iJRASET
International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 14 **Issue:** VII **Month of publication:** July 2026

DOI: https://doi.org/10.22214/ijraset.2026.84054

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 VII July 2026- Available at www.ijraset.com

A Closed-Loop Agentic AI Framework for Self - Configuring, Self-Healing, and Optimized CI/CD Automation

A. Hitesh Reddy[1], A. Jitendra[2]
[1, 2]*Department of Computer Science and Engineering, Holy Mary Institute of Technology and Science, Hyderabad, India*

Abstract: ***Static continuous integration and continuous delivery pipelines remain difficult to adapt when repositories, dependencies, deployment targets, and operational conditions change. Manual configuration also slows failure diagnosis and prevents pipelines from learning from previous executions. This paper presents DevOps-Pilot, a closed-loop agentic AI framework for intelligent DevOps automation. The framework analyses repository context, detects languages, dependency files, build tools, test frameworks, Docker assets, and Kubernetes manifests, and then uses LLM-assisted planning to generate Jenkins-compatible pipeline definitions. Pipeline execution is monitored through a Flask dashboard and SQLite-backed history store, while anomaly detection and self-healing logic recommend recovery actions for installation, testing, and deployment failures. A reinforcement learning-inspired optimisation policy updates cache, parallel testing, retry, and timeout settings from observed rewards. The academic prototype was evaluated in simulation mode across four repositories and nine pipeline runs. Results show six successful and three failed runs, a success rate of 66.67%, an average execution time of 120.52 seconds, and an average reward of 3.2641. The findings indicate the feasibility of closed-loop CI/CD automation for adaptive software delivery.***

I. INTRODUCTION

Agile development, DevOps practices, and cloud-native deployment have shortened release cycles and made continuous integration and continuous delivery (CI/CD) pipelines central to modern software delivery. Continuous delivery emphasises reliable release automation [1], while DevOps connects development, operations, and architecture concerns across the delivery lifecycle [2]. In practice, however, many pipelines are still maintained as static scripts tied to a particular repository, build tool, or deployment environment.

Manually configured pipelines create four recurring prob-lems. First, build and deployment logic is often dupli-cated across repositories and becomes stale when dependencies change. Second, failure diagnosis relies on engineers reading logs and applying tool-specific knowledge. Third, scripts for Jenkins, GitHub Actions, GitLab CI, Docker, or Kubernetes are rarely connected to a shared execution memory. Finally, con-ventional automation executes instructions but does not learn which policy choices improve reliability or speed.

These limitations become more visible in polyglot reposito-ries. A small organisation may operate Python Flask services, Node.js dashboards, Java APIs, containerised workloads, and Kubernetes manifests at the same time. A static Jenkinsfile that works for one service can be unsuitable for another be-cause dependency installation, test discovery, build artefacts, and deployment commands differ. Engineers therefore spend effort on pipeline maintenance instead of feature delivery, and knowledge about previous failures remains fragmented across build logs, chat messages, and issue trackers.

The central research question addressed in this paper is whether CI/CD automation can be treated as a closed-loop decision process rather than as a one-time script-generation task. In such a process, repository evidence is converted into a pipeline plan, execution produces observations, failures trigger recovery, and outcomes update future planning. This view is especially suitable for agentic AI because agents can decom-pose pipeline work into perception, planning, action, monitor-ing, and learning steps.

DevOps-Pilot addresses these issues through a closed-loop agentic AI framework. The system accepts a repository path or URL, extracts repository context, plans a Jenkins-compatible pipeline, orchestrates execution, monitors run-time behaviour, records execution history, applies self-healing recommenda-tions, and updates an optimisation policy. The prototype delib-erately includes a simulation mode so that academic evaluation can be performed without requiring a full production Jenkins, Docker, and Kubernetes installation.

The contributions of this work are: context-aware reposi-tory analysis for language, dependency, build, test, container, and Kubernetes cues; LLM-assisted Jenkinsfile generation grounded in detected project context; closed-loop monitoring and feedback; self-healing recovery for common install, test, and deploy failures; reward-based optimisation of cache, paral-lelism, retry, and timeout settings; and a Flask prototype with dashboard, REST API, SQLite storage, and experimental vali-dation.

