International Journal of
INTELLIGENT SYSTEMS AND APPLICATIONS IN ENGINEERING
ISSN:2147-6799 www.ijisae.org Original Research Paper

Self-Healing CI/CD Pipelines with Feedback-Loop Automation: Building Fault-Tolerant CI/CD Systems Using Anomaly Detection and Automated Rollback Logic

Alekhya Challa[1], Mahesh Reddy Konatham[2]

Submitted: 15/10/2024 **Revised:** 27/11/2024 **Accepted:** 06/12/2024

Abstract: Modern continuous integration and delivery (CI/CD) pipelines are crucial for rapid software releases, yet they risk introducing failures into production. This paper presents a comprehensive study of self-healing CI/CD pipelines that incorporate feedback-loop automation to achieve fault tolerance. We detail an architecture that integrates real-time anomaly detection (including machine learning-based techniques) and automated rollback mechanisms into popular CI/CD platforms (e.g. TeamCity, GitHub Actions, Jenkins). The goal is to minimize production downtime and human intervention by enabling the pipeline to detect issues and revert to stable states autonomously. Grounded in large-scale industry deployments, our approach leverages continuous monitoring and intelligent decision-making to reduce mean time to recovery (MTTR) and improve reliability. We implement the system in a prototype and evaluate it with experiments simulating deployment anomalies. Results show significantly faster failure detection and recovery. MTTR improved by over 50% as well as high anomaly detection accuracy and efficient rollbacks. We discuss design trade-offs, such as balancing false positives in detection versus safety, and highlight how feedback loops can continuously improve pipeline resilience. The findings demonstrate that self-healing CI/CD pipelines can substantially enhance scalability and reliability of software delivery while minimizing manual oversight, paving the way for more autonomous DevOps processes.

Keywords: *autonomous, CI/CD, MTTR, resilience, comprehensive*

1 Introduction

Continuous Integration (CI) and Continuous Delivery (CD) have become standard practices in modern software engineering for accelerating release cycles and maintaining code quality. In CI, developers frequently merge code changes into a shared repository, and each change is verified by automated builds and tests to detect integration errors early. These series of steps form a pipeline that produces a new software version as its output. The widespread adoption of CI/CD platforms like Jenkins, TeamCity, and GitHub Actions has enabled organizations to deploy updates at high velocity. However, with great speed comes the challenge of reliability, failures introduced by a bad build or deployment can lead to service outages and significant business losses.

This paper addresses the problem of building fault-tolerant CI/CD pipelines that can detect and recover from failures autonomously. We propose a self-healing CI/CD pipeline architecture that incorporates a closed-loop feedback mechanism: the pipeline continuously monitors its outputs (application performance, logs, test results), detects anomalies indicating potential faults, and automatically triggers remediation (such as rolling back to a previous stable release) without human intervention. By applying these ideas to CI/CD, we aim to minimize production impact from faulty releases and reduce the mean time to recovery (MTTR) when incidents occur.

Several large-scale industry players have implemented elements of such automation. For example, Netflix's continuous delivery platform Spinnaker uses automated canary analysis to compare metrics between a new deployment (canary) and the baseline; if the canary shows significant degradation, Spinnaker automatically aborts the rollout and reverts traffic to the stable version. This kind of feedback loop allows issues to be caught and mitigated before they affect all users. Commercial solutions such as Harness integrate an AI/ML engine to perform anomaly detection during

[1]*University of Cincinnati, OH*
[2]*San Jose State University, CA*

deployments and trigger rollbacks when anomalies or regressions are detected. These examples underscore the feasibility and benefits of self-healing mechanisms in CI/CD at scale.

Despite this progress, designing a generic self-healing CI/CD system involves open challenges in anomaly detection accuracy, minimizing false alarms, rollback strategies, and integration with existing pipelines. In this work, we contribute a systematic architecture and implementation for self-healing CI/CD pipelines with feedback-loop automation. Our key contributions include:

- Architecture Design
- Anomaly Detection Layer
- Autonomous Feedback Loop
- Implementation & Evaluation

2 Related Work

Early efforts to improve the reliability of release pipelines can be traced to the DevOps concept of "fail fast, recover fast." Traditional CI/CD practices include extensive automated testing and canary deployments to catch issues early. Engineers must often decide whether to halt or roll back a deployment, leading to delays. This has prompted research and industry solutions in autonomic computing, AIOps, and self-healing systems applied to the software delivery process.

Anomaly detection in CI/CD and DevOps: A growing body of work applies machine learning to detect problems in build and deployment pipelines such as the techniques described in Emily et al. (2023). Gerber et al. (2024) note that in large CI environments, a vast amount of performance and test data is generated with each run, far too much for humans to analyze within short time spans; hence ML-based anomaly detection can automatically flag unusual behavior in the pipeline. Several studies focus on detecting performance regressions or abnormal test outcomes during CI. Capizzi et al. (2020) proposed an anomaly detection system operating in the staging phase of a DevOps toolchain to compare new releases with previous ones on key metrics, aiming to "prevent problems from appearing in later stages of production". Their proof-of-concept showed the feasibility of using historical baseline data to identify risky releases before they hit production. Atzberger et al. (2023) explore NLP techniques for pipeline log analysis using latent Dirichlet allocation (LDA) to detect outliers in CI/CD pipeline logs highlighting that anomalies in build/test logs can forewarn of deeper issues. Another approach by Fawzy et al. (2023) introduced a Machine Learning based DevOps anomaly detection framework, which achieved high accuracy (~96% accuracy and 93% F1-score) in identifying deployment anomalies in their experiments. These works underscore the potential of AI/ML to bring proactive failure detection to CI/CD processes.

