iJRASET
International Journal For Research in Applied Science and Engineering Technology



INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 14 **Issue:** X **Month of publication:** October 2026

DOI: https://doi.org/10.22214/ijraset.2026.85239

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 14 Issue X Oct 2026- Available at www.ijraset.com

AI Self-Healing DevOps: A Closed-Loop Framework for Anomaly Detection, Contextual Root-Cause Analysis and Policy-Governed Automated Recovery

Rajshri Pote, Shreya Prasad, Atharva Lutade, Swati Chaudhari, Krunal Vighre, Mahesh Khadilkar
Dept. of Computer Science and Engineering Priyadarshini College of Engineering Nagpur, India

Abstract: ***Modern cloud-native applications generate large volumes of telemetry, yet incident response still depends heavily on engineers who must notice a fault, diagnose it and apply a fix by hand. This paper presents AI Self-Healing DevOps, a working prototype that closes this loop through six stages: Monitor, Detect, Decide, Heal, Verify and Learn. Telemetry (metrics, logs, traces and events) flows through a topic-based event router into an anomaly detector that combines an Isolation Forest with a z-score guard. A Random Forest classifier labels the failure type, a dependency-graph and correlation analysis localises the root cause, and a retrieval step over a growing incident knowledge base supplies similar past cases for an explanation. A policy and risk engine then validates the remediation plan against an allowlist, playbooks, a flapping guard and a human-approval rule for high-risk actions, and dispatches it to Ansible, Argo CD and GitHub Actions executors. A verifier confirms recovery, retries, or escalates to an engineer, and the outcome updates the knowledge base and the approval policy. The prototype runs in a simulation mode and in a live mode against a real Express application, and is evaluated on six fault scenarios. We describe the design, state clearly which components are real and which are stand-ins, and discuss limitations and directions for production use.***

Index Terms: ***AIOps, self-healing systems, anomaly detection, Isolation Forest, root-cause analysis, retrieval-augmented generation, GitOps, Kubernetes, DevOps automation.***

I. INTRODUCTION

Microservice and container-orchestrated deployments have raised the pace of software delivery, but they have also multiplied the ways in which a system can fail. A single misbehaving service can degrade its dependants, and the symptoms that operators observe are often far from the true cause. Site reliability practice addresses this with monitoring, alerting, runbooks and on-call rotation [1], yet the time between a fault and its repair, commonly measured as mean time to recovery (MTTR), is still dominated by human reaction and diagnosis.

The idea of systems that monitor, analyse, plan and execute their own corrections was articulated in the autonomic computing vision [2]. Recent advances in unsupervised anomaly detection [3], [4], tree-based classification [5], retrieval-augmented language models [6] and declarative GitOps tooling [7], [8] make a practical version of that vision feasible for ordinary DevOps teams. What is often missing is an integrated, inspectable pipeline in which every automated action is justified, gated by policy and verified afterwards.

This paper describes AI Self-Healing DevOps, an open-source prototype [9] that implements such a pipeline. Its contributions are as follows.

- A six-stage closed loop (Monitor, Detect, Decide, Heal, Verify, Learn) in which all telemetry and decisions travel through a topic-based event router.
- A detection layer that pairs an Isolation Forest with a z-score guard, so that values far outside the training range are still flagged.
- A contextual diagnosis layer combining failure classification, dependency-aware root-cause analysis and retrieval from a knowledge base that grows with every resolved incident.
- A policy and risk engine that gates remediation using an allowlist, playbook checks, risk levels, human approval, a flapping guard and restrictions learned from past success rates.

©IJRASET: All Rights are Reserved

- An explicit statement of which blocks are real, which are simulated, and which are dry-run, so that the claims of the prototype are not overstated.

The remainder of the paper is organised as follows. Section II reviews related work, Section III presents the architecture, Section IV details the algorithms, Section V describes the implementation, Section VI discusses the evaluation scenarios, Section VII lists limitations, and Section VIII concludes.

II. RELATED WORK

A. Autonomic and Reliability Engineering

Kephart and Chess described autonomic computing as a control loop of monitoring, analysis, planning and execution over a shared knowledge base [2]. The framework presented here follows the same structure, adding explicit verification and policy stages. Beyer et al. document the operational practices of production services at scale, including postmortems, error budgets and automation of toil [1], which motivate the reduction of manual intervention targeted by this work.

B. Anomaly Detection and Failure Classification

Chandola et al. survey anomaly detection techniques and the challenges of defining normal behaviour [3]. The Isolation Forest of Liu et al. isolates anomalies through random partitioning and requires no labelled faults, which suits operational data where failures are rare [4]. Random Forests offer robust supervised classification with interpretable feature importance [5], and both are available in scikit-learn [10].

