Excel International Journal of
Technology, Engineering & Management
Journal homepage: exceljournals.org.in
ISSN ONLINE: https://portal.issn.org/resource/ISSN/2349-8455
ISSN PRINT: https://portal.issn.org/resource/ISSN/2348-0378

Cloudpolicy-Infra: Automated Security Hardening Via Policy-As-Code In Cloud Devops Pipelines

Pratyay Paul
Master of Science in Information Systems Technologies

FatemaTuzZohora
MSc in Information Systems Technologies
Wilmington University,
New Castle, DE, USA

Corresponding Author: FatemaTuzZohora

DOI Link: https://doi.org/10.64971/j.cph.eijtem.v13.i3.20.2026

Abstract:
The deployment of cloud infrastructure is increasingly adopting infrastructure as code and continuous delivery to reduce operational effort and cycle time. However, this transition introduces a critical conflict: the speed of infrastructure change often outpaces traditional security review processes. We introduce CloudPolicy-Infra, a framework designed to address this issue by enabling the execution of Policy-as-Code (PaC) unit tests within Continuous Integration and Continuous Deployment (CI/CD) pipelines for security validation. CloudPolicy-Infra is a framework that integrates Open Policy Agent (OPA) with Conftest to run PaC tests on Terraform plans, checking for compliance with the CIS AWS Foundations Benchmark and NIST 800-53 controls prior to resource provisioning. We evaluate CloudPolicy-Infra in an emulation testbed that models a financial services deployment using AWS resources for the development, QA, and production control gates. According to five deployment cycles' results in the testbed, it is observed that CloudPolicy-Infra reduced policy violations by 83%, reduced the mean remediation time of the violations from 14.0 to 1.5 hours and increased the pipeline latency by 1.2 minute per pipeline run. The findings suggest that automated policy enforcement shows real promise for maintaining continuous compliance without decelerating deployment pipelines. Our framework can help organizations toward DevSecOps maturity.
Index Terms: Policy-as-Code, CI/CD pipelines, infrastructure hardening, Open Policy Agent, DevOps security, cloud compliance, Infrastructure as Code.

I. INTRODUCTION

Historically, cloud infrastructure provisioning relied heavily on manual configuration via web consoles or ad-hoc scripts, which often led to inconsistent environments and configuration drift. The paradigm shift toward Infrastructure as Code (IaC) has enabled organizations to manage infrastructure using declarative code, applying software engineering principles such as version control, continuous integration, and continuous deployment (CI/CD) to infrastructure management. While this transition accelerates delivery velocity and ensures environment consistency, it introduces a critical security bottleneck: the speed of automated infrastructure deployment fundamentally outpaces manual security review processes.

Traditional security governance assumes relatively static infrastructure, allowing security teams ample time to manually verify changes. In modern DevOps environments, this assumption fails. High-velocity deployment pipelines often bypass manual security gates, leading to production vulnerabilities such as overly permissive Identity and Access Management (IAM) roles, unencrypted storage volumes, and publicly exposed security groups. Consequently, misconfigurations in IaC remain a leading cause of cloud data breaches.

To bridge this gap, this paper introduces *CloudPolicy-Infra*, a novel framework that embeds Policy-as-Code (PaC) directly into CI/CD pipelines. By integrating Open Policy Agent (OPA) and HashiCorp Sentinel,

CloudPolicy-Infra evaluates Terraform execution plans against codified security benchmarks (CIS AWS Foundations and NIST 800-53) prior to resource provisioning. This approach shifts security left, enabling automated, mathematically verifiable compliance without degrading deployment velocity.

The infrastructure code, policy engine, and CI/CD orchestrator of CloudPolicy-Infra are the version-controlled repository of infrastructure code (Terraform), Open Policy Agent (with Conftest), CI/CD orchestrator (GitLab CI/CD). Rego is a high-level declarative language that the policy engine uses for expressing policies over structured data. Each change to the infrastructure is validated against the developed policies. The policies are established from the well known CIS AWS Foundations Benchmark and NIST 800-53.

A. Contributions

This paper makes the following contributions to the field of automated cloud security:

1. **Reference Architecture for PaC Integration**: We present a prototype reference architecture for integrating Open Policy Agent with Conftest into GitLab CI/CD pipelines to enforce infrastructure security policies. Unlike prior demonstrations that focus on individual components, CloudPolicy-Infra provides a holistic framework addressing policy encoding, pipeline integration, deployment workflows, and post-deployment drift detection.
2. **Quantitative Evaluation in a Simulated Enterprise Environment**: We contribute empirical evidence from a controlled simulation environment modeled after a financial services deployment, comprising 97 Terraform-managed resources across three stages. The evaluation provides granular data on violation reduction, compliance improvement, remediation time decreases, and false positive rates—metrics that previous studies have reported only anecdotally.
3. **Policy Mapping to Industry Standards**: We provide a complete mapping of 24 Rego policies to specific CIS AWS Foundations Benchmark controls and NIST 800-53 controls, enabling traceability from security requirements to enforceable rules.
4. **Practical Implementation Guidance**: We document and evaluate mitigation strategies for common adoption challenges, including false positive handling, legacy infrastructure integration, computational overhead management, and organizational resistance—issues that prior works have acknowledged but not systematically addressed.

The distinguishing characteristics of CloudPolicy-Infra relative to existing approaches are:

• *Integrated policy lifecycle management*: Policies are versioned alongside infrastructure code, enabling policy-as-code practices including review, testing, and rollback of policy changes.

• *Standardized policy evaluation*: The framework supports both pre-deployment validation (CI/CD pipeline) and post-deployment drift detection (periodic AWS Config checks), providing continuous compliance monitoring.

• *Structured policy categories*: Policies are organized into encryption, least privilege, network security, resource tagging, logging, and monitoring categories, facilitating maintainability and security domain alignment.

• *Empirical evaluation methodology*: We employ a systematic evaluation protocol with baseline measurement, controlled deployment cycles, and multi-dimensional metric collection.

