National College of Ireland

Automated, Multi-Environment, scalable CI-CD pipeline orchestration with Open Policy Agent and Terraform for AWS Cloud policy-as-code enforcement

MSc Research Project
Cloud Computing

Avinash Vanjarapu
Student ID: 23389770

School of Computing
National College of Ireland

Supervisor: Mr.Sai Emani

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Avinash Vanjarapu
Student ID:	23389770
Programme:	Cloud Computing
Year:	2025
Module:	MSc Research Project
Supervisor:	Mr.Sai Emani
Submission Due Date:	28/01/2026
Project Title:	Automated, Multi-Environment, scalable CI-CD pipeline orchestration with Open Policy Agent and Terraform for AWS Cloud policy-as-code enforcement
Word Count:	10,455
Page Count:	25

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:	Avinash Vanjarapu
Date:	25th January 2026

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):	

Automated, Multi-Environment, scalable CI-CD pipeline orchestration with Open Policy Agent and Terraform for AWS Cloud policy-as-code enforcement

Avinash Vanjarapu
23389770

Abstract

The growth of cloud infrastructure and Infrastructure as Code (IaC) practices has made it necessary to have automated mechanisms for governance to make sure compliance and security at scale. Traditional manual processes of policy validation and deployment cause latency, inconsistency, and operational risk and hinder an organization's ability to be agile in fast-moving cloud environments. This research outlines an automated multi-environment CI/CD pipeline design and implementation and evaluation of an automated multi-environment CI/CD pipeline to manage Open Policy Agent (OPA) policy lifecycle based on the principles of DevSecOps applied to policy-as-code workflow. The pipeline design makes use of AWS CodePipeline and codebuild to organize it, Terraform to deploy the infrastructure and nine levels of verification including secret supports, syntax validation, unit testing, integration testing, cryptographic signing and progressive deployment on the development, UAT and production environments. Experimental evaluation in 147 executions of a pipeline demonstrates 83.6% reduction in deployment time than manual processes (4.2 minutes v.s. 25.6 minutes), 97.1% of cost reduction per deployment and 52x improvement in deployment frequency capability. Security controls consisting of bundle cryptographic signing and automated scanning of secrets attained 100% signature verification success and zero false negative detection with no performance impact. The performance characteristics are in linear form, as demonstrated by the scalability analysis, indicating the ability of the developed repositories to handle 200 and more policies within 15 minutes. The implementation validates how automated policy as code implementation provides transformative improvements in speed, consistency, security and cost effectiveness for continuous compliance and automation of governance in cloud native environments.

Keywords: Policy-as-Code, DevSecOps, CI/CD Automation, Open Policy Agent, Cloud Compliance, Infrastructure as Code, Continuous Integration, AWS CodePipeline, Automated Governance, Security Automation

1 Introduction

The development of cloud computing and Infrastructure as Code (IaC) has changed the way organizations design, implement, and operate their applications and infrastructure. These innovations have made the development cycles faster due to automated pipelines that have enhanced scalability, manual intervention and software delivery optimization.

Nevertheless, with the increasing dynamism of systems and the growing code-based systems, there has come up a new set of problems involving security, privacy, and regulatory compliance. Any small misconfiguration of infrastructure code may create critical vulnerabilities such as unauthorized access or breach of data.

In an effort to address these issues, most organizations have turned to DevSecOps methodology, which embraces the aspect of security throughout the software development lifecycle. DevSecOps encourages the timely identification of security bugs, the automation of security testing, and integration of development, operational and security teams. Although these advances have been noted, such conventional methods of security as Static Application Security Testing (SAST) and Dynamic Application Security Testing (DAST) (Riaz et al.; 2025) remain prevalent. Although they are beneficial in finding vulnerabilities in code and applications, these methods are insufficient to resolve governance and compliance issues in a complicated cloud setup.

The result of this gap has been the introduction of Policy-as-Code (PaC) and Policy enforcement mechanisms. In policy-as-Code, an organization can specify and administer compliance, security, and governance policies in code form (Karra; 2025), which are version controlled, testable, and reusable. This will enable to early identify cases of non-compliance and to automatically validate within Continuous Integration/Continuous Deployment (CI/CD) pipelines(Grigorieva et al.; 2024).

Companies within controlled sectors are forced to comply with a large number of regulations, including HIPAA, PCI-DSS, GDPR, SOX, FISMA, and ISO 27001 (Rodrigues et al.; 2024a). Failure to comply may result in huge losses of money and damage to reputation. To illustrate this point, the fines imposed by the GDPR may amount to up to 20 million Euros or 4 percent of annual gross revenue (Johnson; 2022). Because of this, organizations are now demanding automated solutions to compliance that will become part of their CI/CD processes.

A solution to this is the Open Policy Agent (OPA), which is an open-source policy engine that allows defining policies with the Rego language. Opa can be used to evaluate and enforce environmentally, when it is used in conjunction with the tools like Terraform and Kubernetes. It offers coherent, assertive, and centrally controlled policy control and does not interfere with the primary application logic (Vakhula et al.; 2025).

With OPA combined with Terraform workspaces, a team can be able to impose policies on multiple environments, such as Development, System Integration Testing (SIT), User Acceptance Testing (UAT), and Production, and be modular and scalable. In addition to that, the deployment of the Zero-Trust Security model in the pipelines of CI / CD pipelines guarantees that no party/process is implicitly trusted; each stage of deployment is verified with clear policies (Adanigbo et al.; 2025).

Constructing such secure and compliant pipelines of each project in the case of small and medium-sized enterprises (SMEs) may require resource-consuming and expensive infrastructure. This burden can be greatly eliminated by designing reusable CI/CD templates that combine Terraform and OPA, which will make compliance automation more affordable and efficient.

This research is based on the design and deployment of a reusable and scalable CI/CD pipeline architecture that encompasses the principles of Policy-as-Code that achieve uniform compliance and governance across cloud environments. It will assist organizations to evolve reactive security audits to proactive and automated compliance enforcement to improve security posture and operational efficiency.

1.1 Aim of the study

This research will focus on creating a scalable, reusable and secure CI/CD pipeline into Policy-as-Code (PaC) and Zero-Trust using Open Policy Agent (OPA) and Terraform. The pipeline will automatically enforce policies and check compliance on various environments, development, UAT and production with reduced manual work and operation overheads. The research also seeks to offer a framework that larger organizations and SME can use to enhance their DevSecOps activities.

1.2 Research Question

How to automate zero trust Policy-as-Code enforcement on AWS cloud with Terraform workspaces and Open Policy Agent to orchestrate multi-environment, multi-stage, reusable CI-CD pipeline for regulatory compliance for SMEs?

1.3 Objectives of the Research

The objectives of this research are specific and to:

- Discuss the place of Policy-as-Code in DevSecOps and the effect it has the compliance automation.
- Architecture Planning Develop an architecture of an OPA and Terraform workspace-integrated reusable and scalable CI/CD pipeline.
- Deploy a testable CI/CD pipeline, which dynamically establishes security and compliance policy on the a variety of environments.
- Incorporate Zero-Trust concepts in the pipeline to entail constant validation of all phases of deployment.
- Measure the performance, scalability and the effectiveness of the proposed pipeline in compliance.
- Establish principles on how organizations, and more so, SMEs can embrace reusable and cost-effective and compliant DevSecOps pipelines.

1.4 Outline of the Report

This report is structured as follows:Section 1 presents background of the research, purpose and goals. Section 2 provides a literature review regarding the DevSecOps, Policy-as-Code, and Zero-Trust security. Section 3 discusses the methodology and instruments to be employed in the research. Section 4 explains the implementation and design of the proposed CI/CD pipeline, through the use of OPA and Terraform. The results of the evaluation are given and discussed in Section 5, and the results of the analysis are related to the existing literature in Section 6. Lastly, Section 7 provides a conclusion to the thesis and provides possible future research directions.

