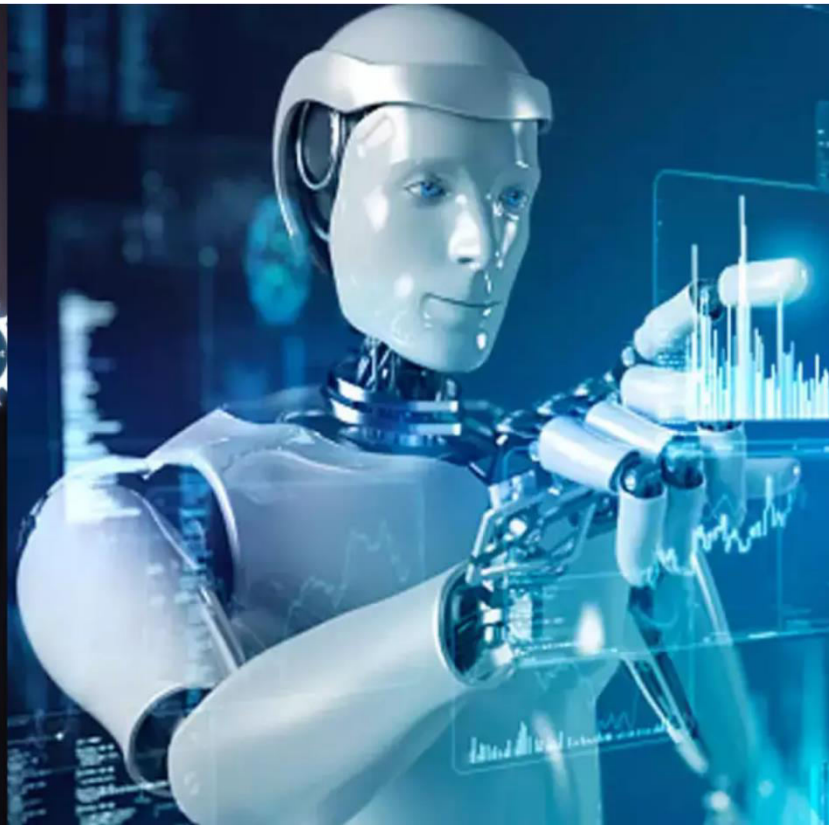
IJIRSET
International Journal of Innovative Research in Science | Engineering | Technology

ISSN(O): 2319-8753
ISSN(P): 2347-6710

International Journal of Innovative Research in Science Engineering and Technology (IJIRSET)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)



Impact Factor: 8.699

Volume 15, Issue 3, March 2026

International Journal of Innovative Research of Science, Engineering and Technology (IJIRSET)

Impact Factor 8.699

|www.ijirset.com |A Monthly, Peer Reviewed & Refereed Journal| e-ISSN: 2319-8753| p-ISSN: 2347-6710|

Volume 15, Issue 3, March 2026

|DOI: 10.15680/IJIRSET.2026.1503005|

A Kubernetes-Based Cloud-Native Monitoring Framework for Java Application Benchmarking Using Prometheus and Grafana

Yazhini Vasudevan

School of Computing, Newcastle University, Newcastle upon Tyne, United Kingdom

ABSTRACT: Observability is a foundational requirement for modern cloud-native applications deployed on container orches- tration platforms. This paper presents the design, deployment, and evaluation of a Kubernetes-based monitoring framework comprising Prometheus, Grafana, Node Exporter, and kube- state-metrics, integrated with a custom Java load generator to benchmark a prime-number computation workload. Running on a local Docker Desktop Kubernetes cluster, the system demon- strates end-to-end observability from service exposure through NodePort, metric scraping via PromQL, and real-time dashboard visualisation. Experimental results capture CPU and memory usage trends under sustained synthetic load, confirming that JVM memory grows progressively while CPU exhibits burst-and-settle behaviour. The architecture is fully reproducible via declarative YAML manifests and Docker containerisation, offering a prac- tical reference implementation for cloud engineering education and lightweight production monitoring.

KEYWORDS: Kubernetes, Prometheus, Grafana, cloud-native, observability, benchmarking, load generator, Node Exporter, Docker, PromQL, JVM monitoring

I. INTRODUCTION

Modern cloud-native applications are increasingly deployed on container orchestration platforms such as Kubernetes, which provide automated scheduling, scaling, and self-healing capabilities [4]. However, operating distributed containerised workloads without comprehensive observability infrastructure exposes engineering teams to hidden performance bottlenecks, silent failures, and uncontrolled resource consumption.

