

AutoOps Intelligence: An AI-Powered Predictive and Self-Healing DevOps Automation Platform

Deepak

Department of CSE
Chandigarh University
Mohali, India
dpkjangra.011@gmail.com

Monika Bishnoi

Department of CSE
Chandigarh University
Mohali, India
monikabishnoi3@gmail.com

Parveen

Department of CSE
Chandigarh University
Mohali, India
pits48617@gmail.com

Hemant Yadav

Department of CSE
Chandigarh University
Mohali, India
hemant031104@gmail.com

Anuj

Department of CSE
Chandigarh University
Mohali, India
bhanwala2005@gmail.com

Abstract—Continuous Integration and Continuous Deployment (CI/CD) pipelines are very important for the success of modern software development. They help make it easier to release applications often, consistently, and reliably. However, even with these improvements, many CI/CD processes still use manual steps to set up infrastructure, check for possible problems during deployment, and fix issues when they happen. This can lead to more failed releases, longer periods without service, and higher costs. This paper introduces AutoOps Intelligence, an AI-based DevOps automation tool. It uses predictive risk analysis to foresee potential deployment problems and automatically handles fixes after deployment. The platform looks at a connected GitHub repository to figure out the application's technology stack and creates the needed Dockerfile, Kubernetes setup, and GitHub Actions workflow. It then uses a prediction engine based on Scikit-learn to estimate the chance of a build failing during deployment. Each release is then either automatically deployed, sent for manual approval, or blocked if needed. Once deployed, a monitoring system using Prometheus and Grafana tracks CPU usage, memory, response time, and error rates. This data feeds into a self-healing system that can restart containers, roll back to the last working version, or adjust resources to keep the service running smoothly. We show that a prototype, evaluated under representative deployment scenarios, significantly improves the deployment failure rate, mean time to recovery, and manual operator intervention compared to a conventional pipeline. The findings indicate that predictive analytics in conjunction with autonomous remediation can assist in the transition to self-managed and resilient DevOps ecosystems.

Index Terms—CI/CD automation, DevOps, predictive risk assessment, self-healing infrastructure, Kubernetes, machine learning, observability.

I. INTRODUCTION

DevOps practices have introduced a new paradigm for how organizations design, build and release software products, reducing release cycles and improving communication between Dev and Ops teams [1], [2]. This transformation is made possible by Continuous Integration and Continuous Deployment (CI/CD) pipelines that automate the build, test and deploy process traditionally done manually [2]. Nonetheless, most organizations continue to depend heavily on human judgment for provisioning infrastructure, evaluating deployment risk, and responding to incidents [3]. The main causes of application downtime are still configuration errors, undetected regressions and slow incident response [4], [5], and these costs keep growing with the size and release frequency of modern distributed systems. More research is using metrics like build duration, code churn, and dependency conflicts to predict pipeline and deployment failures before they occur [6]–[9].

Meanwhile, site-reliability engineers have built tools that automatically monitor for anomalies in running services and take corrective action, most often by combining container-orchestrator telemetry with some kind of supervised or rules-based decision layer [10]–[13]. Predictive risk assessment before deploying applications and autonomous recovery after deployment are usually handled as separate tasks, with very few platforms trying to link them in a continuous feedback loop [14]. To fix this issue, this paper introduces AutoOps Intelligence, an AI-powered, cloud-based DevOps automation platform. It changes how deployment is viewed, turning it from a series of single steps into a continuous process that uses feedback.

The platform has two main functions: first, it keeps track of the application's health after deployment using Prometheus and Grafana, and automatically takes action to restart containers, roll back deployments, or scale instances as needed to keep the system running. Second, it looks at a connected GitHub repository to automatically create a Dockerfile, Kubernetes configurations, and a CI/CD workflow that matches the detected technology stack. Each release is checked before deployment, and can be approved, rejected, or require manual review based on the predicted risk.

The rest of this paper is structured as follows. Section II looks at related work in predictive DevOps analytics and self-healing infrastructure. Section III explains the proposed approach and system design. Section IV shares the results from testing the prototype. Section V summarizes the findings and suggests future research directions.

