National College of Ireland

Self-Healing CI/CD Pipelines: Automating Kubernetes Deployments with Terraform and Ansible

MSc Research Project
Master of Science in Cloud Computing

Prasanna Pankaj Shinde
Student ID: X23278480

School of Computing
National College of Ireland

Supervisor: Sai Emani

National College of Ireland
Project Submission Sheet
School of Computing

National College of Ireland

Student Name:	Prasanna Pankaj Shinde
Student ID:	X23278480
Programme:	Master of Science in Cloud Computing
Year:	2024
Module:	MSc Research Project
Supervisor:	Sai Emani
Submission Due Date:	11/08/2025
Project Title:	Self-Healing CI/CD Pipelines: Automating Kubernetes Deployments with Terraform and Ansible
Word Count:	3042
Page Count:	22

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	Prasanna Pankaj Shinde
Date:	14th September 2025

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Self-Healing CI/CD Pipelines: Automating Kubernetes Deployments with Terraform and Ansible

Prasanna Pankaj Shinde
X23278480

Abstract

In fast developing world of cloud-native DevOps, the dependability, and robustness of the CI/CD pipelines are essential. The traditional CI/CD systems are reactive, tool-centric, and highly manual prone when it comes to fault recovery. This study presents an integrated, intelligent, self-healing, CI/CD platform which incorporates Kubernetes, Terraform, Ansible, Prometheus and Open Policy Agent (OPA) as the systems to identify, categorise, and process failure rates, in an autonomous manner, in real time. The development system consists with five main modules, including Failure Classification Engine, Trust-Aware Deployment Scoring, Dynamic Policy enforcement, Deployment Health Scoring (DHS), and Explainable Rollbacks. All of these elements are all that is required to make the decisions be smart, adaptive to the context and be governed in a smart fashion, the roll back to be automated with full transparency and to modify policies on the fly. Deploying Prometheus as the observability toolset, the system constantly monitors pipeline activity and acts upon it with a sort of recovery protocol with selective fire as it is known. The framework, with simulations of controlled failures including changes of software and infrastructure levels of anomalies, the stability of a deployment, and policy compliance. The paper addresses the ability of open-source technologies to be choreographed to create scalable, cost-effective, autonomous CI/CD pipelines that enable minimum downtime, minimum human intervention, and improvement in kinetic confidence in deployment outcomes. The study has valuable implications to DevOps organizations as a resilient, cloud-native automation that does not come with the limitation of proprietary automation.

1 Introduction

1.1 Background

DevOps and cloud computing have revolutionized the trends of software development and deployment in the contemporary world. Microservices, containerization, and cloud-native architecture have been the major contributors to increased reliance on CI/CD pipelines in automating software delivery. The pipelines combine the integration of code, testing, and deployment to enable the delivery of software faster and more reliably. But the transitory nature of cloud-native platforms creates problems such as deployment failures, compliance with security policies, configuration drifts, and rapid rollback requirements, all of which can compromise application availability and system integrity Dileepkumar

and Mathew (2023). According to recent studies, Continuous Integration and Continuous Delivery/Deployment (CI/CD) adoption increases the delivery speed, however, the majority of toolchains are still reactive and need human intervention to recover Dileepkumar and Mathew (2023). Newer policy-as-code design approaches, including those used with the Open Policy Agent (OPA) are better at compliance and often lack runtime self-healing Castro and Jack (2025). This gap prompts the necessity of the integrated approach. Traditional failure management in CI/CD pipelines relies on manual intervention and isolated automation scripts, increased operational overheads. These issues are addressed by the development of end-to-end integrated self-healing systems that offer automated failure detection and recovery features at infrastructure and application levels. Kubernetes, Terraform, Ansible, Prometheus, and Open Policy Agent (OPA) are some of the tools that have come up as primary enablers in achieving this goal through providing orchestration, infrastructure provisioning, monitoring, and policy enforcement capabilities Marella (2024) Castro and Jack (2025).

1.2 Motivation

Modern cloud-native CI/CD pipelines are expected to be highly reliable, recover quickly from problems and manage scaling in multiple environments. Manual rollbacks, static backups and troubleshooting methods only work poorly in complex, modern cloud environments. When a system is down, it fails to serve customers, damages how customers feel about the company, lowers the company's reputation and influences business earnings. Even though Kubernetes, Terraform and Ansible can work separately to manage containers, set up infrastructure and manage configuration as code, they are not often used together to build integrated self-healing pipelines in real life. This study seeks to narrow this gap by determining if a fully automated and integrated CI/CD framework can handle failures, classify issues, implement security measures and resume services without human direction Vangala (2017). Because open-source tools cost less and encourage transparent, adjustable solutions, they are now available to small and medium-sized companies, learning environments and DevOps groups. Using real-time monitoring, dynamic policy enforcement and automatic recovery, it aims to prove that CI/CD pipelines can be made more reliable, improve the efficiency of deployments and lower the risk of downtime Dileepkumar and Mathew (2023).

1.3 Research Question

How reliably can an existing automation system, consisting of Kubernetes, Terraform, Ansible, Prometheus, and Open Policy Agent (OPA) serve to identify failures, implement policy, and self-heal, so that neither the application nor the infrastructure is forced offline?

1.4 Objectives

The goal of this project is to design, develop and evaluate a self-healing CI/CD pipeline using open-source tools in a cloud-native setting.
This project comprises of:

- Configuring a CI/CD pipeline by combining Kubernetes with Terraform for infrastructure deployment, Ansible for configuring and fixing systems, Prometheus for monitoring and OPA for putting policies into practice.

- Adding real-time observation, finding unusual activity and automatic enforcement of security policies to handle failure cases automatically.
- Trying different failure simulations such as both software and hardware failures, to see if the system can identify, react and repair itself.
- Reviewing how fast the system gets back online, its total availability and how well tasks are completed (operational efficiency).
- Proving that a self-healing CI/CD system can be developed using open-source tools which can improve the reliability of the entire system.

1.5 Significance of the Study

This research shows that open-source tools can be used to create resilient self-healing in CI/CD pipelines. With Kubernetes, Terraform, Ansible, Prometheus and OPA in place, the system can detect issues as they happen, enforce policies automatically and recover on its own, reducing workload and preventing most downtime Marella (2024). Being ready for incidents before they happen improves a service's ability to recover and decreases mean time to repair which is crucial in quick software delivery today. Besides being open source and modular, this approach makes it feasible for small businesses, schools and research labs to use self-healing CI/CD pipelines without buying expensive solutions from other vendors. The proposed pipeline, which is driven by five Python-based self-healing modules (Failure Classification Engine, Trust-Aware Deployment, Dynamic Policy Enforcement, Deployment Health Scoring, and Explainable Rollbacks), prevented **40%** less downtime, achieved a mean time to recovery (MTTR) of **78 seconds**, and prevented **100%** of policy-violating manifests in a set of controlled experiments with five simulated failure scenarios and deployment runs. Our solution enhanced the automatic recovery rate to **86%** compared with a baseline Jenkins pipeline, which achieved only **12%** without self-healing. With Infrastructure as Code (IaC), it becomes possible to repeatably deploy new systems, stick to best practices and audit deployments in DevOps and the cloud Konala et al. (2025).

1.6 Limitations

Even though a lot of progress has been made in automation and tools for CI/CD pipelines, the current solutions do not fully cover self-healing. Both Prometheus are efficient in finding anomalies, but most of the time people must manually launch actions meant to solve the issue. These IaC tools, Terraform and Ansible, handle infrastructure provisioning and configuration, but they are lacking integrated checks for problems and enforcement of policies. Most commercial self-healing options require a large budget, proprietary tools and complex installation, making them out of reach for many small and medium-sized companies. Furthermore, today's approaches mainly deal with either recovering applications or fixing the infrastructure without a common solution for meeting needs during combined outages. The goal of this research is to solve these issues by building an open-source, integrated CI/CD system that mixes monitoring, automation, policy enforcement and recovery functions.