C. Retrieval-Augmented Diagnosis

Retrieval-augmented generation grounds a language model in documents retrieved at query time [6]. In an operations setting the retrieved documents are past incidents and their resolutions. This work uses cosine-similarity retrieval over an incident knowledge base and treats the language-model explanation as an optional layer.

D. Orchestration, Messaging and GitOps

Kubernetes provides declarative orchestration with built-in restart and scaling primitives [11]. Apache Kafka supplies a durable publish–subscribe log for event streams [12], and Prometheus provides a pull-based metrics system [13]. Argo CD reconciles cluster state with a Git repository [7], and Ansible offers agentless automation through playbooks [8]. Distributed tracing, as introduced by Dapper, completes the observability signals [14]. The present prototype integrates the interfaces of these tools while keeping the demonstration self-contained.

III. SYSTEM ARCHITECTURE

Fig. 1 shows the four layers of the framework. Telemetry enters the system from the target infrastructure, is routed through events, analysed, converted to a governed remediation plan, executed, and verified. The outcome of each incident feeds back into the knowledge base and policy.

A. Target Infrastructure and Observability

The simulation mode models four services running as replicas across two nodes. The live mode targets a real Express application that exposes Prometheus-format metrics. In both cases the four observability signals (metrics, logs, traces and events) are produced and passed on. A Kubernetes deployment manifest and a Prometheus configuration are provided for containerised deployment, but they are not applied by the default demonstration.

B. Event Router

All telemetry and all decisions are published to named topics on an in-process router. Components subscribe independently, which keeps detection, diagnosis and remediation decoupled. An optional mirror forwards events to a Kafka broker when a bootstrap address is configured; this path has not been exercised against a live broker.

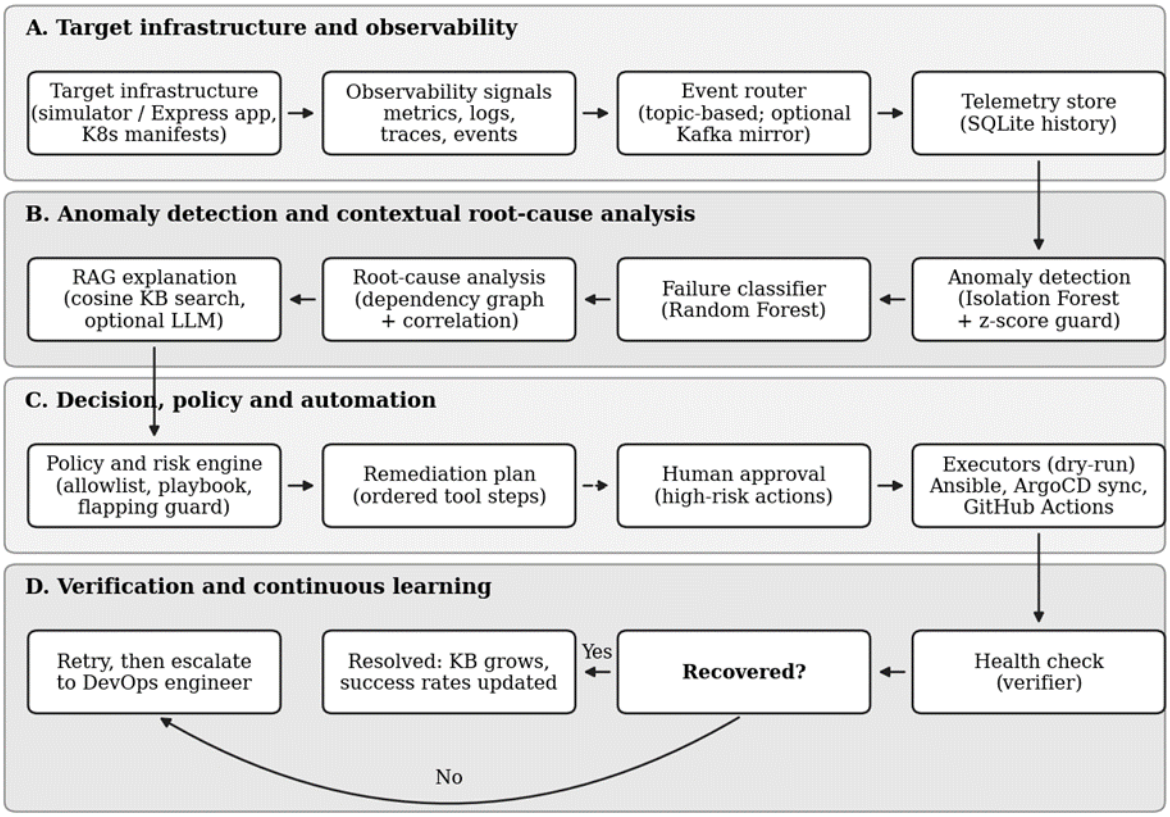


Fig. 1. Block diagram of the AI Self-Healing DevOps framework.

