International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 10 Issue: 11 | Nov - 2026 SJIF Rating: 8.659 ISSN: 2582-3930

# Workload-Aware Auto-Scaling for Authentication Microservices in Cloud-Native Environments

**Akhil Tiwari[#1], Azaz Ahmad[#2], Madhurendra Giri[#3], Gaurav Raj[#4], Dr. Mohamed Najmus Saqhib[#5]**

[#]Department of CSE, Don Bosco Institute of Technology, Bengaluru, India

[1]akhiltiwari.dev@gmail.com
[2]azazahmaddb2003@gmail.com
[3]madhurendragiri07@gmail.com
[4]gauravraj5804@gmail.com
[5]mn.saqhib@dbit.co.in

***Abstract*** — Workloads faced by cloud-native applications are extremely dynamic and unpredictable in nature; hence effective management of resources is a very important concern in today's era. Containers like Kubernetes have built-in capabilities of auto-scaling which help in managing workload of the applications. But the conventional methods of scaling mainly involve CPU thresholds which often might not react effectively towards unexpected traffic surges or any workload changes. The Proposed Custom V2 mechanism considers performance indicators: Requests Per Second (RPS), p95 response latency and CPU utilization to make the scaling decisions. The microservice model has been implemented using a Node.js based authentication microservice with REST API operations like user registration, login, forgot password, reset password and profile retrieval and integrated with MongoDB. The implementation also includes multiple security mechanisms, namely JWT authentication, bcrypt password hashing, Joi request validation and authentication middleware. The system has been benchmarked against the standard CPU-based Horizontal Pod Autoscaler (HPA) in Kubernetes to evaluate its performance, scalability and resource consumption for different workloads.

**Keywords — Cloud Computing, Kubernetes, Auto-Scaling, Microservices, Docker, Workload-Aware Scaling, Microservices, Prometheus,**

## I. INTRODUCTION

The advent of cloud computing has changed the way applications are delivered and managed today. Companies are now embracing cloud computing in order to enhance their scalability and flexibility. One of the popular approaches used in cloud computing is the use of microservices architecture. This approach entails breaking down a larger application into smaller services.

The authentication service is one of the most critical microservices in distributed systems. The authentication microservice deals with functionalities such as registering users, logging users in, handling tokens, recovering passwords, and implementing authorization logic. Because of their constant use by clients, authentication services may encounter abrupt increases in load during peak hours. are frequently unable to trace how funds are used or verify whether they reached valid recipients

The conventional methodology is based on static allocation of infrastructure wherein resources are assigned regardless of the load condition of the system. As a consequence, such systems have been found to be highly inefficient due to inefficient allocation of resources when traffic is low and overload of the same when traffic is high.

Kubernetes is among the leading container orchestration tools utilized in managing cloud native applications. Kubernetes offers various services like container orchestration, service discovery, load balancing, self-healing, and horizontal pod auto-scaling (HPA). However, Kubernetes scaling techniques depend mostly on CPU threshold techniques that cannot address real world authentication loads.

To overcome this drawback an workload-aware auto-scaling system for a Node.js authentication microservice in a cloud-native environment is developed. The system optimizes the application performance and resource usage based on the key metrics such as Requests Per Second (RPS), p95 response latency, and CPU utilization.

It utilizes Kubernetes for managing containerized application instances and implements a custom scaling mechanism that makes scaling decisions based on the workload. The proposed approach is compared with the standard Horizontal Pod Autoscaler (HPA) of Kubernetes in terms of response time, scalability and utilization of resources under different traffic conditions.

## II. LITERATURE SURVEY

Resource management and auto-scaling techniques have been investigated extensively in cloud computing environments. The rise in the usage of cloud native applications and microservices within containers calls for scalable and intelligent solutions that are able to manage dynamic loads.