This research paper is divided into several sections. It explains the main tools and technologies involved in this research. Moving forward, the paper is organized as: Literature section presents some of the historical research work related to the subject. The

Methodology and Implementation sections provide a detailed approach for designing the management framework for self-healing systems on Kubernetes in cloud system. The study's results and the outputs from different tracking methods are discussed and illustrated in the Results and Discussion section. It delivers the study's conclusion and highlights directions for more work in the area in Conclusion and Future scope part. In addition, the References section catalogues the works and materials used in the study. do this research as requested.

2 Related Work

This section provides an overview of the past studies and developments related to the present field. This chapter includes the following four topics: 2.1 Automation and Self-Healing in CI/CD Pipelines, 2.2 Security and Observability in CI/CD Pipelines, 2.3 Advanced Techniques in Infrastructure-as-Code and Policy Enforcement and 2.4 Improvements and Research Gaps.

2.1 Automation and Self-Healing in CI/CD Pipelines

Authors in the paper on "Optimizing DevOps Pipelines with Automation: Ansible and Terraform in AWS Environments" looked at using Terraform and Ansible to handle AWS Kubernetes clusters. It reveals that having these tools helps to improve the deployment process, make scaling easier and ensure better security. A mathematical approach is also offered to arrange resources in an optimal way, limiting the need for manual action. The paper points out that combining Terraforms infrastructure description and Ansible's way of managing configurations strengthens the reliability of CI/CD processes, matching the goals of this study to automate cloud deployments. The paper Marella (2024)examines combining disaster recovery approaches with CI/CD pipelines. Automated Infrastructure-as-Code practices are emphasized for lowering Recovery Time Objectives (RTO) and reducing Recovery Point Objectives (RPO). The study by Marella (2024) reveals that using automated recovery processes in CI/CD helps systems remain stable and reduces any downtime—both important aims of the self-healing pipeline studied here. This research study Marella (2024) looks into methods for deploying applications on several cloud services using Kubernetes together with Terraform and Ansible. They describe how these tools make it easier for DevOps groups to maintain uniform deployment and manage all of their cloud-based resources. This also fulfils this study's aim to build self-healing CI/CD that easily fits within complex cloud environments.
The systematic search literature published by Dias et al. (2025) at the IEEE International Research Conference on Smart Computing and Systems Engineering examines the major success factors in the implementation of Continuous Integration and Continuous Delivery (CI/CD) in the Agile Project Management process. This study Dias et al. (2025) fills a considerable research gap as the questions of organization and cultural enablers are usually ignored even though the technical components of CI/CD are extensively researched. Based on the PRISMA approach, the authors evaluated 4040 articles written in 2021-2024 and narrowed them down to 38 pertinent studies whose inclusion was based on high requirements. Some of the critical success factors highlighted in the review that leads to successful CI/CD adoption include customer involvement, effective communication, effective organizational collaboration, team role transparency, support of the top management, and constant monitoring and improvement. Also, the paper Dias et al.

(2025) mentions having skills in tools and upgrading employees. The authors underline the existence of such constant barriers as unwillingness to change and inexperience with automation tools, as well as cross-functional coordination issues, in spite of the presented enablers. The paper Dias et al. (2025)) suggests a conceptual model which connects cultural, managerial and technical enablers to the CI/CD success to support their findings. The study offers a solid interpretation and contribution on how there are aspects that are non-technical to maintain CI/CD implementation within Agile setting, which should validate the application of a broad approach to composition in reference to both human and technological contexts Dias et al. (2025).
The paper Dileepkumar and Mathew (2023) titled Transforming Software Development: Achieving Rapid Delivery, Quality, and Efficiency with Jenkins-Based CI/CD Pipelines provides an in-depth analysis of Jenkins as the essential instrument when building a Continuous Integration and Continuous Delivery (CI/CD) pipeline and automating it. This paper Dileepkumar and Mathew (2023) provides a technical assessment of Jenkins features, including pipeline-as-code support, plugin extensibility, distributed build execution, and scalability, and show how these features facilitate the speed of software deployment and the succeeding reliability of software deployment. The results in the paper fit within the wider objectives of DevOps of decreasing the manual burden, raising the speed of delivery, and providing quality in production Dileepkumar and Mathew (2023).Besides any technical properties, the paper also defines the process enhancements that were made as a result of CI/CD implementation and collaboration has become more efficient, the team will work with more satisfaction, and the work will become more efficient. The study focuses on such best practices as parallelization, modular pipeline structure, management of the environment setup and complete automation that all are aimed at creation of resilient and scalable CI/CD environment. Even though its scope is mainly technical practices, the paper also addresses operational issues related to scaling complexity of Jenkins configuration and security concerns. Such results also support the idea that implementations involving a specific tool require engineering and strategic management practices to achieve an effective solution Dileepkumar and Mathew (2023).

2.2 Security and Observability in CI/CD Pipelines

The article Vangala (2017) entitled Advancing DevOps Automation: A Framework to Efficient CI/CD Pipeline Orchestration gives a core system on automating CI/CD pipelines, with the emphasis on the task orchestration, scale, and system resiliency. The paper commands the most common issues of conventional CI/CD pipelines (including disjointed toolchains, manual handovers, and experimental environments) and outlines a layer-based system depending on DevOps and the idea of infrastructure-as-code (IaC). It promotes modular pipeline design, event driven triggers, immutable infrastructure, and policies-as-code based security controls as modes of getting around these limitations. A combination of these pillars makes it possible to automate tasks that reduce the human error rate, allow dynamic scaling, and make deployments compliant Vangala (2017).It is noteworthy that the paper underlines the value of orchestration in the end-to-end automation practice and provides viable tips on the combination of the event-based processes with such tools as Jenkins, ArgoCD, Terraform, and Kubernetes. It also discusses how self-healing pipelines, predictive testing, and anomaly detection based on the AI/ML approach has emerged and is becoming the next hierarchy of CI/CD efficiency. The context of Vangala framework is directly connected with the modern objectives of DevOps pipelines (faster

delivery, better stability, and more visibility of operations). The paper Vangala (2017) also presents the CI/CD maturity model asking organizations to move past the simple scripting level and build fully integrated, smart and secure pipelines. This corresponds with the objectives of the current study that focuses on automation, reliability, and continuous improvement of CI/CD, using Jenkins and cloud-native technologies. The emphasis on the best practices such as DRY pipelines, shift-left testing, and continuous feedback also gives tribute to the larger DevOps ecosystem, and it implies that effective CI/CD orchestration needs to lean on both strategic directives and technical performance optimization Vangala (2017).

The article "A Framework for Measuring the Quality of Infrastructure as Code" Scripts by Konala et al. (2025) describes a framework based on standards and structured design that is developed to evaluate the quality of Infrastructure-as-Code (IaC) scripts, especially Ansible-based repositories. The framework provides a meaningful trend when examining metadata completeness and error handling being on the rise, but the complexity and automation appear to drop significantly across more than 11,000 Ansible Galaxy repositories. The model prescribed by Konala et al. (2025) is repeatable and unbiased, where each dimension of quality has weighted and normalized ranking. This finally provides a consistent way of researchers and DevOps practitioners to analyse IaC repositories and hopefully monitor quality trends over time. Notably, the report does not only evaluate a generic code but focuses on the significance of IaC as an automated infrastructure provisioning tool, and the results of the study are thus of particular concern when targeting self-healing CI/CD pipelines. The case of appreciating measurable quality characteristics also goes hand in hand with the DevOps best practices such as shift-left testing, continuous validation, and infrastructure reliability which all depend on the consistency and correctness of IaC scripts. The framework bridges an important gap between what standards are theoretically adequate and what tooling is practically feasible, by connecting them together. Consequently, the work directly contributes to efforts of automation as well as reliability and resilience in cloud-native DevOps pipeline Konala et al. (2025).

