Our method measures how much each pod's CPU and memory are being used. In the meantime, it uses ELBA, a milli-bottleneck detector, to determine the queue lengths of the Httpd and Tomcat pods and then assess how well their thread pools are being used. Container orchestration has huge potential to advance distributed cloud services and network management. A series of experiments were conducted to demonstrate the improvement of containers over virtual machines with performance evaluations. Ruíz et al. [10] describe a Kubernetes and Docker-based on-premise architecture that aims to enhance the QoS in terms of resource use and SLOs. Their primary contribution is their ability to dynamically autoscale the resources to match the present workload while enhancing QoS. Orchestration will be a beneficial addition to the design because it will automate the behavior required to achieve reliability and responsiveness. To properly match the orchestrator's operations with the platform's SLOs, however, it might be necessary to specify unique orchestration rules. Developing a unique monitoring service and enhancing the QoS service will be essential.

The current Kubernetes resource allocation policy for application instances cannot dynamically modify following business requirements, resulting in significant resource waste during fluctuations. Additionally, the introduction of new cloud services has increased the need for resource management. The adaptive horizontal pod auto-scaling system (AHPA), a recently implemented AI algorithm framework, is used in this study to examine the management of horizontal POD resources in Alibaba Cloud Container Services. AHPA presents an ideal pod number adjustment strategy that could reduce POD resources and keep company stability based on a reliable decomposition forecasting algorithm and performance training model [11–13]. With the advancement of cloud computing technologies, more and more businesses are adopting and promoting the Kubernetes-led container management architecture. With the help of one of its features, Autoscaling, clients' dynamic requirements can be handled automatically. Although it appears that K8s' autoscaling solution is adequate, the majority of scaling solutions, including HPA, are based on reactive autoscaling, which uses CPU or memory utilization as a metric. To obtain or allocate computing resources on request, autoscaling is an important notion of cloud infrastructure. It enables users to automatically scale the resources provided to applications under fluctuating workloads to reduce resource costs while meeting QoS demands.

The widely used popular container orchestration system, Kubernetes, contains built-in autoscalers to address the autoscaling issues at the container level for both horizontal and vertical scaling, although it still has significant restrictions. Because of a predetermined number of resources for each instance, only horizontal scalability may result in low container utilization, especially during periods of low demand. In contrast, if the workload spikes owing to hitting the upper limit, just vertical scaling might not be able to maintain the required QoS. In addition, to ensure service performance, burst identification for auto-scalers is also required [14, 15]. Microservices have been acknowledged by the cloud community as the leading architecture for putting cloud-native applications into practice. Application owners must carefully scale the necessary resources to effectively run microservice-based applications while taking into account the fluctuating workload of each of the microservices. The difficulty of resource supply for these applications emphasizes how important autoscaling methods are [16, 17]. Dogani et al. [18] suggest a multi-objective autoscaling approach to prevent resource waste and achieve the required average response time of microservices. They provide a toolchain for microservices in autoscaling with hybrid nature in Kubernetes based on machine learning (ML) approaches. They also suggested the best model for resolving the issue after comparing various ML approaches and our performance modeling tool, termed Extra-P. A thorough analysis of a benchmark component demonstrates a considerable decrease in resource utilization while still matching the user-specified average response time, outperforming the outcomes of the standard HPA in Kubernetes.

The timely management of virtualized resources is necessary to maximize the performance of contemporary applications. However, it can be very difficult to deploy resources proactively to satisfy specific application requirements when there is a fluctuating workload from incoming requests. To do this, it is necessary to jointly solve the basic issues with task scheduling and resource autoscaling [19]. To address both, a scalable design that is compatible with Kubernetes' decentralized structure is described in previous studies [10, 20–23]. They dynamically reroute requests that arrive at the containerized application by utilizing the stability guarantees of a unique Additive Increase Multiplicative Decrease (AIMD)-like task scheduling mechanism. A prediction method enables us to predict the number of incoming requests to handle fluctuating workloads. The scaling issue is also addressed by introducing ML-based application profiling modeling that integrates the theoretically computed service rates acquired from the AIMD technique with the existing performance measurements. The remainder of the paper is organized in such a way that Section 3 describes the proposed methodology, which describes the AWS services, Grafana, Prometheus, and Helm. The deployment strategy is also explained with the EKS cluster in Kubernetes. Section 4 discusses the

results and evaluation, where the HPA is demonstrated along with the CPU usage analysis. Section 5 discusses the conclusion of the proposed work, and future enhancements concerning the proposed methodology are also given.

3 Proposed Methodology

3.1 Preliminaries

The architecture necessary for deploying the application on AWS is illustrated in Figure 2. A VPC is set up in AWS to host the required services. An Internet Gateway (IG) is established to enable external access to these services. For database needs, AWS RDS is utilized. Amazon EKS manages the deployment orchestration for the application within AWS. Once the EKS cluster is deployed, a node group is created using AWS EC2, which serves as the cloud computing platform for application deployment. Security responsibilities are shared between both users and AWS.

