

Evaluating Kubernetes Autoscaling and Load Balancing to Improve Student Feedback Application Performance

Arvita Agus Kurniasari^{1)*}, Bety Etikasari²⁾, Aji Seto Arifianto³⁾, Lukie Perdanasari⁴⁾,
Ahmad Fahriyannur Rosyady⁵⁾

^{1,2,3,4,5)}Department of Information Technology, State Polytechnic of Jember, Jember, Indonesia

¹⁾arvita@polije.ac.id, ²⁾bety.etikasari@polije.ac.id, ³⁾ajiseto@polije.ac.id, ⁴⁾lukieperdanasari@polije.ac.id,

⁵⁾ahmad.fahriyannur@polije.ac.id

Submitted : Aug 3, 2026 | **Accepted :** Aug 27, 2026 | **Published :** Oct 5, 2026

Abstract: The increasing use of digital platforms in higher education requires student feedback applications to maintain stable performance under growing workloads. This study evaluates Kubernetes autoscaling and load balancing through an integrated deployment in which multiple application pods, Kubernetes Service-based traffic distribution, Traefik routing, and Horizontal Pod Autoscaler configuration were enabled together. A conventional single-container deployment and the Kubernetes-based deployment ran on the same virtual machine and used the same intended duration-based workload pattern with a maximum of 294 virtual users. Each deployment scenario was evaluated through one independent workload execution. The results are interpreted as descriptive observations of the tested deployment configurations rather than statistically replicated experiments. Interval-level observations within each run were treated as temporally correlated measurements rather than independent experimental replications. Across the recorded runs, the Kubernetes-based deployment recorded 75.36% lower mean virtual-machine-level processor utilization, 95.87% lower mean interval-level p99 request duration, and 139.13% higher mean request rate than the single-container deployment. For the retained workload-band time series above 200 virtual users, mean processor utilization was 99.22% for the single-container deployment and 39.53% for the Kubernetes-based deployment. The corresponding mean interval-level p99 values were 23.605 and 0.188 seconds, while mean request rates were 45.32 and 255.10 requests per second. The single-container deployment recorded failed-request rates of up to 3.14 requests per second in the retained time series, whereas no failed requests were recorded for the Kubernetes deployment. These results support the performance comparison of the complete configuration in which autoscaling and load-balancing mechanisms were enabled. However, replica-count, HPA-event, and per-pod traffic-distribution data were not collected, so the individual causal contributions of autoscaling and load balancing cannot be isolated.

Keywords: Autoscaling; Kubernetes; Load balancing; Resource management; Student feedback application

INTRODUCTION

The rapid adoption of digital platforms in higher education has increased the demand for reliable and scalable information systems to support academic services. One of the commonly implemented systems is a student feedback application, which facilitates communication between students and educational institutions regarding learning processes, academic services, and institutional improvements. As the number of users and submitted feedback increases, these applications require sufficient computational resources to maintain responsive services and prevent performance degradation during periods of high demand.

Conventional deployment approaches using a single application instance often face scalability limitations because all incoming requests are processed by a fixed amount of available resources. Under increasing workloads, this architecture may experience higher response latency, inefficient resource utilization, and increased risk of service disruption. Therefore, adaptive infrastructure management mechanisms are required to enable applications

*Arvita Agus Kurniasari



This is anCreative Commons License This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.

to adjust computational resources according to workload conditions. Container-based technologies provide lightweight application packaging, portability, and flexible resource management, while Kubernetes has become a standard platform for orchestrating containerized applications (Burns et al. 2016)(Carrión 2022).

Kubernetes has emerged as a widely adopted container orchestration platform that provides mechanisms for application deployment management, service discovery, load distribution, and automatic scaling. Through autoscaling mechanisms, Kubernetes can dynamically adjust the number of application instances based on resource utilization, while load balancing distributes incoming requests across available instances to improve application responsiveness (Burns et al. 2016)(Carrión 2022). Several studies have investigated Kubernetes performance optimization, including resource allocation, workload management, and automatic scaling strategies. However, many previous studies have focused on general cloud applications, microservices environments, or benchmark workloads rather than educational applications with specific user interaction patterns (Mondal et al. 2023)(Martínez Saucedo et al. 2025).

The implementation of Kubernetes for academic service applications requires further evaluation because educational platforms often have different workload characteristics compared with commercial cloud systems. Student feedback applications generally experience fluctuating access patterns, where usage intensity may increase during specific academic periods. This condition creates a need for an infrastructure approach that can maintain application performance while optimizing resource utilization. However, empirical evaluations of integrated Kubernetes-based deployment architectures for student feedback applications remain limited.

Therefore, this study aims to evaluate Kubernetes autoscaling and load balancing as jointly implemented within a Kubernetes-based deployment for a student feedback application. The evaluation compares the complete Kubernetes configuration with a conventional single-container deployment under a controlled, duration-based workload. Performance is assessed using virtual-machine-level processor utilization, interval-level p99 request duration, request processing rate, and failed-request rate. The analysis quantifies the observed performance of the integrated configuration and does not claim that the contribution of HPA or load distribution was isolated experimentally.

The contributions of this study are threefold. First, it provides an empirical evaluation of a Kubernetes deployment in which Horizontal Pod Autoscaler configuration and service-based load balancing were implemented for a student feedback application. Second, it reports overall and workload-band performance evidence using processor utilization, interval-level p99 request duration, request processing rate, and failed-request rate. Third, it identifies the configuration evidence, monitoring data, and experimental limitations required to interpret autoscaling and load-balancing results responsibly. The study does not propose a new autoscaling algorithm or load-balancing policy.

LITERATURE REVIEW

Container Orchestration and Kubernetes-Based Deployment

The increasing adoption of cloud-native applications has encouraged the use of container technology due to its ability to provide lightweight, portable, and consistent application environments. Unlike traditional virtualization approaches, containers share the host operating system kernel, allowing applications to be deployed with lower overhead while maintaining isolation between services. However, managing multiple containers in production environments requires an orchestration mechanism to handle deployment, resource allocation, service discovery, and scaling processes (Carrión 2022) (Dogani, Namvar, and Khunjush 2023).