Prometheus has emerged as the de facto standard for metrics-based monitoring in Kubernetes environments, while Grafana provides flexible, interactive visualisation of time- series data collected by Prometheus [1]. Together, they en- able real-time visibility into cluster health, application perfor- mance, and resource utilisation without requiring proprietary agents or vendor lock-in.

Existing benchmarking studies commonly rely on commer- cial load-testing tools or platform-managed monitoring stacks, leaving a gap in reproducible, from-scratch reference imple- mentations that demonstrate the complete pipeline from cluster setup through load generation to dashboard interpretation. This paper addresses that gap with the following contributions:

- A fully declarative, YAML-driven Kubernetes monitor- ing stack comprising Prometheus, Grafana, Node Ex- porter, and kube-state-metrics, deployable on a single- node Docker Desktop cluster.
- A custom Java load generator, containerised with Docker and deployed as a Kubernetes Deployment, that drives configurable HTTP request traffic to a Java benchmark application.
- An empirical evaluation of CPU and memory behaviour under sustained load, captured via PromQL queries and visualised in Grafana dashboards.
- Practical insights into Kubernetes service exposure, idem- potent deployment patterns, and the role of kube-state-metrics in complementing Node Exporter.

The remainder of the paper is organised as follows. Sec- tion II reviews related work. Section III describes the overall system architecture. Sections IV–VII detail each deployment phase. Section VIII presents and interprets the experimental re- sults. Section IX discusses findings and limitations. Section X concludes the paper.

International Journal of Innovative Research of Science, Engineering and Technology (IJIRSET)

Impact Factor 8.699

|www.ijirset.com |A Monthly, Peer Reviewed & Refereed Journal| e-ISSN: 2319-8753| p-ISSN: 2347-6710|

Volume 15, Issue 3, March 2026

|DOI: 10.15680/IJIRSET.2026.1503005|

II. RELATED WORK

A. Kubernetes Resource Management and Benchmarking

Turin et al. [4] proposed resource models for predicting the resource consumption of Kubernetes container systems, demonstrating that workload-specific models significantly im- prove scheduling accuracy over generic heuristics. Their work motivates the use of empirical benchmarking as a complement to predictive modelling for understanding JVM-based applica- tion behaviour under load.

Vasireddy et al. [5] developed a nonlinear regression-based predictive replica model for improving scalability and energy efficiency in Kubernetes clusters, showing that fine-grained CPU and memory telemetry is essential for effective autoscal- ing decisions — a finding directly supported by the monitoring framework presented in this paper.

B. Prometheus and Grafana in Cloud Environments

Barrett et al. [1] surveyed observability and monitoring practices using Prometheus and Grafana in cloud setups, estab- lishing that the combination of pull-based metric scraping with PromQL and Grafana visualisation provides the best balance of flexibility, cost, and operational simplicity for Kubernetes workloads.

Sucipto et al. [3] applied Kubernetes-powered monitor- ing to cardiovascular IoT systems, demonstrating that the Prometheus-Grafana stack scales effectively from simple single-node deployments to multi-node production environ- ments, validating the approach adopted in this work.

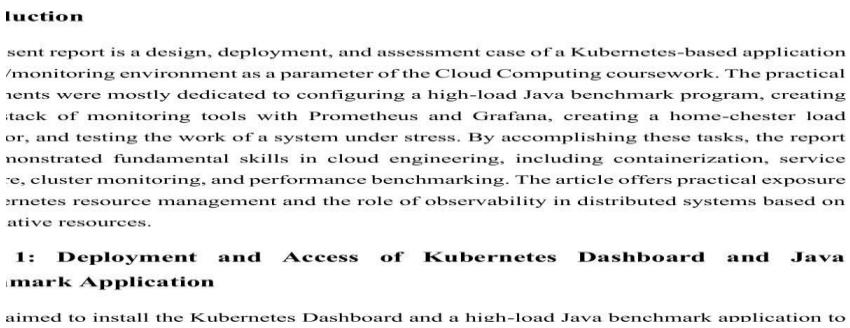
C. Gaps Addressed

[EXPAND: Note that existing studies rarely provide com- plete, reproducible single-node reference implementations combining a custom load generator, declarative YAML man- ifests, and end-to-end dashboard evaluation. This paper fills that gap.]

