

TerraRepair: A Tool-Grounded LLM Agent for Infrastructure-As-Code Repair

Minase Mekete Mengistu ✉ 

University of L'Aquila, Italy

Juri Di Rocco ✉ 

University of L'Aquila, Italy

Phuong T. Nguyen ✉ 

University of L'Aquila, Italy

Davide Di Ruscio ✉ 

University of L'Aquila, Italy

Abstract

Background. Infrastructure-as-Code (IaC) scanners detect cloud misconfigurations in Terraform and other IaC languages before deployment, but repairing the flagged configurations remains largely manual. Recent Large Language Model (LLM)-based repair approaches can repair some findings, but may hallucinate unsupported constructs or suppress warnings without fixing the issue.

Aims. We study whether tool grounding can improve LLM-based Terraform repair, and when a finding should be escalated because the required deployment-specific context is not available.

Method. We present **TerraRepair**, a prototype of a tool-grounded LLM agent for Terraform repair with structured escalation. **TerraRepair** retrieves dependency context from Terraform references, consults the installed provider schema, and re-runs the scanner before returning a candidate repair. When the required context is absent, **TerraRepair** escalates instead of fabricating a plausible fix.

Results. We evaluate our tool on two vulnerable-by-design Terraform repositories using two IaC security scanners, Checkov and Trivy, across AWS, Azure, and GCP. On the combined AWS benchmark, **TerraRepair** improves scanner-verified fix rates from 26.6% to 78.4% on Checkov and from 44.8% to 72.4% on Trivy, compared with a controlled one-shot baseline. It also reduces the baseline's 44.8–73.6 percentage point (pp) claimed-vs-verified repair gap to under 5 pp. In a sampled semantic audit covering AWS only, 78.9% of **TerraRepair**'s scanner-verified AWS repairs are labeled as correct under a majority-vote protocol with two LLM judges and one author.

Conclusions. These emerging results show that tool grounding can substantially improve scanner-verified LLM-based IaC repair on the studied benchmarks, while missing deployment-specific context remains the main knowledge boundary for full autonomy.

2012 ACM Subject Classification Software and its engineering → Software maintenance tools; Software and its engineering → Software defect analysis; Security and privacy → Software security engineering

Keywords and phrases Infrastructure as Code, automated repair, large language models, cloud security

Digital Object Identifier 10.4230/LIPIcs.ESEM.2026.68

Category Emerging Results, Vision & Reflection Track Paper

Related Version *Full Version:* <https://arxiv.org/pdf/2607.11390>

Funding This paper has been partially supported by the MOSAICO project (Management, Orchestration and Supervision of AI-agent COmmunities for reliable AI in software engineering) that has received funding from the European Union under the Horizon Research and Innovation Action (Grant Agreement No. 101189664).



© Minase Mekete Mengistu, Juri Di Rocco, Phuong T. Nguyen, and Davide Di Ruscio; licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).

Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;

Article No. 68; pp. 68:1–68:14



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Infrastructure as Code (IaC) is widely used to provision and manage cloud infrastructure through declarative configuration files, such as Terraform modules written in HashiCorp Configuration Language (HCL) [15, 7]. This practice improves repeatability, reviewability, and automation, but it also moves cloud security and reliability risks into code [5]. Misconfigured IaC can expose storage buckets, grant overly broad Identity and Access Management (IAM) permissions, disable encryption, or leave network access unrestricted. Prior work has studied such recurring weaknesses as IaC security smells [20].

IaC security scanners detect many of these issues before deployment by reporting policy violations over Terraform resources [5, 17, 2]. However, detection is only the first step. Developers must still determine how to modify the affected resource, whether related resources must also be changed, and whether the repair is valid for the installed Terraform provider version. Many repairs also require deployment-specific context, such as Key Management Service (KMS) keys, Classless Inter-Domain Routing (CIDR) ranges, logging destinations, certificates, or least-privilege IAM scopes.

LLMs offer a possible way to reduce this manual repair effort, following recent work on LLM-based program and vulnerability repair [18, 11]. In the IaC setting, Low et al. [13] showed that LLMs can repair many Terraform scanner findings, especially when human-provided context is available. However, it has been shown that LLM-based IaC repair can hallucinate Terraform constructs, introduce validation errors, or clear scanner warnings without fixing the underlying security issue [13]. This suggests that the main challenge is not only generating a plausible edit, but grounding the repair in the codebase, provider schema, scanner feedback, and available deployment context.

IaC repair differs from conventional program repair because IaC programs specify desired infrastructure state rather than local program behavior [22]. A syntactically valid HCL edit may still be invalid for the installed provider schema, undeployable in the target cloud environment, or unsafe for the intended deployment. Prior work on semantic checks for cloud IaC shows that IaC programs that pass compilation can still fail at deployment time because syntax alone does not capture deployment-relevant requirements [19]. For this reason, scanner success is an incomplete repair oracle. An autonomous IaC repair system needs a way to decide when a repair is sufficiently grounded and when the finding should be escalated.

To bridge such a gap, we present **TerraRepair**, a tool-grounded LLM agent for bounded Terraform repair. Given a scanner finding and the affected Terraform block, **TerraRepair** treats repair as a bounded tool-using process. The agent can query a dependency graph for cross-resource context, inspect the installed Terraform provider schema, and re-run the scanner on candidate repairs. If the required information is absent from the codebase, **TerraRepair** emits a structured escalation instead of fabricating a plausible repair.

To evaluate the effectiveness of tool grounding, its behavior across cloud providers, the contribution of individual components, and the limits of autonomous repair, we investigate the following research questions:

- **RQ₁**: How does **TerraRepair** compare with direct one-shot LLM prompting?
- **RQ₂**: How does **TerraRepair** vary across cloud providers?
- **RQ₃**: What is the individual contribution of each architectural component?
- **RQ₄**: What kinds of findings cannot be repaired autonomously?

