Journal of Engineering and Technology for Industrial Applications

ISSN ONLINE: 2447-0228

ITEGAM-JETIA

Manaus, v.11 n.55, p. 268-275. September/October., 2025.
DOI: https://doi.org/10.5935/jetia. v11i55.2712

RESEARCH ARTICLE OPEN ACCESS

INVESTIGATION ON PERFORMANCE EVALUATION OF CONTAINERIZED MICROSERVICE FOR EDGE COMPUTING ENVIRONMENT

Pranjit Kakati[1], Abhijit Bora[2]

[1,2]Assam Don Bosco University, Assam

[1]https://orcid.org/0009-0001-3312-7847, [2]http://orcid.org/0000-0002-7754-639X

E-mail: kpranjit05@gmail.com, abhijit.bora0099@gmail.com

ARTICLE INFO

Article History
Received: October 18, 2025
Revised: October 20, 2025
Accepted: October 21, 2025
Published: October 31, 2025

Keywords:
Microservice,
Containerized technology,
Edge computing,
Performance metrics.

ABSTRACT

Microservices have emerged as one of the most popular software development architectures among developers, industries, and academia due to their advantages over traditional monolithic architectures. Many large technology companies are transitioning from monolithic systems to microservice-based architectures. Containerization technologies offer scalable and efficient deployment solutions. This study proposes a methodology for deploying a microservice designed for greenhouse-related computations using Docker and Kubernetes containerization method. The experimental architecture is presented, and the deployed microservice is evaluated using key performance metrics, including Response time(RT), Throughput (Th), and Hits per second (HpS), under varying user loads. Statistical methods are applied to analyze the microservice performance results. The findings conclude that the quality aspects of deploying microservices through the containerization method are stable up to a specific stress level within an edge computing environment.

Copyright ©2025 by authors and Galileo Institute of Technology and Education of the Amazon (ITEGAM). This work is licensed under the Creative Commons Attribution International License (CC BY 4.0).

I. INTRODUCTION

With the fast development of technologies in the communication and information sector, the past decade has witnessed significant advancements in smart living, data transmission, and related fields such as surveillance, navigation, sensors, infotainment, networking, and low-power circuits[1]. Consequently, data traffic across internetwork and processing of data in the cloud has surged tremendously. Although, cloud infrastructure has been leveraged to manage this demand, it faces challenges like high RT, network congestion, and heavy storage burdens. These limitations are particularly critical for applications that are delay-sensitive such as real-time video streaming, augmented reality and autonomous driving which require substantial data processing and low-latency performance [2]. Edge computing emerges as a solution to the increased RT in cloud, where edge computing enabling processing of data at the proximity of the network for downstream data as a representative of services in the cloud and upstream data in representation of services of smart devices/IoTs [3].

The emerging paradigm Edge computing integrates networking, computation, and storage capabilities nearer to end users by shifting them from remote cloud servers to the edge of the network being used. Edge computing has enormous potential to deliver applications, more intelligent services and with enhanced experiences of the users in the of IoT and 5G context [4]. Edge computing is ideal for real-time processing, decision making locally and low latency applications. However, resource constraints and heterogeneity in edge demands a flexible, lightweight and scalable system that support easy and fast development and deployment among service providers. To deploy the functions of cloud computing on the edge, it is required to adopt the architecture and technologies that are suitable for available resources in edge [5]. In this context, microservice applications come up with characteristic like loose coupling, scalability, fine granularity, fault tolerance, low maintenance cost etc [6]. An emerging architectural style called microservices was created to overcome the drawbacks of conventional monolithic software structures. All of an application's components are closely connected and function as a single process in monolithic systems. While this approach can simplify initial development, it often leads to

significant challenges in maintenance, scalability, and system evolution as the application grows in complexity. To get around these issues, the microservice design suggests breaking up an application into a number of tiny, loosely connected, and autonomously deployable services. Each microservice is designed to carry out a certain business task and runs in its own process [7]. These services communicate with one another through lightweight mechanisms, typically HTTP-based APIs or messaging queues. The ability to scale each components independently as per developer's requirement is one of the major advantage of microservices instead of scaling the program as a whole. This approach also enhances resilience of the system, because the system as a whole is not always impacted when one service fails. Additionally, microservices provides the facility of continuous integration and delivery by allowing development teams to independently build, test, deploy, and update services. Thus, microservices promote modularity, scalability and agility in development of application, making them an attractive choice for modern, cloud, and edge native applications.