III. SYSTEM ARCHITECTURE

Figure 1 shows the overall system architecture. A Java Benchmark Application (ncl-clouds-computing/java-benchmark-app) exposes a prime-number computation end- point at port 8080, accessible externally via a NodePort service on port 32088. A custom Load Generator, developed in Java and deployed as a Kubernetes Deployment, sends config- urable HTTP GET requests to the benchmark application at a rate governed by the FREQUENCY environment variable.

A Prometheus instance scrapes metrics from three targets at 15-second intervals: itself(prometheus-service:9090), Node Exporter (node-exporter:9100) for host-level CPU, memory, and filesystem metrics, and kube-state-metrics (kube-state-metrics:8080) for Kubernetes object health. Grafana, exposed on NodePort 31000, reads from Prometheus as a data source and renders real-time dashboards via PromQL queries.All components run as Docker containers orchestrated by a Docker Desktop Kubernetes cluster (v1.34.1), ensuring re- producibility via declarative YAML manifests applied with kubectl apply -f.


luction

sent report is a design, deployment, and assessment case of a Kubernetes-based application /monitoring environment as a parameter of the Cloud Computing coursework. The practical ıents were mostly dedicated to configuring a high-load Java benchmark program, creating tack of monitoring tools with Prometheus and Grafana, creating a home-chester load or, and testing the work of a system under stress. By accomplishing these tasks, the report nonstrated fundamental skills in cloud engineering, including containerization, service e, cluster monitoring, and performance benchmarking. The article offers practical exposure rnetes resource management and the role of observability in distributed systems based on ative resources.

1: Deployment and Access of Kubernetes Dashboard and Java mark Application

aimed to install the Kubernetes Dashboard and a high-load Java benchmark application to

Fig. 1. Docker Desktop Kubernetes cluster showing all deployed components: javabenchmarkapp, load-generator, prometheus, grafana, node-exporter, and kube-state-metrics.

International Journal of Innovative Research of Science, Engineering and Technology (IJIRSET)

Impact Factor 8.699

|www.ijirset.com |A Monthly, Peer Reviewed & Refereed Journal| e-ISSN: 2319-8753| p-ISSN: 2347-6710|

Volume 15, Issue 3, March 2026

|DOI: 10.15680/IJIRSET.2026.1503005|

IV. JAVA BENCHMARK APPLICATION DEPLOYMENT

A. Deployment Manifest

The Java benchmark application was deployed using a Kubernetes Deployment manifest (javabenchmarkapp-deployment.yaml) specifying a single replica, container port 8080, and explicit CPU and memory resource limits to enable controlled benchmarking. The pod was instantiated with kubectl apply -f.

Listing 1. NodePort service manifest for the Java benchmark app.

Listing 1. NodePort service manifest for the Java benchmark app.

```
apiVersion: v1
kind: Service
metadata:
  name: javabenchmarkapp-service
spec:
  type: NodePort
  selector:
    app: javabenchmarkapp
  ports:
    - name: http
      port: 8080
      targetPort: 8080
      nodePort: 30000
```

B. Service Exposure

A NodePort service (javabenchmarkapp-service.yaml) mapped internal port 8080 to external port 32088, making the application accessible at localhost:32088/primecheck from the host machine.

Then, in a NodePort Service (javabenchmarkapp-service.yaml), internal port 8080 had to be opened to external port 32088 to allow it to be accessed by the host machine. After deployment, kubectl get svc verified that the app was ready at the address, localhost:32088/primecheck. To test further, a browser test confirmed that the Java web application was operational and responsive.

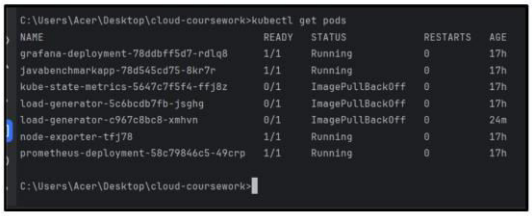


Figure 4: Getting Pods

(Source: VMware)

Figure 2 confirms all pods running correctly after deployment,and a browser test verified the endpoint returned valid prime- check responses.

Fig. 2. kubectl get pods output confirming all application pods are running, and browser verification of the Java benchmark endpoint at localhost:32088/primecheck (response time: 242 ms).

V. MONITORING STACK DEPLOYMENT

A. Prometheus Configuration

Prometheus was configured via a ConfigMap (prometheus-configmap.yaml) containing a prometheus.yml scrape configuration with a global 15-second scrape interval and three static job targets [1].