To answer these questions, we evaluate **TerraRepair** on two vulnerable-by-design Terraform repositories used in prior IaC repair research, TerraGoat [4] and KaiMonkey [25]. The evaluation uses Checkov and Trivy across AWS, Azure, and GCP configurations. We

also construct a controlled one-shot baseline inspired by Low et al.’s autonomous first-pass prompt [13]. This baseline is not a reproduction of their 2024 environment. It isolates direct prompting under the same model, scanner versions, finding population, patching logic, and scoring procedure used for **TerraRepair**. On the combined AWS benchmark, **TerraRepair** improves scanner-verified fix rates, outperforming the considered baseline for both Checkov and Trivy. The ablation and escalation analyses show that provider-schema grounding and dependency retrieval are the main contributors to repair effectiveness, while missing deployment-specific context remains the main knowledge boundary for full autonomy.

As an emerging-results study, this paper makes the following contributions:

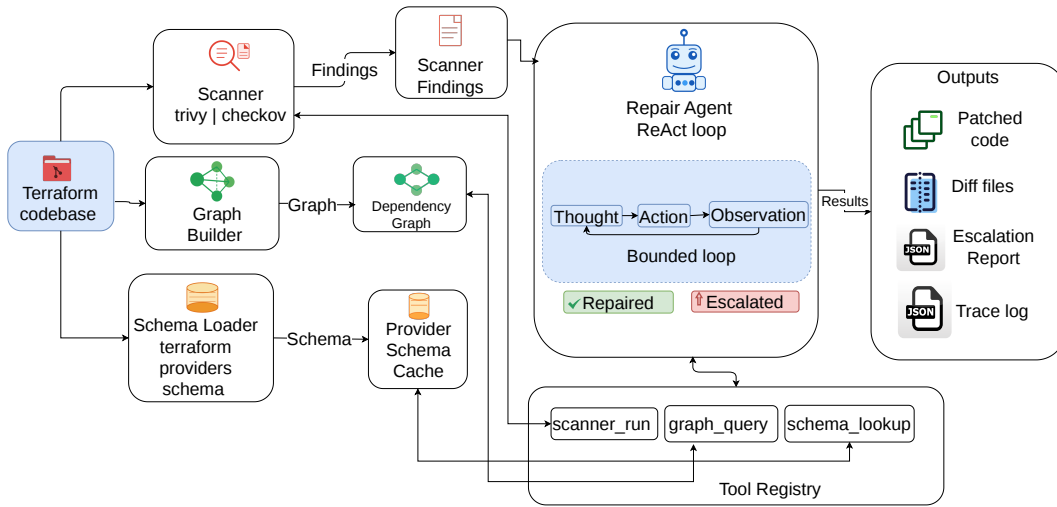
- We formulate Terraform security repair as a bounded tool-grounded process that returns either a scanner-verified candidate repair or a structured escalation.
- The introduction of the claimed-vs-verified repair gap as an operational measure of unsupported repair claims, characterizing a key failure mode of direct LLM repair.
- The **TerraRepair** tool as an empirical prototype that combines dependency-graph retrieval, provider-schema lookup, in-loop scanner verification, and escalation. To the best of our knowledge, this is the first tool of this type to tackle the issue.
- An initial empirical evaluation on vulnerable-by-design Terraform repositories across AWS, Azure, and GCP, comparing **TerraRepair** with a controlled one-shot LLM baseline.
- A replication package with the developed code and curated data has been published to foster open science [14].

2 Related Work

IaC security scanners detect policy violations before deployment, but most prior IaC analysis work focuses on detection and verification rather than repair. Rahman et al. [20] identify recurring IaC security smells, including hard-coded secrets, empty passwords, and unrestricted network access. Chiari et al. [5] survey static analysis for IaC, covering smell detection, defect prediction, and verification. InfraFix [22] addresses IaC repair by inferring a desired system state, repairing a normalized intermediate representation, and propagating changes back to the original script. **TerraRepair** instead repairs Terraform security findings reported by IaC scanners, using installed provider schemas, dependency references, scanner feedback, and escalation when required deployment context is absent.

TerraRepair also builds on automated program repair (APR) and recent LLM-based repair. APR generates patches with respect to evidence such as a test suite, crash, compiler error, or static-analysis warning [12], but a patch may overfit the repair oracle while failing to preserve intended behavior [24]. This issue is especially relevant for security repair because clearing a warning does not imply that the underlying vulnerability has been fixed. Recent LLM repair work has explored zero-shot vulnerability repair, static-analysis-guided and retrieval-augmented repair, tool-using repair agents, and repository-history-aware repair [18, 11, 3, 23]. **TerraRepair** follows this broader direction, but the artifact and oracle are different. It repairs Terraform configurations against scanner findings, provider schemas, dependency context, and explicit escalation conditions.

Several studies apply LLMs to IaC and configuration-security tasks. TerraFormer [10] studies Terraform generation and mutation from natural-language prompts using verifier-guided training over syntax, deployability, and policy compliance. LLMSecConfig [30] repairs Kubernetes container misconfigurations using static analysis, retrieval, prompting, and validation. Reyes et al. [21] construct a Terraform remediation dataset from Trivy findings and evaluate fine-tuning and in-context learning for open-source LLMs. Toprani and Madiseti [26]



■ **Figure 1** TerraRepair architecture and repair workflow.

propose a multi-agent RAG workflow for detecting CloudFormation vulnerabilities and producing remediation guidance. NSync [28] addresses Terraform reconciliation rather than security repair by using cloud API traces to infer out-of-band infrastructure changes and synthesize Terraform updates. Together, these works show that LLMs can support IaC generation, configuration repair, vulnerability remediation, and cloud-state reconciliation.

