Journal of Basics and Applied Sciences Research (JOBASR)
ISSN (print): 3026-9091, ISSN (online): 1597-9962
Volume 4(3) May 2026
DOI: https://dx.doi.org/10.4314/jobasr.v4i3.12

A Novel Multi-Authority Access Control Scheme for Fine Grained Access to Users Data in The Cloud-Based Storage

Shamsuddeen Rabiu[1*], Sani Muhammad Tanko[2], Eli Adama Jiya[3]
[1]Computer Science Department, Federal College of Education, Katsina State, Nigeria
[2,3]Computer Science Department, Federal University Dutsin-Ma, Katsina State, Nigeria
[*]Corresponding Author Email: shamsrabiu@gmail.com

ABSTRACT
The widespread adoption of microservices architectures on Kubernetes has introduced significant challenges in resource management, particularly the inadequacy of default load balancing under dynamic workloads and issues with pod resource sharing under contention. This paper proposes an integrated auto-scaling platform that combines the Horizontal Pod Autoscaler (HPA), Metrics Server, and Prometheus to dynamically optimize resource utilization in a Kubernetes-in-Docker (KIND) cluster. The experimental platform was deployed on an Ubuntu 22.04 host with a three-node KIND cluster (one master, two workers), using Kubernetes v1.27.3 and Docker v24.0.7, with performance evaluated through CPU utilization and requests per second (RPS) metrics collected via Prometheus and visualized in Grafana. Results demonstrate that HPA effectively responds to workload increases by provisioning additional pods, maintaining system stability and throughput during high-demand periods, with CPU usage and RPS exhibiting predictable scaling behavior aligned with the 15-second Metrics Server scraping interval. The novelty of this work lies in the systematic integration of HPA with Prometheus custom metrics within a KIND environment, extending evaluation across multiple lightweight Kubernetes distributions including microk8s and minikube. This approach enhances scalability, computational efficiency, and cost-effectiveness for microservice-based systems.

Keywords:
Microservice,
Container,
Kubernetes,
Docker,
Auto-scaling,
Load Balancing,
Horizontal Pod Autoscaler,
Prometheus

INTRODUCTION

The traditional monolithic architecture, in which all application components are tightly coupled into a single deployable unit, is increasingly being decomposed into multiple loosely-coupled microservices, each independently implemented and deployed. This paradigm shift improves the flexibility and scalability of deployment, enhances service portability, and increases operational efficiency (Wauters et al., 2023). Microservices, as outlined by Rubak (2023), are compact, autonomous applications characterized by loose coupling, each tailored to a specific business capability. The microservices architecture possesses the ability to be deployed, scaled, and tested independently, and encompasses the development of complex application software by integrating small, self-contained services that communicate through language-independent interfaces (APIs). This method effectively breaks down complex tasks into smaller, autonomous processes, allowing for the dynamic scaling of application instances to accommodate high loads and the reduction of instances during periods of low demand (Balasirisha et al., 2023).

Despite these advantages, critical gaps remain in the existing literature. As application workloads increase, the default Kubernetes load balancing strategy proves inadequate in managing fluctuating traffic due to its static nature, leading to suboptimal performance (Shafiq et al., 2022). The coexistence of multiple applications on the same pod further amplifies concerns about resource contention. While prior studies have evaluated HPA using either Kubernetes Resource Metrics (KRM) or Prometheus Custom Metrics (PCM) individually (Nguyen et al., 2020), none have systematically examined their combined deployment within a Kubernetes-in-Docker (KIND) environment, nor assessed performance across lightweight distributions such as microk8s and minikube. Furthermore, iptables-based load balancing in Kubernetes can degrade performance when excessive

How to cite this article: Shamsuddeen R., Sani M. T. & Eli A. J. (2026). Optimizing Microservice Performance via Horizontal Pod Autoscaling in a Kubernetes-in-Docker (KIND) Clusters. *Journal of Basics and Applied Sciences Research*, *4*(3), 109-220. https://dx.doi.org/10.4314/jobasr.v4i3.12

routing rules are added, and security risks arise from multiple applications residing in the same pod (Shitole, 2022). Effectively managing unpredictable workloads, identifying nodes facing underutilization or excessive demand, regulating traffic, and transferring workloads to ensure optimal performance, remains an open challenge.

The core problem addressed in this paper is the inadequacy of default Kubernetes load balancing and resource management mechanisms under dynamic, real-world microservice workloads. Specifically, the reliance on a simple round-robin strategy and static resource metrics leads to performance degradation, resource contention, and poor scalability when demand fluctuates unpredictably. There is a need for an integrated, metrics-driven auto-scaling solution that can dynamically adapt pod counts and resource allocation without requiring manual intervention or system-wide restarts.Addressing these challenges is significant for several reasons. First, microservices are now the dominant architectural pattern in cloud-native software development, making their performance and reliability critical to industry and research alike. Second, the Kubernetes ecosystem is widely deployed across both enterprise and academic environments, yet practical guidance on integrating HPA with custom monitoring tools in lightweight cluster environments remains limited. Third, optimizing auto-scaling directly translates to reduced operational costs, improved user experience, and more sustainable use of computational resources.