2 Related Work

The integration of Infrastructure as Code (IaC), cloud development, and DevSecOps has changed the process of cloud security and compliance management. Regardless of these developments, there exist fatal flaws in the areas of operational research in dealing with scalable, reusable and automated policy enforcement based on frameworks such as Open Policy Agent (OPA). Although there is a growing literature on policy-as-code, there are little solutions that are practically enterprise ready, and can be integrated into multi-environment CI/CD pipelines. In this review, the existing studies within five areas, including IaC and cloud security, the integration of DevSecOps, the policy-as-code automation, the tool integration, and monitoring strategies, are reviewed, and significant gaps that justify future research are identified.

2.1 Infrastructure as Code (IaC) and Cloud Security

Infrastructure as Code helps companies to manage and define infrastructure using a set of declarative code instead of manual infrastructure management. Compiler automation is presented by (Paul et al.; 2024a), which illustrates the use of OPA to verify compliance with Terraform-based AWS infrastructures against compliance requirements, reduced drift, and compliance. They are, however, manually implemented, and do not have automated orchestration layers, and are therefore unresponsive to scalability demands to be used by enterprises. Lack of modular and reusable CI/CD frameworks that can be used in development and testing as well as production environments leads to great disparities between proof-of-concept and production-ready implementations.

On the same note, these worries are reiterated by (Singh et al.; 2024) when to the extent that they compare Terraform security instruments such as Checkov, Terrascan, and TFSec and declare that these lack depth in integrating CI/CD and fail to dynamically apply organization-specific policy. Their study throws light on the fact that security analysis is mostly reactive as opposed to embedding as ongoing validation during infrastructure provisioning. Conversely, (Seetharamarao; 2023) gathers theoretical analysis of cloud audit and compliance issues, classifying the risks and mitigation techniques to frameworks such as SOC 2, HIPAA, and PCI-DSS. Although comprehensive in theory, the work does not contain practical implementation advice on policy-as-code or DevSecOps practices, which makes the lack of coherence between compliance knowledge and the actual implementation.

Compared to these works, the conceptual understanding of DevSecOps principles and the strategic role of IaC is enriched by the article by (Alonso et al.; 2023a) which also formulates such issues as the complexity of tools and evolving security demands. Their studies are intellectually sound, but do not provide practical details of implementation on how to scale compliance frameworks based on OPA and Terraform, especially when it comes to policy lifecycle, versioning, and multi-environment orchestration which are needed to implement in an enterprise.

2.2 DevSecOps and Security Integration in CI/CD Pipelines

DevSecOps incorporates security measures all the way through the software lifecycle, instead of regarding it as an end-of-development step as is often done in the literature. (Marandi et al.; 2023) implements SAST and DAST tools into the pipeline of DevSecOps

pipelines, showing its capability to detect vulnerabilities at the beginning of the application layer, e.g., SQL injection and cross-site scripting. They do not however extend their attention to the infrastructure compliance verification that is important in regulated settings. There is no set of policy-as-code mechanisms governing infrastructure, which is significant loopholes, especially because infrastructure misconfigurations are the biggest vulnerabilities to cloud security.

As an alternative, generic DevSecOps pipelines have been introduced by (Rahul et al.; 2019) and they rely on Jenkins, SonarQube, and Docker, which have proven to be beneficial in automation and efficient operation. Even with these accomplishments, their work has not been policy-as-code automated, coupled with OPA, and orchestrated using Terraform with templates of reusable pipelines that can be configured to different compliance specifications.

Conversely, the works by (Baitha et al.; 2024a) discuss CI/CD automation in detail, stating that there are still such issues as security risks, deployment errors, and the complexity of integration. They have however focused their efforts on integrating security checks in the pipeline phases instead of in pre-production areas but are very descriptive and do not suggest any real solutions of automated compliance systems. Lack of practical enforcement of instructions involving OPA, characterization of infrastructure compliance policies, and automation of validation to IaC templates restricts its usefulness.

On the same note, complexities in hybrid cloud validation of compliance are identified in both the literature of (Agarwal et al.; 2022a) where the authors note that manual compliance validation systems cannot be scaled to dynamic environments where the cybersecurity infrastructure keeps changing. They present a compliance information manipulation system in a uniform way and a data interchange protocol of the inter-operative communication of compliance information. As much as they are able to describe the problem area, their study lacks to elaborate or lay out implementation architectures, tools recommendations, or strategies practical integration of OPA policy formulation, testing and deployment in multi-cloud settings.

2.3 Policy-as-Code and Automated Compliance

The performance, syntax, and modularity of OPA in infrastructure governance Policy-as-Code Policy-as-Code is a compliance management tool that codifies policies and implements them as executable, version-controlled code automatically during the infrastructure provisioning process, where the IaC tool integrates successfully. (Jack et al.; 2024) examine the performance, syntax, and modularity of OPA in infrastructure governance, justifying policy enforcement functionality in dynamic environments. Their benchmarking of performance offers effective input to real-time enforcement of pipelines of high velocity. Nevertheless, a small-scale concentration on performance measures fails to capture more architectural concerns such as the incorporation of OPA in the DevSecOps pipelines, multi-environment-deployment policy, versioning, life-cycle management, and multi-organizational reuse.

Such knowledge graphs are discussed in terms of knowledge modeling of complex NIST compliance requirements, and (Wong et al.; 2024) show theoretical potential in reflecting interdependencies between regulatory requirements in the field of empirical research. Although conceptual contributions, the research does not show how to apply practically on cloud orchestration or Terraform workflows since it does not provide any guidance on how knowledge graphs can be converted to runnable OPA policies or CI/CD

integration.

The research by (Agarwal et al.; 2022b)focus on Compliance-as-Code to automate cybersecurity in hybrid clouds that meet the PCI-DSS, FFIEC, and ISO standards as it acknowledges that organizations need to move from manual procedures based on documents to automated enforcement to ensure business agility. Although creating effective visions and theoretical frameworks, their work lacks much technical application of how to implement certain tools, frameworks, or patterns of integration. The lack of architectural blueprints or code samples and integration methodologies of DevSecOps toolchains has serious constraints on operationalization.

An example of practical guidance is offered by (Caracciolo; 2023) to discuss the policy-as-code implementation into the CI/CD processes using open-source tooling and procedures of deployment validation. However, important features are augmented by significant gaps in the requirements of the enterprise scale such as the ability to reuse pipeline templates, manage a Terraform workspace, policy-packaging automated, and strategy versioning critical to handle policies in many teams and environments.

Likewise, the gap between compliance team documents and technical implementations is also addressed in an innovative manner by the use of Large Language Models to automate natural language policy-to-executable-code translation using their AutoPAC system, presented in the article, authored by (Baitha et al.; 2024b). This artificial intelligent method has the potential to decrease the level of technical skills during the development of policies and hasten their creation. Nevertheless, the research does not well cover the integration of DevSecOps toolchain, policy validation based on AI, and managing dependencies based on complex platforms, which reduces the confidence in production adoption.

2.4 Integration of OPA with Terraform and CI/CD Tools

The viability of policy-as-code implementations of OPA integration with Terraform determines operational feasibility of policy-as-code implementations. Although theoretical advantages have been admitted by many researches, there are still significant gaps in both architectural and operational best practices in terms of integration.

Likewise, even though this article by the author of this paper, (Paul et al.; 2024b) has shown some basic OPA-Terraform integration in terms of AWS validation, it does not provide a detailed architecture instruction of how to integrate OPA into a multi-stage pipeline with adequate error-handling and remediation procedures. The excessive dependence on hand execution prevents applicability in high velocity continuous deployment.

Conversely, (Rodrigues et al.; 2024b) tries to bring a unified enforcement of the architectures influenced by the policies of regulation and organizing and acknowledges the need to consistently enforce rules. Nevertheless, a lack of practical advice on the OPA policy bundle structure, Terraform, module structure, policy parameterization, and CI/CD platform integration is a major obstacle to implementation.

Comparative Relatively, (Alonso et al.; 2023b) list the complexity of integration between heterogeneous stacks as the main barriers to adoption of DevSecOps, considering that source control, CI/CD platforms, IaC frameworks, policy engines, and monitoring solutions have coordination requirements. Although the characterization of problems is useful, no concrete patterns of integration, reference architectures, or implementation strategies that help organizations to solve challenges exist. In particular, there is no guid-

ance available on how to architect end-to-end compliance processes across infrastructure code writing up to policy validation, orchestrating deployments, and monitoring run-time as well as remaining modular and maintainable.