The research article in [1] talks about various auto-scaling approaches employed in cloud computing and emphasizes the need for elastic cloud architectures. The article describes how cloud infrastructures assign resources depending on the workload conditions and system parameters like CPU usage and number of requests. Yet, the research also recognizes the problems associated with delays in scaling and effective resource

The research presented in [2] is about adaptive pod autoscaling for horizontal pods on Kubernetes-enabled cloud platforms. The proposed methods provide mechanisms that

are capable of scaling pods based on the demand from applications. This paper proves that using Kubernetesenabled scaling increases the system's availability and scalability. Saurabh Agarwal et al. (2023) discussed reinforcement learning and deep learning-enabled auto-scaling approaches for cloud-based applications. This paper describes how prediction-based scaling methods may enhance the response time of auto-scaling by studying the past behavior of workloads to predict future demand on resources.

Reinforcement learning and deep learning-enabled autoscaling approaches for cloud-based applications have been presented in [3]. This paper describes how prediction-based scaling methods may enhance the response time of autoscaling by studying the past behavior of workloads to predict future demand on resources. S. K. Mondal et al. (2024) investigated into the optimization methodologies that are adopted in Kubernetes for effective management of cloud resources. The optimization is done by means of implementing effective strategies that ensure proper allocation of resources to reduce cost.

According to the research provided in [4], there is a time series forecast model that can be used to manage Kubernetes elasticity. In particular, the paper explores the benefits of workload prediction models in order to enhance scaling efficiency and decrease scaling latency in cloud-native systems.

The paper [5] is an investigation into the optimization methodologies that are adopted in Kubernetes for effective management of cloud resources. The optimization is done by means of implementing effective strategies that ensure proper allocation of resources to reduce cost.

The research in [6] focuses on QoS-based auto-scaling systems for Kubernetes applications. The study provides information on how scaling can be done efficiently to ensure reliability and quality of service for applications in varying workload conditions.

The paper proposed in [7] discusses the concept of using machine learning to scale cloud applications. It illustrates how intelligent scaling can help improve performance and decrease costs through effective workload pattern analysis.

The work done in [8] explores the application of autoscaling methods in cloud-native architectures, including the significance of container orchestration tools for managing scalable infrastructures. Different scaling approaches are examined, and the necessity of adaptive scaling is emphasized.

From the literature review, it is evident that most systems currently rely on CPU threshold scaling techniques. While the technique works well in simple scenarios, it might not capture the true nature of traffic during real authentication operations. Microservices in authentication have the possibility of facing sudden bursts in requests, yet CPU utilization may not be a true indicator of the workload in such cases.

Many studies in the field have focused on infrastructure scaling or predicting future behavior, whereas few works are on workload-awareness in scaling of authentication microservices. Another important aspect about most existing approaches is that they do not incorporate any form of monitoring, analysis, evaluation, or visualization capabilities into their design.

Thus there is a need for a workload-aware scaling system that can analyze authentication traffic, rate of requests, and system indicators simultaneously.

## III. PROPOSED SYSTEM

The Proposed model integrates Central Processing Unit (CPU), Requests Per Second(RPS) and Latency to authenticate microservice in the Node.js framework working in its native environment. In contrast to the pure CPU consumption-based scaling approach, the system takes into account application workload metrics and infrastructure-related metrics for workload assessment to determine the scaling moment. The metrics utilized for the purpose are: request per second (RPS), p95 latency and CPU consumption. The obtained information helps in determining the state of the workload and makes decisions about whether the number of application replicas needs to be scaled up or down or left as it is.