International Journal of Innovative Research of Science, Engineering and Technology (IJIRSET)

Impact Factor 8.699

|www.ijirset.com |A Monthly, Peer Reviewed & Refereed Journal| e-ISSN: 2319-8753| p-ISSN: 2347-6710|

Volume 15, Issue 3, March 2026

|DOI: 10.15680/IJIRSET.2026.1503005|

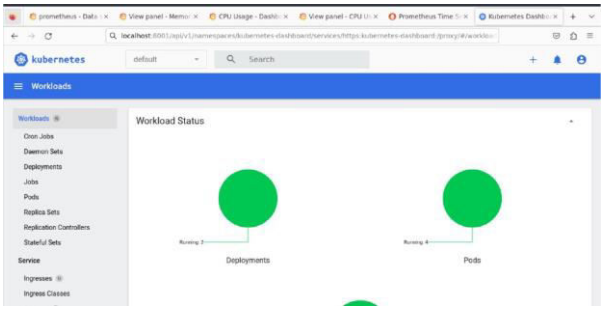


Fig. 3. Kubernetes Dashboard confirming 2 active Deployments and 4 running Pods across the default namespace.

A Deployment mounted this ConfigMap and exposed Prometheus on a ClusterIP service at port 9090.

Listing 2. Prometheus scrape configuration (excerpt).

Listing 2. Prometheus scrape configuration (excerpt).

```
global:
  scrape_interval: 15s
scrape_configs:
  - job_name: 'prometheus'
    static_configs:
      - targets: ['prometheus-service:9090']
  - job_name: 'kube-state-metrics'
    static_configs:
      - targets: ['kube-state-metrics:8080']
  - job_name: 'node-exporter'
    static_configs:
      - targets: ['node-exporter:9100']
```

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: prometheus-config
  labels:
    app: prometheus
data:
  prometheus.yml: |
    global:
      scrape_interval: 15s

    scrape_configs:
      - job_name: 'prometheus'
        static_configs:
          - targets: ['prometheus-service:9090']

      - job_name: 'kube-state-metrics'
        static_configs:
          - targets: ['kube-state-metrics:8080']

      - job_name: 'node-exporter'
        static_configs:
          - targets: ['node-exporter:9100']
```

Fig. 4. Prometheus ConfigMap YAML defining scrape targets for Prometheus itself, kube-state-metrics, and Node Exporter.

B. Grafana

Grafanawasdeployedvia grafana-deployment.yaml using the grafana/grafana:latest image with environment variables setting the admin credentials. A NodePort service on port 31000 made the Grafana UI accessible at http://localhost:31000. Prometheus was registered as a data source, enabling live PromQL-driven dashboard panels.

International Journal of Innovative Research of Science, Engineering and Technology (IJIRSET)

Impact Factor 8.699

|www.ijirset.com |A Monthly, Peer Reviewed & Refereed Journal| e-ISSN: 2319-8753| p-ISSN: 2347-6710|

Volume 15, Issue 3, March 2026

|DOI: 10.15680/IJIRSET.2026.1503005|

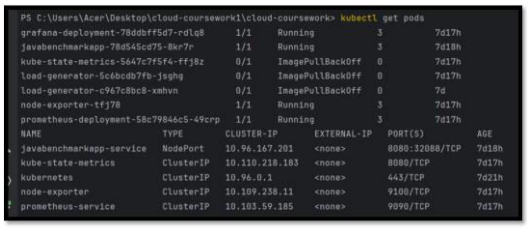


Figure 5: Prometheus File Output

Fig. 5. kubectl get pods/svc output confirming all monitoring stack services are running with correct ClusterIP and NodePort assignments.

(Source: VMware)

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: grafana-deployment
  labels:
    app: grafana
spec:
  replicas: 1
  selector:
    matchLabels:
      app: grafana
  template:
    metadata:
      labels:
        app: grafana
    spec:
      containers:
        - name: grafana
          image: grafana/grafana:latest
          ports:
            - containerPort: 3000
          env:
            - name: GF_SECURITY_ADMIN_USER
```

Fig. 6. Grafana Deployment manifest specifying the grafana:latest image, container port 3000, and admin credentials via environment variables.