Kubernetes has become one of the most widely adopted container orchestration platforms because it provides automated mechanisms for managing containerized applications. Kubernetes introduces several components, including pods, services, and controllers, which enable applications to maintain desired operational states. Previous studies have demonstrated that Kubernetes can improve application flexibility by allowing dynamic resource management and automated deployment processes, particularly in cloud-based and distributed application environments (Burns et al. 2016)(Carrión 2022).

Although Kubernetes provides comprehensive management capabilities, its effectiveness depends on proper configuration of workload characteristics, resource allocation, and scaling strategies. Poor resource configuration may lead to unnecessary resource consumption or insufficient capacity during workload increases. Therefore, evaluating Kubernetes performance in specific application domains remains important because workload behavior differs between commercial, industrial, and educational applications.

Kubernetes Autoscaling Mechanism

Autoscaling is one of the important mechanisms in Kubernetes for maintaining application performance under dynamic workloads. The Horizontal Pod Autoscaler enables Kubernetes to automatically increase or decrease the number of application instances based on monitored resource utilization, while other scaling approaches focus on adjusting computational resources allocated to existing instances. These mechanisms allow applications to adapt to changing workloads without requiring manual intervention.

Several studies have demonstrated that autoscaling effectiveness depends on workload characteristics, monitored metrics, scaling thresholds, and resource configurations (Augustyn, Wycislik, and Sojka 2024)(Alharthi et al. 2024).

However, most existing studies focus on benchmark applications, cloud computing scenarios, or microservice-based systems. The evaluation of autoscaling for educational information systems remains limited, particularly for applications with fluctuating but relatively predictable usage patterns, such as student feedback platforms. Therefore, further empirical evaluation is required to understand how Kubernetes autoscaling affects application performance metrics in academic service environments.

Load Balancing in Containerized Applications

Load balancing plays an important role in distributed application architectures by distributing incoming requests among available application instances. In containerized environments, load balancing mechanisms help prevent individual instances from becoming overloaded and improve overall request handling capability. Kubernetes provides service-based load distribution mechanisms that allow traffic to be directed toward available pods.

Previous research has shown that load balancing strategies can improve scalability and reduce response delays in distributed systems. Effective traffic distribution enables applications to utilize available resources more efficiently, especially when multiple application instances operate simultaneously. However, the performance impact of load balancing depends on factors such as workload characteristics, routing mechanisms, application architecture, and available infrastructure resources.

In educational applications, workload distribution becomes increasingly important because user access patterns may vary depending on academic activities and institutional schedules. A student feedback application may experience sudden increases in requests during evaluation periods, requiring infrastructure capable of distributing workloads efficiently. Nevertheless, limited research has examined the combined impact of Kubernetes load balancing and autoscaling mechanisms specifically for educational applications. Kubernetes-based load balancing can improve workload distribution by directing requests toward available application instances (Nguyen, Phan, and Kim 2022) (Sujarwo, Istikmal, and Irawan 2023).

Related Studies and Research Gap

Several previous studies have explored Kubernetes implementation for improving application scalability and performance. Table 1 summarizes related research based on application domain, methodology, evaluation metrics, and limitations.

Table 1. Summary of Related Studies

Study	Research Focus	Method	Main Findings	Limitation
(Carrión 2022)	Kubernetes as a container orchestrator	Bibliometric analysis	Kubernetes has become a standard orchestration platform	Did not evaluate application performance experimentally
(Mondal et al. 2023)	Kubernetes workload prediction and autoscaling	Experimental modeling	Customizable autoscaling improves resource adaptation	Focused on general Kubernetes workloads
(Nguyen et al. 2022)	Kubernetes load balancing	Experimental evaluation	Resource-adaptive proxies improve workload distribution	Focused on edge computing infrastructure
(Gurohman et al. 2024)	Autoscaling for e-learning websites	Performance testing	Horizontal autoscaling improves web application performance	Did not compare with a student feedback application

Based on previous studies, Kubernetes has been widely investigated as a platform for cloud-native application management. However, several research gaps remain. First, previous research predominantly evaluates commercial cloud applications, benchmark workloads, or microservice environments rather than educational information systems. Second, studies often analyze individual Kubernetes mechanisms, such as autoscaling or resource optimization, while fewer studies evaluate the combined effect of autoscaling and load balancing on application performance. Third, empirical comparisons between conventional single-container deployment and Kubernetes-based deployment under controlled workload scenarios are still limited.

Therefore, this study addresses a practical evaluation gap by descriptively comparing a conventional single-container deployment with an integrated Kubernetes-based deployment under the tested workload conditions. The

comparison reports request-duration, request-rate, processor-utilization, and failed-request measurements, while explicitly limiting its causal claims to the complete deployment configurations observed in one workload run for each deployment scenario.

METHOD

Research Design

A controlled comparative case-study design was used to evaluate two complete deployment configurations. The baseline used one application container. The Kubernetes scenario enabled multiple pods, Kubernetes Service-based traffic distribution, Traefik routing, and Horizontal Pod Autoscaler configuration. This design directly evaluates the performance of the configuration described in the article title, but it does not separate the individual effect of autoscaling from the effect of pod replication, service routing, or application configuration. Deployment architecture was treated as the comparison factor, while virtual-machine-level processor utilization, interval-level p99 request duration, request processing rate, and failed-request rate were treated as measured outcomes. Each deployment scenario was executed through one independent workload run. Therefore, the reported values are descriptive and do not establish experimental repeatability or statistical significance. The research workflow is illustrated in Fig 1.

