J. Sci. Trans. Environ. Technov.2024, 18 (1) :1 - 9

Software Development Improvement in CC using Containerization as a Service (CaaS)

Mr. A. Karthikeyan [1*], Dr. K. Loheswaran[2], Ms. M. Vani[3] and Ms. S. Geetha[4]

[1] Assistant Professor, Department of Computer Science and Engineering, Mahendra Engineering College (Autonomous), Namakkal. E mail: karthikeyana.be@gmail.com

[2,3,4] Assistant Professors, Department of Computer Science and Engineering, Mahendra Engineering College (Autonomous), Namakkal.

Article History
Received: **10.07.2024**
Revised and Accepted: **20.08.2024**
Published: **25.09.2024**

https://doi.org/10.56343/STET.116.018.001.001
www.stetjournals.com

ABSTRACT

The evolution of Cloud Computing (CC) in Software Development (SD) has been significantly influenced by containerization, which provides application developers with an efficient method for distributing portable web applications. Docker, a widely adopted containerization platform, generates lightweight containers that simplify application delivery by reducing complexity in software component integration and configuration. Similarly, Kubernetes offers containerized application automation with capabilities for scaling, deployment, and resource management across clusters. This study focuses on enhancing software development in CC through an optimized Kubernetes approach, comparing it with Docker and conventional Kubernetes in terms of resource management and advanced integrations. Modern DevOps-based SD increasingly leverages containers and CC to achieve stability, portability, security, and scalability. Container as a Service (CaaS) integrates containerization with cloud infrastructure, offering flexible, scalable solutions. To date, no prior study has examined the technical configuration and deployment of CaaS using openSUSE Kubic in relation to the Linux kernel and operating system, tested via Pods from both service and replication perspectives. Experimental results demonstrate that openSUSE Kubic is highly streamlined and effective for CaaS development and deployment, providing software developers with a highly extensible and efficient environment for SD.

Keywords: Cloud computing, Container as a Service, containerization, DevOps, openSUSE Kubic, software development

Mr. A. Karthikeyan

Assistant Professor, Department of Computer Science and Engineering, Mahendra Engineering College (Autonomous), Namakkal.
E mail: karthikeyana.be@gmail.com

P-ISSN 0973-9157
E-ISSN 2393-9249

INTRODUCTION

Cloud Computing (CC) is a method of sharing system resources among multiple users through various virtualization technologies, which has gained prominence due to its pay-as-you-go business model (Dimitri, 2020). This model requires payment for the exact amount of resources or services consumed, thereby reducing service costs and enabling users to scale services in line with evolving business needs. To access any type of service, a user must first register with

a Cloud Service Provider (CSP) by submitting an online request (Azadi et al., 2022). The CSP is responsible for responding to client requests while effectively managing computational resources. Various scheduling strategies are employed to ensure optimal resource management, as resource allocation and scheduling significantly influence the performance of CC applications.

Effective task completion within specified deadlines is essential for maintaining high Quality of Service (QoS) and meeting Service Level Agreement (SLA) requirements. Load balancing plays a crucial role in enhancing CC performance (Kishor et al., 2021; Shahid et al., 2020). User workloads should be evenly distributed across available computing nodes, ensuring that no node becomes overloaded or underloaded (Kazeem Moses et al., 2021). Overloaded nodes may cause delays and missed deadlines, while underloaded nodes result in wasted computing resources and idle capacity. Therefore, the appropriate implementation of load balancing techniques can reduce response times and improve overall user satisfaction (Shafiq et al., 2021; Tong et al., 2020).
Load balancing in CC can be achieved through several methods, including efficient task scheduling algorithms, resource migration, optimal resource allocation, resource reservation, and service migration (Nawrocki et al., 2021; Yin et al., 2021). In addition to the traditional CC service models—Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS)—various CSPs have introduced a newer model known as Container as a Service (CaaS). In CC, a container is a form of operating system virtualization. A container image comprises the runtime environment, system tools, application code, configuration settings, and library packages in a lightweight, self-contained, executable format, providing all that is required to run the software. Containers incorporate core components of OS-level virtualization, offering isolated environments without the need for an additional management layer such as a hypervisor (Maenhaut et al., 2020).