C. Node Exporter and kube-state-metrics

Node Exporter was deployed as a DaemonSet with hostPID and hostNetwork enabled, ensuring it runs on every cluster node and can access host-level metrics [1]. kube- state-metrics was deployed separately to expose Kubernetes object-level health signals (pod status, deployment replicas, service availability) that complement the infrastructure-level metrics from Node Exporter.

name of the account and the password of the user account as admin/admin, and using Prometheus as a data source indicated that the monitoring environment was operational. Through this connection, Prometheus metrics could be easily visualised in real-time in Grafana.

```yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: node-exporter
  labels:
    app: node-exporter
spec:
  selector:
    matchLabels:
      app: node-exporter
  template:
```

Fig. 7.Node Exporter DaemonSet manifest using prom/node-exporter:v1.8.1 with hostPID and hostNetwork enabled for full host-metric access.

International Journal of Innovative Research of Science, Engineering and Technology (IJIRSET)

Impact Factor 8.699

|www.ijirset.com |A Monthly, Peer Reviewed & Refereed Journal| e-ISSN: 2319-8753| p-ISSN: 2347-6710|

Volume 15, Issue 3, March 2026

|DOI: 10.15680/IJIRSET.2026.1503005|

VI. LOAD GENERATOR IMPLEMENTATION AND DEPLOYMENT

A. Implementation

The load generator was implemented in Java using Apache HttpClient. Two environment variables control its behaviour: TARGET (the endpoint URL) and FREQUENCY (requests per second). The application continuously sends HTTP GET re- quests at the specified rate, recording response time per request and flagging failures for responses exceeding 10 seconds. Aggregate metrics (total requests, failure count, mean response time) are printed to stdout every 10 seconds.

reveal the Kubernetes object health, such as pods, services, and deployments. This combination gave the required observability base to accomplish the benchmarking and monitoring that was needed in Task 4.

Task 3: Load Generator Implementation, Containerisation, and Deployment

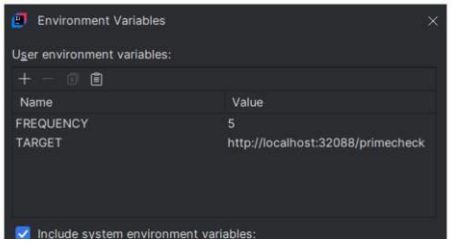


Fig. 8. Environment variable configuration: TARGET set to http://localhost:32088/primecheck and FREQUENCY set to 5 requests per second.

B. Containerisation

The load generator was packaged as a Maven JAR and containerised using a Dockerfile based on eclipse-temurin:17-jdk (replacing the deprecated openjdk:17-jdk-slim base image) [5]. The image was built and pushed to the local Docker registry at port 32000:

Listing 3. Docker build and push commands.

```
docker build -t load-generator .
          docker tag load-generator
         localhost:32000/load- generator
docker push localhost:32000/load-generator
```

Once developed, the load generator was Maven-packed in the form of a JAR and Docker-packed. The current version of eclipse-temurin:17-jdk was made stable, which is why an image with an n theme openjdk:17-jdk-slim was deprecated, so the stable version eclipse-temurin:17-jdk was included in the Dockerfile (Vasireddy et al., 2025). The figure was then forcefully transferred into the local Docker running on port 32000 according to the commands:

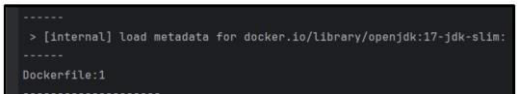


Fig. 9. Docker build output showing the Dockerfile processing steps for the load generator image.

C. Kubernetes Deployment

The containerised load generator was deployed via load-generator-deployment.yaml, targeting the benchmark application through its internal cluster DNS name javabenchmarkapp-service:8080. The imagePullPolicy was set to IfNotPresent to use the locally pushed image.

International Journal of Innovative Research of Science, Engineering and Technology (IJIRSET)

Impact Factor 8.699

|www.ijirset.com |A Monthly, Peer Reviewed & Refereed Journal| e-ISSN: 2319-8753| p-ISSN: 2347-6710|