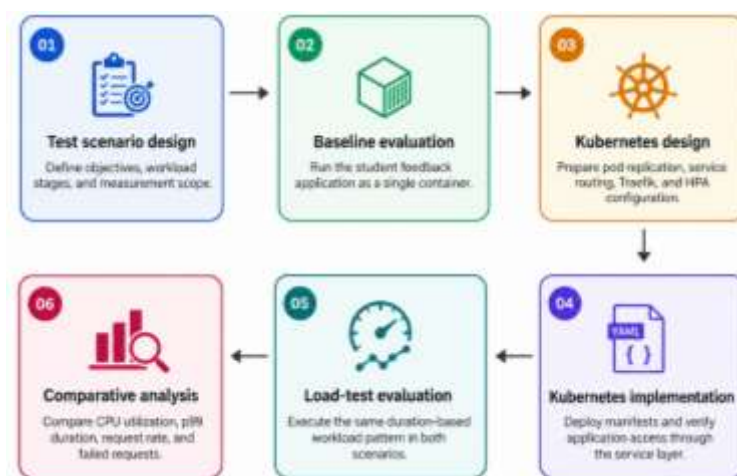


Fig 1. Research Workflow

The first stage defined the test objectives, workload pattern, and measurable indicators. The second stage evaluated the single-container baseline. The third and fourth stages designed and implemented the integrated Kubernetes-based architecture. The fifth stage evaluated the Kubernetes deployment with the same intended workload pattern. The final stage compared the recorded processor utilization, interval-level p99 request duration, request rate, and failed-request rate. Replica count over time and HPA event logs were not collected and are therefore not presented as measured outcomes.

Application and Runtime Specification

The evaluated system is JTIFORM, a student feedback application built with Laravel v12.0 running on PHP 8.4. The application is served by Laravel Octane v2.13 with FrankenPHP as the application server and worker runtime, so persistent workers handle requests instead of a per-request bootstrap process. The container image is built from the base image `dunglas/frankenphp:php8.4-alpine` and published as `ghcr.io/jtinnovation/jtiform:production`, built for the linux/amd64 platform through GitHub Actions. The number of Octane/FrankenPHP request workers is not fixed manually; it follows the default behaviour that derives worker count from the CPU cores visible to the container. Persistent data are stored in MySQL 8, and database connections are reused through the persistent Octane worker processes rather than an external connection-pool proxy. Asynchronous tasks are executed by dedicated Redis queue pods, each running a single worker process, for example:

```
php artisan queue:work redis --queue=generate-report-evaluation --sleep=2 --tries=3 --timeout=120 --max-jobs=500
```

Both deployment scenarios used the same application image, the same PHP and Laravel versions, the same MySQL 8 instance, and the same queue-worker configuration, so runtime differences between scenarios are limited to the orchestration layer.

Existing and Proposed System Architecture

The existing system used a conventional deployment architecture where the student feedback application was executed as a single application container. In this architecture, all incoming requests were processed by one application instance with fixed computational resources. This approach simplifies deployment but limits scalability because additional resources cannot be dynamically allocated when workload increases.

The proposed architecture implemented Kubernetes as a container orchestration platform. Multiple application pods were exposed through Kubernetes Service-based load distribution and Traefik routing, while Horizontal Pod Autoscaler configuration was enabled to permit dynamic replica adjustment. The retained HPA and resource parameters, together with the available Service, Traefik, and kube-proxy settings, are reported in the Kubernetes Deployment and Autoscaling Configuration subsection. The recorded performance dataset verifies VM-level CPU, interval-level p99, request rate, and failed-request rate, but it does not contain a time-synchronized replica-count trajectory, HPA-event log, or per-pod request-distribution measurements. Consequently, the performance results support the integrated deployment comparison, while direct verification of scaling events and traffic allocation remains a limitation.

The proposed architecture was designed to improve workload distribution and enable adaptive resource management during increasing user requests. The existing and Kubernetes-based deployment architectures are compared in Fig. 2.