To address these challenges, this paper proposes an integrated platform deploying the Horizontal Pod Autoscaler (HPA) in conjunction with the Kubernetes Metrics Server and Prometheus within a KIND cluster. Experiments were conducted on a three-node cluster (one master, two workers) running Kubernetes v1.27.3 and Docker v24.0.7 on an Ubuntu 22.04 host. Performance was evaluated using CPU utilization and requests per second (RPS) metrics, collected via Prometheus and visualized through Grafana, with the HPA scraping interval set to the default 15 seconds.The novelty of this research work lies in the systematic combination of HPA, Metrics Server, and Prometheus custom metrics within a KIND environment, a configuration not previously evaluated in the literature, and in the extension of this evaluation to multiple lightweight Kubernetes distributions. Unlike prior works that focus on a single scaling metric or distribution, this research provides a comparative, multi-metric perspective on auto-scaling behaviour under realistic workload conditions.

The main contributions of this research are:

(i) a Kubernetes cluster architecture that integrates HPA with Metrics Server and Prometheus to ensure high performance and reliability; and
(ii) an empirical evaluation of HPA responsiveness to CPU utilization changes and increased workloads through dynamic pod provisioning, demonstrating the system's scalability, cost-effectiveness, and computational efficiency.

In the realm of containerized microservices infrastructure, the research aims to achieve the following objectives:

1. To propose a Kubernetes cluster architecture that utilizes container technologies to ensure high performance and reliability, integrating Horizontal Pod Autoscaler (HPA) for streamlined auto-scaling and efficient resource allocation to handle increased workloads.
2. To examine the significance of HPA within the Kubernetes-in-Docker (KIND) cluster, emphasizing its responsiveness to changes in CPU utilization and its ability to manage increased workloads through dedicated client pods. Through the utilization of Metrics Server and Prometheus, HPA enhances performance and upholds high availability by dynamically adjusting the number of pods based on predefined criteria.

Research on Kubernetes, Docker, and microservices has expanded significantly in recent years, spanning autoscaling strategies, load balancing, custom metrics integration, and container scheduling. This section reviews the most relevant prior works, identifies their limitations, and positions the current study within the existing literature.

Microservices and Container Orchestration

The adoption of microservices architectures deployed on Docker and Kubernetes has demonstrated significant performance advantages across various dimensions (Barak & Shiloh, 2024). Lightweight container technologies — including Docker, Kubernetes, and related orchestration tools — have proven instrumental in optimizing cloud resource utilization through increased concurrent deployment (Francois, 2021). However, managing stateful microservices in Kubernetes and achieving consistent distributed deployment remain open challenges (Yepuri et al., 2023). To address these, Yepuri et al. (2023) proposed an HPA-based solution to improve stateful microservice availability, alongside a performance model for refining resource allocation in Kubernetes clusters. While useful, their work focuses narrowly on stateful applications and does not evaluate HPA performance under varying load conditions using custom metrics.

Barbieri (2023) provides a comprehensive exploration of the Horizontal Pod Autoscaler (HPA), covering its basic mechanisms, operational behaviour, and methods for improving performance through custom metrics and resource limits. Although valuable as a conceptual overview, this work lacks empirical evaluation of HPA under realistic workloads or comparative benchmarking against alternative autoscaling strategies.

Load Balancing in Kubernetes

Kubernetes provides built-in load balancing mechanisms; however, default strategies show limitations under dynamic workloads (Hu & Wang, 2021). Shafiq et al. (2022) conducted a broad review of load balancing techniques in cloud computing environments, highlighting the inadequacy of static policies when traffic fluctuates unpredictably. Gao et al. (2023) proposed K-TAHP, a Kubernetes load balancing strategy that integrates TOPSIS and AHP for load evaluation based on CPU, memory, and bandwidth usage. While effective for node-level load distribution, the study omits network-related load factors such as latency and packet loss, a significant gap in scenarios where network performance influences cluster behavior. Furthermore, the paper does not benchmark K-TAHP against existing Kubernetes load balancing methods, leaving its comparative performance unsubstantiated (Basaky et al., 2024).Shitole (2022) addressed dynamic load balancing of microservices in Kubernetes using a service mesh approach, identifying iptables-based load balancing as a performance bottleneck when excessive routing rules are added, and raising security concerns when multiple applications share the same pod. While this work highlights important practical limitations, it does not propose an integrated autoscaling solution or evaluate HPA-driven scaling under CPU and request-rate metrics.

Horizontal Pod Autoscaling and Custom Metrics

Nguyen et al. (2020) provide one of the most directly relevant studies to the current work, focusing on assessing the performance of HPA with respect to Kubernetes Resource Metrics (KRM) and Prometheus Custom Metrics (PCM) through controlled experiments. Their work evaluates HPA operational behavior under both metric types and demonstrates the influence of metric source on autoscaling responsiveness. However, the study does not investigate combined deployment of KRM and PCM, does not test HPA within a Kubernetes-in-Docker (KIND) environment, and does not assess performance across lightweight Kubernetes distributions such as microk8s or minikube. These omissions represent the core gap that the present study addresses.

Song et al. (2019) proposed an auto-scaling system for API gateway services built on Kubernetes and Prometheus, dynamically adapting the number of service instances based on load to enhance resource utilization and service quality. Although their system demonstrates the value of Prometheus-driven scaling, the specific custom metrics employed are not detailed, and no comparison with HPA-based approaches using standard resource metrics is provided.

Casalicchio (2025) and Casalicchio & Perciballi, (2017) highlighted a fundamental measurement problem in Kubernetes: the platform relies on relative CPU metrics obtained from the /cgroup virtual file system via cAdvisor, which may differ from actual CPU usage derived from the /proc file system, potentially causing HPA to underestimate required resources. They propose a correlation model to adjust relative metrics and improve HPA performance for CPU-intensive applications. While this addresses metric accuracy, the study does not evaluate HPA in a KIND or multi-distribution context.

