

Cross-Resource Risk Scoring and a Composite Health Index for Self-Hosted Kubernetes Monitoring on Prometheus

1st. Namrata Khade

Assistant Professor at
Computer Science and Engineering
Priyadarshini College of Engineering
Nagpur, India

2nd Anjali Yadav

Computer Science and Engineering
Priyadarshini College of Engineering
Nagpur, India

3rd Apurva Rakshak

Computer Science and Engineering
Priyadarshini College of Engineering
Nagpur, India

4th Avanti Nandanwar

Computer Science and Engineering
Priyadarshini College of Engineering
Nagpur, India

5th Shital Tiwari

Computer Science and Engineering
Priyadarshini College of Engineering
Nagpur, India

6th Nisarga Hadke

Computer Science and Engineering
Priyadarshini College of Engineering
Nagpur, India

7th Riya Shahu

Computer Science and Engineering
Priyadarshini College of Engineering
Nagpur, India

Abstract—Static per-metric thresholds are the default way to alert on Kubernetes resources, but they treat Nodes and Pods as independent and say nothing about overall cluster condition. This paper presents a self-hosted monitoring system built on Prometheus that addresses both gaps with two lightweight, interpretable mechanisms: (i) a cross-resource Pod risk score that combines a Pod's restart count, its CPU and memory use, and the severity of the Node hosting it, and (ii) a composite Cluster Health Score (0–100) that weights Node health, Pod health, and resource utilization at 0.40, 0.35, and 0.25. Alerts follow an explicit lifecycle: each is raised once per resource, escalated in place as severity increases, and cleared automatically on recovery. The system was evaluated on a Minikube cluster with 31 Pods under four controlled scenarios: steady state, Node memory pressure, a Pod crash loop, and recovery. It raised no alerts in steady state, escalated the memory-pressure alert from warning to critical, raised the risk scores of Pods on the stressed Node, classified the crash-looping Pod as high risk, and cleared every alert on recovery. The evaluation is functional rather than a large-scale performance study, and its limits are stated explicitly. Because the scoring is rule-based, each alert can be traced to its inputs, which makes the design a transparent baseline for learned anomaly detectors.

Keywords — *Kubernetes; Prometheus; cluster monitoring; risk scoring; cross-resource alerting; cluster health index; anomaly detection; observability; cloud-native infrastructure.*

I. INTRODUCTION

Kubernetes schedules containers as Pods onto Nodes and keeps them available through Deployments and Services [1], [2]. A production cluster can contain dozens of Nodes and hundreds of Pods whose state changes continuously, so operators rely on metrics rather than manual inspection to detect problems such as Node memory exhaustion, CPU saturation, or Pods stuck in crash loops [3], [4]. Prometheus [5] is the usual metric store for this purpose and Grafana [6] the usual visualization layer.

Two limitations of this standard stack motivate the present work. First, alert rules are normally written per metric with a

fixed threshold. A Pod whose own metrics look acceptable but that runs on a Node close to memory exhaustion is at higher risk than its metrics alone suggest, yet no single-metric rule expresses that dependency. Second, there is no built-in summary of overall cluster condition: operators read several panels and maintain alert rules by hand, metric by metric.

This paper presents a self-hosted monitoring system that addresses both limitations. Its contributions are: (1) a Pod risk score that incorporates the severity of the host Node, so that risk propagates from Node to Pod; (2) a composite Cluster Health Score defined by an explicit weighted formula; (3) an alert lifecycle with per-resource deduplication, in-place escalation,

and automatic clearing; and (4) an open-source implementation with a controlled functional evaluation on a Minikube cluster. Every mechanism is rule-based, so each alert can be traced to the inputs that produced it.

The rest of the paper is organized as follows. Section II reviews related work, Section III describes the method, Section IV the implementation, Section V the evaluation, Section VI discusses results and limitations, and Section VII concludes.