For that, security groups are created to protect the service against unknown threats. All the services that are created will be maintained by both AWS and the user. The cloud architecture of the deployment for the application is shown in Figure 3.

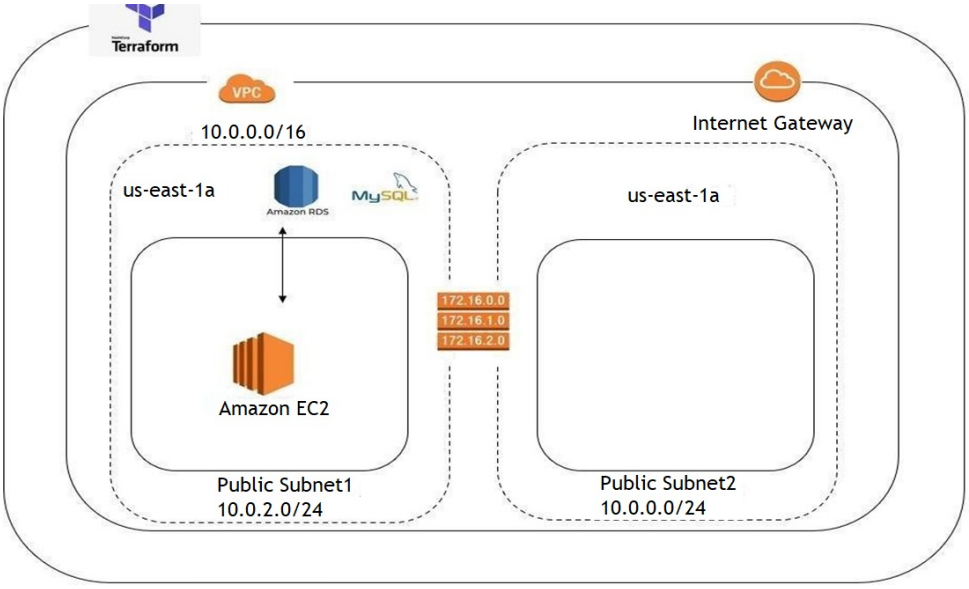


Figure 2. AWS architecture

Note: This figure was prepared by the authors

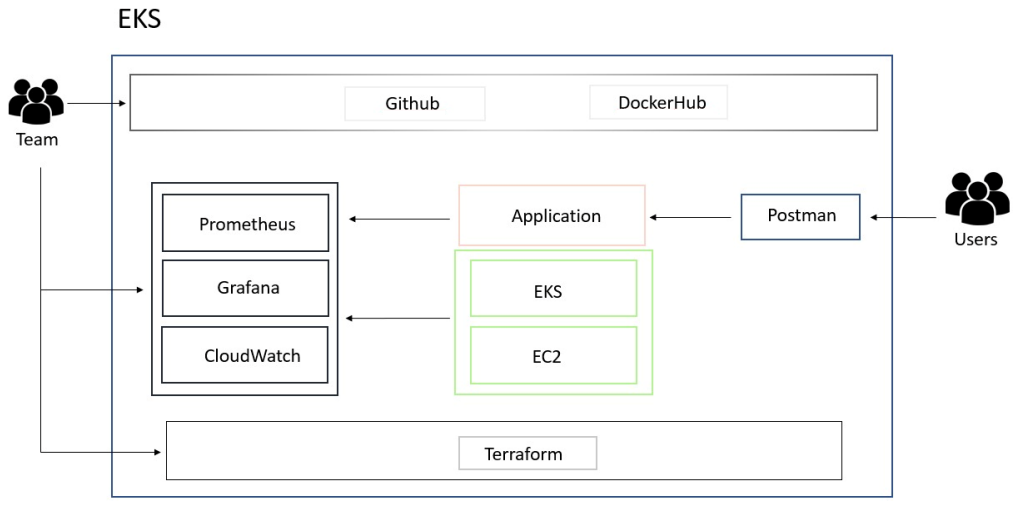


Figure 3. EKS architecture

Note: This figure was prepared by the authors

The application code and Docker file are uploaded to GitHub. A Docker image is then generated to package the application, which is subsequently deployed to an EKS cluster on AWS. Inside the Kubernetes cluster, Prometheus is

set up to collect data and issue alerts if any issues arise. To manage workload fluctuations, HPA is implemented to automatically adjust the number of pods during peak usage or upgrade processes. Helm is utilized as an orchestration tool to ensure seamless application deployment with no downtime. Grafana provides an interactive dashboard to visualize the data and alerts gathered by Prometheus. This setup allows for straightforward interaction with the Kubernetes cluster and efficient use of AWS services.

3.2 Docker and Kubernetes

Kubernetes simplifies the management of applications by automating container-related operational tasks and providing built-in commands for deployment. It facilitates rolling updates, dynamic scaling to meet varying demands, and monitoring. Known as K8s, this open-source container orchestration platform was developed by Google to manage and organize clusters of containerized applications—a process referred to as orchestration.

