SIRG Scientific Innovation Research Group

Journal of Smart Algorithms and Applications (JSAA)

ISSN: XXXX-XXXX

Journal Homepage:
https://pub.scientificirg.com/index.php/JSAA/en

Cite this: JSAA, xxxx (xx), xxx

Adaptive Pod Autoscaling for Enhanced Quality of Service in On-Premise Kubernetes Environments

Vani Rajasekar[1], *Malliga S*[2], Gokula Krishnan K[3], Sathya K[4], Vandana Sharma*[5]

[1,2,3] Department of CSE, Kongu Engineering College, PerunduraiErode, India, (vanikecit@gmail.com, mallisenthil@kongu.ac.in, krishnansai71@gmail.com *)*

[4] Department of CT/UG, Kongu Engineering College, Perundurai Erode, India, (pearlhoods@gmail.com *)*

*[*5] Amity Institute of Information Technology, Amity University, Noida, U.P, (* vandana.juyal@gmail.com*)*

**Corresponding Author: e-mail: (* vandana.juyal@gmail.com *)*

Abstract - Micro-services and cloud systems are developing into effective business tools nowadays. These container-based designs have made it possible to construct sophisticated SaaS applications efficiently. Managing and creating microservices with a wide range of different capabilities is a difficult task, from data processing and data warehousing to computing, prescriptive, and predictive analytics. Data centers made up of enormous, heterogeneous virtualized systems that are constantly expanding and diversifying over time are the foundation for computing service providers. Additionally, these technologies must be integrated with existing designs while adhering to Quality of Service (QoS) requirements. The primary objective of the proposed work is to provide an on-premises architecture based on Kubernetes and Docker.

Received: *15 September 2025*
Revised: *20 October 2025*
Accepted: *18 November 2025*
Available online: *11 December 2025*

Keywords:
-Kubernetes
-On-premises
-Quality of Service
-Docker
-Container

Introduction

Cloud computing has become incredibly popular in recent years. Companies are implementing on-premises infrastructure along with public clouds in greater numbers. Online services are used for almost all consumer services, including banking, education, and health care [1]. These services are crucial, continuous integration models that are set up in the cloud. Among these services, nowadays, microservice architecture is gaining more attention as a desired strategy for dealing with the constantly expanding monolithic software. The goal behind microservices is to break down large software systems into a collection of smaller services with limited business scopes that can communicate with one another easily. Thus, a small group of application developers is organized around a single

DOI: https://doi.org/XXXX.XXXX

microservice, enabling rapid continuous integration and delivery. Auto-scaling of crucial components is made simple and natural by the autonomously deployable microservices feature, ensuring Quality-of-Service (QoS). Since the introduction of Cloud Native, the majority of microservice implementers have chosen to use containers to launch their applications in cloud environments [2]. Compared to typical virtual machines, containers guarantee more consistent, segregated, and lightweight development environments. One of the major advantages of a cloud environment is resource provision that offers dynamic scalability and elasticity.

The migration of several legacy programs to cloud infrastructures that only use and operate on a predefined static set of resources. An infrastructure built on containers can be utilized to meet these requirements. This infrastructure aids in both deploying it to the cloud and attaining all the above-mentioned benefits of cloud computing, as well as maintaining logic created by various businesses and integrating and communicating with other services. Applications have changed from large decoupled monolithic microservices that encompass the entire system to tiny decoupled microservices for doing specific tasks. However, when compared to conventional cloud services, microservices on the cloud have different challenges, such as communication growth and QoS [3]. Cloud services and applications are strongly constrained by quality of service (QoS) requirements regarding metrics like efficiency, availability, reliability, and power awareness, regardless of their underlying architecture. Currently, a service-level agreement (SLA) governs QoS, and it is essential (SLA). The specific quantifiable elements of the SLA, such as availability, bandwidth, frequency, reaction time, or quality, are known as the service level objectives (SLOs). Qualities of service and customer satisfaction are closely related. User satisfaction declines as a result of an SLO violation that lowers QoS. As a result, service providers must make every effort to make these contracts to protect user privacy and to maximize revenue [4]. The major difficulty a service provider faces is choosing the right balance between profits and client satisfaction. Managing the QoS constraint is the key to success for challenges in SLOs and compliance in QoS.

