Revista Eletrônica de Iniciação Científica em Computação, 2026, 24:1
ISSN: 3085-8461 • doi: 10.5753/reic.2026.8471

Research Paper

Design of Empirically-Grounded Mutation Operators for Terraform IaC

Isadora Pacheco Ribeiro [Universidade Federal Fluminense | *ribeiroisadora@id.uff.br*]
Vânia de Oliveira Neves [Universidade Federal Fluminense | *vania@ic.uff.br*]

Institute of Computing, Universidade Federal Fluminense, Av. Gal. Milton Tavares de Souza, s/n, São Domingos, Niterói, RJ, 24210-590, Brazil.

Abstract. Defects in Infrastructure as Code (IaC) scripts, particularly Terraform configurations, can cause severe operational failures, yet systematic criteria for designing effective test suites remain scarce. Mutation testing addresses this need by seeding representative faults into code and measuring whether tests detect them. Its effectiveness, however, depends on mutation operators that faithfully model real developer mistakes and no such operators currently exist for Terraform. This paper fills that gap with an empirically grounded catalog of mutation operators derived from a quantitative analysis of 6,749 corrective commits across 54 open-source Terraform repositories. Commit messages were encoded with Sentence-RoBERTa, projected via UMAP, and clustered with HDBSCAN to identify recurring defect patterns. The resulting clusters were synthesized into a defect taxonomy from which mutation operators were formally defined. The contributions are twofold: (1) a catalog of mutation operators for Terraform grounded in real-world defect data, and (2) a replicable multi-stage pipeline for mining defect patterns from version-control histories. The technical feasibility of these operators was validated through a proof-of-concept tool, paving the way for future large-scale automated evaluations.

Keywords: Mutation Testing, Infrastructure as Code, Terraform, Mining Software Repositories, Defect Taxonomy, Software Testing

Received: 15 June 2026 • **Accepted:** 15 June 2026 • **Published:** 10 July 2026

1 Introduction

Infrastructure as Code (IaC) has become a cornerstone of modern software engineering, enabling teams to provision and manage computing infrastructure through machine-readable definition files rather than manual processes [Chittibala, 2023; Carnegie Mellon University, 2019]. Tools such as Terraform allow developers to declaratively specify the desired state of cloud resources, ensuring consistency, repeatability, and auditability across environments [Howard, 2022; Ozdoğan *et al*., 2023]. As organizations scale their cloud operations, IaC scripts have evolved into critical software artifacts whose correctness directly governs the reliability of production systems.

Despite this criticality, quality assurance for IaC remains underdeveloped. Studies report low adoption rates for systematic testing of infrastructure scripts [Shimizu and Kanuka, 2020; Sokolowski *et al*., 2024], and the consequences of undetected defects can be severe. For example, a single configuration error at Amazon Web Services reportedly caused losses of $150 million [Hassan and Rahman, 2022]. A key obstacle is the absence of structured, evidence-based criteria to guide test design for IaC languages such as Terraform. Without such criteria, engineers lack a principled basis for constructing test suites capable of exposing common and high-impact faults before deployment [Sokolowski *et al*., 2024].

This work addresses that gap by applying mutation analysis as a formal test-adequacy criterion for Terraform. Mutation analysis drives the creation of effective tests by requiring them to detect systematically injected faults (modeled as mutation operators [Jia and Harman, 2011; Offutt and Untch, 2001]) which must reflect defects that actually occur in practice. Accordingly, the central goal of this study is the empirical derivation of a set of mutation operators for Terraform, grounded in a large-scale analysis of real-world corrective commits. The investigation is guided by one overarching research question: what mutation operators can be defined for Infrastructure as Code? This inquiry is decomposed into identifying common IaC defect patterns via quantitative repository analysis, and determining how these empirical patterns can be formalized into a concrete set of operators.

To address these questions, the study pursues three integrated objectives: mining a corpus of real-world IaC defects from public repositories, classifying the evidence to synthesize a taxonomy of recurring defect patterns, and formalizing each pattern into a specific mutation operator. Consequently, the primary contributions of this work include a quantitative mining pipeline that isolates 6,749 corrective commits from 54 Terraform repositories, and an unsupervised defect pattern discovery workflow using `Sentence-RoBERTa`, UMAP, and HDBSCAN to cluster commit messages into thematic categories. These components culminate in a machine-readable catalog, establishing a taxonomy of mutation operators derived directly from their statistical prevalence in real-world infrastructure code. The technical feasibility of these operators was validated through a proof-of-concept tool, paving the way for future large-scale automated evaluations.

2 Related Work

Despite the critical role that Infrastructure as Code scripts play in production environments, systematic testing methodologies for these artifacts remain scarce. Sokolowski *et al*. [2024] proposed Automated Configuration Testing

(ACT), a framework for IaC written in general-purpose languages with a Pulumi implementation that automates mocking and test-input generation. Complementarily, Shimizu *et al.* [2024] introduced a test-driven approach for automatically discovering least-privilege permissions through iterative policy refinement. Both approaches represent important advances but presume the existence of an adequate test suite, a prerequisite that itself remains unverified.