C. Detection and Contextual Diagnosis

Incoming metric vectors are scored by the anomaly detector. When an anomaly is raised, a classifier assigns a failure type, the root-cause module walks the service dependency graph and correlates the signals of upstream and downstream services, and the knowledge base is queried for similar past incidents. A natural-language explanation is produced from this context, by a language model if an API key is configured and by a template otherwise.

D. Decision, Automation and Verification

The policy engine evaluates the proposed remediation and either approves it, requests human approval or blocks it. Approved plans are expanded into ordered tool steps and dispatched to the Ansible, Argo CD and GitHub Actions executors. The verifier then checks service health. If the service has not recovered, remediation is retried and, after repeated failure, the incident is escalated to a DevOps engineer. Resolved incidents are added to the knowledge base and their outcomes update the per-action success statistics.

IV. METHODOLOGY

A. Anomaly Detection

Let x denote a feature vector of service metrics (for example CPU, memory, latency and error rate). An Isolation Forest is fitted on telemetry gathered while the system is healthy. The anomaly score of x for a sample of size n is [4]

$$s(x, n) = 2^{(-E[h(x)] / c(n))} \quad (1)$$

where h(x) is the path length of x in an isolation tree and c(n) = 2H(n−1) − 2(n−1)/n is the average path length of an unsuccessful search, with H(i) the harmonic number. Scores close to 1 indicate anomalies. Because tree-based isolation cannot express how far a value lies beyond the range seen in training, a z-score guard is added:

$$z_j = | x_j - \mu_j | / \sigma_j > \tau \quad (2)$$

where μ_j and σ_j are the healthy mean and standard deviation of metric j and τ is a threshold. An observation is flagged if either the Isolation Forest or the guard fires.

B. Failure Classification

A Random Forest [5] maps the anomalous feature vector to a failure class such as crash, memory leak, CPU overload, bad deployment or dependency failure. It returns class probabilities and feature importances, which the dashboard displays and which also support a what-if analysis in which metric values are varied interactively. The classifier is trained on synthetic symptom profiles, since labelled production incidents are not available to the prototype. Fig. 2 shows the AI Lab page, which presents the Isolation Forest anomaly score, the Random Forest class probabilities and feature importances, the what-if analyzer and the incident knowledge base.

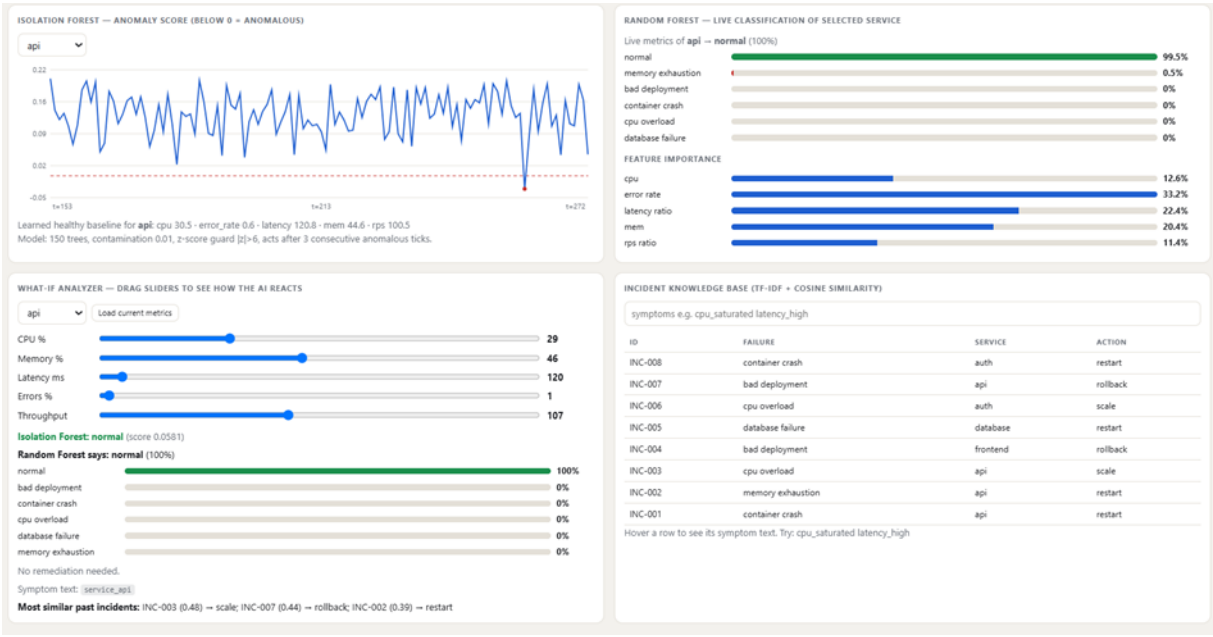


Fig. 2. AI Lab page: Isolation Forest anomaly score, Random Forest classification with feature importance, what-if analyzer and incident knowledge base retrieval.