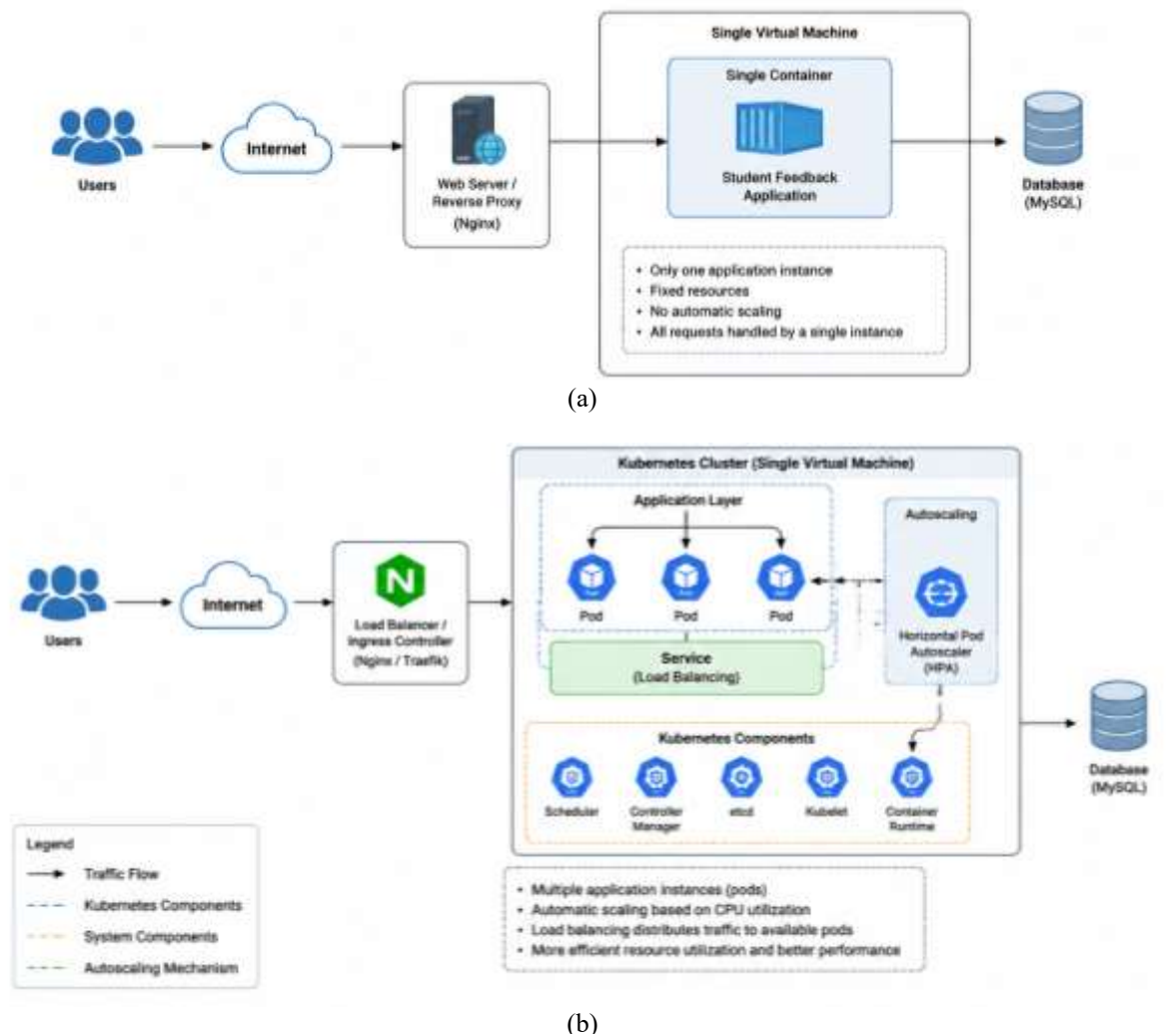


Fig. 2. Comparison of Deployment Architectures: (a) Existing Single-Container Deployment and (b) Proposed Kubernetes-Based Deployment

Experimental Environment

*Arvita Agus Kurniasari



This is anCreative Commons License This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.

The experiments were conducted on the same virtual machine to control the available host capacity. The infrastructure consisted of four virtual CPU cores, four gigabytes of memory, and forty gigabytes of solid-state storage. Operating-system, application, database, and network settings were intended to remain consistent between runs. Processor utilization was measured at the virtual-machine level as the sum of Busy System and Busy User values sampled at approximately 15-second intervals. Pod-level CPU, memory, replica count, and HPA events were not available in the recorded dataset; the retained host, runtime, resource-limit, autoscaling, ingress, and routing configuration is reported in Table 2 and the configuration subsections above.

The application tested in this study was JTIFORM, the student feedback application described in the Application and Runtime Specification subsection. Table 2 presents the host infrastructure and deployment scenarios; detailed application runtime, Kubernetes resource, autoscaling, ingress, and routing parameters are reported separately below to support reproducibility.

Table 2. Experimental Environment and Deployment Configuration.

Component	Specification
Application	JTIFORM student feedback application
Deployment scenario 1	Single-container deployment
Deployment scenario 2	Kubernetes-based deployment
Operating system	Ubuntu 24.04
Processor	4 virtual central processing unit cores
Memory	4 GB
Storage	40 GB SSD
Database	MySQL 8
Container platform	Docker and K3s (Kubernetes)

Kubernetes Deployment and Autoscaling Configuration

The Kubernetes scenario was deployed on a single-node K3s cluster in the jtiform namespace. Traffic entered the cluster through a standard Kubernetes Ingress handled by Traefik, which forwards requests to the application Service on port 8000; no custom Traefik middleware, sticky session, or weighted routing was configured. In-cluster load distribution therefore relied on the default ClusterIP Service behaviour with kube-proxy in iptables mode, which distributes new connections across ready pod endpoints. Table 3 and Table 4 report the resource and autoscaling parameters that were previously omitted.

Table 3. Resource Requests and Limits

Component	CPU request	CPU limit	Memory request	Memory limit
jtiform-app (application pod)	200m	500m	512Mi	1Gi
jtiform-queue- evaluation (queue pod)	100m	500m	128Mi	512Mi

Table 4. Horizontal Pod Autoscaler Configuration

Parameter	Value
API version / kind	autoscaling/v2, HorizontalPodAutoscaler
Scale target	Deployment jtiform-app
minReplicas	1
maxReplicas	10
Metric	Resource CPU, Utilization
Target average utilization	60% (of the 200m CPU request)
Scale-down stabilization window	Kubernetes default (300 s)
Scale-up stabilization window	Kubernetes default (0 s)
Metric evaluation interval	Kubernetes default HPA sync period (15 s)
Metrics source	metrics-server

The autoscaling target is expressed relative to the 200m CPU request, so the 60% target corresponds to approximately 120m CPU per application pod. The HPA synchronization period and stabilization behavior follow the Kubernetes defaults reported in Table 4. To avoid duplicating the configuration already summarized in Tables 3 and 4, the full YAML manifest is condensed into the operational pseudocode in Algorithm 1.

Algorithm 1. HPA-Based Scaling and Service Routing Procedure

*Arvita Agus Kurniasari



This is anCreative Commons License This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.

```
Input : CPU request C_req = 200m; target utilization U_t = 60%; R_min = 1; R_max = 10; HPA
sync period = 15 s
Output: Application replica count R and request distribution across Ready pods
1: Initialize jtiform-app with R <- 1
2: while the workload is active do
3:   Read the average CPU utilization of the application pods
4:   Evaluate the desired replica count using the Kubernetes HPA controller
5:   Bound R within [R_min, R_max] and apply the configured stabilization behavior
6:   Create or remove replicas according to the HPA decision
7:   Expose Ready pod endpoints through the Kubernetes Service
8:   Forward ingress traffic through Traefik to the Service
9:   Distribute new connections across Ready endpoints through kube-proxy
10: end while
Note: Exact HPA event timestamps, pod-readiness timestamps, and per-pod request counts were
not retained.
```

A spot check using `kubectl get pod -n jtiform` confirmed that the application pod and three Redis queue pods were Running with 1/1 Ready containers and zero reported restarts at the capture time. This snapshot verifies the observed state only at that moment and does not establish the absence of restarts or readiness transitions throughout the complete workload run.

HPA Implementation Result

Table 5 presents the actual HPA implementation result observed during the Kubernetes workload run. The HPA was configured with `minReplicas = 1`, `maxReplicas = 10`, CPU request = 200m, and target average CPU utilization = 60% (approximately 120m CPU per application pod). The replica ranges shown for each virtual-user band are observed deployment data from the implementation, not simulated values or an analytical reconstruction.