Low et al. [13] are the closest prior work. They repair Terraform scanner findings by passing the flagged block and scanner output to an LLM, with a human-provided second pass for remaining findings. Their results show that LLMs can remove many scanner warnings, but may also hallucinate Terraform constructs, introduce validation errors, or clear warnings without fixing the underlying issue. **TerraRepair** targets this failure mode through provider-schema lookup, dependency retrieval, scanner feedback, and structured escalation.

3 TerraRepair: Proposed Approach

TerraRepair repairs Terraform scanner findings as a bounded tool-using process. Given a scanner finding, the flagged resource block, and the corresponding cause lines, **TerraRepair** returns either a scanner-verified candidate repair or a structured escalation. The escalation is used when the required information is not available from the codebase, provider schema, or scanner feedback, or when the bounded repair process does not converge.

The proposed approach is shown in Fig. 1. **TerraRepair** first runs a security scanner to obtain misconfiguration findings. It builds a dependency graph from HCL references in the codebase and loads the relevant Terraform provider schemas using the Terraform CLI provider-schema interface [8]. Each finding is then given to a single ReAct-style repair agent [29]. The agent receives the rule identifier, severity (if available), resolution hint, resource identifier, cause lines, and the complete original HCL block. We bound each repair attempt at ten agent steps. This fixed budget gives the agent room to query dependency context, inspect the provider schema, propose a patch, run the scanner, and revise when needed, while keeping repair attempts finite and comparable across findings. The agent has access to three tools. `GRAPH_QUERY` retrieves cross-resource information from the dependency graph, e.g., KMS key ARNs, network identifiers, and security group references. `SCHEMA_LOOKUP` retrieves the installed provider schema for the target resource type, so

■ **Table 1** Benchmark datasets and baseline finding counts.

Dataset	Provider	Checkov	Trivy
TerraGoat	AWS	158	113
TerraGoat	Azure	124	90
TerraGoat	GCP	55	49
KaiMonkey	AWS	69	55
Combined AWS (primary)	–	227	168
Grand total	–	406	307

that the agent can avoid unsupported attributes, invalid nested blocks, and assignments to provider-computed fields. `SCANNER_RUN` re-runs the scanner on the candidate repair and returns the result as feedback. If the scanner still reports the original finding, the agent may revise the patch. If the scanner no longer reports the original finding, the agent can terminate and emit the candidate repair.

The repair prompt defines the agent role, available tools, stopping criteria, and output schema. It specifies three constraints. First, the agent must consult the provider schema before using unfamiliar attributes or nested blocks. Second, the agent is instructed to obtain scanner verification before returning a candidate repair. Third, it must output exactly one repaired resource block. If a correct repair requires deployment-specific information that is absent from the codebase or provider schema, the agent is instructed to emit a structured escalation instead of fabricating a plausible repair. The full prompt templates, tool interfaces, and output schemas are included in the replication package [14].

4 Evaluation Methodology

The evaluation is organized around four research questions motivated in Section 1.

We use two vulnerable-by-design Terraform repositories used in prior IaC repair research, namely TerraGoat (commit `729f8da`) [4] and KaiMonkey (commit `3feb0bf`) [25]. TerraGoat contains AWS, Azure, and GCP configurations. KaiMonkey contains four AWS slices covering compute, network, storage, and a cross-module Server-Side Request Forgery (SSRF) scenario. These seven benchmark slices are evaluated with Checkov and Trivy, yielding 14 dataset–scanner configurations. The primary comparison dataset is the combined AWS subset, which contains 227 Checkov findings and 168 Trivy findings, as shown in Table 1. We use AWS as the primary comparison because Low et al.’s Terraform repair evaluation used AWS. Azure and GCP assess cross-provider behavior.

We use Checkov 3.2.510, Trivy 0.69.3, `gpt-4o-mini-2024-07-18` at temperature 0.0 [16], Terraform CLI 0.14.11, and `python-hcl2` 7.3.1 [1]. API inference may vary at temperature 0.0, so we repeat each **TerraRepair** configuration three times. The AWS configurations resolve the AWS provider at versions 4.67.0 for TerraGoat and 3.76.1 for KaiMonkey. The non-AWS configurations resolve the AzureRM provider at version 4.68.0 and the Google provider at version 7.27.0. Low et al. [13] used earlier scanner versions and also included Terrascan, which is now archived. Therefore, their published numbers are used only as historical context. We construct a within-study baseline based on Low et al.’s autonomous first-pass prompt [13]. The baseline uses the prompt from their Figure 2a verbatim and performs one LLM call per finding, with no dependency-graph retrieval, provider-schema lookup, or in-loop scanner verification. Both systems use the same model, scanner versions, finding population, file-patching logic, and post-run rescanning procedure. Within each run, a parity check confirms that both pipelines receive bit-identical scanner findings before repair.

The primary metric is the *scanner-verified fix rate*, the proportion of originally flagged findings that are no longer flagged when the same scanner is re-run on the repaired output. We also report the *claimed-vs-verified repair gap*, defined as the claimed repair rate minus the scanner-verified fix rate. For the baseline, a claimed repair is an LLM response with `is_fixed=true`. For TerraRepair, it is a final repaired status that typically follows in-loop `SCANNER_RUN` verification. Both systems are evaluated with the same final full-codebase rescan.

All TerraRepair configurations are executed three times. The controlled baseline is executed three times on the AWS configurations used for RQ₁. Each run first pools verified repairs across the relevant datasets, and the reported value is the mean and standard deviation of these pooled per-run rates.

Within each run, both systems are evaluated on the same initial finding population and repository commit, yielding paired binary outcomes. We apply the two-sided exact McNemar test per run on the discordant counts n_{10} (TerraRepair repaired, baseline did not) and n_{01} (the reverse), and report the paired risk difference $RD = (n_{10} - n_{01})/N$, signed in TerraRepair’s favour. Although each attempt targets one finding, findings in the same block may share context or be affected by one block-level edit. We therefore estimate the 95% CI for the three-run mean effect using a cluster bootstrap ($B = 10,000$, seed 20260805) that resamples complete blocks and averages run-level RDs. We report one McNemar result per run rather than pooling the $3N$ repeated observations.