C. Root-Cause Analysis

Services are nodes in a dependency graph. When several services show symptoms at once, the module correlates their anomalies and prefers the service that sits furthest upstream among the affected nodes, so that a failing database is reported ahead of the front-end errors it causes. The resulting root cause is highlighted in the architecture view of the dashboard.

D. Knowledge Retrieval

Each resolved incident is stored with its symptoms, root cause and applied fix. For a new incident with representation q, stored incidents d_i are ranked by cosine similarity

$$sim(q, d_i) = (q \cdot d_i) / (\|q\| \|d_i\|) \quad (3)$$

and the highest-ranked cases are supplied to the explanation module. Because every resolved incident is appended, retrieval quality improves as the system operates.

E. Policy and Risk Engine

A proposed action is accepted only if it passes the following checks: (i) the action is on an allowlist; (ii) a playbook exists for the diagnosed failure; (iii) its risk level is acceptable, otherwise human approval is required; (iv) it would not cause flapping, that is, repeated remediation of the same service in a short period; and (v) its historical success rate has not fallen below a restriction level. For an action a with n_a past executions of which s_a succeeded, the learned success rate is

$$r_a = s_a / n_a \quad (4)$$

and a low r_a tightens the approval requirement for that action. These updates are published on a policy-update topic and are visible in the audit trail. Only the approval rules change; no model is retrained. Fig. 3 shows the Policy & Approvals page, which lists the remediation playbook with risk levels, the learned success rates and the guard rails.

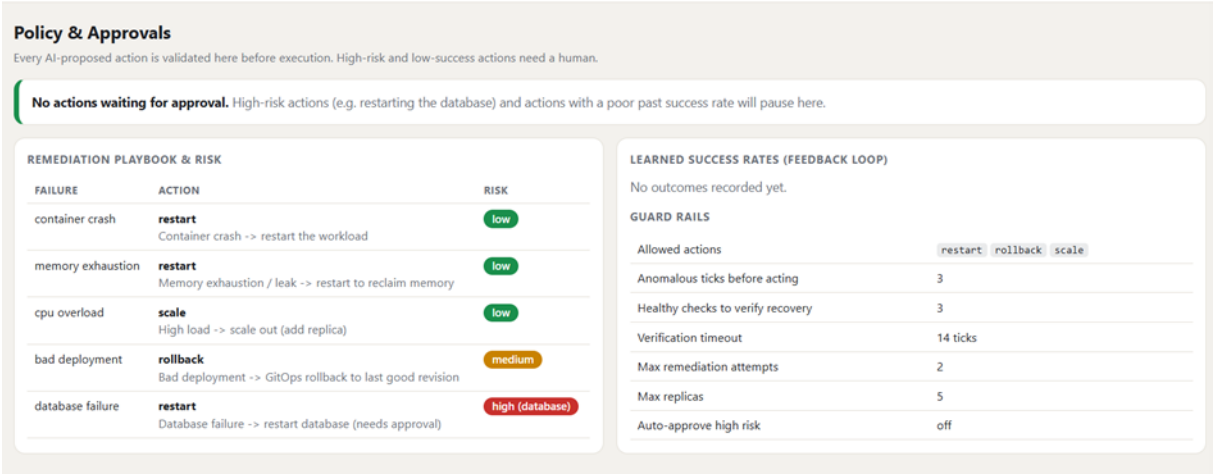


Fig. 3. Policy & Approvals page: remediation playbook and risk levels, learned success rates and guard rails.

F. Verification and Escalation

After execution the verifier re-checks service health. A positive result closes the incident. A negative result triggers a retry and, if retries are exhausted, escalation to an engineer, who can mark the incident as fixed through the dashboard.

V. IMPLEMENTATION

The prototype is written in Python and uses scikit-learn for the models [10]. A web dashboard of sixteen pages provides project information, live monitoring, architecture and dependency views, incident history with export, a chaos page for fault injection, an AI page for model inspection, a policy page with the approval queue, logs, an audit trail, reports and settings. Incidents, metrics, logs and reports are persisted in a local database so that they survive restarts. The target application used in live mode is a Node.js Express service. Fig. 4 shows the Overview page, which summarises incident statistics, per-service health and the incident timeline, and Fig. 5 shows the Live Monitor page, in which real-time metrics are plotted and anomalous samples are marked.