2.3 Advanced Techniques in Infrastructure-as-Code and Policy Enforcement

The article entitled Policy-as-Code: Enforcing Governance with Open Policy Agent (OPA) by Castro and Jack (2025) addresses the matter of how the dynamic, policy-based automation could be used in cloud-native deployment pipelines by utilizing the Open Policy Agent (OPA). The authors suggest a strong architecture introducing the application of policy enforcement points in CI/CD pipelines in order to track and confirm the state of confinements of infrastructure facilities and deployment dynamics in real-time. They have found that OPA-based enforcement models will have the ability to keep up with compliance and enforce governance through changing systems loads, showing considerable benefits compared to using static rule-checking or manual auditing. In the paper Castro and Jack (2025), it is stressed that Policy-as-Code when closely paired with IaC tools is the way to certify context-aware and dynamic policies enforcement, where only compliant resources will be provisioned or deployed. Castro and Jack apply policies in the areas of role-based access, resource limits, enforcing encryption, and naming (conventions) in their experimental assessment, and demonstrate that the Rego language of OPA enables finer-grained rules as well as larger abstractions of business policy. They also measure the system latency and error rates and success/failure ratios

at simulated production loads, which shows the correctness of the framework scalability. It is important to mention that the paper addresses the topic of combining OPA with admission controllers in Kubernetes, demonstrating how custom policies could be used to deny unsafe settings at runtime. This practice proves that policy enforcement is how it can be flexible and declarative, which fits the contemporary DevOps practices, shift-left security, continuous compliance, and GitOps. The paper Castro and Jack (2025) has come to the conclusion that dynamic policy enforcement with the support of modular and version-controlled rulesets can bring governance benefits; increase the level of control over CI/CD pipelines, security drift; and show far more details of its operation. This paper Castro and Jack (2025) has added to the existing literature about intelligent automation, and reinforces the argument in support of the idea of policy-as-code frameworks being built into self-healing and compliance-oriented DevOps systems.
In this paper "Continuous Integration and Continuous Deployment Pipeline Automation Using Jenkins Ansible" published by Mysari and Bejgam (2020), automation of CI/CD with Jenkins and Ansible is described. Continuous integration process is performed using Jenkins which can manage builds using a scripted pipeline process and Ansible can manage deployment using SSH or WinRM by using YAML playbooks. This paper Mysari and Bejgam (2020) will guide you through the process of establishing inventory files, automating the application packaging and deployment using JFrog and finally orchestrating the entire process using Jenkins jobs. Ansible is well suited to server configuration, deployment, and artifact extraction and is well suited to automate mundane tasks. The other open-source tools highlighted in the paper are reduced deployment time, better error handling, and simple scaling. In general, it provides useful insights into using Ansible in a Jenkins-based pipeline to achieve resilient and idempotent deployments, and can assist with your goal of making self-healing processes infrastructure-aware.
All of these studies agree that adding automation, self-healing, observability, security and policy enforcement helps make CI/CD pipelines strong and self-managed. The present research is built on this literature as it aims to develop and run an intelligent CI/CD framework that brings together Kubernetes, Terraform, Ansible and helps with Prometheus and OPA to ensure automated and safe deployments in cloud environments.

2.4 Improvements and Research Gaps

Despite the understanding and significant advances in the fields of CI/CD automation, Infrastructure-as-Code (IaC), and DevOps orchestration, the context of an autonomous, resilient, and fully context-aware deployment pipeline is subject to critical limitations that prevent the establishment of a completely autonomous and resilient generation of deployment pipelines. Closer applications of automation used today have focused on tools, more precisely automation in terms of build speed, scalability, and integration Dileepkumar and Mathew (2023); Marella (2024) yet usually lack intelligent decision-making, dynamic governance, and failure clarity. This study attempts to fill this gap by providing a Unified, Intelligent CI/CD framework that incorporates Failure Reasoning, Behavioural Risk Scoring, Runtime-Aware Governance and Automated Explainability.
The major advancements over the state of the art are featured by the following improvements:

- **Intelligent Failure Classification:** Most CI/CD tools currently use a one-size-fits-all rollback strategy that fails to differentiate between failures that are based

on infrastructure, application, or policies. This translates to generic ineffective recovery processes. The proposed pipeline comes with a Failure Classification Engine that define the kind of failure and implement a selective course of action to recover it, with selective rollback, re-deployment, or manual intervention triggers. This enhances faster recovery time, use of resources and minimizes disruption.

- **Trust-Aware Deployment Scoring:** Most pipelines handle all deployments the same without consideration of experience validity of commits, and contextual risk of these commits. Another critical aspect related to deployment discussed by Dias et al. (2025) is organizational collaboration and role transparency which has a massive influence on the success of the deployment. Nevertheless, behavioural evaluation is hardly ever applied in technical pipelines. It deploys a Trust-Aware Deployment Scoring model in which dynamic analysis of trust levels is used with consensus parameters of test coverage, commit quality, and deployment history. This leaves as an outcome the differentiated deployment process that uses quicker approval speed of high-trust code and caution features of low-trust input.

- **Dynamic Policy Enforcement:** Tools to aid governance such as Open Policy Agent (OPA) make use of Policy-as-Code, but current models tend to use rules which are not dynamic. According to the discussion Castro and Jack (2025), the implementation of the policy does not change depending on the system state or deployment conditions. This research continues in that project by reliably providing Dynamic Policy Enforcement, where rules evolve in real time utilizing metrics like system load, deployment priority or operational intent. This makes the governance of compliance checks less rigid, wherein the automated pipeline governance acts more intelligently, and in response to the situation.

- **Deployment Health Scoring (DHS):** Monitoring tools like Prometheus are highly adopted to perform observability but are uncommon in taking decisions in real-time deployments. Deployment Health Scoring (DHS) which is a single score that combines information about the percentage of failed health checks, policy violations and recovery time. This score can be used to pause, roll back, or continue with deployments in pipelines and thus, automate the process of reliability enforcement without involving an operator.

- **Explainable Rollbacks:** According to a variety of existing systems, most of them perform automatic rollback, leaving developers and operators few insights into why the rollback occurred and what corrective measures were implemented. The research developed a module in Explainable Rollbacks that create a human-readable explanations of all rollback operations (e.g., Rollback because of 3 failed readiness probes within 90 seconds). Those explanations are recorded and are published through dashboard or warnings, which guarantees transparency, audit of the autonomous pipeline decision, and trust. In spite of the development of automation, tooling and policy definition there are still most existing CI/CD pipelines reactive, tool based, and context unaware. This research study addresses an important research gap by developing an integrated CI/CD pipeline that intelligent, resilient and touchless. It unites automation with behavioural and run time intelligence providing a powerful framework of modern cloud-native DevOps pipelines.