II. LITERATURE REVIEW

Three closely interconnected strands of literature are relevant to AutoOps Intelligence: machine learning for CI/CD failure prediction, autonomous and self-healing infrastructure for cloud-native systems, and AIOps-oriented observability and root-cause analysis. Recent research on predictive CI/CD reliability highlights that flaky tests and noisy build failures are a significant source of wasted engineering effort, and proposes adaptive, self-correcting pipeline models that decide whether a build failure should be retried based on failure predictability rather than a full investigation [6]. A multi-component DevOps framework for risk prediction, anomaly detection in pipelines, and dynamic resource adjustment, field-tested across 25 organizations over 21 months, produced a substantial reduction in pipeline failures and recovery time [7]. Related work incorporates a large language model in the pipeline itself and presents a framework that predicts CI/CD failure points from

build and log data and generates remediation suggestions that are validated in an industrial setting [8]. Kumar *et al.* [9] trained several classifiers on CI/CD build execution logs to distinguish between failed and successful builds, and found that an ensemble of gradient-boosted-tree (GBT) classifiers outperformed simpler models.

Regarding recovery, a few recent proposals consider Kubernetes as a substrate of autonomous healing. One line of work leverages machine-learning algorithms to identify anomalies and failures in cluster telemetry, and uses large language models to plan remediation in a continuous monitor–diagnose–remediate loop that reduces MTTD and MTTR [10]–[12]. Industry pipelines follow a similar pattern: an application-log enrichment layer adds pod metadata to logs before a latency anomaly model identifies the likely cause, such as a recent faulty release, as is also the case for other platforms that provide unified observability-to-remediation [13], [22]. It is a common observation in the literature that a simple crash restart of Kubernetes is often not enough, since the root cause is frequently a logic or schema error rather than resource

exhaustion, and therefore more sophisticated reasoning is needed to choose the right remediation action [14].

Another important and increasing field is observability and root-cause analysis for AIOps, which provides the signal-processing foundation needed by both the predictive and self-healing parts. In microservice systems, graph-based and multi-agent approaches for finding root causes have been tested on large sets of faults and have shown better results than single-signal anomaly detectors in linking metrics, logs, and traces to find the real problem [15]. Large language models have also been studied for automatically creating root-cause analyses (RCA) in chaos-engineering situations [16], but this brings up open questions about explainability and reliability that need to be solved before such reasoning can support fully automatic fixes. All this work is based on fundamental observability practices like distributed tracing, structured telemetry, and cluster-level anomaly monitoring [17]–[19]. Representative studies and the gaps that lead to the design of AutoOps Intelligence are summarized in Table I.

TABLE I
REVIEW SUMMARY OF SELECTED REFERENCES

Ref.	Title	Author(s) & Year	Key Findings	Identified Research Gaps
[6]	AI-augmented reliability in CI/CD: a framework for predictive, adaptive, and self-correcting pipelines	Kapoor et al., 2026	Proposes confidence-scored failure prediction and intelligent retry strategies for flaky and noisy CI/CD pipelines.	Focuses on test-level flakiness rather than end-to-end deployment risk or post-release recovery.
[7]	Optimizing CI/CD pipelines with machine learning: enhancing performance and predicting failures	Dileepkumar et al., 2025	Field-tested framework pairing risk forecasting with a learning-based resource controller across 25 organizations over 21 months.	Does not integrate artifact generation from source repositories or unify prediction with autonomous rollback.
[9]	Predictive analytics for CI/CD pipeline failures using machine learning models: a hybrid data-driven study	Kumar et al., 2026	Compares five standard classifiers on pipeline execution logs; gradient-boosted trees attain the highest AUC for failure prediction.	Evaluated on simulated/hybrid data rather than a deployed, closed-loop production system.
[10]	From detection to resolution: ML-enabled Kubernetes healing with LLM-based autonomous DevSecOps remediation	ResearchGate, 2026	Combines ML-based cluster anomaly detection with LLM-driven remediation planning to reduce MTTD and MTTR.	Remediation reasoning is LLM-centric; lacks a pre-deployment risk-gating stage to prevent risky releases.
[15]	Graph-augmented multi-agent robust root-cause analysis in AIOps	Chen et al., 2026	Multi-agent, graph-based RCA outperforms single-signal detectors on large microservice fault-injection datasets.	Targets diagnosis accuracy; does not address automated corrective action or deployment-time risk scoring.