In Kubernetes, the smallest deployable units are called Pods, each representing an individual instance of an active process within the cluster. A Pod can contain one or more containers, such as Docker containers, which share resources and are managed collectively. Helm Charts play a crucial role in defining, installing, and upgrading even the most complex Kubernetes applications. Helm is a deployment tool that automates the development, packaging, setup, and distribution of applications to Kubernetes clusters, functioning like a package manager in an operating system. Just as CentOS uses yum and Ubuntu uses apt, Kubernetes employs Helm to manage packaged applications, organized as charts, before deployment.

With simple command-line interface (CLI) commands, Helm streamlines the installation, updating, and configuration processes in Kubernetes. It significantly reduces the time needed for developers to deploy and test environments, facilitating a smooth transition from development to production. Moreover, Helm provides developers with an efficient way to package and distribute applications for end users. The architecture of Helm is illustrated in Figure 4.

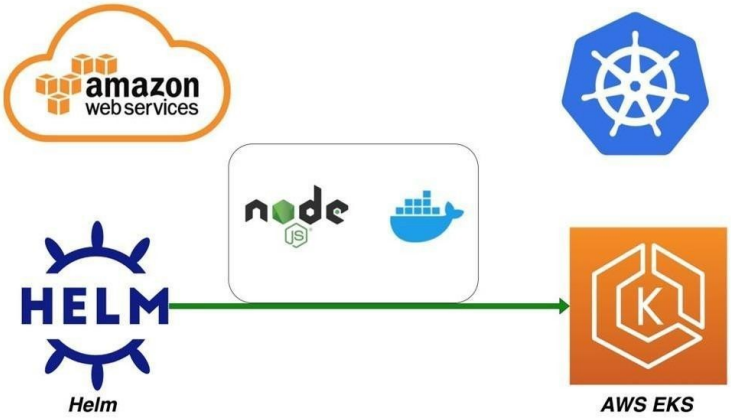


Figure 4. Helm

Note: This figure was prepared by the authors

Prometheus is an open-source toolkit designed for monitoring and alerting in containerized environments and microservices. It provides flexible query capabilities and real-time notifications. When issues arise with APIs or other linked applications, Prometheus alerts the IT department, helping them monitor and address problems effectively. The tool also identifies unusual traffic patterns that may indicate potential security threats or vulnerabilities.

Grafana, developed by Grafana Labs, is an open-source platform for interactive data visualization. It allows users to display their data through a unified dashboard featuring charts and graphs for easier analysis and understanding. Grafana Cloud enhances this experience by integrating metrics, traces, and logs into one cohesive interface.

3.3 Deployment

A deployment file is created to oversee the application deployment process. To facilitate external access to the application, a service file is utilized to establish a URL or load balancer. Additionally, a horizontal autoscaling file is employed to enable the automatic scaling of pods during application upgrades. The deployment configuration for a Kubernetes object, which can generate and update a set of identical pods, is defined in YAML format. Each pod runs specific containers indicated in the spec.template section of this YAML configuration.

The Deployment object not only creates the pods but also manages their scalability, oversees continuous updates, and ensures that the correct number of pods is operational in the cluster. Various fields in the deployment YAML are used to configure these functions. An example of the deployment YAML can be found in Appendix A1, detailing how to manage the application deployment.

In Kubernetes, a Service is a REST object similar to a pod. A service definition can be submitted to the API server to create a new instance, adhering to the label name standards set by RFC 1035. A Service object can also be configured with multiple port definitions. To access the application externally, the YAML service outlined in Appendix A2 must be used.

3.4 Horizontal Auto Scaling

By continually updating the workload assets, the proposed HPA in Kubernetes aims to proactively grow the workload following demand. Horizontal scaling is the process of adding new Pods when a load rises. Contrary to vertical scaling, which Kubernetes would imply, the task currently being handled by Pods would receive greater resources. If the load reduces, the HPA notifies the workload resource, installation, stateful set, or additional resource and scales back down. The results of the horizontal pod autoscaling do not affect objects that cannot be scaled. Both a controller and an API resource from the Kubernetes platform are used to implement the HPA. The resource controls the actions of the controller. The metrics like average CPU or memory usage are by the horizontal pod autoscaling controller in the Kubernetes control plane, which modifies the target's desired scale periodically. When the application is stressed, auto-scaling of pods is deployed using the HPA shown in Appendices A3 and A4.

The HPA informs the demand on the resource that it should scale back down if the demand decreases and the total amount of Pods is greater than the specified threshold. For stationary things, HPA does not apply. The HPA operates as a controller and a resource in the Kubernetes API. The resource dictates how the controller will act. The Kubernetes control plane's horizontal pod autoscaling controller regularly modifies the objective's target scale to correspond to observable metrics like mean CPU and mean memory utilization. The controller manager does a query on resource utilization against the metric listed in each HPA specification once per period. The controller manager locates the desired resource specified by the scalTargetRe, chooses the pods to use following the target source's requirements, and then retrieves the metrics that are needed using either the resource metrics API for each pod resource metrics or the customized metrics API for remaining all other metrics.