Taherizadeh & Grobelnik (2024) proposed influencing factors for HPA tuning, including a conservative constant that establishes a buffer zone for metric fluctuation and an adaptation interval that reduces unnecessary scaling actions. These refinements improve HPA stability under volatile workloads but are studied independently of the integrated monitoring pipeline (Metrics Server + Prometheus) considered in the present work.

Advanced Autoscaling Strategies

Mondal et al. (2023) introduced a deep learning-based autoscaling strategy using a Gated Recurrent Unit (GRU) model within the Custom Pod Autoscaler (CPA) framework, which forecasts load and adjusts replicas proactively. Their approach outperforms the default HPA in response time and scalability, demonstrating the potential of predictive autoscaling. However, the added model complexity and training data requirements make deployment in lightweight or resource-constrained environments such as KIND impractical. Townend et al. (2025) similarly investigated holistic scheduling in Kubernetes to improve data center efficiency, noting the application of Prometheus custom metrics for HPA operations, though without detailing the specific metrics used or providing a reproducible experimental setup.

Hu & Wang (2021) proposed a Kubernetes autoscaler based on pod replicas prediction, improving upon the default HPA's reactive behavior by forecasting demand. Jiang & Wu (2021) developed a fine-grained horizontal scaling method for container-based cloud platforms. Both works confirm the limitations of the default HPA but focus on predictive extensions rather than the empirical evaluation of existing HPA mechanisms under realistic monitoring integrations.

Research Gap and Positioning

The foregoing review reveals several recurring gaps in the literature. First, prior works that evaluate HPA (Nguyen et al., 2020; Barbieri, 2023) tend to assess either KRM or PCM in isolation, without examining their integrated deployment. Second, no research has systematically evaluated HPA performance within a Kubernetes-in-Docker (KIND) environment in a lightweight, accessible cluster configuration increasingly used in development and testing workflows. Third, cross-distribution evaluation of HPA behavior (e.g., across KIND, microk8s, and minikube) remains absent from the literature. Fourth, existing load balancing studies (Gao et al., 2023; Shitole, 2022) do not integrate a full autoscaling

pipeline combining HPA, Metrics Server, and Prometheus.The present research addresses these gaps by proposing and empirically evaluating an integrated platform that combines HPA with both the Metrics Server and Prometheus within a KIND cluster, while extending evaluation to additional lightweight Kubernetes distributions. This configuration and scope distinguish the current work from all reviewed prior studies.

Table 1: Summary of prior works and gaps

Author's	Focus	Gap addressed
Nguyen et al. (2020)	HPA with KRM vs PCM	No KIND env; no combined KRM+PCM; no multi-distro
Barbieri (2023)	HPA mechanisms overview	No empirical evaluation; no custom metrics testing
Gao et al. (2023)	K-TAHP load balancing	Ignores network latency; no HPA comparison
Shitole (2022)	Service mesh load balancing	No autoscaling solution; no metric-driven evaluation
Mondal et al. (2023)	GRU-based predictive autoscaling	High complexity; not tested in KIND/lightweight envs
Casalicchio (2025)	Metric accuracy in HPA	No KIND evaluation; no Prometheus pipeline
Song et al. (2019)	API gateway autoscaling	Metrics not specified; no KRM comparison
Taherizadeh & Grobelnik (2024)	HPA influencing factors	No integrated monitoring pipeline assessment

MATERIALS AND METHODS

This research adopts an experimental research design in which a Kubernetes-in-Docker (KIND) cluster is configured and instrumented to evaluate the performance of the Horizontal Pod Autoscaler (HPA) under controlled, incrementally increasing workloads. The design follows a baseline-versus-proposed comparison: the baseline represents Kubernetes default resource metric monitoring via the Metrics Server alone, while the proposed configuration augments this with Prometheus custom metrics. Performance is measured across two primary evaluation metrics (CPU utilization and requests per second (RPS)), which is collected at a fixed scraping interval and visualised in Grafana. All configurations, YAML manifests, and Prometheus query expressions used in this study are documented to support reproducibility.

Experimental environment

Experiments were conducted on a physical host machine running Ubuntu 22.04.3 LTS (64-bit, GNOME 42.9, X11 windowing). The host machine specifications are provided in Table 1. A three-node KIND cluster was provisioned on this host, consisting of one master node and two worker nodes, each operating as a virtual machine on the host. The cluster runs Kubernetes client version 1.27.0, server version 1.27.3, Kustomize 5.0.1, and Docker 24.0.7. This environment closely replicates a real-world lightweight deployment scenario and is representative of development and testing workflows in which full cloud infrastructure is unavailable.

Table 2: Host machine specifications

Device name	Gridserver
Processor	12th Gen Intel® Core™ i5-12400 × 12
Memory	64.0 GiB
Graphics	NVIDIA GeForce RTX 4070 / PCIe / SSE2
Disk capacity	500.1 GB
OS	Ubuntu 22.04.3 LTS (64-bit)
Kubernetes version	Client 1.27.0 / Server 1.27.3
Docker version	24.0.7
Kustomize version	5.0.1

Cluster architecture

As depicted in Figure 1, the Kubernetes cluster is divided into a master node (control plane) and two worker nodes. The master node hosts the Kube API Server, Kube Scheduler, Kube Controller Manager, ETCD, Ingress controller, and the Horizontal Pod Autoscaler (HPA). Worker nodes host application pods, Kubelet, cAdvisor, Metrics Server, and Prometheus. The Kube Controller Manager executes the HPA control loop, querying the Metrics Server every 15 seconds (the default sync cycle) to compare observed metric values against configured thresholds and determine whether to scale the number of pods up or down.