II. RELATED WORK

A. Prometheus-Based Monitoring

Prometheus scrapes metrics at fixed intervals, stores them as time series, and exposes a query language for aggregation [5]. Its Alertmanager component groups and routes alerts that fire from rules defined on those queries [7], and Grafana builds dashboards over the same data [6]. The stack is flexible but leaves two tasks to the operator: writing a rule for each metric of interest and combining the resulting signals into a judgment about the cluster as a whole. Surveys of cloud-native observability describe the same fragmentation across metrics, logs, and traces [3], and site reliability practice advises alerting on conditions that require action rather than on every internal metric [4].

B. Anomaly Detection for Kubernetes

Surveys group anomaly detection techniques into classification-based, nearest-neighbor, clustering, and statistical families, and note that suitability depends on how well normal behavior can be defined [8]. For Kubernetes specifically, the Kubernetes Anomaly Detector (KAD) applies machine-learning models to metrics collected inside the cluster [9]; KubAnomaly uses neural networks to detect anomalous container behavior on an orchestration platform [10]; a graph-based framework represents relations between cluster components and combines unsupervised and supervised models [11]; and learned state machines have been used to monitor runtime behavior and explain detected anomalies [12]. These approaches need training data and tuning, which adds cost for small clusters and makes individual alerts harder to explain. The method in this paper takes a more modest position: it uses explicit thresholds and a weighted score, and obtains context sensitivity by making a Pod's score depend on its host Node, a dependency that the graph-based approach [11] also exploits but with learned models.

C. Comparison with Existing Tools

Table I compares the proposed system with three common alternatives on four capabilities.

TABLE I. COMPARISON OF KUBERNETES MONITORING APPROACHES

Approach	Visual.	Alerting	Self-Host	Cross-Res. Risk
Prometheus / kubectl only	None	Manual rules	Yes	No
Grafana + Prometheus	Yes	Manual rules	Yes	No
Commercial platforms	Yes	Built-in	No	Vendor-specific
Proposed dashboard	Yes	Automatic	Yes	Yes (risk score)

Raw Prometheus queries and kubectl output give direct access to the data but no visualization or alerting. Grafana adds visualization, yet alerts must still be configured manually for each metric. Commercial platforms bundle visualization and alerting, but they are licensed products and their alert logic is generally not open to inspection. The proposed system does not match their breadth (logs, traces, multi-cluster support); it targets small self-hosted deployments in which alert logic must remain inspectable and adjustable.

D. Research Gap

Existing work either provides collection and visualization without context-aware alerting [5], [6], or provides learned anomaly detection that requires training data and is harder to interpret [9]–[12]. The gap addressed here is a transparent middle ground: an integrated, self-hosted pipeline in which risk propagates from Nodes to Pods, cluster condition is summarized by a single index, and alerts have a defined lifecycle, all without a training phase.

III. PROPOSED METHOD

A. System Architecture

The system has four layers (Fig. 1). The Configuration Layer holds the YAML manifests that declare the cluster's workloads. The Kubernetes Layer runs those workloads as Nodes, Pods, Deployments, and Services. The Monitoring Layer is a Prometheus deployment in the same cluster that scrapes Node, Pod, and Deployment metrics. The Processing and Visualization Layer consists of a backend that queries Prometheus and computes risk scores, alerts, and the health score, and a web dashboard that presents them. Layers interact only through defined interfaces (manifests, the Prometheus query API, and the backend REST API), so a component such as the rule-based scorer can be replaced by a learned model without changing the layers around it.