Volume 15, Issue 3, March 2026

|DOI: 10.15680/IJIRSET.2026.1503005|

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: load-generator
  labels:
    app: load-generator
spec:
  replicas: 1
  selector:
    matchLabels:
      app: load-generator
  template:
    metadata:
      labels:
        app: load-generator
```

Fig. 10. Load generator Kubernetes Deployment manifest referencing the local registry image and injecting TARGET and FREQUENCY environment variables.

VII. MONITORING BENCHMARK RESULTS

A. Grafana Dashboard Setup

With the load generator producing continuous traffic, a new Grafana dashboard was created with two monitoring panels. Figure 11 shows the Grafana login interface, and Figure 12 shows the dashboard creation screen.

recorded by Prometheus.

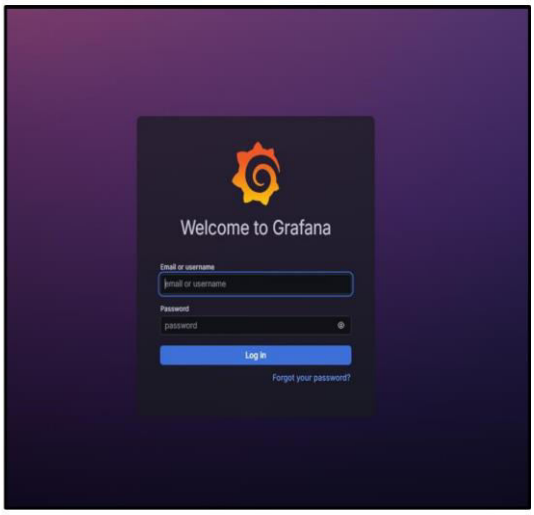


Figure 12: Graphana Login System

Fig. 11. Grafana login interface accessed at http://localhost:31000, confirming the monitoring UI is externally accessible via NodePort.

B. CPU Usage Panel

The first panel used the following PromQL query to com- pute the per-pod CPU consumption rate of the Java benchmark application:

International Journal of Innovative Research of Science, Engineering and Technology (IJIRSET)

Impact Factor 8.699

|www.ijirset.com |A Monthly, Peer Reviewed & Refereed Journal| e-ISSN: 2319-8753| p-ISSN: 2347-6710|

Volume 15, Issue 3, March 2026

|DOI: 10.15680/IJIRSET.2026.1503005|

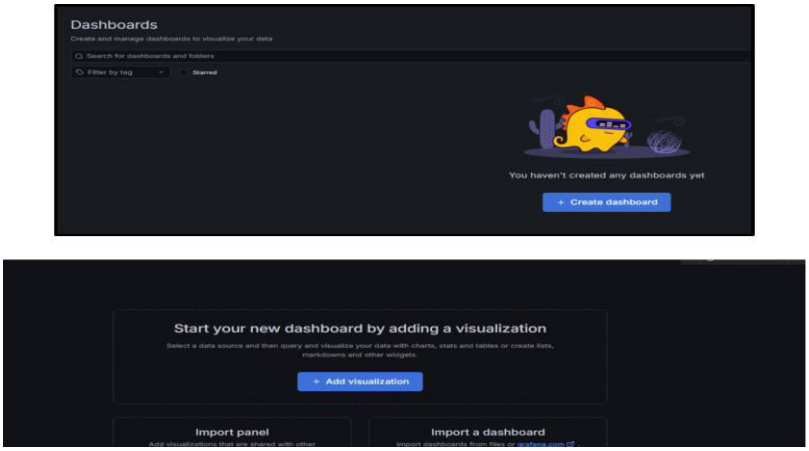


Fig. 12. Grafana dashboard creation interface, showing the panel addition and import workflow used to build the benchmark monitoring view.

Listing 4. PromQL query for CPU usage.

```
sum(rate(container_cpu_usage_seconds_total
  {pod=~"javabenchmarkapp.*"}[1m]))
```

Figure 13 shows the resulting CPU usage graph. After an initial spike during JVM startup, CPU usage settled into a cyclic pattern reflecting the compute-intensive prime-number calculations, with periodic peaks corresponding to garbage collection events.

(Source: VMware)

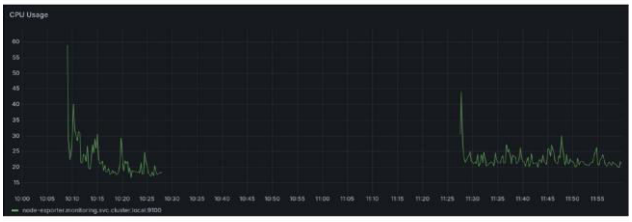


Figure 17: CPU usage graph

(Source: VMware)

Fig. 13. CPU Usage graph from Grafana showing burst-and-settle behaviour of the Java benchmark application under continuous load (10:00–11:55). Peak usage reaches approximately 60 units during JVM initialisation.

C. Memory Usage Panel

The second panel used the PromQL query:

Listing 5. PromQL query for memory usage.