II. RELATED WORK

Traditional CI/CD platforms such as Jenkins, GitHub Actions, and GitLab CI provide mature mechanisms for pipeline execu-tion, plugin integration, and repository-triggered automation. Jenkins remains widely used because of its extensible pipeline model and documentation ecosystem [3]. Container and or-chestration platforms, especially Docker and Kubernetes, have also become foundational for reproducible builds, image delivery, and cloud-native rollout management [4, 5].

These tools are powerful but generally expect the pipeline author to define the correct build, test, and deploy stages. They execute pipeline logic efficiently, yet they do not automatically infer repository context, rewrite stages after failure, or compare a failed run with previous reward patterns. Template repositories and shared libraries partially reduce duplication, but they still depend on human selection and maintenance.

AIOps research connects monitoring, anomaly detection, log analysis, event correlation, and remediation. Reinforce-ment learning provides a formal basis for policy improve-ment from delayed rewards [6], and multi-agent systems pro-vide coordination concepts for autonomous software agents [7]. These foundations are relevant to CI/CD systems because pipeline failures are operational events with measurable out-comes, costs, and feedback signals.

Recent work on LLMs for code and software engineering shows that language models can support generation, repair, documentation, and workflow automation [8, 9]. LLM-based software agents extend this capability by combining planning, tool use, and interaction with external systems [10]. AIOps surveys further show that LLMs are being explored for failure data interpretation, root-cause analysis, and operational deci-sion support [11, 12].

LLM-based pipeline generation alone is insufficient for de-pendable automation because a generated Jenkinsfile may be syntactically plausible but operationally unsuitable. The gen-eration step must be constrained by repository evidence and evaluated through execution feedback. DevOps-Pilot there-fore positions the LLM-assisted planner as one component in a broader control loop, not as a stand-alone oracle.

The research gap is that existing systems usually address pipeline execution, monitoring, or optimisation separately. Very few systems provide a unified closed-loop framework that combines repository context analysis, LLM-assisted CI/CD generation, monitoring, self-healing, execution history, and re-inforcement learning-inspired optimisation.

III. PROPOSED CLOSED-LOOP AGENTIC CI/CD FRAMEWORK

The proposed DevOps-Pilot framework is organised as a set of cooperating agents and services. The Repository Input module receives project locations and metadata. The Con-text Analysis Agent detects language, dependency files, build tools, test frameworks, Dockerfiles, and Kubernetes manifests. The Pipeline Planning Agent transforms that context into a Jenkins-compatible pipeline plan using LLM-assisted or rule-based generation. The Orchestration Layer coordinates execu-tion stages, while Execution Agents interface with simulated or optional Jenkins, Docker, and Kubernetes adapters.

The Monitoring Module observes status, duration, failure stage, and deployment outcome. The Self-Healing Mod-ule maps common failure stages to recovery recommenda-tions, such as reinstalling dependencies, rerunning tests, rolling back deployments, or pausing rollout. The Execution History Database stores repositories, pipelines, runs, rewards, and pol-icy updates. The Optimisation Agent updates the execution policy and feeds it back to planning and orchestration. Figure 1 shows the overall architecture.

The agents communicate through structured intermediate records rather than through unstructured logs alone. A reposi-tory context record stores detected files and inferred technology choices. A pipeline plan record stores generated stages and op-tional deployment targets. An execution record stores status, execution time, failed stage, deployment result, and reward. This separation allows each module to be tested independently and also makes the framework explainable for conference eval-uation.

Self-configuration occurs when the planner selects stages from repository evidence. Self-healing occurs when the system interprets a failed stage and recommends recovery. Optimisa-tion occurs when the policy is updated from accumulated re-wards. These three behaviours form the closed-loop capability of the proposed framework: the system configures itself before execution, reacts during failure, and modifies later choices after observing outcomes.

The full closed-loop flow is Repository Input → Context Analysis → Pipeline Planning → Orchestration → Execution → Monitoring → Self-Healing → Execution History → Opti-misation → Feedback to Planning and Orchestration. Figure 2 presents this control loop in compact form.