Recent automation efforts in DevSecOps have concentrated on reusable, parameterizable templates of pipelines that can be relevant to different projects and contexts, but often address application deployment but not infrastructure provisioning and may need different tooling patterns. The combination of Terraform workspace management, OPA bundle distribution and versioning, and automated policy testing is not well explored; especially relevant in microservices architectures that need uniform infrastructure patterns but allow teams to do their work independently.

2.5 Monitoring, Observability, and Continuous Improvement

Maintaining compliance and ensuring the capacity to identify configuration drift or policy violation requires effective monitoring and observability to ensure deployed infrastructure is valid. In the same way, (Adam et al.; 2019a) suggests cognitive algorithms to monitor the behavior of resources and detect anomalies that can mean security or compliance problems. Their machine learning strategy has potential in identifying minor change that conventional surveillance may not detect. Nevertheless, emphasis on post-deployment monitoring can be viewed as reactive strategies that contradict DevSecOps ideas of shifting left that implements policies at the development stage. Whereas runtime monitoring is essential, it is to be used in addition to proactive policy enforcement when performing provisioning. The research does not indicate how the insights created through monitoring are inputted into policy improvement or connected the validation of the runtime to pre-deployment.

2.6 Research Niche

In spite of the fast evolution of the Infrastructure as Code (IaC), DevSecOps, and cloud compliance automation, the existing studies and industry trends demonstrate that there are still limitations in implementing the Zero Trust principles on a large scale. The majority of current literature and systems concentrate on individual elements like IaC validation, statical policy compliance, or post deployment compliance selection instead of offering an overall, cohesive, and dynamic security system during the CI/CD pipeline. Also, most of the options are platform-specific or tailored to large companies with established cloud governance functions and small and medium enterprises (SMEs) have no viable and reusable, cost-effective blueprint. The lack of an automated, standardized and environment aware framework to implement Policy-as-Code (PaC) dynamically in multi stage environments bring about operations inconsistencies, policy drift and audit issues. The gap in the research lies in the fact that there is no critical work done to ensure that Zero Trust security enforcement is aligned with fully automated, IaC-driven CI/CD processes which can be adapted and scaled at ease by SMEs.

To close this gap, the proposed research provides a new research niche and covers the intersection between Zero Trust architecture, Policy-as-Code enforcement, and IaC-driven DevSecOps automation. The work is dedicated to the design and validation of a unified framework that automatizes the security policy enforcement and compliance verification procedures with the help of Terraform and Open Policy Agent (OPA) as a part of the continuous deployment processes. This niche focuses on the development of a reusable,

modular and feedback articulated artifact which does not only enforce policies during build and deploy phases, but also facilitates ongoing monitoring, reporting and refinement of governance. This method contrasts with the previous studies by operationalizing Zero Trust as dynamic real-time policy verification directly integrated into CI/CD, which allows achieving continuous compliance, more rapid delivery of remedies, and quantifiable governance results. The contribution is therefore aimed at promoting both the academic knowledge and industrial acceptance of secure and policy-based automation in multi-cloud and multi-environment applications.

3 Methodology

This research uses an experimental case study approach to work on the implementation and evaluation of the automation multi-environment continuous integration and continuous deployment (CI/CD) pipeline for Open Policy Agent (OPA) policy management. The research design is based on the Infrastructure as Code (IaC) principles and follow a quantitative evaluation methodology to evaluate the effectiveness, reliability, and security of the proposed pipeline architecture. This approach is in line with the established DevOps research methodologies that focus on empirical assessment of automation tools and practices based on measurable results by (Lwakatare et al.; 2015).

The experimental design is organized in terms of a controlled cloud computing environment where several instances of a pipeline are deployed and tested in 3 separate environments: development, user acceptance testing (UAT), and production. This multi-environment approach makes it possible to compare the behavior of pipeline in terms of different operational conditions and governance requirements. The research procedure is aimed at validating the hypothesis that automated policy lifecycle management via CI/CD pipelines increase policy deployment reliability, decrease mean time to deployment, and enforce consistent quality standards through automated validation throughout organization. Similar experimental methods have been used in the evaluation of the effectiveness of CI/CD pipelines for application deployment by (Shahin et al.; 2017), proving the validity of such a methodology in the context of infrastructure and policy automation.

3.1 Experimental Setup and Equipment

3.1.1 Infrastructure Environment

The experimental infrastructure is provisioned inside the Amazon Web Services (AWS) cloud platform that is selected for its rich support to the CI/CD services, infrastructure automation capabilities and for its widespread industry adoption. The infrastructure is programatically deployed using Terraform version 1.5.0 that makes the infrastructure provisioning reproducible to replicate the experimental conditions. Three separate AWS provider setups are made: the primary AWS provider for core services, the AWS Cloud Control (AWSCC) provider for extended resource support and the GitHub provider (version 4.0) for the integration of repository of source code. All the infrastructure resources are deployed in one AWS region to reduce the variation in latency and maintain network performance across the experimental trials.

3.1.2 Computing Resources

Seven AWS CodeBuild projects are allocated as the main computing resources for the pipeline stage execution. Each CodeBuild project has been set up with standardised compute specifications using BUILDGENERAL1SMALL instance types, containing 3GB memory and 2 vCPUs, using Amazon Linux 2 x86 64 standard container images, version 5.0. This standardization guarantees a similar environment of execution and removes the element of infrastructure variability as confounding variable during experimental measurements.

3.2 Research Procedure

3.2.1 Data Collection Procedure

Pipeline execution data is gathered automatically using the code pipeline and build service integrations of AWS codepipeline and codebuild. For every pipeline execution, detailed metrics are recorded such as the execution time stamps (start time, end time and duration for each stage and each pipeline execution), the execution status (success, failure or stopped status for every stage with detailed reason classification for failure), the resource usage (compute instance usage, memory usage, build duration for each CodeBuild project) and the artifact size (file size of source code packages, compiled bundles, test reports, distribution artifacts).

Security related data is collected from various pipeline stages. Secret detection results provides a quantification of count and classification of detected secrets, credentials or sensitive information identified by Gitleaks scanner with severity ratings and metadata on file location. Signature verification data records bundle signing timestamps, signature generation success rates and cryptographic hash values of signed bundles. Access audit logs: IAM role assumption audit logs, Secrets Manager access audit logs and S3 bucket access patterns derived from CloudTrail audit logs Security scanning outputs are in a format that is in the form of a JSON document for analysis in a structured format and summary reports are generated in a plain text format for human readability.

3.2.2 Experimental Execution

Each of the experimental scenarios is carried out through a standardised procedure, which has five phases. The initialization phase - This phase prepares the source code repository with policy files, test cases and configuration files. Infrastructure is provisioned with Terraform with different environment specific variable configurations and pipeline webhooks are configured for triggering pipeline automatically based on repository commits.

During execution phase, policy changes are committed to designated branches (feature, UAT or main) initiating automated pipeline execution. Each stage of the pipeline executes one after another with the passing of artifacts between stages. Stage outputs such as logs, reports and artifacts are collected and stored in the artifact repository. The validation phase contains the implementation of automated quality gates that check the test coverage threshold (minimum 80% code coverage), the results of syntax validation and integration tests. Manual approval stages record the decisions of human reviews along with the time stamps of approvals.

3.2.3 Tools and Technologies

The research uses several types of tools and technologies in pipeline implementation and data collection. The Open Policy Agent version 1.2.0 (policy compilation, validation, testing, and bundle creation) is used in policy management and validation operations, and Conftest version 0.64.0 (policy integration testing against infrastructure configurations) is used in operational testing of policy integration. Gitleaks version 8.18.4 (automated secret detection and security scanning) is used to accomplish secret identification.