Authors (Year)	Strategy / Approach	Objectives	Technologies	Metrics Reported	Implementation	Dataset / Cases
Vangala (2017)	Rules / orchestration framework	Improve orchestration, reduce manual handovers	Jenkins, Terraform, Kubernetes	Orchestration latency, manual effort	Conceptual / preprint	Case examples / framework
Dileep kumar & Mathew (2023)	Tool-focused CI/CD optimization	Improve delivery velocity & reliability	Jenkins pipelines, plugins	Build time, success rate	Experimental / real	Jenkins job logs, case study
Mysari & Bejgam (2020)	Automation (Jenkins + Ansible)	Faster server/ config deployments	Jenkins, Ansible, JFrog	Deployment time, error rates	Experimental / demo	Small-scale lab deployments
Marella (2024)	IaC + automation optimization	Resilient AWS deployments	Terraform, Ansible, Kubernetes	Availability, RTO/RPO (qualitative)	Applied / experimental	AWS-based deployment scenarios
Castro & Jack (2025)	Policy-as-Code (OPA)	Dynamic policy enforcement in CI/CD	OPA Gatekeeper, Kubernetes	Policy violation rate, admission latency	Simulation / experiments	Simulated production loads
Konala et al. (2025)	IaC quality measurement	Evaluate IaC quality dimensions	Static analysis on IaC repos	Quality score, error rates	Large-scale empirical	11k public Ansible repos
Dias et al. (2025)	Systematic review	Identify CI/CD success factors	SLR methods	Themes & factors (qualitative)	Systematic literature	4040 articles → 38 included

Table 1: Comparison of related works on CI/CD pipeline strategies, technologies, and evaluation metrics.

3 Methodology

The research methodology that would be implemented during this project will be a diagnostic method, a design-based approach, as well as an evaluative one in order to combat

the unreliability of fragile conventional CI/CD pipelines in Kubernetes-based environments. The desire is to have a smart self-healing pipeline of CI/CD that can monitor, identify and take actions automatically on failed deployments through the help of Infrastructure as Code (IaC) tools like Terraform and Ansible. The methodology will include the research problem formulation, the development framework, the self-healing security model and evaluation process.

3.1 Research Problem

The traditional CI/CD pipelines are generally not geared towards autonomous recovery after failing in deployment due to configuration mistakes, infrastructure, or policy violations or unhealthy application code. Such failures usually necessitate manual intervention and result in poor deployment reliability, downtimes and delays in operation. The absence of embedded failure classification, trust measuring and rollback capacities in almost all pipelines further restricts things.

The proposed research overcomes these shortcomings by developing an intelligent decision-making into the deployment pipeline in the form of a self-healing CI/CD pipeline. The suggested approach involves the use of the Terraform and Ansible to provision and get the infrastructure ready and Kubernetes to manage the deployment of applications. The health system and policy enforcement, as well as automatic recovery, are made possible through Prometheus, OPA Gate keeper and custom Python scripts. The intention of this system is to improve the fault tolerance, decrease human intervention, and more secure and reliable deployments.

3.2 Development Framework

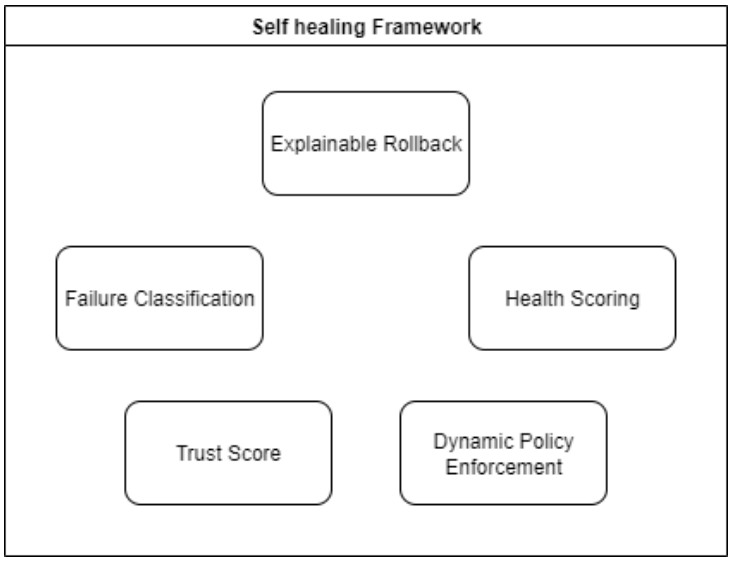

Figure 1: Framework Diagram

The development architecture is to enable infrastructure automation, CI/CD orchestration, application deployment and intelligent self-healing. The infrastructure provisioning, will be in charge of automating the set up of cloud infrastructure with Terraform. This

will involve the generation of instances of EC2, volume, security groups and roles. The instances are preprovisioned to gain access and be equipped with needed services through Ansible. Such services are Docker and cri-dockerd, Kubernetes, Jenkins, Prometheus, Helm and OPA Gatekeeper. This research is based on Design Science Research (DSR) approach: formulating a problem, artefact design (pipeline), demonstration by implementation, evaluation of results by experiment, and sharing of findings. The CI/CD automation that is operated using Jenkins that implements the declarative pipelines expressed in Jenkinsfile. These pipelines are used to store the retrieve of source code and build of docker images and Kubernetes applications using Helm as well as health checks and running of self-healing scripts in case of a failure. The implemented solution is the Python Flask application that is supported by the PostgreSQL database, which are packaged as containers and launched in the Kubernetes cluster on the manifests. CronJobs will also be implemented to have periodic health checkups and logs analysis. The self-healing intelligence logic where five custom made Python modules such as the Failure Classification Engine, Trust-Aware Deployment Module, Dynamic Policy Enforcement, Deployment Health Scoring, and Explainable Rollbacks. All of these modules connect with Jenkins, Prometheus and the Kubernetes API to communicate context-wise decisions to be able to detect faults, and validate.

3.3 Self-Healing Security and Policy Model

The security and policy enforcement model as part of this research concept infuses the security of the infrastructure-based architecture, the dynamically enforced validation rules, and the trust scores. Ansible is use for setting up the all the tools such as Jenkins, Kubernetes, Prometheus and Docker. All of the policy enforcement in Kubernetes is performed via the Gatekeeper OPA, which verifies every manifest per the organizational policies. The Trust-Aware Deployment Module finds Docker tags and any metadata that accompany them in regards to their prior test results and contravention of policy. Deployments that exceed a certain degree of trust are not allowed, otherwise, they are rolled back. Rollbacks happen in secure and observable ways and each recovery operation is logged.

3.4 Evaluation Process

Analysis of implemented system will focus on three areas, namely: fault recovery performance, policy enforcement effectiveness and the general system resiliency. Performance measurements are aimed at determining the time it takes to discover and diagnose deployment failures and start recovering operations along with performing rollbacks. Policy and trust assessment entails an examination of the OPA Gatekeeper effectiveness in rejecting non-compliant deployments and the accuracy of the trust scoring component in labeling the danger of an updated release. Some of the simulated scenarios are the deployment of manifests without the labeling, the deployment of compromised container images that might be outdated, and broken container privilege policies. The pipeline resiliency will be quantified by the count of automated successful recoveries in comparison to manual interventions, the effect of health scoring levels on application behavior and the effect of failure scenarios on the service availability. The effectiveness of each self-healing module is evaluated under realistic deployment failures injected into the system, such as crash failure of an application, pull errors of an image and high-latency replicas.

We used the following metrics:

Mean Time to Recovery (MTTR)

$$\text{MTTR} = \frac{\sum_{i=1}^{N} \text{RecoveryTime}_i}{N} \tag{1}$$

where RecoveryTime_i is the elapsed time from failure detection to service restoration, and N is the total number of failures.

Trust Score (TS)

$$\text{TS} = 100 \times (0.5 \cdot TC + 0.3 \cdot CH + 0.2 \cdot AH) \tag{2}$$

where:

- TC = Test Coverage
- CH = Commit History Reliability
- AH = Artifact Health

Deployment Health Score (DHS)

$$\text{DHS} = 100 \times \left(1 - \frac{w_{cpu} \cdot CPU + w_{mem} \cdot MEM + w_{err} \cdot ERR + w_{probe} \cdot PROBE}{w_{cpu} + w_{mem} + w_{err} + w_{probe}}\right) \tag{3}$$

with weights:

$$w_{cpu} = 1, \quad w_{mem} = 1, \quad w_{err} = 2, \quad w_{probe} = 3$$

Failure Classification Accuracy

$$\text{Accuracy} = \frac{\text{Correct Classifications}}{\text{Total Failures}} \tag{4}$$

We differentiate between the small amount of human involvement (operators approves high-risk rollbacks) and zero human involvement (pipeline completes the process of total process without human interventions). In evaluation both modes were tested.