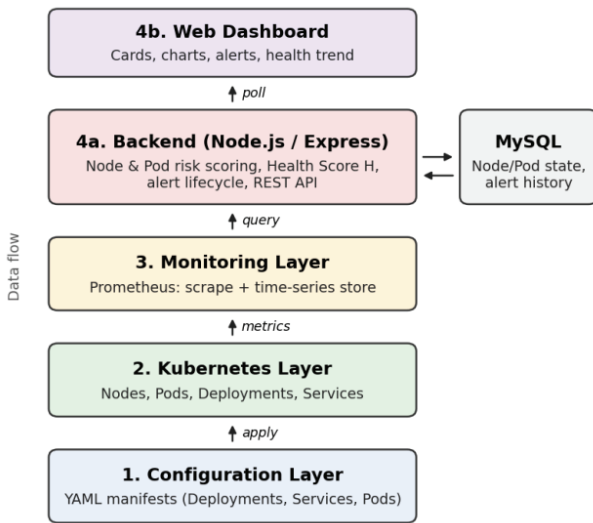


Fig. 1. Layered architecture and data flow of the proposed monitoring system.

B. Metric Acquisition and Data Model

Prometheus scrapes Node availability, Pod status, Deployment state, and CPU and memory consumption at Node and Pod level [5]. On each collection cycle the backend queries these series, attaches every measurement to the Node or Pod it belongs to, and groups records by resource type. The current state of each Node and Pod, together with all alerts, is persisted in a relational database, so the dashboard can show trends and the backend can restart without losing history.

C. Node Risk Evaluation

Let u_n be the larger of a Node's CPU and memory utilization, in percent. The Node severity s_n is CRITICAL if $u_n \geq \theta_c$, WARNING if $\theta_w \leq u_n < \theta_c$, and NORMAL otherwise, with $\theta_w = 78\%$ and $\theta_c = 90\%$ (Table II). Taking the maximum flags a Node when either resource is under pressure.

D. Cross-Resource Pod Risk Score

The risk of a Pod p combines four inputs: its restart count r_p , its CPU utilization c_p , its memory utilization m_p , and the severity $s_{h(p)}$ of the Node $h(p)$ that hosts it:

$$\text{risk}(p) = 100 \cdot (w_r \cdot \rho_p + w_c \cdot c_p + w_m \cdot m_p + w_n \cdot g_p) \quad (1)$$

where $\rho_p = \min(r_p/r_{\max}, 1)$ is the restart count normalized by a saturation value, c_p and m_p are utilization fractions in $[0, 1]$, g_p maps the host-Node severity to 0, 0.5, or 1 for NORMAL, WARNING, or CRITICAL, and the weights w sum to 1. A Pod is classified HIGH risk when $\text{risk}(p) \geq T_H$ and MEDIUM when $T_M \leq \text{risk}(p) < T_H$. The last term couples the Pod to its host: a Pod with unremarkable CPU, memory, and restart values still receives a higher score when its Node is in WARNING or CRITICAL state, which a rule on Pod metrics alone cannot express.

E. Alert Lifecycle

Each alert is keyed by the affected resource and alert type, and carries a severity, a plain-language probable cause, and a suggested corrective kubectl action. An alert is created when a resource first crosses a threshold. On later collection cycles the same alert is updated in place, for example escalated from WARNING to CRITICAL, instead of being duplicated. When the condition no longer holds, the alert is resolved automatically. Algorithm 1 gives the evaluation performed on each cycle.

ALGORITHM 1: PER-RESOURCE RISK EVALUATION

```

1: for each Node n:
2:    $u \leftarrow \max(\text{cpu}(n), \text{mem}(n))$ 
3:   if  $u \geq \theta_c$ :  $\text{sev} \leftarrow \text{CRITICAL}$ 
4:   elif  $u \geq \theta_w$ :  $\text{sev} \leftarrow \text{WARNING}$ 
5:   else:  $\text{sev} \leftarrow \text{NORMAL}$ 
6:   if  $\text{sev} \neq \text{NORMAL}$ :  $\text{raiseOrUpdate}(n, \text{sev})$ 
7:   else:  $\text{clearIfExists}(n)$ 
8: for each Pod p:
9:    $\text{risk}(p) \leftarrow \text{Eq. (1)}$ 
10:  if  $\text{risk}(p) \geq T_H$ :  $\text{raiseOrUpdate}(p, \text{HIGH})$ 
11:  elif  $\text{risk}(p) \geq T_M$ :  $\text{raiseOrUpdate}(p, \text{MED})$ 
12:  else:  $\text{clearIfExists}(p)$ 
    
```