Container technology is one of the recent advancements in cloud services. It is a container based virtualization where the strategy of machine based virtualization has been upgraded. Here, hardware infrastructure is broken down into virtual entities and different containers used the shared operating system[8]. Containerization offers the advantages of resource allocation and isolation, much like server virtualization technologies. Encapsulation of lightweight, executable package containing codes, setting and system libraries in containerization is the main advantages of the technology [9]. The Docker container was used in this experiment. The management of microservice containers in manual mode with starting and stopping them, monitoring them, balancing of load, and updating them is difficult and error prone for a project. Kubernetes is a container cluster management system that will automatically orchestrate all containers. It is an open source platform used to automate the maintenance, scaling and deployment of containerized microservice applications.

Running a microservice application in Kubernetes involves several stages: development, containerization, deployment, monitoring, and scaling. Each microservice is developed independently and remains loosely coupled using the developer's preferred programming language along with necessary dependencies and settings. To containerize the application, a dockerfile is written for each microservice, which is then used to build Docker images. These containers can be locally built and tested to ensure initial correctness. Once containerization is complete, Kubernetes manifests are created for each microservice. These typically include: (a) Deployment which specifies the Docker image and the number of replicas (pods) to run, (b) Service which exposes the application internally within the cluster or externally to users, and (c) ConfigMaps/ Secrets which store data related to configuration and sensitive information required by the microservices.

At the core of execution, Kubernetes schedules Pods, which are the smallest units of deployment that encapsulates containers. Every deployment ensures that the designated quantity of pod replicas are operational, thereby enabling fault tolerance and scalability. The Kubernetes scheduler dynamically places pods on available nodes, considering resource availability and workload balancing. In a Kubernetes cluster, each pod has a unique IP address that enables inter-Pod communication via APIs or services. To allow external access, services that expose Pods either within the cluster or to external clients. Thus, Kubernetes manages the scheduling, scaling, and orchestration of containerized microservices, providing a robust and flexible platform for running distributed applications[10]. With the increased adaptation of containerization and orchestration technologies in the management and scaling of large applications, it is imperative to evaluate the different performance parameters of microservice execution under varying workloads. Evaluation with an increase in concurrent user threads provides insights into the system's ability to handle high load situations. Performance evaluation will not only help with responsive benchmarking but also help identify limitations and bottlenecks in microservice deployment through Docker and Kubernetes.

II. RELATED WORK

Microservices is now a widely used development and deployment architectural style by different production organizations and is a topic of research and experiments by academics and rehearses. Different aspects of microservice like its architectures, its scaling and deployment, performance evaluation etc. are being studied and researched around the globe. *Vayghan et al.* discussed about architecture of microservices, its deployment, maintainability and scalability through Kubernetes. The author stated that Kubernetes increases application availability through self-healing and failure recovery. They examined the availability of Kubernetes applications by considering the default configuration [10]. Rossi et al. presents on Multilevel Elastic Kubernetes (me-kube) as Kubernetes extension that introduces a hierarchical architecture to control elasticity of application based on microservice.

The author proposes novel proactive and reactive policies of hierarchical control on queuing theory[11]. *Amaral etal.* discuss about the two models for implementation of microservice architecture through containers - master-slave and nested-container. The author experimented in these two models to compare performance of CPU and network running benchmark[12]. *Čilić et al.*presents analysis of edge orchestration platform for Quality of Service (QoS) in terms of their workflow. According to their research, Kubernetes and its distributions may be able to schedule resources on the edge of a network effectively [13]. *Medhi et al.* presents a study on SOAP based service on windows communication. The authors proposes a design and implementation of service oriented prototype research on services of automated teller machine using technology of Windows communication foundation to investigate and predict some aspects of reliability of web services. Here the authors presentnovel prototype architecture, procedures of testing, HTTP transactions and reliability analysis under massive user load[14]. *Aitlmoudden et al.* in their paper proposes a framework based on.

Microservice for scalable data analysis which contain IoT integration and Microservices in agriculture domain. The framework maintains scalability and fault tolerance while offering speedier data processing and agriculture data analysis activities. The authors also address the obstacles like data integration, development of analytical microservices on agriculture and ensuring data security[7]. *Qu et al.* proposed a microservice architecture designed to deliver scalable as well as extensible services within large-scale distributed systems. Their research investigated several microservice deployment strategies on an edge computing platform with Docker containers. Furthermore, the authors conducted a performance analysis of microservices and examined the interference effects arising from concurrent microservice execution under benchmark scenarios [6]. *Huang et al.* proposes a cloud and edge computing framework

with integration of docker container and Kubernetes. They deploy a model for machine learning (Inception V3) in the edge computing platform. The authors conducted an experiment to prove the feasibility of their design and idea [15] .