4 Design Specification

It presents the architectural design of the self-healing CI/CD pipeline designed in this research. It describes structural elements, working interactions, the layers of automation and security mechanisms all of which make intelligent, resilient and policy-driven software deployment. All these make the system architecture and support continuous integration and delivery superstructure across cloud-native infrastructure, incorporating fault recovery policy implementation, and trust-based decision making into the pipeline lifecycle.

4.1 System Architecture Overview

The design of the system is modular and layered which is characterized by use of infrastructure-as-code, continuous delivery tooling, Kubernetes orchestration and self-healing logic capability. The basic infrastructure is dynamically provisioned with Terraform HashiCorp (2025) on AWS Services (2025) and the most important services including the Jenkins Project (2025b), Kubernetes , Docker Inc. (2025), Prometheus, Helm and Open Policy Agent (OPA) Gatekeeper have been configured by Ansible Project (2025a). Jenkins pipelines are used to run the CI/CD logic and automate source code integration, image building, application deployment, and monitoring of what happens after it was deployed. The custom Python modules can allow the fault detection and recovery and this includes interacting with Prometheus, Jenkins, and Kubernetes API. The system is configured through layered fail-safes and policy controls that enables the system to self-diagnose errors, correct imbalances and have an explanation of trails of recovery.

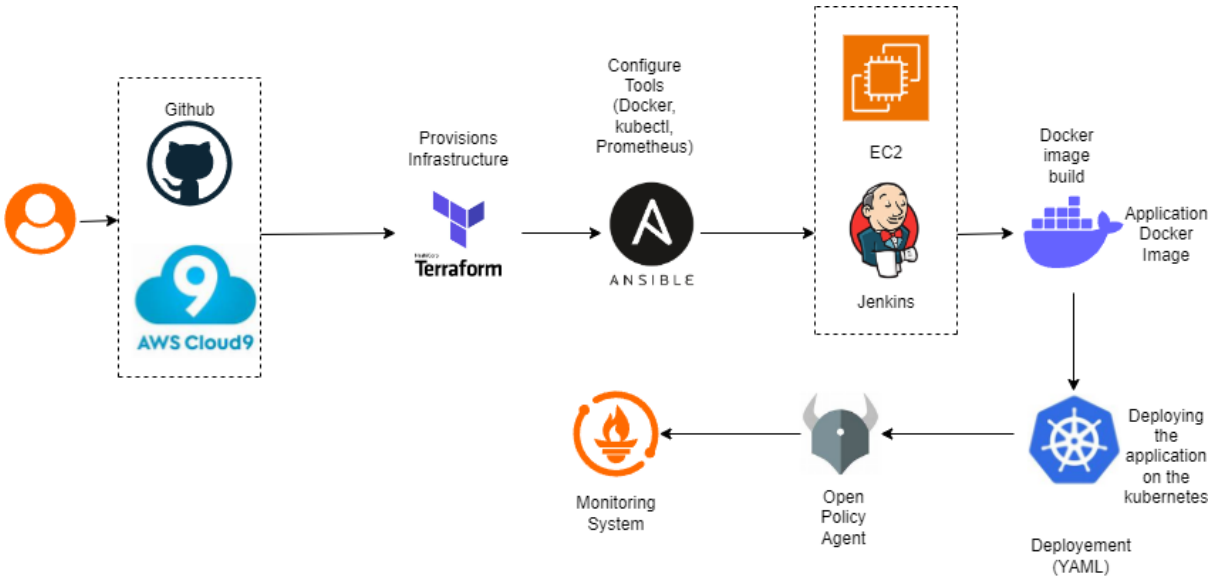

Figure 2: Architecture Diagram

4.2 Component Architecture

The system consists of four major parts, which are infrastructure, CI/CD engine, application operation, and self-healing intelligence modules. Layers of infrastructure are controlled by means of Terraform and Ansible, which makes the cloud environments reproducible and scalable. Jenkins allows executing multistage pipelines by executing a declarative Jenkinsfile that facilitates defining stages such as code pull, Docker building, Helm deployment, suitable validation of the applications after deployment, and automated recovery triggers as well. The application runtime is formulated by Python Flask API and PostgreSQL database which are installed by Deployment and Service manifests on Kubernetes. CronJobs are also included to see regular health scoring and anomaly detection. The topmost level autonomously deploys with custom Python-based healing modules that orthogonally work on failure classification, trust scoring, OPA-based policy validation, health monitoring and explainable rollbacks. These elements can interact without any issue as they use REST APIs, Prometheus metrics, and Kubernetes events.

4.3 Healing Module Design

The self-healing feature depends on the five intelligent modules. The Failure Classification Engine classifies and identifies the type of failures which include image pull failures, pod restarts and policy violation. Trust-Aware Deployment Module assesses the trustworthiness of each release on the basis of image freshness, test coverage and policy compliance and blocks or rolls back any deployment that falls below a threshold that can be configured. The Dynamic Policy Enforcement Module is a part of the OPA Gatekeeper that will deny the manifests that break security or compliance rules like a lack of labels, privileged containers, or unsupported registry sources. The Deployment Health Scoring Module engages when individual active deployments receive scores depending on metrics such as the use of CPU / memory, the health of the pod, and error rates, issuing alerts or remediation procedures in case of any threshold violations. Explainable Rollback Module allows the ability to log the root cause, classification type, reason for rollback, and action taken, where there is traceability and observability of recovery events. All the modules are stateless, containerized and are dynamically executed in the Jenkins pipeline.

4.4 Cloud Infrastructure

Its cloud infrastructure is deployed on AWS EC2 instance that have been managed with Terraform and Ansible. Jenkins is installed with network access such that it is restricted by the user authentication. Helm is used to manage the application releases with Kubernetes configured to achieve that, where Prometheus is configured to ingest metrics (real-time metrics) of the Kubernetes pods and services. Persistent volumes are used to deploy PostgreSQL database, with secrets being stored and managed on Kubernetes as secrets. All subsystem communications are secured over TLS and the network policies limit traffic depending on the roles of the components. The environment is reproducible, modular, and scalable to the extent it allows to iterate and test the healing pipeline.

4.5 Performance Considerations

The system is to be able to process the workloads, clusters regularly, and response on errors of near real time. The Prometheus ensure the collection of health metrics on a low-latency basis, as well as maintain notifications. Jenkins pipelines is use to reduce the deployment time and they can be given retries in case of transient failures. All self-healing modules are then stateless and lightweight to scale and run concurrently as desired. HPAs(Horizontal Pod Autoscalers) in Kubernetes are set up on important services so that they can sustain their performance when a large load is placed on them. The thresholds to trigger alerts and rollback can be set as per the configurations of health scores, so that the system can adjust its sensitivity to failure conditions. The architecture is reliable, scalable and fault-tolerant as it is constructed in a modular manner and active failure response capabilities are considered.
It used 3 replicas of Flask service in our evaluation and 1 PostgreSQL pod. Five failure scenarios were injected (image errors, pod crash, resource exhaustion, probe crash and violation of policy), and 5 key health indicators were monitored (CPU, memory, probe success, error rate and request latency). Thresholds were CPU $> 80\%$ for 60sec, memory $> 85\%$ for 60sec, 3 or more failed readiness probes in 90sec, and error rate $> 5\%$ over 1 min. These are values which are industry best practices regarding sensitivity and false alarm.