4 Results and Discussion

4.1 Scaling Based on Pod Load

The process of deploying an application on the cloud can be a daunting task, especially when there are multiple services and components involved. However, with the recommended technique, the process becomes more straightforward and efficient. The first step is to transform the application into a Docker image, which is a self-contained package that includes all the dependencies and libraries needed to run it. This image is then deployed on the AWS platform, where all the necessary services are installed. To manage the deployment of the Docker image, the AWS EKS cluster is used. EKS is a fully managed service that simplifies the deployment and scaling of containerized applications. In addition to EKS, a Load Balancer is also used to enable access to the application from outside the system. This LoadBalancer helps to distribute the incoming traffic across the pods in the EKS cluster, ensuring that the application is highly available and responsive. To enable horizontal auto-scaling of the application, Kubernetes is used to automatically scale the pods in the cluster when the system is under stress or any upgrade activity is performed. This ensures that the application can handle increased traffic or workloads without any performance degradation. The Docker image in ECR is shown in Appendix A5.

To manage the Kubernetes components and ensure zero downtime while upgrading the application, Helm is used. It provides a simple and flexible way to manage the deployment of applications on Kubernetes, making it an ideal tool for managing the deployment of the Docker image. To monitor the EKS cluster and obtain information about its performance, Prometheus is launched inside the cluster. Prometheus is a monitoring system that collects metrics from different sources and stores them in a time series database. Prometheus rules are established to send alarms to email, Slack, Microsoft Team, Opsgenie, and other services, which helps ensure that any issues are promptly addressed. Finally, Grafana is integrated with Prometheus to provide dynamic dashboards that show data about the performance and health of the system. This allows for easy visualization of data and helps identify potential issues before they become critical (see Appendix A6). By combining and optimizing different tools, the suggested technique provides a highly efficient solution for the deployment of containerized applications. The use of continuous integration and deployment using ArgoCD, a declarative GitOps continuous delivery solution based on Kubernetes, also ensures that the application can be easily updated and maintained. Overall, the suggested technique provides a robust and scalable solution for deploying applications on the cloud (scaling of the pod using HPA is shown in Appendix A7).

4.2 Scaling Based on Pod Load

To assess the proposed strategy, we run several experiments in this part. Since a pod's CPU request in Kubernetes often does not exceed one CPU, our experiment sets consider two CPU request types: 0.5 CPU and 1 CPU. Because many pods are frequently installed on the same host, co-location is also taken into consideration. The tests were run to demand the platform and observe how it responded. First, a preliminary study is done to see if the specific

architecture mechanisms such as scalability and work queuing are following SLOs. Following that, a comprehensive result analysis is carried out to gain insight into the platform's normal behavior and identify instances in which it performs better than a conventional Kubernetes cluster (see Figure 5).