F. Composite Cluster Health Score

Overall cluster condition is summarized by a Cluster Health Score H in the range 0–100, recomputed after every collection cycle:

$$H = 0.40 \cdot H_{\text{node}} + 0.35 \cdot H_{\text{pod}} + 0.25 \cdot U_{\text{res}} \quad (2)$$

Here H_{node} starts at 100 and is reduced for every Node in WARNING or CRITICAL state; H_{pod} is reduced in proportion to the number of Pods classified MEDIUM or HIGH; and U_{res} reflects average CPU and memory consumption across the cluster relative to a comfortable operating range. The weights give Node health more influence than Pod health because a Node fault can affect every Pod scheduled on it, whereas a single Pod fault is usually contained.

IV. IMPLEMENTATION

The backend is written in Node.js with the Express framework and exposes REST endpoints for cluster status, individual Node and Pod records, health-score history, and active alerts. State is stored in MySQL. The frontend is a browser-based single-page application that polls these endpoints and re-renders the affected components when new data arrives, so the display stays close to real time without manual refresh. It presents Node health, Pod status, Deployment availability, CPU and memory usage, health-score history, and alerts using cards, charts, and status indicators.

Prometheus is deployed in the same cluster as the monitored workloads. Development and testing used Minikube [13] and Docker [14] on a laptop with a multi-core CPU and at least 8 GB of RAM; the monitoring pipeline itself is lightweight relative to the workloads it observes. The deployment workflow

is: apply the YAML manifests, let Prometheus begin scraping the resulting resources, have the backend retrieve and process the metrics, and render the results on the dashboard.

V. EXPERIMENTAL EVALUATION

A. Setup

The system was deployed on a Minikube cluster running a demo application (k8s-demo-app) alongside ArgoCD, Flux, Prometheus, Loki, and Rancher components, giving 31 monitored Pods. Table II lists the parameters used.

TABLE II. EVALUATION PARAMETERS

Parameter	Symbol	Value
Node WARNING threshold	θ_w	78%
Node CRITICAL threshold	θ_c	90%
Pod risk thresholds (MEDIUM / HIGH)	T_M / T_H	[T_M] / [T_H]
Pod risk weights (restarts, CPU, memory, host Node)	w_r, w_c, w_m, w_n	[w_r, w_c, w_m, w_n]
Restart saturation value	r_{max}	[r_max]
Health score weights (Node, Pod, utilization)	—	0.40 / 0.35 / 0.25
Collection interval	—	[interval] s
Cluster	—	Minikube, [N] Node(s), 31 Pods
Software versions	—	Kubernetes [ver], Prometheus [ver]

B. Test Scenarios

Four controlled scenarios exercised the alerting and scoring logic (Table III). In S1 the cluster ran without induced faults. In S2, Node memory utilization was driven up by running a memory-consuming workload on a target Node until it crossed the warning and then the critical threshold. In S3 a Pod was forced into repeated restarts to simulate a crash loop. In S4 the induced faults were removed and the alerts were observed until the metrics returned to a normal range.

TABLE III. TEST SCENARIOS: EXPECTED AND OBSERVED BEHAVIOR

ID	Condition	Expected	Observed
S1	None (steady state)	No alerts; high health score	No alerts; consistently high Cluster Health Score
S2	Node memory raised past θ_w , then θ_c	Node alert WARNING → CRITICAL; Node-health component and H fall; Pods on the Node gain risk	Alert escalated from WARNING to CRITICAL with the recommended action changing accordingly; Node-health component reduced; Pod risk scores rose
S3	Pod forced into repeated restarts	HIGH-risk Pod alert citing restarts	Pod classified HIGH independent of its Node; alert cited the restart count as likely cause
S4	Faults removed	Alerts cleared; no duplicates	Persistent conditions produced one alert updated in place; each alert cleared automatically on recovery

In all four scenarios the observed behavior matched the expected behavior. These are functional checks: they show that the logic behaves as specified, not how accurate or fast detection is on production workloads (Section VI-B).