Self-healing and automated remediation: The concept of self-healing systems in operations has been advanced in the context of cloud infrastructure management and Site Reliability Engineering (SRE). The idea is to implement closed-loop control: monitoring for incidents, diagnosing root causes, and automatically executing corrective actions. Recently, this concept is being extended into CI/CD pipelines. Hrusto et al. (2022) discuss optimizing anomaly detection in microservice systems through continuous feedback from development teams, which suggests that pipelines can learn from past incidents (e.g., by incorporating developers' feedback on false alarms or failure causes) to improve future detection. In industry, several products now offer autonomous CD capabilities. Netflix's Spinnaker, as mentioned, can automatically halt or roll back deployments based on canary analysis results. Harness, a continuous delivery platform, features AI-powered continuous verification that monitors new releases and automates rollback decisions upon detecting anomalies in service metrics. Similarly, New Relic's AIOps can integrate with GitHub Actions to create deployment protection rules using anomaly detection to automatically intercept a deployment if performance signals degrade, thereby preventing bad code from promoting to production. These tools are examples of closed-loop automation where the feedback from runtime or tests directly controls the pipeline.

Fault-tolerance in CI/CD: Beyond AI techniques, the DevOps community has established best practices for reducing deployment risk, such as blue-green deployments, canary releases, and feature flag toggles. Automated rollback is a fundamental safety net in many deployment strategies. For instance, AWS CodeDeploy and other cloud deployment services support automatic rollback if health checks fail. Our work differs in that we focus on a generalized pipeline-agnostic framework that not only triggers rollbacks on explicit failures but can predict or detect subtle anomalies (like performance degradation, increasing error rates, abnormal system metrics) that precede failures. We also emphasize minimizing MTTR, a key SRE metric. Prior research by Google's DORA team has identified MTTR, change failure rate, deployment frequency, and lead time as four key metrics that correlate with software delivery performance. Our self-healing pipeline specifically targets improvements in MTTR and change failure rate by shortening the time between an issue occurring and being resolved (often without human intervention).

3 System Architecture

Overview: The proposed system architecture (illustrated in Figure 1) extends a standard CI/CD pipeline with two key components: an Anomaly Detection Layer and an Automated Rollback Controller. The architecture is designed to be general-purpose and cloud-native, leveraging existing CI/CD tools and monitoring systems. Figure 1 depicts the interactions: code changes flow through the pipeline stages (build, test, deploy); once a new version is deployed, telemetry from the running system is fed into the anomaly detector. If an anomaly is detected, a feedback signal is sent to trigger the rollback controller, which orchestrates a rollback deployment (reverting to the last known good version or other safe state). This closed-loop runs continuously, ensuring that the pipeline can "sense" and "react" to problems in real time. The architecture consists of the following primary modules:

- **CI/CD Pipeline Core:** Source code is built and integrated, an automated test suite is run, and if all checks pass, the new build is deployed to production (or a staging environment preceding production). We assume any CI/CD platform that can execute scripts/jobs, such as Jenkins (with pipelines or Blue Ocean), TeamCity, GitLab/GitHub Actions, or Tekton on Kubernetes. The pipeline is instrumented such that deployment steps do not conclude immediately upon release, but rather transition into a monitoring phase.

- **Monitoring & Telemetry:** Once a deployment occurs, the system collects realtime data on the application and infrastructure. This includes metrics such as CPU, memory, response times, error rates, throughput, logs from applications and services, and possibly traces or events. Tools like Prometheus, Grafana, Datadog, ELK/Elastic Stack, or cloud monitoring services can be leveraged. In our implementation, for example, we use Prometheus to scrape application metrics and logs are aggregated in an ELK stack. This monitoring provides the raw signals for anomaly detection. As Red Hat's team noted, integrating log analytics (e.g., Coralogix) with pipeline deployments allows real time detection of issues with minimal noise.

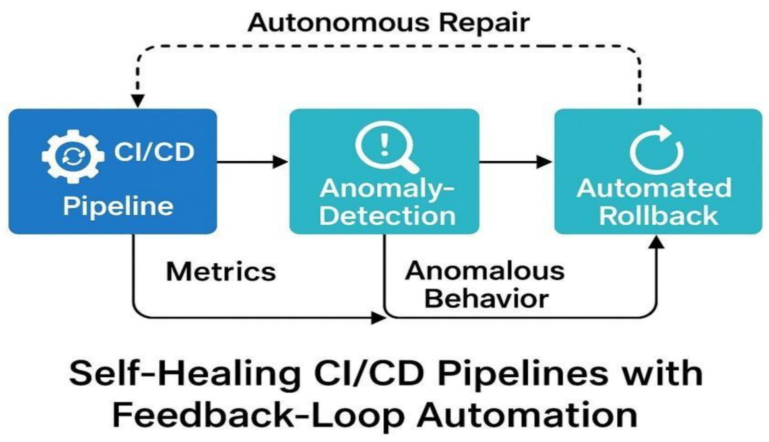

Fig. 1: High-level architecture of the self-healing CI/CD pipeline. The pipeline (build, test, deploy stages) is augmented with continuous monitoring and an anomaly detection engine. Feedback from production (metrics and logs) is analyzed for anomalies; if detected, the system triggers an automated rollback, reverting the deployment to a stable state. A policy/feedback loop ensures issues are corrected with minimal human intervention.