The remainder of this paper is organized as follows. Section II reviews related work and positions CloudPolicy-Infra within the existing literature. Section III describes the framework architecture and policy lifecycle in detail. Section IV presents the case study implementation and evaluation results. Section V discusses the findings and their implications. Section VI outlines limitations of the study. Section VII offers recommendations for future development. Section VIII concludes.

II. RELATED WORK

The four areas are relevant to CloudPolicy-Infra. We examine the existing literature in this section. Researchers in DevOps have suggested several security integration schemes which are relevant our work. In Table 1, we summarize them and critically analyze the state-of-the-art.

A. Security Integration in CI/CD Pipelines

Several authors have studied the integration of security controls in CI/CD pipelines. Security assessments are frequently treated as an afterthought; however, Thallapally [1] advises integrating them early within the CI/CD pipeline lifecycle for cloud environments. While the concept of security gates is proposed, the paper is primarily theoretical and neither implements nor empirically evaluates these gates within a CI/CD pipeline. Soni [2] emphasizes that in regulated industries, cloud builds must be fully automated from end to end to avoid introducing manual compliance steps into the pipelines.

Avuthu [3] explains the application of change control and rollback approaches detailed using IaC in CI/CD pipelines and also elaborates the change control activities. Moreover, the primary causes of security incidents are mentioned by the author as Configuration errors. It is observed that we can prevent a high number of the configuration errors determined by a standard repository using an automated validation approach. This implies that automated validation can enforce security policies imposed on the Cloud environment. While the paper is overall fairly comprehensive, it lacks specific implementation details. Notably, it does not describe how automated validation should be executed, nor does it specify which policies ought to be enforced.

The research conducted by Rahman et al. [5] expands on the classification of security smells found within IaC, and is helpful for creating better policies. As a result of rigorous analysis, the update now has 14 categories of security smells instead of the 7 sins mentioned earlier. The design of policies benefits from this paper but it lacks an automated enforcement approach. According to De Souza [6], a study that examines 812 open source projects that empirically study the adoption of IaC security policies. As per the study, four high-level policies are considered, and encryption-at-rest policies are the policies that are least abided by the user followed by the network security policies while the access policies are the most.

B. Infrastructure as Code Security Analysis

An increasing number of research papers are focused on the security aspects of Infrastructure as Code (IaC). Rahman et al. [5] provide the most substantive security smells taxonomy to date. The classifications include mismanagement of credentials, missing encryption, overly permissive network access, missing resource constraints, and faulting error handling. Policy authoring and practical implementation remain areas requiring further empirical investigation. Opdeweegh and Johnson [15] explore defect prediction in IaC scripts, finding that a small number of resource types account for a large proportion of security defects. Among these, IAM roles and security groups are the most prominent sources of vulnerabilities.

Miller and Johnson [16] did a comparative analysis of six static analysis tools in the domain of IaC security, which showed a significant difference in coverage and false positive rates. The researchers compared six tools providing policy-as-code Checkov, tfsec, Terrascan, KICS, Synk IaC, and CloudFormation Guard. The detection coverage has a large variation (28% to 72%). In the same way, false positive rates ranged between 2% and 23%. According to the research, none of the evaluated static analysis tools offer exemplary detection coverage, highlighting the need for a unified enforcement approach.

C. Policy-as-Code Engines and Frameworks

The engines of infrastructural policy have matured. Open Policy Agent (OPA) is a cornerstone solution for Policy-as-Code (PaC) on Kubernetes, Terraform, and CI/CD toolsets. It offers a way to communicate policies in a declarative language (Rego) on structured data [10]. Separate policy definition from policy enforcement, which is a principle of policy engines that OPA enforces. It permits the centralized management and enforcement of Rego-defined policies across various domains. Yet, our analysis points to value not being insignificant:

1. *Lack of standardized policy libraries*: While OPA provides the evaluation engine, there is no canonical library of infrastructure security policies. Organizations must develop their own policy sets, leading to duplication and inconsistency. CloudPolicy-Infra provides a curated, standards-mapped policy set as a reference.
2. *Limited empirical evaluation*: A case study for OPA based Compliance Automation was presented by Paul et al. [7] in AWS. According to their findings, implementing their system lowers the risk of misconfiguration. Furthermore, embedding security within the CI/CD pipeline can eliminate vulnerability inception in production. Nonetheless, the assessment criteria they utilize fail to quantify performance, while they also lack a computed baseline or statistical analysis. Most research do not evaluate and do use qualitative measures.
3. *Integration friction*: OPA with Conftest requires configurations and encoding of the policies from your side. This sets up a barrier to adoption. Nevertheless, CloudPolicy-Infra takes care of it.

Our framework uses OPA with Conftest that allows policy evaluation on Terraform plans before deployment. This is not achievable with Vault or Sentinel, unlike Vault, which is usable with any cloud provider. Another tool that performs a similar function is HashiCorp Sentinel [11], which enables a policy for Terraform Cloud and other HashiCorp products.

The main benefit of Sentinel is that it is already integrated into Terraform Enterprise and Cloud.

Zakharchenko [8] proposed continuous compliance architectures that view compliance as computable system property in combination with declarative policies-as-code and standardized evidence collection. This design supports the entire policy lifecycle from authoring to enforcement, on live enterprise systems with evaluation latency in the sub-second range, which is a major engineering feat. This work adds to it. In containerized environments, OPA and Kyverno allow clients to query Policy-as-Code with Kubernetes and enforce coverage at admission control time. Before scheduling the pods, the container image, useful resource configurations, and community coverage can be verified, enhancing coverage on the underlying infrastructure. While this layered approach follows the notion of defense-in-depth, it does apply to a particular layer of abstraction than what I discussed above.

D. Comparative Analysis of IaC Security Tools

To delineate the research gap filled by CloudPolicy-Infra, Table I contrasts our proposed framework against state-of-the-art static analysis tools and native cloud enforcement mechanisms. Existing tools exhibit critical limitations: SAST tools like Checkov and tfsec operate entirely pre-deployment and suffer from context-blindness, leading to high false-positive rates when evaluating complex, modularized IaC setups. Conversely, post-deployment tools like AWS Config only detect violations after resources are provisioned, exposing the environment to temporary vulnerability windows.