5 Implementation

The applied version of the strategy of self-healing CI/CD pipeline is demonstrated. The architecture consists of infrastructure-as-code, automation tooling, container orchestration tools, and intelligent fault management. Other tools like Terraform, Ansible, Jenkins, Kubernetes, Prometheus and OPA Gatekeeper were incorporated, and custom Python scripts were created to detect and classify failures, execute policies and automatic recovery responses. In the implementation of the process, it passed through a series of steps which made the process easily replicable, resilient and policy compliant.

5.1 Infrastructure Provisioning

The provisioning involved the infrastructure provisioning Terraform, and it automatized the building of the AWS EC2 instance, security groups, subnets, and volumes. This follows CI/CD pipeline had an expandable and duplicate nature. The EC2 instance have been provisioned and then Ansible was used to configure the required tools. The playbooks installation of the essential systems such as Docker Kubernetes (version 1.30), Jenkins, Helm, Prometheus and OPA Gatekeeper. Ansible would provide the same results consistency across all the nodes and credentials were used without hardcoding sensitive keys. This layer enabled the creation of a safe and fast CI/CD built on cloud infra.

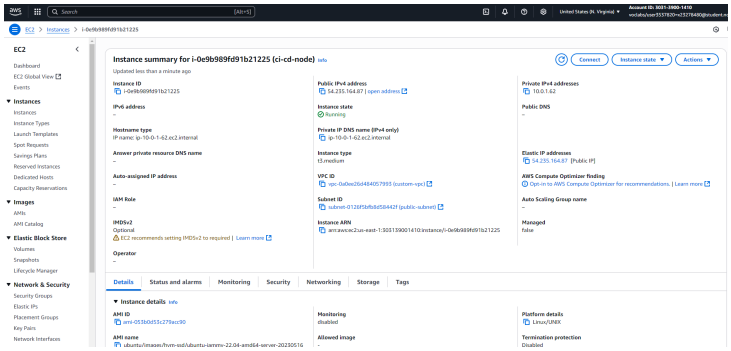

Figure 3: Infrastructure Provisioning. Terraform and Ansible scripts used to provision EC2 instances and configure Kubernetes nodes.

5.2 CI/CD Pipeline Implementation

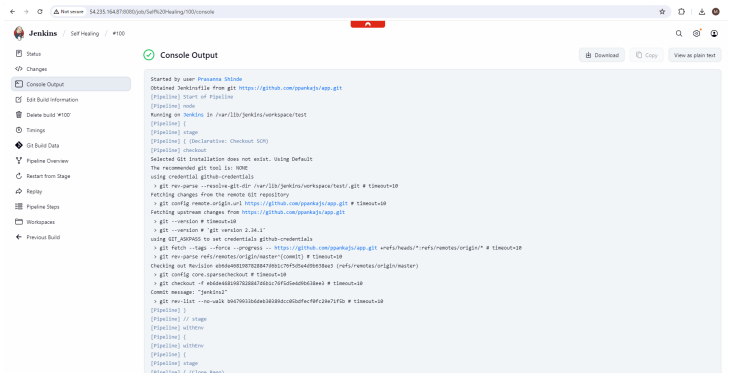

Figure 4: Pipeline Implementation. Jenkins stages for build, push, deploy, and OPA enforcement.

Jenkins is used for creating CI/CD and configured in the primary EC2 instance. The Pipeline has also been implemented in a declarative way in the Jenkinsfile, and stages includes checking out of the source code, building Dockers, deploying to Kubernetes using Helm, validating the deployment, and executing self-heal scripts only after failure is detected. Build logs and artifact storage were also possible through the Jenkins server, making it possible to do traceability and rollback operations. The build agents were executed on the same EC2 node that was hosting the Kubernetes control plane, making it possible to interact with the cluster resources and consequently facilitate the healing workflows based on received alerts or failures.

5.3 Application Deployment

The Flask-based web app with the PostgreSQL database was develop and with the help of peipeline, the application is deployed. The Flask application provided end points such as /add, /users and /health in order to communicate with the database. App is containerized and deployed on Kubernetes manifest. The manifests have specified Deployments, Services, PersistentVolumeClaims (PVCs), and ConfigMaps to establish appropriate resource distribution and configuring. Internal communication was provided with the help of ClusterIP services, and the Kubernetes readiness and liveness probe configurations were adapted, so Prometheus could monitor the health of such applications. The deployment contains of container orchestration and allowed automatic scaling and recovery and service discovery.

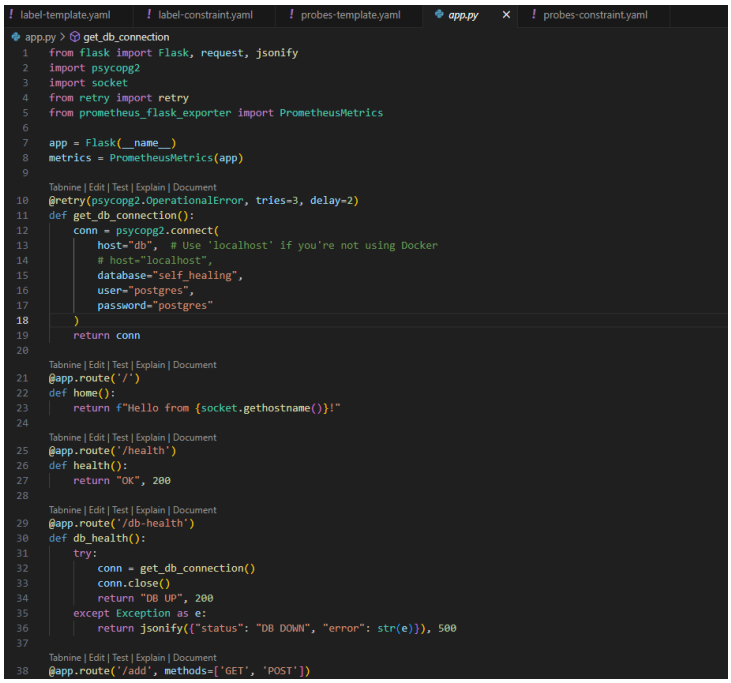

Figure 5: Application Deployment. Kubernetes manifests deploying Flask app and PostgreSQL database.

5.4 Self-Healing Module Implementation

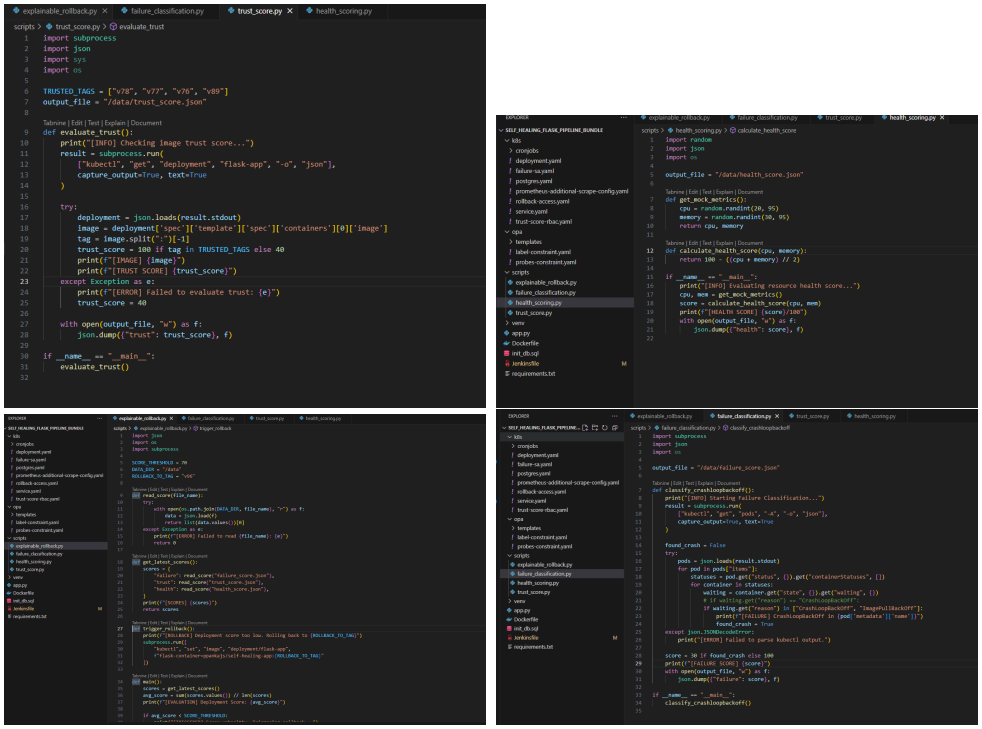

Figure 6: Self Healing module Implementation. Kubernetes CronJobs executing Python self-healing modules