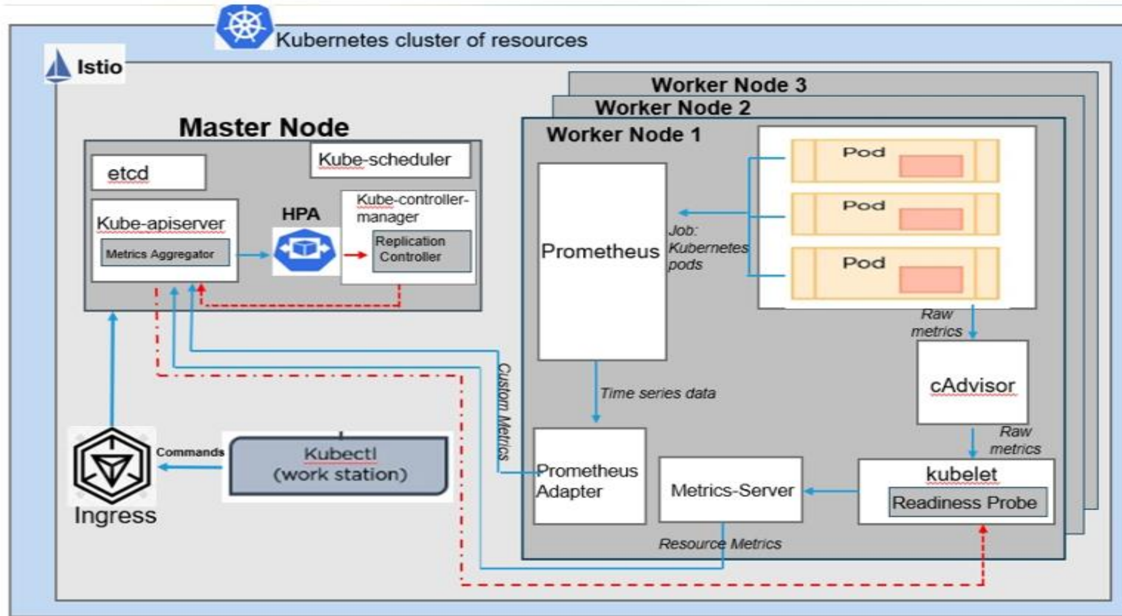

Figure 1: Kubernetes Cluster Architecture

The functions of the components within the Kubernetes cluster architecture are elaborated as delineated in the Figure 1 above.

Master Node:

The master node has the total control over the cluster through the primary components of the Control Plane, encompassing:

- **Kube-controller-manager:** Monitors and guarantees that the cluster operates in the intended state. For instance, if an application should run with three pods, and one is evicted or missing, the kube-controller-manager ensures the creation of a new replica.
- **Kube-scheduler:** Identifies newly created and unscheduled pods, assigning them to nodes while considering factors such as node resource availability and affinity specifications. For example, when a new pod is created and unscheduled, kube-scheduler searches for a suitable node in the cluster and assigns the pod to run on that node.
- **Etcd:** Serves as the backend storage containing all configuration data of the cluster.
- **Kube-apiserver:** Functions as the foundational management component facilitating communication with all other components. Any change to the cluster's state must pass through kube-apiserver. It can also interact with worker nodes through kubelet, and users can manage the cluster by passing kubectl commands to kube-apiserver.
- **HPA (Horizontal Pod Autoscaler):** Efficiently manages scaling events for Deployments, resizing them in response to changes in CPU utilization. This ensures optimization of resource allocation and sustained cluster performance and reliability.
- **Ingress:** Acts as an API and a controller managing external access to services within the cluster. Serving as a gateway for incoming traffic, it routes traffic to the appropriate services based on defined rules.

Worker Node:

Worker nodes are responsible for allocating computing resources in the form of pods and executing them based on instructions from the master node. The components under the worker node include:

- **Pod:** Pods serve as the fundamental unit on the Kubernetes platform. When a Deployment is created, it generates Pods containing containers, linking each Pod to the scheduled Node. Pods persist in their assigned Node until termination or deletion according to the restart policy.
- **Kubelet:** Operating as a local agent, kubelet follows instructions from the master node's kube-apiserver, ensuring the health and vitality of pods. Kubelet plays a crucial role in facilitating metrics scraping by the Metrics-Server and executing readiness probes, which assess the status of new pods and allow traffic once they are deemed ready.
- **cAdvisor (Container Advisor):** cAdvisor supplies statistical running data for the local host or containers, offering insights into resource usage. This data can be exported to kubelet or management tools like Prometheus for monitoring. cAdvisor acts as a monitoring agent,

collecting core metrics such as CPU and memory usage from host machines and running pods, publishing these metrics through an HTTP port.