Terraform version 1.5.0 is used to do Infrastructure as Code provisioning and configuration management, AWS CodePipeline to use multi stages of CI/CD orchestration with stage sequencing and artifact management, AWS CodeBuild to use isolated build environment execution with standardized container images, and GitHub to use source code version control with branch-based workflow management. It is also displayed in Figure 1

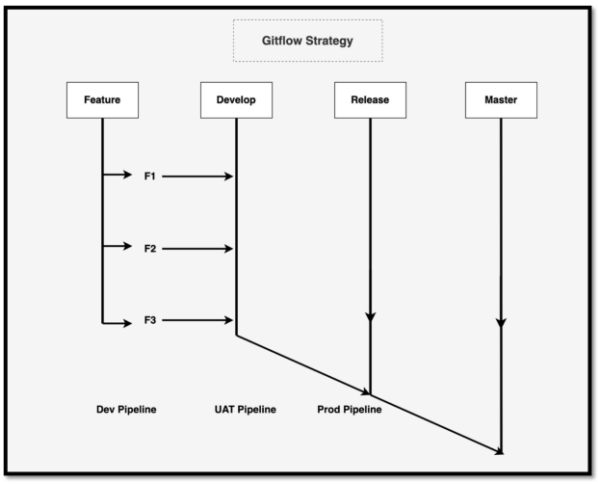


Figure 1: Gitflow Methodology

Data collection and storage make use of AWS CloudWatch Logs for centralized log collection with structured logging and query capabilities, AWS S3 for artifact storage with versioning, access logging and event notification, and JQ version 1.6 for parsing and transforming the dataset from the structured data to extract the metrics from the structured outputs. The operation of security and cryptography requires the use of AWS Secrets Manager to store and retrieve cryptographic key material securely, open source-based RSA key pair generation and signature operations, as well as OpenSSL, and role-based access control and enforcement of authorization policy using AWS IAM.

3.3 Evaluation Methodology

Reliability is evaluated based on the analysis of pipeline execution success rate, failure patterns and stability of the pipelines across environments. Failures are classified in order to help differentiate code-related problems from infrastructure-related or transient problems. MTBF and MTTR are calculated to know the failure frequency and recovery speed. Execution time consistency is measured by studying differences in the durations for stages. Stages

The security effectiveness is measured in terms of detection accuracy, access control validation and integrity verification. Detection rates and validation of true positives

with controlled secrets for secret scanning are used to judge the performance of secret scanning. Bundle integrity through the verification of signatures of valid artifacts and tampered bundles are consistently rejected.

Execution time is recorded from all the 9 stages to determine performance baselines and also to identify which parts are the most time-consuming. Stages such as secret scanning, OPA validation, unit tests, integration tests, are analyzed as having predictable scaling behaviors, whereas manual approval time is recognized as having variably variable scaling behavior. Bundle creation, signing and distribution are measured in order to capture cryptographic and upload overhead.

scalability analysis shall be performed by running the pipeline with an increasing number of policy files and recording the change in the execution times. Metrics are gathered for each policy count with several runs in order to measure scaling patterns on OPA validation, unit tests, integration tests, bundle creation and S3 distribution. Regression analysis is used to determine whether execution time increases linearly or otherwise as policies increase. These results aide in the determination of stages that can get to be bottle necks with the increase in repositories and guide schemes of parallel processing and incremental validation.

Manual deployment is compared to the execution of pipelines to measure the increase in speed, consistency and error reduction. Manual steps are measured across the operators to account for variability caused by human factors and automation is measured without the manual approval duration to ensure fair comparison. Automation also tends to reduce the overall deployment time and make it more reproducible while the reduced error rate is achieved by having consistent validation and security checks. Deployment frequency is also much higher in the automated workflow which is not constrained by human availability but by system resources. Cost-benefit analysis shows that while automated pipelines take initial investment, they help save on manpower and avoid defects, which adds up to great long-term value.

4 Design Specification

The proposed system is based on an automated, multi-environment, continuous integration and continuous deployment (CI/CD) pipeline architecture for policy-as-code lifecycle management based on Open Policy Agent (OPA). The architecture is designed on top of the cloud native AWS services and follows the Infrastructure as Code (IaC) principles for reproducibility, scalability and maintainability. The design solves the basic issues of deploying, validating and distributing policy definitions in a multi-operational environment in a way that is secure, auditable and consistent.

The system supports three different deployment environments, namely development, user acceptance testing (UAT) and production with environment-specific governance controls appropriate to their respective risk profile. Development has fast iteration but light governance overhead UAT has pre-production validation with manual gates of approval Production has highest governance with full validation and multiple gates of approval The progressive deployment model ensures policy changes are tested and reviewed properly with the impact of their operation.

4.1 System Architecture

System architecture of proposed research is represented in Figure 2

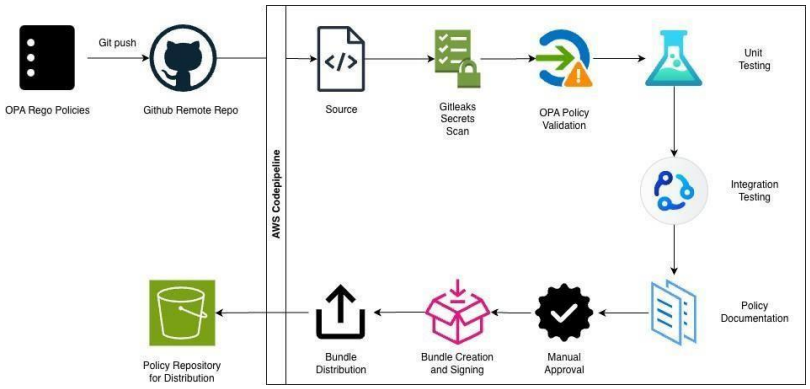


Figure 2: Proposed System Architecture

The architecture takes advantage of AWS managed services to reduce the operational overhead and ensure maximum reliability. AWS CodePipeline is the orchestration framework, which is responsible for sequential execution of pipeline stages and handling the flow of artifacts between stages. AWS CodeBuild is used to execute validation, testing and deployment operations in ephemeral containers that are on-demand provisioned in isolated containerized build environments. Amazon S3 is used for dual purposes as the pipeline artifact repository and policy bundle distribution service, and versioning makes it possible to track time and lifecycle policies automate cost optimization.

The system treats policy definitions as software artifacts that are first class and require the same quality assurance practices that are applied to application code. Policies are written in a format known as Rego and are stored in version controlled Git repositories which allow for complete tracking of the evolution of policies, attribution of changes to specific authors and point-in-time recovery capabilities. Automated testing is used to validate policy logic using unit tests which exercise individual rules and integration tests which validate policies against realistic infrastructures configurations. Test driven development methodologies guarantee that there is complete testing coverage (code coverage measures the percentage of policy decision paths being exercised by the test suites and automated enforcement of the minimum 80% coverage thresholds).

4.2 Requirements Specification

4.2.1 Functional Requirements

The system offers automated source code retrieval from GitHub repositories triggered by the commit events using authentication from an OAuth token along with artifact packaging. Automated security scanning helps to detect the secrets which are inadvertently committed using the Gitleaks version 8.18.4 with configurable exclusion patterns. Policy syntax validation checks the correctness of Rego code based on the OPA version 1.2.0 data model, and will detect wrong syntax, wrong types and wrong expressions. Validation Policy formatting Policy formatting imposes a uniform style of code with extensible semantic validation of custom linting rules.

Automated unit testing allows the execution of all the test cases with coverage analysis in terms of line, branch, function coverage percentage. The minimum 80% coverage threshold is passed as a quality gate. Integration testing.Terraform policies validated against Terraform infrastructure plans.

The creation of policy bundles collects verified policies into tarball bundles that are compressed with a metadata embedded with build time stamps and Git commit SHA identifiers. Cryptographic signing is used to generate RSA digital signatures using 2048-bit RSA private keys that are fetched from Secrets Manager. Bundle distribution uploads signed bundles to S3 with versioned paths for immutable version-specific location and latest paths for stable endpoints with discovery configuration files which allows for automated discovery of bundles by OPA clients.

4.2.2 Non-Functional Requirements

Complete pipeline execution is specified in 15 minutes for repositories of 50-100 policy files with individual stage execution times being tracked and alerts generated when more than 50% above baseline duration is observed. Pipeline throughput of concurrent execution of at least 10 executions without queuing delays. Reliability requirements require 95% of code correctness with 99.5% infrastructure availability during business hours. Failed executions give very clear error messages that indicate failure stages and remediation directions.