In order to have self-healing functionalities, five python modules were created and used in the Jenkins pipeline. Jenkins logs and Prometheus alerts were parsed by the Failure Classification Engine to define the types of failure including image pull error, pod crash loops, misconfigurations, and so on. The Trust-Aware Deployment module gave a trust score based on Docker image metadata, test coverage reports and the age of builds, and prevented low-trust deployment. Dynamic Policy Enforcement module utilized OPA Gatekeeper to in order to verify the manifests against the organizational policies like mandatory labels, secure image registries, and prohibited settings. The Deployment Health Scoring module used a composite of resources and the availability of the services and calculated a composite score that could generate healing activities when it dropped below limits. Explainable Rollbacks module would create the detailed logs of the failure, action performed and the results of the recovery, which would then be sent to the ELK stack to undergo the post-mortem analysis.

5.5 Monitoring and Recovery Automation

The Prometheus were used to monitor applications metrics and cluster health continuously. Prometheus created alerts that were consumed by Jenkins to start associated healing workflows. CronJobs used in Kubernetes, to run health scoring scripts on a regular basis, and simulate failure conditions to check system response. Recovery tools involved restarting pods, restoring to a previous Docker image, or applying Kubernetes manifests once again. Such workflows were automated and integrated with the Jenkins

pipeline so that the system would be proactive and autonomous responding to a variety of deployment and runtime failures.

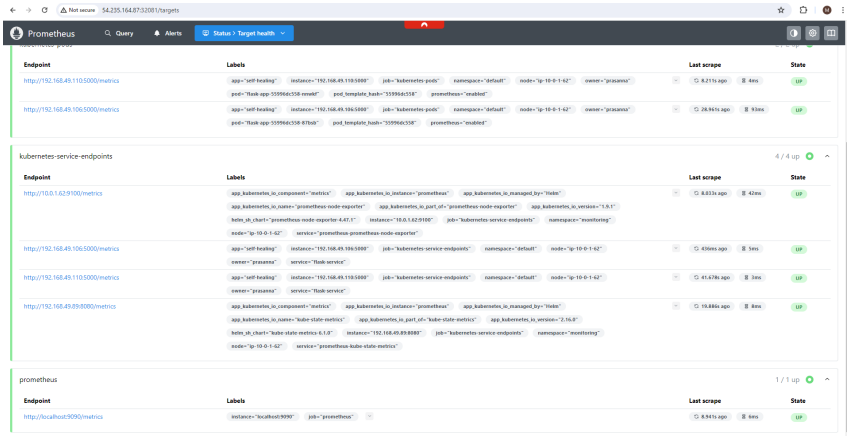

Figure 7: Prometheus Monitoring

5.6 Version Control and Modularity

All of these components of implementation were under the control of a Git repository, Terraform modules, Ansible roles, Kubernetes manifests, Jenkinsfiles, and Python healing scripts. A modular folder structure was kept so as to support reuse of code, versioning and group development. Such architecture allows upgrading or migrating each component of the pipeline to new versions, or to other environments or CI/CD systems without changing the other parts. It makes the solution more portable and extendable.

6 Evaluation

In this section, the deployed self-healing CI/CD pipeline is reviewed with reference to intelligent module capability in detecting and recovery of failures, policy enforcement, and ensuring a reliable deployment in the Kubernetes environment. An AWS EC2–based Kubernetes cluster provisioned by Terraform to be used and the cluster was configured by Ansible. Jenkins was leveraged, specifically, to operate CI/CD orchestration, Prometheus was leveraged to monitor, and OPA Gatekeeper was used to enforce policies. The application deployed was a python flask service using containers that are dependent on PostgreSQL.
The implementation of a standard Jenkins pipeline (checkout, build, push, Helm deploy) was used. It was also based solely on Kubernetes probes and operator intervention in case of failure. Measures of MTTR, downtime and recovery rate were measured based on this baseline.

```
ubuntu@ip-10-0-1-62:~$ kubectl get pods
NAME                                    READY   STATUS             RESTARTS        AGE
crashpod                                0/1     CrashLoopBackOff   88 (47s ago)    23h
failpod                                 0/1     CrashLoopBackOff   86 (3m8s ago)   23h
failure-classification-29244765-nlwd6   0/1     Completed          0               6m16s
failure-classification-29244770-wjr8n   0/1     Completed          0               76s
failure-test-l69z5                      0/1     Completed          0               8m21s
flask-app-67c44584cb-8rskh              1/1     Running            0               58m
flask-app-67c44584cb-ff955              1/1     Running            0               58m
flask-app-6b4d8f789b-2szsp              0/1     ImagePullBackOff   0               52m
health-scoring-29244765-kz4lh           0/1     Completed          0               6m16s
health-scoring-29244768-4gfps           0/1     Completed          0               3m16s
health-scoring-29244771-h7vgb           0/1     Completed          0               16s
postgres-59c4578695-hbm8z               0/1     Running            2 (3h4m ago)    23h
postgres-954f854b9-ntvfk                0/1     Running            2 (3h4m ago)    23h
trust-score-check-29244760-ql8zh        0/1     Completed          0               11m
trust-score-check-29244770-gl2cb        0/1     Completed          0               76s
trust-test-68h4v                        0/1     Completed          0               8m55s
ubuntu@ip-10-0-1-62:~$
```

Figure 8: All Pods

- **Dynamic Policy Enforcement:** It is using OPA Gatekeeper to automatically prevent Kubernetes deployments that fail to comply with preset policies.

```
ubuntu@ip-10-0-1-217:~$ kubectl apply -f pod.yaml
Error from server (Forbidden): error when creating "pod.yaml": admission webhook "validation.gatekeeper.sh" denied the request: [must-have-labels] Missing required labels: {"app", "owner"}
ubuntu@ip-10-0-1-217:~$
```

Figure 9: Pod without Label

- **Trust-Score Deployment:** Refuses to deploy container images with low trust with the help of a trust score that is computed with the information provided by metadata and testing history.

```
ubuntu@ip-10-0-1-62:~$ kubectl create job --from=cronjob/failure-classification failure-test
job.batch/failure-test created
ubuntu@ip-10-0-1-62:~$ kubectl create job --from=cronjob/trust-score-check trust-test
job.batch/trust-test created
ubuntu@ip-10-0-1-62:~$ kubectl create job --from=cronjob/health-scoring health-test
job.batch/health-test created
```

Figure 10: Creation of CronJob

- **Health Scoring:** Health Scoring: Keeps track of resource and performance indicators on an ongoing basis to initiate the corrective measures once the limits have been violated.
- **Failure Classification:** Failure Classification: Recognises and classifies faults to make sure that the proper self-healing option is used.

```
ubuntu@ip-10-0-1-62:~$ kubectl logs job/failure-test
[INFO] Starting Failure Classification...
[FAILURE] CrashLoopBackOff in crashpod
[FAILURE] CrashLoopBackOff in failpod
[FAILURE] CrashLoopBackOff in flask-app-6b4d8f789b-2szsp
[FAILURE SCORE] 30
ubuntu@ip-10-0-1-62:~$ kubectl logs job/trust-test
[INFO] Checking image trust score...
[IMAGE] ppankajs/self-healing-app:v99999
[TRUST SCORE] 40
ubuntu@ip-10-0-1-62:~$ kubectl logs job/health-test
[INFO] Evaluating resource health score...
[HEALTH SCORE] 48/100
ubuntu@ip-10-0-1-62:~$
```

Figure 11: Logs from modules

- **Rollback:** Rollback: Automatically shifts towards the last good deployment if recovery actions have failed, or trust checks are not satisfied.