- **Metrics-Server:** This component exposes metrics to the Metrics Aggregator in kube-apiserver, providing information on CPU and memory usage of individual pods and nodes. These metrics are essential for Horizontal Pod Autoscaler (HPA) decisions, as it collects resource utilization metrics from worker nodes.
- **Prometheus:** Renowned for flexible monitoring, Prometheus exposes monitored targets as endpoints, periodically retrieving metrics through an HTTP server. It has the capability to monitor various targets, including nodes, pods, services, or even itself. Prometheus includes a Pushgateway component, allowing jobs to push metrics directly, even after job termination, and later scrape these metrics by exposing the Pushgateway as an endpoint.

These components collaboratively deliver a comprehensive set of functionalities for managing, monitoring, and scaling applications within the Kubernetes cluster. Prometheus and Metric Server handle metric collection and monitoring, Istio manages service-to-service communication, Ingress directs external traffic routing, cAdvisor monitors container resource usage, and HPA automates pod scaling based on metrics. Their synergy enhances the performance and reliability of containerized microservice applications in Kubernetes.

Scaling mechanisms: Horizontal Pod Autoscaler (HPA)

The Horizontal Pod Autoscaler (HPA) operates as a control loop within the Kube Controller Manager, automatically adjusting the number of pods in a deployment without requiring a system-wide restart. By default, the HPA sync period is 15 seconds. At each sync cycle, the HPA collects resource metrics from the Metrics Server, compares them against predefined thresholds, and calculates the required number of replicas using the following formula:

$$desiresReplicas = \left\lceil currentReplicas * \frac{currentMetricValue}{desiredMetricValue} \right\rceil,$$

where *desiredReplicas* is the target number of pods after scaling, *currentReplicas* is the number of pods currently running, *currentMetricValue* is the latest collected metric value, and *desiredMetricValue* is the configured target threshold. In the experimental setup, *minReplicas* is set to 2 and *maxReplicas* to 4. HPA triggers a scale-up event when the average CPU utilization across running pods exceeds the target threshold, and a scale-down event when it falls below it. Three HPA operational stages are evaluated: (i) monitoring, (ii) scaling policy enforcement, and (iii) scaling decision execution.

Custom metrics and Prometheus integration

Beyond default CPU and memory metrics provided by the Metrics Server, this study integrates Prometheus as an external custom metrics source. Prometheus scrapes metrics from instrumented application pods at a configurable interval, stores them as time-series data, and exposes them via an API that the HPA can query through the Kubernetes Custom Metrics API (Balasirisha et al., 2023). This allows autoscaling decisions to be based on application-level indicators such as requests per second (RPS), which better reflect actual user-facing load than CPU alone (Barbieri, 2023). The specific Prometheus query used for RPS evaluation applies the *rate()* function to the *request_per_second* counter metric over a one-minute window: rate(request_per_second[1m]).

This yields the per-second average rate of increase over the preceding minute, capturing short-term demand spikes while smoothing transient noise. Grafana is connected to Prometheus as a visualisation backend, producing the CPU utilization and RPS dashboards shown in Figures 3 and 4.

Load balancing configuration

Two load balancing strategies are configured within the cluster. First, a Kubernetes Ingress resource managed by Nginx Ingress Controller routes external HTTP/HTTPS traffic to the appropriate backend services using path-based rules. Second, Kubernetes internal service load balancing distributes traffic across available pods using the default round-robin strategy. Both are active throughout all experiments, ensuring that workload distribution is consistent across scaling events. The limitations of iptables-based routing, specifically the degradation of search speed when excessive rules are added, and acknowledged as a boundary condition of the experimental setup.

Table 3: Evaluation metrics and their Description

Metric	Description	Collection method	Unit
CPU utilization	Average percentage of CPU capacity consumed across all running pods in the deployment, relative to the pod resource request	Metrics Server via cAdvisor → Prometheus → Grafana	% (0–100)
Requests per second (RPS)	The per-second rate of HTTP requests received by the application, computed over a 1-minute sliding window using the Prometheus rate() function	Application instrumentation → Prometheus → Grafana	req/s

Metrics are scraped at the default Kubernetes Metrics Server interval of 15 seconds for CPU, and at a configurable Prometheus scrape interval (set to 15 seconds in this study) for RPS. All raw metric data is retained in the Prometheus time-series database for post-experiment analysis.

Benchmarking and validation strategy

To evaluate the effect of HPA integration, a two-condition experimental comparison is conducted:

Table 4: HPA integration experimental comparison

Condition	Configuration	Purpose
Baseline	Kubernetes cluster with Metrics Server only; no HPA active; static pod count	Represents default Kubernetes behavior without autoscaling
Proposed	HPA enabled; Metrics Server + Prometheus custom metrics; dynamic pod scaling	Evaluates autoscaling responsiveness and resource efficiency

Workload is generated by incrementally increasing the rate of HTTP requests to the deployed microservice. CPU utilization and RPS are recorded continuously across both conditions. HPA behavior is assessed by observing pod count changes in response to metric threshold crossings, with the 15-second sync period as the expected response latency. Scaling decisions are logged via *kubectl get hpa* and *kubectl describe hpa* commands at regular intervals throughout each experiment.

Reproducibility details

To support full reproducibility of results, the following configuration artifacts are documented and available:

Table 5: configuration artifacts and their description

Artifact	Description
HPA YAML manifest	Defines minReplicas=2, maxReplicas=4, target CPU utilization threshold, and Metrics Server reference
Prometheus scrape config	Defines scrape interval (15s), target job (nginx-exporter), and metrics endpoints
Grafana dashboard config	Visualises CPU utilization and RPS time-series using the rate(request_per_second[1m]) query
KIND cluster config	Three-node cluster (1 master, 2 workers) with Kubernetes 1.27.3 and Docker 24.0.7 on Ubuntu 22.04
Workload generator	HTTP load generator incrementally increasing request rate to trigger HPA scaling events

All YAML configurations are structured to allow direct replication on any compatible Ubuntu 22.04 host running Docker and KIND. The Prometheus query language expressions used in this study are fully specified in Section 3.4 above.