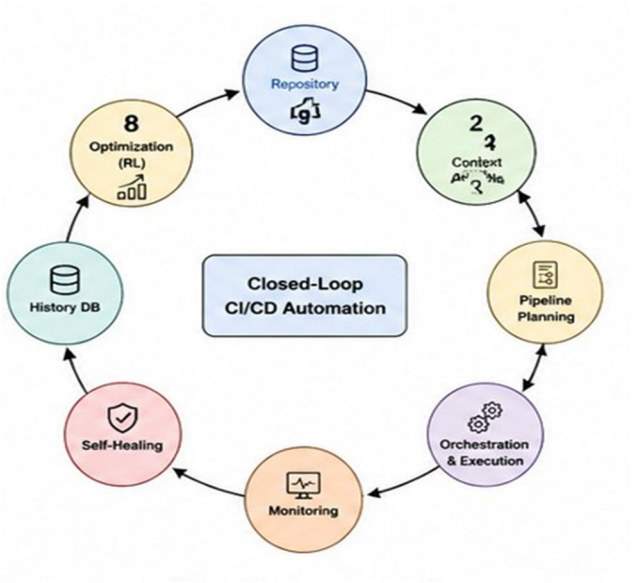


Figure 2: Closed-loop operation flow.

For example, when inventory-api is analysed, dependency and deployment cues indicate an API service that can be in-stalled, tested, packaged, and deployed to a staging namespace. If the Test stage fails, the self-healing module records that de-ployment should be skipped and the run should receive a neg-ative reward. If a later run succeeds with a shorter time, the policy receives positive evidence for the selected retry, time-out, cache, and parallel-test configuration.

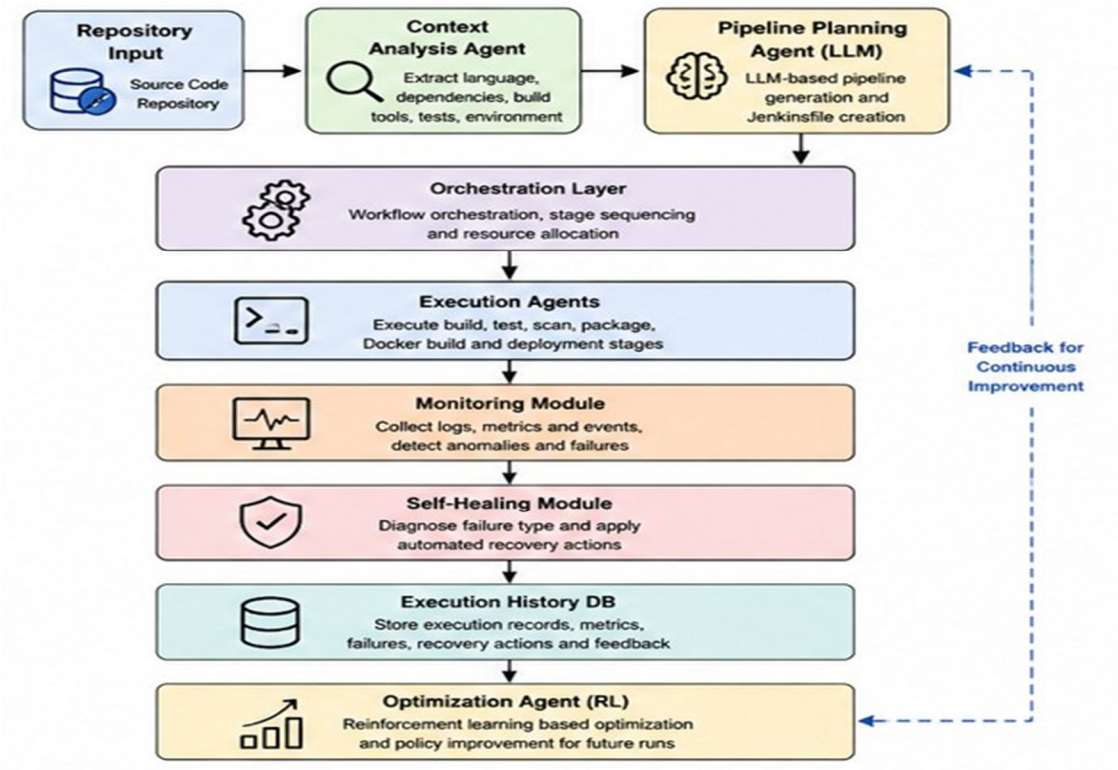


Figure 1: Architecture of the proposed DevOps-Pilot framework.