Mutation testing has proven effective at revealing faults that conventional metrics such as code coverage miss, even in well-tested systems. Ramler *et al.* [2017] demonstrated that mutation testing exposed major defects in a safety-critical industrial system that had already achieved 100% statement coverage. Mutation operators (small syntactic modifications that simulate common programmer errors) are well established for traditional languages such as Java and C [Jia and Harman, 2011; Offutt and Untch, 2001], but no prior work has defined or validated such operators for IaC languages. This study fills that gap by introducing mutation testing to the Terraform domain.

A complementary strand of research has sought to empirically characterize the defects that arise in IaC scripts. Rahman *et al.* [2018] systematically classified IaC faults using Orthogonal Defect Classification (ODC) across more than 12,000 commits from four open-source projects, showing that "assignment" defects (configuration and syntax errors) are the predominant category, accounting for 49–62% of all faults and occurring significantly more often than in traditional software. Follow-up work identified ten source-code properties (notably `lines_of_code` and `hard_coded_string`) as the most consistent defect predictors, yielding models with higher precision and recall than text-based alternatives [Rahman and Williams, 2019].

Hassan and Rahman [2022] conducted a large-scale study of 4,831 Ansible test scripts from 104 repositories, identifying three testing anti-patterns as significant defect predictors: `local_only_testing`, where tests are confined to local environments; `remote_mystery_guest`, which introduces unreliable external dependencies; and `assertio n_roulette`, which groups multiple assertions into a single test case Hassan and Rahman [2022]. Dalla Palma *et al.* [2022] developed a machine-learning framework using product and process metrics to predict defective IaC scripts, confirming that metrics like `NUMTOKENS` and `LINESCODE` are strong predictors.

3 Methodology

This study follows a quantitative approach to derive mutation operators for Infrastructure as Code (IaC) through two phases: mining public repositories for corrective commits (Section 3.1) and clustering commit messages via NLP to identify recurring defect patterns (Section 3.2).

3.1 Phase 1: Data Collection

Defect-related data were collected from Terraform projects using Mining Software Repositories (MSR) techniques [Hassan, 2008; Kalliamvakou *et al.*, 2014]. The collection process comprises three steps (Figure 1): automated repository selection, manual curation, and keyword-based commit filtering, as summarized in Table 1.

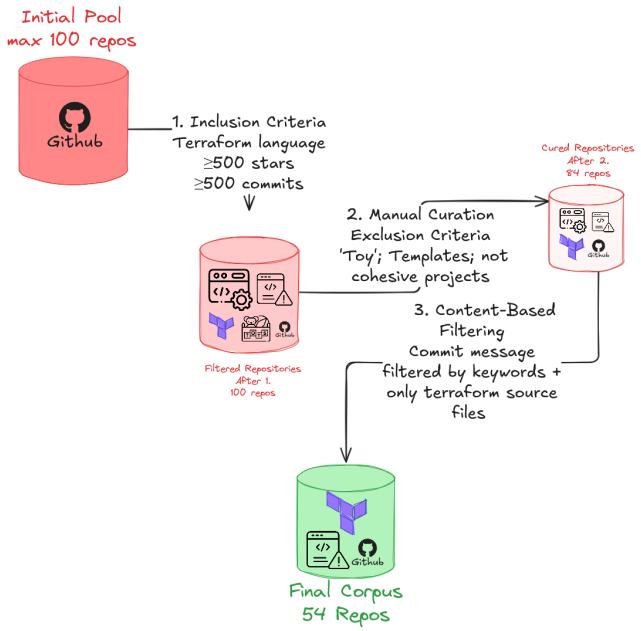

Figure 1. Repository-filtering workflow.

Table 1. Repository filtering stages.

Stage	Description	Count
1. Initial Pool	Repositories from GitHub API matching search criteria.	100
2. Manual Exclusion	Removal of non-representative projects.	-16
3. Content Filtering	Removal of repositories lacking fix-related commits in `.tf` files.	-30
4. Final Dataset	**Repositories used for analysis.**	**54**

Repositories were included if hosted on GitHub with Terraform (HCL) as the primary language [Rahman, 2018] and having at least 500 stars and 500 commits as activity proxies. Non-representative projects (identified by keywords such as `awesome`, `boilerplate`, `demo`, `example`, and `tutorial`) were manually removed [Kalliamvakou *et al.*, 2014]. Commit messages were then filtered using keywords `fix`, `bug`, `issue`, `resolve`, `error`, `failure`, `incorrect` derived from IEEE Computer Society [2010] (e.g., `bug`, `fix`, `error`), retaining only changes to `.tf` files.

A configurable Python pipeline automates data extraction in three stages: (1) repository discovery via the GitHub API, capped at 100 candidates; (2) concurrent mining, where each repository is cloned and commit histories are filtered and diffed; (3) post-processing, where raw data are exported and aggregated. Further implementation details are outside the scope of this paper.