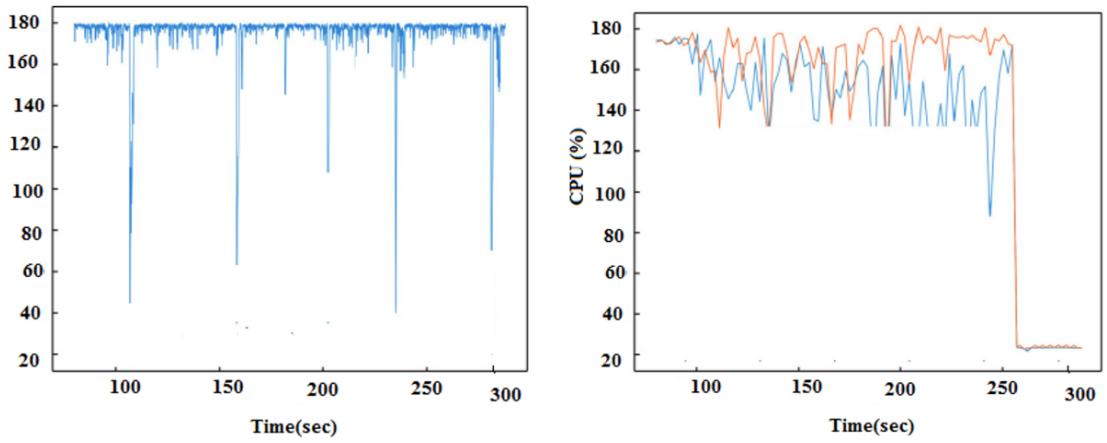


Figure 5. Queue length and CPU utilization of Pod

Note: This figure was prepared by the authors

Thus, our goal is to demonstrate the circumstances in which the suggested QoS technique improves upon existing tools like the default QoS methods in Kubernetes and gets higher performance in terms of using computational resources while resolving some of the problems with typical Kubernetes clusters. Since CPU and memory utilization are the tracked metrics under the current system, they are also the focus of this results section. To replicate stressful situations, CPU and memory wasters were used in a setting that included older apps, APIs, databases, and other tools. This section demonstrates the system's reaction. All of these systems would ideally cooperate and share resources. The CPU utilization Grafana dashboard is shown in Appendix A4. The results of this study show that observing QoS restrictions is essential for guaranteeing security satisfaction. The execution findings demonstrate the proposed architecture's potential for maintaining compliance with security requirements at both the software and computer hardware levels. Even though the measurements used, focused on CPU and memory utilization, are encouraging, the system's effectiveness in terms of common security requirements. However, this solution is still far from being finished and commercially viable. The tests made clear some problems that must be resolved before using the architecture commercially. The platform would be regarded as being in its bare minimum marketable condition if these conditions were satisfied. On the other hand, several enhancements are desired to simplify deployment, enhance performance, and therefore, improve the user experience. It would be possible to reduce in-script modifications by using global configurations in the cloud that include parameters for every service in the architecture. Next, each supported programming language should have a separate procedure submission that is suited to it. As a result, dependencies would be minimized, and there would be transparency.

5 Conclusions

The proposed methodology is for monitoring and alerting of auto-scaling pods in Kubernetes using Prometheus in cloud computing with AWS as the cloud provider. The suggested technique deployed all services to build the necessary infrastructure for deploying the application in the EKS cluster. Helm orchestrates Kubernetes cluster deployment, service, and horizontal auto-scaling. Prometheus is installed within the Kubernetes cluster and is used to scrape all data about the cluster in which the application is deployed. Grafana is used to present an interactive dashboard with alarms and information about the Kubernetes cluster, where data is scraped from the cluster using Prometheus.

The recommended solution eventually resulted in the application being deployed with minimal downtime once the upgrading work was completed. When the application is affected by any security attacks, the pods are scaled using Kubernetes' horizontal autoscaling feature. In conclusion, all of these findings support carrying out the proposed research in a cloud environment. They suggest that this is the route for improving the QoS awareness, automaticity, security, and robustness of the current cloud systems.

In the future, the proposed work can be tested with vertical auto-scaling of pods with security requirements.

Data Availability

The data used to support the research findings are available from the corresponding author upon request.

Conflicts of Interest

The authors declare no conflict of interest.

References

[1] D. Balla, C. Simon, and M. Maliosz, "Adaptive scaling of Kubernetes pods," in *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium, Budapest, Hungary*, 2020, pp. 1–5. https://doi.org/10.1109/NOMS47738.2020.9110428

[2] T. T. Nguyen, Y. J. Yeom, T. Kim, D. H. Park, and S. Kim, "Horizontal pod autoscaling in Kubernetes for elastic container orchestration," *Sensors*, vol. 20, no. 16, p. 4621, 2020. https://doi.org/10.3390/s20164621

[3] R. Ben David and A. Bremler Barr, "Kubernetes autoscaling: YoYo attack vulnerability and mitigation," *arXiv e-prints*, 2021. https://doi.org/10.48550/arXiv.2105.00542

[4] L. H. Phuc, L. A. Phan, and T. Kim, "Traffic-aware horizontal pod autoscaler in Kubernetes-based edge computing infrastructure," *IEEE Access*, vol. 10, pp. 18 966–18 977, 2022. https://doi.org/10.1109/ACCESS.2022.3150867

[5] A. A. Khaleq and I. Ra, "Agnostic approach for microservices autoscaling in cloud applications," in *2019 International Conference on Computational Science and Computational Intelligence (CSCI), Las Vegas, NV, USA*, 2019, pp. 1411–1415. https://doi.org/10.1109/CSCI49370.2019.00264

[6] V. Rajasekar, M. P. Vaishnnave, S. Premkumar, V. Sarveshwaran, and V. Rangaraaj, "Lung cancer disease prediction with CT scan and histopathological images feature analysis using deep learning techniques," *Results Eng.*, vol. 18, p. 101111, 2023. https://doi.org/10.1016/j.rineng.2023.101111

[7] M. Niazi, S. Abbas, A. H. Soliman, T. Alyas, S. Asif, and T. Faiz, "Vertical Pod Autoscaling in Kubernetes for elastic container collaborative framework," *Comput. Mater. Continua*, vol. 74, no. 1, pp. 591–606, 2022. http://doi.org/10.32604/cmc.2023.032474

[8] A. Abdel Khaleq and I. Ra, "Intelligent microservices autoscaling module using reinforcement learning," *Cluster Comput.*, vol. 26, no. 5, pp. 2789–2800, 2023. https://doi.org/10.1007/s10586-023-03999-8

[9] C. P. Zhu, B. Han, and Y. L. Zhao, "A bi-metric autoscaling approach for N-tier web applications on Kubernetes," *Front. Comput. Sci.*, vol. 16, p. 163101, 2022. https://doi.org/10.1007/s11704-021-0118-1

[10] L. M. Ruíz, P. P. Pueyo, J. Mateo-Fornés, J. V. Mayoral, and F. S. Tehàs, "Autoscaling pods on an on-premise Kubernetes infrastructure QoS-aware," *IEEE Access*, vol. 10, pp. 33 083–33 094, 2022. https://doi.org/10.1109/ACCESS.2022.3158743