### A. System Architecture

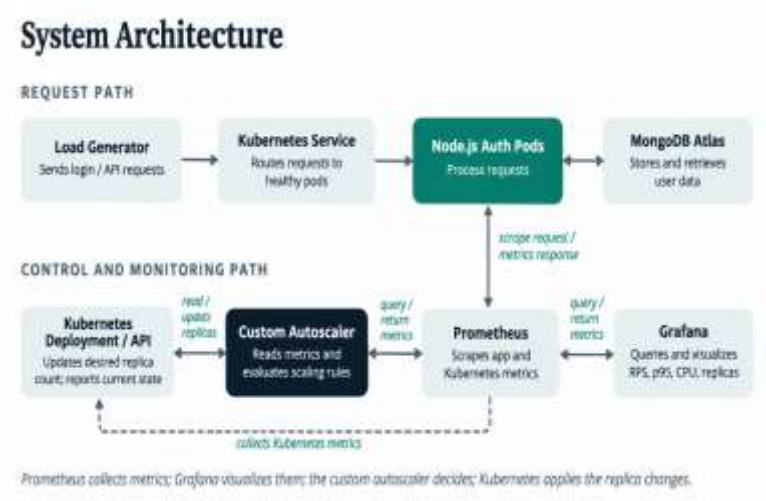



Fig. 3.1 Architecture of the proposed custom V2 model

The architecture includes the following components: an authentication microservice, container orchestration, monitoring, and an auto-scaling controller. An authentication microservice will be implemented on Node.js, packed into a Docker image. Google Kubernetes Engine (GKE) is the platform where the application instances will be deployed. Kubernetes will manage the state of the deployment and change the number of replicas based on the scaling decisions of the controller.

Metrics about application performance will be available via a metrics endpoint. Prometheus will scrape the metrics from the application as well as metrics about containers and Kubernetes resources. Grafana will provide visualization of the collected data about request throughput, latency, CPU usage, and behavior of the replicas when the load is changing.

The auto-scaling controller will query Prometheus about the current performance metrics. The controller will evaluate the collected metrics in regard to scaling conditions and calculate the required number of replicas that is in the range of the defined minimum and maximum replicas. Finally, the controller will interact with the Kubernetes API and set the desired replica number for the authentication service deployment.

**B. Workload-Aware Scaling Mechanism**

In general, auto-scaling strategies rely on various workload and performance metrics to make effective resource allocation decisions.

1) Requests per Second (RPS):* RPS is the number of requests received and processed per second. An increasing RPS may suggest increased load on the authentication service. The controller applies RPS as one of the means to gauge the load that the replicas of the authentication service process and to detect cases where new replicas are needed.

2) 95th-Percentile Response Latency (p95):* p95 latency is the time it takes for the service to respond to 95% of requests. It gives an estimation of the responsiveness of the service and helps detect possible issues that cannot be detected with CPU utilization metric only. Increased p95 latency in conjunction with workload and resources metrics might trigger a scaling decision.

3) CPU Utilization:* The CPU utilization refers to the use of processing resources by the application containers compared to the specified CPU requests. This measure enables the controller to recognize the presence of resource pressure and determine whether the number of replicas available has enough processing power.

The combination of all three measures allows the controller to measure not only the demand on the applications but also its performance. The output of the scaling decision will be a response to growing demand without changing the number of replicas if it is possible.

**C. Scaling Decision and Replica Management**

Controller periodically analyzes the gathered metrics and makes a decision whether scaling is needed. In case when workload shows that capacity is not enough, the controller suggests adding replicas. Scale up decisions could be made due to excessive requests per replica, too high p95 latency when there is enough traffic and high CPU utilization. The controller takes into account the maximal replica number that is set in order to avoid uncontrolled number of replicas.

In case when workload gets decreased, the controller decides whether the current number of replicas should be decreased. The scale down decision is made within the period of stabilization when the low workload state needs to persist. The cool down period allows to control how often the scaling decisions are made. Such approach is taken in order to avoid frequent changes in replica numbers called scaling oscillations.

Another observation made by the controller is a configured minimum number of replicas in order to guarantee the availability of a certain amount of application capacity within the deployment. The controller then checks whether the state of the Kubernetes deployment has changed after making the request for the scaling operation.

**D. Monitoring and Performance Evaluation**

The components of the monitoring subsystem of the proposed system include Prometheus and Grafana. The authentication service has HTTP request counters and response-duration histograms that are used to measure throughput and latency values. The Prometheus tool is used to collect the metrics and provide them via its API for the custom controller to use. Metrics related to the CPU usage and deployment are provided by container and Kubernetes tools.