Unlike virtual machines (VMs), containerization operates directly on the host OS infrastructure without a hypervisor. The container engine functions similarly to a hypervisor by creating and managing active container instances. As shown in Figure 1, multiple containers run on top of the container engine, sharing the same host OS while using only the necessary files—such as libraries and binaries—to execute application code. This makes containers lighter and more efficient than VMs (Bentaleb et al., 2022). In the CaaS model, infrastructure operates on the host OS, while the hypervisor—if used—resides on top of the host OS, managing virtual machines with their own guest operating systems. The container engine, however, operates directly on the user's OS, building and maintaining active container instances. Each container package pulls required files from the OS, combining hardware and software virtualization in a hybrid approach to CC.

Today, cloud customers often deploy applications to a single CSP. However, when extending these applications closer to users and edge devices, resource security across multiple CSP edges becomes important. This is more manageable when providers are integrated, enabling the customer to interact with one CSP interface while the provider manages workload distribution to other CSPs (Asad et al., 2020). Multitenancy ensures that each CSP can accommodate workloads from different providers, with a rule set governing resource allocation, isolated environments, user permissions, and resource sharing. These rules determine access privileges and their implications for tenants sharing similar assets.

This paper presents a framework demonstrating the use of openSUSE Kubic for enhancing software development in CC applications. Deploying CaaS to the edge cloud is made possible through freely licensed, open-source extensions to the Kubernetes container orchestration system. The motivation for developing an optimized Kubernetes environment stems from the widespread familiarity and industry-standard adoption of Kubernetes as a container orchestration technology.

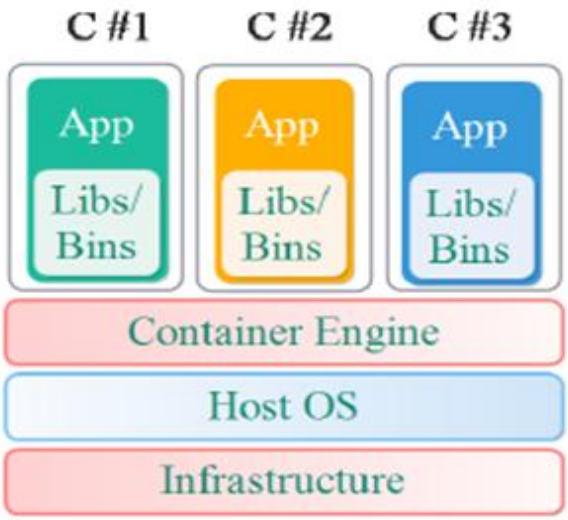

Figure 1 Container Architecture

LITERATURE REVIEW

Al Jawarneh et al. (2019) discussed that clients interested in deploying containerized services in Cloud Computing (CC) can utilise one of four widely available container orchestration systems—Apache Mesos, Docker Swarm, Rancher's Cattle, and Kubernetes. Their study highlights Kubernetes as having emerged as the de facto industry standard, with all major Cloud Service Providers (CSPs) offering Kubernetes-based CaaS to consumers (Table 1).

Table 1 Main CSP offered with kubernetes-based CaaS

CSP	Offered Kubernetes-based CaaS
Microsoft Azure	Azure Kubernetes Services
Amazon Web Services (AWS)	Elastic Kubernetes Services
Google Cloud Platform	Google Kubernetes Services
IBM Cloud	IBM cloud Kubernetes Services
Oracle Cloud	Oracle container engine for kubernetes
Alibaba Cloud	Alibaba cloud container service for kubernetes
Tencent Cloud	Tencent Kubernetes engine

Kumari et al. (2021) observed that, despite the growing adoption of containers, CC remains largely focused on supplying Virtual Machines (VMs), with cloud users typically billed according to the number of VMs allocated. Many applications, however, fail to fully utilise the allocated resources, leading to inefficiencies. They proposed a fine-grained cost model, making container-based deployment a cost-effective option and enabling serverless computing—a model in which cloud users supply only the code, while the CSP manages the execution environment for its entire lifecycle.