The microservice idea promotes a variety of tools and facilitators for quickly developing systems, such as Kubernetes for container orchestration and Spring Boot for Java development. The majority of microservice implementers have used a container approach and deployed their solutions to cloud environments. However, containers by themselves are unable to scale or replace themselves in the event of a failover. This is where Kubernetes comes to the rescue as a platform for container orchestration. Kubernetes has been widely adopted and relied upon in production environments as a graduate project under the Cloud Native Computing Foundation (CNCF) [5]. Automated scaling, self-healing, network configuration, and storage orchestration are a few of its advanced features. One characteristic that many businesses find appealing about Kubernetes is that it allocates resources on behalf of the owners of microservice applications, enabling dynamic service auto-scaling. The Horizontal Pod Auto-scaler is the default auto-scaler in Kubernetes (HPA). However, different corporate operating models have varying expectations for service auto-scaling performance, as per service-level objectives (SLO) [6]. One crucial SLO is the response time, which measures how long it takes for an end user to complete a service request. The issue arises when heavy traffic starts to flow.

Related Works

An open-source technology called Kubernetes masks the difficulty of managing the availability of containerized microservices while orchestrating their deployment. Kubernetes enables the resilient execution of stateless microservices. The same does not apply to stateful microservices, though. The ephemeral state is a characteristic of containers, and stateful microservices add another layer of complexity to orchestration beyond what the original Kubernetes controllers were intended to handle. The Internet of Things (IoT) will become quicker, lighter, and more dependable because of the convergence of edge and cloud computing. The fields of IoT and cloud-edge computing are separate and have developed independently over time. For each service on the Edge side, Kubernetes Minion (Nodes) is built separately in the Docker container service by the authors using container-based virtualization [7]. The shortcomings of the existing Kubernetes support for stateful microservices are examined by the authors [8]. With the management of secondary labels, we suggest a method for enhancing Kubernetes with a State Controller that enables state duplication and automated service redirection to the healthy entities. To test our solution and compare the various designs from the standpoint of availability, we ran tests using both Kubernetes' default configuration and its most responsive one. Zero Trust is demonstrated by Silvia et al [9] as a different approach to internet security. It examines the study and earlier work that have been done on the application of zero trust. It talks about the possibility of Zero Trust for future network security. The design, which responds to many forms of threats, is implemented using containers. It concentrates on security at each layer of the OSI model as well as the benefits and drawbacks of zero-trust architecture.

Pakkonen et al [10] used machine learning-based models to anticipate energy usage in the private cloud domain and transferred the models to edge nodes for Kubernetes-based prediction. Predictors at the edge nodes can also be automatically updated without disrupting service. According to performance data with sensor-based devices (Raspberry Pi 4 and Jetson Nano), a suitable prediction latency (7-9 s) can be attained in the context of the research. Serving numerous clients with a variety of services is difficult for traditional cloud computing. Traditional cloud computing will experience a reduction in workload by spreading the services among multiple-edge servers. One of the platforms utilized for cloud management is Kubernetes. Kubernetes aids application deployment and scaling. Nowadays, many groups create a lightweight version of Kubernetes that is appropriate for edge devices like the Raspberry Pi [11]. In the Internet of Things (IoT) application, all data is kept in the cloud,

DOI: https://doi.org/XXXX.XXXX

which results in a distance between the cloud and the device or client-side that might cause network delays or sluggish reaction times. How to improve client response times in cloud computing and IoT environments is a difficult problem. We offer a complete set of Edge Computing architecture [12]. Three levels exist: the cloud side, the edge side, and the device side. By examining the usefulness of the Kubernetes container orchestration tool in the fog computing concept, Kayal et al [13] close the research gap. The study also identifies shortcomings in the Kubernetes strategy as it is now implemented and offers suggestions for future research to address the requirements of the fog environment. Finally, we offer experiments that show how deploying and overseeing containerized IoT apps in a fog computing environment is both feasible and useful in the real world.