Table 5. HPA Implementation Result by Workload Band

VU band	Observed replica range	Implementation result
≤50	1–2 pods	The implementation operated with 1–2 application pods in this workload band.
51–100	2–3 pods	The observed replica range increased to 2–3 pods as workload increased.
101–200	3–6 pods	The implementation operated with 3–6 pods under the higher workload.
>200	6–8 pods	The implementation operated with 6–8 pods and remained below <code>maxReplicas = 10</code> .

The HPA sync period of approximately 15 seconds represents the controller evaluation cadence. Table 5 reports observed replica ranges by workload band; however, exact HPA event, pod-creation, image-pull, and readiness timestamps were not retained, so end-to-end scaling latency cannot be calculated reliably.

Load Testing Scenario

Load generation was performed with Grafana k6 using a duration-based virtual-user model with a maximum of 294 concurrent virtual users. The exact k6 version was not retained in the archived experiment material. The retained monitoring series was aggregated at approximately 15-second intervals, producing 57 intervals for the single-container run and 58 intervals for the Kubernetes-based run, corresponding to about 14.00 and 14.25 minutes of recorded monitoring. Both scenarios were driven by the same intended virtual-user stage pattern, the same target endpoints of the student feedback module, and the same application build; requests exercised read operations on the feedback form and write operations on feedback submission, while report generation was delegated to the Redis queue workers with a 120-second job timeout.

Because the k6 workload was duration based rather than iteration based, request throughput is an output variable rather than a controlled input: a faster deployment can complete more request-response cycles within the same intended virtual-user schedule. The difference between 41.86 and 100.10 requests per second is therefore consistent with different service capacity under that intended schedule and should not, by itself, be interpreted as evidence of an unequal offered workload.

The versioned k6 script and raw request-level output were not preserved; therefore, exact ramp timing, pacing, HTTP timeout, checks, authentication-token reuse, database size, and input-data seeding cannot be independently reconstructed. Workload equivalence is asserted only for the documented virtual-user pattern and endpoint set. Future experiments should archive the k6 script and version, raw request-level output, and exporter or aggregation configuration.

Table 6. Load Testing and Data-Collection Configuration

*Arvita Agus Kurniasari



This is anCreative Commons License This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.

Parameter	Description
Testing method	Duration-based virtual-user load testing
Testing tool	Grafana k6 (exact version not retained)
Workload pattern	Same intended time-based virtual-user stage pattern for both deployment scenarios
User simulation	Concurrent virtual users
Stage configuration	Time-based k6 stages; exact ramp-up/ramp-down timing not independently verifiable from the archived material
Stage duration	Recorded monitoring: ~14.00 min (single container) and ~14.25 min (Kubernetes); exact active-test duration not independently verifiable
Maximum virtual users	294 virtual users
Iteration model	Duration-based k6 virtual-user execution; completed request iterations depend on server response speed and script pacing
Total request execution	Not fixed as a controlled input; request rate is treated as a measured output
Testing repetition	One independent workload execution containing continuous request observations for each deployment scenario

Performance Metrics

The retained dataset contains interval-level virtual-user count, request throughput, failed-request throughput, and p99 request duration. VM-level processor utilization was calculated as Busy System + Busy User. The reported mean interval-level p99 is the arithmetic mean of one p99 value per approximately 15-second interval and is not equivalent to a global p99 from pooled request-level latencies.

Data Analysis

The collected measurements were analyzed descriptively. For each deployment run, the analysis reported the mean, median, and maximum of the interval-level CPU, p99, and request-rate series, together with the maximum failed-request rate. Observations collected within one time series were not treated as independent experimental replications because they are temporally related. No standard deviation across independent runs, confidence intervals, or hypothesis tests were calculated because only one workload execution was available for each deployment scenario.

The results describe associations observed under the tested application, workload pattern, and single-virtual-machine environment. They do not establish statistical significance, infrastructure-level high availability, or the isolated effect of autoscaling or load balancing. Because only one independent workload run was conducted for each scenario, stronger statistical claims require repeated independent executions; stronger causal claims additionally require an ablation design that separately evaluates a single container, one Kubernetes pod without HPA, fixed multiple pods with service-based load distribution, and multiple pods with HPA.

RESULT

Result Overview

Each deployment scenario was executed once under the same intended duration-based workload, reaching 294 virtual users. The retained series contains 57 intervals for the single-container run and 58 for the Kubernetes run. Because these observations are temporally related within each run, all reported statistics are descriptive and are not treated as independent experimental replications.

Single-Container Deployment Result

The single-container deployment recorded mean VM-level CPU utilization of 78.09%, mean interval-level p99 request duration of 9.272 seconds, and mean request rate of 41.86 requests/s. Median CPU utilization was 92.90%, maximum CPU utilization was 99.60%, median interval-level p99 was 2.240 seconds, maximum interval-level p99 was 38.500 seconds, and the failed-request rate reached 3.14 requests/s.

Kubernetes-Based Deployment Result

The Kubernetes-based deployment, which combined multiple application pods, Service-based distribution, Traefik routing, and HPA, recorded mean VM-level CPU utilization of 19.24%, mean interval-level p99 request duration of 0.383 seconds, and mean request rate of 100.10 requests/s. Median CPU utilization was 14.72%, maximum CPU utilization was 44.20%, median interval-level p99 was 0.172 seconds, maximum interval-level p99 was 2.640 seconds, and maximum request rate was 409.00 requests/s.

Performance Comparison Between Deployment Approaches

*Arvita Agus Kurniasari



This is anCreative Commons License This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.

Table 7 summarizes the overall descriptive comparison, while Tables 8 and 9 retain workload-band evidence for CPU utilization, interval-level p99, request rate, and failed requests. Figures 3 and 4 provide compact visual comparisons of CPU utilization and interval-level p99 across the same workload bands. The results represent the complete Kubernetes configuration and do not isolate the individual contribution of autoscaling or load balancing.