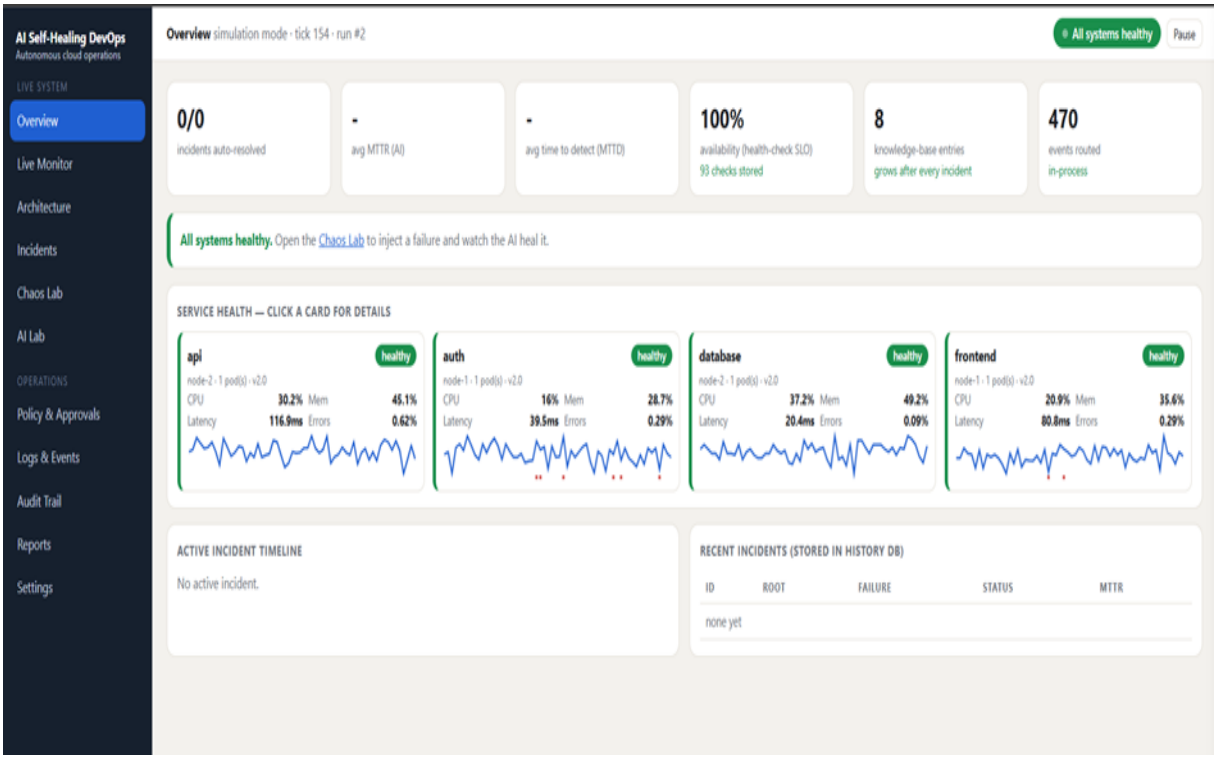


Fig. 4. Overview page: incident statistics, service health cards, active incident timeline and recent incidents.