Maenhaut et al. (2020) explained that cloud users are often charged based on the computing resources consumed by their applications. Serverless computing can simplify cloud deployment by removing the need to set up and maintain multiple services, thereby reducing costs—especially for small-scale operations. Containers, due to their fast start-up times and low overhead, have the potential to play a significant role in the future of serverless computing. Singh et al. (2020) proposed a CaaS technique for processing applications via edge computing, deploying it through Docker. Their integration of container services with data transfer solutions improved load balancing performance and supported Internet of Things (IoT) applications requiring low latency and high data transfer rates. They further presented a lightweight, energy-efficient CaaS method for delivering workloads to low-latency IoT applications, showing comparative advantages over other CaaS approaches.

Saxena and Sharma (2021) emphasised that, in today's diverse technological landscape, containers enhance operational efficiency, version control, developer productivity, and environmental consistency. Their work evaluated Docker's performance in a CC environment using a range of applications and technologies. Similarly, Qiao et al. (2024) introduced EdgeOptimizer, a programmable containerised scheduler for time-critical tasks in Kubernetes-based edge-cloud clusters. Their work addressed the increasing need for efficient service orchestration and request distribution in edge

computing, noting that traditional CC architectures are often unable to meet the demands of latency-sensitive services. Experimental results showed that EdgeOptimizer, which uses an optimised D3QN method, outperforms competing algorithms such as modified DDPG and Kubernetes-native approaches in terms of request execution rates.

Oleghe (2021) examined challenges and strategies for container placement and migration in edge computing, noting that the proliferation of wireless devices and the demand for low-latency processing are expected to result in 75% of data being managed outside conventional data centres by 2025. The study highlighted container orchestration—particularly scheduling algorithms for allocating computational loads across heterogeneous edge nodes—as an NP-hard problem. Solutions explored include optimisation models, multidimensional selection approaches, and Markov Decision Processes. The research also emphasised the role of heuristic algorithms for achieving near-optimal solutions quickly and the benefits of decentralised scheduling to manage the growing complexity of edge environments.

In summary, the reviewed literature highlights a strong trend towards container-based solutions in CC, with Kubernetes widely adopted as the orchestration standard. Containers improve performance by isolating application dependencies and streamlining OS patch management. Docker, among other platforms, has enabled microservices deployment on cloud infrastructures, allowing applications and data to be packaged for rapid deployment and reuse (Journal, 2020).

RESEARCH METHODOLOGY

This research was conducted using both hardware and software resources to implement and evaluate the proposed approach. The hardware setup included a system with 16 GB RAM, an Intel Core i7 processor, a 500 GB hard disk, and an internet modem. The primary software platform utilised was openSUSE Kubic (MicroOS) with 64-bit Docker, Pods, and Kubernetes installed. The experimental phase took place between May 2020 and mid-December 2020, conducted entirely online due to the COVID-19 pandemic.

A **Systematic Literature Review (SLR)** methodology was adopted, drawing on references from published research articles, books, and reputable online sources related to Docker, Kubernetes, CaaS, containers, and other associated studies. The research emphasised operational functions within openSUSE Kubic, focusing on its ability to facilitate the development and deployment of CaaS through available modules. Each development step was carefully considered, with testing performed using Pods to assess service availability and replication processes. Observations were documented with screenshots, and all testing results were recorded, analysed, and discussed to address the study objectives.

The openSUSE Linux online repository was used to support the setup, deployment, and testing stages. This repository provides a digital storage space for applications and libraries, which can be easily installed on the operating system as required. The repository, maintained by SUSE and mirrored locally in Indonesia, was accessed during the implementation phase. Both execution and evaluation phases followed the same test scenario design and research flowchart for a cloud user working with openSUSE Kubic.

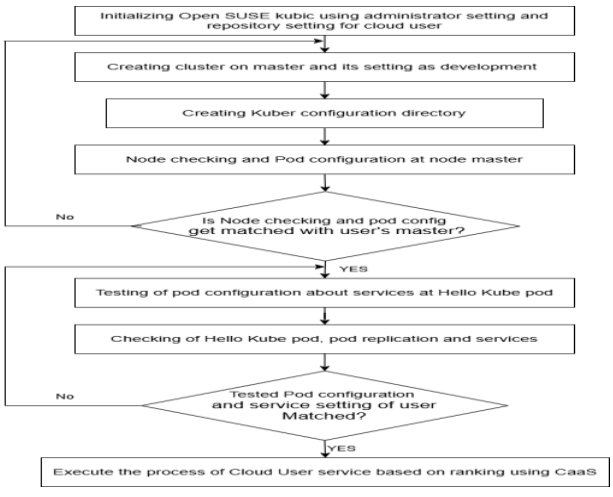

Figure 1. Proposed architecture of CaaS for providing improved CSP

The **test scenario** involved installing and configuring openSUSE Kubic to develop, deploy, and test CaaS using Pods. The PC used in testing also had Kubernetes and Docker with Pods installed. The research flow outlined each stage and method employed in the investigation. Figure 1 illustrates the complete process of CaaS operation in cloud services using containers.