Proposed Methodology

A. Preliminaries:

Microservices have been a popular term among software architects in recent years. Researchers have struggled to create effective software while balancing quality and velocity as software becomes vaster and more complicated. Software architecture has evolved in the industry from an N-tier monolith through service-oriented architecture (SOA) to microservices. A single deployment could include several modules from several business areas. Each module is included in a single, sizable binary file. Monolithic design has limitations in maintenance and scalability despite being simple to start. The first difficulty in the process of designing a microservice application is correcting the service boundary. Microservices should be independent of other microservices and be made up of closely linked features that are aligned with the limits of business capabilities [14-15]. Each microservice offers domain-specific CRUD operations as an autonomous, atomic unit. According to Domain-Driven Design (DDD), each module should have a unique set of delimited contexts that define its scope of application. Delegated data services, data lakes, distributed transactions, event sourcing, and CQRS are just a few of the data modeling paradigms that have emerged to address the problem (Command Query Responsibility Segregation). Coherent data changes that behave appropriately and consistently across chains of microservices are ensured by distributed transaction models. Using the Sgags design is one method of doing distributed transactions. To allow distributed transactions in Sagas, every microservice needs to implement the two operations commit and rollback.

B. Inter-service communication:

Microservices communicate with one another using a quick, asynchronous method, which is not aware of the presence of other microservices by design. Inter-service communication, such as RESTful APIs, gRPC, or GraphQL, honors contract-based interfaces as a result. The format and exchange of message objects must be properly and fully defined as part of the "contract." The event-driven asynchronous paradigm defines a true microservice architecture and communication model by drastically reducing coordination costs and latency, as shown in **Fig. 1**.

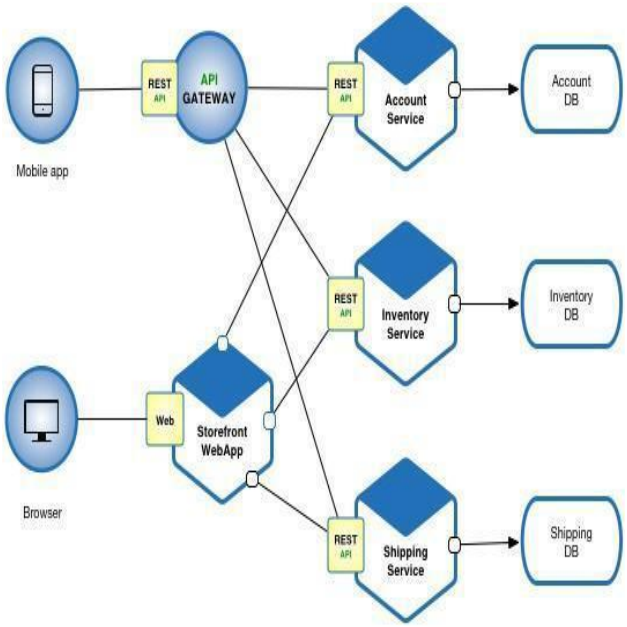

Fig.1: Microservice Architecture

C. Container virtualization and container runtime:

Modern containers did not become popular until Docker was introduced in 2013, even though container core technology first appeared in the early Linux kernel. The container is a lightweight sandbox that achieves process-level isolation in comparison to virtual machines. Containers run on the host operating system and do not require a hypervisor, unlike virtual machines (VMs), which considerably reduce overhead and improve portability. Linux namespaces, control groups (c-group), and union file systems are the fundamental technologies that enable and support modern containers [16-18]. Linux namespaces divide up the kernel's resources so that container processes are completely independent of one another, safe, and secure. Container processes believe they have exclusive ownership of the kernel and compute resources because of the namespace. The tool that offers a segregated environment for the execution of code and is based on the technologies is referred to as a container runtime. The most well-known and popular OCI-compliant container engine is Docker. The Open Containers Initiative (OCI), sometimes known as the container industry's governing body, works to standardize the image format, runtime file system bundle, container lifecycle, and image distribution APIs. The required components listed below are hosted by a group of master nodes that make up the control plane. They work together to protect a highly available cluster. Other than the elements listed below, other add-ons, including a DNS server for registering Kubernetes services and Prometheus for monitoring, are suggested