III. OBJECTIVE AND METHODOLOGY

The primary objective of the proposed experiment is to deploy a containerized orchestration microservice for scalable computation related to greenhouse applications in the edge computing environment. Figure 1 shows the experimental microservice deployment arrangement in Docker and Kubernetes. The key performance parameters of microservice execution are evaluated against various user load conditions. The experimental setup was implemented on an edge server equipped with an Intel Core i3 processor clocked at 2.20 GHz and 8 GB of RAM. The software tools and configurations used included Docker Desktop version 4.39.0 for containerization, Kubernetes CLI (*kubectl*) for orchestration management, Eclipse IDE version 1.34.0, Apache Tomcat 9.0, Maven 3.9.9 for microservice development, and Apache JMeter 5.6.3 for performance testing.This configuration provided a controlled environment for evaluating the microservices while simulating realistic computational loads relevant to greenhouse applications. In the proposed study, a microservice has been developed to compute the equilibrium temperature inside a greenhouse. The equilibrium temperature is estimated on the basis of energy balance approach that considers the balance between incoming solar radiation and outgoing thermal radiation. The equation for equilibrium temperature is shown in Equation 1.

$$T = \left(\frac{S.(1-a)}{4.\sigma.\varepsilon}\right)0.25 \quad [1]$$

Here, T is the equilibrium temperature of the greenhouse (Kelvin),Sis the incoming solar radiation flux (W/m²),ais the albedo or reflectivity of the greenhouse surface (dimensionless, fraction of solar energyreflected), σ presents Stefan Boltzmann constant (5.67×10^{-8} $Wm^{-2}K^{-4}$), ε is the emissivity of the greenhouse surface (dimensionless, accounts for radiative heat loss). In Equation 1, the greenhouse receives solar radiation, of which a fraction is reflected and the rest is absorbed. Hence, the effective absorbed energy is S·(1−a). The factor of 4 in the denominator accounts for the redistribution of the incoming solar energy over the total surface area of the greenhouse.

The greenhouse, in turn, emits heat as longwave radiation according to the Stefan–Boltzmann law, which is proportional to $\sigma\varepsilon T^4$. At thermal equilibrium, the absorbed solar energy equals the emitted radiation, giving rise to the above expression for the greenhouse temperature. In the proposed study, the Docker is employed for containerization of the developed microservice, enabling the encapsulation of the microservice and its dependencies into a portable and lightweight container. Kubernetes is used for orchestration, providing automated deployment, scaling of application and management of the containerized microservices. The computational task selected for the experiment involves determining the equilibrium temperature of soil inside a greenhouse, a process that can benefit from the scalable and distributed computation capabilities of a microservice-based architecture.

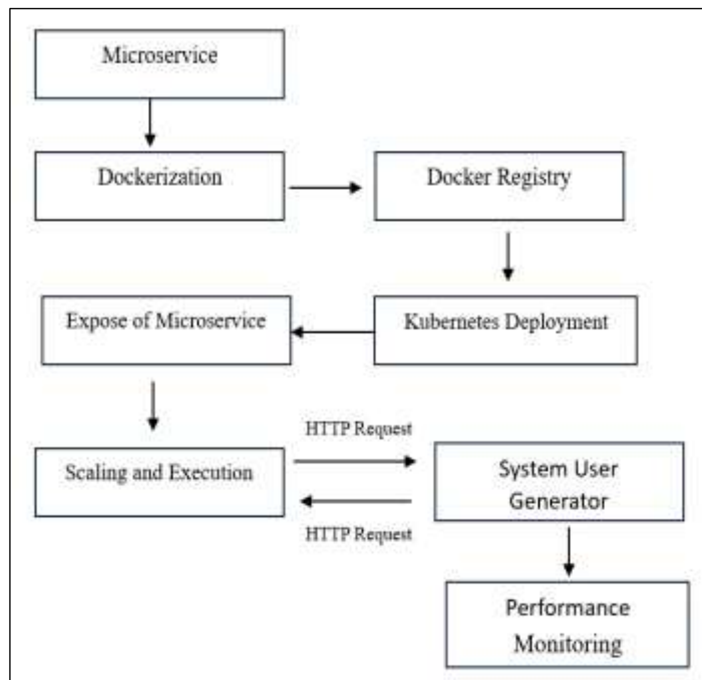

Figure 1: Architecture of experimental microservice deployment arrangement in Docker and Kubernetes.
Source: Authors, (2025).

To evaluate the performance and responsiveness of the deployed architecture, three key performance metrics were considered: (a) RT is a metric that calculates how long it takes to issue a request and get a response; (b) Th, which shows how many requests are handled in a certain amount of time; and (c) HpS, representing the frequency of incoming requests handled by the system. The above mentioned metrics were selected to provide a comprehensive evaluation of both the efficiency and scalability of the deployed microservice under varying load conditions.

IV. EVALUATION OF OVERALL ASSESSMENT

Apache JMeter is used to monitor microservice execution. The metrics considered provides a comprehensive view of the system's behaviour under different load conditions. In the experimental setup, the load on the system was progressively increased by simulating concurrent users (threads) starting from 100 threads, with increments of 100 threads per iteration, continuing up to 3000 threads, or until the system became non-responsive. The load testing was performed using a performance testing tool configured to mimic real-world usage patterns by sending consecutive HTTP requests to the Kubernetes-managed microservices.