IV. METHODOLOGY

The methodology converts repository input into a technical context, uses that context for pipeline generation, executes the generated stages, and then records monitoring, self-healing, and reward feedback for subsequent optimisation. Repository input is first analysed to identify language, dependencies, build tools, test frameworks, and deployment cues. The orchestra-tion layer then executes the plan, monitoring detects anomalies, self-healing applies recovery actions, and the history database records both failures and successful outcomes for later policy updates.

Repository context is represented as

$$C = \{l,d,b,t,e\}, \quad (1)$$

where l is the programming language, d is the dependency file, b is the build tool, t is the testing framework, and e is the de-ployment environment. The analyser maps package.json to Node.js/npm, requirements.txt to Python/pip, pyproject.toml to Python, pom.xml to Java/Maven, build.gradle to Java/Gradle, Dockerfile to Docker support, and deployment.yaml, service.yaml, or k8s/*.yaml to Kubernetes support.

The context representation is intentionally compact because it must fit inside a planner prompt, a database row, and a dash-board summary. Additional attributes such as repository name, branch, detected framework, Docker image tag, Kubernetes namespace, and previous reward can be attached as metadata without changing the core tuple. This design keeps the aca-demic model simple while allowing practical extension.

Pipeline generation is represented as

$$P = f_{\mathrm{LLM}}(C). \quad (2)$$

In the prototype, f_{LLM} may be realised through an LLM-assisted planner or a deterministic rule-based planner. Both modes are grounded in the same repository context so that gen-erated Jenkins stages remain traceable to detected project evi-dence. The generated pipeline is organised into checkout, in-stall, build, test, package, and deploy stages where applicable, as shown in Figure 3. A Python repository with requirements.txt receives pip-based dependency installation, whereas a Node.js repos-itory with package.json receives npm installation and script execution. Docker support is enabled only when a Dock-erfile is detected, and Kubernetes deployment is enabled only when manifest evidence is present. This prevents the planner from adding unsupported stages merely because they are com-mon in generic CI/CD examples.

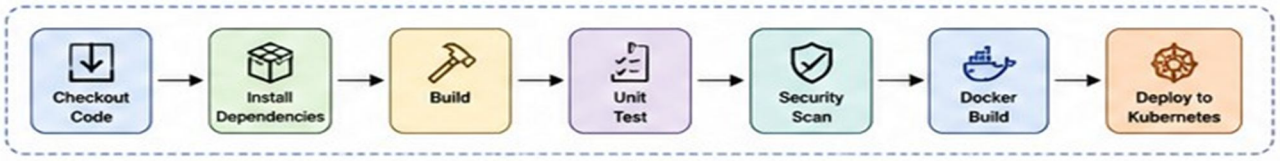


Figure 3: Pipeline execution stages.

Execution output is represented as

$$E = \{s,t,r\}, \quad (3)$$

where s is execution status, t is execution time, and r is reward or performance feedback. A compact anomaly condition is

$$|t - \mu_t| > \alpha\sigma_t, \quad (4)$$

where μ_t is historical mean execution time, σ_t is standard de-viation, and α is a threshold parameter. The reward function is

$$R = -\lambda_1 T - \lambda_2 F + \lambda_3 S, \quad (5)$$

where T is execution time, F is failure count, S is the success indicator, and λ_1, λ_2, λ_3 are weighting parameters.

The self-healing workflow in Figure 4 begins when a fail-ure is detected. The failure is classified by stage, a recovery strategy is selected, an action is applied, and the failed stage is re-executed. If the failure persists, the system returns to classi-fication or escalates for manual review.

The reward function is not presented as a final production formula. It is a prototype scoring mechanism that penalizes long execution time and failure while rewarding successful completion. In a real deployment, the terms can be extended to include resource cost, security scan status, rollout risk, service-level objectives, and manual approval requirements.

Anomaly detection is applied conservatively. A long execu-tion time alone does not imply failure, but it indicates that a pipeline may need a timeout update, dependency cache, or test parallelisation. Similarly, repeated failures at the same stage indicate that an automated retry may be less useful than escalation. This distinction prevents the system from repeatedly applying the same recovery action when the underlying application or environment needs human attention.

V. IMPLEMENTATION

DevOps-Pilot was implemented as a Python Flask backend with a Bootstrap dashboard and SQLite storage. The backend exposes dashboard pages and REST API endpoints for repos-itory submission, pipeline creation, execution history, and op-timisation policy inspection. The pipeline output is Jenkins-compatible, while execution can run in simulation mode or in-tegrate with Jenkins, Docker, and Kubernetes through adapters when those environments are configured.