The measurements made are represented via Grafana dashboards. It allows visualizing and evaluating the behavior of the authentication service during scale-up, steady state, and scale-down operations. The gathered measurements can also be used to compare the performance of the proposed controller with the built-in Kubernetes Horizontal Pod Autoscaler (HPA) with 50% CPU target utilization.

The evaluation process takes into account successful HTTP replies, throughput, average latency of reply, median latency of reply, latency p95, maximum latency of reply, and the number of replicas encountered. This data is collected in order to conduct a performance and scalability analysis of the application under test under the specified workloads. These findings are an initial analysis of the suggested approach; repeated experiments with different workloads are needed to prove the reliability of such performance variations.

## IV. IMPLEMENTATION

**A. Implementation Environment**

The proposed workload-aware autoscaler was built for a Node.js authentication microservice that runs on Google Kubernetes Engine (GKE). The service was packaged with Docker. Launched as a Kubernetes workload. Kubernetes managed the application pods. Kept the service available while the load changed.

Prometheus gathered metrics from the application and from

each container. Grafana displayed the system's behavior. The application recorded the number of HTTP requests and how long each response took, which allowed us to compute request throughput and response-time metrics. These metrics fed the monitoring needed to judge how the authentication service behaved under load.

### B. Baseline and Proposed Autoscaling Approaches

The baseline setup used the Kubernetes Horizontal Pod Autoscaler (HPA). The HPA autoscaler was set to a CPU utilization target of 50%. That means the autoscaler would change the number of application replicas whenever CPU usage crossed that threshold.

The custom autoscaler was built to use three signals: requests per second (RPS) 95th-percentile latency (p95) and CPU usage. The controller queried Prometheus for these numbers. Decided whether the current replica count could handle the workload. Scaling was limited by maximum and minimum replica counts, a period for scale-ups and a stabilization period for scale-downs so that replica changes were not made too often.

### C. Experimental Procedure

The experiments ran on GKE to compare the HPA with the custom autoscaler. Each test used five clients and ran for about 60 seconds. The login endpoint of the authentication service was. The returned HTTP responses, throughput, average latency, median latency, p95 latency, maximum latency and replica counts were logged.

The baseline run was labeled GKE-HPA-01 while the custom autoscaler run was labeled GKE-CUSTOM-V2. The logged data were used to compare how the two autoscalers performed and how they scaled the replicas.

The study looked at experimental runs and the replica counts were not perfectly aligned with the start and finish of each load test. Therefore, the results are descriptive than definitive proof. Running tests, with the same workload would be needed to confirm that the differences are reliable.

## V. RESULT AND DISCUSSION

### A. Quantitative Performance Comparison

The performance of the proposed custom V2 model that was compared with existing CPU-based Kubernetes Horizontal Pod Autoscaler model. This comparison was done using the different evalution metrices, from the GKE experiments. Both the tests that were chosen used five clients at the time and lasted about 60 seconds. Table I shows the application performance that was measured and the number of replicas that were seen.

**TABLE I. Google Kubernetes Engine Autoscaler Performance Comparison**

| Metric | CPU-Based HPA | Proposed Custom V2 |
|---|---|---|
| Successful HTTP 200 responses | 55 | 56 |
| Throughput (requests/s) | 0.8916 | 0.9098 |
| Average latency (s | 5.5625 | 5.4394 |
| Median latency (s) | 5.4875 | 5.4389 |
| 95th-percentile latency (s) | 7.0425 | 5.9913 |
| Maximum latency (s) | 8.3517 | 6.8029 |
| Initial replicas | 1 | 1 |
| Peak observed replicas | 5 | 5 |
| Final observed replicas | 5 | 1 |

Metrics like throughput and average response latency can be used to evaluate auto-scaling performance. In one comparative evaluation the proposed custom V2 autoscaler reached 0.9098 requests per second versus 0.8916 requests per second for CPU based HPA, a 2.04% increase. Average latency decreased from 5.5625 seconds to 5.4394 seconds (a 2.21 % decrease). These metrics can be used to evaluate the effectiveness of different scaling approaches in terms of throughput and response latency.