[11] Z. Zhou, C. Zhang, L. Ma, J. Gu, H. Qian, Q. S. Wen, L. Sun, P. Li, and Z. M. Tang, "AHPA: Adaptive horizontal pod autoscaling systems on alibaba cloud container service for Kubernetes," in *Proceedings of the AAAI Conference on Artificial Intelligence, Washington DC, USA*, vol. 37, no. 13, 2023, pp. 15 621–15 629. https://doi.org/10.1609/aaai.v37i13.26852

[12] V. Rajasekar, M. Saračević, D. Karabašević, D. Stanujkić, E. Dobardžić, and S. Krishnamoorthi, "Efficient cancelable template generation based on signcryption and bio hash function," *Axioms*, vol. 11, no. 12, p. 684, 2022. https://doi.org/10.3390/axioms11120684

[13] S. K. Mondal, X. Wu, H. M. D. Kabir, H. N. Dai, K. Ni, H. G. Yuan, and T. Wang, "Toward optimal load prediction and customizable autoscaling scheme for Kubernetes," *Mathematics*, vol. 11, no. 12, p. 2675, 2023. https://doi.org/10.3390/math11122675

[14] M. N. Tran, D. D. Vu, and Y. Kim, "A survey of autoscaling in Kubernetes," in *2022 Thirteenth International Conference on Ubiquitous and Future Networks (ICUFN), Barcelona, Spain*, 2022, pp. 263–265. https://doi.org/10.1109/ICUFN55119.2022.9829572

[15] D. D. Vu, M. N. Tran, and Y. Kim, "Predictive hybrid autoscaling for containerized applications," *IEEE Access*, vol. 10, pp. 109 768–109 778, 2022. https://doi.org/10.1109/ACCESS.2022.3214985

[16] A. Horn, H. M. Fard, and F. Wolf, "Multi-objective hybrid autoscaling of microservices in Kubernetes clusters," in *Lecture Notes in Computer Science*. Springer, Cham, 2022, pp. 233–250. https://doi.org/10.1007/978-3-031-12597-3_15

[17] S. Krishnamoorthi, P. Jayapaul, V. Rajasekar, R. K. Dhanaraj, and C. Iwendi, "A futuristic approach to generate random bit sequence using dynamic perturbed chaotic system," *Turk. J. Electr. Eng. Comput. Sci.*, vol. 30, no. 1, pp. 35–49, 2022. https://doi.org/10.3906/elk-2010-137

[18] J. Dogani, F. Khunjush, and M. Seydali, "K-agrued: A container autoscaling technique for cloud-based web applications in Kubernetes using attention-based GRU encoder-decoder," *J. Grid Comput.*, vol. 20, no. 4, p. 40, 2022. https://doi.org/10.1007/s10723-022-09634-x

[19] D. Spatharakis, I. Dimolitsas, E. Vlahakis, D. Dechouniotis, N. Athanasopoulos, and S. Papavassiliou, "Distributed resource autoscaling in Kubernetes edge clusters," in *2022 18th International Conference on*

Network and Service Management (CNSM), Thessaloniki, Greece, 2022, pp. 163–169. https://doi.org/10.23919/CNSM55787.2022.9965056

[20] J. E. Joyce and S. Sebastian, "Reinforcement learning based autoscaling for Kafka-centric microservices in Kubernetes," in *2022 IEEE 4th PhD Colloquium on Emerging Domain Innovation and Technology for Society (PhD EDITS)*, Bangalore, India, 2022, pp. 1–2. https://doi.org/10.1109/PhDEDITS56681.2022.9955300

[21] I. Sfiligoi, T. DeFanti, and F. Würthwein, "Auto-scaling HTCondor pools using Kubernetes compute resources," in *Practice and Experience in Advanced Research Computing*, Boston MA, USA, 2022, p. 57. https://doi.org/10.1145/3491418.3535123

[22] C. Carrión, "Kubernetes scheduling: Taxonomy, ongoing issues and challenges," *ACM Comput. Surv.*, vol. 55, no. 7, pp. 1–37, 2022. https://doi.org/10.1145/3539606

[23] A. A. Pramesti and A. I. Kistijantoro, "Autoscaling based on response time prediction for microservice application in Kubernetes," in *2022 9th International Conference on Advanced Informatics: Concepts, Theory and Applications (ICAICTA), Tokoname, Japan*, 2022, pp. 1–6. https://doi.org/10.1109/ICAICTA56449.2022.9932943