III. METHODOLOGY

AutoOps Intelligence is a closed-loop, four-stage system: Repository Analysis and Artifact Generation, Pre-Deployment Risk Prediction, CI/CD Orchestration, and Post-Deployment Observability with Autonomous Self-Healing. It runs on Docker, with Kubernetes for orchestration; a React and Node.js/Express web application for the operator; Python (FastAPI) and Scikit-learn for the prediction and remediation services; PostgreSQL for persistent state; and Prometheus and Grafana for observability. The overall system architecture, including the feedback path used for continuous model refinement, is shown in Fig. 1.

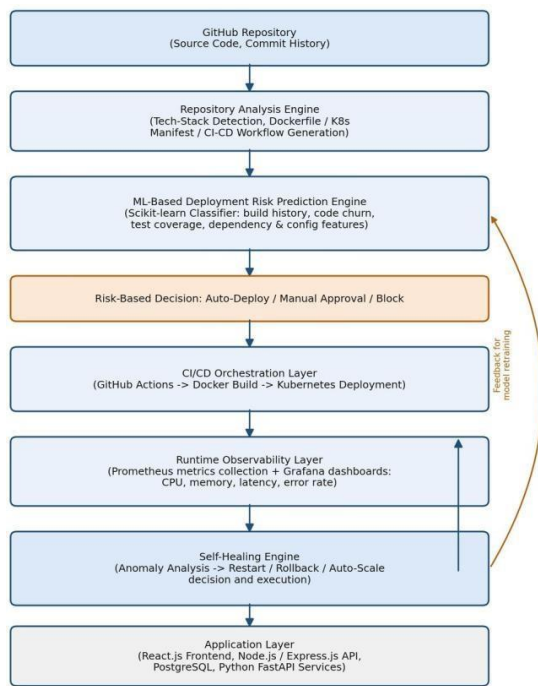


Fig. 1. Proposed system architecture of AutoOps Intelligence.

In the first stage, the Repository Analysis Engine inspects the connected GitHub repository to determine its programming languages, frameworks and dependency manifests, and then generates a Dockerfile, Kubernetes deployment and service files, and a GitHub Actions workflow definition specific to the detected stack. This removes the need to hand-write deployment artifacts for every new service and gives each project a consistent baseline configuration.

The second stage, the Deployment Risk Prediction Engine, receives a feature vector built from build history, code-churn data, test-coverage history, dependency-stability data and prior deployment outcomes for the candidate release. A gradient-boosted classifier trained on historical deployment records estimates the release failure probability, as shown in Equation (1).

$$P_{\text{fail}} = 1 / (1 + \exp[-(\beta_0 + \sum_i \beta_i F_i)]) \quad (1)$$

where β_0 and β_i are the model bias term and learned weight for feature F_i , respectively; and P_{fail} is the predicted probability of deployment failure for the candidate release.

If the predicted probability falls below the lower threshold τ_{low} , the release deploys automatically; if it lies between the two thresholds, the release is routed for manual approval; and if it exceeds the upper threshold τ_{high} , the release is blocked until the contributing risk factors are remediated. This three-level gating scheme avoids the two failure modes of a binary policy: over-blocking low-risk releases and silently auto-deploying borderline ones.