C. Dashboard Observations

Figs. 2–4 show the deployed dashboard running against the Minikube cluster. Tables IV–VI summarize values read from these captures.

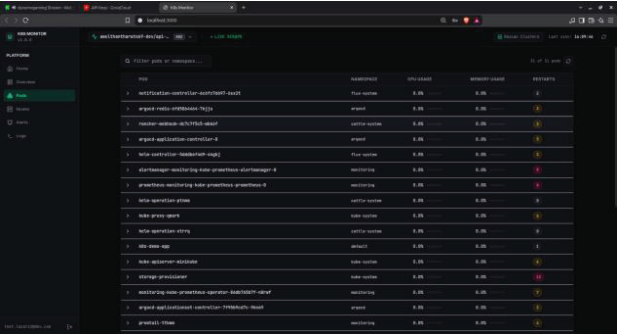


Fig. 2. Pods view of the dashboard: live CPU usage, memory usage, and restart counts (31 of 31 Pods tracked).

TABLE IV. SAMPLE OF MONITORED POD METRICS (16 OF 31 PODS SHOWN IN FIG. 2)

Pod	Namespace	CPU %	Mem %	Restarts
notification-controller-...-6sx2t	flux-system	0.0	0.0	2
argocd-redis-...-7kjjs	argocd	0.0	0.0	3
rancher-webhook-...-mbk6f	cattle-system	0.0	0.0	3
argocd-application-controller-0	argocd	0.0	0.0	3
helm-controller-...-4ngbj	flux-system	0.0	0.0	3
alertmanager-...-alertmanager-0	monitoring	0.0	0.0	8
prometheus-...-prometheus-0	monitoring	0.0	0.0	8
helm-operation-ptnmm	cattle-system	0.0	0.0	0
kube-proxy-qmrsk	kube-system	0.0	0.0	6
helm-operation-xtrrq	cattle-system	0.0	0.0	0
k8s-demo-app	default	0.0	0.0	1
kube-apiserver-minikube	kube-system	0.0	0.0	6
storage-provisioner	kube-system	0.0	0.0	13
monitoring-kube-prometheus-operator-...-n8rwf	monitoring	0.0	0.0	7
argocd-applicationset-controller-...-9kn69	argocd	0.0	0.0	3
promtail-ttkwm	monitoring	0.0	0.0	4
Total / Average (n = 16 of 31 pods)	—	0.0	0.0	Σ = 70, avg = 4.4

In this 16-Pod sample the dashboard reported 0.0% CPU and memory utilization for every Pod, consistent with the lightly loaded development cluster. Restart counts varied from 0 (helm-operation-ptnmm, helm-operation-xtrrq) to 13 (storage-provisioner), totaling 70 (mean 4.4 per Pod). Because the Pod risk score (Eq. 1) includes restart count, storage-provisioner and the two Prometheus-stack Pods with 8 restarts would be the first flagged if the corresponding thresholds were reached.

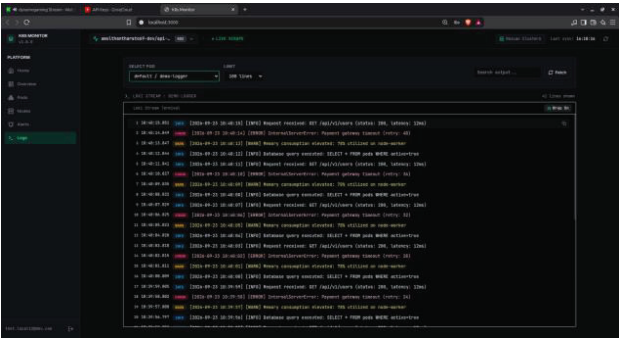


Fig. 3. Live Loki log stream for the demo-logger Pod, captured through the dashboard's Logs view.