### B. Throughput and Latency Analysis

Response-time measurements provide a useful basis for comparing the performance of different auto-scaling approaches under varying workloads. In the HPA experiment the p95 latency was 7.0425 seconds. In the proposed custom V2 experiment it dropped to 5.9913 seconds. That's a drop of about 14.93%. The maximum latency also went down. From 8.3517 seconds to 6.8029 seconds. That's a reduction of, around 18.54%.

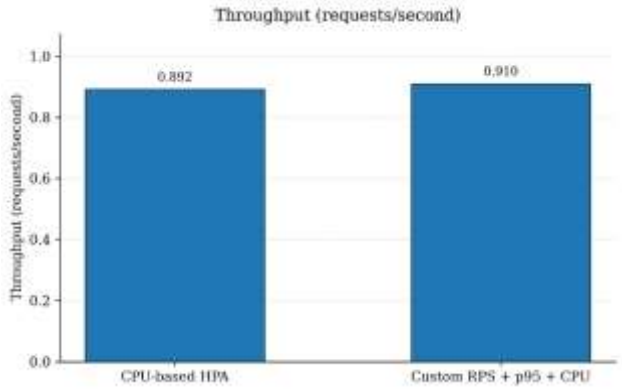



Fig. 5.1 Throughput comparison between CPU-based HPA and the proposed custom V2 model.

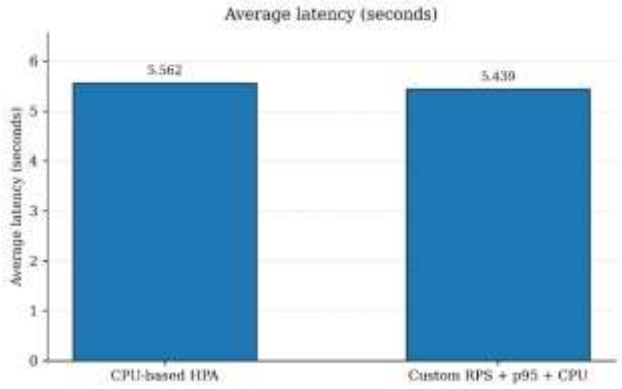



Fig. 5.2 Average response latency comparison between CPU-based HPA and the proposed custom V2 model.

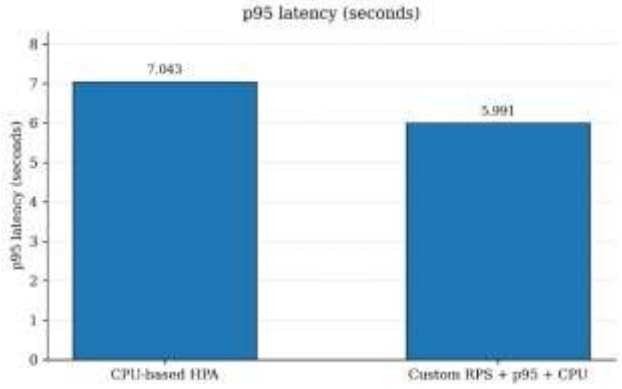



Fig. 5.3 Comparison of 95th-percentile response latency for the two autoscaling approaches.