Table 7. Comparison of Experimental Results

Performance Metric	Single-Container Deployment	Kubernetes-Based Deployment	Relative Difference
Mean VM CPU utilization	78.09%	19.24%	75.36% lower
Mean interval-level p99 request duration	9.272 s	0.383 s	95.87% lower
Mean request rate	41.86 req/s	100.10 req/s	139.13% higher
Max failed-request rate	3.14 req/s	0.00 req/s	—
Independent workload executions	1 run	1 run	—

High-Workload Behavior

Above 200 virtual users, the Kubernetes run recorded mean VM-level CPU utilization of 39.53%, mean interval-level p99 of 0.188 seconds, and mean request rate of 255.10 requests/s. The single-container run recorded 99.22%, 23.605 seconds, and 45.32 requests/s, respectively. These values remain descriptive observations from one run per deployment.

Supporting Workload-Band Analysis

The retained interval data were grouped into four recorded virtual-user bands (≤ 50 , 51-100, 101-200, and >200). Sample counts differ slightly because monitoring timestamps were not perfectly aligned; the observations remain temporally correlated within each run.

Table 8. Workload-Band CPU Utilization and Interval-Level p99

VU band	n: single / K8s	Mean CPU single (%)	Mean CPU K8s (%)	Mean p99 single (s)	Mean p99 K8s (s)
≤ 50	29 / 30	58.70	10.45	4.011	0.580
51-100	6 / 6	94.40	16.57	7.967	0.164
101-200	11 / 12	99.21	25.65	9.520	0.161
>200	11 / 10	99.22	39.53	23.605	0.188

Table 9. Workload-Band Request Rate and Failed-Request Evidence

VU band	n: single / K8s	Mean rate single (req/s)	Mean rate K8s (req/s)	Max error single (req/s)	Max error K8s (req/s)
≤ 50	29 / 30	35.77	37.92	0.297	0.000
51-100	6 / 6	46.52	88.42	0.297	0.000
101-200	11 / 12	51.91	132.25	0.297	0.000
>200	11 / 10	45.32	255.10	3.140	0.000

Across the recorded runs, the complete Kubernetes configuration showed 75.36% lower mean VM-level CPU utilization, 95.87% lower mean interval-level p99 request duration, and 139.13% higher mean request rate than the single-container deployment. These percentages describe the observed configuration-level difference rather than an isolated causal effect of HPA or load balancing.