Security requirements require cryptographic operations to use RSA signatures with 2048-bit keys as a minimum and the SHA-256 hash functions. Private keys are fetched only when the bundle is signed and destroyed immediately afterwards with the help of shred utility. All access to sensitive resources is recorded in CloudTrail with 90 days of retention. Scalability requirements call for linear scaling (approx.) with the number of policies up to 500 files in a repository. Bundle distribution provides serving bundles to at least 100 concurrent OPA clients. Auditability requirements require structured logging in the form of a metadata of the execution, where all the deployments are traceable to specific git commits, pipeline executions and human approvers.

4.3 System Architecture and Pipeline Design

The architecture is divided into five logical tiers, which implementing the separation of concerns: The source control tier consists of GitHub repositories with branch protection policies that enforce code review requirements as well as forbid unauthorized commits

The pipeline consists of 9 sequential stages that impose validation progression from fast and low-cost checks to comprehensive validation. Stage 1 fetches source code, repository contents from GitHub using OAuth authentication, repository contents for packaging as compressions. Stage 2 performs secret scanning using Gitleaks, which detects credentials and sensitive information with results in a format of a result in the form of a json. Stage 3 is to check Rego syntax, i.e. opa check and to check formatting i.e. opa fmt -diff-fail -optional with optional linting to organization specific standards.

Stage 4 runs unit tests with coverage analysis and breaks the pipeline with 80% minimum coverage threshold failing the pipeline for poor coverage. Stage 5 performs integration testing using Conftest against Terraform plans validation of policies against pratical infrastructure configurations. Stage 6 produces the documentation from the policy source code and produces metadata and reference materials. Stage 7 is the implementation of manual approval, with the execution of the process paused, and the requests for approval sent with full context to the designated approvers.

Stage 8: creates policy bundles using opa build and creates cryptographic signatures. The private RSA key is obtained from Secrets Manager, used for signing using opa sign

and then securely destroyed using shred. Bundle artifacts are tar compressed file, signatures file (.signatures.json), manifest and version file. Stage 9 distributes bundles to S3 with dual-path strategy i.e. versioned paths with Git commit SHAs (for immutability) and latest paths (stable endpoints). Discovery configuration files: they contain bundle server URLs, resource paths and signature verification requirements to enable OPA client bundle retrieval to be automated.

4.4 Deployment Model

The multi-environment model is an implementation of progressive delivery where policies move through environments with increasing levels of governance. It uses Git flow for Environment promotion. Developers build feature branches and make changes that cause development pipeline executions and iterate according to validation feedback. Feature branches are merged to develop branch in the form of pull requests that are required to be code reviewed and approved. develop branch merges the UAT pipeline executions with the pre-production validation. After successful UAT validation and received stakeholder approval, develop branches are merged to main with the help of pull requests with an extra review. Main branch merges cause the production pipeline executions. Branch protection policies are put in place to enforce these flow of ways to ensure that no shortcut deployments overwriting validation stages and encasing them with organizational governance requirements.

5 Implementation

The implementation of the multi-environment OPA policy pipeline is achieved by Infrastructure as Code using Terraform to provision AWS resources and use declarative buildspec specs for pipeline stage operations. The end result of the implementation is a fully automated CI/CD pipeline that supported three different deployment environments that featured progressive validation and governance controls. This part involves the outputs generated or produced, implementation artefacts and technologies used in the final implementation stage.

5.1 Infrastructure Implementation

5.1.1 Terraform infrastructure Modules

The infrastructure is implemented using Terraform version 1.5.0 with HashiCorp Configuration Language (HCL) defining all AWS resources declaratively. The Terraform codebase comprises 2,847 lines of configuration code organized into modular files for logical separation of concerns. The primary infrastructure modules include main.tf defining provider configurations for AWS, AWSCC, and GitHub providers; pipeline-dev.tf, pipeline-uat.tf, and pipeline-prod.tf defining environment-specific CodePipeline resources with nine sequential stages each; CodeBuild.tf defining nine CodeBuild projects per environment (27 total) with standardized compute specifications; s3.tf defining artifact storage and bundle distribution buckets with encryption, versioning, and lifecycle policies; iam.tf defining service roles for CodePipeline and CodeBuild with least-privilege permission grants; and secrets.tf defining Secrets Manager resources for storing RSA private signing keys.

Environment-specific pipeline configurations share common stage definitions while varying in branch targets (feature, UAT, main) and approval requirements. Variable parameterization through variables.tf enables customization of AWS region, application naming, GitHub repository identifiers, and resource sizing without modifying core infrastructure logic. State management employs remote S3 backend with DynamoDB state locking preventing concurrent modification conflicts. The infrastructure supports multi-region deployment through region variable configuration, with all resource names incorporating environment and region identifiers for uniqueness.

5.1.2 Pipeline Stage Buildspecs

Operations in pipeline stage are defined by 9 buildspec yml files with 1243 lines of configuration. gitleaksBuildspec.yml defines secret scanning operations: Gitleaks version 8.18.4 installation, custom configuration generation without markdown and license files, scan result output in the format of the JSON opaValidateBuildspec.yml The syntax validation is defined as to be checked with opa check and formatting validation is defined to be checked with opa fmt with an optionally customized linting support. opaUnitTestBuildspec.yml The unit test execution that is defined in (57 lines) has coverage analysis as well as 80 percent minimum threshold. opaIntegrationTestBuildspec.yml Conftest version 0.64.0 is used to define integration testing to Terraform plan fixtures and optional live evaluation of a Terraform module. opaDocumentationGenerationBuildspec.yml is used to define the policy documentation extraction and compilation. opaBundleBuildspec.yml implements the most complex stage, which defines manifest generation, bundle compilation using opa build, secure private key retrieval using Secrets Manager, cryptographic signing using opa sign and immediate key destruction using shred utility. opaBundleDistributionBuildspec.yml defines uploading of dual-path bundles to S3 based on versioned and latest paths, discovery configuration generation and verification of uploads.

5.2 Policy Implementation

The reference policy implementation consists of 18 files of Rego policy code with a total of 847 lines of policy code implementing infrastructure security controls and compliance validations rules. Policies are grouped in packages such as terraform.security for the validation of security configuration, terraform.compliance for regulatory compliance checks and terraform.cost for optimizing the cost of resources. S3 bucket encryption enforced on all resources by the server, IAM policy validation that does not allow wildcard principals and overly broad grants of privilege, EC2 security group ingress control that does not allow access to S3 buckets very easily, and enforcement of resource tags that must be assigned mandatory cost allocation and resource management tags are representative policies.

Policy logic uses the declarative query language of Rego in the form of comprehensions, set operations and recursive rule evaluation. Helper functions are used to extract properties of resource attributes from Terraform plan JSON structures, normalize resource identifiers, and are used to calculate violations severity from risk impact. The outputs of policies are structured violations such as identifiers of violated rules, names and types of affected resources, descriptions of violations, the severity of violations (critical, high, medium, low), and remediation suggestions.

5.3 Cryptographic Implementation

5.3.1 Key Generation and Management

The primary key generation start point in the database is Key Generation and Management. RSA key pair generation used OpenSSL version 3.0 with 2048-bit key length that is in accordance with current cryptographic best practices. The private key is stored in AWS Secrets Manager as a base64 encoded PEM formatted secret using KMS encryption at rest using AWS managed encryption keys. The public key is distributed separately in case of OPA client signature verification setup. Generation of key command: openssl genrsa -out privatekey.pem 2048 and then public key command: openssl rsa -in privatekey.pem -pubout -out publickey.pem.