Each test iteration involved the capture of the aforementioned performance metrics across all active microservices. The system exhibited stable performance up to 3600 concurrent threads. However, at 3700 threads, the local Kubernetes cluster on which the application was hosted began to fail. The cluster stopped responding and exhibited behaviour indicative of resource saturation or failure in orchestrating container pods due to excessive load. The performance degradation observed beyond this point demonstrates the limitations of the local execution environment, highlighting the need for either hardware resource enhancement or migration to a more scalable production-grade Kubernetes cluster. Table 1 presents variations in performance metrics with increasing trend of concurrent users from 100, 200, 300, 400, and up to 3600 users until the system begins to fail.

Table 1: Variations of performance metrics with increasing trend of concurrent users.

S/N	Nos. of Active Threads	Th	HpS	RT
1	100	238174	1242	121.077
2	200	547243	2849	71.540
3	300	450533	2350	128.667
4	400	10996132	4366	12.516
5	500	10973484	4211	1.418
6	600	9552119	3887	72.843
7	700	10261131	4438	100.643
8	800	10113415	4171	89.645
9	900	11203628	4454	18.429
10	1000	10674796	4233	65.864
11	1100	10364628	4137	74.580
12	1200	11700000	4490	1.771
13	1300	11200000	4321	8.957
14	1400	11700000	4777	132.535
15	1500	11700000	4649	124.609
16	1600	10200000	3986	48.365
17	1700	11900000	4573	235.226
18	1800	10500000	4146	288.839
19	1900	9770220	4099	26.853
20	2000	10929928	4189	35.462
21	2100	9739428	3937	197.728
22	2200	9966200	4258	91.593
23	2300	10900000	4515	79.752
24	2400	9834536	4007	222.614
25	2500	10500000	4066	296.796
26	2600	12300000	4756	108.855
27	2700	9966948	4034	739.483
28	2800	11100000	4081	25.784
29	2900	8617092	3895	1765.557
30	3000	8437656	4084	558.533
31	3200	7159768	1908	120.252
32	3300	5259496	1355	323.741
33	3400	7746748	3069	307.188
34	3500	5663908	2284	747.250
35	3600	2383128	1085	293.261
34	3500	5663908	2284	747.250
35	3600	2383128	1085	293.261

Source: Authors, (2025).

Figure 2 illustrates the comparative behavior of three critical performance metrics including RT, Th, and HpS with increasing numbers of users (active threads). The RT remains relatively stable with only minor fluctuations at low user loads. However, as the number of active threads increases around 2500 - 3500, the RT sharply increases with noticeable spikes. This behavior indicates that as the system approaches saturation, it experiences resource contention and queuing delays, resulting in degraded responsiveness. In the case of Th and HpS increase sharply with the initial increase in active threads, reaching a relatively steady state in the mid-range of user loads up to ~2400 threads. Eventually, Th and HpS show a downward trend as user count grows further, suggesting that the system cannot handle additional requests efficiently once critical resource limits are approached. Overall, the comparative analysis reveals that while the system initially scales well with increasing user load, its performance metrics deteriorate beyond a certain threshold, marking the saturation point. This behaviour is crucial for understanding the system's scalability, capacity planning, and performance

bottlenecks. In this experiment the configuration used in Apache Jmeter are – 100 nos. of threads, 30 seconds as ramp-up period, loop to be continued for 60 seconds. Active threads over time, RT, Th and HpS are the performance metric that has been used while executing microservice. 60 numbers of sample data generated by Jmeter are considered for evaluation of performance. The experimental results for each performance metric are graphically represented to illustrate the system's behaviour under load. Figure3, Figure 4, Figure 5 and Figure 6 presents the variations of increasing active users (threads) with time, RT with respect to increasing concurrent users, Th (requests per second processed successfully) with time and HpS - the total number of requests received by the system per second respectively.