DOI: https://doi.org/XXXX.XXXX

by official Kubernetes publications for use in production environments, as shown in **Fig. 2**.

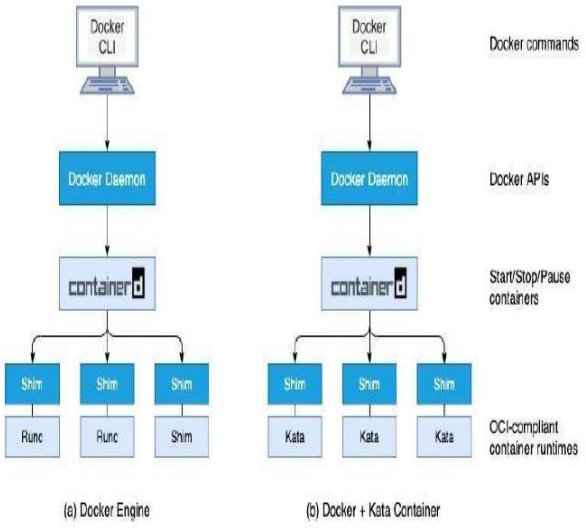

Fig. 2: Docker Container

D. Kubernetes

An open-source orchestrator for executing scalable production workloads, Kubernetes was created at Google under the moniker Borg. Google publicized the Borg system in 2014 and changed its name to Kubernetes. As the container market has expanded, Kubernetes has received more attention, making it the de facto container management option in comparison to other rivals like Docker Swarm and Mesos. To grow, Kubernetes manages the booting, updating, and coordination of container workloads. It includes a lot of characteristics that facilitate quick and safe product delivery [19]. Each container workload (referred to as a "Pod" in the Kubernetes language) is given its cluster by the container orchestration software Kubernetes. Kubernetes separates cluster nodes into two parts: master nodes and worker nodes. The master nodes represent the control plane whose job is to make global.

E. Kubernetes resource type

The required components listed below are hosted by a group of master nodes that make up the control plane. They work together to protect a highly available cluster. Other than the elements listed below, other add-ons, including a DNS server for registering Kubernetes services and Prometheus for monitoring, are suggested by official Kubernetes publications for use in production environments. The RESTful APIs for the Kubernetes control plane are exposed by the kube-apiserver facade. By default, the host port 6443 of the Kubernetes API server is open for CRUD operations against Kubernetes objects. The object state recorded in the cluster database is updated as well as the authentication and permission processes are handled, as shown in **Fig. 3**.

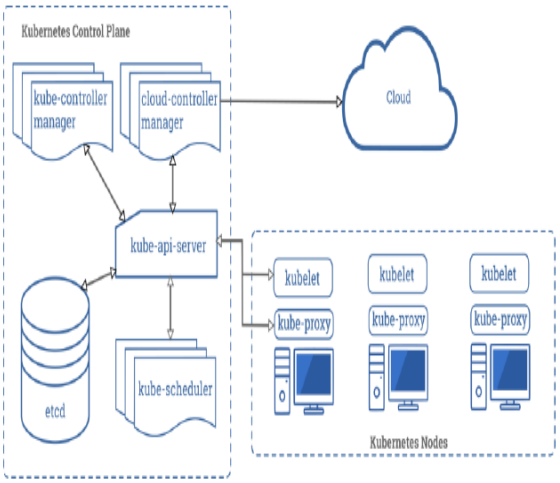

Fig. 3. Kubernetes cluster architecture

F. Pod:

Containerized workloads are run by Kubernetes in a single Pod unit. The fundamental unit of labor in the Kubernetes universe is the pod. A Pod has at least one application container and a hidden pause container that serves as a stand-in. While users define workloads in application containers, Kubernetes is in charge of managing the pause container, which is used to create kernel namespaces before others. Pods are designed to be transient [20]. High-level resources, such as Deployment or Stateful Set, oversee managing their lifespans.