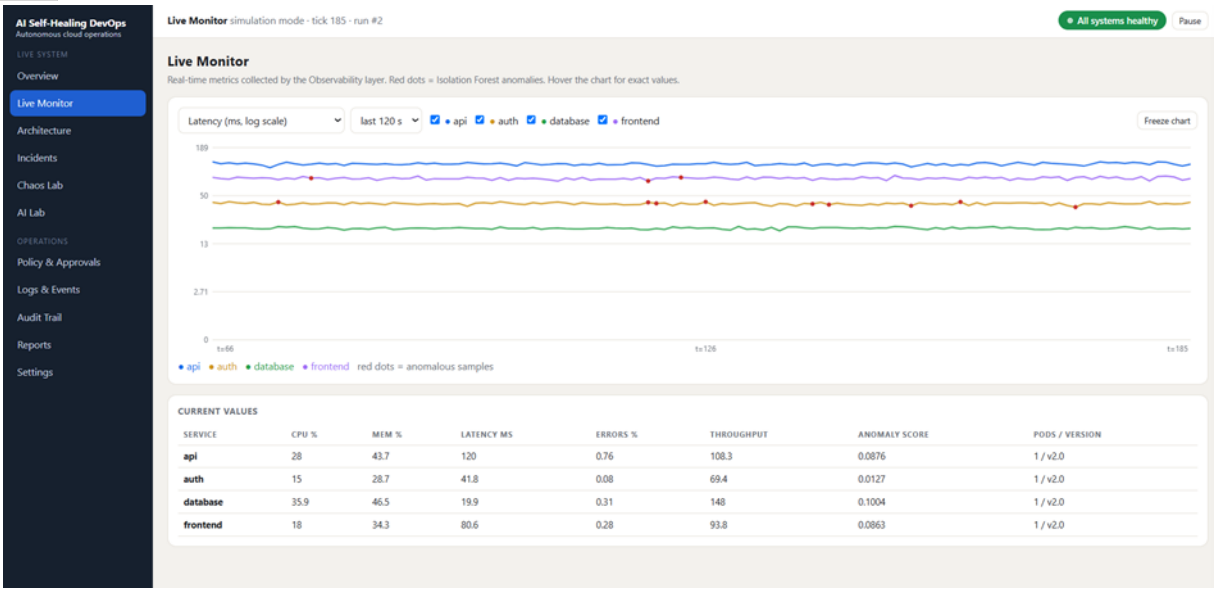


Fig. 5. Live Monitor page: real-time service metrics with Isolation Forest anomalies marked as red dots.

Table I maps each block of the framework to its implementation status. We include this table because several integrations are intentionally dry-run, and a reader should be able to see precisely what has been demonstrated.

TABLE I
IMPLEMENTATION STATUS OF FRAMEWORK BLOCKS

Block	Implementation	Status
Target infrastructure	Simulator (4 services, 2 nodes); Express demo app; Kubernetes manifest	Simulated or local; manifest not applied by the demo
Observability	Metrics, logs, traces and events; Prometheus-format endpoint	Real signals from simulator and live app
Event router	Topic-based in-process router; optional Kafka mirror	In-process; Kafka mirror untested on a live broker
Anomaly detection	Isolation Forest with z-score guard	Real
Root-cause analysis	Dependency graph and correlation	Real
Knowledge retrieval	Cosine retrieval; knowledge base grows per incident	Real
Explanation	Language model if API key present, else template	LLM optional
Policy and risk engine	Allowlist, playbooks, risk, approval, flapping guard, learned restriction	Real
Ansible, Argo CD, GitHub Actions	Playbooks, Application manifest and rollback workflow	Provided; dispatch is dry-run and audited
Verifier and escalation	Health check, retry, escalate to engineer	Real
Continuous learning	Knowledge base growth and success-rate tracking	Real for rule updates; models not retrained

©IJRASET: All Rights are Reserved | SJ Impact Factor 7.538 | ISRA Journal Impact Factor 7.894 |

In live mode, restart and rollback are real process restarts, whereas scaling is emulated with a single local process. These stand in for kubectl rollout restart, Argo CD rollback and kubectl scale respectively. Replacing these executors with real calls is the principal step towards production use.

VI. EVALUATION SCENARIOS AND DISCUSSION

The terminal demonstration exercises six fault scenarios that cover the main behaviours of the loop. Table II summarises each scenario and the behaviour the framework is designed to show. The dashboard also provides a chaos page for injecting these faults, including combined faults, and for defining custom failures. Fig. 6 shows the chaos page with its one-click scenarios and custom failure injection.

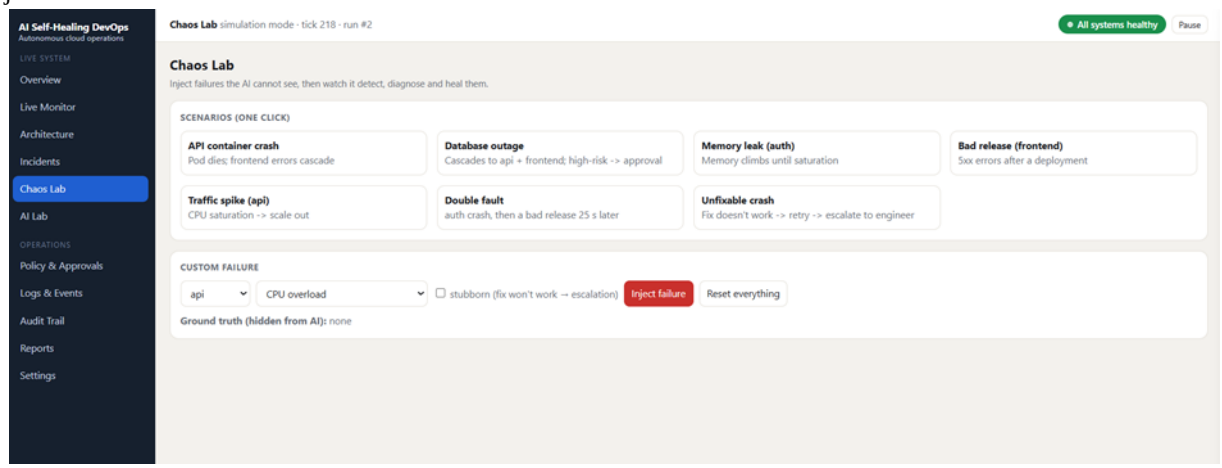


Fig. 6. Chaos Lab page: one-click fault scenarios and custom failure injection.