We assess Terraform validity using differential `terraform validate`, counting only newly introduced validation errors. We assess semantic correctness using a sampled jury-of-judges audit, informed by LLM-as-a-Judge work in software engineering [9]. From Run 3 AWS, whose fix rates are closest to the three-run mean, we sample 171 of 303 scanner-verified repairs using Cochran’s formula with finite population correction [6]. The sample is allocated proportionally across dataset–scanner strata, and Wilson score intervals are used for the 95% confidence interval [27].

Each sampled repair is assessed by the first author and two LLM judges, Claude Sonnet 4.5 (`claude-sonnet-4-5-20250929`) and GPT-5.4 (`gpt-5.4-2026-03-05`). The first author inspects the original and repaired code with repository context, while the LLM judges use a conservative judging prompt that defaults to `WRONG` when the repair is ambiguous. Final labels are determined by majority vote. Since two raters are LLM judges and the human rater is an author, we treat this as a semantic audit rather than ground-truth correctness.

For RQ₃, we conduct a three-run leave-one-out ablation on TerraGoat AWS with Checkov ($n = 158$), removing `GRAPH_QUERY`, `SCHEMA_LOOKUP`, or `SCANNER_RUN`. We use the same paired tests and resource-block cluster bootstrap, with negative RDs indicating lower performance than the full system. For RQ₄, we classify structured escalations into missing external context, max-step termination, missing schema support, and unresolved Terraform variable or local references.

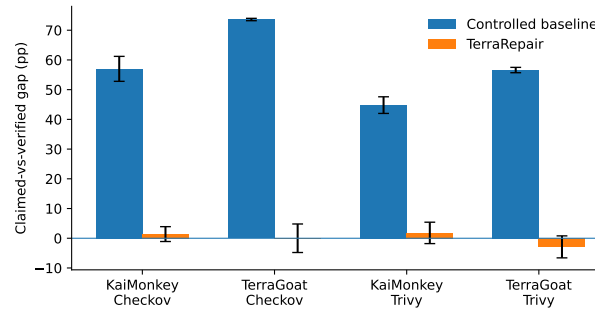
5 Experimental Results

5.1 RQ₁: How does TerraRepair compare with direct one-shot LLM prompting?

The controlled one-shot baseline described in Section 4 reproduces Low et al.’s autonomous first-pass prompt inside our repair harness. Table 2 reports the results on the combined AWS benchmark. The three Low et al. rows in Table 2 are historical context. They used different scanner versions, a different model, and a different finding population, and cannot be compared

■ **Table 2** Scanner-verified fix rates on the combined AWS benchmark. Mean $\pm$ std over three runs. Low et al. rows are historical context.

System	Approach	Checkov	Trivy
Low et al. GPT-3.5 pass 1	autonomous, 2024 scanners	9.0%	17.1%
Low et al. GPT-4 pass 1	autonomous, 2024 scanners	27.4%	44.2%
Low et al. GPT-4 pass 2	human-assisted, 2024 scanners	87.4%	67.3%
Controlled baseline	one-shot, gpt-4o-mini	26.6% $\pm$ 1.4 pp	44.8% $\pm$ 1.4 pp
TerraRepair	agent+tools, gpt-4o-mini	78.4% $\pm$ 0.8 pp	72.4% $\pm$ 4.0 pp
Gain over controlled baseline –		+51.8 pp	+27.6 pp



■ **Figure 2** Claimed-vs-verified repair gap on AWS configurations. Values closer to zero indicate better calibration. Negative values indicate finding-level underclaiming.

directly with the current experiment. The primary comparison is between **TerraRepair** and the controlled baseline, which share the same model (**gpt-4o-mini-2024-07-18**), scanner versions, finding population, file-patching logic, and scoring procedure. The controlled baseline achieves a scanner-verified fix rate of **26.6% $\pm$ 1.4 pp** on Checkov and **44.8% $\pm$ 1.4 pp** on Trivy. **TerraRepair** achieves **78.4% $\pm$ 0.8 pp** and **72.4% $\pm$ 4.0 pp**, respectively. The mean paired risk difference is **+51.8 pp** on Checkov and **+27.6 pp** on Trivy. The across-run cluster-bootstrap 95 % confidence intervals are $[+34.3, +63.1]$ pp for Checkov and $[+18.4, +38.3]$ pp for Trivy. The exact McNemar tests are significant in all three runs ($p < 0.001$ each).

The baseline also exposes a large gap between claimed and scanner-verified repairs. For the baseline, a claimed repair is an LLM response with `is_fixed=true`. For **TerraRepair**, a claimed repair is a final repaired status that typically follows in-loop `SCANNER_RUN` verification. As shown in Fig. 2, the one-shot baseline has a claimed-vs-verified gap of 44.8–73.6 pp across the AWS configurations. This means that the baseline frequently reports a repair as successful even though the same scanner still reports the original finding after patching and rescanning. **TerraRepair** reduces this gap to between -2.9 and $+1.8$ pp. A small negative gap means that the final scanner run cleared slightly more findings than **TerraRepair** explicitly claimed to repair. This can happen because Terraform repairs are applied at the block level, while claims are recorded per scanner finding. If several findings refer to the same block, one repair may clear more than one finding. Thus, we interpret small negative gaps as finding-level underclaiming, not as evidence of additional semantic correctness. Positive gaps can occur when a repair passes **TerraRepair**’s in-loop scanner check but the finding reappears during the final full-codebase rescan. Scanner verification does not prove semantic correctness, but it reduces this specific unsupported-claim failure mode.