Installation of openSUSE Kubic Repository and with Administration Setting

The first stage involved installing openSUSE Kubic on the system. This included loading the OpenSUSE MicroOS.iso file, following the on-screen setup instructions, specifying the system role in the configuration screen for the Kubeadm Node using the kubeadm init command, setting the root password, and configuring the online repositories. In this study, the system role was configured as a Kubeadm Node (Figure 2).

Figure 2 Installation of Kubic admin node

Following installation, the system was accessed via root login. The master cluster was established by executing the kubeadm init command. After the cluster was built and initialised, the kubeadm join command was used to connect nodes to the cluster via an authentication token (Figure 3).

Figure 3 Root login process of kube configuration

The node status was then verified using the kubectl get nodes command, which confirmed a single node, localhost, with master status. To remove restrictions on running Pods on the master node, the command kubectl taint nodes --all node.role.kubernetes.io/master-node/localhost untainted was executed. Figure 4 shows the verification process for node status and master node configuration.

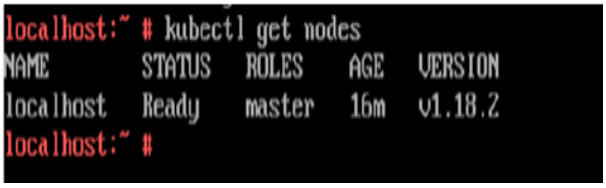

Figure 4 Verification of Master node in kube configuration

Once the CaaS setup with openSUSE Kubic was completed successfully, the **deployment stage** began. The hello-kubic.yaml file, located at /usr/share/k8s-yaml/hello-kubic, was used as a Pod for deployment. The deployment was executed using the command:

kubectl apply -f /usr/share/k8s-yaml/hello-kubic/hello-kubic.yaml

(Figure 5). After deployment, the Pod status was verified with the kubectl get deployment and kubectl get pods commands, showing that the hello-kubic Pod successfully replicated three times, with all replicas ready and running

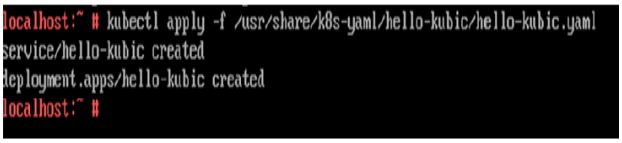

Figure 5 Testing of Hello-Kubic pod

Finally, the status of running services was checked using the kubectl get svc command (Figure 6). The results confirmed that the services were successfully developed and deployed, meeting the intended CaaS configuration and testing requirements.

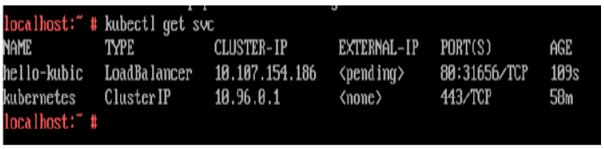

Figure 6 Outcome of user service Hello-kube pod

RESULT AND DISCUSSION

Containers function as a form of virtualisation in which resources and services operate in separate drivers, libraries, and binaries while sharing the host operating system (OS). This approach allows multiple containers to run on the same system using the OS kernel, with each container functioning as a distinct process. Containers include only the necessary libraries and dependencies required for execution, making them more efficient and lightweight compared to other virtualisation methods.

Container as a Service (CaaS) is a container-based service model within the CC infrastructure that consolidates all operational services. It accelerates software development (SD) by enhancing scalability, security, and efficiency, while also providing greater flexibility. In the proposed approach, both containerisation using openSUSE Kubic and Docker-based containerisation were implemented for comparison.