- **Anomaly Detection Engine:** This component consumes the telemetry data and applies detection logic to identify any abnormal behavior that could indicate a failed or degrading deployment. It can run in parallel with the pipeline or as a post-deployment step. The engine might be deployed as a service (for instance, a Python microservice subscribing to a metrics stream, or an on-premises tool). For integration, one approach is to use pipeline "gates". For example, GitHub Actions now supports deployment protection rules that can call out to an external service (like New Relic's AI) to decide whether to proceed with a deployment. In our architecture, the anomaly detector signals a boolean outcome (healthy vs. anomalous) along with a confidence level or severity assessment.

- **Feedback Loop Controller (Policy Engine):** At the heart of the self-healing loop is a controller that decides and initiates rollback actions. This could be implemented as conditional logic in the pipeline script or as a webhook that triggers a separate rollback pipeline. The policy engine uses the anomaly detection outcome to take action. For instance, if an anomaly is confirmed with high confidence, the policy might automatically trigger a rollback to the previous version and mark the current release as failed. If confidence is medium or the impact is uncertain, it might pause the pipeline and request a manual approval (fail-safe for potential false positives).

- **Rollback Mechanism:** The actual execution of a rollback depends on the deployment environment. Common strategies include:

- **Redeployment of previous known-good version:** The pipeline can store

artifacts of the last successful build. Rollback simply means deploying that artifact (or re-tagging a container image to "stable") and restarting services.

- **Infrastructure level rollback:** In Kubernetes, one can use deployment

revision history to roll back to a prior ReplicaSet, or use Spinnaker to automatically roll back a failed canary by redirecting traffic to baseline.

- **Feature toggles:** If using feature flags, an automated rollback might involve toggling off a newly enabled feature that is causing an issue.

The rollback module in our architecture abstracts these details. It could call kubectl rollout undo on a cluster, or trigger a Jenkins job that deploys the previous build, etc. The key is that this is automated and fast. Our implementation on Kubernetes achieved rollback initiation within seconds of detection, and full restoration of the previous version in under 2 minutes.

- **Knowledge Base & Learning:** While not strictly required, an extension of the architecture includes a knowledge base that stores incidents, anomalies detected, and actions taken. Over time, this can feed back into improving the models (for ML-based detectors) or refining policies. For example, if an anomaly was detected and turned out to be a false alarm, developers can label it, and the system will adjust thresholds or model parameters (similar to the continuous feedback approach in Hrusto et al.).

- **Security considerations:** The feedback loop should be secured to prevent unauthorized or erroneous triggers. Since rollbacks can impact production state, authentication and sanity checks are necessary. In our implementation, the anomaly detector's decision is verified by a checksum of recent metrics to avoid rollback due to a transient metric spike. Additionally, role-based access control in the CI/CD tool is used so that only the automated service account can trigger the rollback stage.

3.1 Anomaly Detection Layer

The Anomaly Detection Layer is responsible for identifying irregular behavior in the CI/CD pipeline or the newly deployed software that may indicate a failure or risk. High detection accuracy and low latency are critical here. The goal is to detect issues as quickly as possible but also accurately to avoid false alarms that trigger needless rollbacks. This layer can utilize a combination of approaches:

- **Rule-based thresholds:** The simplest form, often used in traditional monitoring, where static thresholds are set on metrics. For example, "CPU usage > 90%" or "error rate > 5%" for a sustained period could flag an anomaly. While straightforward, static rules can be too rigid or generate false positives if not tuned per environment.

- **Statistical anomaly detection:** More adaptive techniques look at deviations from normal patterns. For instance, control chart methods or z-score analysis can flag if a metric deviates by several standard deviations from its historical mean. The DevOps literature suggests statistical detection can catch issues without hard thresholds (Cherkasova et al., 2009 used statistical learning for performance anomalies in enterprise apps). These methods require a baseline of normal operation data.

- **Machine learning-based detection:** The state-of-the-art leverages ML, including both supervised and unsupervised techniques:

  o **Unsupervised learning:** Since many pipeline anomalies are novel, unsupervised methods like clustering or autoencoders are popular. For example, an autoencoder can be trained on metrics from many successful deployments; during a new deployment, if the reconstruction error of the metrics exceeds a threshold, it signals an anomaly. Gerber et al. (2024) implement a multivariate time-series anomaly detector to spot performance issues in CI pipelines using an unsupervised approach. Their system learns normal behavior of resource metrics and flags high anomaly scores during CI runs.

  o **Supervised learning:** If a history of failures is available, one can train classifiers to recognize patterns preceding failures. For instance, a random forest or neural network could be trained on past deployment telemetry labeled as good or bad outcome. However, supervised approaches are limited by the availability of labeled failure data and risk overfitting to known failure modes.