The implementation is modular. The repo analyzer.py component extracts repository context; pipeline planner.py builds the pipeline plan; pipeline executor.py runs or simulates stages; monitor.py records run-time observations; anomaly detector.py identifies abnormal duration or failed stages; self healer.py selects recovery ac-tions; optimizer.py updates policy parameters; and jenkins adapter.py, docker adapter.py, and kubernetes adapter.py isolate external tool integration.

SQLite was selected for the academic prototype because it provides a lightweight persistent store without requiring server administration. The database stores repository records, gener-ated pipeline metadata, execution history, and the current opti-misation policy. The data path is intentionally simple: reposi-tory input is analysed by DevOps-Pilot, execution records are stored or retrieved from the history database, and dashboard reports expose the resulting insights.

Table 1: Technology Stack

Component	Technology	Purpose
Programming Language	Python	Backend services and automation
Web Framework	Flask	Dashboard and REST APIs
Database	SQLite	Execution history and policy storage
CI/CD Tool	Jenkins	Pipeline generation and optional execution
Containerisation	Docker	Image build support
Orchestration	Kubernetes	Deployment support
Frontend	Bootstrap	User interface
Optimisation	Reinforcement learning-inspired policy	Pipeline improvement

The Flask dashboard is designed as an operator-facing view rather than as a marketing interface. The Dashboard page sum-marises repository count, pipeline count, successful runs, failed runs, success rate, average execution time, and average reward. The Submit Repository page accepts new repository input. The Execution History page lists each run with status, time, failed stage, deployment result, and reward. The Optimisation Policy page displays cache, parallel-test, retry, timeout, update count, and average reward values.

Simulation mode is implemented to model realistic pipeline outcomes while avoiding dependency on external infrastruc-ture. Each simulated run still follows the same data path as a real run: the repository is analysed, a plan is selected, execution status is recorded, self-healing logic is invoked when needed, and the optimiser updates policy state. This ensures that replac-ing simulation with real adapters does not require changing the conceptual workflow.

Security-sensitive production features are intentionally treated as future integration points. Credential storage, secret masking, signed artefacts, vulnerability scanning, and approval gates should be connected through organisation-approved De-vSecOps controls. The prototype therefore demonstrates closed-loop automation without making unrealistic claims about unattended production release governance.

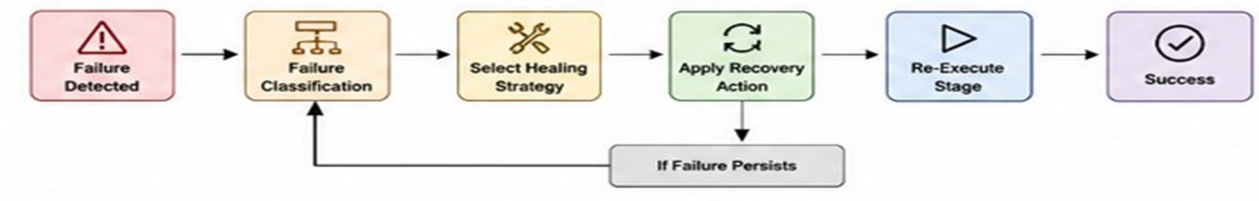


Figure 4: Self-healing workflow.

VI. RESULTS AND ANALYSIS

The prototype evaluation used four repositories and nine simu-lated pipeline runs. Figure 5 shows the main Flask dashboard, while Tables 2 and 3 report the numerical outcomes used for analysis.

Table 2: Dashboard Summary

Metric	Value
Repositories	4
Pipelines	4
Successful Runs	6
Failed Runs	3
Success Rate	66.67%
Average Execution Time	120.52 seconds
Average Reward	3.2641

The learned optimisation policy was {"enable cache": true, "parallel tests": true, "retry count": 2, "timeout seconds": 1500, "updates": 8}. The system recorded nine pipeline runs, with six successful and three failed runs. The observed success rate was 66.67%, the average execution time was 120.52 seconds, and the average reward was 3.2641.

Positive rewards were assigned to successful runs because successful deployment offsets execution-time cost. Negative rewards were assigned to failed runs because install, test, and deploy failures either prevented deployment or required recov-ery. The optimisation policy therefore enabled cache reuse, parallel tests, retry count, and timeout increase. These results show the feasibility of closed-loop CI/CD automation within the limits of an academic prototype.