Analysis of performance metrics and data sources

Performance metrics were collected from the Kubernetes cluster using Prometheus as the primary time-series data source and Grafana as the visualisation frontend. Two metrics were monitored continuously throughout all experiments: CPU utilization across cluster nodes (Figure 3) and the request rate of the deployed microservice, measured in requests per second (Figure 4). The request rate was derived from the request_per_second counter, which accumulates the total number of HTTP requests processed by the application. Applying the Prometheus rate() function over a one-minute sliding window

converts this cumulative counter into a per-second average rate of increase:

rate(request_per_second[1m])

This expression computes the average rate of change over the preceding 60 seconds, effectively smoothing transient spikes while remaining responsive to sustained shifts in application load.

CPU consumption by the Prometheus process was tracked using the process_cpu_seconds_total metric, scoped to the Prometheus job:

rate(process_cpu_seconds_total{job="prometheus"}[1m])

The rate() function is designed exclusively for counter-type metrics; those representing monotonically increasing cumulative values such as total CPU time or total request counts. Applied to such metrics, it yields the per-second rate of change over the specified time window, enabling meaningful comparison of load levels across different time periods and scaling events. All query results were streamed into Grafana for real-time dashboard visualisation, as presented in Figures 2 and 3. Table 3 summarized the Prometheus Query Language (PromQL) components used in this research.

Table 6: Prometheus Query Language (PromQL) Components Summary

PromQL element	**Description**
rate(<expr>[<range>])	Computes the per-second average rate of increase of a counter metric over the specified time range
request_per_second	Cumulative counter of total HTTP requests received by the application
process_cpu_seconds_total{job="prometheus"}	Total CPU time (seconds) consumed by the Prometheus process, scoped to the "prometheus" job
[1m]	Time range selector — rate is calculated over the preceding one-minute window

RESULTS AND DISCUSSION

Performance metrics were collected at the default Kubernetes Metrics Server scraping interval of 15 seconds. This section evaluates HPA behavior using Kubernetes Resource Metrics, specifically CPU utilization and requests per second (RPS), under incrementally increasing workloads within this default scraping period.

CPU utilization and request rate analysis

Figures 2 and 3 present the time-series graphs for CPU utilization and RPS, respectively, recorded during the experiment. CPU utilization exhibits dynamic variation, rising sharply during periods of high incoming traffic and declining as the number of concurrently processed requests stabilizes. This behavior is consistent with the kubelet scraping interval of approximately 30 seconds, which governs how frequently raw metrics are collected from cAdvisor and reported to the Metrics Server, as illustrated in the cluster architecture diagram (Figure 1).

Concurrently, the RPS metric follows a corresponding pattern — surging as incoming request volume increases and tapering as the system absorbs the additional processing load. Together, these two metrics capture the interaction between CPU resource consumption, request throughput, and the periodic metric collection mechanism. The observed patterns provide a comprehensive view of system dynamics under load, highlighting the responsiveness of CPU resources and request rates to workload fluctuations, both of which are critical indicators for scalability planning and performance optimization.