TABLE V. LOG-LEVEL DISTRIBUTION IN FIG. 3

Log Level	Count	Share of Sample
INFO	12	60%
WARN	4	20%
ERROR	4	20%
Total (first 20 of 41 lines shown)	20	100%

In the visible 20-line window, INFO entries account for 60%, and WARN (elevated memory utilization on node-worker) and ERROR (a simulated payment-gateway timeout) for 20% each. The repeating five-line pattern shows that the source is a synthetic workload generator, which suits controlled validation but means the distribution should not be read as representative of production traffic.

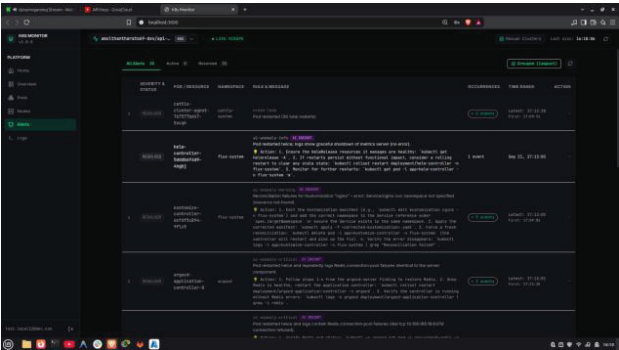


Fig. 4. Alerts view showing crash-loop, info, warning, and critical alerts with suggested kubectl actions.

TABLE VI. ALERT SUMMARY FROM FIG. 4

Metric	Value
Total alerts logged	30
Active alerts	0
Resolved alerts	30
Alert types observed	crash-loop, ai-anomaly-info, ai-anomaly-warning, ai-anomaly-critical

All 30 alerts logged in this window had been resolved at capture time (0 active), and each carried a plain-language cause and a suggested kubectl step, for example restarting the ArgoCD application controller after repeated Redis connection-pool failures. This is consistent with the lifecycle in Section III-E. The ai-anomaly-* labels are alert-type names used in the

dashboard; the underlying logic is the rule-based scoring of Section III, not a trained model.

VI. DISCUSSION AND LIMITATIONS

A. Discussion

The evaluation supports three observations. First, the host-Node term in Eq. (1) has the intended cross-resource effect: Pods on the stressed Node showed rising risk scores while the Node was escalating. Second, in-place alert updates kept the alert list bounded, since a persistent condition produced one alert instead of one per collection cycle. Third, the rule-based design keeps alerts explainable, as the generated cause identifies the inputs that triggered it (for example, the restart count in S3). Because the whole stack is open source and self-hosted, it also avoids licensing costs, and its thresholds and weights can be inspected and adjusted.

B. Limitations and Threats to Validity

The results should be read with the following limits in mind. (1) Scale: all tests ran on a single Minikube development cluster with 31 Pods and synthetic faults, so behavior on production-scale clusters and real workloads is not established. (2) No baseline: the system was not quantitatively compared with static per-metric Alertmanager rules, so any advantage in detection time is supported by the design and scenario behavior rather than by measured latency. (3) Unmeasured quantities: detection latency, false-positive and false-negative rates under varied workloads, dashboard response time, the delay between a metric change and its display, and the monitoring overhead on the cluster were not measured. (4) Parameter selection: thresholds and weights were set by design judgment and were not tuned or validated against labeled incidents. (5) Fault coverage: only memory pressure and crash loops were tested; CPU saturation, Deployment rollout failures, and network faults were not.