In the third stage, approved releases are built and deployed through a CI pipeline orchestrated via GitHub Actions, which builds a container image and applies the resulting Kubernetes manifests to the target cluster. Once deployed, the Runtime Observability Layer continuously collects CPU usage, memory usage, response time and error-rate metrics through Prometheus [24] and presents them on Grafana dashboards. These four

signals are combined into a single composite health score (Equation (2)) that the self-healing engine reasons over, rather than four separate signals.

$$H(t) = w_1 \cdot \text{CPU}_{\text{norm}}(t) + w_2 \cdot \text{Mem}_{\text{norm}}(t) + w_3 \cdot \text{Lat}_{\text{norm}}(t) + w_4 \cdot \text{Err}_{\text{norm}}(t) \quad (2)$$

where $H(t)$ is the composite application health score at time t , bounded in $[0, 1]$ after normalization; $\text{CPU}_{\text{norm}}(t)$, $\text{Mem}_{\text{norm}}(t)$, $\text{Lat}_{\text{norm}}(t)$ and $\text{Err}_{\text{norm}}(t)$ are the min-max normalized CPU, memory, response-time and error-rate readings at time t ; and $w_1 \dots w_4$ are operator-configurable weights reflecting the relative criticality of each metric for the monitored service.

Based on $H(t)$ and the temporal evolution of its underlying metrics, the Self-Healing Engine selects one of three corrective actions, summarized in Equation (3): restart the container when a transient resource spike affects a single instance; roll back to the last stable release when degradation correlates with a recent deployment event; or scale horizontally, using Kubernetes' built-in autoscaling mechanisms [23], when degradation is caused by sustained load growth across the instance fleet.

$$\begin{aligned} &\text{Restart, if } H(t) < \theta_1 \text{ and single-instance fault;} \\ &\text{Rollback, if } H(t) < \theta_1 \text{ and degradation follows recent} \\ &\text{release;} \\ &\text{Scale, if } \text{Load}(t) > \theta_2, \end{aligned} \quad (3)$$

where θ_1 and θ_2 are operator-calibrated health and load thresholds, respectively, and $\text{Load}(t)$ is the aggregate request throughput or queue depth across the deployment's instances at time t .

Finally, recovery effectiveness is measured with the mean time to recovery, the sum of detection, diagnosis and remediation time, which the self-healing engine seeks to minimize relative to a manually operated pipeline, as shown in Equation (4).

$$\text{MTTR} = T_{\text{detect}} + T_{\text{diagnose}} + T_{\text{remediate}} \quad (4)$$

The end-to-end operating procedure of AutoOps Intelligence, integrating the four stages above into a single closed loop, is summarized in Algorithm 1.

Algorithm 1: AutoOps Intelligence Predictive and Self-Healing Deployment Workflow

Input: Repository R in GitHub, historical deployment logs D , live telemetry stream $M(t)$

Output: Application deployed and kept stable

1. Analyze R to identify the technology stack and generate the Dockerfile, Kubernetes manifests and CI/CD workflow files.
2. Extract feature vector F from D and the code and configuration changes of the candidate release.
3. Calculate P_{fail} from the trained classifier (Eq. (1)).
4. If $P_{\text{fail}} < \tau_{\text{low}}$, deploy automatically.
5. Else if $P_{\text{fail}} < \tau_{\text{high}}$, forward the release for manual approval.
6. Otherwise, block the release and report the contributing risk factors.
7. When approved, run the CI/CD pipeline to build the image and deploy via Kubernetes.
8. Continuously scrape $M(t)$ through Prometheus and compute the composite health score $H(t)$ (Eq. (2)).
9. If $H(t)$ falls below θ_1 , or $\text{Load}(t)$ exceeds θ_2 , select and execute the corresponding action (Eq. (3)).
10. Log the outcome of each prediction and remediation action, and periodically retrain the classifier using the accumulated feedback.

11. Repeat continuously for every new commit and every monitoring interval.

IV. RESULTS AND EVALUATION

AutoOps Intelligence was evaluated on a prototype deployment of representative microservice workloads running