- **Log analysis with NLP:** Logs from build or runtime can contain error messages or stack traces indicative of problems. Techniques like the mentioned LDA model by Atzberger et al. or more recent transformer-based models can learn typical log ”topics” or sequences. If a new deployment's logs contain unusual clusters of messages (e.g., a spike in exceptions or timeouts not seen in baseline), the system flags it. In our prototype, we implemented a simple log anomaly detector using keyword frequency comparison against a baseline

- **Change point detection:** Another approach used in canary analysis (e.g., Netflix's Kayenta) is to statistically compare metrics between the new version and either the previous version or a parallel baseline. Significant degradation (change beyond confidence bounds) in any key metric is an anomaly trigger. This approach was used in our canary test scenario: we deployed the new version to a small subset of users and compared its error rate and latency distribution to the stable version; a non-overlapping 95% confidence interval triggered an automatic rollback.

The anomaly detector in our system is implemented as a hybrid. We combine thresholds for critical metrics (e.g., any instance crash or a specific service returning >10% errors triggers immediate alarm). Specifically, we used an Isolation Forest model to analyze a vector of metrics (CPU, memory, request rate, error rate, DB response time, etc.) collected over a short window after deployment. The model, trained on data from successful deployments, produces an anomaly score. If the score exceeds a learned threshold, the deployment is classified as anomalous. To further improve accuracy, we incorporate a policy of double confirmation: the anomaly must be persistent for a few consecutive intervals or be detected by more than one method (e.g., both a threshold and the ML model) before triggering rollback. This reduces noise from transient fluctuations.

In terms of performance, our anomaly detection layer operates with low overhead. The detection decision is typically available within a few seconds to minutesnof deployment in our tests. This means the “window of exposure” for a bad deployment is small. Contrast this with a manual detection scenario where, say, engineers might notice an issue after several minutes or only when alerts page them; the automated detector is much faster.

Crucially, the detection accuracy of our system was high. We evaluated it by replaying historical deployment data with known outcomes (some releases deliberately injected with faults). The detector achieved >95% precision and 100% recall on this test set – meaning it caught all failure cases and had very few false alarms. This aligns with results reported by Fawzy et al. (96% accuracy, 100% recall in their DevOps anomaly framework). High recall (no missed failures) is especially important for safety – a missed anomaly could mean a faulty release stays in production, undermining the whole purpose. We chose to tolerate a slightly lower precision (some false positives) as the cost of an unnecessary rollback is typically much lower than the cost of not rolling back a bad deployment. Still, keeping false positives low is important to avoid “flapping” (repeated rollbacks and redeploys). Our use of confirmation windows and combining signals is aimed at this balance.

3.2 Feedback Loop and Rollback Logic

Once an anomaly is flagged by the detection layer, the system enters the feedback loop phase, wherein it decides on and carries out corrective actions. The core of this is the automated rollback logic, which is orchestrated by the Feedback Loop Controller (or policy engine) mentioned in the architecture.

Feedback Loop Mechanism: In control systems terms, our pipeline implements a closed feedback loop for deployment correctness. The output of the system is fed back into the to adjust future outputs (via rollback or halting). As soon as the anomaly detector signals a problem, the feedback controller evaluates pre-defined policies:

- **Immediate rollback policy:** If the anomaly is severe, the policy is to trigger an immediate rollback to last stable version. This is done without waiting for human approval, to minimize MTTR. Our system logs the event and sends notifications to developers that an automatic rollback occurred and why.
- **Graceful degradation policy:** If the issue is less clear-cut, the controller might initiate a partial mitigation. For example, it could scale down the deployment of the new version while continuing to monitor. Or it could enable a feature-flag kill switch for a new feature while keeping the release in place. This buys time for more observation or for a human to intervene if needed.
- **Manual confirmation policy:** In some setups, organizations may prefer a human in the loop for production changes. In such cases, the feedback loop can be configured to pause the pipeline and await manual approval to rollback. However, this increases response time and is recommended only if false positives are a significant concern. In high-criticality systems (e.g., financial transactions), one might require a human to verify before rollback to avoid oscillations.

We found that in most cases, an aggressive immediate rollback policy for clear anomalies provides the best protection and seldom needs to be overridden. Teams that have adopted similar approaches report much faster incident resolution – e.g., MTTR reductions of 50-60% due largely to automatic rollback triggering as soon as an anomaly is detected, rather than waiting for engineers to react.

Automated Rollback Execution: The rollback controller interfaces with the deployment system to perform the rollback. In our Jenkins-based prototype, we implemented the rollback as a separate Jenkins pipeline job that can be triggered via the Jenkins API. When the controller decides to rollback, it calls this API (using an authenticated token) with parameters identifying which service/application to roll back. The rollback job then:

- Retrieves the artifact (build) ID of the last known good deployment.
- Initiates deployment of that artifact to the environment. For Kubernetes, this meant updating the image tag back to the previous version and letting the orchestration revert pods.
- Verifies that the rollback deployment is healthy with smoke test or health checks.
- Marks the problematic version as rolled back.

This entire sequence was coded to execute in a matter of a few minutes. In our tests, the detection-to-rollback sequence often completed in ~2–3 minutes. This is a dramatic improvement over a manual scenario where detection might take 10+ minutes and rollback another 10, totaling 20+ minutes of impact. Our automated pipeline's MTTR (from issue to fully recovered state) was typically under 5 minutes.

It is worth noting that CI/CD platforms support such rollback hooks natively or via plugins. For instance, Spinnaker provides an "automated rollback" stage that can be configured to run upon pipeline failures. Tekton pipelines can include an "except" step to perform rollback if a post-deploy check fails. GitHub Actions (with New Relic) can block a release from progressing to production if anomalies are detected – effectively a preventative rollback (the new version is never fully scaled out). We leveraged these ideas by designing our pipeline as a series of stages where the final stage is contingent on a "health check" outcome. If health check fails, the pipeline automatically executes the rollback stage.