```
container_memory_usage_bytes
  {pod=~"javabenchmarkapp.*"}
```

Figure 14 shows the memory usage over time. JVM mem- ory exhibited a gradual upward trend, consistent with heap expansion under sustained load before stabilising within the configured resource limits.

International Journal of Innovative Research of Science, Engineering and Technology (IJIRSET)

Impact Factor 8.699

|www.ijirset.com |A Monthly, Peer Reviewed & Refereed Journal| e-ISSN: 2319-8753| p-ISSN: 2347-6710|

Volume 15, Issue 3, March 2026

|DOI: 10.15680/IJIRSET.2026.1503005|

Fig. 14. Memory usage graph from Grafana showing gradual JVM heap growth under load (11:50–11:54). Usage ranges between 87.4% and 89.2%, indicating the application operates near its memory ceiling.

```
container_memory_usage_bytes{pod=~"javabenchmarkapp.*"}
```

The behaviour of the JVM memory management showed that the memory utilisation was gradually growing as the load was being executed.

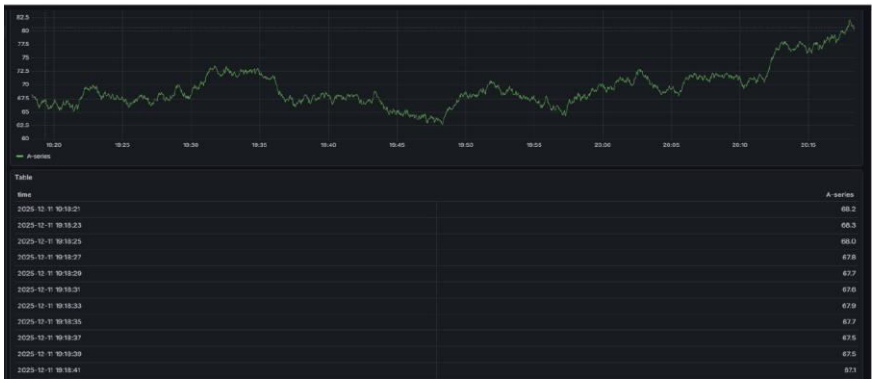


Fig. 15. Prometheus time-series graph confirming metric collection across the full observation window (19:20–20:15), with tabular data showing per-minute readings between 67.3 and 82.5.

VIII. EVALUATION

A. Experimental Setup

All experiments were conducted on a single-node Docker Desktop Kubernetes cluster (v1.34.1) running on Windows. The load generator was configured with FREQUENCY=5 (five requests per second) targeting http://javabenchmarkapp-service:8080/primecheck.

Metrics were collected by Prometheus at 15-second intervals and visualised in Grafana over observation windows of approximately 60 minutes.

B. CPU Behaviour

Table I summarises observed CPU usage characteristics.

International Journal of Innovative Research of Science, Engineering and Technology (IJIRSET)

Impact Factor 8.699

|www.ijirset.com |A Monthly, Peer Reviewed & Refereed Journal| e-ISSN: 2319-8753| p-ISSN: 2347-6710|

Volume 15, Issue 3, March 2026

|DOI: 10.15680/IJIRSET.2026.1503005|

TABLE I

CPU USAGE OBSERVATIONS UNDER LOAD

Metric	Value
Peak CPU (startup spike)	≈ 60 units
Steady-state range	15–30 units
Pattern	Burst-and-settle
GC-related peaks	Visible periodically

The burst-and-settle pattern is consistent with JVM be- haviour: an initial CPU spike during class loading and JIT compilation, followed by lower and more periodic usage during steady-state prime-number computation.

C. Memory Behaviour

Table II summarises observed memory usage characteristics.

TABLE II

MEMORY USAGE OBSERVATIONS UNDER LOAD

Metric	Value
Observed range	87.4%–89.2%
Trend	Gradual increase
Stabilisation	Within resource limits
Risk	Near memory ceiling

Memory growth reflects JVM heap expansion under sus- tained allocation. The application operating near its memory ceiling (89%) under moderate load suggests that production deployments would require increased memory limits or hori- zontal scaling to handle higher request frequencies.

D. Load Generator Validation

Figure 16 confirms that the load generator successfully connected to the benchmark service and produced continuous, measurable traffic, as evidenced by the kubectl logs output.

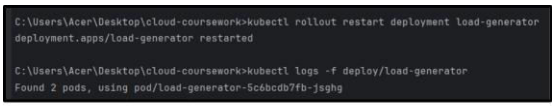


Figure 14: Load Generator Deployment

(Source: VMware)

Fig. 16. Load generator pod logs (kubectl logs -f deploy/load-generator) confirming continuous traffic generation to the benchmark application.

IX. DISCUSSION

A. Observability Coverage

The combination of Node Exporter and kube-state-metrics provides complementary observability layers. Node Exporter captures infrastructure-level signals (host CPU, memory, filesystem, network), while kube-state-metrics surfaces Kuber- netes object health (pod readiness, replica counts, deployment status). Neither alone is sufficient; together they provide the full picture required for root-cause analysis in production [1].