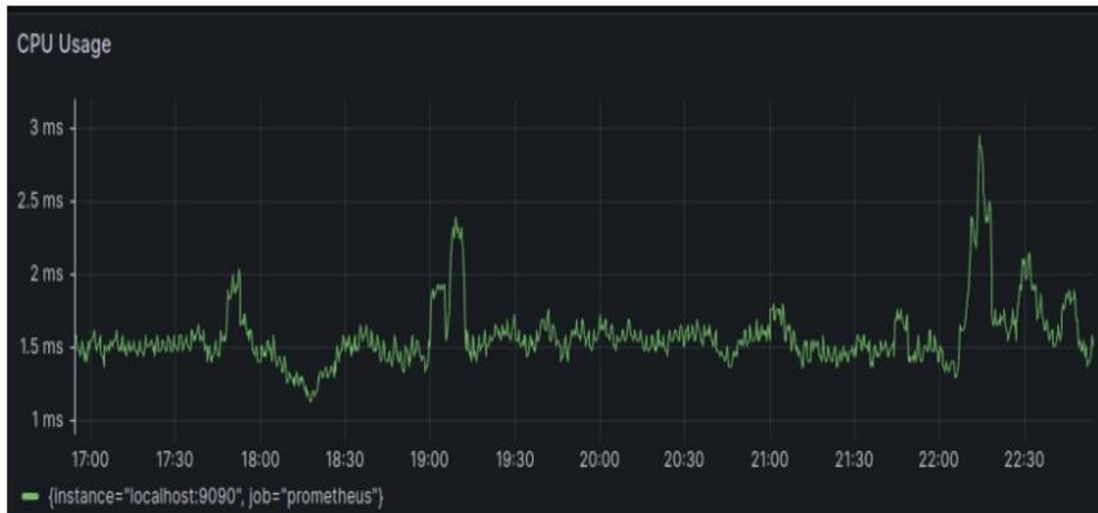

Figure 2: Monitoring CPU usage in Kubernetes system using the Grafana tool

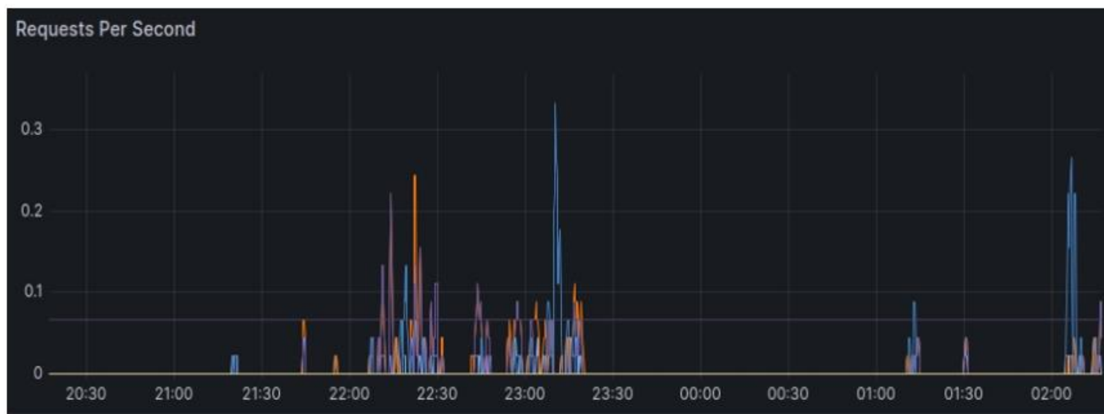

Figure 3: Monitoring request Per Second within the Kubernetes system using the Grafana tool

HPA performance and platform evaluation

The proposed platform combines Kubernetes and Docker to deliver a high-performance, reliable microservice deployment environment. HPA serves as the core autoscaling mechanism, dynamically adjusting pod count in response to real-time metric thresholds, thereby ensuring efficient resource allocation without manual intervention. This approach reduces unnecessary resource consumption during low-demand periods while maintaining throughput during peak loads, making the system scalable, cost-effective, and computationally efficient.

Evaluation of HPA within the KIND cluster confirms its responsiveness to CPU utilization changes and its capacity to manage increased workloads through dynamic pod scaling. The configured 15-second Metrics Server scraping interval proved effective in capturing workload fluctuations with sufficient granularity to trigger timely scaling decisions. Analysis of CPU utilization and RPS collectively revealed predictable, load-driven scaling behavior, with pod counts adjusting in alignment with metric threshold crossings, providing useful performance insights that underpin the system's reliability and suitability for production-grade microservice deployments.

CONCLUSION

This research addressed the critical challenge of inefficient resource management in Kubernetes-deployed containers, where static resource allocation and default load-balancing mechanisms often fail to sustain operational efficiency. To resolve these limitations, this study implemented and evaluated an automated scaling

platform integrating the Horizontal Pod Autoscaler (HPA), Metrics Server, and Prometheus within a localized Kubernetes in Docker (KIND) environment.

Experimental results validated the practical viability of the proposed platform, demonstrating that the HPA effectively mitigates workload surges by dynamically provisioning additional pods. This automated response preserved system stability and throughput during peak demand periods. Analysis of CPU utilization and requests per second (RPS) revealed a highly predictable, load-driven scaling trajectory, while the Metrics Server's 15-second scraping interval proved sufficiently granular to capture rapid workload fluctuations for timely scaling execution. Ultimately, these findings clarify the crucial interplay between resource metrics, request throughput, and dynamic pod scaling, proving that the platform delivers a scalable, computationally efficient, and cost-effective framework for containerized microservices.

Despite these promising results, sveral limitations warrant acknowledgment. The evaluation was restricted to a single-machine, three-node KIND cluster using synthetic HTTP traffic, which does not fully capture the network complexity, scale, or erratic burstiness of a production cloud deployment. Furthermore, the analysis relied on high-level behavioral observations of CPU and RPS metrics rather than granular quantitative benchmarks, such as tail latencies, pod scale-out delays, or direct comparisons against a non-HPA baseline, and was restricted to CPU-bound scenarios rather than memory-constrained or I/O-bound workloads.

To build upon this foundation, future research should transition the platform to multi-host, production-grade managed cloud environments (e.g., GKE, Amazon EKS) and explore lightweight distributions like MicroK8s and MiniKube to improve generalizability. Additionally, future work should investigate multi-dimensional resource optimization by combining the HPA with Vertical Pod Autoscaling (VPA) and Cluster Autoscalers. Finally, benchmark studies comparing this reactive HPA model against predictive, machine learning-based autoscaling strategies, while incorporating broader Prometheus custom metrics such as memory utilization and network I/O will be vital to further minimizing response latencies and optimizing complex containerized architectures.

Conflict of Interest

The authors declare there is no competing interest regarding publication of this article

REFERENCE

Balasirisha, J., Mounika, K., Saxena, M., Svsrk, K., & Mv, R. K. (2023). *High Performance and Scalable Microservices Architecture using Kubernetes*. *12*(1), 85–90. https://doi.org/10.21275/SR221229142739

Barak, A., & Shiloh, A. (2024). A distributed load-balancing policy for a multicomputer. *Software: Practice and Experience*, *15*(9), 901–913. https://doi.org/10.1002/spe.4380150905

Barbieri, C. (2023). *The basic working mechanism of the Horizontal Pod Autoscaler (HPA) in Kubernetes involves monitoring, scaling policies, and the Kubernetes Metrics Server*. https://caiolombello.medium.com/kubernetes-hpa-custom-metrics-for-effective-cpu-memory-scaling-23526bba9b4.