For performance evaluation, various **Quality of Service (QoS)** metrics were considered for CC application testing, including:

- Deployment speed
- Resource utilisation
- Scalability
- Fault tolerance
- Network efficiency

Table 2 presents a comparison of the implemented openSUSE Kubic Pods and Docker containerisation across these metrics

Table 2 Comparison of CC application metrics with various types of containerization

Metrics	**openSUSE Kubic**	**Dockers**
Deployment speed (Sec)	32	47
Resource utilization (CPU %)	83	69
Scalability (Max pod)	995	255
Network efficiency (ms)	20	30
Fault Tolerance (Sec)	3	15

The results indicate that **openSUSE Kubic** achieves a shorter deployment time (32 seconds) compared to Docker (47 seconds) and supports significantly higher scalability (995 Pods versus 255 Pods). Network efficiency is also improved, with a latency of 20 ms compared to Docker's 30 ms. Resource utilisation is higher for openSUSE Kubic (83%) compared to Docker (69%), indicating more efficient CPU usage. Additionally, fault tolerance—the time taken to recover from a failure—is markedly lower in openSUSE Kubic (3 seconds) than in Docker (15 seconds).

These findings are visually represented in **Figure 7**, which compares QoS metrics for the two containerisation approaches.

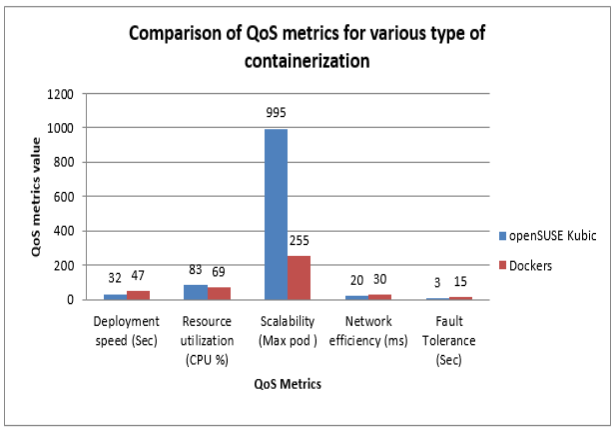

Figure 7 Comparison of QoS metrics for various type of containerization

From these results, it is evident that openSUSE Kubic offers advantages in scalability, network efficiency, and fault tolerance, making it particularly well-suited for small-scale applications and development environments that require rapid and efficient virtual application deployment. Furthermore, its compatibility with CI/CD pipelines enables faster development cycles, continuous integration, and seamless testing.

In contrast, Docker's relative simplicity and flexibility make it an attractive choice when deployment requirements are less complex or when continuous development must continue after deployment. While Docker remains widely adopted for its lightweight architecture and adaptability, the enhanced scalability and performance characteristics of openSUSE Kubic provide compelling benefits for cloud-native application development in CC environments.

CONCLUSION

The results of this study demonstrate that **openSUSE Kubic** provides a reliable and efficient platform for developing and deploying **Container as a Service (CaaS)** to manage multiple applications and services within a containerised environment. By incorporating Kubernetes and Pod support, openSUSE Kubic enables rapid deployment, scalability, load balancing, and self-recovery—features essential for modern cloud-based software systems.

Kubernetes, when properly configured, offers a robust solution for orchestrating distributed virtual nodes, making it highly effective for large-scale and geographically dispersed deployments. Although Kubernetes can be complex to implement—particularly in multi-cloud environments—its flexibility and operational efficiency make it a preferred choice for organisations seeking advanced container orchestration capabilities.

This study recommends that organisations evaluate both **Docker** and **Kubernetes** before selecting a deployment strategy, considering factors such as application complexity, required scalability, and operational objectives. Ultimately, containerisation delivers key benefits in portability, scalability, and optimised resource utilisation, positioning it as a cornerstone technology in current CC-based software development. Leveraging openSUSE Kubic, combined with the open-source Linux kernel, provides developers and software entrepreneurs with a streamlined, cost-effective, and powerful solution for deploying CaaS in diverse application environments.