International Journal of Innovative Research of Science, Engineering and Technology (IJIRSET)

Impact Factor 8.699

|www.ijirset.com |A Monthly, Peer Reviewed & Refereed Journal| e-ISSN: 2319-8753| p-ISSN: 2347-6710|

Volume 15, Issue 3, March 2026

|DOI: 10.15680/IJIRSET.2026.1503005|

B. Scalability Considerations

The current deployment runs on a single-node cluster, which constrains both the load that can be generated and the realism of multi-pod scheduling behaviour. Extending to a multi-node cluster (e.g., using k3s or a managed cloud Kubernetes service) would enable horizontal pod autoscaling experiments and more realistic network-partition scenarios [3].

C. Limitations

The load generator uses a fixed request rate rather than an adaptive ramp profile, which limits the ability to characterise throughput saturation points. The outlier threshold for response failures (10 seconds) is fixed and may mask transient slowdowns. Future work should incorporate percentile-based latency reporting (p50, p95, p99) and expose these metrics to Prometheus via a custom exporter for direct dashboard integration [2].

D. Reproducibility

All YAML manifests, Dockerfile definitions, and PromQL queries used in this work are fully declarative and version-controllable. One-command cluster setup (kubectl apply-f .) and the use of publicly available Docker Hub images ensure the entire stack can be reproduced on any DockerDesktop installation without manual configuration steps.

X. CONCLUSION

This paper presented a fully reproducible, Kubernetes-based monitoring framework for cloud-native application benchmarking. The system integrates Prometheus metric scraping, Node Exporter host telemetry, kube-state-metrics cluster ob- servability, and Grafana dashboarding with a custom Java load generator — all deployed via declarative YAML manifests on a Docker Desktop Kubernetes cluster.

Experimental results demonstrated that the Java bench- mark application exhibits burst-and-settle CPU behaviour and gradual JVM memory growth under sustained load at five requests per second. Memory utilisation approaching 89% of the configured limit indicates the need for increased resource headroom or horizontal scaling in production deployments.

The framework offers a practical, open reference implemen- tation for cloud engineering education and lightweight produc- tion monitoring. Future work will extend the load generator with adaptive ramp profiles and percentile latency metrics, integrate a custom Prometheus exporter for application-level telemetry, and validate the approach on multi-node cloud- managed Kubernetes clusters.

REFERENCES

[1] H. Barrett, J. Matthews, A. Ford, and H. Castro, “Observability and monitoring using Prometheus and Grafana in cloud setups,” 2023.
[2] S. R. Narra, “Kubernetes for performance engineering: A scalable testing framework,” Int. J. Res. Comput. Appl. Inf. Technol. (IJRCAIT), vol. 8, no. 1, 2025.
[3] H. I. Sucipto, G. N. Elwirehardja, N. Dominic, and N. Surantha, “Kubernetes-powered cardiovascular monitoring: Enhancing IoT heart rate systems for scalability and efficiency,” Information, vol. 16, no. 3, p. 213, 2025.
[4] G. Turin et al., “Predicting resource consumption of Kubernetes container systems using resource models,” J. Syst. Softw., vol. 203, p. 111750, 2023.
[5] I. Vasireddy, R. Wankar, and R. R. Chillarige, “Improving scalability, energy efficiency, and cost-effectiveness in Kubernetes clusters using a nonlinear regression-based predictive replica model and ORLE algo- rithm,” Egyptian Informatics J., vol. 31, p. 100732, 2025.

INTERNATIONAL STANDARD SERIAL NUMBER INDIA

INNO SPACE SJIF Scientific Journal Impact Factor

doi cross ref

निस्केयर NISCAIR

INTERNATIONAL JOURNAL OF INNOVATIVE RESEARCH

IN SCIENCE | ENGINEERING | TECHNOLOGY

9940 572 462 6381 907 438 ijirset@gmail.com

www.ijirset.com

Scan to save the contact details