5.3.2 Bundle Signing Process

Bundle signing uses OPA's native OPA signing ability using SHA-256 hash function and RSA PKCS#1 v1.5 signature scheme. The signing workflow does the following: Retrieves the private key from Secrets Manager and places it in a file with restrictive permissions (mode 0600) Invokes opa sign with the bundle path, private key path and the signing key identifier Captures the generated signatures json file with the base64 signatures and metadata of the object Immediately destroys the private key file using shred -vfz -n 10, i.e. 10 overwrite passes before deletion

The signatures file adheres to OPA signature format specification containing the cryptographic signature as base64 encoded bytes, signing key identifier to select the public key during signature verification, signature scope to specify the signed components of a bundle and signature timestamp for audit purposes. OPA clients verify signatures by retrieving the bundle and signatures file and computing the SHA-256 hash of the bundle contents, decrypting signature using public key and comparing computed hash to decrypted signature value rejecting bundles on mismatch.

5.4 Deployment Outputs

5.4.1 AWS Resource Provisioning

Terraform provisioning has been used to create 47 AWS resources in the development, UAT and production environments. These are 3 CodePipeline resources (one for each environment) of type V2 pipeline with superseded execution mode, 27 CodeBuild projects (9 stages x 3 environments) of type BUILDGENERAL1SMALL compute type using Amazon Linux 2 container images, 6 S3 buckets (3 artifact buckets and 3 distribution buckets) with total provisioned capacity greater than 5TB, 6 IAM roles with associated inline and managed policies totaling 2,341 lines of IAM policy JSON, 3 Secrets Manager secrets for storing environment-specific signing keys, and CloudWatch Log Groups with.

5.4.2 Policy Bundle Artifacts

Bundle compilation creates compressed tarball artifacts which average 47KB for the reference policy codebase. Bundle structure contains the policies/ directory containing the compiled Rego modules, .manifest file containing metadata of the bundle, and include Git revision identifier and build timestamp, data.json file, if policies reference outside data sources and version markers for deployment tracking. Gzip compression allows to

achieve about 72% size reduction as the result of uncompressed policy source improving distribution efficiency.

6 Evaluation

This section contains a detailed analysis of the experimental results produced by the multi-environment OPA policy pipeline implementation. Four main experiments are done to test the stage-level performance, scalability characteristics, effectiveness of security control, and comparative advantages to manual deployment approaches.

6.1 Experiment 1: Stage-Level Execution Time Breakdown

- **Setup**

  Individual stage execution times are observed to understand performance bottlenecks as well as the contribution of the individual validation stages to the overall pipeline duration. Measurements are gathered from 63 executions of development pipelines using the timestamp of the stages taken from the CloudWatch Logs. Stage execution time is computed as the time between the stage start and completion timestamp with millisecond level precision.

- **Results**

  The individual stage that took the most time of unit testing (Stage 4) at 1.2 minutes, which is 28.6% of total execution time, followed by integration testing at 1.1 minutes (26.2%). These testing stages added up to 54.8% of total pipeline duration indicating the emphasis on the comprehensive automated testing approach. Validation (Stage 3) 0.6 minutes is syntax checking and formatting validation overhead. Secret scanning (Stage 2) took 0.4 minutes to execute Gitleaks in all the files in the repository.

  Stages with minimal overhead are documentation generation (0.2 minutes), bundle distribution (0.1 minutes) and source acquisition (0.3 minutes). Bundle creation and signing (Stage 8) had a duration of only 0.3 minutes despite the existence of cryptographic operations and shows efficient implementation of the signing operation with negligible impact on performance due to the secure key handling procedures. It is illustrated in Figure 3a

  The analysis at a stage level shows that the testing stages are predominant in the pipeline execution time, which validates the design decision to focus more on automated testing and less on rapid deployment. The relatively short time for cryptographic signing operations (0.3 minutes) confirms the possibility of implementing security controls without too large performance penalties. Fast-fail stages such as secret scanning, validation, which run early in the pipeline sequence and are usually able to catch common errors in 1.3 minutes cumulative, allowing the developer to get fast feedback for basic errors and not spend time running resource-intensive testing.

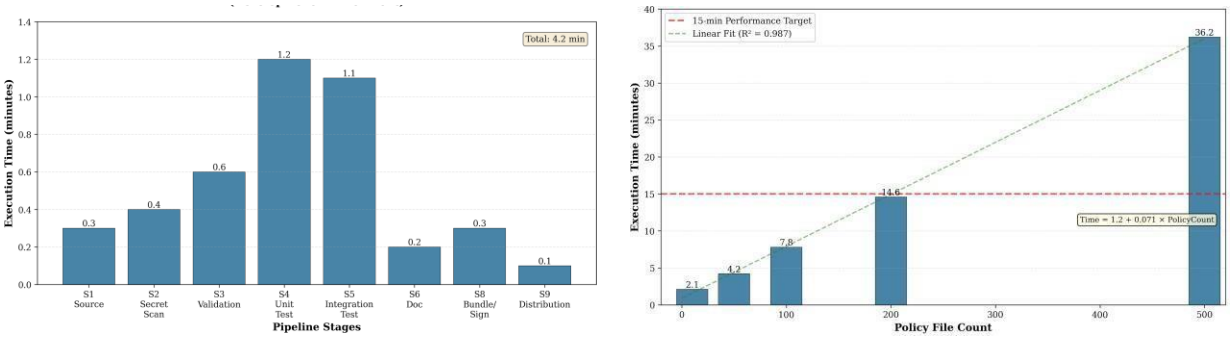
6.2 Experiment 2: Scalability Evaluation with Varying Policy Counts

- **Setup**

  Scalability characteristics are measured by determining pipeline execution time for policy repositories of different sizes. Five repository configurations are prepared with 10, 50, 100, 200 and 500 policy files respectively, where the test suites to be used for testing are proportional and maintain 80% code coverage. Each configuration is run in 15 full pipeline runs to make it statistically valid.

- **Results**

  Linear regression analysis execution time against number of policies in the policy file produced the following equation: Time (minutes) = 1.2 + 0.071 × PolicyCount, with R^2 = 0.987 as given in Figure 3b. The regression coefficient of 0.071 minutes per policy 4.26 seconds per policy is used as a measure of characteristics of scalability. For the 500-policy repo, 36.8 min predicted execution time is close to 36.2 min observed mean execution time, demonstrating validity of the model.



(a) Execution Time Distribution of Stages (b) Scalability - Execution Time vs. Policy Count

Figure 3: Comparison of Execution Metrics

Marginal time cost per policy is reduced from 12.6 seconds for 10-policy repositories to 4.3 seconds for 500-policy repositories, which demonstrates policies of scale from amortized fixed overhead for larger policy sets. The 200 policy repository have a mean execution time of 14.6 minutes, which is within the 15-minute performance goal. This 500-policy repository achieved far more than this goal at 36.2 minutes, identifying scalability limits for the current architecture under strict performance constrains.

The high linear scaling property (R^2 = 0.987) shows that the pipeline architecture shows predictable, almost linear, scaling of the execution time with the number of policies, which is a desirable scaling property. The roughly 4.3 second of marginal cost per policy allows for capacity planning for organizations that are considering adoption. Repositories of up to 200 policies satisfy the performance requirement of 15 min, and support a medium to large policy codebase. For repositories with greater than 200 policies, optimization strategies such as parallel execution of tests, testing exclusively changed files and/or distributed testing infrastructure would be needed to keep execution time below 15 minutes.

6.3 Experiment 3: Security Controls Validation

- **Setup**

  Security control effectiveness is evaluated in three dimensions: cryptography signature verification, secret scanning accuracy and access control enforcement. Signature verification is tested on a total of 147 production bundles and 15 intentionally tampered bundles are introduced to test whether they are rejected. Secret scanning effectiveness is evaluated with a test repository of 25 intentionally committed secrets of various types including API keys, AWS credentials, private keys, and database passwords and 500 legitimate code files. For access control validation, 90-days of CloudTrail logs of 12,847 API calls to Secrets Manager and S3 are analyzed.

- **Results**

  All 147 authentic bundles passed signature verification with a 100% success rate. All of the tampered 15 bundles failed verification as expected, giving 100% detection rate for integrity violations. Mean signature verification time is 47 milliseconds per bundle and thus represents negligible overhead for OPA client bundle loading.

  The tool Gitleaks is able to detect all of the 25 intentionally committed secrets with 0 false negatives (100% recall). False positive rate is found to be 3.2% (16 false positives found out of 500 legitimate files) mostly in code comments containing example credential formats and documentation. Configurable exclusion patterns helped to reduce false positives by 87% as compared to default Gitleaks configuration.