■ **Table 3** Multi-cloud repair effectiveness and validity. Values are mean $\pm$ std across three runs. Validation errors are combined totals across both scanners and three runs.

Dataset	Prov.	Checkov	Trivy	Val.	Main validity issue
TerraGoat	AWS	83.1% $\pm$ 2.6	70.5% $\pm$ 2.8	0	none observed
TerraGoat	Azure	74.7% $\pm$ 1.2	72.2% $\pm$ 2.2	11	missing required attrs.
TerraGoat	GCP	71.5% $\pm$ 6.4	71.4% $\pm$ 2.0	15	invalid provider fields
KaiMonkey	AWS	67.7% $\pm$ 6.7	76.4% $\pm$ 7.3	0	none observed

Repair validity is assessed at two additional levels. First, on the AWS benchmark and across all three runs, **TerraRepair** introduces *zero* new `terraform validate` errors. This provides evidence that the repaired configurations remain syntactically valid and acceptable to Terraform validation on the primary benchmark, although `terraform validate` is not a complete semantic oracle. Second, in the sampled semantic audit, 135 of 171 scanner-verified AWS repairs are judged semantically correct by majority vote, giving an estimate of **78.9%** (95% CI [72.2%, 84.4%]). Inter-rater agreement is moderate (Fleiss’ $\kappa=0.54$), with 130/171 findings (76.0%) receiving unanimous verdicts. The judging prompt and rater-level labels are included in the replication package [14].

Answer to RQ₁. **TerraRepair** improves scanner-verified fix rate by +51.8 pp on Checkov and +27.6 pp on Trivy, reduces the claimed-vs-verified gap to within -2.9 to $+1.8$ pp, and has 78.9% of sampled scanner-verified AWS repairs judged correct.

5.2 RQ₂: How does TerraRepair vary across cloud providers?

Table 3 reports scanner-verified fix rates and validation errors across the four provider/dataset configurations. **TerraRepair** achieves broadly similar scanner-verified fix rates across the evaluated settings, with 67.7%–83.1% on Checkov and 70.5%–76.4% on Trivy. There is no clear drop that can be explained by provider alone. The observed variation appears to be influenced by dataset composition, especially KaiMonkey’s concentration of wildcard IAM policy repairs and TerraGoat GCP’s smaller finding population.

Repair validity, however, shows a cross-provider limitation that scanner-verified fix rates alone hide. Across the three runs, all 26 newly introduced `terraform validate` errors occur in Azure and GCP repairs. These errors cluster around provider-specific schema failures. Missing required attributes and unsupported or removed attributes account for 19/26 errors (73.1%). The remaining errors involve assignments to provider-computed fields, invalid enumeration values, and one block serialization failure. For example, for CKV_AZURE_7,¹ the agent generated `network_profile { network_policy = "calico" }`, which satisfies Checkov’s network-policy check but omits the required `network_plugin` argument. Similarly, several GCP repairs used unsupported, removed, provider-computed, or invalid-enum attributes that were sufficient to address the scanner rule locally but invalid for the installed provider version. These cases show that provider-schema validity remains a separate repair requirement even when scanner-level behavior appears to generalize.

¹ CKV_AZURE_7 checks that an Azure Kubernetes Service cluster has a network policy configured.

■ **Table 4** Leave-one-out ablation on TerraGoat AWS with Checkov (mean $\pm$ std over three runs).

Configuration	Verified	Δ	Agent	Escalations
Full system	83.1% $\pm$ 2.6	–	83.1% $\pm$ 2.2	26.7 $\pm$ 3.5
No GRAPH_QUERY	67.5% $\pm$ 4.5	–15.6 pp	68.1% $\pm$ 3.0	50.3 $\pm$ 4.7
No SCHEMA_LOOKUP	61.4% $\pm$ 2.5	–21.7 pp	59.1% $\pm$ 2.4	64.7 $\pm$ 3.8
No SCANNER_RUN	82.1% $\pm$ 2.2	–1.1 pp	81.4% $\pm$ 2.9	29.3 $\pm$ 4.5

Answer to RQ₂. TerraRepair varies mainly in repair validity rather than scanner-verified fix rate. Scanner-verified fix rates remain broadly similar across AWS, Azure, and GCP, but all new validation errors occur in Azure and GCP repairs. Thus, scanner-level repair transfers better than schema-correct repair.

5.3 RQ₃: What Is the Individual Contribution of Each Architectural Component?

Table 4 reports the leave-one-out ablation study on TerraGoat AWS with Checkov. Here, *Verified* denotes the post-run scanner-verified fix rate, and *Agent* denotes the agent’s in-loop repaired status. The largest measurable contributions come from SCHEMA_LOOKUP and GRAPH_QUERY. Removing SCHEMA_LOOKUP reduces the scanner-verified fix rate by 21.7 pp (cluster-bootstrap 95 % CI [–30.2, –12.2] pp, $p < 0.001$ in all three runs) and more than doubles the mean number of escalations. This indicates that training-time model knowledge is insufficient for reliable Terraform repair. Removing GRAPH_QUERY reduces the scanner-verified fix rate by 15.6 pp (cluster-bootstrap 95 % CI [–24.2, –8.6] pp, $p < 0.001$ in all three runs) and similarly increases escalations.

Removing SCANNER_RUN reduces the mean verified fix rate by only 1.1 pp (cluster-bootstrap 95 % CI [–5.2, +3.5] pp, with McNemar tests nonsignificant in all three runs). This small delta reflects the fact that SCANNER_RUN is primarily a verification and calibration tool rather than a source of new repair context. It does not supply missing resource identifiers or schema information, and therefore contributes less to raw repair capability than GRAPH_QUERY or SCHEMA_LOOKUP. However, the controlled-baseline comparison in RQ₁ shows that scanner feedback is still important operationally as it reduces unsupported repair claims by preventing repairs that still trigger the original scanner finding from being reported as successful.