The run distribution also illustrates why a closed-loop view is useful. inventory-api failed once at the Test stage and then later succeeded with a shorter execution time of 88.1 sec-onds. ops-dashboard failed at Install but later succeeded in 69.8 seconds after static assets were published. orders-service failed during Deploy because readiness probes prevented roll-out completion, yet a later run completed canary deployment. These transitions show that failure records are not only error reports; they are training signals for planning and recovery.

The highest positive reward in the recorded history was 9.302 for ops-dashboard after static assets were published in 69.8 seconds. The lowest reward was -6.843 for orders-service when deployment failed after 184.3 seconds. This contrast demonstrates how the reward signal captures both status and time cost. A successful but slow deployment receives a lower positive reward than a successful fast run, and a late deployment failure is penalised more strongly than a short install fail-ure.

Table 4: Self-Healing Mapping

Failure Stage	Failure Meaning	Suggested Recovery
Install	Dependency or environment failure	Clear cache and reinstall dependencies
Test	Test failure or application logic issue	Rerun tests or mark for manual review
Deploy	Deployment rollout/readiness issue	Rollback or pause rollout

Table 5: Comparison with Traditional CI/CD

Feature	Traditional CI/CD	Proposed DevOps-Pilot
Pipeline creation	Manual	Automated and context-aware
Failure handling	Manual debugging	Self-healing recommendation
Monitoring	Tool-specific	Integrated dashboard and history
Optimisation	Manual tuning	Reward-based policy update
Execution history	Scattered	Centralised SQLite storage
Adaptability	Limited	Feedback-driven

Average execution time should be interpreted carefully. A mean of 120.52 seconds across nine runs is useful for com-paring policy updates in the prototype, but production systems would require per-repository baselines because a frontend as-set pipeline and a canary-deployed backend service have dif-ferent normal durations. DevOps-Pilot supports this extension through execution history because each run is tied to a reposi-tory and stage outcome.

VII. DISCUSSION

The evaluation indicates that repository-aware pipeline plan-ning can improve adaptability because generated stages are based on detected project evidence rather than a fixed script. Execution history improves traceability by connecting each run to status, failed stage, deployment result, reward, and policy update. Reward values support optimisation because they con-vert execution outcomes into a compact feedback signal.

Self-healing reduces manual debugging for common failures by mapping install, test, and deploy failures to specific recov-ery choices. The simulation mode is also useful for academic validation because students and researchers can evaluate the closed-loop concept without complex infrastructure. However, real-world deployment requires deeper integration with Jenk-ins credentials, Docker registries, Kubernetes clusters, secret