G. Deployment

The template for Pods is stored in a deployment. The deployment controller controls and creates the Pods specified in the template field while keeping an eye on the Deployment API objects. It is feasible to roll out and roll back workloads predictably thanks to declarative management's feature. Kubernetes cluster administrators typically communicate with Deployment resources rather than the Pod directly.

Results and discussion

The suggested study analyses how various fictitious circumstances and alternative solution options can affect performance and lessen the cold-start effect. It also assesses auto-scaling performance when a cold start. To gain insight into the overall picture of the auto-scaling performance, the authors first ran two load tests on the Acme Air application. Second, fictitious elements are examined to determine their impact on the performance of auto-scaling using micro benchmarking, and recommendations are made for Kubernetes operators to follow. Before analyzing fictitious factors and our suggested auto-scaler, it is essential to understand how auto-scaling works with HPA, as shown in **Tab. 1**.

DOI: https://doi.org/XXXX.XXXX

Tab 1: Kubernetes Setup

Node Role	Instance Type	CPU/MEM Setting
1 Master Node	t3.medium	2 vCPU, 4 GB
3 Worker Nodes	t3.xlarge	4 vCPU, 16 GB

The preliminary experiments in this part, therefore, look at two scenarios: the auto-scaling of a single microservice and of numerous microservices with upstream-downstream relationships. On AWS, we built up a Kubernetes cluster with four nodes for the two tests, with one node designated as the master. Kubernetes version 1.20.6 is included in the installation manifest. Acme Air application, Cilium v1.9.4 for the CNI solution, and Docker v19.03.15 for the container runtime are installed. Kops v1.20.1, a production-grade Kubernetes installer, is used to set up the cluster. Prometheus for monitoring and the EFK stack for logging are optional components. We first issue low throughput for five minutes, then add it up to a high level in an instant to see how HPA and Kubernetes will react to burst traffic. In our single-master setup, as all components must interact with the kube-apiserver to update the cluster database, etc., we can capture the timestamp by enabling Kubernetes auditing logs.

A. Experiment 1: Single Microservice Scaling

In the first experiment, we examined auto-scaling on a single microservice, Flight Service, by continuously invoking the get reward mile API through JMeter. In the first part of the step-up load test, we instruct JMeter to emit requests at the rate of 200 qps and let the application fly with one pod for 5 minutes, then load the traffic up to 1200 qps. Since we configure HPA to keep every pod under 200 qps on average, HPA needs to trigger extra pod creation to balance traffic demand. We divide the auto-scaling process into four stages. By collecting and analyzing auditing logs, we determine Stage 1 of scaling decision making, starting from the traffic peak occurring (load test shift), terminates when the kube-apiserver successfully audits pod creation requests. Stage 2, pod-to-node binding, takes its turn and ends with kubelet, on the selected worker node, noticing the scheduling decision via a GET pod request. Stage 3, container initialization, competes when the container Started event occurs. Finally, Stage 4 of application startup covers the transition of pod readiness from false to true via a patch request from kubelet, as shown in **Tab. 2** and **Fig. 4**.

