International Journal for Innovative Engineering and Management Research
PEER REVIEWED OPEN ACCESS INTERNATIONAL JOURNAL
www.ijiemr.org

A Taxonomy of Failure Modes in LLM-Generated Infrastructure-as-Code Remediation

[DEVA RAJU GANTAKORA]
[Lead DevOps Engineer, Kent State University]
[Kent, USA] | [dgantako@kent.edu]

Abstract

Infrastructure as Code lets teams define cloud resources in machine readable files such as Terraform or CloudFormation templates. As these files multiply across many cloud regions, security misconfigurations and configuration drift become difficult to repair by hand. A growing body of work uses large language models to propose fixes directly inside the delivery pipeline, which can shorten the time needed to repair a problem. A generated fix, however, can fail in ways that a passing check does not always reveal, and the field still lacks a shared vocabulary for those failures. This paper does not run a new experiment. It reviews published studies on language model code generation, infrastructure as code security, and model hallucination, and from them derives a structured taxonomy of the ways an automatically generated infrastructure fix can go wrong. The failures are grouped into three families: correctness failures, where the fix does not work as code; fabrication failures, where the fix points to resources or attributes that do not exist; and contextual failures, where the fix is valid on its own but wrong for the target region, policy, or surrounding modules. The families are then compared along three practical dimensions: how readily each is caught by static tools, how readily each is caught at plan time, and how damaging each is if it reaches production. The comparison indicates that contextual failures are both the hardest to detect automatically and the most costly when missed, which points to where human review and retrieval based grounding should be concentrated.

Keywords: Large Language Models, Infrastructure as Code, Automated Remediation, Failure Taxonomy, Cloud Security, DevOps.

1. Introduction

1.1 Background of the Study

Infrastructure as Code is the practice of describing servers, networks, storage, and access rules in declarative files rather than configuring them by hand. Tools such as Terraform, CloudFormation, and Pulumi read these files and create the matching cloud resources. Modern enterprises run the same workloads in several regions at once to lower latency and to keep services available if one region fails. This scale brings a cost. A single platform can hold thousands of resource definitions, and small differences between regions accumulate into configuration drift and security gaps over time [9].

For years the standard response has been detection. Policy as code engines and static scanners read the templates and flag risky patterns, for example a storage bucket left open to the public or an access role granted more permission than it needs [8]. Detection alone does not close the gap, because a human still has to write the corrected code. Recent work moves from detection to repair by placing a language model

in the pipeline so that the model proposes the corrected template, which is then checked and either accepted or sent back for another attempt [6].

1.2 Statement of the Problem

Automated repair raises a question that detection never had to answer: can the proposed fix be trusted. A fix that compiles and survives a dry run is not always a correct fix. It may resolve the reported issue while quietly breaking a reference in another module, it may apply settings that suit one region but violate the rules of another, or it may rely on an identifier that the model invented. General studies of model written code already report high rates of insecure output and of invented references, yet these findings are scattered across different problems and are not organized around the specific task of repairing existing infrastructure files [1,4,5]. Without a clear map of how these fixes fail, teams cannot decide which guardrail belongs at which stage of the pipeline.

1.3 Objectives of the Study

- To build a literature grounded taxonomy of the failure modes that are specific to language model generated infrastructure as code remediation.
- To compare the resulting failure families by how easily each is detected and how severe each is in production.
- To indicate where in the delivery pipeline each family should be caught and which mitigation suits it best.

2. Related Work

2.1 Remediation as Generation Followed by Validation

Automated remediation can be read as two steps: the model generates a candidate fix, and an external check decides whether the candidate is acceptable. The reasoning and acting pattern formalizes this loop, letting a model alternate between proposing an action and reading the result of that action so it can revise its next step [6]. In a delivery pipeline the action is the proposed template and the result is the output of a linter or a plan command, whose error log can be fed back for another attempt. This loop is the mechanism that lets a model recover from many simple mistakes, but it only catches failures that the validator can see.

2.2 Known Failure Patterns in Model Written Code

Several studies measure how often model written code is unsafe. An assessment of a popular code assistant found that close to forty percent of its completions for security sensitive scenarios contained a weakness drawn from a standard catalog of common flaws [1]. A field study of real projects reported that around a third of analyzed assistant generated snippets carried vulnerabilities spanning many distinct weakness types [2], and a controlled user study found that developers with an assistant tended to produce less secure code while believing the opposite [3]. A separate line of work measures invention rather than insecurity. A survey of hallucination distinguishes output that contradicts its source from output that the source simply cannot support [4], and a large analysis of package hallucination showed that code models

recommend non existent libraries at rates that ranged from roughly five percent on commercial models to over twenty percent on open source models, creating a supply chain risk when attackers register the invented names [5].