3.2 Phase 2: Corpus Refinement and Thematic Analysis

Simple keyword matching is insufficient to capture the semantic nuance of commit messages [Lee and Chieu, 2021; Amit and Feitelson, 2021; Al-Fraihat *et al.*, 2024; Rebai *et al.*, 2020]. This phase employs transformer-based NLP to cluster messages by developer-described intent (Figure 2).

The initial 7,240 commits are refined by deduplication, PR metadata augmentation, and removal of commits lacking `.tf` modifications, yielding 6,749 unique commits. Each message is then normalized [Yang *et al.*, 2022]: the primary line is extracted, lowercased, and stripped of non-semantic

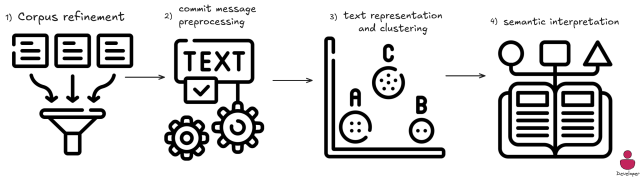

Figure 2. Thematic analysis pipeline.

Algorithm 1 NLP Clustering Pipeline (Phase 2)

Require: Commit messages M (after corpus refinement)
1: **Preprocess:** normalize each $m \in M$ (lowercase; strip URLs and issue IDs)
2: **Embed:** encode each m to a 768-dim vector via Sentence-RoBERTa
3: **Reduce:** project embeddings to 5 dimensions via UMAP
4: **Cluster:** apply HDBSCAN (*min_cluster_size*= 30, *min_samples*= 10)
5: **Label:** lemmatize, filter stopwords, extract top keywords, verify sample, assign label
Ensure: Labeled clusters C and noise set $N \subset M$

artifacts (URLs, issue identifiers).

Algorithm 1 summarizes the NLP clustering pipeline. Messages are encoded into 768-dimensional vectors using **Sentence-RoBERTa** [Reimers and Gurevych, 2019; Tong *et al.*, 2023]; **UMAP** [McInnes *et al.*, 2018] reduces embeddings to 5 dimensions to mitigate the curse of dimensionality [Malzer and Baum, 2020]; **HDBSCAN** [Campello *et al.*, 2013] clusters messages with `min_cluster_size`= 30 and `min_samples`= 10, producing 17 clusters covering 92% of commits (8% outliers). Each cluster is labeled by lemmatizing messages, filtering domain-specific stopwords, extracting frequent keywords, and verifying a random sample manually. Of the corpus, 542 messages (8.03%) were classified as noise; cluster characteristics are presented in Section 4. Further details on parameterization are outside the scope of this paper.

Threats to Validity. Keyword-based search misclassification is mitigated by tight exclusions and `.tf` file restrictions (construct validity); hyperparameter influence on clustering follows established textual heuristics [Malzer and Baum, 2020] and human-in-the-loop validation (internal, conclusion validity). Results are bounded to public, English-language Terraform repositories (external validity).

4 Results

This section presents (1) the thematic clusters derived from the methodology and (2) the formal mutation operator catalog.

4.1 Defect-Related Thematic Clusters

The clustering process identified several themes corresponding to common corrective activities in Terraform projects. Table 2 summarizes the clusters representing authentic defect patterns. Four additional clusters (0, 1, 2, and 3), dominated by Git-workflow terms (`merge`, `conflict`, `rebase`, `main`), were excluded as they reflect version-control operations rather than IaC defects.

4.1.1 Cluster 16: General Maintenance Super-Cluster

Comprising 5,052 messages (74.86% of the corpus), this cluster functions as the central hub for corrective maintenance. Its high frequency of general-purpose verbs (e.g., `add`, `update`, `remove`) reflects a continuous cycle of refactoring. The dominant keywords (`module` 477, `policy` 252, `variable` 242, `cluster` 235, and `test` 198) highlight regular upkeep across IAM access, provider constraints, parameter hygiene, and infrastructure components. Commits coalesce around these core pillars of day-to-day Terraform management.

4.1.2 Cluster 15: Dependency and Version Management

This cluster (231 messages, 3.42%) centers on synchronizing external dependencies. The co-occurrence of `module` (38), `provider` (27), `version` (22), `update` (28), and `bump` (11) indicates frequent version increments, while `issue` (23) and `plan` (11) point to conflicts and debugging of `terraform plan` errors caused by incompatible constraints.

4.1.3 Code Hygiene, Minor Errors, and Linter Fixes

Cluster 11 (93 messages, 1.38%) addresses typographical and spelling errors, dominated by `typo` (83 occurrences); although seemingly minor, such errors in resource names can cause deployment failures. Cluster 4 (68 messages, 1.01%) relates to HCL formatting (`format` 39, `indentation` 6, `whitespace` 7), typically enforced via `terraform fmt`. Cluster 9 (46 messages, 0.68%) targets violations flagged by linters, particularly `tflint` (13 occurrences); the term `ci` (3) indicates linting is integrated into automated pipelines, and the prefix `chore` categorizes these fixes as routine maintenance.