Answer to RQ₃. TerraRepair’s scanner-verified fix rate appears to be driven primarily by live provider-schema grounding and graph-based dependency resolution, while scanner feedback mainly supports verified termination and calibrated success reporting.

5.4 RQ₄: What Kinds of Findings Cannot Be Repaired Autonomously?

Across all 14 dataset–scanner configurations and three runs, TerraRepair escalates a mean of 181.3 findings per run. This corresponds to a stable escalation rate of 25.4% (std = 1.3 pp). Across the three runs, 544 findings are escalated in total. Table 5 summarizes the escalation taxonomy.

The category labels denote missing external context (*context_ext.*), max-step termination (*max_step*), missing schema support (*schema_unk.*), and unresolved Terraform variable or local references (*context_var.*). The dominant category is missing external context

■ **Table 5** Escalation categories across all runs (mean escalation rate: 25.4%).

Category	R1	R2	R3	Mean	%
context_ext.	150	149	151	150.0	82.7%
max_step	31	21	16	22.7	12.5%
schema_unk.	8	6	7	7.0	3.9%
context_var.	3	1	1	1.7	0.9%
Total	192	177	175	181.3	100%

(82.7%). This includes absent cross-resource references and missing deployment-specific values, such as KMS ARNs, certificates, logging targets, secrets, disk-encryption sets, logging buckets, and network identifiers that cannot be inferred from the local codebase. Missing schema support and unresolved Terraform references account for another 4.8% of escalations.

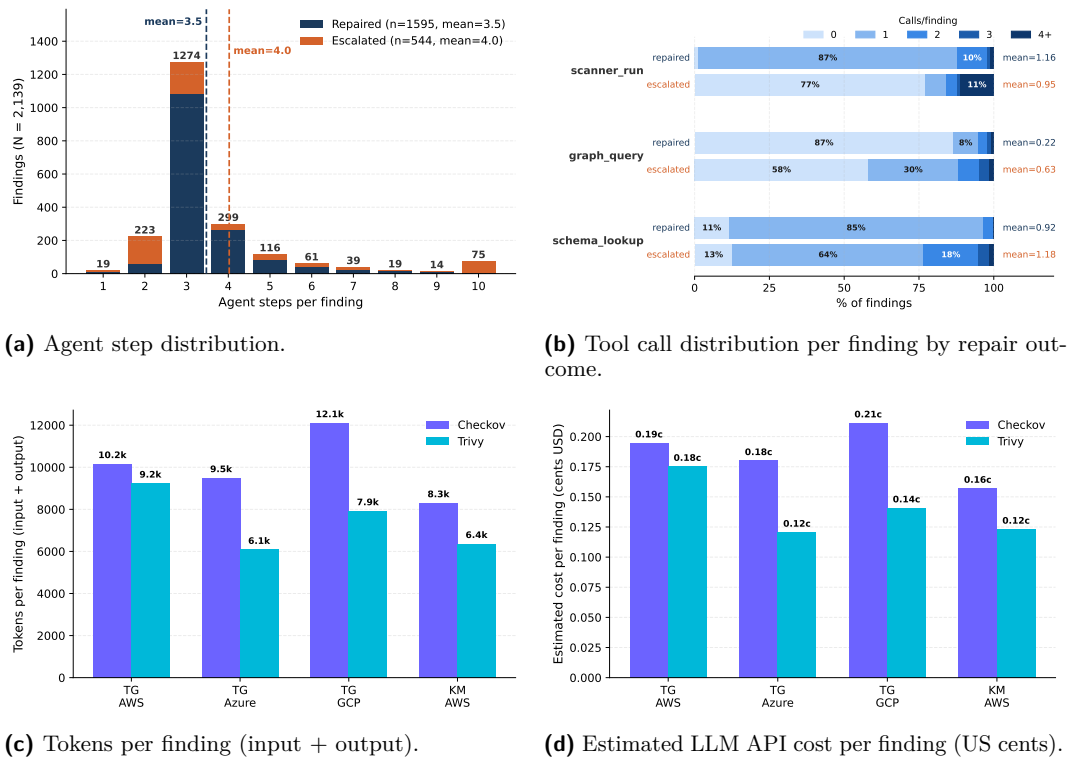
A further 12.5% of escalations are max-step terminations. These should not be interpreted as graceful agent escalations. The repair process reached the 10-step budget while the agent was still trying to converge. Heredoc-containing blocks such as IAM policies and `user_data` scripts caused `SCANNER_RUN` parse failures that the agent treated as repair syntax errors (20 cases). Scanner feedback did not identify the unsatisfied condition (20 cases). `SCHEMA_LOOKUP` lacked enough nested-block detail (14 cases). The LLM produced invalid tool-call formatting before any repair was attempted (7 cases). Finally, the agent exhausted steps on tool lookups or emitted `FINISH` as an invalid tool action (7 cases). These causes indicate repair-loop limitations rather than missing deployment intent.

Answer to RQ₄. Most escalations are caused by unavailable or unresolved information, with missing external context alone accounting for 82.7% of escalations. A smaller set of escalations comes from max-step termination, where the bounded repair loop does not converge because of representation, scanner-feedback, or schema-detail limitations.

5.5 Operational Metrics

Fig. 3 reports TerraRepair’s operational metrics across all 14 dataset–scanner configurations and three runs ($N = 2,139$ finding-level repair attempts). Repaired findings require a mean of 3.5 agent steps, escalated findings a mean of 4.0, and only 0.4% of repaired findings (6 of 1,595) reach the 10-step budget (Fig. 3a). Fig. 3b breaks tool calls down by repair outcome. `SCHEMA_LOOKUP` is invoked about once per finding regardless of outcome (mean 0.92 repaired, 1.18 escalated). `SCANNER_RUN` is called at least once for 98.8% of repaired findings (mean 1.16) but only 22.8% of escalated ones (mean 0.95). `GRAPH_QUERY` shows the opposite pattern, with calls on 13.5% of repaired but 41.9% of escalated findings (means 0.22 and 0.63). Token consumption ranges from 6.1k to 12.1k per finding (Fig. 3c), an estimated 0.12–0.21 US cents with the `gpt-4o-mini-2024-07-18` API at the time of the experiment (Fig. 3d).