TABLE II
FAULT SCENARIOS AND EXPECTED FRAMEWORK BEHAVIOUR

#	Scenario	Expected behaviour
1	Service crash	Detected, classified, restarted automatically and verified
2	Memory leak	Gradual growth detected before exhaustion; corrective action chosen by policy
3	CPU overload	Detected under load; scaling or equivalent action planned
4	Bad deployment	Regression after a release diagnosed; GitOps rollback planned
5	Database failure	Dependent services show symptoms; root cause traced to the database; human approval required before action
6	Unfixable failure	Remediation retried, verification fails, incident escalated to a DevOps engineer

These scenarios illustrate three properties of the design. First, low-risk faults are repaired without human involvement. Second, high-risk actions, such as those touching the database, are held for approval, which demonstrates that automation is bounded by policy. Third, a fault that cannot be fixed is not hidden: after retries the system escalates and records the outcome. The dashboard reports MTTR, mean time to detect (MTTD) and availability, and reports compare MTTR with a manual baseline.

We deliberately do not report numerical accuracy or MTTR improvements in this paper. In the prototype, MTTR is counted in monitoring ticks (one tick represents one second) and the manual figure shown for comparison is an assumed value, not a measurement. Claims of quantitative benefit would therefore be unsupported. A controlled study on real workloads is left to future work.

VII. LIMITATIONS AND THREATS TO VALIDITY

- The failure classifier is trained on synthetic symptom profiles. Its behaviour on real incidents will depend on labelled historical data.

- Ansible, Argo CD and GitHub Actions actions are dispatched as audited dry runs. Their real effects, latency and failure modes are not measured.
- Simulation mode produces its own metrics and applies fixes inside the simulator, so detection and recovery are evaluated against a model the authors control.
- The Kafka mirror has not been tested against a live broker, and the default router is in-process and therefore not fault-tolerant.
- Learning is limited to updating knowledge-base content and approval rules; the Isolation Forest and the classifier are not retrained online.
- The language-model explanation is optional and has not been evaluated for faithfulness to the retrieved incidents.

VIII. CONCLUSION AND FUTURE WORK

We presented AI Self-Healing DevOps, a prototype that integrates anomaly detection, contextual root-cause analysis, policy-governed remediation, verification and learning into one observable loop. Its main design choices are a hybrid detector with a range guard, retrieval over a self-growing incident knowledge base, and a policy layer that keeps high-risk actions under human control. The prototype is explicit about its scope: the intelligence components are implemented and runnable, whereas infrastructure executors remain dry-run.

Future work includes replacing the dry-run executors with real kubectl, Ansible and Argo CD calls on a managed cluster, training the classifier on labelled production incidents, retraining detectors on drift, evaluating retrieval and explanation quality, and measuring MTTR and MTTD on real workloads against a manual baseline.