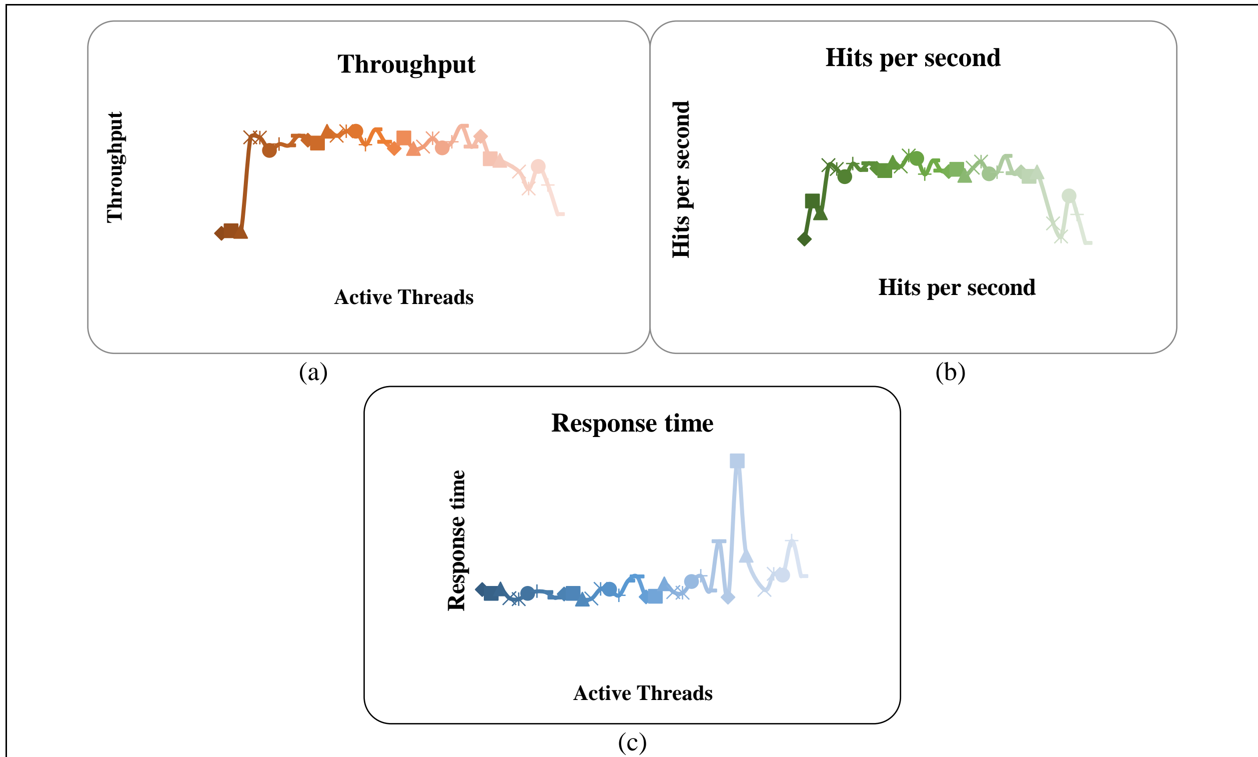

Figure 2. Comparison of responses of performance metrics – RT, Th and HpS with increase of users (active threads)
Soucer: Authors, (2025).

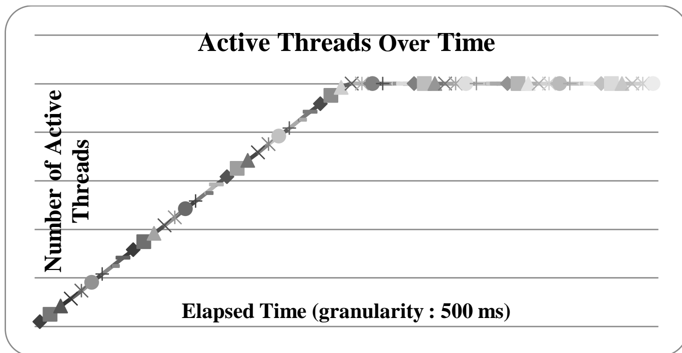

Figure 3: variations of increasing Active users (threads) with time.
Soucer: Authors, (2025).

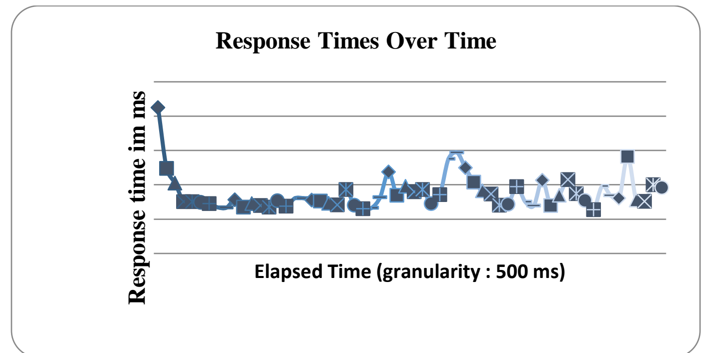

Figure 4: RT over time with respect to increasing concurrent users.
Soucer: Authors, (2025).