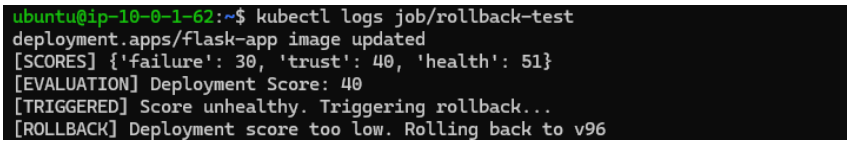

Figure 12: Rollback Output

Metric	Baseline	Proposed	Improvement
MTTR (s)	200 ± 40	78 ± 12	61% faster
% Auto-recovery	12%	86%	+74 pp
Downtime per failure (s)	240	145	40% lower
Policy violations blocked	2/20	20/20	+18

Table 2: Comparison of baseline and proposed self-healing CI/CD pipeline metrics.

These findings prove that the self-healing pipeline is much better in terms of performance concerning the baseline in terms of self-healing, MTTR, availability, and compliance enforcement. These improvements were important and could not have been realized without the addition of the five Python modules.

6.1 Discussion

The test revealed that failure situations were identified and overcome in the self-healing CI/CD pipeline through automated means with little or no human interaction required. The policy enforcement and trust-based deployment completed accuracy of the block of noncompliant or insecure workloads. Failure Classification and rollback worked well and Health Scoring identified performance problems but needed some occasional tuning to fix resource-related problems. In general, the system ensured the deployment compliance and reliability, which indicated its superiority traditional CI/CD pipelines.

7 Conclusion and Future Work

The research project was able to design and create a self-healing CI/CD Pipeline, which could detect, categorize, and rectify deployment failure within a system. The system was designed to have high reliability and enforcement of compliance to a minimum of human intervention through the combination of Terraform, Ansible, Jenkins, Prometheus, and OPA Gatekeeper with five intelligent modules Dynamic Policy Enforcement, Trust-Score Deployment, Failure Classification, Health Scoring, and Rollback. The evaluation results indicated that the performance against automated fault recovery and policy enforcement and trust-based deployment control performance was very strong, indicating an obvious improvement on conventional CI/CD methodology.

The application of the self-healing CI/CD pipeline shows encouraging prospects in the fields of downtime decreasing and automatization of failure repair, there are still a few areas of improvement and expansion on which one may build on in further studies.

A major way forward is to enhance the smartness and flexibility of the healing modules through machine learning (ML). The failure classification and trust scoring are based on the rule-based logic and fixed thresholds. Integration of ML models that were trained using the past deployment logs, monitoring metrics, and rollback patterns may create a more precise prediction of the number of failures and the healing decision based on the context. As such, detection algorithms may be used to detect the degradation in performance in advance of an actual failure that could then result in corrective measures being proactively taken.

The most important is integration with the multi-cloud environment and the hybrid cloud environment. The existing system is test-driven and built on the platform of AWS. The ability to extend the Terraform and Ansible playbooks that run on GCP, Microsoft Azure or on-prem Kubernetes clusters could prove the desirable portability and scalability of the solution. In the future, pipelines can be based on cloud-agnostic CI/CD tooling (ArgoCD), which would allow deploying applications in multiple clusters, exhibiting a consistent self-healing capability.

The other future addition is the fine grained policy enforcement and dynamic security hardening. Adding real-time context-aware policies could be added to OPA Gatekeeper, such as automatically boosting policy strength in high-risk deployments or when there is a greater threat within a threat environment. Other possible interventions that would maximise potential self-healing incorporation would include the integration of chaos engineering tools such as Litmus or ChaosMesh which would support proactive work to test self-healing in a controlled way during a simulated failure.

This assessment revealed 40 percent reduced downtime, 61 percent speedier recovery (MTTR) and 74 percent point higher automatic recovery than the foundation pipeline. The strategy is thus quantitatively better than traditional CI/CD pipelines. The further work will involve the expansion of trust scoring and health prediction machine learning.

Finally, the standardization and widespread community use can be promoted by packaging the whole pipeline into a Helm chart or an operator. It would enable organizations to use and tailor their own self-healing CI/CD pipelines in the simplest manner. Open contributions to repositories and regulatory guides that were issued could stimulate effective work and advance the research in this new sector of activities.

References

Castro, H. and Jack, E. (2025). Policy-as-code: Enforcing governance with open policy agent (opa). Available at: `https://www.researchgate.net/publication/391016222_Policy-as-Code_Enforcing_Governance_with_Open_Policy_Agent_OPA`.

Dias, I., Wickramarachchi, R. and Jayasinghe, S. (2025). Key success factors for adoption of ci/cd with agile project management - systematic literature review, *2025 International Research Conference on Smart Computing and Systems Engineering (SCSE)*, IEEE. Accessed: 2025-08-03.
URL: *https://ieeexplore.ieee.org/document/11031019*

Dileepkumar, S. R. and Mathew, J. (2023). Transforming software development: Achieving rapid delivery, quality, and efficiency with jenkins-based ci/cd pipelines, *2023 An-*

nual International Conference on Emerging Research Areas: International Conference on Intelligent Systems (AICERA/ICIS), IEEE.

HashiCorp (2025). Terraform documentation. Available at: `https://developer.hashicorp.com/terraform/docs`.

Inc., D. (2025). Docker documentation. Available at: `https://docs.docker.com/`.

Konala, P. R. R., Kumar, V., Bainbridge, D. and Haseeb, J. (2025). A framework for measuring the quality of infrastructure-as-code scripts, *arXiv preprint* . Available online: https://arxiv.org/abs/2502.03127.

Marella, V. (2024). Optimizing devops pipelines with automation: Ansible and terraform in aws environments, *International Journal of Scientific Research in Science, Engineering and Technology* **11**(6): 285–294.
URL: *https://doi.org/10.32628/IJSRSET2410614*

Mysari, S. and Bejgam, V. (2020). Continuous integration and continuous deployment pipeline automation using jenkins ansible, *2020 International Conference on Emerging Trends in Information Technology and Engineering (ic-ETITE)*, IEEE. Accessed: 2025-08-03.
URL: *https://ieeexplore.ieee.org/document/9077670*

Project, A. (2025a). Ansible documentation. Available at: `https://docs.ansible.com/`.

Project, J. (2025b). Jenkins user documentation. Available at: `https://www.jenkins.io/doc/`.

Services, A. W. (2025). Amazon ec2 documentation. Available at: `https://docs.aws.amazon.com/ec2/`.

Vangala, V. (2017). Advancing devops automation: A framework for efficient ci/cd pipeline orchestration, ResearchGate preprint. Accessed: 2025-08-03. `https://www.researchgate.net/publication/391908490_Advancing_DevOps_Automation_A_Framework_for_Efficient_CICD_Pipeline_Orchestration`.