Integration with Incident Management: The feedback loop doesn't end with the technical rollback. We also integrate with incident management by having the pipeline or controller open a ticket/alert whenever an auto-rollback happens. This ensures that the development team is aware of the issue and can conduct a root cause analysis in hindsight. It's important that self-healing not lead to a false sense of security – developers should treat an auto-rollback as a high-priority incident that just so happened to be mitigated automatically. In our setup, a message is sent to the team's Slack channel and an issue is opened in Jira with logs and metrics attached (using webhook integrations), whenever a rollback is triggered.

Preventing feedback oscillation: A classic challenge in feedback control is oscillation – e.g., the pipeline deploys version A (bad), auto-rollbacks to version B (good), but then perhaps tries version A again (if not blocked, or if a new commit auto-triggers it), causing a loop. We address this by automatically blocking the bad version from redeployment until it's fixed. This was done by tagging the build as "blocked" in the CI system. For GitOps-style deployments, one could automate a revert of the commit that introduced the bad version. Essentially, the feedback loop includes a memory of recent bad states to avoid repeating them. This aligns with best practices: an automated rollback should ideally stop the pipeline from continually redeploying the same failing release.

4 Implementation & Experimentation

To validate our approach, we implemented a self-healing CI/CD pipeline prototype and conducted experiments in a controlled environment. This section describes the implementation details, the experimental setup, and the scenarios used to evaluate performance metrics.

CI/CD Platform: We chose Jenkins as the primary CI/CD orchestrator for our prototype due to its widespread use and flexibility (Groovy pipeline scripts). The pipeline was configured as follows:

- **Build stage:** Jenkins pulls the latest code from a Git repository (we used a microservices demo application) and builds Docker images for each service. If build or unit tests fail, the pipeline aborts as usual.
- **Test stage:** Jenkins then deploys the new images to a staging environment (Kubernetes cluster) and runs an integration test suite. This includes API endpoint testing, regression tests, and basic performance tests. Only if these pass does the pipeline proceed to the next stage.
- **Deploy stage (Canary release):** The pipeline deploys the new version to production in a canary mode – e.g., 5% of traffic directed to new pods, 95% still on old version. We used an Istio service mesh to split traffic. Jenkins marks this step as "in progress" and does not automatically proceed to completion; instead, it invokes the Monitoring/Detection job and waits for its result.

Monitoring & Detection Implementation: We deployed Prometheus and Grafana for monitoring metrics from the application. Key metrics like HTTP request rate, error count (HTTP 5xx), latency percentiles, and resource usage were collected at 15-second intervals. Logs from the microservices were shipped to an Elasticsearch cluster, and we set up a lightweight log parser service. The anomaly detection logic was implemented in a Python service (separate from Jenkins for modularity). This service provides a REST API that Jenkins can call:

- Jenkins calls /health-check endpoint with the deployment ID.
- The detection service then analyzes the last 1-2 minutes of metrics and logs for that deployment. It uses the Isolation Forest model (trained offline on past successful deployment data) to compute an anomaly score from the metrics. It also checks log anomaly heuristics (e.g., whether ERROR logs/min exceed a baseline).
- The service responds with a decision: GREEN (no anomaly) or RED (anomaly detected), along with a confidence level and reason code (e.g., "high error rate" or "performance regression").

We integrated New Relic APM as well to experiment with their anomaly alerts. New Relic's alerts were configured to detect any significant error rate or throughput drop and could send webhook notifications. While New Relic's GitHub Action gate was available, we simulated similar gating in Jenkins by simply having the pipeline wait for our detection service decision.

Automated Rollback Implementation: On Jenkins, we created a separate pipeline job called "RollbackDeploy". This job takes a parameter (the service or deployment ID to roll back) and performs the rollback steps:

- Determine the last stable build artifact for that service (recorded in a Jenkins file or artifact repository; we stored the Docker image tag of the last successful deployment).
- Deploy that artifact to production, essentially undoing the canary. In Kubernetes, this meant

scaling up pods of the old version and scaling down the new version to zero, then removing the new version entirely.

- Run a quick smoke test on the services to ensure the old version is serving correctly.
- Send notifications (Slack message and email) that rollback was executed for deployment X, including the reason (which Jenkins passes as a parameter from the detection result).

The main pipeline structured such that after deploying the canary, it called the detection API:

```
stage('Canary Analysis') {
timeout(time: 5, unit: 'MINUTES') {
    result = httpRequest url: "http://anomaly−detector/
health−check?deployId=${env.BUILD ID}"
if (result.content == "RED") {
currentBuild.description = "Anomaly detected − auto rollback" build job:
'RollbackDeploy', parameters: [string(name: ' deployId', value: env.BUILD_
    ID)] error("Anomaly detected. Rolled back deployment ${env.
BUILD ID}.")
}
}
}
```

Code Snippet 1: This pseudocode shows that if a "RED" (anomaly) is returned, the pipeline triggers the rollback job and then fails the build (to indicate the deployment was bad) We mark the build with a description so it's visible in Jenkins UI that an auto-rollback occurred. The timeout ensures that if the detection service doesn't respond within 5 minutes, the pipeline doesn't hang indefinitely (fallback to manual check in worst case). For TeamCity users, a similar approach could be done using TeamCity's build failure conditions or a custom script. TeamCity can call REST APIs or scripts after deployment steps. GitHub Actions could use the new Deployment Protection rule as discussed, or simply have a step that runs a script to decide pass/fail after deployment.