These figures quantify the RQ₁ effort gap: the one-shot baseline issues a single LLM call per finding, whereas TerraRepair spends a mean of 3.5 agent steps and several thousand tokens, for scanner-verified gains obtained under a quarter of a US cent per finding. Read against the RQ₃ ablation, `SCHEMA_LOOKUP` is invoked for 88.5% of repaired findings and produces the largest ablation delta (−21.7 pp when removed), whereas `GRAPH_QUERY` is invoked for only 13.5% yet costs 15.6 pp when removed. Cross-resource context is thus needed for a minority of findings, but decisive when required.



■ **Figure 3** Agent operational metrics across all 14 dataset-scanner configurations and three runs (N = 2,139 finding-level repair attempts).

6 Discussion and Implications

The results suggest three design implications for LLM-based IaC repair. First, repair systems should distinguish generated, scanner-verified, and semantically reviewed outcomes. In-loop scanner feedback can prevent unsupported success claims, but scanner success should remain an intermediate validation signal rather than a complete correctness oracle.

Second, live provider schemas and repository dependency context should be treated as first-class repair inputs rather than relying only on training-time model knowledge. Access to the installed schema helps the agent select valid attributes and nested blocks, while dependency retrieval supplies values that are absent from the flagged resource.

Third, production deployment requires access to organization-specific context and a clear escalation path. Approved KMS keys, logging destinations, certificates, and IAM constraints are often unavailable from the local Terraform code. Integrating these knowledge sources may expand the set of findings that can be repaired autonomously, while structured escalation remains necessary when the required deployment context is unavailable.

7 Threats to Validity

Construct validity. Scanner-verified fix rate measures whether the original scanner finding disappears after repair, but it is not a complete measure of repair correctness. A scanner-verified repair may still be undeployable, invalid for the intended cloud environment, or inconsistent with the security policy. We therefore also report differential `terraform validate` results and a sampled semantic audit. The claimed-vs-verified repair gap is a reporting-calibration metric, not a correctness metric.

Internal validity. The semantic audit uses one author and two LLM judges, which may introduce author bias and model-judging bias. We mitigate this threat by using a structured judging prompt, reporting inter-rater agreement, and using majority vote. However, the audit should still be interpreted as an estimate under the jury protocol rather than as ground-truth repair correctness. LLM inference may also vary despite fixed settings; we therefore execute each configuration three times.

External validity. The evaluation uses TerraGoat and KaiMonkey, which are public and reproducible but vulnerable-by-design repositories. Production IaC may include private modules, custom providers, remote state, provider aliases, dynamic blocks, workspace-specific variables, and organization-specific deployment conventions. The experiments also use one repair model, `gpt-4o-mini-2024-07-18`, at temperature 0.0. Therefore, the fix rates, claim gaps, and ablation ordering should not be interpreted as model-independent properties of LLM-based IaC repair.

Conclusion validity. The primary RQ₁ comparison is a controlled within-study comparison, but it is an architecture-level comparison rather than an ablation of individual mechanisms. TerraRepair differs from the one-shot baseline in tool grounding, chain repair within blocks, in-loop scanner verification, and agent-oriented prompting. The leave-one-out ablation isolates the three tools within TerraRepair, but does not separately isolate prompting and repair-ordering differences. Only three runs were feasible, so we report McNemar tests per run and resource-block cluster-bootstrap intervals rather than pooling the $3N$ repeated observations. The McNemar tests do not adjust for within-block dependence, which is addressed by the cluster-bootstrap intervals.

8 Conclusion and Future Work

This paper presented TerraRepair, a tool-grounded LLM agent for bounded Terraform repair. TerraRepair retrieves dependency context, inspects the installed provider schema, and re-runs the scanner before returning a candidate repair. When required information is absent, it emits a structured escalation instead of fabricating a plausible fix.

In a controlled comparison with a one-shot baseline using the same model, scanner versions, finding population, patching logic, and scoring procedure, TerraRepair improves scanner-verified fix rate on Checkov and on Trivy on the combined AWS benchmark. It also reduces the baseline’s repair gap. The ablation, audit, and escalation analyses suggest that provider-schema grounding and dependency retrieval are important contributors to repair effectiveness, while missing deployment-specific context appears to be a major knowledge boundary for autonomous repair.

The long-term goal is to extend TerraRepair to production-like Terraform repositories. Toward this goal, future work will address the observed failure modes by improving schema detail for nested HCL blocks and tool interfaces for heredoc-containing repairs. Future work will also integrate organization-specific knowledge sources, such as approved KMS keys, logging destinations, certificate inventories, and IAM policy templates, to support context-aware repair. Finally, semantic correctness should be assessed through independent review by IaC and cloud-security practitioners.

Data Availability Statement

The replication package supporting this study is available in a public GitHub repository [14]. It includes the **TerraRepair** source code, evaluation scripts, semantic-audit artifacts, escalation labels, and instructions for reproducing the results. The benchmark datasets are available from the TerraGoat and KaiMonkey repositories at the commits listed in Section 4.