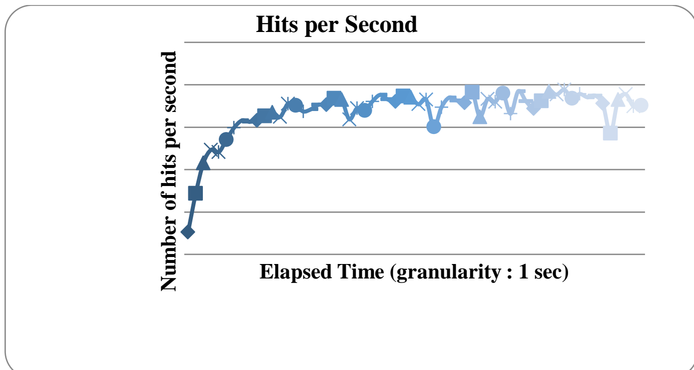

Figure 5: HpS with respect to increasing concurrent users and time
Soucer: Authors, (2025).

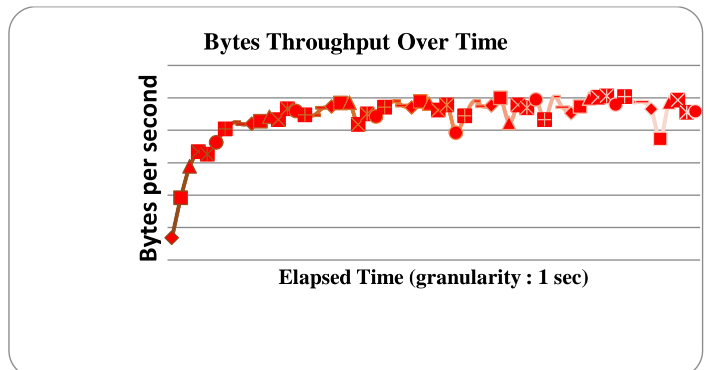

Figure 6: Th over time.
Soucer: Authors, (2025).

These figures collectively provide insights into how the system scales and where its performance bottlenecks begin to manifest. The results serve as a basis for performance tuning and infrastructure planning for high-load microservice deployments in Kubernetes environments.

Figure 7: Relations among the performance metrics including active users, RT, Th and HpS.
Soucer: Authors, (2025).

The correlation among the performance metrices for 100 consecutive request is shown in Figure 7. 60 records are chosen as a data sample for a thread group of 100 users. Table 2 displays the evaluation of the descriptive statistics for the data sample.

Table 2: Descriptive statistics of Th, HpS and RT.

Parameters	Th	HpS	RT
Mean	8850369.733	3367.916667	0.87815331
Standard Deviation	1493271.115	579.4468225	0.25459349
Median	9345600	3547	0.792436217

Soucer: Authors, (2025).

To visually evaluate the distribution nature of the recorded data sample with respect to RT, Th, and HpS, we employ a normal probability plot. Figures 7, 8, and 9 display the normality plots for RT, Th, and HpS respectively.

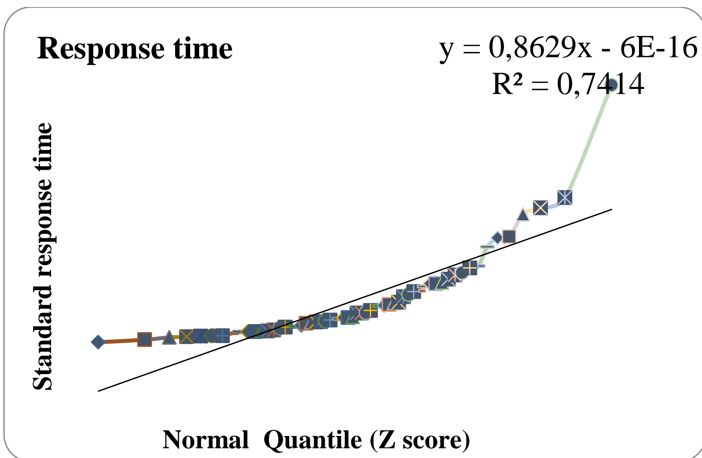

Figure 7: Plot of normality for RT (msec).
Soucer: Authors, (2025).

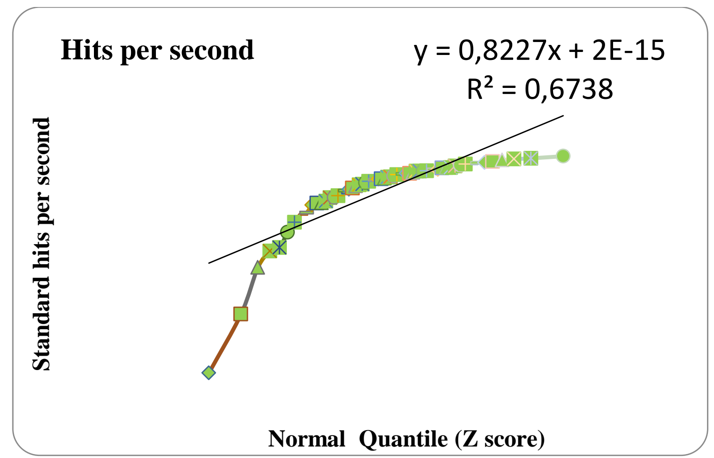

Figure 8: Plot of normality for HpS (msec)
Soucer: Authors, (2025).