Appendix

```
apiVersion: {{ .Values.apiversion }}
kind: Deployment
metadata:
  name: {{ .Values.app_name }}-dep
  labels:
{{ toYaml .Values.labels | indent 4 }}
spec:
  replicas: {{ .Values.replicaCount }}
  selector:
    matchLabels:
{{ toYaml .Values.labels | indent 6 }}
  template:
    metadata:
      name: {{ .Values.app_name }}-app
      labels:
{{ toYaml .Values.labels | indent 8 }}
    spec:
      containers:
      - name: {{ .Values.app_name }}-container
        image: {{ .Values.imageTag }}
        resources:
          requests:
              cpu: "{{ .Values.request_cpu }}"
              memory: {{ .Values.request_memory }}
          limits:
              cpu: "{{ .Values.limit_cpu }}"
              memory: {{ .Values.limit_memory }}
        ports:
          - containerPort: {{ .Values.port }}
```

A1. Deployment.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: {{ .Values.app_name }}-svc
spec:
  type: {{ .Values.svc_type }}
  ports:
    - port: {{ .Values.port }}
      targetPort: {{ .Values.port }}
  selector:
{{ toYaml .Values.labels | indent 4 }}
```

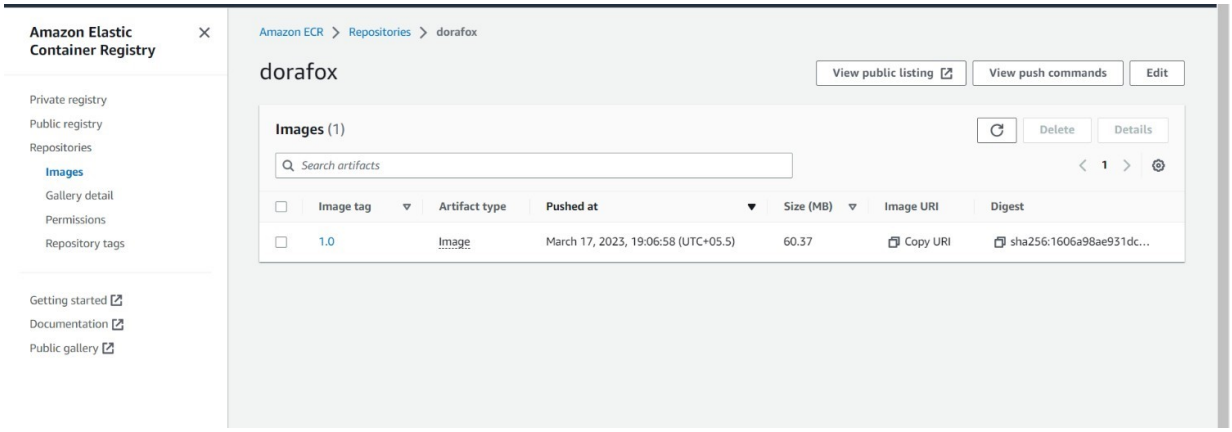
A2. Service.yaml

```
apiVersion: {{ .Values.hpa.apiVersion }}
kind: {{ .Values.hpa.kind }}
metadata:
  name: {{ .Values.app_name }}-hpa
  annotations:
    autoscaling.alpha.kubernetes.io/behavior: '{"ScaleUp":
    {"StabilizationWindowSeconds":{{ .Values.hpa.StabilizationWindowSeconds }},
    "SelectPolicy":"Max","Policies":[{"Type":"Pods","Value":{{ .Values.hpa.Policies.Value }},
    "PeriodSeconds":{{ .Values.hpa.Policies.PeriodSeconds }}},
    {"Type":"Percent","Value":{{ .Values.hpa.Policies.Value1 }},
    "PeriodSeconds":{{ .Values.hpa.Policies.PeriodSeconds1 }}}]},
    "ScaleDown":{"StabilizationWindowSeconds":{{ .Values.hpa.StabilizationWindowSeconds }},
    "SelectPolicy":"Max","Policies":[{"Type":"Pods","Value":{{ .Values.hpa.Policies.Value }},
    "PeriodSeconds":{{ .Values.hpa.Policies.PeriodSeconds }}}]}}'
spec:
  scaleTargetRef:
    apiVersion: {{ .Values.apiversion }}
    kind: Deployment
    name: {{ .Values.app_name }}-dep
  minReplicas: {{ .Values.hpa.minReplicas }}
  maxReplicas: {{ .Values.hpa.maxReplicas }}
  targetCPUUtilizationPercentage: {{ .Values.hpa.targetCPUUtilizationPercentage }}
```

A3. Hpa.yaml