4.1.4 Cluster 14: Security and SCA Compliance

This cluster (49 messages, 0.73%) captures responses to findings from static-analysis security tools, particularly Checkov (`checkov`, 17 occurrences). Two distinct action patterns emerge: *remediation* (commits that fix code to comply with a rule: `address` 7, `fix`) and *suppression* (commits that explicitly skip or ignore findings: `ignore` 15, `suppress` 4, `skip` 3, `exclude` 5), documenting accepted risks or false positives.

4.2 Mutation Operator Catalog

The thematic clusters (Section 4.1) indicate three primary categories of recurrent faults in Terraform code: *version-constraint faults* (Cluster 15), *static-analysis suppression faults* (Cluster 14), and *general maintenance faults* (Cluster 16). Table 3 maps each derived operator to the fault class it models and its source cluster.

4.2.1 Version-Constraint Faults

VCR (Version Constraint Replacement) targets the range delimiter of a version constraint without altering the version literal, simulating accidental loosening or unintended pinning (Cluster 15; `provider` 27, `version` 22, `bump` 11). Three variants enumerate all delimiter permutations: *VCR_1* (= ↔ ~>), *VCR_2* (= ↔ >=), and *VCR_3* (~> ↔ >=), each producing accidental loosening or pinning (e.g., `version = "~> 4.0"` → `version = ">=`

Table 2. Summary of Thematic Clusters Representing Defect Patterns

ID	Cluster Theme	Commit Count	Representative Keywords
16	General Maintenance Super-Cluster	5,052 (74.86%)	add, update, issue, fix, module, policy, variable
15	Dependency & Version Management	231 (3.42%)	provider, module, version, bump, constraint
11	Typographical & Spelling Errors	93 (1.38%)	typo, spelling, copy/paste error, comment
4	HCL Syntax & Formatting	68 (1.01%)	format, formatting, whitespace, indentation
14	Security & SCA Compliance	49 (0.73%)	sca, checkov, ignore, skip, suppress, failure
9	Static Analysis & Linter Fixes	46 (0.68%)	tflint, linter, linting, warning, violation

Table 3. Consolidated catalog of mutation operators traced to empirical fault clusters.

Operator	High-Level Description	Source Cluster
	Version-Constraint Faults	
VCR	Replaces the range delimiter of a version constraint (e.g., `~>` ↔ `>=`).	15
VLI	Increments or decrements a semantic version literal.	15
	Static-Analysis Suppression Faults	
SAR	Mutates an entire suppression-directive comment (e.g., insert, delete, alter rule ID).	14
	General Maintenance Faults	
IPR	Alters IAM policy statements (e.g., remove action, add wildcard).	16
PDR	Mutates a `required_providers` block (e.g., delete block, alter region).	16
MDR	Mutates a `module` block (e.g., change source, alter version).	16
VDR	Mutates a `variable` block (e.g., change type, alter default).	16
PAR	Mutates arguments in `resource` or `module` calls (e.g., flip boolean).	16
RCR	Mutates attributes within a `resource` block (e.g., delete tag, open security group).	16

`4.0"` for VCR_3). **VLI** (Version Literal Increment) modifies a semantic version literal (`MAJOR.MINOR.PATCH`) by ± 1 on exactly one component, reproducing premature bumps or unintended regressions: *VLI_MAJ*$^{+/-}$ ($MAJOR \leftarrow MAJOR \pm 1$), *VLI_MIN*$^{+/-}$ ($MINOR \leftarrow MINOR \pm 1$), and *VLI_PAT*$^{+/-}$ ($PATCH \leftarrow PATCH \pm 1$), each simulating a premature bump or regression.

4.2.2 Static-Analysis Suppression Faults

SAR (Suppression Annotation Replacement) mutates the inline suppression directive (`#<tool>:<action>=<RULE>`) that governs Checkov [Bridgecrew, 2025] or TFSec [Aqua Security, 2025] behavior, leaving the associated resource block unchanged following the principle of minimal mutation (Cluster 14; `checkov` 17, `ignore` 15, `suppress` 4). Four variants cover the observed fault space: *SAR_DEL* deletes the suppression line entirely, re-exposing the suppressed violation; *SAR_INS* inserts a new suppression directive above a resource block, introducing an unwarranted exception (e.g., `#checkov:skip=CKV_AWS_20 "public read accepted"` inserted above an `aws_s3_bucket` block); *SAR_TOOL* replaces `#checkov:` with `#tfsec:`, rendering the directive unrecognized by the target tool; and *SAR_ID* alters one character in the rule identifier, so a typographic error invalidates the rule match.