Tab 2: Auto-scaling process with pods

Pod Name Suffix	Stage 1	Stage 2	Stage 3	Stage 4
xpz2f	13.799	0.038	1.116	16.962
chfcn	44.553	0.016	1.139	23.654
2pm9g	44.566	0.021	1.252	26.236
gc7nz	60.281	0.029	1.126	25.513
285g7	60.302	0.029	1.139	24.154
AVG	44.7002	0.0266	1.1544	23.3038

```
ubuntu@ip-172-31-29-48:~$ kubectl get po
NAME                                    READY   STATUS    RESTARTS   AGE
acmeair-flight-db-69df8f87b4-tg7h4      1/1     Running   0          111m
acmeair-flightservice-76d48bfb54-285g7  1/1     Running   0          66s
acmeair-flightservice-76d48bfb54-2pm9g  1/1     Running   0          82s
acmeair-flightservice-76d48bfb54-chfcn  1/1     Running   1          82s
acmeair-flightservice-76d48bfb54-gc7nz  1/1     Running   0          66s
acmeair-flightservice-76d48bfb54-hcff4  1/1     Running   0          25m
acmeair-flightservice-76d48bfb54-xpz2f  1/1     Running   0          112s
```

Fig. 4. Pod restart during scale-out

B. Experiment 2: Multiple Microservices Scaling

We next reproduce the experiment on multiple service scaling since the upstream-downstream pattern is more typical in microservice data flow. In the experiment, we called the booking flight API to reserve a flight for one customer, which will query flight reward mile information, and update the customer's total reward miles. Therefore, the booking request will invoke the Booking Service, Flight Service, and Customer Service in sequence. Each service exposes QPS at the Prometheus metric endpoint. We deploy three HPA objects for each service to perform multiple service auto-scaling. We set the scaling threshold as 100 qps and ramp up traffic loads from 100 qps for the first 5 minutes to 400 qps for the rest of the time, so that for each service, three additional pods need to be brought up.

C. Container Runtime

The speed of container initialization in Stage 3, or container initialization, is the next aspect we examine. We contrast the three container runtimes that are currently available: CRI-O, container, and Docker (v19.03.15). (v1.20.3). They differ in how the Kubernetes Container Runtime Interface is implemented. (CRI). Containerd uses the same dockershim capability as a CRI plugin in its code; however, Docker needs a separate dockershim process to bind the Docker daemon and Kubernetes. Since CRI-O incorporates CRI by design, there is no additional overhead when interacting with the kubernet. We gauge the startup time of the containers under various container runtimes to compare their performance. The startup time is the period between the Kubernetes receiving a newly formed pod assigned to it and the broadcasting of the container Started event. (covering Stage 3). The details and event timestamps are reported in the Kubernetes logs as shown in **Fig. 5**.

DOI: https://doi.org/XXXX.XXXX

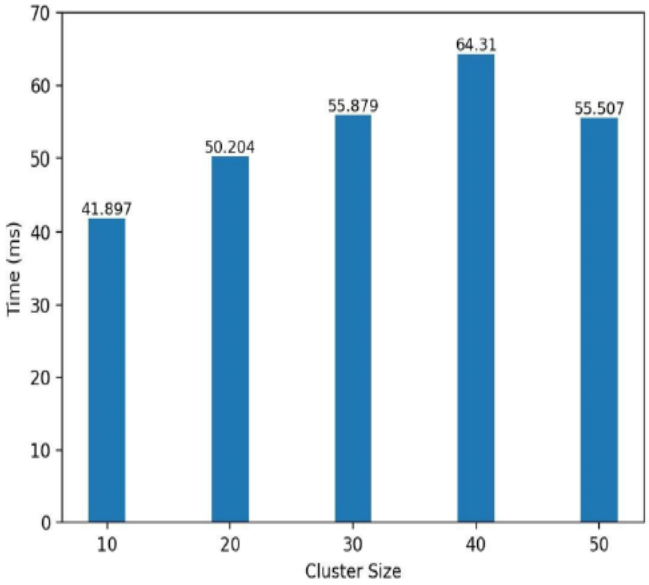

Fig. 5: Pod Scheduling with different runtimes

D. Network Setup

The third hypothetical factor we examine is the choice of CNI solutions. CNI plugins are responsible for attaching the network to containers during container initialization in Stage 3. In the previous experiments, we stuck to Cilium for networking. In this section, we expand our evaluation and compare five common solutions as listed as shown in **Tab.3 and Fig. 6**.