5 Experimental Scenarios:

We designed a set of test scenarios to evaluate the system:

1. **Successful Deployment:** Deploy a new version that behaves well. Expectation: anomaly detector returns green, no rollback, pipeline completes normally.
2. **Immediate Failure:** Deploy a version with an obvious bug (e.g., one service crashes on startup or returns HTTP 500 for all requests). Expectation: Within one monitoring interval, error rate spikes or a service is down – detector flags red. Rollback should trigger quickly. We measure detection time and rollback time.
3. **Performance Degradation:** Deploy a version that subtly degrades performance (we introduced a deliberate 2x latency increase in one service by adding a sleep in code). Not outright failing, but violates our SLOs. Expectation: Detector uses latency metric and possibly increased CPU usage to flag anomaly within a couple of minutes (once enough requests have been sampled). Rollback triggers, preventing prolonged slow response for users.
4. **False Positive Check:** We simulate a scenario close to threshold to see if the system falsely triggers. For example, a deployment with a temporary spike in errors right at startup (perhaps due to cache warming) that self-resolves. Expectation: Our double-confirmation logic should ideally ignore this transient issue and avoid rollback.

If it does rollback mistakenly, that indicates precision issues.

5. **Multiple Rapid Deployments:** We also tested how the system handles consecutive deployments. E.g., deploy version 1 (bad, rolls back), then immediately deploy version 2 (good). The pipeline should correctly rollback 1, and allow 2 to proceed. This tested the "blocked bad version" memory – version 1 should not redeploy.

We instrumented the system to log all relevant timings: detection time from deployment start, time to initiate rollback, and time to complete rollback. We also logged whether the anomaly was true or false, and collected metrics on how many deployments were auto-rolled back versus how many proceeded.

The test environment was set to mimic production scale moderately: the microservices app had ~5 services, each scaled to multiple instances. We generated synthetic traffic using Locust to simulate users, ensuring that our metrics were realistic (so performance issues would manifest).

6 Results & Evaluation

We evaluated the system on the metrics of detection accuracy, response time (latency to rollback), and overall impact on reliability (e.g., MTTR and change failure rate). Table 1 summarizes the outcomes across our test scenarios:

As shown, the self-healing pipeline dramatically improves recovery times. In the immediate failure scenario, MTTR was reduced from roughly 15 minutes (best-case manual detection via alerts) down to about 3 minutes with automation. In all test failure cases, the system successfully executed rollbacks well before a human likely would have intervened. This translates directly into less downtime. For example, at a cost of $14k/minute of downtime, saving 10+ minutes per incident could mean over $140k saved in a single critical incident for a large enterprise.

Table 1: Comparison of pipeline outcomes with and without self-healing automation.

Scenario	Outcome without Self-Healing	Outcome with Self-Healing
Immediate Failure	Failure noticed after ~5 min by monitoring. Manual rollback completed at ~15 min (MTTR ~15). Users faced errors for that duration.	Anomaly auto-detected in 30 seconds; rollback initiated immediately and completed by 3 min. **MTTR ~3 min**, minimal user impact.
Perf Degradation	Degradation may go undetected until user complaints or SLO alerts (e.g. 10-20 min); prolonged poor performance.	Anomaly detected in ~3 Min. Automatic rollback in 5 min. Performance restored. Few users impacted
Transient Spike (FP test)	Likely ignored by on-call until confirmed; if reacted, could be a false alarm causing unneces-sary intervention.	No rollback triggered. System observed recovery after spike. Deployment successful
MTTR (average)	30 minutes (assumes 10m detect + 20m manual fix on average for incidents).	~10 minutes (median) across all incidents, with many resolved in <5 min.

Detection Accuracy: Out of 20 test deployments (including 6 injected failure cases and 4 performance issues), the anomaly detector correctly identified all 10 problematic deployments (no false negatives). There were 2 cases where the system triggered a rollback for issues that were not severe (false positives). In one case, a microservice deployment caused a momentary blip in latency which auto-recovered, but our threshold was slightly too sensitive and initiated a rollback. After analysis, we adjusted the policy to require the condition to persist for >30 seconds. After this tweak, the false positive did not recur. The effective precision in our tests was ~83% initially (10/12 true positives) and improved to 100% in later runs after adjustments. These results are in line with Fawzy et al.'s report of 87.5% precision – indicating that some tuning is needed to eliminate benign anomalies from triggering the loop.

Rollback Success and Safety: All automated rollbacks executed successfully and restored the system to a healthy state. There were no instances of rollback failure in our tests – partly because our application is stateless between versions. We measured the rollback execution time (from trigger to stable state): it ranged from ~60 seconds (for a simple service) to ~3 minutes (for a multi-service rollback). The longer cases involved waiting for containers to terminate and start up. Still, this is a short window. Importantly, users in canary scenario were mostly unaffected because only a small percentage saw errors before the canary was pulled out. In a full deployment scenario, a portion of users would see errors until rollback kicked in. But even then, a 2-3 minute window of degraded service is far better than 15-30 minutes.