4.2.3 General Maintenance Faults

Six operators derived from Cluster 16 (5,052 commits, 74.86%; `module` 477, `policy` 252, `variable` 242) each target a single syntactic locus and apply a minimal transformation. **IPR** (IAM Policy Replacement) mutates policy statements (`policy` 252, `role` 196): *IPR_ACT_DEL* removes one item from an `Action` list (missing permission); *IPR_ACT_WC* replaces the list with `"*"` (over-privilege); *IPR_RES_WC* replaces a resource ID with `"*"` (scope expansion). **PDR** (Provider Declaration Replacement) targets `required_providers` blocks (`provider` 167): *PDR_DEL* deletes a provider block (initialization failure); *PDR_ADD* inserts a duplicate block (version conflict); *PDR_CFG* alters one attribute such as `region` (misconfiguration, e.g., `region = "us-east-1"` → `"eu-west-1"`). **MDR** (Module Declaration Replacement) targets `module` blocks (`module` 477): *MDR_DEL* removes a block (incomplete resource graph); *MDR_ADD* inserts a duplicate (double provisioning); *MDR_SRC* changes `source` to an invalid path (incorrect code base); *MDR_VER* shifts one version component by ± 1 (version incompatibility). **VDR** (Variable Declaration Replacement) targets `variable` blocks (`variable` 242): *VDR_TYPE* replaces the declared type (type validation failure); *VDR_DEF* alters or removes the `default` value (unexpected runtime value, e.g., `default = true` → `false`); *VDR_SENS* toggles `sensitive` (exposure of secrets). **PAR** (Parameter Argument Replacement) mutates arguments in `resource` or `module` calls: *PAR_BOOL* flips a boolean literal (feature silently disabled or enabled); *PAR_NUM* adds or subtracts 1 from a numeric literal (quota limit exceeded); *PAR_STR* applies a typographic error to a string value (misspelled unique identifier). **RCR** (Resource Configuration Replacement) mutates attributes within a `resource` block: *RCR_TAG_DEL* deletes a mandatory tag (cost mis-allocation); *RCR_SG_OPEN* sets CIDR to `0.0.0.0/0` or port to all ports (traffic open to the world, e.g., `cidr_blocks = ["10.0.0.0/8"]` → `["0.0.0.0/0"]`); *RCR_ATTR* modifies a critical resource attribute (regression); *RCR_DEL* removes the resource block entirely (dependency failure).

Collectively, the nine operators model recurrent fault

patterns drawn from the empirical analysis of corrective commits. Each targets a single syntactic locus, and its variants apply minimal transformations that emulate faults observed in practice, enabling test-suite assessment against configuration errors spanning dependency versioning, static-analysis suppressions, and resource configuration.

4.3 Proof-of-Concept Implementation

To demonstrate the technical viability of the derived operators, a proof-of-concept tool named *Terramutate* was implemented. The prototype automatically injects mutations defined in the catalog into Terraform projects and executes the associated test suite against each mutant. When applied to a reference IaC project, Terramutate successfully generated mutants for operators from the VCR, PDR, and RCR families and revealed concrete gaps in the project's test coverage. These results provide initial evidence that the proposed operators are syntactically well-formed and practically injectable. A detailed description of the tool's architecture and systematic evaluation of its effectiveness are beyond the scope of this paper, whose primary focus is the empirical design of the operator catalog.

5 Discussion

The thematic clusters and the derived operator catalog reveal several insights about the nature of IaC defects in Terraform, their relationship to prior defect classifications, and their implications for testing practice.

5.1 What the Taxonomy Reveals About IaC Defects

The six clusters covering authentic defect patterns (spanning general maintenance, version management, typographical errors, formatting, security compliance, and linter violations) indicate that defects in Terraform are not uniformly distributed. They are concentrated in maintenance activities and scattered across well-defined, tool-supported quality concerns.

The taxonomy aligns with and extends prior work. Rahman *et al*. [2018] ODC-based classification identified "assignment" defects (configuration and syntax errors) as the dominant category in IaC, accounting for 49–62% of all faults. The operators derived from the General Maintenance Super-Cluster (IPR, PDR, MDR, VDR, PAR, RCR) model precisely this class of assignment defects, providing testable, syntactic instantiations of what ODC labels broadly. Similarly, the Security & SCA Compliance cluster (Cluster 14) parallels Rahman et al.'s "security" defect class, while the Dependency & Version Management cluster (Cluster 15) represents a concern specific to declarative IaC tools that does not feature prominently in traditional software defect taxonomies. This suggests that mutation operators for IaC must extend beyond classical models designed for general-purpose languages.

5.2 Interpreting the Dominance of the General Maintenance Super-Cluster

The General Maintenance Super-Cluster concentrates approximately 75% of corrective commits, which at first glance may appear to undermine the precision of the taxonomy. However, this concentration reflects two independent forces.

First, it mirrors a known phenomenon in software defect data: commit messages for routine corrective maintenance employ generic, high-frequency verbs (e.g., `fix`, `add`, `update`, `remove`) that resist fine-grained semantic clustering. The semantic diversity within this super-cluster (spanning IAM policies, provider blocks, module declarations, variable definitions, parameter arguments, and resource configurations) means that HDBSCAN correctly identified these commits as a single, semantically coherent class of miscellaneous maintenance rather than disaggregating them into fictitiously precise sub-categories. Artificially splitting this cluster via stricter hyperparameters would likely yield sub-clusters anchored to superficial co-occurrences rather than distinct fault classes.