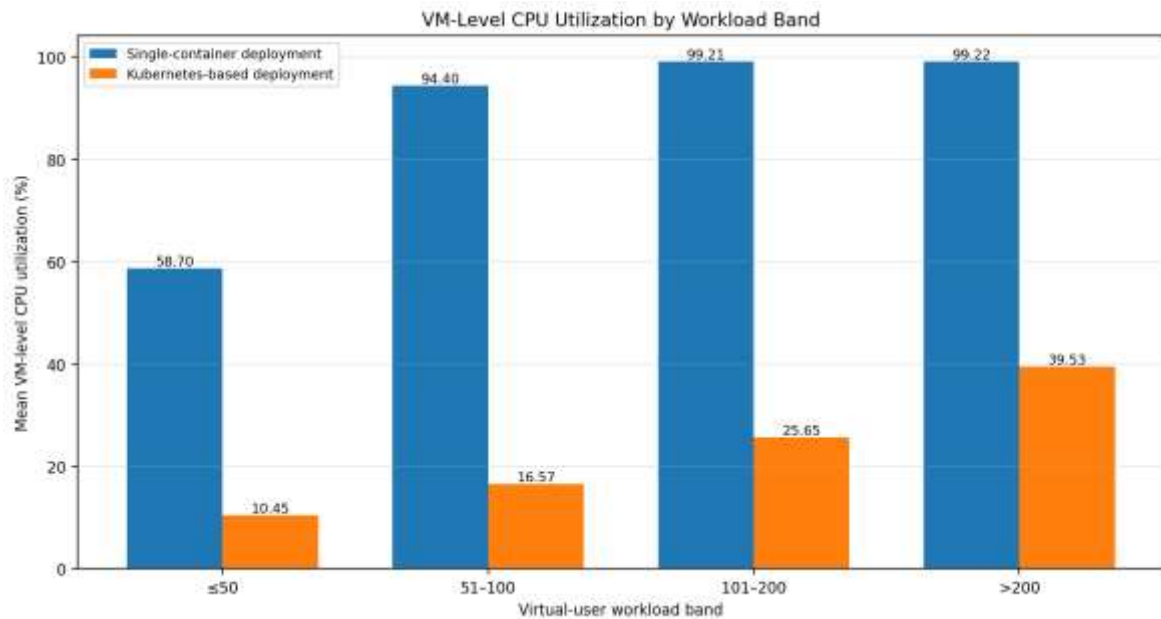


Fig. 3. Mean VM-Level CPU Utilization by Workload Band

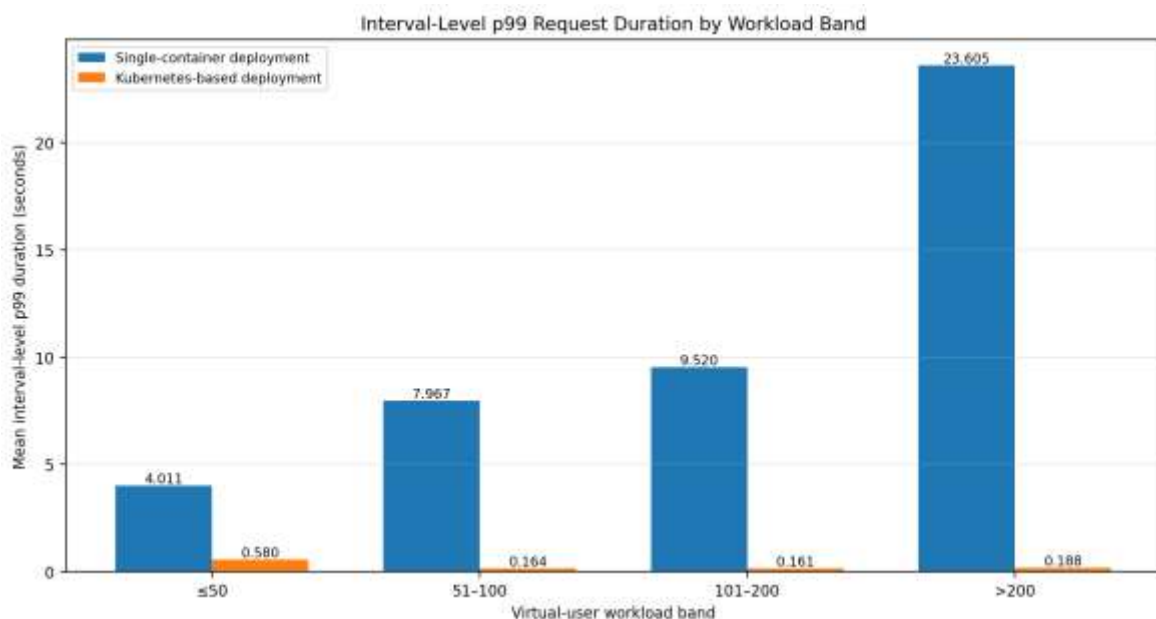


Fig. 4. Mean Interval-Level p99 Request Duration by Workload Band

DISCUSSIONS

Interpretation of Performance Results

The recorded workload run shows a consistent descriptive advantage for the complete Kubernetes configuration. At more than 200 virtual users, request rate was 255.10 versus 45.32 requests/s, mean interval-level p99 was 0.188 versus 23.605 seconds, and mean VM-level CPU utilization was 39.53% versus 99.22% for Kubernetes and the single-container deployment, respectively. Because pod replication, Service routing, Traefik, and HPA were enabled simultaneously, these differences cannot be attributed to one mechanism in isolation.

CPU was measured at the same VM level in both scenarios (Busy System + Busy User). The Kubernetes run combined lower host CPU utilization with higher request throughput, but the retained data do not identify a single explanation. Pod-level CPU, cgroup throttling, replica trajectories, HPA decisions, run-queue behavior, database response time, and per-pod request counts were not collected; therefore, the result is interpreted as a property of the integrated deployment rather than verified evidence of a specific scaling or load-balancing mechanism.

*Arvita Agus Kurniasari



This is anCreative Commons License This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.

The reported p99 values are interval-level percentiles sampled at approximately 15-second intervals. The lower >200-VU mean interval-level p99 for Kubernetes (0.188 s) than its whole-run mean (0.383 s) can occur because several larger early-interval values are excluded from the >200-VU subset. Fig. 5 shows these early higher values followed by comparatively low later intervals. This pattern is compatible with a transient warm-up or scaling phase, but the cause cannot be verified without request-level traces and time-synchronized HPA and replica telemetry. The absence of failed requests in the Kubernetes run indicates better request-processing stability under the tested workload, not proof of node-level high availability.

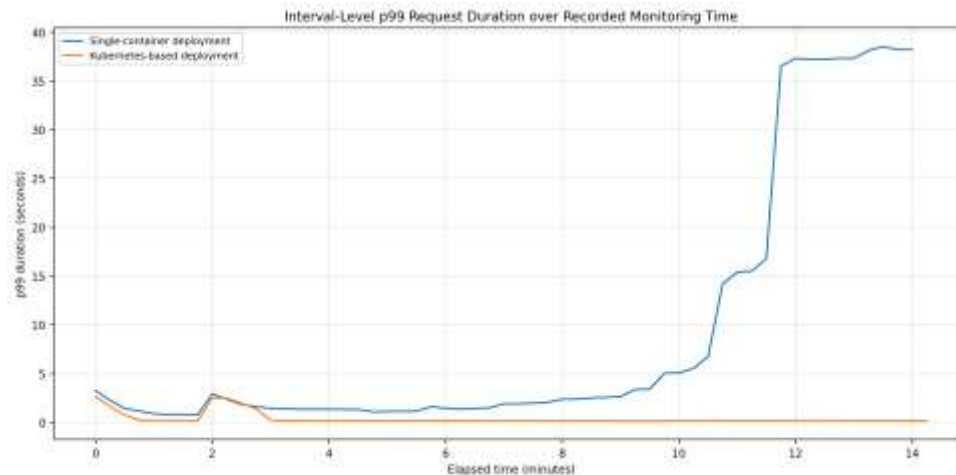


Fig. 5. Interval-Level p99 Request Duration over Recorded Monitoring Time

Comparison with Previous Studies

The findings are consistent with prior work showing that Kubernetes autoscaling and resource-aware traffic distribution can support workload adaptation (Nguyen et al. 2020)(Augustyn et al. 2024). Unlike studies that isolate a scaling policy, the present experiment evaluates a complete deployment architecture; therefore, the comparison concerns configuration-level performance rather than the independent effect of HPA.

The educational context also aligns with (Gurohman et al. 2024) and (Pradipta, Saskara, and Yudistira 2026), who reported Kubernetes-based performance improvements in e-learning and academic information systems. However, the present cluster remained on a single virtual machine, so stronger high-availability claims require node redundancy, failure-recovery testing, and continuous monitoring of scaling and application/database metrics.

Practical Implications

For student feedback systems with concentrated access near evaluation deadlines, multiple pods and Service-based distribution can increase application capacity compared with a single application instance. The benefit must be weighed against added operational requirements, including image management, resource requests and limits, autoscaling thresholds, ingress, monitoring, and cluster security; Kubernetes is most appropriate when workload variability and scalability requirements justify this complexity.

Limitations and Threats to Validity

The study has four principal limitations. First, both scenarios ran on one VM (4 vCPU, 4 GB RAM, 40 GB SSD), so node-level resilience and high availability were not evaluated. Second, autoscaling and load balancing were tested as one integrated configuration and their individual causal contributions were not isolated. Third, the retained telemetry did not include pod-level CPU/memory, replica timelines, HPA events, pod readiness, network/storage I/O, or database response metrics. Fourth, only one workload execution was available for each scenario, so between-run variability, confidence intervals, and significance testing cannot be estimated.