Tab 3: CNI Solution setup

CNI Plugins	Network Model		Tunneling Protocol	Version
	intra-host	inter-host		
Cilium	Layer 3	Overlay	VXLAN	v1.9.4
Flannel	Layer 2	Overlay	VXLAN	v0.13.0
Calico	Layer 3	Overlay	IP in IP	v3.18.3
Weave Net	Layer 2	Overlay	VXLAN	v2.8.1
Kube-router	Layer 2	Overlay	IP in IP	v1.1.1

We considerably shorten the formation time of the last pods because there is no laggard pod, even if the creation time of the earliest pod depends on the time of metric retrieval by the monitoring system. The improvement reveals that our suggested CHPA can assist microservices in becoming ready to serve traffic quickly, as evidenced by the experiment's three services changed QPS.

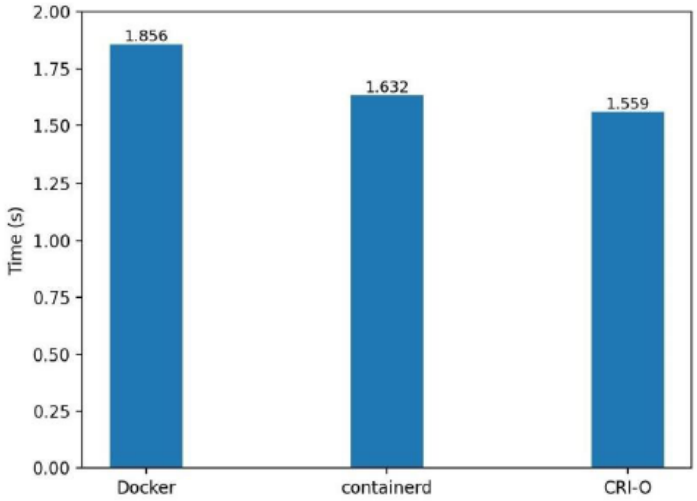

Fig. 6. Container Startup time per routine

Conclusions and Future Work

The era of cloud-native is being ushered in by Kubernetes. It introduces a few fresh concepts for designing and deploying microservice applications. The ability of Kubernetes to support horizontal auto-scaling on top of workloads in containers is one advantage of using it in production. Applications that require low latency demand agile scaling. When the system receives bursty requests, there may not be enough pod instances, which could lead to a cold-start auto-scaling process that slows response time and degrades user experience. We carefully analyze the mechanism and principle of the horizontal pod auto scaler to determine the causes of the cold start. Our research shows that there is room for advancement in the area of agile auto-scaling. As more businesses migrate their microservice applications to Kubernetes clusters. In the future, the proposed process can be enhanced by incorporating an efficient monitoring tool for auto-scaling.

References

[1] Nguyen, N., & Kim, T. (2020). Toward highly scalable load balancing in kubernetes clusters. IEEE Communications Magazine, 58(7), 78-83.

[2] Rossi, F., Cardellini, V., & Presti, F. L. (2020, August). Hierarchical scaling of microservices in kubernetes. In 2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS) (pp. 28-37). IEEE.

[3] Raj, P., Vanga, S., & Chaudhary, A. (2023). Kubernetes Architecture, Best Practices, and Patterns.

[4] Vayghan, L. A., Saied, M. A., Toeroe, M., & Khendek, F. (2021). A Kubernetes controller for managing the availability of elastic microservice based stateful applications. Journal of Systems and Software, 175, 110924.