References

- 1 Amplify Education. `python-hcl2`: A parser for hcl2 written in python, 2026. Accessed: 2026-04-28. URL: <https://github.com/amplify-education/python-hcl2>.
- 2 Aqua Security. Trivy, 2026. Accessed: 2026-04-28. URL: <https://trivy.dev/>.
- 3 Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. RepairAgent: An autonomous, llm-based agent for program repair. In *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE/ACM, 2025. doi:10.1109/ICSE55347.2025.00157.
- 4 Bridgecrew. TerraGoat: Vulnerable Terraform Infrastructure, 2026. Accessed: 2026-04-28. URL: <https://github.com/bridgecrewio/terragoat>.
- 5 Michele Chiari, Michele De Pascalis, and Matteo Pradella. Static analysis of infrastructure as code: A survey. In *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*, pages 218–225. IEEE, 2022. doi:10.1109/ICSA-C54293.2022.00049.
- 6 William G. Cochran. *Sampling Techniques*. John Wiley & Sons, New York, 3 edition, 1977.
- 7 HashiCorp. Terraform, 2026. Accessed: 2026-04-28. URL: <https://www.terraform.io/>.
- 8 HashiCorp. Terraform CLI: providers schema command, 2026. Accessed: 2026-04-28. URL: <https://developer.hashicorp.com/terraform/cli/commands/providers/schema>.
- 9 Junda He, Jieke Shi, Terry Yue Zhuo, Christoph Treude, Jiamou Sun, Zhenchang Xing, Xiaoning Du, and David Lo. Llm-as-a-judge for software engineering: Literature review, vision, and the road ahead. *ACM Trans. Softw. Eng. Methodol.*, February 2026. Just Accepted. doi:10.1145/3797276.
- 10 Prithwish Jana, Sam Davidson, Bhavana Bhasker, Andrey Kan, Anoop Deoras, and Laurent Callot. TerraFormer: Automated infrastructure-as-code with llms fine-tuned via policy-guided verifier feedback, 2026. doi:10.48550/arXiv.2601.08734.
- 11 Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. InferFix: End-to-end program repair with llms. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1646–1656. ACM, 2023. doi:10.1145/3611643.3613892.
- 12 Claire Le Goues, Michael Pradel, Abhik Roychoudhury, and Satish Chandra. Automatic program repair. *IEEE Software*, 38(4):22–27, 2021. doi:10.1109/MS.2021.3072577.
- 13 En Low, Carmen Cheh, and Binbin Chen. Repairing infrastructure-as-code using large language models. In *2024 IEEE Secure Development Conference (SecDev)*, pages 20–27. IEEE, 2024. doi:10.1109/SecDev61143.2024.00008.
- 14 Minase Mekete Mengistu, Juri Di Rocco, Phuong T. Nguyen, and Davide Di Ruscio. Replication Package “TerraRepair: A Tool-Grounded LLM Agent for Infrastructure-as-Code Repair”. <https://github.com/ManassehV2/TerraRepair>, 2026. Accessed: 2026-07-18.
- 15 National Security Agency. Enforce secure automated deployment practices through infrastructure as code. Technical report, National Security Agency, March 2024. URL: <https://media.defense.gov/2024/Mar/07/2003407857/-1/-1/0/CSI-CLOUDTOP10-INFRASTRUCTURE-AS-CODE.PDF>.
- 16 OpenAI. Best practices for prompt engineering with the openai api, 2026. Accessed: 2026-03-07. URL: <https://help.openai.com/en/articles/6654000-best-practices-for-prompt-engineering-with-the-openai-api>.
- 17 Palo Alto Networks. Checkov, 2026. Accessed: 2026-04-28. URL: <https://www.checkov.io/>.

- 18 Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. Examining zero-shot vulnerability repair with large language models. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2339–2356. IEEE, 2023. doi:10.1109/SP46215.2023.10179324.
- 19 Yiming Qiu, Patrick Tser Jern Kon, Ryan Beckett, and Ang Chen. Unearthing semantic checks for cloud Infrastructure-as-Code programs. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 574–589. ACM, 2024. doi:10.1145/3694715.3695974.
- 20 Akond Rahman, Chris Parnin, and Laurie Williams. The seven sins: Security smells in infrastructure as code scripts. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*, pages 164–175. IEEE/ACM, 2019. doi:10.1109/ICSE.2019.00033.
- 21 Raul Y. Reyes, Benjamin M. Ampel, and Hsinchun Chen. Large language models for infrastructure as code vulnerability remediation. In *WISP 2025 Proceedings*, 2025. URL: <https://aisel.aisnet.org/wisp2025/1>.
- 22 Nuno Saavedra, João F. Ferreira, and Alexandra Mendes. InfraFix: Technology-agnostic repair of infrastructure as code, 2025. doi:10.48550/arXiv.2503.17220.
- 23 Yu Shi, Hao Li, Bram Adams, and Ahmed E. Hassan. HAFixAgent: History-aware automated program repair agent, 2025. doi:10.48550/arXiv.2511.01047.
- 24 Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 532–543. ACM, 2015. doi:10.1145/2786805.2786825.
- 25 Tenable. KaiMonkey: Vulnerable Terraform Infrastructure, 2026. Accessed: 2026-04-28. URL: <https://github.com/tenable/KaiMonkey>.
- 26 Dheer Toprani and Vijay K. Madiseti. LLM agentic workflow for automated vulnerability detection and remediation in infrastructure-as-code. *IEEE Access*, 13:69175–69181, 2025. doi:10.1109/ACCESS.2025.3560911.
- 27 Edwin B. Wilson. Probable inference, the law of succession, and statistical inference. *Journal of the American Statistical Association*, 22(158):209–212, 1927. doi:10.1080/01621459.1927.10502953.
- 28 Zhenning Yang, Hui Guan, Victor Nicolet, Brandon Paulsen, Joey Dodds, Daniel Kroening, and Ang Chen. Automated cloud infrastructure-as-code reconciliation with ai agents, 2025. doi:10.48550/arXiv.2510.20211.
- 29 Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023. arXiv:2210.03629.
- 30 Ziyang Ye, Triet Huynh Minh Le, and M. Ali Babar. LLMSecConfig: An llm-based approach for fixing software container misconfigurations, 2025. doi:10.48550/arXiv.2502.02009.