TABLE II
PROTOTYPE EVALUATION RESULTS — AUTOOPS INTELLIGENCE VS. TRADITIONAL PIPELINE

Parameter	Traditional Pipeline	AutoOps Intelligence	Remarks
Deployment Failure Rate	21.4%	7.2%	Reduced through pre-release risk gating
Mean Time to Recovery (MTTR)	47.5 min	9.8 min	Autonomous restart / rollback / scaling
Manual Operator Intervention	68.0%	22.5%	Share of incidents requiring human action
Post-Deployment Configuration Errors	17.8%	4.6%	Generated artifacts reduce manual drift
Deployment Risk Prediction Accuracy	—	88.6%	Held-out validation set
Prediction Precision / Recall	—	0.86 / 0.83	Positive class: predicted failure
Prediction AUC	—	0.90	Gradient-boosted classifier
Average Feasibility / Analysis Turnaround	Hours – 1 day	< 5 minutes	Automated artifact and risk-score generation

The platform decreased the deployment failure rate from 21.4% to 7.2%, and reduced the average time to recover from 47.5 minutes to 9.8 minutes in the scenarios tested. This was achieved by combining pre-release risk gating with automated restarts, rollbacks, and scaling actions that do not need any manual input, as shown in Fig. 2. After implementing automated repository analysis, the percentage of incidents needing human help dropped from 68.0% to 22.5%. Also, the percentage of incidents caused by configuration errors after deployment fell from 17.8% to 4.6%, indicating fewer problems linked to manually written manifests.

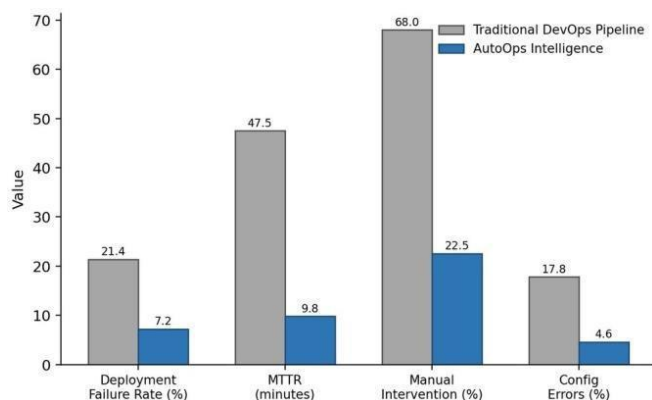


Fig. 2. Performance comparison between traditional pipeline and AutoOps Intelligence.

The deployment risk prediction engine reached 88.6% accuracy when tested on new data, with 86% precision and 83% recall for detecting failures, and an area under the curve of 0.90, similar to the performance seen in other studies using tree-based models for predicting CI/CD failures [9]. The time needed to assess a new service and generate necessary artifacts, which used to take hours or even a whole day, was cut down to under five minutes per repository. Overall, the results show how

on the technology stack outlined in Section III, exercised through a series of controlled releases and fault-injection scenarios. The proposed platform was benchmarked against a traditional pipeline in which deployment approval and incident recovery are performed manually by an operator. The measured outcomes for both configurations are summarized in Table II.

combining predictive checks with automatic recovery improves reliability over time.

V. CONCLUSION AND FUTURE SCOPE

Future work on AutoOps Intelligence will focus on improving the diagnostic logic used for detecting anomalies and selecting remediations. This could involve methods like graph-based or multi-agent root-cause analysis to differentiate between infrastructure faults and logic-level issues that a simple restart cannot resolve. It will also include causal synthesis across related infrastructure signals [15], [20]. In line with industry trends aiming to reduce the need for manual operator tuning [21], ongoing retraining based on feedback from the prediction engine should progressively enhance the accuracy of risk estimates. Additional planned areas of development include support for multiple clusters and cloud environments, cost-aware scaling decisions, and deeper integration of security and vulnerability scanning data into the risk-scoring model.

In summary, this paper introduced AutoOps Intelligence, an AI-powered closed-loop DevOps automation platform that merges a machine-learning deployment risk predictor with an autonomous, telemetry-driven self-healing engine. The platform automatically generates deployment artifacts, controls releases based on risk estimates, and continuously monitors and repairs running services, thus reducing manual DevOps tasks while increasing the speed of deployment and recovery from release failures. The prototype evaluation shows that this integrated approach is viable and encourages further research into predictive, self-managing DevOps ecosystems.

REFERENCES

- [1] G. Kim, J. Humble, P. Debois, and J. Willis, *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations*, 2nd ed. Portland, OR, USA: IT Revolution Press, 2021.