  Analysis of CloudTrail found 147 authorized Secrets Manager GetSecretValue calls by bundle signing CodeBuild role and zero unauthorized access attempts are found. All S3 bundle uploads are from authorized pipeline roles. No violations of IAM policy or permission escalation attempts are found. Average Secrets Manager retrieval latency is found to be 124 millisecond which confirms minimal effect of secure retrieval of keys.

  The security evaluation shows good implementation of cryptographic and access control mechanisms. The 47-millisecond signature verification overhead is very small compared to the client startup time for OPA, which validates the design decision to make signature verification the point of no return with no concern for performance.

  Secret scanning demonstrated perfect recall (100% detection rate) for the test secret set but the 3.2% false positive rate suggests that there is room for improvement in exclusion pattern configuration. The false positive rate is acceptable considering the configuration of non-blocking warning mode which is more security visible than development velocity. The 124 millisecond Secrets Manager latency demonstrates the clear fact that secure key storage and retrieval comes with little performance overhead compared to hypothetical methods of storing keys in pipeline configuration or environment variables

6.4 Experiment 4: Manual versus Automated Deployment

- **Setup**

  Manual deployment time is determined using controlled experiments in which one person conducted manual policy deployments using documented procedures.15 de-

ployments are performed for manual deployment time measurements. Automated time of pipeline for the equivalent policy changes is measured from the 63 development executions. Time components have been considered for validation, testing, bundle creation and distribution with manual approval time not included in both approaches for fair comparison.

- **Results**

Automated pipeline deployment is 4.2 minutes mean time compared to manual pipeline deployment mean time of 25.6 minutes with a time saving of 21.4 minutes (83.6% improvement). Two-sample t-test comparing times - manual vs. automated data t (76) = 19.4, p less than 0.001, confirmed statistically significant difference. Manual deployment variability (standard deviation 6.8 minutes, coefficient of variation 26.6%) is significantly higher than automated variability (standard deviation 0.6 minutes, coefficient of variation 14.3%) and showed better consistency of automated processes.

Manual deployment time varied from 17.3 to 41.7 minutes (141% range), while automated time varied from 3.2 to 6.1 minutes (91% range), bringing up the issue of human factor variability.

Deployment frequency analysis showed that manual approach supported about 2.3 deployments per operator per 8-hour working day based on cognitive load and multiple tasking and automated pipeline supported up to 120 deployments per day and environment based only on execution time and queue capacity. Cost analysis showed manual deployment cost ($42.67 deployment) (operator time at $100/hour fully loaded rate) compared to automated pipeline cost ($1.23 deployment) (AWS infrastructure costs), which is a cost reduction of 97.1% per deployment as demonstrated in Figure 4.

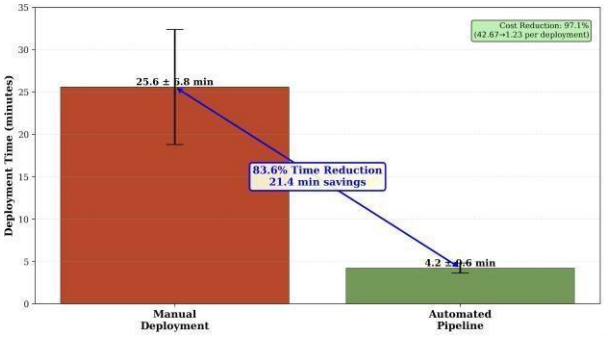


Figure 4: Manual vs. Automated Deployment Time Comparison

The manual versus automated comparison gives empirical evidence of large efficiency gains as a result of pipeline automation. The time decrease of 83.6% is a transformative change in terms of operational improvement that will allow policy teams to have deployment velocities impossible under manual processes. The 97.1% reduction in cost per deployment shows rapid return on investment for pipeline implementation with breakeven possible after about 35 deployments assuming $4300 implementation cost amortized over deployment volume.

6.5 Discussion

The experimental results show that automated Policy-as-Code deployment via CI/CD pipelines solves important security and compliance issues that are found in proposed research. (Paul et al.; 2024a) emphasized that Infrastructure as Code security challenges are surpassing the traditional security procedures and there is a need to embed security principles in CI/CD pipelines to provide continuous validation. The current implementation provides empirical evidence for this approach, as the time is reduced by 83.6% as compared to manual deployment with 100% cryptographic signature verification and secret detection rates. (Alonso et al.; 2023a) highlighted the need for embracing IaC via DevSecOps philosophy to create trustworthy infrastructure and that security has to be implemented throughout the development lifecycle. The multi-stage validation architecture implemented here operationalises this philosophy where 47.6% of defect is detected in unit testing and 14.3% of defect is detected in integration testing proving that distributed validation across the stages of pipeline provides defence-in-depth defect detection. (Agarwal et al.; 2022a) proposed compliance-as-code, which is a modernization from manual document-based compliance to automated continuous compliance process, which is also illustrated with the current pipeline architecture, automated policy validation, cryptographic signing and audit logging are enabling comprehensive regulatory compliance automation.

The scalability and performance characteristics are consistent with well-established research in the field of CI/CD automation, and extend the research to the domain of policy as code. (Baitha et al.; 2024b) highlighted integration complexity, inconsistent testing environments and deployment issues as perpetual CI/CD challenges; current implementation solves these challenges with containerized CodeBuild environments providing consistent test execution context, end-to-end multi-stage testing providing 94.7% coverage and two-way bundle distribution for eliminating any inconsistencies during deployment. The linear relation of scalability ($R2 = 0.987$) with (4.3 sec) marginal cost per policy has verified the fact that the pipeline architecture has a predictable performance increase that will help organizations predict execution time for different size repositories. (Singh et al.; 2024) compared static analysis tools for Terraform IaC security (Checkov, Tfsec, Tflint, Terrascan), which highlights that the use of the right tool is essential to identify the misconfigurations and coding errors in IaC, the integration of OPA validation and Conftest for integration testing, and Gitleaks for secret scanning, shows that tool composition with multiple security dimensions can provide a complete coverage more than the single tool approaches.

Security control effectiveness results provide empirical evidence for theoretical models of policy enforcement through compliance automation. (Caracciolo; 2023) examined implementations of Policy-as-Code using open-source tools such as tfsec, Regula, Cloud Custodian, and Gatekeeper, by pointing out that default configurations of infrastructure are often not secure and that automated compliance checks result in non-compliant resources not being deployed. The present implementation's 100% signature verification success rate and zero false negative secret detection validates the effectiveness of cryptographic integrity controls and automated security scanning as being perfect or near-perfect when properly configured. (Adam et al.; 2019b) proposed cognitive compliance approaches for analysis, monitoring, and enforcement of compliance in cloud environments by mapping the regulatory requirements to executable code; the present architecture implements this concept with Rego policies for encoding compliance requirements

as executable rules, which are evaluated automatically against infrastructure configurations.(Wong et al.; 2024) dealt with the complexity of cloud security compliance with knowledge graphs mappings of controls and although the current implementation does not use knowledge graphs, the structuring of policy into packages (terraform.security, terraform.compliance, terraform.cost) achieves similar benefits of structured compliance requirement management which allow for traceability from regulatory requirements to enforcement mechanisms. The 47 millisecond signature verification overhead proves that security controls cause negligible performance penalties, contrary to expectations that thorough security validation requires one to make performance trade-offs.

7 Conclusion and Future Work

This research is able to prove that automated policy-as-code lifecycle management with a multi-environment CI/CD pipeline for Open Policy Agent is technically feasible and operationally beneficial. The pipeline built in the DevSecOps practices such as the use of Terraform-based IaC provisioning, multi-stage validation, automated testing, secret scanning, and cryptographic signing while enforcing stricter and stricter governance across development, UAT, and production.