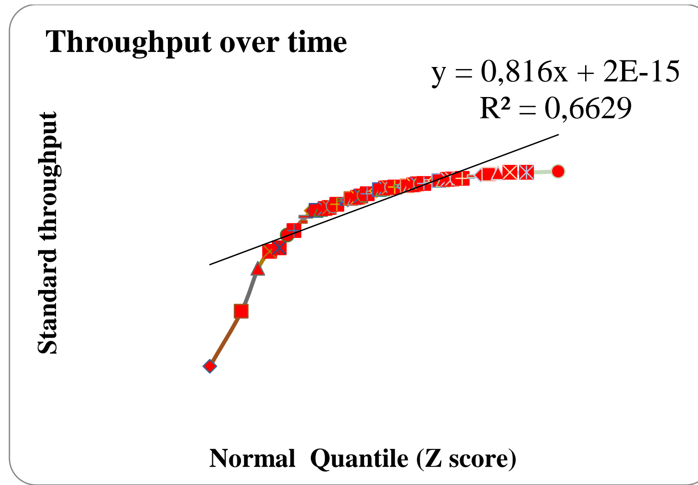

Figure 9: Plot of normality for Th (msec).
Soucer: Authors, (2025).

The normal probability plot revealed a reasonably linear trend with a regression equation y=0.862x, R^2= 0.741 for RT, y=0.822x, R^2= 0.673 for HpS and y=0.816x, R^2= 0.662 for Th. This indicates that the RT data approximately follows a normal distribution, although some deviations exist, likely due to skewness or variability at higher load conditions.

To observe the Goodness of Fit, we evaluate the performance parameters with Anderson Darling (AD) Test. This test is a statistical method used to evaluate whether a given dataset follows a specific probability distribution. It is a type of goodness-of-fit test, to measures how well the observed data align with the theoretical distribution being tested. To further analysis the distribution of data set for RT, Th and HpS, we evaluate through AD test as shown in Equation 2 [14]. The hypothesis H_0 is set that the data of all the performance parameter follows a normal distribution.

$$AnD = N - \left(\frac{1}{N}\right)\sum_{i=1}^{N}(2i-1)[(ln(F(Y_i)) + ln(1 - F(Y_{N+1-i}))] \quad [2]$$

where, N is the number of samples, F() is the Cumulative Distribution Function, Y_i is the ith sample in ascending order, Y_{N+1-i} is the ith sample in descending order. Since the AD statistic depends on the sample size, a modified form is often used to improve accuracy for small samples. It is shown in Equation 3.

$$AnD^* = (AnD) * (1 + 0.75/N + 2.25/N) \quad [3]$$

From the computed AD statistic, a p-value is obtained. p-value is the probability of sample data that is used to check how well the sample follows as specific probability distribution. We have calculated the p-value of AD test using Equation 2 and 3, respectively. The hypothesis of AD test will be rejected if the p-value is smaller than α, the significance level and fail to reject if p-value is higher than than α. Here in this experiment, we use significance level (α) as 0.05 (5%).The AD normality test was applied to the datasets taken for the experiment corresponding to RT, Th, and HpS. The computed p-values for RT, Th, and HpS are found to be 4.521, 8.715, and 1.090 respectively. Since these p-values are significantly higher than the commonly accepted significance threshold (α = 0.05), the test provides strong evidence against the null hypothesis. As such, it can be included that the null hypothesis that the data follows a normal distribution is rejected. This indicates that the distributions of the observed performance metrics deviate from normality. This may be due to workload variations and system resource constraints at higher user loads.

V. CONCLUSION

This study demonstrates the deployment of microservices in Docker and Kubernetes for scalable computation within an edge computing environment. In addition, a performance evaluation of the deployed microservice applications was carried out, focusing on three key metrics including RT, Th, and HpS. The evaluation was conducted using Apache JMeter, which generated test samples to simulate varying user loads and operational conditions. The experimental results offer important insights on how microservice running in Docker and Kubernetes behave. Specifically, the results highlight how the system responds under different levels of concurrency and high user loads. By analyzing the collected performance data, it was possible to identify performance trends, detect bottlenecks, and determine the scalability limits of the deployed architecture. These findings contribute to a deeper understanding of microservice performance in containerized and orchestrated environments. Moreover, the outcomes serve as a reference for optimizing microservice architectures, fine-tuning Kubernetes resource allocation, and guiding future improvements in system design. Ultimately, this ensures higher efficiency, reliability, and scalability in edge computing deployments.