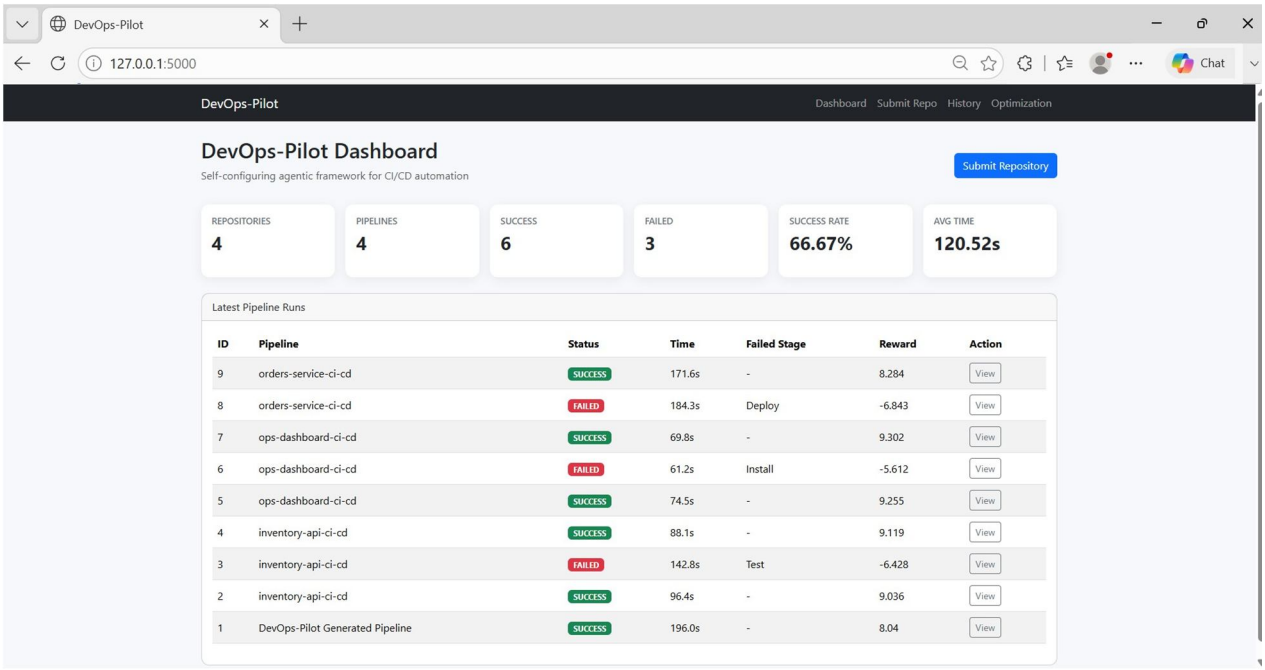


Figure 5: Dashboard screenshot.

Table 3: Execution History Results

Repository	Status	Time	Failed Stage	Deployment Result	Reward
flask	SUCCESS	196.0 s	-	deployment successful	8.040
inventory-api	SUCCESS	96.4 s	-	deployed to staging namespace	9.036
inventory-api	FAILED	142.8 s	Test	skipped after failed tests	-6.428
inventory-api	SUCCESS	88.1 s	-	deployed to staging namespace	9.119
ops-dashboard	SUCCESS	74.5 s	-	static assets published and deployment restarted	9.255
ops-dashboard	FAILED	61.2 s	Install	no deployment attempted	-5.612
ops-dashboard	SUCCESS	69.8 s	-	static assets published	9.302
orders-service	FAILED	184.3 s	Deploy	rollout paused due to readiness probe failures	-6.843
orders-service	SUCCESS	171.6 s	-	canary deployment completed	8.284

management, and organisation-specific policies.

The main limitations are that the prototype evaluation is simulation-based, more repositories are required for large-scale validation, test logic failures may require manual correction, and security scanning needs deeper DevSecOps integration. Future work should add real pipeline execution traces, richer anomaly models, security gates, and stronger human-in-the-loop controls.