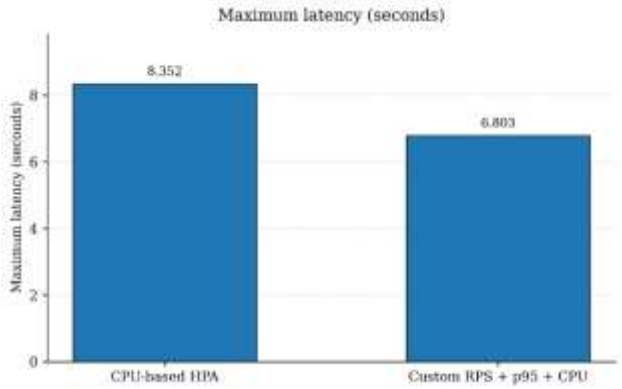



Fig. 5.4 Maximum observed response latency in the recorded GKE experiments.

The recorded measurements show average, p95 and maximum latency in the custom V2 run along with a small increase in throughput. These findings indicate that using application-level demand indicators with CPU utilization might be helpful when deciding how to scale authentication workloads.. Because the results are, from individual runs the differences can't be said to be caused only by the autoscaling strategy.

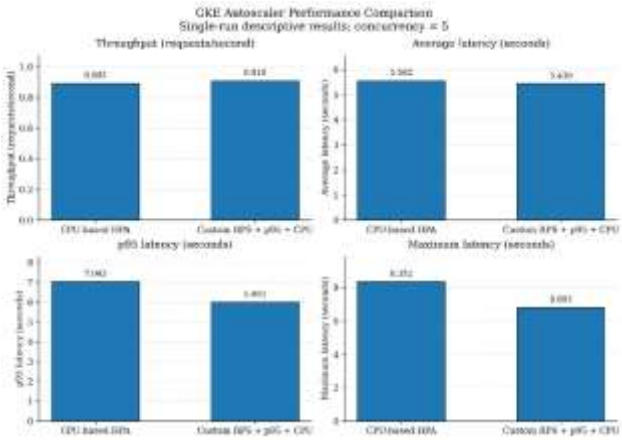



Fig. 5.5 Combined comparison of throughput, average latency, p95 latency, and maximum latency.

### C. Autoscaling Behavior

Both experiments began with one replica. Eventually grew to five. In the custom V2 experiment the controller logs clearly showed the scale-up from one to five replicas. After the peak it then slowly scaled down back to one with delays built in. These delays included cooldown periods and a stabilization window before reducing the number of replicas. This helped avoid dropping replicas quickly when the load dropped just for a short time.

In the HPA experiment the desired number of replicas went from one to four then to five. But the timing of these changes didn't line up with when the load test started or ended. So the final replica counts should be seen as observations, not proof of which system uses resources better.

## VI. CONCLUSION

Cloud-native applications need good resource management to handle changing workloads. workload-aware autoscaling approach for a Node.js authentication microservice based on: requests per second (RPS), p95 response latency, and CPU utilization is proposed. The system combines Kubernetes, Prometheus and Grafana to monitor app performance and manage scaling decisions.

The experimental results on Google Kubernetes Engine (GKE) show that the custom autoscaler achieved 0.9098 requests per second, while the CPU based HPA achieved 0.8916 requests per second, which is an improvement of 2.04%. The average latency reduced from 5.5625 seconds to 5.4394 seconds and the p95 latency reduced from 7.0425 seconds to 5.9913 seconds. The custom autoscaler also scaled replicas between one and five depending on the changes in the workload.

Overall, these results indicate that the proposed workload-aware autoscaler can provide improved throughput and response-time performance compared with CPU-based HPA while dynamically adjusting application replicas according to workload variations.

## VII. REFERENCES

[1]. Sultan Alharthi, "Auto-Scaling Techniques in Cloud Computing: Issues and Research Directions," Sensors, vol. 24, no. 17, pp. 1–20, 2024, DOI: 10.3390/s24175551.

[2]. Zhiqiang Zhou, Chaoli Zhang, Lingna Ma, Jing Gu, Huajie Qian, Qingsong Wen, Liang Sun, Peng Li, and Zhimin Tang, "AHPA: Adaptive Horizontal Pod Autoscaling Systems on Alibaba Cloud Container Service for Kubernetes," arXiv, vol. 2303.03640, pp. 1–12, 2023, DOI: 10.48550/arXiv.2303.03640.