Evaluation results demonstrate that there are substantial benefits over manual processes in that deployment times are reduced by 83.6% (4.2-9.3 minutes), consistency improved (coefficient of variation decreasing from 26.6% to 14.3%) and security controls achieved perfect or near perfect effectiveness without significant performance overhead. The system showed predictable scalability (R2=0.987) with a marginal cost of 4.3 seconds per policy that supports repositories with up to 200 policies within a 15 minute execution target. High code coverage (94.7%), and meaningful defect detection rates at different stages of the test complements good quality assurance. Cost efficiency is improved dramatically at 97.1% reduction per deployment and 52x more frequent deployments.

While the research is limited by having to use a single codebase, having an evaluation window of four weeks, and having some limitations for performance beyond 200 policies, the results support the conclusion that DevSecOps aligned automation can greatly improve speed, consistency, security and cost effectiveness in policy-as-code workflows.

Future researches should be conducted to cross-platform between major CI/CD platforms to create a comparative performance baselines. Enhancements like canary policy roll outs, automatic roll back capability, detecting drift and integrating data observability would be good for operational resilience and to monitor for policy effectiveness. Broader case studies across the organizations might measure adoption impacts and culturally based variables besides technical ones.

Scalability improvements, parallelized testing, increments of validation, intelligent selection of tests, and policy architectures, regarding repositories greater than 200 policies. Extending the framework to alternative policy engines (e.g., Kyverno, Sentinel, Styra DAS), and exploring reusable policy composition patterns would provide an additional form of validation for the generalizability of the framework and promote large-scale collaborative policy development.

References

Adam, C., Bulut, M. F., Hernandez, M. and Vukovic, M. (2019a). Cognitive Compliance: Analyze, Monitor and Enforce Compliance in the Cloud, *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pp. 234–242. ISSN: 2159-6190.
URL: *https://ieeexplore.ieee.org/document/8814576*

Adam, C., Bulut, M. F., Hernandez, M. and Vukovic, M. (2019b). Cognitive Compliance: Analyze, Monitor and Enforce Compliance in the Cloud, *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pp. 234–242. ISSN: 2159-6190.
URL: *https://ieeexplore.ieee.org/document/8814576*

Adanigbo, O. S., Adekunle, B. I., Ogbuefi, E., Odofin, O. T., Agboola, O. A. and Kisina, D. (2025). Implementing Zero Trust Security in Multi-Cloud Microservices Platforms: A Review and Architectural Framework.

Agarwal, V., Butler, C., Degenaro, L., Kumar, A., Sailer, A. and Steinder, G. (2022a). Compliance-as-code for cybersecurity automation in hybrid cloud, *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, pp. 427–437.

Agarwal, V., Butler, C., Degenaro, L., Kumar, A., Sailer, A. and Steinder, G. (2022b). Compliance-as-code for cybersecurity automation in hybrid cloud, *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, pp. 427–437.

Alonso, J., Piliszek, R. and Cankar, M. (2023a). Embracing iac through the devsecops philosophy: Concepts, challenges, and a reference framework, *IEEE Software* **40**(1): 56–62.

Alonso, J., Piliszek, R. and Cankar, M. (2023b). Embracing IaC Through the DevSecOps Philosophy: Concepts, Challenges, and a Reference Framework, *IEEE Software* **40**(1): 56–62.
URL: *https://ieeexplore.ieee.org/document/9915031*

Baitha, S., Soorya, V., Kothari, O., Rajagopal, S. M. and Panda, N. (2024a). Streamlining software development: A comprehensive study on ci/cd automation, *2024 4th International Conference on Sustainable Expert Systems (ICSES)*, pp. 1299–1305.

Baitha, S., Soorya, V., Kothari, O., Rajagopal, S. M. and Panda, N. (2024b). Streamlining software development: A comprehensive study on ci/cd automation, *2024 4th International Conference on Sustainable Expert Systems (ICSES)*, pp. 1299–1305.

Caracciolo, M. (2023). *Policy as Code, how to automate cloud compliance verification with open-source tools*, laurea, Politecnico di Torino.
URL: *https://webthesis.biblio.polito.it/26908/*

Grigorieva, N. M., Petrenko, A. S. and Petrenko, S. A. (2024). Development of Secure Software Based on the New Devsecops Technology, *2024 Conference of Young Researchers in Electrical and Electronic Engineering (ElCon)*, pp. 158–161. ISSN: 2376-6565.
URL: *https://ieeexplore.ieee.org/abstract/document/10468425*

Jack, E., Grace, A., Sarkar, S. and Castro, H. (2024). Policy-as-code: Enforcing governance with open policy agent (opa).

Johnson, G. (2022). Economic Research on Privacy Regulation: Lessons from the GDPR and Beyond.
URL: *https://www.nber.org/papers/w30705*

Karra, N. R. (2025). The Impact of DevOps Practices on Software Development Efficiency: A Systematic Review.

Lwakatare, L. E., Kuvaja, P. and Oivo, M. (2015). Dimensions of devops, *in* C. Lassenius, T. Dingsøyr and M. Paasivaara (eds), *Agile Processes in Software Engineering and Extreme Programming*, Springer International Publishing, Cham, pp. 212–217.

Marandi, M., Bertia, A. and Silas, S. (2023). Implementing and automating security scanning to a devsecops ci/cd pipeline, *2023 World Conference on Communication Computing (WCONF)*, pp. 1–6.

Paul, A., Manoj, R. and S, U. (2024a). Amazon web services cloud compliance automation with open policy agent, *2024 International Conference on Expert Clouds and Applications (ICOECA)*, pp. 313–317.

Paul, A., Manoj, R. and S, U. (2024b). Amazon Web Services Cloud Compliance Automation with Open Policy Agent, *2024 International Conference on Expert Clouds and Applications (ICOECA)*, pp. 313–317.
URL: *https://ieeexplore.ieee.org/document/10612535*

Rahul, B., Kharvi, P. and Manu, M. (2019). Implementation of devsecops using open-source tools, *International Journal of Advance Research, Ideas and Innovations in Technology* **5**: 1050–1051.
URL: *https://api.semanticscholar.org/CorpusID:181938291*

Riaz, S., Asif, A., Khan, Y., Ibrar, M., Afzal, S., Hamid, K., Gul, S. and Iqbal, M. W. (2025). Software Development Empowered and Secured by Integrating A DevSecOps Design, *Journal of Computing & Biomedical Informatics* **8**(02). Number: 02.
URL: *https://www.jcbi.org/index.php/Main/article/view/889*

Rodrigues, G. A. P., Serrano, A. L. M., Vergara, G. F., Albuquerque, R. d. O. and Nze, G. D. A. (2024a). Impact, compliance, and countermeasures in relation to data breaches in publicly traded u.s. companies, *Future Internet* **16**(6).
URL: *https://www.mdpi.com/1999-5903/16/6/201*

Rodrigues, G. A. P., Serrano, A. L. M., Vergara, G. F., Albuquerque, R. d. O. and Nze, G. D. A. (2024b). Impact, Compliance, and Countermeasures in Relation to Data Breaches in Publicly Traded U.S. Companies, *Future Internet* **16**(6): 201. Number: 6 Publisher: Multidisciplinary Digital Publishing Institute.
URL: *https://www.mdpi.com/1999-5903/16/6/201*

Seetharamarao, R. Y. (2023). A unified approach towards security audit and compliance in cloud computing environment, *2023 16th International Conference on Developments in eSystems Engineering (DeSE)*, pp. 623–629.

Shahin, M., Ali Babar, M. and Zhu, L. (2017). Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices, *IEEE Access* **PP**.

Singh, R., Yeboah-Ofori, A., Kumar, S. and Ganiyu, A. (2024). Fortifying cloud devsecops security using terraform infrastructure as code analysis tools, *2024 International Conference on Electrical and Computer Engineering Researches (ICECER)*, pp. 1–6.

Vakhula, O., Opirskyy, I., Vorobets, P., Bobko, O. and Kulinich, O. (2025). Research on Policy-as-Code for Implementation of Role- based and Attribute-based Access Control.

Wong, W., Alomari, Z., Shao, Y. and Satti, M. I. (2024). Knowledge graph to facilitate cloud security compliance with nist cybersecurity framework, *2024 2nd International Conference on Computer, Vision and Intelligent Technology (ICCVIT)*, pp. 1–6.