Second, the breadth of this cluster is a design strength: it motivated the derivation of six distinct operators (IPR, PDR, MDR, VDR, PAR, RCR), each targeting a separate syntactic locus within Terraform's HCL grammar. These operators are grounded in both the message-level semantics of the cluster and in the specific Terraform language constructs most frequently referenced in the corresponding corrective diffs. The super-cluster thus functions as a broad empirical motivation for a structurally diverse, syntactically precise operator set.

This distribution also carries a practical signal: the dominance of general maintenance defects implies that no single narrow fault class accounts for most IaC errors, reinforcing the need for a diverse operator catalog rather than a focused, single-category approach.

5.3 Semantic Differentiation of the Smaller Clusters

The five smaller clusters stand apart from the super-cluster precisely because their commit messages contain tool-specific or syntactically distinctive vocabulary. Cluster 15 (Dependency & Version Management) is defined by terms such as `bump`, `constraint`, `provider`, and `version`, reflecting a well-understood engineering activity with a narrow lexical signature. Cluster 14 (Security & SCA Compliance) is anchored by the names of specific tools (`checkov`, `tfsec`), which act as strong semantic discriminators. Clusters 11 and 4 (typographical errors and HCL formatting) are defined by orthographic meta-language (`typo`, `format`, `indentation`), while Cluster 9 (linter fixes) is characterized by tool names (`tflint`) and workflow terms (`ci`).

This pattern suggests a general heuristic for future taxonomy work in IaC: defect clusters with distinctive vocabulary (whether tool names, version identifiers, or meta-linguistic terms) form coherent semantic groups amenable to automated clustering, while generic maintenance language produces unavoidable super-clusters that must be further decomposed via patch-level static analysis of the actual configuration diffs.

5.4 Alignment with Prior Defect Classifications

The proposed taxonomy does not replace but complements existing IaC defect classifications. Compared to Rahman *et al*. [2018], who used ODC to classify faults at the commit level, this work operates at a higher level of abstraction: cluster themes are derived from semantic embeddings of commit messages rather than from manual inspection of individual defects. The alignment between the two results

(assignment defects dominating both taxonomies) provides cross-method validity for the General Maintenance operators.

Dalla Palma *et al*. [2022] identified metrics such as `NUMTOKENS` and `LINESCODE` as defect predictors for IaC scripts. While predictive models identify *where* defects are likely to occur, mutation operators define *what* defects look like. These perspectives are complementary: high-risk scripts identified by predictive models are natural targets for mutation-based test augmentation using the catalog presented here.

Hassan and Rahman [2022] work on Ansible anti-patterns identified testing weaknesses such as `local_only_testing` and `assertion_roulette`. The operators derived here, particularly RCR_SG_OPEN and SAR, address faults that specifically evade superficial structural test suites and require behavior-aware assertions, motivating a move beyond the anti-patterns they identified.

5.5 Implications for IaC Testing Practice

The distribution of clusters carries direct implications for test-suite design. The dominance of general maintenance faults implies that the six operators derived from Cluster 16 will achieve the highest empirical coverage of common defects. For teams with limited testing resources, prioritizing PAR, RCR, and VDR operators targets the most prevalent fault class.

Security operators (SAR, RCR_SG_OPEN) address a smaller but disproportionately high-impact category: a single misconfigured security group or suppressed Checkov rule can expose cloud infrastructure to external attack. Version-constraint operators (VCR, VLI) are critical in environments that rely on automated dependency updates, where inadvertent constraint relaxation can introduce breaking provider changes.

Together, the catalog supports a risk-stratified testing strategy: prioritize broad-coverage general maintenance operators, add security operators for risk-sensitive pipelines, and include version operators where automated dependency management is in use. Integrating the catalog into CI/CD pipelines as a mutation gate (analogous to code-coverage gates) enables proactive identification of test-suite weaknesses before deployment.

5.6 Limitations of the Commit-Message Approach

A core assumption of this work is that corrective commit messages adequately describe the defects they fix. While prior research has shown that commit messages are a reliable proxy for developer intent [Amit and Feitelson, 2021; Rebai *et al*., 2020], they do not always precisely describe the syntactic change applied to the Terraform configuration. Consequently, some operators were derived from both message-level semantics and knowledge of the Terraform language constructs referenced in the messages, rather than from automated patch analysis alone. A deeper inspection of the corresponding `.tf` diffs (using static analysis to categorize the exact AST-level changes) would strengthen the traceability from observed faults to derived operators and reduce any ambiguity in operator mapping.

A further limitation, acknowledged by prior work in similar contexts [Wang *et al*., 2024], is that the derived operators have not yet been subjected to a formal mutation testing assessment in a controlled setting. While the empirical grounding provides strong face validity, since the operators model defect patterns actually observed in real repositories, a rigorous evaluation would make it possible to investigate whether tests capable of killing the proposed mutants also detect faults from the analyzed repositories, as recommended by established mutation testing methodology [Papadakis *et al*., 2019]. To partially address this gap, a proof-of-concept tool named *Terramutate* was implemented and applied to a reference Terraform project, offering preliminary evidence that the operators are syntactically well-formed and automatically injectable (Section 4.3). As future work, we plan to extend this assessment through a systematic evaluation of Terramutate at scale.