Basaky F. D, Omonori J D., Durotola I. T., & Ogbe K. U. (2024). *Performance Evaluations of the Round Robin Load Balancing Algorithm Using Nine Qualitative Metrics.* *3*(6), 67-71. https://dx.doi.org/10.4314/jobasr.v3i6.9

Casalicchio, E. (2025). A study on performance measures for auto-scaling CPU-intensive containerized applications. *Cluster Computing*, *22*(3), 995–1006. https://doi.org/10.1007/s10586-018-02890-

Casalicchio E. & Perciballi V., (2017). Auto-Scaling of Containers: The Impact of Relative and Absolute Metrics. *Proceedings - 2017 IEEE 2nd International Workshops on Foundations and Applications of Self Systems, FASW 2017*. 207-214. https://doi.org/10.1109/FAS-W.2017.149

Francisco, S. (2023). *Kubernetes HPA*. https://www.kubecost.com/kubernetes-autoscaling/kubernetes-hpa/

Francois, A. (2021). *What Are Containers Technology and How to Choose Your Container Platform?* https://www.alibabacloud.com/blog/what-are-containers-technology-and-how-to-choose-your-container-platform_598303

Gao, R., Xie, X., & Guo, Q. (2023). K-TAHP: A Kubernetes Load Balancing Strategy Base on TOPSIS+AHP. *IEEE Access*, *11*(September), 1–1. https://doi.org/10.1109/access.2023.3313643

Hu, T., & Wang, Y. (2021). A Kubernetes Autoscaler Based on Pod Replicas Prediction. *Proceedings - 2021 Asia-Pacific Conference on Communications Technology and Computer Science, ACCTCS 2021*, 238–241. https://doi.org/10.1109/ACCTCS52002.2021.00053

Jiang, C., & Wu, P. (2021). A Fine-Grained Horizontal Scaling Method for Container-Based Cloud. *Scientific Programming*, *2021*. https://doi.org/10.1155/2021/6397786

Mondal, S. K., Wu, X., Kabir, H. M. D., Dai, H. N., Ni, K., Yuan, H., & Wang, T. (2023). Toward Optimal Load Prediction and Customizable Autoscaling Scheme for Kubernetes. *Mathematics*, *11*(12), 1–30. https://doi.org/10.3390/math11122675

Nguyen, T. T., Yeom, Y. J., Kim, T., Park, D. H., & Kim, S. (2020). Horizontal pod autoscaling in kubernetes for elastic container orchestration. *Sensors (Switzerland)*, *20*(16), 1–18. https://doi.org/10.3390/s2016462

Prometheus.io. (2023). *From metrics to insight; Prometheus Overview*. https://prometheus.io/docs/introduction/overview/

Rubak, A. (2023). *Dimensioning microservices on Kubernetes platforms using machine learning techniques* [Master's thesis, Karlstad University]. DiVA. https://www.diva-portal.org/smash/record.jsf?pid=diva2:1776407

Shafiq, D. A., Jhanjhi, N. Z., & Abdullah, A. (2022). Load balancing techniques in cloud computing environment: A review. *Journal of King Saud University - Computer and Information Sciences*, *34*(7), 3910–3933. https://doi.org/10.1016/j.jksuci.2021.02.007

Shitole, A. S. (2022). *Dynamic load balancing of microservices in Kubernetes clusters using service mesh* [Master's thesis, National College of Ireland]. NORMA@NCI Library. https://norma.ncirl.ie/5943/

Song, M., Zhang, C., & Haihong, E. (2019). An Auto Scaling System for API Gateway Based on Kubernetes. *Proceedings of the IEEE International Conference on Software Engineering and Service Sciences, ICSESS*, *2018-Novem*, 109–112. https://doi.org/10.1109/ICSESS.2018.8663784

Srirama, S. N., Adhikari, M., & Paul, S. (2020). Application deployment using containers with auto-scaling for microservices in cloud environment. *Journal of Network and Computer Applications*, *160*(August 2019). https://doi.org/10.1016/j.jnca.2020.102629

Taherizadeh, S., & Grobelnik, M. (2024). Key influencing factors of the Kubernetes auto-scaler for computing-intensive microservice-native cloud-based applications. *Advances in Engineering Software*, *140*(October 2019), 102734. https://doi.org/10.1016/j.advengsoft.2019.102734

Townend, P., Clement, S., Burdett, D., Yang, R., Shaw, J., Slater, B., & Xu, J. (2025). Invited paper: Improving data center efficiency through holistic scheduling in kubernetes. *Proceedings - 13th IEEE International Conference on Service-Oriented System Engineering, SOSE 2019, 10th International Workshop on Joint Cloud Computing, JCC 2019 and 2019 IEEE International Workshop on Cloud Computing in Robotic Systems, CCRS 2019*, *1*, 156–166. https://doi.org/10.1109/SOSE.2019.00030

Wauters, T., Volckaert, B., & Turck, F. De. (2023). *gym-hpa : Efficient Auto-Scaling via Reinforcement Learning for Complex Microservice-based Applications in Kubernetes*.

Yepuri, V. K., Polamarasetty, V. K., Donthi, S., & Gondi, A. K. R. (2023). *Containerization of a polyglot microservice application using Docker and Kubernetes*. 1–10. http://arxiv.org/abs/2305.00600.