[5] Siwach, G., Haridas, A., & Chinni, N. (2022, November). Evaluating operational readiness using chaos engineering simulations on Kubernetes architecture in Big

DOI: https://doi.org/XXXX.XXXX

Data. In 2022 International Conference on Smart Applications, Communications and Networking (SmartNets) (pp. 1-7). IEEE.
[6] Shah, J., & Dubaria, D. (2019, January). Building modern clouds: using docker, kubernetes & Google cloud platform. In 2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC) (pp. 0184-0189). IEEE.
[7] Kristiani, E., Yang, C. T., Huang, C. Y., Wang, Y. T., & Ko, P. C. (2021). The implementation of a cloud-edge computing architecture using OpenStack and Kubernetes for air quality monitoring application. Mobile Networks and Applications, 26, 1070-1092.
[8] L. Abdollahi Vayghan, M. A. Saied, M. Toeroe and F. Khendek, "Microservice Based Architecture: Towards High-Availability for Stateful Applications with Kubernetes," 2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS), Sofia, Bulgaria, 2019, pp. 176-185, doi: 10.1109/QRS.2019.00034.
[9] D'Silva, D., & Ambawade, D. D. (2021, April). Building a zero trust architecture using Kubernetes. In 2021 6th international conference for convergence in technology (i2ct) (pp. 1-8). IEEE.
[10] Pääkkönen, P., Pakkala, D., Kiljander, J., & Sarala, R. (2020). Architecture for enabling edge inference via model transfer from cloud domain in a kubernetes environment. Future Internet, 13(1), 5.
[11] Fathoni, H., Yang, C. T., Chang, C. H., & Huang, C. Y. (2019). Performance comparison of lightweight kubernetes in edge devices. In Pervasive Systems, Algorithms and Networks: 16th International Symposium, I-SPAN 2019, Naples, Italy, September 16-20, 2019, Proceedings 16 (pp. 304-309). Springer International Publishing.
[12] Kristiani, E., Yang, C. T., Wang, Y. T., & Huang, C. Y. (2019). Implementation of an edge computing architecture using openstack and kubernetes. In Information Science and Applications 2018: ICISA 2018 (pp. 675-685). Springer Singapore.
[13] P. Kayal, "Kubernetes in Fog Computing: Feasibility Demonstration, Limitations and Improvement Scope : Invited Paper," 2020 IEEE 6th World Forum on Internet of Things (WF-IoT), New Orleans, LA, USA, 2020, pp. 1-6, doi: 10.1109/WF-IoT48130.2020.9221340.
[14] Dhanaraj, R. K., Islam, S. H., & Rajasekar, V. (2022). A cryptographic paradigm to detect and mitigate blackhole attack in VANET environments. Wireless Networks, 28(7), 3127-3142.
[15] Rajasekar, V., Sathya, K., & Premalatha, J. (2018, December). Energy efficient cluster formation in wireless sensor networks based on multi objective bat algorithm. In 2018 International Conference on Intelligent Computing and Communication for Smart World (I2C2SW) (pp. 116-120). IEEE.
[16] Janani, K., Anuhya, K., Manaswini, V. L., Likitha, V., Suneetha, B., & Vignesh, T. (2022). Analysis of CI/CD Application in Kubernetes Architecture. Mathematical Statistician and Engineering Applications, 71(4), 11091-11097.
[17] Fiori, S., Abeni, L., & Cucinotta, T. (2022, April). RT-kubernetes: containerized real-time cloud computing. In Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing (pp. 36-39).
[18] Rajasekar, V., Venu, K., Jena, S. R., Varthini, R. J., & Ishwarya, S. (2021). Detection of cotton plant diseases using deep transfer learning. Journal of Mobile Multimedia, 18(2), 307-324.
[19] Barriga, J. J., Sulca, J., León, J., Ulloa, A., Portero, D., García, J., & Yoo, S. G. (2020). A smart parking solution architecture based on LoRaWAN and Kubernetes. Applied Sciences, 10(13), 4674.
[20] Todorov, M. H. (2021, October). Design and deployment of Kubernetes cluster on Raspberry pi OS. In 2021 29th National Conference with International Participation (TELECOM) (pp. 104-107). IEEE.

DOI: https://doi.org/XXXX.XXXX