6 Conclusions

This research addresses critical gaps in IaC quality assurance by establishing an evidence-based foundation for Terraform mutation testing. To move beyond mere diagnostics, this study analyzed 6,749 corrective commits from 54 public repositories. Leveraging `Sentence-RoBERTa`, UMAP, and HDBSCAN, the investigation formalized recurring defect patterns into a taxonomy of mutation operators targeting versioning constraints (VCR, VLI), static-analysis suppressions (SAR), and general maintenance activities (IPR, PDR, MDR, VDR, PAR, RCR).

The primary contribution is this empirically derived operator catalog. The taxonomy aligns with prior research: parameter and variable operators (**PAR**, **RCR**, **VDR**) provide testable instances of broad "assignment" defects [Rahman *et al*., 2018], while security-misconfiguration operators (**RCR_SG_OPEN**) corroborate known vulnerabilities [Hassan and Rahman, 2022] and complement automated remediation approaches [Shimizu *et al*., 2024]. The formalization of static-analysis suppression faults (**SAR**) introduces a specific defect class not previously emphasized. Practically, DevOps and security teams can integrate this catalog into CI/CD pipelines to systematically expose test-suite weaknesses, guiding the creation of resilient infrastructure. All artifacts are fully reproducible at `https://github.com/Asunnya/terraform-miner`.

These findings remain bounded by their context: the analysis relies exclusively on public repositories, which may not completely reflect proprietary enterprise practices, and 8% of data points were classified as algorithmic noise, meaning some infrequent fault patterns may be absent. As discussed in Section 5, a key limitation is that the derived operators have not yet been subjected to formal mutation testing assessment; demonstrating that tests killing the proposed mutants also detect repository-observed faults is the primary next step. A dedicated study focusing on the Terramutate proof-of-concept implementation and its large-scale empirical evaluation is currently underway. Nevertheless, this baseline taxonomy enables multiple avenues for future research. Key priorities include conducting objective fault-detection efficacy studies against existing test suites; expanding practitioner surveys to capture faults practically invisible in commit histories; enhancing classification via semantic patch analysis of Terraform diffs; replicating the pipeline for frameworks like Ansible or

Pulumi; and utilizing large language models to autonomously discover novel defect semantics.

By shifting from reactive diagnosis to proactive evaluation, this research offers a pathway to mitigate the severe operational and financial risks of configuration failures, establishing an evidence-based standard for IaC quality assurance.

Declarations

Authors' Contributions

I.P.R. contributed to the conceptualization of this study, designed the research methodology, performed all data collection and curation, conducted the formal analysis and clustering pipeline, developed the Terramutate proof-of-concept software, created the visualizations, and is the primary author of this manuscript (writing – original draft). V.O.N. contributed to the conceptualization and research direction, provided supervision throughout all phases of the study, and reviewed and edited the manuscript (writing – review & editing). Both authors read and approved the final manuscript.

Competing interests

The authors declare that they have no competing interests.

Availability of data and materials

The datasets and software generated and analyzed during the current study are openly available. The corrective commit corpus and the multi-stage mining pipeline are archived at `https://github.com/Asunnya/terraform-miner`. The Terramutate proof-of-concept mutation testing tool is available at `https://github.com/Asunnya/Terramutate`.

Further relevant information

This work used generative AI assistance (Gemini 3.1 Pro) solely for structural organization and revision of the manuscript text and to expedite LaTeX formatting. No AI tool was used for data collection, analysis, clustering, or derivation of mutation operators. All AI-generated suggestions were reviewed and verified by the authors, who retain full responsibility for the content.

References

Al-Fraihat, D., Sharrab, Y., Al-Ghuwairi, A.-R., Sbaih, N., and Qahmash, A. (2024). Detecting refactoring type of software commit messages based on ensemble machine learning algorithms. *Scientific Reports*, 14(1):21367.

Amit, I. and Feitelson, D. G. (2021). Corrective commit probability: a measure of the effort invested in bug fixing. *Software Quality Journal*, 29(4):817–861.

Aqua Security (2025). tfsec: static analysis security scanner for terraform code. Software.

Bridgecrew (2025). Checkov: policy-as-code scanner for infrastructure as code. Software.

Campello, R. J. G. B., Moulavi, D., and Sander, J. (2013). Density-based clustering based on hierarchical density estimates. In *Advances in Knowledge Discovery and Data Mining, PAKDD 2013*, volume 7819 of *Lecture Notes in Computer Science*, pages 160–172. Springer Berlin Heidelberg. DOI: 10.1007/978-3-642-37456-2_14.

Carnegie Mellon University (2019). Infrastructure as code: Final report. Technical report, Software Engineering Institute, Carnegie Mellon University. Accessed: February 1, 2025.

Chittibala, D. R. (2023). Infrastructure as Code (IaC) and its role in achieving DevOps goals. *International Journal of Science and Research (IJSR)*, 12(1):1258–1262. DOI: 10.21275/SR24304170702.