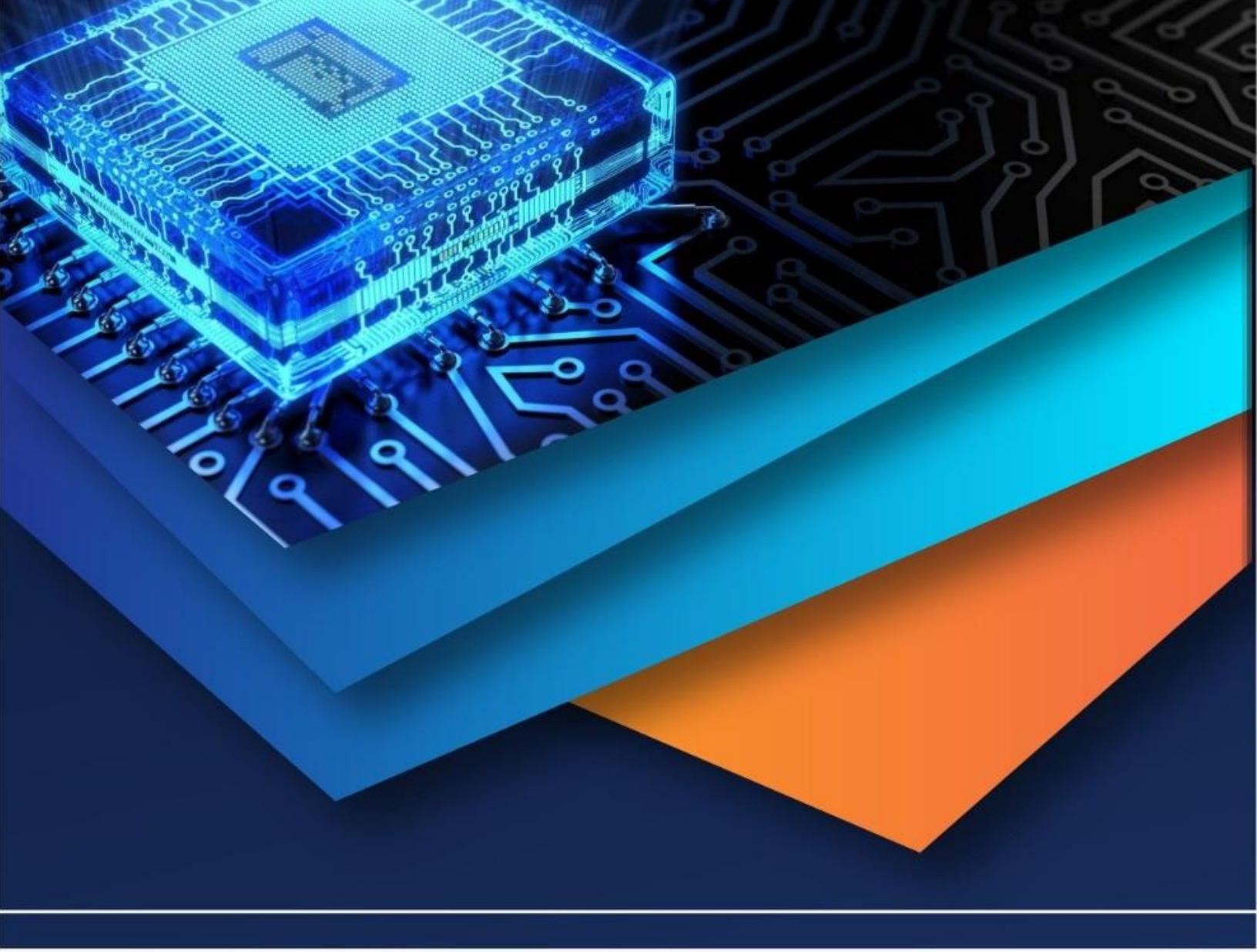
IX. ACKNOWLEDGMENT

The author thanks the project guide and colleagues for feedback on the prototype and its presentation. [Edit as appropriate.]

REFERENCES

[1] 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.
[2] J. O. Kephart and D. M. Chess, “The vision of autonomic computing,” Computer, vol. 36, no. 1, pp. 41–50, Jan. 2003.
[3] V. Chandola, A. Banerjee and V. Kumar, “Anomaly detection: A survey,” ACM Comput. Surv., vol. 41, no. 3, pp. 1–58, Jul. 2009.
[4] F. T. Liu, K. M. Ting and Z.-H. Zhou, “Isolation forest,” in Proc. 8th IEEE Int. Conf. Data Mining (ICDM), Pisa, Italy, 2008, pp. 413–422.
[5] L. Breiman, “Random forests,” Mach. Learn., vol. 45, no. 1, pp. 5–32, Oct. 2001.
[6] P. Lewis et al., “Retrieval-augmented generation for knowledge-intensive NLP tasks,” in Advances in Neural Information Processing Systems (NeurIPS), vol. 33, 2020, pp. 9459–9474.
[7] Argo Project, “Argo CD: Declarative GitOps continuous delivery for Kubernetes.” [Online]. Available: https://argo-cd.readthedocs.io
[8] Red Hat, “Ansible documentation.” [Online]. Available: https://docs.ansible.com
[9] M. Khadilkar, “AI-Self-Healing-DevOps,” GitHub repository. [Online]. Available: https://github.com/maheshkhadilkar07/AI-Self-Healing-DevOps
[10] F. Pedregosa et al., “Scikit-learn: Machine learning in Python,” J. Mach. Learn. Res., vol. 12, pp. 2825–2830, 2011.
[11] 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.
[12] J. Kreps, N. Narkhede and J. Rao, “Kafka: A distributed messaging system for log processing,” in Proc. NetDB Workshop, 2011.
[13] Prometheus Authors, “Prometheus: Monitoring system and time series database.” [Online]. Available: https://prometheus.io
[14] B. H. Sigelman et al., “Dapper, a large-scale distributed systems tracing infrastructure,” Google, Mountain View, CA, USA, Tech. Rep., 2010.

©IJRASET: All Rights are Reserved | SJ Impact Factor 7.538 | ISRA Journal Impact Factor 7.894 |



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Call : 08813907089 (24*7 Support on Whatsapp)