The controlled gradually increasing workload also may not represent abrupt production spikes, authentication bursts, unstable connections, or changing database operations. Accordingly, the findings should be generalized only to the tested application, workload pattern, and infrastructure.

CONCLUSION

This study descriptively compared a conventional single-container deployment with an integrated Kubernetes configuration for the JTIFORM student feedback application. In the recorded runs, Kubernetes showed 75.36% lower mean VM-level CPU utilization, 95.87% lower mean interval-level p99 request duration, and 139.13% higher mean request rate. No failed requests were recorded in the Kubernetes run, whereas the single-container run reached 3.14 failed requests/s.

These findings support the observed performance of the complete configuration in which pod replication, Service-based distribution, iptables-mode kube-proxy forwarding, Traefik Ingress, and HPA were enabled together. They do not establish statistical significance or isolate the causal effect of autoscaling and load balancing because each deployment was executed once and time-synchronized replica, HPA-event, and per-pod traffic data were not retained.

Future work should repeat each scenario independently and extend the experiment to a multi-node cluster to distinguish application-level scalability from node-level resilience. Database-side bottlenecks should be tested explicitly because horizontal pod growth can increase concurrent MySQL connections and query pressure, potentially shifting the bottleneck from the application layer to the database. A stronger design should compare a single container, one Kubernetes pod without HPA, fixed multiple pods with Service-based distribution, and multiple pods with HPA while synchronously recording replica count, HPA events, pod CPU/memory, pod readiness, request counts per pod, database connections, query latency, transaction throughput, and database CPU utilization.

ACKNOWLEDGMENT

The authors gratefully acknowledge the financial support provided through the PNPB Assignment Grant Program of the State Polytechnic of Jember under the assignment grant decree Number: 11575/DST/PL17/SK/AL.04.02/2026 issued by the Director of the State Polytechnic of Jember. The authors also express their appreciation to the Center for Research and Community Service (P3M) of the State Polytechnic of Jember for coordinating the research support and administrative assistance. Furthermore, the authors thank the Head of the Information Technology Department, department staff, the Software Engineering Laboratory, and the Engineering and Information Systems Laboratory for providing facilities and technical support during this research. Appreciation is also extended to the Information Technology students who participated in the application evaluation and contributed to the implementation and testing process.

REFERENCES

- Alharthi, Saleha, Afra Alshamsi, Anoud Alseiri, and Abdulmalik Alwarafy. 2024. "Auto-Scaling Techniques in Cloud Computing: Issues and Research Directions." *Sensors* 24(17):5551. doi: 10.3390/s24175551.
- Augustyn, Dariusz, Łukasz Wyciślik, and Mateusz Sojka. 2024. "Tuning a Kubernetes Horizontal Pod Autoscaler for Meeting Performance and Load Demands in Cloud Deployments." *Applied Sciences*. doi: 10.3390/app14020646.
- Burns, B., B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes. 2016. "Borg, Omega, and Kubernetes." *Communications of the ACM* 59(5):50–57. doi: 10.1145/2890784.
- Carrión, C. 2022. "Kubernetes as a Standard Container Orchestrator: A Bibliometric Analysis." *Journal of Grid Computing* 20(4):42. doi: 10.1007/s10723-022-09629-8.
- Dogani, J., R. Namvar, and F. Khunjush. 2023. "Auto-Scaling Techniques in Container-Based Cloud and Edge/Fog Computing: Taxonomy and Survey." *Computer Communications* 209:120–50. doi: 10.1016/j.comcom.2023.06.010.
- Gurohman, D. T., B. M. Susanto, A. Hariyanto, E. S. J. Atmadji, M. A. Gumilang, E. Antika, and N. A. Mukhlisoh. 2024. "Penerapan Horizontal Pod Autoscaler Dan Redis Cluster Berbasis Kubernetes Untuk Meningkatkan Performa Website E-Learning." *SKANIKA: Sistem Komputer Dan Teknik Informatika* 7(2):224–35. doi: 10.36080/skanika.v7i2.3211.
- Martínez Saucedo, A., G. Rodríguez, F. Gomes Rocha, and R. dos Santos. 2025. "Migration of Monolithic Systems to Microservices: A Systematic Mapping Study." *Information and Software Technology* 177:107590. doi: 10.1016/j.infsof.2024.107590.
- Mondal, Subrota Kumar, Xiaohai Wu, H. M. Dipu Kabir, Hong-Ning Dai, Kan Ni, Honggang Yuan, and Ting Wang. 2023. "Toward Optimal Load Prediction and Customizable Autoscaling Scheme for Kubernetes." *Mathematics*. doi: 10.3390/math11122675.



- Nguyen, Quang-Minh, Linh-An Phan, and Taehong Kim. 2022. "Load-Balancing of Kubernetes-Based Edge Computing Infrastructure Using Resource Adaptive Proxy." *Sensors*. doi: 10.3390/s22082869.
- Nguyen, Thanh-Tung, Yu-Jin Yeom, Taehong Kim, Dae-Heon Park, and Se-Han Kim. 2020. "Horizontal Pod Autoscaling in Kubernetes for Elastic Container Orchestration." *Sensors*. doi: 10.3390/s20164621.
- Pradipta, K. S., G. A. J. Saskara, and B. G. K. Yudistira. 2026. "Implementasi Kubernetes Cluster Untuk High Availability Dan Load Balancing SIAK Undiksha." *RIGGS: Journal of Artificial Intelligence and Digital Business* 4(4):12831–40. doi: 10.31004/riggs.v4i4.5846.
- Sujarwo, M. A. I. F. P., I. Istikmal, and A. I. Irawan. 2023. "Analysis of Load Balancing Least Connection and Shortest Expected Delay Algorithm for Web Server Using Kube-Proxy on Kubernetes." *ELKOMIKA: Jurnal Teknik Energi Elektrik, Teknik Telekomunikasi, \& Teknik Elektronika* 11(2):439. doi: 10.26760/elkomika.v11i2.439.