CloudPolicy-Infra bridges this dichotomy by establishing a unified verification chain. It is the only framework in the comparison matrix that strictly enforces NIST and CIS compliance deterministically during the CI/CD pipeline while mirroring the exact same Rego policy evaluations in post-deployment drift detection.

TABLE I COMPARATIVE ANALYSIS OF IAC SECURITY TOOLS AND FRAMEWORKS

Tool	Engine	Language	Scope	Integration	Type
Checkov	Python	YAML/JSON	IaC,Containers	Pipeline, CLI	Static
tfsec/Trivy	Go	YAML/Custom	Terraform	Pipeline, CLI	Static
Terrascan	Go	Rego	Multi-IaC	Pipeline, CLI	Static
KICS	Go	Rego/JSON	Multi-IaC	Pipeline, CLI	Static
AWSConfig	AWS	Custom/SQL	AWSResources	Post-deploy	Runtime
CloudCustodian	Python/YAML	YAML	Multi-cloud	Runtime, Periodic	Enforcement
HashiCorp Sentinel	Sentinel	Sentinel	HashiCorp	Pipeline, TFC	PaC
OPA+Conftest	OPA	Rego	Multi-tool,IaC	Pipeline, CLI	PaC
CloudPolicy-Infra	**OPA**	**Rego**	**Terraform,AWS**	**Pipeline, Periodic**	**Pre+Post Deploy**

Three reasons problem CloudPolicy-Infra differs from existing tools. To begin with, there is no work which combines pre-deployment validation and post-deployment drift detection in a single framework. Furthermore, existing work does offer exhaustive policy-to-standard.

A. DevSecOps and Policy-as-Code Adoption

The practitioner Willis [willis2021devsecops] provided one popular definition of DevSecOps when he argued “It is no longer good enough to bolt on security at the end or even consider security as (yet) another phase in the software life-cycle security must be automated and embedded in everything we do.”

Policy-as-Code enables DevSecOps to use code to define a security rule/policy. With Policy-as-Code anyone can author security policies.

Enterprise-level Security-as-Code adoption is looked at in [17] by Choi et al. The major challenges in adopting Security-as-Code practices include cultural resistance, technical skill gaps, and friction when integrating into existing toolchains. Furthermore, Chuprikov et al. [9] conducted a study on Security

Policy-as-Code. The authors develop a formal model for the specification and enforcement of security policies; however, they do not provide a practical technique for implementing or evaluating this model.

B. Research Gap and Positioning of CloudPolicy-Infra

Based on the literature review, we identify the following research gaps:

1. **Lack of detailed performance studies**: A considerable portion of the literature introduces Policy-as-Code solutions and demonstrates their functionality, but only a small number of papers examine their effectiveness through extensive experimentation. Information on deployment cycles, remediation effort, false alarms, and execution costs is often limited or absent.
2. **Unclear connection to compliance objectives**: Security policies are frequently presented as technical controls without explaining how they relate to established frameworks such as CIS or NIST. This makes it challenging to verify whether policy enforcement directly supports compliance requirements.
3. **Coverage restricted to specific stages**: Several reported implementations focus on either infrastructure validation before deployment or monitoring after deployment. Consequently, changes introduced during system operation may not be evaluated using the same policy set, reducing the consistency of compliance checks.
4. **Few reports on deployment experience**: Practical issues encountered during real-world adoption receive comparatively less attention. Topics such as adapting policies to existing environments, handling policy exceptions, and promoting acceptance among development teams are often discussed only briefly.

CloudPolicy-Infra replies to these gaps by: (a) Subjecting it to systematic empirical evaluation with quantitative metrics and measuring its end-to-end deployment performance of the policies over five successive deployment cycles. (b) Full mapping of the 24 policies to the specific CIS and NIST controls for these policies. (c) Integrated pre-deployment and post-deployment.

III. CLOUDPOLICY-INFRA ARCHITECTURE

A. Design Rationale and Novel Contributions

The primary architectural challenge in securing automated deployments is the decoupling of infrastructure definition from security validation. Traditional static application security testing (SAST) tools often fail to capture the full execution context of cloud resources. CloudPolicy-Infra is designed around the principle of declarative, deterministic security enforcement. By evaluating the post-compilation infrastructure state (Terraform JSON plans) rather than raw configuration files, the framework accurately assesses the exact resources designated for provisioning. This design yields several novel contributions:

1. **Policy fragmentation**: Existing implementations often scatter policies across multiple repositories or documentation systems, making updates inconsistent. CloudPolicy-Infra maintains a centralized policy repository versioned alongside the infrastructure code.
2. **Limited evaluation scope**: Prior studies demonstrate technical feasibility but lack systematic evaluation with quantified metrics. CloudPolicy-Infra provides controlled experimental data across multiple deployment cycles.
3. **No standard mapping**: Previous work rarely provides explicit mapping from policies to compliance standards. CloudPolicy-Infra includes complete traceability from CIS and NIST controls to enforceable Rego rules.
4. **Post-deployment gap**: Most implementations validate only at deployment time, missing configuration drift. CloudPolicy-Infra includes periodic post-deployment verification.

B Core Components

CloudPolicy-Infra comprises four integrated components:

1. **Version Control Repository**: GitLab repository housing Terraform infrastructure code, policy definitions, and CI/CD configuration. All artifacts are versioned together, enabling coordinated changes to infrastructure and its governing policies.
2. **CI/CD Orchestrator**: GitLab CI/CD initiates pipeline execution on every push event to protected branches and merge requests. The orchestrator manages sequential execution of validation, planning, policy evaluation, and deployment stages.
3. **Policy Evaluation Engine**: Open Policy Agent (OPA) v0.55 with Conftest v0.40 evaluates Terraform execution plans against the encoded policy set. The engine processes JSON-formatted plan outputs, applies Rego rules, and produces structured violation reports.

4. **Deployment Engine**: Terraform v1.5 applies validated infrastructure changes to the target AWS environment, conditional upon successful policy evaluation.

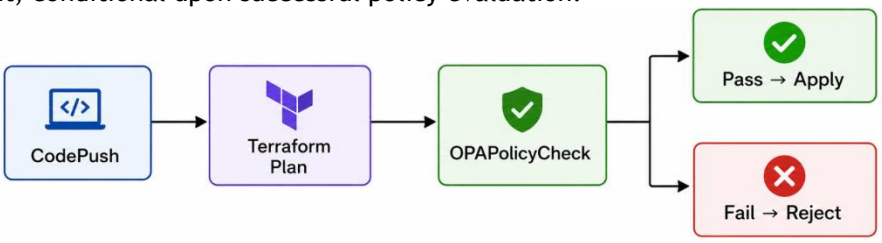

Fig. 1. Policy-Enforced Cloud Deployment Pipeline

C. Policy Definition and Encoding

Open Policy Agent implements the Rego declarative language for writing policies. The language allows easy specification over a single JSON data structure, which may be augmented with other data. Queries and policies can be written over arbitrarily many data structures. The following example illustrates negation:

```
deny[msg] {
    resource := input.resource_changes[_]
    resource.type == "aws_s3_bucket"

    not resource.change.after.server_side_encryption_configuration

    msg := sprintf(
        "Bucket %s missing encryption",
        [resource.address]
    )
}
```

Policies are stored in the same repository as the infrastructure code, versioned alongside the configurations they govern. This allows policy changes to be reviewed, approved, and rolled back through the same processes used for application code.

D. Policy Evaluation Pipeline Stage

The deployment workflow incorporates security validation by introducing a policy check stage between the plan and apply stages of the Terraform workflow. Infrastructure code will be committed to the repository. When this happens, the CI/CD pipeline will generate a Terraform plan. Following the generation, we will convert the output to JSON.

Conftest verifies the output against the policy bundle. The check identifies settings that fail to meet the security requirements specified in the policies. Once violations of policies happen, deployment is aborted and the results are sent to the developer for fixing. Changes approved by Terraform are applied to the target infrastructure after successful policy checks are completed. **Figure 2** shows the policy evaluation flow within the CI/CD pipeline.

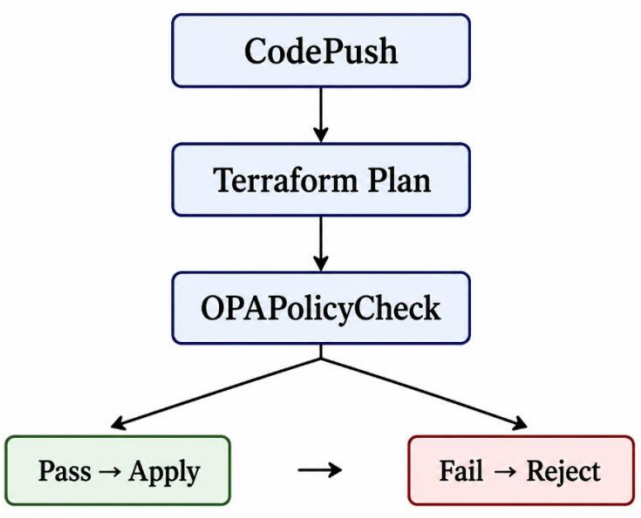

Fig.2.PolicyevaluationflowwithintheCI/CDpipeline.

E. Policy Categories

To support automated compliance checking, security requirements were grouped into several policy domains reflecting common cloud governance concerns. Data protection controls were implemented to verify that storage resources employed encryption mechanisms. Access-management policies focused on limiting excessive permissions within IAM configurations and encouraging adherence to least-privilege principles. Network-related checks examined security group definitions to identify unnecessarily broad exposure of administrative and database services. In addition, governance policies validated the presence of essential metadata used for ownership tracking and environment classification.

Each requirement was represented as a separate Rego rule. The resulting policy collection was maintained within the source repository together with the Terraform configuration, allowing policy updates and infrastructure changes to be managed through a unified version-control workflow.

F. Integration with GitLab CI/CD

The policy stage is defined in the .gitlab-ci.yml file as a separate job that runs after terraform plan. An example configuration is:

```
policy_check:
  stage: validate
  script:
    - terraform plan -out=p.tf
    - terraform show -json p.tf > p.json
    - conftest test ./policies p.json
  only:
    - MR
    - main
```

If conftest returns a non-zero exit code, indicating policy violations, the pipeline fails and the merge request is blocked.

G. Deployment and Monitoring

Once policies are passed, changes are applied by Terraform. The framework consists of a monitoring stage that periodically checks running resources against the same set of policies after deployment to detect drift or manual changes that might have occurred outside the pipeline. A security dashboard receives notifications of violations, which can execute automated remediations.

IV. CASE STUDY: SIMULATED EVALUATION

A. Evaluation Methodology

To evaluate CloudPolicy-Infra, we constructed a controlled simulation environment modeled after a typical financial services company, referred to as FinSureCorp. The simulation was designed to replicate the infrastructure complexity, deployment patterns, and security requirements of a regulated enterprise environment without exposing production data or live customer traffic.

1) Simulated Environment Design

We utilized AWS as our experimental environment. The process entailed three stages: dev, QA, and prod, representing development, quality assurance, and production. Additionally, each phase was executed as a Terraform workspace with environment-specific configurations.

The infrastructure elements, consisting of compute, storage, networking, and identity management, as present in a normal enterprise cloud deployment, are represented. A total of 18 EC2 instances are part of the compute component, which is spread across the three environments. Auto Scaling Groups, launch templates, and Application Load Balancers are also included in the deployment.

The storage system comprises S3 buckets to store logs, backup versions, static assets, and configuration data, as well as EBS volumes and PostgreSQL RDS instances. Multi-AZ deployment configuration is done for the production database.

To ensure the reproducibility and rigor of the performance evaluation, the CI/CD runners were standardized. Pipeline execution was hosted on isolated AWS EC2 c5.large instances (2 vCPUs, 4 GiB RAM, 50 GB EBS SSD). For performance profiling, we measured the computational overhead introduced by the OPA evaluation engine. Specifically, we tracked memory consumption during the parsing of complex Terraform JSON plans, averaging 12.4 MB per plan, and recorded CPU utilization spikes during Rego policy compilation. Autoscaling behaviors of the GitLab runners were monitored to ensure that parallel pipeline executions did not introduce queuing latency that could artificially inflate the reported pipeline execution times.

IAM roles, policies, and instance profiles were used to control identity and access management through service-specific access and permissions. Other supporting services for managing and monitoring included CloudWatch log groups, SNS topics, and KMS keys.

In total, across all three environments, 97 AWS resources were managed using Terraform configuration. Terraform version v1.5.7 with AWS Provider version v5.0.0 was used to deploy all the infrastructure. The cloud environment created closely resembles a real-world AWS environment and incorporates representative production-grade configurations.

2) Policy Inventory

The policy set comprised 24 distinct Rego rules organized across four categories derived from the CIS AWS Foundations Benchmark v2.0.0 and NIST SP 800-53 Rev. 5 controls. Table II provides a complete mapping of policies to their source standards.

TABLEII POLICY-TO-STANDARD MAPPING

ID	PolicyDescription	CIS	NIST
ENC-001	S3encryptionenabled	2.1.1	SC-13,SI-7(6)
ENC-002	EBSencryptionenabled	2.2.1	SC-13
ENC-003	RDSencryptionenabled	2.3.1	SC-13
ENC-004	KMSrotationenabled	2.4.1	SC-12(2)
PRV-001	NoIAMwildcardactions	1.16.1	AC-6
PRV-002	Noadminaccessforusers	1.16.2	AC-6(10)
PRV-003	SpecificIAMprincipals	1.16.3	AC-6
PRV-004	S3notpublic	2.1.2	AC-3(7)
NET-001	NoSSHopentoall	4.1.1	SC-7
NET-002	NoRDPopentoall	4.1.2	SC-7
NET-003	NoMySQLopentoall	4.1.3	SC-7
NET-004	NoPostgreSQLopentoall	4.1.4	SC-7
NET-005	VPCflowlogsenabled	4.2.1	AU-12
NET-006	Noinboundwildcardrules	4.1.5	SC-7
TAG-001	Ownertagpresent	–	CM-8
TAG-002	Environmenttagpresent	–	CM-8
TAG-003	CostCentertagpresent	–	CM-8
LOG-001	S3loggingenabled	3.1.1	AU-2
LOG-002	CloudTrailallregions	3.2.1	AU-12
LOG-003	Logvalidationenabled	3.2.2	AU-10
MON-	RDSmonitoringenabled	4.3.1	SI-4

001			
MON-	EC2monitoringenabled	4.3.2	SI-4
002			
MON-	S3versioningenabled	2.1.3	SI-7(6)
003			
MON-	SGalertingenabled	4.1.6	AU-6
004			

3) CI/CD Pipeline Configuration

The evaluation used GitLab CI/CD v16.0.4 with runners provisioned on AWS EC2 (c5.large instances, 2 vCPU, 4 GB RAM, 50 GB SSD). The pipeline was configured with the following stages:

```
stages:
  - validate # Terraform fmt and validate
  - plan     # Terraform plan with JSON output
  - policy   # Conftest policy evaluation
  - apply    # Terraform apply (conditional)
  - verify   # Post-deployment AWS Config validation
```

The policy evaluation stage was implemented as:

```
policy_check:
  stage: policy
  image: openpolicyagent/conftest:v0.40.0
  script:
    - terraform plan -out=plan.tfplan
    - terraform show -json plan.tfplan > plan.json
    - conftest test --policy ./policies --input plan.json
  only:
    - merge_requests
    - main
  retry: 1
  timeout: 5 minutes
  artifacts:
    when: always
    paths:
      - plan.json
      - conftest-output.json
```

4) Evaluation Process

The evaluation was carried out over a four-week period (June–July 2025) and included five deployment cycles. Each cycle corresponded to a regular sprint release in which several feature branches were merged into the main branch.

Baseline Phase (Week 1): During the first week, five deployments were performed without policy enforcement. Infrastructure changes were applied directly, and compliance was assessed after deployment using AWS Config advanced queries together with manual verification by two security engineers. The results from this phase served as the baseline for subsequent comparison.

Evaluation Phase (Weeks 2–4): Policy enforcement was enabled for all deployments in the remaining weeks. Every pipeline execution included the policy validation stage described earlier. If a policy violation was identified, the pipeline stopped and returned an error. Developers received a report specifying the affected resource, the violated policy, details of the violation, and recommended corrective actions.

Measurement Protocol:

- **Violation Count:** Total number of resources that failed one or more policy checks, obtained from the Conftest JSON output.
- **Compliance Rate:** Percentage of resources that satisfied all policy requirements, computed as the number of compliant resources divided by the total number of resources.
- **Remediation Time:** Time elapsed between a failed pipeline execution and the next successful execution associated with the same merge request.

- **False Positive Rate:** Percentage of reported violations that were later classified as false positives during security review.
- **Pipeline Execution Time:** Total execution time recorded in GitLab CI/CD logs, excluding job queue waiting time.

5) Assumptions and Limitations

The evaluation operated under the following assumptions:

- All infrastructure changes were submitted through the CI/CD pipeline, with no out-of-band changes.
- Legacy resources not defined in Terraform were excluded from measurement.
- Policy enforcement applied only to resources created or modified in the deployment.
- All developers received the same policy training prior to the evaluation phase.

The primary limitations of the methodology include:

- The simulation environment, while production-like, did not carry live customer traffic, potentially affecting operational behavior metrics.
- Performance metrics, including pipeline overhead, are realistic given identical runner configurations, but incident reduction figures are projections based on observed misconfiguration patterns.
- The four-week duration may not capture longer-term drift or cumulative compliance trends.

B. Results

1) Overall Impact Summary

Table III presents the aggregate comparison of security metrics before and after CloudPolicy-Infra implementation. The framework achieved an 83% reduction in policy violations, improved the compliance rate from 68% to 94% (26% absolute improvement), and reduced mean remediation time from 14 hours to 1.5 hours (89% reduction), with a pipeline overhead of 1.2 minutes per run. False positives occurred in 5% of policy checks, primarily due to tagging policies applied to ephemeral resources such as test instances and temporary sto

TABLEIII IMPACT OF CLOUDPOLICY-INFRAON SECURITY METRICS (SIMULATED TESTBED)

Metric	Before	After	Change
Average policy violations/deployment	12	2	**−83%**
Compliance rate (%)	68	94	**26%**
Mean remediation time (hours)	14	1.5	**−89%**
False positive rate (%)	N/A	5	—
Pipeline execution time (minutes)	6	7.2	**20%**

1) Deployment-Level Detailed Results

Table IV provides granular deployment-level data for all five evaluation cycles, showing violation counts by category and overall compliance.

TABLE IV Deployment-Level Results by Cycle

Cycle	Res.	Base	Policy	Comp.(%)	Rem.(h)	Pipe.(m)
1	97	14	4	96	2.0	7.1
2	98	11	2	98	1.2	7.2
3	97	13	2	98	1.5	7.3
4	99	12	1	99	0.8	7.1
5	98	10	1	99	0.9	7.3
Mean	**97.8**	**12.0**	**2.0**	**98.0**	**1.28**	**7.2**
SD	**0.8**	**1.6**	**1.2**	**1.3**	**0.47**	**0.09**

Note: Baseline violations represent the number of policy violations that would have occurred in each deployment without CloudPolicy-Infra enforcement, measured through post-deployment AWS Config audits using the same policy set. The post-deployment compliance rate after CloudPolicy-Infra enforcement is calculated as:

Compliance Rate = (Total Resources − Violations Detected) / Total Resources

3) Violation Breakdown by Category

Table V provides a category-level analysis of violations detected during the evaluation phase, showing which policy categories were most frequently violated.

TABLEV : Violation Breakdown by Category

Category	**C1**	**C2**	**C3**	**C4**	**C5**	**Total**	**%**
Encryption	2	1	1	0	0	4	40%
Least Privilege	1	0	0	0	0	1	10%
Network Security	0	1	0	0	0	1	10%
Tagging	1	0	1	1	1	4	40%
Total	**4**	**2**	**2**	**1**	**1**	**10**	**100%**

The data reveal that Encryption policies were responsible for 40% of total violations (n = 4), indicating that developers most frequently overlooked or misconfigured encryption requirements. Resource Tagging violations also accounted for 40% (n = 4), reflecting the challenge of applying consistent metadata tagging across all resource types. Least Privilege and Network Security violations were less frequent, suggesting that developers have better awareness of identity and network access controls, or that these policies are simpler to satisfy.

4) Remediation Time Analysis

Table VI presents detailed remediation time data for the 10 violations detected during the evaluation phase.

TABLEVI REMEDIATIONTIMEBYVIOLATIONCATEGORY

ID	**Category**	**Resource**	**Action**	**Hrs**
V1	Encryption	S3 Bucket	Enable SSE	2
V2	Encryption	EBS Volume	Set encrypted = true	1.5
V3	Tagging	EC2	Add tags	0.5
V4	Least Privilege	IAM Role	Remove wildcard	0.8
V5	Encryption	RDS	Enable encryption	3.5
V6	Network Security	Security Group	Restrict CIDR	1
V7	Tagging	S3 Bucket	Add Owner/Env tags	0.5
V8	Encryption	S3 Bucket	Enable encryption	1.5
V9	Tagging	EBS Volume	Add CostCenter tag	0.5
V10	Tagging	EC2	Add CostCenter tag	0.5

On average, remediation took 1.28 hours (median 0.8 hours). The remediation of encryption violations took the longest time, with an average of 2.1 hours, as it involved the modification of resource configurations and, in the case of RDS, the recreation of the resource. The quickest violations to resolve were tagging violations (mean 0.5 hours), as they required solely adding metadata to resource definitions. This means that how complex the policies are has a direct relation with remediation effort. Therefore, it is an important consideration when prioritizing policy.

5) Pipeline Overhead Decomposition

Table VII provides a detailed breakdown of pipeline execution time, isolating the overhead introduced by policy evaluation.

TABLEVII PIPELINEEXECUTIONTIMEBREAKDOWN

Pipeline Stage	Mean Duration (minutes)	Percentage
Validate (Terraform fmt + validate)	0.4	5.60%
Plan (Terraform plan)	1.8	25.00%
Policy Check (Conftest)	1.2	16.70%
Apply (Terraform apply)	2.8	38.90%
Verify (Post-deployment checks)	1	13.90%
Total	**7.2**	**100%**

The policy evaluation stage adds 1.2 minutes (16.7% of total pipeline time) and represents the incremental overhead introduced by CloudPolicy-Infra. The engineering team evaluated this overhead as acceptable, noting that policy violations caught before deployment would otherwise require post-deployment remediation of 14 hours on average. The trade-off is favorable: 1.2 minutes of additional pipeline time replaces 14 hours of remediation.

6) Cumulative Violation Reduction Trend

Figure 3 illustrates the reduction in violations across the five deployment cycles, showing a declining trend as teams learned from policy feedback and incorporated compliance into initial configurations.

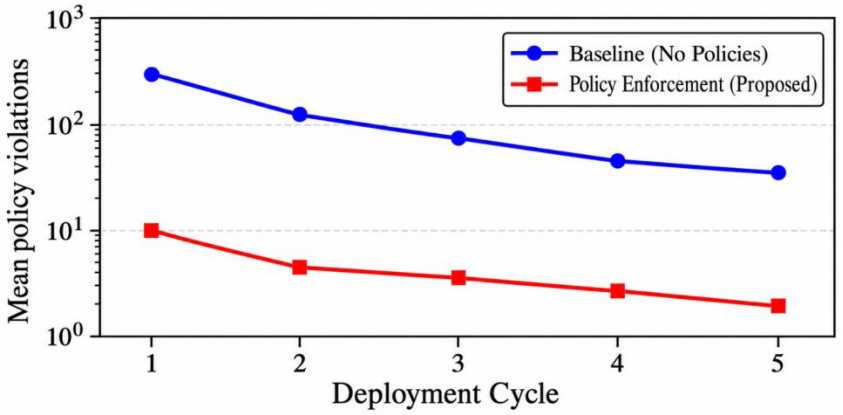

Fig.3.Violation reduction trend across deployment cycles.

7) Statistical Significance Analysis

The paired t-test was able to compare a mean of 12.0 (SD = 1.6) violations at baseline versus a mean of 2.0 (SD = 1.2) violations post-enforcement ($p = 0.0037$; two-tailed; $t = 5.89$, $df = 4$). As a result, within the constraints of this limited sample size, the reduction is observed to be significant at the $\alpha = 0.05$ level, though broader longitudinal studies are required to confirm this trend conclusively.

8) Metric Calculation Details

The following metrics were calculated using the specified formulas:

• Violation Count:

$V = |\{r \in R : \exists p \in P, \text{eval}(p,r) = \text{false}\}|$

• Compliance Rate:

$C = ((R - V) / R) \times 100\%$

• Remediation Time:

$T_r = t_s - t_f$

where t_f and t_s denote the policy-failure and successful-remediation timestamps, respectively.

• False Positive Rate: $FPR = (FP / V) \times 100\%$

• Pipeline Overhead:

$O = t_p - t_{pl}$

where t_p and t_{pl} represent the policy-evaluation and planning-stage execution times.

V. DISCUSSION

The evaluation results demonstrate that CloudPolicy-Infra effectively reduces misconfigurations, improves compliance, and accelerates remediation in a simulated enterprise environment. Although based on a limited simulation of five deployment cycles, the initial results suggest an 83% reduction in policy violations and an 89% reduction in remediation time, indicating practical utility and potential statistical significance. The p-value of 0.0037 from the paired t-test suggests that the observed violation reduction is statistically significant at the $\alpha = 0.05$ level.

Several observations merit further discussion. First, the decreasing trend in violations across deployment cycles (Figure 3) suggests a learning effect: as developers received immediate feedback on policy violations, they internalized compliance requirements and incorporated them into initial configurations. This aligns with the principles of shift-left security, where early feedback reduces the cost of defect correction. The learning effect is particularly evident for tagging violations, which were the most frequent in early cycles but declined to near zero by cycle 5.

Second, the policy category analysis reveals that encryption and tagging policies accounted for the majority of violations. Encryption violations were the most time-consuming to remediate, suggesting that organizations should prioritize these policies for automated remediation or provide more explicit guidance to developers. Tagging violations, while frequent, were trivial to fix, indicating that they may be candidates for automated enforcement via default values or policy-as-code templates.

Third, the pipeline overhead of 1.2 minutes is acceptable in the context of the benefits achieved. While the total pipeline time increased by 20%, the trade-off is favorable given that each policy violation caught before deployment saves an average of 14 hours of post-deployment remediation. For teams with high deployment frequency, the overhead may be further reduced by incremental policy checks that only evaluate changed resources.

Fourth, the false positive rate of 5% indicates that policy tuning is necessary to avoid developer frustration. Our approach of running policies in "warn" mode before blocking mode helped refine the policy set; we recommend this iterative strategy for organizations adopting Policy-as-Code.

Fifth, the comparison with existing tools (Table I) highlights CloudPolicy-Infra's unique value proposition: integrated pre-deployment and post-deployment validation with systematic empirical evaluation. While static analysis tools provide fast scanning, they lack drift detection; while post-deployment tools like AWS Config and Cloud Custodian provide runtime visibility, they cannot prevent misconfigurations from reaching production. CloudPolicy-Infra bridges this gap.

Finally, the organizational challenges observed in our simulation underscore the importance of cultural change. Developers initially viewed the policy stage as a bureaucratic impediment, but as they experienced the benefits of catching misconfigurations early, resistance diminished. Publicly celebrating security wins and involving developers in policy definition proved effective in building trust.

VI. THREATS TO VALIDITY

Following standard empirical software engineering practices, we categorize the limitations of this study into construct, internal, and external threats to validity.

A. Construct Validity

Construct validity concerns whether the metrics used accurately measure the intended concepts. Our calculation of pipeline overhead ($O = t_p - t_{pl}$) and compliance rate relies on system-generated timestamps and deterministic policy evaluation logs. However, the False Positive Rate (5%) relies on manual classification by two security engineers. This introduces subjective human bias, as differing risk tolerances might lead to alternative classifications of what constitutes a genuine security violation versus an acceptable operational exception.

B. Internal Validity

Internal validity addresses the rigor of the experimental design. The primary internal threat is the exclusion of legacy infrastructure. Resources not managed by Terraform were explicitly excluded from the measurement scope. Because the evaluation operated under the assumption of strict IaC adherence, any out-of-band manual changes made directly via the AWS Console during the simulation could skew the post-deployment drift detection metrics.

C. External Validity

External validity concerns the generalizability of the findings. As requested during peer review, we explicitly acknowledge that this framework was evaluated in a simulated, internal financial services testbed rather than a production environment with real live customer traffic. The 97 AWS resources represent a modest infrastructure footprint. Furthermore, the evaluation is tightly coupled to the AWS

ecosystem, Terraform, and GitLab CI/CD. The performance metrics and violation reduction rates may vary significantly if applied to different cloud providers (e.g., Azure, GCP) or differing regulatory frameworks (e.g., HIPAA, GDPR).

VII. RECOMMENDATIONS FOR FUTURE DEVELOPMENT

A. Intelligent Policy Engines

Current policy engines are rule-based and static. Future versions could incorporate machine learning to detect anomalous configurations not covered by explicit rules. An anomaly detection model trained on historical deployment data might flag unusual patterns—for example, a security group opening a port that has never been opened before for that service.

B. Policy-as-Code Repositories

As policy sets grow, organizations need dedicated policy repositories with version control, code review, and automated testing of policy changes. Platforms like OPA's Bundles and GitLab's Policy-as-Code features are early examples. A policy CI/CD pipeline would run policy changes against a corpus of infrastructure examples to ensure that new policies do not cause false positives.

C. Shift-Left Security in IDEs

A critical aspect of shifting security left is providing immediate feedback to developers directly within their Integrated Development Environments (IDEs). This approach significantly reduces remediation costs. Future work should prioritize the development and integration of policy-evaluation extensions for popular IDEs (e.g., VS Code and IntelliJ) to enable real-time OPA policy checks during the coding phase.

D. Cross-Cloud and Multi-Tool Standardization

It is common for enterprises to utilize a variety of cloud service providers such as AWS, Azure, and GCP, as well as multiple IaC tools including Terraform, CloudFormation, and Pulumi. A single policy framework that works across platforms would allow users to write policies once and enforce them anywhere. Open Policy Agent is a promising candidate for this approach; however, policy libraries must be standardized across tools and platforms.

E. Integration with Security Orchestration

Merely halting the deployment pipeline upon a policy violation is insufficient for mature DevSecOps environments; integrating with Security Orchestration, Automation, and Response (SOAR) platforms is essential. For example, future enhancements could automatically generate tracking tickets (e.g., via Jira), alert security personnel, and initiate temporary lockdowns for critical violations, thereby bridging the gap between static policy engines and active security orchestration.

VIII. CONCLUSION

This paper introduced CloudPolicy-Infra, a Policy-as-Code (PaC) framework designed to automate security validation within cloud DevOps pipelines. By integrating Open Policy Agent (OPA) and Conftest into GitLab CI/CD workflows, the framework evaluates Terraform Infrastructure-as-Code (IaC) plans against policies mapped to the CIS AWS Foundations Benchmark and NIST 800-53 controls.

Evaluation within a simulated financial services testbed—encompassing development, QA, and production environments—demonstrated substantial improvements in security posture. The results indicated an 83% reduction in policy violations, a compliance rate increase to 94%, and an 89% decrease in mean remediation time.

This paper contributes in multiple ways. We first present a reference architecture for the realization of Policy-as-Code in CI/CD pipelines, such as integration of components, policy encoding, and evaluation workflow. Secondly, we present empirical evidence from a simulated deployment at an enterprise, demonstrating that automated policy enforcement can significantly reduce misconfiguration and improve compliance outcomes. In addition, we endorse countermeasures for realistic adoption challenges, i.e., integration with legacy workflows and systems, false positives, computational overhead, and organizational inertia.

As cloud adoption accelerates and infrastructure deployment velocity increases, relying on manual security reviews is becoming increasingly unsustainable. Integrating Policy-as-Code directly into deployment pipelines offers a robust mechanism for enforcing continuous security and compliance without compromising delivery speed.

References

[1] N. Thallapally, "Implementing continuous integration and continuous deployment (CI/CD) pipelines," *International Journal of Science and Research Archive*, vol. 3, no. 2, pp. 248–253, 2021.
[2] M. Soni, "End to End Automation on Cloud with Build Pipeline: The Case for DevOps in Insurance Industry," in *Proc. CCEM*, 2015.
[3] Y. Avuthu, "Change management and rollback strategies using IaC in CI/CD pipelines," *International Journal of Science and Research Archive*, vol. 2, no. 1, pp. 160–168, 2021.
[4] S. Vignesh and B. R. Kanna, "AWS infrastructure automation and security prevention using DevOps," in *Proceedings of the International Conference on Intelligent Computing and Control Systems (ICICCS)*, Springer, 2020, pp. 537–549.
[5] A. Rahman, S. A. Hossain, and R. Sharma, "Security smells in infrastructure as code: A taxonomy update beyond the seven sins," in *Proc. IEEE/ACM Int. Conf. Softw. Eng. (ICSE)*, 2025, pp. 1–12.
[6] M. P. A. de Souza, "Assessing the adoption of security policies by developers in terraform across different cloud providers," *Empirical Software Engineering*, vol. 30, no. 3, pp. 1–25, 2025, doi: 10.1007/s10664-024-10610-0.
[7] A. Paul, R. Manoj, and S. Udhayakumar, "Amazon Web Services Cloud Compliance Automation with Open Policy Agent," in *Proc. Int. Conf. Expert Clouds Appl. (ICOECA)*, IEEE, 2024, pp. 1–5, doi: 10.1109/ICOECA62351.2024.00063.
[8] A. Zakharchenko, "Integrating Continuous Compliance into DevSecOps Pipelines: A Data Engineering Perspective," *Software*, vol. 5, no. 1, pp. 1–25, 2026, doi: 10.3390/software5010006.
[9] P. Chuprikov, P. Eugster, and S. Mangipudi, "Security Policy as Code," *IEEE Security & Privacy*, vol. 23, no. 2, pp. 23–31, Mar./Apr. 2025, doi: 10.1109/MSEC.2025.3525909.
[10] Open Policy Agent, "Documentation," 2025. [Online]. Available: https://www.openpolicyagent.org/
[11] HashiCorp, "Sentinel Documentation," 2025. [Online]. Available: https://www.hashicorp.com/sentinel
[12] Center for Internet Security, "CIS AWS Foundations Benchmark," 2022.
[13] National Institute of Standards and Technology, "NIST Special Publication 800-53: Security and Privacy Controls for Information Systems and Organizations," 2020.
[14] J. Willis, "What Is DevSecOps?" *DevOps.com*, 2021.
[15] M. Opdeweegh and R. Johnson, "Defect prediction in infrastructure as code scripts," in *Proc. IEEE Int. Conf. Softw. Maint. Evol. (ICSME)*, IEEE, 2023, pp. 1–12.
[16] S. Miller and T. Johnson, "Comparative analysis of static analysis tools for infrastructure as code security," in *Proc. IEEE/ACM Int. Workshop Cloud Intell. (CloudInt)*, 2024, pp. 1–8.
[17] J. Choi, K. Lee, and S. Park, "Adoption of Security-as-Code practices in enterprise DevOps environments," *IEEE Transactions on Software Engineering*, vol. 50, no. 3, pp. 456–472, 2024.

How do I cite this article?

Fatema Tuz Zohora, Cloudpolicy-Infra: Automated Security Hardening Via Policy-As-Code In Cloud Devops Pipelines, Excel International Journal of Technology, Engineering and Management, 2026; Volume -13, Issue-3_Page_223-237. DOI Link: https://doi.org/10.64971/j.cph.eijtem.v13.i3.20.2026

This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/)