Dalla Palma, S., Di Nucci, D., Palomba, F., and Tamburri, D. A. (2022). Within-project defect prediction of infrastructure-as-code using product and process metrics. *IEEE Transactions on Software Engineering*, 48(6):2086–2104. DOI: 10.1109/TSE.2021.3051492.

Hassan and Rahman (2022). As code testing: Characterizing test quality in open source ansible development. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 208–219. DOI: 10.1109/ICST53961.2022.00031.

Hassan, A. E. (2008). The road ahead for mining software repositories. In *2008 frontiers of software maintenance*, pages 48–57. IEEE.

Howard, M. (2022). Terraform: Automating infrastructure as a service. arXiv preprint arXiv:2205.10676.

IEEE Computer Society (2010). Ieee standard classification for software anomalies. Revision of IEEE Std 1044-1993. Approved 9 November 2009; Published 7 January 2010.

Jia, Y. and Harman, M. (2011). An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678. DOI: 10.1109/TSE.2010.62.

Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D. M., and Damian, D. (2014). The promises and perils of mining github. In *Proceedings of the 11th working conference on mining software repositories*, pages 92–101.

Lee, J. Y. D. and Chieu, H. L. (2021). Co-training for commit classification. In Xu, W., Ritter, A., Baldwin, T., and Rahimi, A., editors, *Proceedings of the Seventh Workshop on Noisy User-generated Text (W-NUT 2021)*, pages 389–395, Online. Association for Computational Linguistics. DOI: 10.18653/v1/2021.wnut-1.43.

Malzer, C. and Baum, M. (2020). A hybrid approach to hierarchical density-based cluster selection. In *2020 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI)*, page 223–228. IEEE. DOI: 10.1109/mfi49285.2020.9235263.

McInnes, L., Healy, J., and Melville, J. (2018). Umap: Uniform manifold approximation and projection for dimension reduction. *arXiv preprint arXiv:1802.03426*.

Offutt, A. J. and Untch, R. H. (2001). *Mutation Testing for the New Century*. Springer.

Ozdoğan, E., Ceran, O., and Üstündağ, M. (2023). Systematic analysis of infrastructure as code technologies. *Gazi University Journal of Science Part A: Engineering and Innovation*, 10. DOI: 10.54287/gujsa.1373305.

Papadakis, M., Kintis, M., Zhang, J., Jia, Y., Traon, Y. L., and Harman, M. (2019). Section six - mutation testing advances: An analysis and survey. In Memon, A. M., editor, *Advances in Computers*, volume 112 of *Advances in Computers*, pages 275–378. Elsevier. DOI: https://doi.org/10.1016/bs.adcom.2018.03.015.

Rahman, A. (2018). Characteristics of defective infrastructure as code scripts in devops. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ICSE '18, page 476–479, New York,

NY, USA. Association for Computing Machinery. DOI: 10.1145/3183440.3183452.

Rahman, A., Elder, S., Shezan, F. H., Frost, V., Stallings, J., and Williams, L. A. (2018). Categorizing defects in infrastructure as code. *CoRR*, abs/1809.07937.

Rahman, A. and Williams, L. (2019). Source code properties of defective infrastructure as code scripts. *Information and Software Technology*, 112:148–163. DOI: https://doi.org/10.1016/j.infsof.2019.04.013.

Ramler, R., Wetzlmaier, T., and Klammer, C. (2017). An empirical study on the application of mutation testing for a safety-critical industrial software system. In *Proceedings of the Symposium on Applied Computing*, SAC '17, page 1401–1408, New York, NY, USA. Association for Computing Machinery. DOI: 10.1145/3019612.3019830.

Rebai, S., Kessentini, M., Alizadeh, V., Sghaier, O. B., and Kazman, R. (2020). Recommending refactorings via commit message analysis. *Information and Software Technology*, 126:106332. DOI: https://doi.org/10.1016/j.infsof.2020.106332.

Reimers, N. and Gurevych, I. (2019). Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084*.

Shimizu, R. and Kanuka, H. (2020). Test-based least privilege discovery on cloud infrastructure as code. In *2020 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 1–8. IEEE.

Shimizu, R., Nunomura, Y., and Kanuka, H. (2024). Test-suite-guided discovery of least privilege for cloud infrastructure as code. *Automated Software Engineering*, 31(1):25.

Sokolowski, D., Spielmann, D., and Salvaneschi, G. (2024). Automated infrastructure as code program testing. *IEEE Transactions on Software Engineering*, 50(6):1585–1599. DOI: 10.1109/TSE.2024.3393070.

Tong, J., Wang, Z., and Rui, X. (2023). Boosting commit classification with contrastive learning. *arXiv preprint arXiv:2308.08263*.

Wang, B., Chen, M., Lin, Y., Papadakis, M., and Zhang, J. M. (2024). An exploratory study on using large language models for mutation testing.

Yang, Y., Ronchieri, E., and Canaparo, M. (2022). Natural language processing application on commit messages: a case study on hep software. *Applied Sciences*, 12(21):10773.