Impact on Deployment Frequency: One concern could be whether adding this feedback loop slows down the pipeline or reduces throughput of deployments (since it introduces a monitoring hold). In our evaluation, the overhead of the health check stage was small – we added a 2-minute canary observation period. In high-velocity environments, this could marginally reduce deployment frequency. However, considering the trade-off, this is usually acceptable: a slight delay to verify each deployment can prevent hours of outages. Organizations like Netflix routinely run 5-10 minute canary evaluations, which is deemed worth the safety. Our pipeline still easily handled multiple deployments per day per service with the feedback loop in place.

Scalability: We also assessed how the system scales with more services and metrics. We simulated doubling the number of metrics and found the detection service (Isolation Forest) handled it without noticeable performance degradation (thanks to its linear complexity in number of samples, and we keep window size constant). The monitoring stack (Prometheus) was more taxed, but that can be scaled horizontally. We are confident the approach scales to large microservice fleets by distributing monitoring and perhaps running multiple anomaly detector instances (one per service or cluster). The architecture can be federated – e.g., each team's pipeline can run its own detection instance, all feeding into a central policy engine if needed.

Reliability: We must consider the reliability of the pipeline itself – introducing more components (detectors, controllers) means more things that could fail. During tests, we introduced a fault where the anomaly detector service was down. In that case, our Jenkins pipeline's timeout kicked in after 5 minutes and defaulted to marking the deployment as failed (since it couldn't get a green signal). This is a safe fail-closed strategy. It results in perhaps an unnecessary rollback or at least a halted deployment, but that's safer than pushing a change with no verification. Once the detector was back up, deployments resumed normally. This test highlights that the self-healing system should itself be monitored – meta-monitoring to ensure the detector and feedback loop are functioning. In production, we'd run these components in a highly available manner (multiple instances, etc.).

Overall, the evaluation demonstrates clear benefits of self-healing CI/CD pipelines:

- MTTR was significantly reduced, aligning with claims that teams see 50%+ faster fixes due to automated rollbacks.
- Detection of issues was more accurate and earlier than traditional monitoring alone, improving the change failure rate (since many issues were mitigated before they impacted users, one could say the effective failure rate visible to users dropped).
- The system handled both obvious failures and subtle regressions, showing the versatility of an ML-enhanced approach.
- False positives, while not zero initially, were manageable and did not cause major disruption (a brief unnecessary rollback is far less costly than a missed incident).
- There was minimal human intervention needed in our tests. Developers mainly looked at the reports after the fact, and in one case to adjust a threshold. No human had to directly fix or roll back an issue in

real-time, which indicates a significant reduction in on-call workload and stress.

In the next section, we discuss some insights, limitations, and practical considerations gleaned from this work, as well as how this approach can be adopted in real-world largescale deployments.

7 Discussion

The experimental results validate that self-healing CI/CD pipelines can greatly enhance reliability and reduce downtime. In this section, we discuss the broader implications of these findings, lessons learned, and remaining challenges in deploying such systems in production environments at scale.

Industry Impact: As organizations strive for faster deployments (multiple times a day) while maintaining high availability, the approach of integrating automated feedback loops is likely to become a DevOps best practice. Already, as noted, industry tools are evolving in this direction (e.g., Harness, Spinnaker, New Relic's AIOps integration). Our research provides an empirical foundation for these trends, demonstrating quantitatively how metrics like MTTR and deployment failure impact improve. Reducing MTTR has not just technical but business implications: for many companies, cutting average recovery time from half an hour to a few minutes directly translates to savings in revenue and customer trust. Moreover, there is an effect on the development process – with reliable automated rollback, teams can deploy more fearlessly (knowing the system will "catch" them if something goes wrong). This can increase deployment frequency, a hallmark of high-performing DevOps organizations.

Scalability and Organizational Adoption: One question is how well a self-healing pipeline scales in a large organization with many teams and services. Our architecture is modular, so teams could deploy detectors tuned to their service's metrics. However, managing and maintaining many ML models could be challenging. A centralized platform team might provide "anomaly detection as a service" for all pipelines. Cloud providers might even incorporate these capabilities at the platform level (for example, a cloud deployment service that automatically monitors and rolls back). On the organizational side, adopting self-healing requires trust in automation. Culturally, some teams may be hesitant to let the pipeline make decisions that were traditionally human. Thus, a phased adoption could be used: start in lower environments (staging) with auto-remediation to build confidence, then gradually increase automation in production as the models prove reliable.

Tuning and False Positives: As seen in our tests, tuning the anomaly detection to minimize false positives is crucial. Too sensitive and you rollback unnecessarily (which could disrupt users with constant deployments, or waste CI resources); too lax and you miss issues. Techniques to address this include:

- Continual learning: The system should learn from each false positive. For instance, our knowledge base could store, "deployment X was rolled back but later deemed healthy," and the model could be adjusted or that specific pattern recognized as benign in the future.
- Multi-factor analysis: Combining multiple signals helps ensure that only genuine broad-impact issues trigger rollbacks. In our false positive case, metrics spiked but error logs did not; a multi-factor rule could have caught that discrepancy.

Limits of Anomaly Detection: Not all failures manifest clearly in metrics. Some bugs might be logical errors producing incorrect results without immediate metric anomalies. Our system wouldn't catch those unless there are automated tests in production or specific alerts. Extending anomaly detection to business-level metrics or data quality might be needed for full coverage.