VII. CONCLUSION AND FUTURE WORK

This paper presented a Prometheus-based Kubernetes monitoring system built around three mechanisms: a cross-resource Pod risk score that includes host-Node severity, a composite Cluster Health Score, and a deduplicating alert lifecycle with automatic clearing. In controlled tests on a Minikube cluster the system raised no alerts in steady state, escalated and then cleared a Node memory-pressure alert, and classified a crash-looping Pod as high risk. The main limitation is the scale and scope of the evaluation.

Future work will (i) compare the risk score against static Alertmanager rules on detection latency and false-alarm rate using replayed fault traces and production-like workloads, (ii) calibrate thresholds and weights against labeled incidents, (iii) measure monitoring overhead and dashboard latency, (iv) augment or replace the rule-based scorer with a model trained on historical metrics, which the layered architecture permits,

and (v) add notification channels such as Slack or email, multi-cluster support, and automated root-cause tracing.

ACKNOWLEDGMENT

This work was carried out in the Department of Computer Science and Engineering, Priyadarshini College of Engineering, Nagpur. We are thankful to the department and its faculty for the academic environment and the guidance that kept the project on track. We are especially grateful to our project guide, who gave us time to review our progress and offered feedback on both the system design and this paper.

We also thank the maintainers and contributors of Kubernetes, Prometheus, Grafana, Minikube, and Loki. Their openly published documentation and example configurations made it possible for a student team to build and test a complete monitoring pipeline without commercial tools.

Finally, we thank our classmates for the discussions and troubleshooting help during development and testing, which shaped several of the design decisions described in this paper.

REFERENCES

- [1] Kubernetes Authors, "Kubernetes Documentation." [Online]. Available: <https://kubernetes.io/docs/home/>
- [2] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," *ACM Queue*, vol. 14, no. 1, pp. 70–93, 2016.
- [3] J. Kosińska, B. Balis, M. Konieczny, M. Malawski, and S. Zielinski, "Toward the observability of cloud-native applications: The overview of the state-of-the-art," *IEEE Access*, vol. 11, pp. 73036–73052, 2023.
- [4] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, Eds., *Site Reliability Engineering: How Google Runs Production Systems*. Sebastopol, CA, USA: O'Reilly Media, 2016.
- [5] Prometheus Authors, "Prometheus: Monitoring System and Time Series Database." [Online]. Available: <https://prometheus.io/docs/>
- [6] Grafana Labs, "Grafana Documentation." [Online]. Available: <https://grafana.com/docs/>
- [7] Prometheus Authors, "Alertmanager." [Online]. Available: <https://prometheus.io/docs/alerting/latest/alertmanager/>
- [8] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM Computing Surveys*, vol. 41, no. 3, Art. 15, 2009.
- [9] J. Kosińska and M. Tobiasz, "Detection of cluster anomalies with ML techniques," *IEEE Access*, vol. 10, pp. 110742–110753, 2022.
- [10] C. W. Tien, T. Y. Huang, C. W. Tien, T. C. Huang, and S. Y. Kuo, "KubAnomaly: Anomaly detection for the Docker orchestration platform with neural network approaches," *Engineering Reports*, vol. 1, no. 5, e12080, 2019.
- [11] V. Anemogiannis, B. Andreou, K. Myrtollari, K. Panagidi, and S. Hadjiefthymiades, "Enhancing Kubernetes resilience through anomaly detection and prediction," *arXiv:2503.14114*, 2025.
- [12] C. Cao, A. Blaise, S. Verwer, and F. Rebecchi, "Learning state machines to monitor and detect anomalies on a Kubernetes cluster," in *Proc. 17th Int. Conf. Availability, Reliability and Security (ARES)*, 2022.
- [13] Kubernetes Authors, "Install Tools — Minikube and Kind." [Online]. Available: <https://kubernetes.io/docs/tasks/tools/>
- [14] Docker Inc., "Docker Documentation." [Online]. Available: <https://docs.docker.com/>