[3]. Saurabh Agarwal, Rahul Kumar, and Ankit Singh, "A Deep Recurrent-Reinforcement Learning Method for Function Autoscaling," arXiv, vol. 2308.05937, pp. 1–10, 2023, DOI: 10.48550/arXiv.2308.05937.

[4]. H. Yuan, Y. Chen, and X. Zhang, "A Time-Series Forecasting Based Approach to Elastic Kubernetes Scaling," Electronics, vol. 13, no. 2,

pp. 1–14, 2024, DOI: 10.3390/electronics13020285.

[5]. S. K. Mondal, S. K. Bera, and A. Ghosh, "On the Optimization of Kubernetes Toward Improved Cloud Resource Management," Mathematics, vol. 12, no. 16, pp. 1–18, 2024, DOI: 10.3390/math12162476.

[6]. V. Rajasekar, P. Kumar, and S. Baskar, "Security-Enhanced QoS-Aware Autoscaling of Kubernetes-Based Applications," Journal of Intelligent Management Decision, vol. 3, no. 3, pp. 45–56, 2024.

[7]. I. Pintye, B. Kovacs, and G. Szeidl, "Enhancing Machine Learning-Based Autoscaling for Cloud Applications," Journal of Grid Computing, vol. 22, no. 4, pp. 1–16, 2024, DOI: 10.1007/s10723-

024-09783-1.

[8]. B. Jeong, "Autoscaling Techniques in Cloud-Native Computing," Journal of Network and Computer Applications, vol. 220, no. 1, pp. 1–15, 2025, DOI: 10.1016/j.jnca.2025.103856.

[9]. H. Guruge, P. B. Guruge, and S. Samarasinghe, "Time Series Forecasting Based Kubernetes Autoscaling Using Hybrid LSTM Models," Frontiers in Computer Science, vol. 7, pp. 1–13, 2025, DOI: 10.3389/fcomp.2025.1509165.

[10]. J. Santos, M. Ribeiro, and A. Costa, "Gwydion: Efficient Auto-Scaling for Complex Containerized Microservices," Journal of Systems Architecture, vol. 150, no. 1, pp. 1–12, 2025, DOI: 10.1016/j.sysarc.2024.103210.

[11]. Jonathan Chamberlain, Jilin Zheng, Zeying Zhu, Zaoxing Liu, and David Starobinski, "Exploiting Kubernetes Autoscaling for Economic Denial of Sustainability," Proceedings of the ACM on Measurement and Analysis of Computing Systems, vol. 9, no. 2, pp. 1–21, 2025, DOI: 10.1145/3727114.

[12]. Vinoth Punniyamoorthy, Bikesh Kumar, Sumit Saha, Lokesh Butra, Mayilsamy Palanigounder, Akash Kumar Agarwal, and Kabilan Kannan, "An SLO-Driven and Cost-Aware Autoscaling Framework for Kubernetes," arXiv, vol. 2512.23415, pp. 1–11, 2025, DOI: 10.48550/arXiv.2512.23415.

[13] A. Abdel Khaleq and I. Ra, "Intelligent Autoscaling of Microservices in the Cloud for Real-Time Applications," *IEEE Access*, vol. 9, pp. 35464–35476, 2021, doi: 10.1109/ACCESS.2021.3061890.

[14] E. Casalicchio, "A Study on Performance Measures for Auto-Scaling CPU-Intensive Containerized Applications," *Cluster Computing*, vol. 22, pp. 995–1006, 2019, doi: 10.1007/s10586-018-02890-1.

[15] D. R. Augustyn, Ł. Wyciślik, and M. Sojka, "Tuning a Kubernetes Horizontal Pod Autoscaler for Meeting Performance and Load Demands in Cloud Deployments," *Applied Sciences*, vol. 14, no. 2, Art. no. 646, 2024, doi: 10.3390/app14020646.