VI. AUTHOR'S CONTRIBUTION

Conceptualization: Pranjit Kakati. Abhijit Bora
Methodology: Pranjit Kakati. Abhijit Bora.
Investigation: Pranjit Kakati. Abhijit Bora.
Discussion of results: Pranjit Kakati. Abhijit Bora.
Writing – Original Draft: Pranjit Kakati. Abhijit Bora.

Writing – Review and Editing: Pranjit Kakati. Abhijit Bora.
Resources: Pranjit Kakati. Abhijit Bora.
Supervision: Pranjit Kakati. Abhijit Bora.
Approval of the final text: Pranjit Kakati. Abhijit Bora

VII. REFERENCES

[1] Md. D. Hossain et al., "The role of microservice approach in edge computing: Opportunities, challenges, and research directions," ICT Express, Jun. 2023, doi: 10.1016/j.icte.2023.06.006.

[2] K. Cao, Y. Liu, G. Meng, and Q. Sun, "An Overview on Edge Computing Research," 2020, Institute of Electrical and Electronics Engineers Inc.doi: 10.1109/ACCESS.2020.2991734.

[3] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge Computing: Vision and Challenges," IEEE Internet Things J, vol. 3, no. 5, pp. 637–646, Oct. 2016, doi: 10.1109/JIOT.2016.2579198.

[4] F. Liu, G. Tang, Y. Li, Z. Cai, X. Zhang, and T. Zhou, "A Survey on Edge Computing Systems and Tools," Proceedings of the IEEE, 2019, doi: 10.1109/JPROC.2019.2920341.

[5] P. Sha, S. Chen, L. Zheng, X. Liu, H. Tang, and Y. Li, "Design and Implement of Microservice System for Edge Computing," in IFAC-PapersOnLine, Elsevier B.V., 2020, pp. 507–511. doi: 10.1016/j.ifacol.2021.04.137.

[6] Q. Qu, R. Xu, S. Y. Nikouei, and Y. Chen, "An Experimental Study on Microservices based Edge Computing Platforms," Apr. 2020, [Online]. Available: http://arxiv.org/abs/2004.02372

[7] O. Aitlmoudden, M. Housni, N. Safeh, and A. Namir, "A Microservices-based Framework for Scalable Data Analysis in Agriculture with IoT Integration," International Journal of Interactive Mobile Technologies, vol. 17, no. 19, pp. 147–156, 2023, doi: 10.3991/ijim.v17i19.40457.

[8] 2020 International Conference on Computer Communication and Informatics (ICCCI). IEEE, 2020.

[9] Y. Huang, R. Zong, K. Cai, and Y. Mao, "Design and implementation of an edge computing platform architecture using docker and kubernetes for machine learning," in ACM International Conference Proceeding Series, Association for Computing Machinery, Mar. 2019, pp. 29–32. doi: 10.1145/3318265.3318288.

[10] L. Abdollahi Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, "Deploying Microservice Based Applications with Kubernetes: Experiments and Lessons Learned," in IEEE International Conference on Cloud Computing, CLOUD, IEEE Computer Society, Sep. 2018, pp. 970–973. doi: 10.1109/CLOUD.2018.00148.

[11] F. Rossi, V. Cardellini, and F. Lo Presti, "Hierarchical scaling of microservices in Kubernetes," in Proceedings - 2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems, ACSOS 2020, Institute of Electrical and Electronics Engineers Inc., Aug. 2020, pp. 28–37. doi: 10.1109/ACSOS49614.2020.00023.

[12] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar, and M. Steinder, "Performance evaluation of microservices architectures using containers," in Proceedings - 2015 IEEE 14th International Symposium on Network Computing and Applications, NCA 2015, Institute of Electrical and Electronics Engineers Inc., Jan. 2016, pp. 27–34. doi: 10.1109/NCA.2015.49.

[13] I. Čilić, P. Krivić, I. Podnar Žarko, and M. Kušek, "Performance Evaluation of Container Orchestration Tools in Edge Computing Environments," Sensors, vol. 23, no. 8, Apr. 2023, doi: 10.3390/s23084008.

[14] S. Medhi, A. Bora, and T. Bezboruah, "Investigations On Some Aspects of Reliability of Content Based Routing SOAP based Windows Communication Foundation Services," International Journal of Information Retrieval Research, vol. 7, no. 1, pp. 17–31, Jan. 2017, doi: 10.4018/ijirr.2017010102.

[15] Y. Huang, R. Zong, K. Cai, and Y. Mao, "Design and implementation of an edge computing platform architecture using docker and kubernetes for machine learning," in ACM International Conference Proceeding Series, Association for Computing Machinery, Mar. 2019, pp. 29–32. doi: 10.1145/3318265.3318288.