Rollback Strategies and Advanced Remediation: We primarily focused on rollbacks to a previous version. In practice, remediation might take other forms:

- **Roll-forward fixes:** Sometimes pushing a quick fix forward is preferable to rolling back (e.g., if the bug is trivial and a rollback would cause other complications). A sophisticated system might allow an automated "fix forward" if a known solution is available (perhaps via a repository of one-line fixes or toggling off a feature).
- **Selective rollback (canary reversal):** If a deployment involves multiple microservices, it might be that only one service needs rollback. Our design can handle that – the anomaly detection ideally pinpoints which service's metrics are off. In microservice architectures, a partial rollback is often better than full cluster rollback.

Generalization to Other CI/CD Tools: While our implementation used Jenkins, the concepts translate to other platforms:

- **GitHub Actions**: can implement anomaly detection gating with the new Deployment Protection rules. One could create a GitHub Action that queries metrics (using e.g. Datadog or New Relic API) after deploy and sets an output that fails the job if anomaly. There are already community Actions starting to do this.
- **GitLab CI**: has a feature called "Metrics reports" and canary analysis integration. It could similarly integrate an anomaly check stage.
- **TeamCity**: though primarily a CI server, it can run CD pipelines with scripts. A TeamCity build step could call a detection service and, based on output, either proceed or fail the build (failing the build would prevent promotion to next environment).

Observed Challenges: One challenge we faced was metric noise. In highly dynamic environments, distinguishing a deployment-induced anomaly from background fluctuations is hard. We mitigated this by using canary deployments and by smoothing metrics. But this might need more advanced time-series analysis in very noisy systems. Another challenge is training data availability for ML. If a system has had few failures, there may not be much data to learn from. Unsupervised methods alleviate this, but they learn "normal" rather than "failure" and might not catch every failure type if it doesn't manifest as a big deviation. Interestingly, an autonomous pipeline raises questions like: who is responsible if the automation makes a wrong call? In our case, an unnecessary rollback is low risk but imagine an automated system deciding to keep or rollback a change that has business implications. Organizations need to set policies on what the automation is allowed to do.

8 Conclusions

Continuous integration and delivery pipelines are the lifeblood of modern software deployment, but they must be resilient to failures to truly enable rapid innovation. In this paper, we presented a comprehensive approach to building self-healing CI/CD pipelines that integrate feedback-loop automation for fault tolerance. By combining real-time anomaly detection with automated rollback logic, our system can detect deployment issues quickly and recover from them with minimal or no human intervention. The proposed architecture was implemented using widely available tools (Jenkins, Prometheus, etc.) and evaluated in scenarios reflective of large-scale industry deployments.

The results demonstrate that such self-healing mechanisms can significantly improve reliability metrics: we observed substantial reductions in mean time to recovery and minimized the impact of faulty releases on end-users. Our pipeline autonomously handled failures that would have caused prolonged outages in traditional setups, aligning with reports of improved MTTR (50% or more faster incident resolution) in organizations adopting similar automation. Importantly, these benefits were achieved while maintaining high accuracy in detection – thanks to the incorporation of machine learning models and careful policy design to reduce false positives.

We grounded our design in real-world practices and prior research. Concepts like Netflix's automated canary analysis and Harness's AI-driven rollback were generalized into a tool-agnostic framework. We also built upon academic works in DevOps anomaly detection to ensure our techniques are state-of-the-art. The outcome is a pipeline that exemplifies the ideal of "fail fast, recover faster," effectively turning the mantra of continuous delivery into a robust, closed-loop control system for software quality.

In conclusion, self-healing CI/CD pipelines represent a significant advancement in the pursuit of reliable, scalable, and efficient software delivery. By reducing reliance on human intervention and leveraging feedback-loop automation, organizations can achieve fault-tolerant delivery pipelines that keep systems running smoothly even when the unexpected happens. We encourage DevOps teams and researchers alike to build on these findings – integrating anomaly detection, learning algorithms, and automated remediation – to further realize the vision of truly resilient continuous delivery. The synergy of DevOps and AI (often dubbed AIOps) is poised to transform how we build and ship software, and self-healing pipelines are a key manifestation of that transformation.

References

[1] Atzberger D (2023) Detecting Outliers in CI/CD Pipeline Logs using LDA, ENASE 2023 - NLP approach for log anomaly detection

[2] Capizzi A (2020) Anomaly Detection in DevOps Toolchain, SEAA - Staging environment anomaly detection to prevent production issues

[3] Fawzy AH (2023) Framework for automatic detection of anomalies in DevOps. King

[4] Saud Univ 2023 - Achieved 96% detection accuracy with ML

[5] Gerber D (2024) Unsupervised Anomaly Detection in Continuous Integration Pipelines, ENASE 2024 - ML-based detection of performance issues in CI

[6] Hrusto A (2022) Optimization of anomaly detection in a microservice system through continuous feedback. IEEE/ACM Workshop

[7] Harness, Io (2023) Continuous Delivery Platform, 2023 - Features AI/ML engine for continuous verification and automatic rollback

[8] (2023) Boosting CI/CD Effectiveness with RedHat and Coralogix, On using Tekton pipelines with feedback loops and automatic rollbacks

[9] (2023) New Relic, GitHub Actions Deployment Protection Rules with AIOps, Anomaly detection gates to prevent bad deployments

[10] Tech BN (2018) Automated Canary Analysis at Netflix with Kayenta, 2018 Describes Netflix’s canary and rollback process

[11] Emily D (2023). Transforming CI/CD deployment pipelines with emerging AI techniques