```
apiversion: apps/v1
app_name: nodejs-
applicationlabels:
  app: nodejs-backend-app
imageTag:
public.ecr.aws/e5q6g0d1/dorafox:1.0port: 80
secretFile: secret-
nodeappreplicaCount: 1
svc_type:
LoadBalancerlimit_cpu:
1limit_memory:
1Girequest_cpu:
1request_memory:
1Ginamespace: node-
apphpa:
  apiVersion:
  autoscaling/v1kind:Horizontal
  PodAutoscaler
  StabilizationWindowSeconds:10
  minReplicas: 1
  maxReplicas: 3
  targetCPUUtilizationPercentage:25P
  olicies:
    Value:1
    PeriodSeconds: 40
    Value1:40
    PeriodSeconds1: 40
```

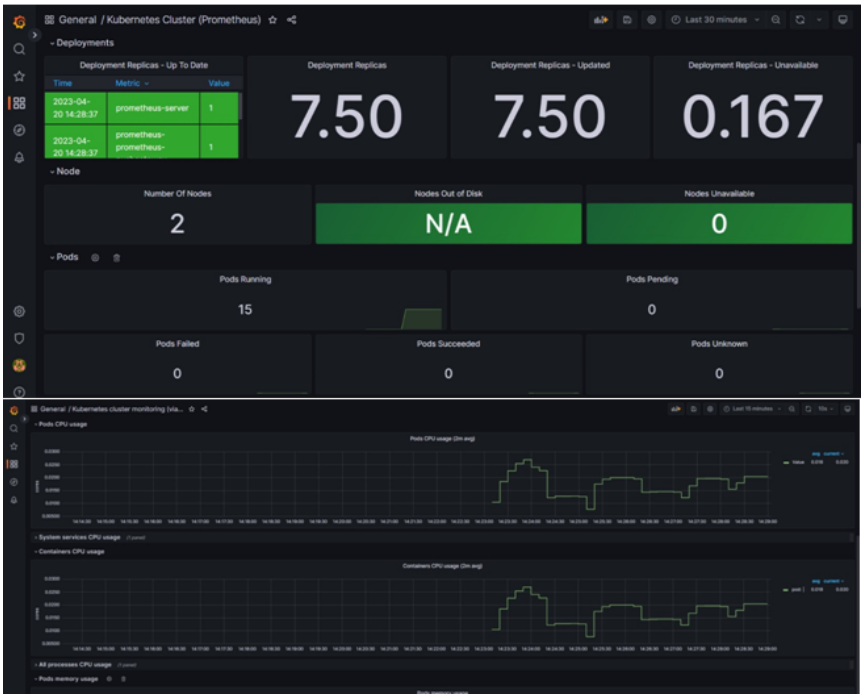
A4. Value.yaml



A5. Docker image in ECR

```
root@SAG-5SXXZY2:~# kubectl get all -n node-app
NAME                                      READY   STATUS    RESTARTS   AGE
pod/nodejs-application-dep-558c48556-75pnh   1/1     Running   0          117m

NAME                                TYPE           CLUSTER-IP     EXTERNAL-IP                                                               PORT(S)        AGE
service/nodejs-application-svc   LoadBalancer   10.100.83.31   aef94ed75718f44b5999b642f25dcfb2-814444297.us-east-2.elb.amazonaws.com   80:31782/TCP   117m

NAME                                         READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/nodejs-application-dep   1/1     1            1           117m

NAME                                            DESIRED   CURRENT   READY   AGE
replicaset.apps/nodejs-application-dep-558c48556   1         1         1       117m

NAME                                                                    REFERENCE                                 TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
horizontalpodautoscaler.autoscaling/nodejs-application-hpa   Deployment/nodejs-application-dep   8%/25%    1         3         1          117m
root@SAG-5SXXZY2:~# kubectl get all -n node-app
NAME                                      READY   STATUS    RESTARTS   AGE
pod/nodejs-application-dep-558c48556-2rfb7   1/1     Running   0          17s
pod/nodejs-application-dep-558c48556-75pnh   1/1     Running   0          117m

NAME                                TYPE           CLUSTER-IP     EXTERNAL-IP                                                               PORT(S)        AGE
service/nodejs-application-svc   LoadBalancer   10.100.83.31   aef94ed75718f44b5999b642f25dcfb2-814444297.us-east-2.elb.amazonaws.com   80:31782/TCP   117m

NAME                                         READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/nodejs-application-dep   2/2     2            2           117m

NAME                                            DESIRED   CURRENT   READY   AGE
replicaset.apps/nodejs-application-dep-558c48556   2         2         2       117m

NAME                                                                    REFERENCE                                 TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
horizontalpodautoscaler.autoscaling/nodejs-application-hpa   Deployment/nodejs-application-dep   97%/25%   1         3         2          117m
root@SAG-5SXXZY2:~# kubectl get all -n node-app
NAME                                      READY   STATUS    RESTARTS   AGE
pod/nodejs-application-dep-558c48556-2rfb7   1/1     Running   0          42s
pod/nodejs-application-dep-558c48556-75pnh   1/1     Running   0          118m

NAME                                TYPE           CLUSTER-IP     EXTERNAL-IP                                                               PORT(S)        AGE
service/nodejs-application-svc   LoadBalancer   10.100.83.31   aef94ed75718f44b5999b642f25dcfb2-814444297.us-east-2.elb.amazonaws.com   80:31782/TCP   118m

NAME                                         READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/nodejs-application-dep   2/3     3            2           118m

NAME                                            DESIRED   CURRENT   READY   AGE
replicaset.apps/nodejs-application-dep-558c48556   3         3         2       118m
```

A6. Scaling of pods in HPA



A7. CPU utilization in Grafana