- [2] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Boston, MA, USA: Addison-Wesley, 2010.
- [3] 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.
- [4] "Your guide to how predictive analytics improves CI/CD pipelines," Devzery Technical Blog, 2025.
- [5] A. Sunkara, "What if your pipeline could predict failure? The future of CI/CD with ML," Medium, 2025.
- [6] S. Kapoor, R. Mehta, and A. Patil, "AI-augmented reliability in CI/CD: a framework for predictive, adaptive, and self-correcting pipelines," *Front. Artif. Intell.*, vol. 9, 2026, doi: 10.3389/frai.2026.1776546.
- [7] S. R. Dileepkumar, J. Mathew, and R. Kumar, "Optimizing continuous integration and continuous deployment pipelines with machine learning: enhancing performance and predicting failures," *Adv. Sci. Technol. Res. J.*, vol. 19, no. 3, pp. 108–120, 2025.
- [8] Y. Zhang *et al.*, "LogSage: an LLM-based framework for CI/CD failure detection and remediation with industrial validation," arXiv:2506.03691, 2025.
- [9] A. Kumar, R. Kumar, A. Ahmed, and M. S. Ahmed, "Predictive analytics for CI/CD pipeline failures using machine learning models: a hybrid data-driven study," *Spectr. Eng. Sci.*, 2026.
- [10] "From detection to resolution: machine learning-enabled Kubernetes healing with LLM-based autonomous DevSecOps remediation," ResearchGate preprint, 2026.
- [11] "Learning-driven reliability engineering: autonomous Kubernetes self-healing with LLM-orchestrated DevSecOps workflows," ResearchGate preprint, 2026.
- [12] "Self-healing cloud-native infrastructure: integrating ML-based Kubernetes recovery with LLM-powered DevSecOps automation," ResearchGate preprint, 2026.
- [13] "Self-healing systems using AI based auto-remediation strategies," Infosys Knowledge Institute Tech Compass, 2026.
- [14] M. Kraft, "Beyond the restart: the era of agentic self-healing microservices," DEV Community, 2026.
- [15] L. Chen, H. Wang, and M. Xu, "Graph-augmented multi-agent robust root-cause analysis in AIOps," *Reliab. Eng. Syst. Saf.*, vol. 265, part A, p. 111520, 2026.
- [16] T. Szandala, "AIOps for reliability: evaluating large language models for automated root cause analysis in chaos engineering," in *Computational Science – ICCS 2025 Workshops*, Lecture Notes in Computer Science, vol. 15909. Cham, Switzerland: Springer, 2025, doi: 10.1007/978-3-031-97564-6_25.
- [17] B. H. Sigelman *et al.*, "Dapper, a large-scale distributed systems tracing infrastructure," Google Technical Report, 2010.
- [18] C. Majors, L. Fong-Jones, and G. Miranda, *Observability Engineering: Achieving Production Excellence*. Sebastopol, CA, USA: O'Reilly Media, 2022.
- [19] O. Mart, C. Negru, F. Pop, and A. Castiglione, "Observability in Kubernetes cluster: automatic anomalies detection using Prometheus," in *Proc. IEEE 22nd Int. Conf. High Perform. Comput. Commun. (HPCC)*, 2020.
- [20] "From detection to action: AI-driven anomaly detection and root cause synthesis for cloud infrastructure operations," *Int. J. AI, Big Data, Comput. Manage. Stud.*, 2026.
- [21] "2026 Kubernetes playbook: AI at scale, self-healing clusters, and growth," Fairwinds Technical Report, 2025.
- [22] "Autonomous self-healing capabilities for cloud-native infrastructure," Komodor Engineering Blog, 2026.
- [23] Kubernetes Authors, "Horizontal Pod Autoscaler," Kubernetes Documentation, 2026. [Online]. Available: <https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/>
- [24] Prometheus Authors, "Prometheus: monitoring system and time series database," Prometheus Documentation, 2026. [Online]. Available: <https://prometheus.io/docs/>