VIII.CONCLUSION

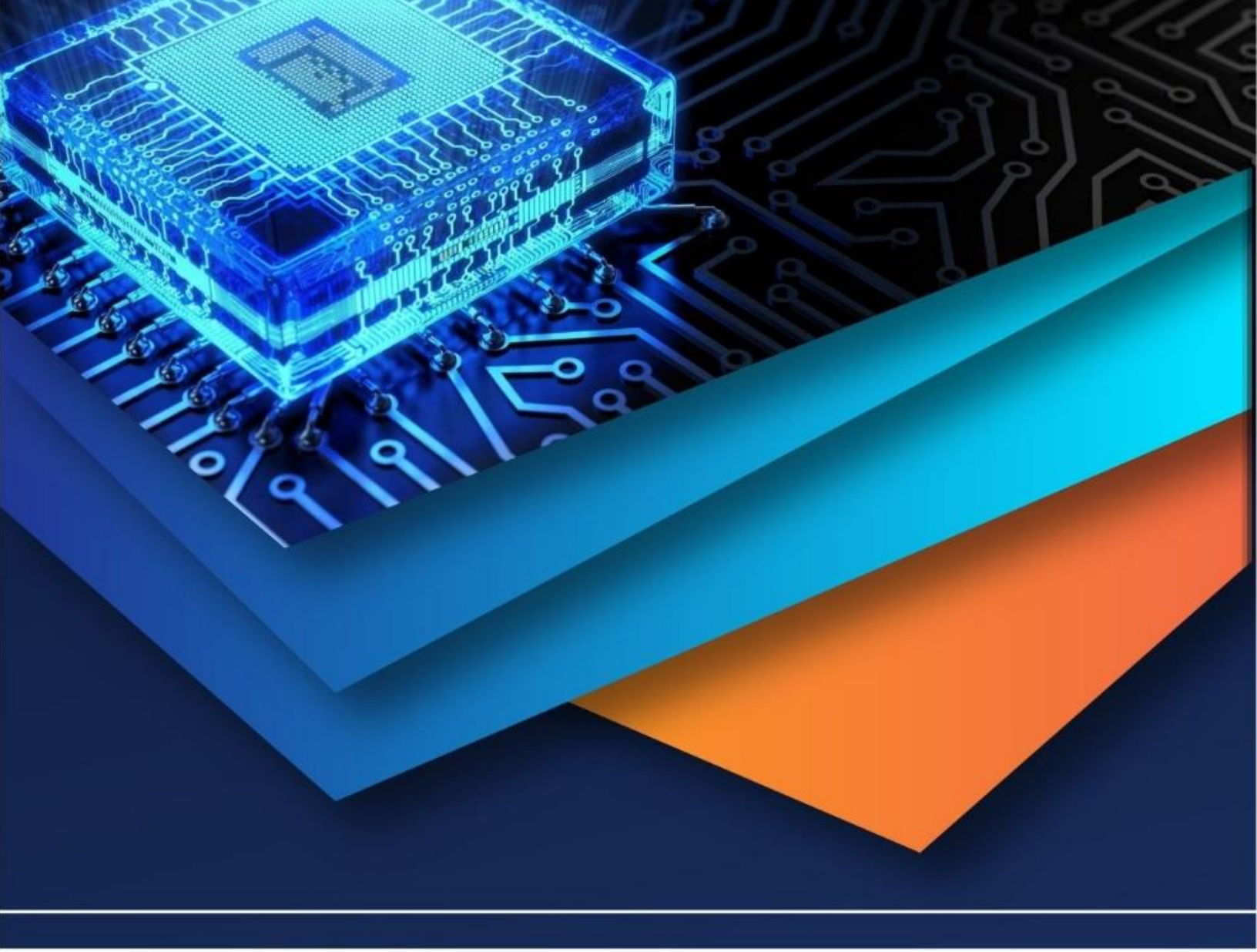
DevOps-Pilot demonstrates a closed-loop agentic AI frame-work for CI/CD automation. It integrates repository analysis, LLM-assisted planning, execution monitoring, self-healing, execution history, and optimisation. Prototype results show a 66.67% success rate and an average execution time of 120.52 seconds across nine simulated runs. Reward-based optimisa-tion enabled cache, parallel testing, retry count, and extended timeout. The framework therefore represents a practical step towards autonomous DevOps and intelligent software deliv-ery while remaining clear about the additional engineering re-quired for production deployment.

IX. ACKNOWLEDGEMENTS

The authors thank the Department of Computer Science and Engineering, Holy Mary Institute of Technology and Science, Hyderabad, India, for providing academic support and facilities for carrying out this work.

REFERENCES

[1] Humble, J., Farley, D.: Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Addison-Wesley, Boston, USA, 2010.
[2] Bass, L., Weber, I., Zhu, L.: DevOps: A Software Architect's Perspective. Addison-Wesley, Boston, USA, 2015.
[3] Jenkins Project: Jenkins User Documentation. Avail-able at: https://www.jenkins.io/doc/, ac-cessed 2026.
[4] Docker Inc.: Docker Documentation. Available at:
[5] https://docs.docker.com/, accessed 2026.
[6] Kubernetes Authors: Kubernetes Documentation. Avail-able at: https://kubernetes.io/docs/, ac-cessed 2026.
[7] Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge, USA, 2018.
[8] Wooldridge, M.: An Introduction to MultiAgent Systems. Wiley, Chichester, UK, 2009.
[9] Chen, M., Tworek, J., Jun, H., et al.: Evaluating Large Language Models Trained on Code. arXiv:2107.03374, 2021.
[10] Fan, A., Gokkaya, B., Harman, M., et al.: Large Lan-guage Models for Software Engineering: Survey and Open Problems. arXiv:2310.03533, 2023.
[11] Liu, J., Wang, K., Chen, Y., et al.: Large Language Model-Based Agents for Software Engineering: A Sur-vey. arXiv:2409.02977, 2024.
[12] Zhang, L., Jia, T., Jia, M., et al.: A Survey of AIOps for Failure Management in the Era of Large Language Models. arXiv:2406.11213, 2024.
[13] Zhang, L., Jia, T., Jia, M., et al.: A Survey of AIOps in the Era of Large Language Models. arXiv:2507.12472, 2025.



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY