



PREVENTING AMAZON S3 CLOUD STORAGE MISCONFIGURATION USING INFRASTRUCTURE-AS-CODE: A POLICY- ENFORCED SECURITY FRAMEWORK

A Prototype for Pre-Deployment Security Validation Using Terraform and Open Policy Agent

ASRAR SB

1M.Tech Data Engineering, 2024–2026

1Department of Computer Science / Data Engineering, GM University, Davangere,
Karnataka, India

Dr. Neelambike S

ABSTRACT

Cloud infrastructure misconfiguration remains a practical security concern because infrastructure definitions can contain permissive access controls, missing encryption, or insecure transport settings before deployment. This paper presents a prototype policy-enforced security framework for identifying Amazon Simple Storage Service (Amazon S3) misconfigurations in Infrastructure-as-Code (IaC). The framework combines Terraform, a lightweight Python input-generation layer, Open Policy Agent (OPA), and Rego policy rules. Two Terraform configurations were prepared: a secure configuration and an intentionally insecure configuration. The Python layer extracts six selected S3 security attributes from the Terraform source configuration and converts them into structured JSON. OPA evaluates the JSON against six policies covering public access control and restriction, server-side encryption, and secure transport. A test harness compares expected and actual policy violations and stores consolidated evidence in JSON and CSV formats. In the reported local evaluation, the secure configuration produced zero expected and zero actual violations, while the insecure configuration produced six expected and six actual violations. All six individual policy tests passed. A 100-repetition local benchmark produced a mean of 78.227 ms and a median of 61.632 ms for the secure input, and a mean of 51.441 ms and a median of 49.993 ms for the insecure input. These timings include OPA process startup and command-line overhead and therefore should be interpreted as preliminary end-to-end prototype measurements rather than pure policy-evaluation latency. The work demonstrates the feasibility of combining IaC and policy-as-code for repeatable S3 security validation while identifying evaluated Terraform plan integration and automated live deployment blocking as future extensions.

IndexTerms - Infrastructure-as-Code, Amazon S3, cloud security, misconfiguration, Open Policy Agent, Rego, policy-as-code, Terraform, DevSecOps.

I. INTRODUCTION

Cloud computing has changed the way organizations provision and operate infrastructure, but the speed and repeatability of cloud deployment also increase the impact of configuration mistakes. Infrastructure-as-Code (IaC) tools such as Terraform represent infrastructure as version-controlled source definitions, making provisioning more consistent but not automatically secure. Recent empirical work examining 812 open-source Terraform projects found that the presence of IaC does not by itself prevent security and privacy risks and reported that encryption-at-rest practices were among the less frequently adopted policy categories. [1]

Amazon S3 is a widely used object-storage service and its security posture depends heavily on configuration choices related to public access, access policies, encryption, and transport security. A misconfigured bucket can weaken confidentiality and create unintended exposure. Research on AWS IaC security has therefore increasingly emphasized testing and policy enforcement before infrastructure reaches production. Parkhomenko and Savonik describe an Infrastructure-as-Tested-Code approach in which automated security assertions are applied to Terraform-defined AWS resources before deployment, including S3 permissions. [2]

Policy-as-Code provides a complementary mechanism by expressing security requirements as machine-evaluable rules. Open Policy Agent (OPA) is a general-purpose policy engine that can evaluate structured input against Rego policies. Recent work on AWS cloud compliance automation demonstrates the use of OPA to validate infrastructure definitions and flag policy violations

within a DevSecOps workflow. [3] A broader policy-as-code study also describes embedding declarative security controls into IaC workflows to support shift-left prevention of misconfiguration. [4]

Despite these approaches, there is a practical need for small, understandable, testable prototypes that demonstrate how a specific cloud service can be protected using IaC and policy-as-code. The present study addresses this need by focusing on six S3 security controls and by building a local proof-of-concept that can be reproduced without deploying resources to AWS. The contribution is not a claim of complete AWS security coverage; rather, it is a structured framework prototype that links Terraform examples, automated input preparation, OPA/Rego policy evaluation, test validation, and machine-readable result reporting.

The objectives of the study are: (i) to define a focused set of S3 security policies for IaC validation; (ii) to convert selected Terraform security attributes into OPA-compatible JSON; (iii) to evaluate secure and intentionally insecure configurations using OPA and Rego; (iv) to verify policy behavior through individual tests and consolidated expected-versus-actual comparisons; and (v) to obtain an initial local timing measurement of the validation workflow.

II. RELATED WORK

Research on IaC security has established that declarative infrastructure does not remove the possibility of security mistakes. Verdet et al. studied security-policy adoption in Terraform across AWS, Azure, and Google Cloud by examining 812 open-source projects and using static-analysis tools. Their findings indicate that access policies are commonly represented, whereas encryption-at-rest controls receive comparatively less attention, supporting the importance of explicit security checks in IaC. [1]

Parkhomenko and Savonik proposed Infrastructure as Tested Code for AWS and argued that infrastructure should receive systematic security testing similar to application code. Their proof of concept covers Terraform-defined AWS resources, including S3 permissions, and emphasizes pre-deployment validation and test assertions. [2] This work is closely related to the present study because both treat infrastructure configuration as testable security-relevant code.

Paul, Manoj, and Udhayakumar investigated AWS cloud compliance automation using OPA. Their approach evaluates infrastructure definitions against specified policies and flags violations so that insecure changes can be stopped in a DevSecOps pipeline. [3] The present framework follows the same policy-enforcement principle but uses a deliberately small S3-focused policy set and a local JSON-based prototype input layer.

Yatam discusses policy-as-code embedded within IaC and describes the shift-left value of enforcing security controls before deployment. [4] The current work similarly separates policy logic from infrastructure examples, which makes the six security rules independently testable and maintainable.

The distinction of the present work is therefore its focused implementation and transparent evaluation: six S3 controls are encoded as Rego policies, secure and insecure Terraform examples are generated into a common JSON representation, individual policies are tested, and results are consolidated into JSON/CSV evidence. The prototype intentionally avoids claiming direct AWS enforcement or full Terraform semantic parsing, leaving those as future research and engineering extensions.

III. RESEARCH METHODOLOGY

3.1 Study Scope and Test Cases

The study uses a controlled prototype evaluation rather than a live-cloud experiment. Two Terraform source configurations are used: one secure configuration designed to satisfy all six selected policies and one intentionally insecure configuration designed to violate all six. The secure example includes S3 public-access blocking, server-side encryption using AES256, and a bucket policy denying insecure transport. The insecure example is used as a negative test case so that expected policy violations are known in advance.

3.2 Security Policy Model

The framework defines six policy identifiers. Four policies address S3 public-access protection settings, one addresses server-side encryption, and one addresses secure transport. The policies are expressed in Rego and evaluated by OPA against structured JSON input.

Policy ID	Security control	Expected secure state
S3-001	Block public ACLs	block_public_acls = true
S3-002	Ignore public ACLs	ignore_public_acls = true
S3-003	Block public bucket policies	block_public_policy = true
S3-004	Restrict public buckets	restrict_public_buckets = true
S3-005	Require server-side encryption	encryption_enabled = true
S3-006	Enforce secure transport	secure_transport = true

3.3 Input Generation and Data Flow

OPA evaluates structured input rather than raw Terraform source in this prototype. Therefore, a Python input generator acts as a preprocessing layer. Its input is the Terraform source configuration (.tf file), and its output is a JSON representation containing the selected S3 security attributes required by the Rego policies. The generated JSON is then supplied to OPA. This is a lightweight source-text extraction approach; it is not a full Terraform parser and does not yet use Terraform's evaluated plan JSON or resource graph.

3.4 Test Procedure

For each configuration, the workflow generates JSON, executes OPA against the six Rego policies, captures the policy result, and compares the detected violations with the predefined expected set. A secure input is expected to produce zero violations. The insecure input is expected to produce all six violations. In addition, each policy is tested individually to confirm that the policy logic behaves as intended.

3.5 Benchmark Procedure

A local benchmark was executed with 100 repetitions for the secure and insecure inputs. The reported measurement includes the time associated with invoking the OPA command-line process and related command-line overhead. The purpose of this benchmark is to characterize the prototype workflow on the development environment, not to establish production-scale latency or a platform-independent performance guarantee.

IV. SYSTEM DESIGN AND IMPLEMENTATION

4.1 Framework Architecture

The framework consists of five main stages: Terraform configuration, Python input generation, OPA/Rego policy evaluation, test and result consolidation, and JSON/CSV evidence storage. The architecture separates infrastructure definition from security policy logic. This separation supports independent policy testing and allows security requirements to be changed without rewriting the Terraform examples.

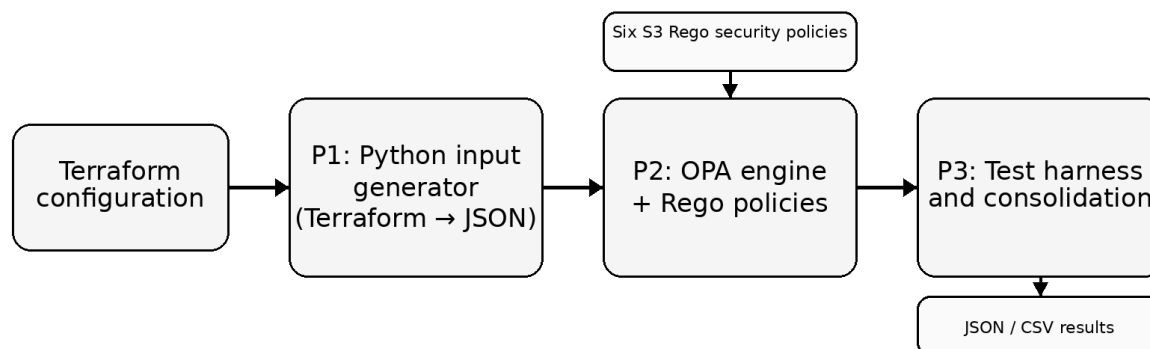


Figure 1. Architecture of the policy-enforced S3 security validation framework.

4.2 Process Flow

The process begins when a secure or insecure Terraform configuration is selected. The Python generator extracts the selected S3 security attributes and creates JSON. OPA receives the JSON and evaluates the six Rego rules. The test harness then checks whether the observed violations match the expected result. Finally, the evaluation artifacts are stored as JSON and CSV files for review, comparison, and reporting.

4.3 Policy Evaluation Logic

Conceptually, each policy can be represented as a Boolean security predicate. For a configuration x , a policy P_i returns compliant when the required security attribute is satisfied and returns a violation otherwise. The overall violation set is the union of all failed policies:

$$V(x) = \{ P_i \mid P_i(x) = \text{violation} \}, \quad i = 1, 2, \dots, 6$$

The test harness compares $V(x)$ with the expected violation set $E(x)$. A test is considered passed when the observed and expected sets are equal. This set-based comparison avoids relying only on a total violation count because it also verifies which policy identifiers were triggered.

4.4 Implementation Environment

The prototype was implemented on a Windows 11 development environment using Terraform v1.16.5, Open Policy Agent, Python, JSON, CSV, and Visual Studio Code/PowerShell. The implementation was deliberately kept local and does not require AWS credentials for the reported tests. No relational database or live AWS deployment component is part of the current evaluated prototype.

V. RESULTS AND DISCUSSION

5.1 Consolidated Security Test Results

The secure configuration produced the expected zero violations, while the intentionally insecure configuration produced the expected six violations. The result demonstrates correct behavior for the two prepared test cases. It should not be interpreted as evidence that every possible S3 misconfiguration will be detected.

Configuration	Expected violations	Actual violations	Status
Secure configuration	0	0	PASS
Insecure configuration	6	6	PASS

Table 1. Consolidated results for secure and intentionally insecure configurations.

5.2 Individual Policy Validation

The six individual policy tests all passed. This confirms that each of the six policy identifiers produced the expected behavior in the prepared test inputs. Individual testing is useful because a successful aggregate result alone could hide a policy-specific defect if one rule failed while another produced an unexpected result.

Measure	Result
Individual policy tests	6 / 6 passed
Secure expected vs. actual	0 vs. 0
Insecure expected vs. actual	6 vs. 6

Table 2. Policy-level validation summary.

5.3 Local Benchmark Results

The local benchmark was repeated 100 times for each input. The secure input produced a mean of 78.227 ms and a median of 61.632 ms. The insecure input produced a mean of 51.441 ms and a median of 49.993 ms.

Input	Repetitions	Mean (ms)	Median (ms)
Secure input	100	78.227	61.632
Insecure input	100	51.441	49.993

Table 3. Reported local benchmark results.

The difference between the secure and insecure timings should not be interpreted as evidence that insecure policy evaluation is inherently faster. The measurement includes process startup and command-line overhead, and the observed values are affected by the local operating environment. A controlled benchmark using a persistent OPA process, larger workloads, repeated independent runs, and statistical confidence intervals would be needed for a rigorous performance study.

5.4 Discussion

The main result is functional rather than performance-oriented: the framework correctly distinguished the supplied secure and insecure configurations and each individual policy test passed. This supports the feasibility of using policy-as-code as a repeatable validation layer for selected S3 security controls. The finding is consistent with prior research arguing that IaC should be subjected to systematic security testing and that policy enforcement can shift security validation earlier in the development lifecycle. [2], [3], [4]

The experiment also illustrates the value of separating policies from infrastructure. The six controls are maintained in Rego rather than embedded in the Terraform examples, allowing the same security rules to be evaluated against different prepared JSON inputs. This separation can support maintainability and auditability as the policy set grows.

However, the current prototype should be interpreted carefully. The Python input generator uses lightweight source-text extraction and therefore does not model all Terraform semantics, module expansion, variable resolution, provider defaults, or resource dependencies. Consequently, the current results validate the prototype's prepared examples rather than the complete security semantics of arbitrary Terraform projects.

VI. LIMITATIONS AND FUTURE WORK

First, the study uses a small controlled test set consisting of secure and intentionally insecure examples. It does not establish false-positive or false-negative rates across a representative corpus of Terraform projects. Second, the input generator does not yet consume Terraform's evaluated plan JSON, which limits semantic coverage. Third, the framework currently reports violations but does not automatically block a real AWS deployment. Fourth, the benchmark includes OPA process startup and CLI overhead and therefore should not be treated as a production performance guarantee.

Future work will focus on replacing lightweight source-text extraction with evaluated Terraform plan or graph-based input, expanding the policy library to cover additional S3 controls and related AWS services, generating a larger and more diverse test corpus, and integrating the validation stage into a CI/CD deployment gate. A further evaluation should measure detection coverage, false positives, false negatives, execution time under controlled conditions, and the effect of policy evaluation on developer workflow. Integration with a real AWS test account can then be considered under controlled and credential-isolated conditions.

VII. CONCLUSION

This paper presented a prototype policy-enforced security framework for preventing selected Amazon S3 configuration weaknesses in Infrastructure-as-Code. The framework combines Terraform, Python-based input preparation, Open Policy Agent, Rego policies, automated testing, and JSON/CSV result reporting. In the reported local evaluation, the secure configuration produced zero violations, the intentionally insecure configuration produced six violations, and all six individual policy tests passed.

The results demonstrate the feasibility of repeatable pre-deployment security validation for the selected S3 controls, while the limitations make clear that the current implementation is a prototype rather than a complete production deployment gate. The most important next step is to connect policy evaluation to Terraform's evaluated representation and then validate the approach on a larger and more representative set of configurations.

ACKNOWLEDGMENT

The author acknowledges the guidance and academic support provided by the Department of Data Engineering, GM University, Davangere, Karnataka, and the project guide during the development and evaluation of the prototype.

REFERENCES

- [1] A. Verdet, M. Hamdaqa, L. D. Silva, and F. Khomh, "Assessing the adoption of security policies by developers in Terraform across different cloud providers," *Empirical Software Engineering*, 2025, doi: 10.1007/s10664-024-10610-0.
- [2] I. Parkhomenko and M. Savonik, "Testing-based prevention of misconfiguration threats in AWS infrastructure as code," *Cybersecurity. Education, Science, Technique*, 2025, doi: 10.28925/2663-4023.2025.29.887.
- [3] A. Paul, R. Joseph Manoj, and S. Udhayakumar, "Amazon Web Services Cloud Compliance Automation with Open Policy Agent," 2024, doi: 10.1109/ICOECA62351.2024.00063.
- [4] S. N. K. Yatam, "Infrastructure as Code with Embedded Security Controls: A Policy-as-Code Approach in Multi-Cloud Environments," *Zenodo*, 2025, doi: 10.5281/zenodo.15818051.