REFERENCE

Al Jawarneh, I.M., Bellavista, P., Bosi, F., Foschini, L., Martuscelli, G., Montanari, R. & Palopoli, A. 2019. Container orchestration engines: A thorough functional and performance comparison. *Proceedings of the IEEE International Conference on Communications (ICC)*. https://doi.org/10.1109/ICC.2019.8762053

Asad, J., Robert, J., Heljanko, K. & Främling, K. 2020. IoTEF: A federated edge-cloud architecture for fault-tolerant IoT applications. *Journal of Grid Computing*, 18(1), pp.57–80. https://doi.org/10.1007/s10723-019-09498-8

Azadi, M., Emrouznejad, A., Ramezani, F. & Hussain, F.K. 2022. Efficiency measurement of cloud service providers using network data envelopment analysis. *IEEE Transactions on Cloud Computing*, 10(1), pp.348–355.

Bentaleb, O., Belloum, A.S.Z., Sebaa, A. & El-Maouhab, A. 2022. Containerization technologies: Taxonomies, applications and challenges. *The Journal of Supercomputing*, 78(1), pp.1144–1181. https://doi.org/10.1007/s11227-021-03914-1

Dai, Y., Xu, D., Maharjan, S., Qiao, G. & Zhang, Y. 2019. Artificial intelligence empowered edge computing and caching for internet of vehicles. *IEEE Wireless Communications*, 26(3), pp.12–18.

Dimitri, N. 2020. Pricing cloud IaaS computing services. *Journal of Cloud Computing*, 9, p.14.

Isam, M., Kumari, A., Sahoo, B., Behera, R.K., Misra, S. & Sharma, M.M. 2021. Evaluation of integrated frameworks for optimising QoS in serverless computing. In: *International Conference on Computational Science and Its Applications*. Cham: Springer, pp.277–288.

Journal, I.J.E.T.R.M. 2020. Containerized web application and deployment on cloud. *IJETRM Journal*.

Kazeem, M.A., Joseph, B.A., Roseline, O.O., Misra, S. & Abidemi, E.A. 2021. Applicability of MMRR load balancing algorithm in cloud computing. *International Journal of Computer Mathematics: Computer Systems Theory*, 6(1), pp.7–20.

Kishor, A., Niyogi, R., Chronopoulos, A. & Zomaya, A. 2021. Latency and energy-aware load balancing in cloud data centres: A bargaining game-based approach. *IEEE Transactions on Cloud Computing*.

Maenhaut, P.J., Volckaert, B., Ongenae, V. & De Turck, F. 2020. Resource management in a containerized cloud: Status and challenges. *Journal of Network and Systems Management*, 28, pp.197–246. https://doi.org/10.1007/s10922-019-09504-0

Nawrocki, P., Grzywacz, M. & Sniezynski, B. 2021. Adaptive resource planning for cloud-based services using machine learning. *Journal of Parallel and Distributed Computing*, 152, pp.88–97.

Oleghe, O. 2021. Container placement and migration in edge computing: Concept and scheduling models. *IEEE Access*, 9, pp.68028–68043.

Qiao, Y., Shen, S., Zhang, C., Wang, W., Qiu, T. & Wang, X. 2024. EdgeOptimizer: A programmable containerized scheduler of time-critical tasks in Kubernetes-based edge-cloud clusters. *Future Generation Computer Systems*, 156, pp.221–230. https://doi.org/10.1016/j.future.2024.03.007

Saxena, D. & Sharma, N. 2021. Analysis of Docker performance in cloud environment. In: *Advances in Information Communication Technology and Computing*. Berlin/Heidelberg: Springer, pp.9–18.

Shafiq, D.A., Jhanjhi, N. & Abdullah, A. 2021. Load balancing techniques in cloud computing environment: A review. *Journal of King Saud University – Computer and Information Sciences*, 34, pp.3910–3933.

Shahid, M.A., Islam, N., Alam, M.M., Su'ud, M.M. & Musa, S. 2020. A comprehensive study of load balancing approaches in the cloud computing environment and a novel fault tolerance approach. *IEEE Access*, 8, pp.130500–130526.

Singh, [Initials], et al. 2020. [Title]. *[Journal/Conference]*.

Tong, Z., Deng, X., Ye, F., Basodi, S., Xiao, X. & Pan, Y. 2020. Adaptive computation offloading and resource allocation strategy in a mobile edge computing environment. *Information Sciences*, 537, pp.116–131.

Yin, L., Li, P. & Luo, J. 2021. Smart contract service migration mechanism based on container in edge computing. *Journal of Parallel and Distributed Computing*, 152, pp.157–166.