2.3 Defects and Security Smells in Infrastructure as Code

Infrastructure files have their own well studied weaknesses. A qualitative study of more than a thousand scripts identified seven recurring insecure patterns, such as hard coded secrets and weak access defaults, and built a linter to find them at scale [8]. Follow up work catalogued development antipatterns in these files and traced how a weakness in one resource can propagate to others [9,10]. What separates infrastructure files from ordinary application code is that they are stateful and platform dependent. A template that is valid on a clean slate can still fail when applied against resources that already exist, and a benchmark of model generated Terraform found that even a strong model passed under twenty percent of compositional scenarios on the first attempt, far below its score on general Python tasks [11,12].

2.4 The Research Gap

Existing work either catalogs weaknesses in human written infrastructure files, or measures failure rates of language models on general code, or surveys hallucination in open ended text. None of these targets the precise task studied here, which is the repair of an existing infrastructure template by a model. Repair is not the same as generation. A repair must change one thing while preserving working behavior, on infrastructure that is stateful and that may span regions with different rules. The absence of a failure taxonomy for this task is the gap this paper addresses.

3. Method

3.1 Approach

This study is a conceptual synthesis rather than an experiment. We collected published empirical studies, surveys, and benchmarks across three areas: the security of model written code, hallucination in language models, and defects in infrastructure as code. From each source we extracted the concrete failure types it reported, then grouped recurring types into families according to two questions: what underlying cause produces the failure, and at which point in the pipeline the failure first becomes visible. The result is the taxonomy presented in Section 4. We state plainly that the comparative ratings in this paper are analytical judgments synthesized from the cited literature, not frequencies measured in a new dataset. Assigning real frequencies would require a labeled corpus of model generated fixes, which we identify as future work.

3.2 Scope

The taxonomy targets declarative infrastructure as code in the style of Terraform, in the remediation setting where a model edits an existing template to remove a reported defect. Single region and multiple region deployments are both in scope. Runtime self healing of live systems and the broad quality of greenfield generation are out of scope, except where studies of generation supply evidence about how models fail.

International Journal for Innovative Engineering and Management Research
PEER REVIEWED OPEN ACCESS INTERNATIONAL JOURNAL
www.ijiemr.org

3.3 Comparison Dimensions

Each failure family is rated on three lenses. The first is detectability by static tools, meaning whether a linter or policy engine can catch the failure before anything is applied. The second is detectability at plan time, meaning whether a dry run that previews changes against current state would surface the failure. The third is production severity, meaning the likely impact if the failure escapes both checks and is applied. Each lens uses three levels, low, medium, and high, defined relative to the other families rather than on an absolute scale.

4. The Taxonomy

4.1 Overview

The taxonomy organizes failures into three families and eight leaf modes, shown in Figure 1. Family A, correctness, covers fixes that do not work as code. Family B, fabrication, covers fixes that reference things which do not exist. Family C, context, covers fixes that are valid in isolation but wrong for their setting. The families are ordered roughly by how late they tend to surface, from failures a compiler can see to failures that only a human who understands intent can judge.

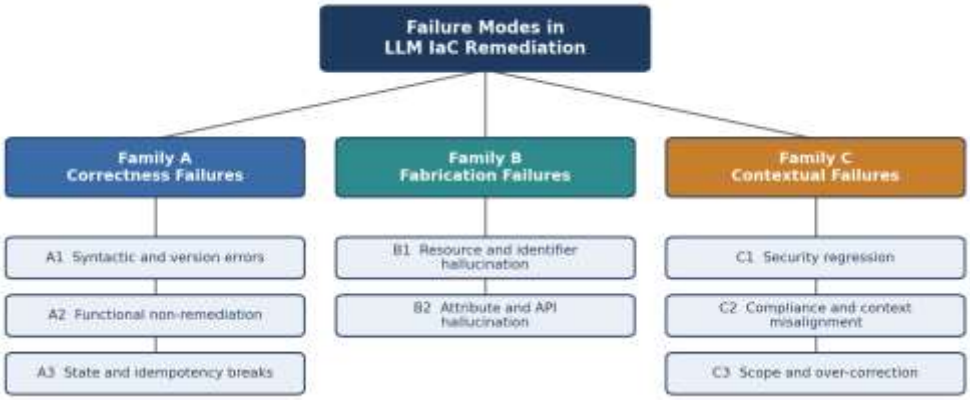

Figure 1. Three families and eight leaf modes of failure in model generated infrastructure remediation.

4.2 Family A: Correctness Failures

A1. Syntactic and version errors. The fix does not parse, or it uses an argument or block that the chosen provider version does not accept. A model trained on a mix of old and new examples can blend syntax from different versions of a provider. These errors are the most visible, since a linter or a plan command rejects them immediately [6,8].

A2. Functional non remediation. The fix parses and applies cleanly yet does not actually close the reported issue, or it closes the issue while disabling the resource it was meant to protect. A bucket may stay reachable through a path the fix did not consider, or an access rule may be tightened so far that a dependent service can no longer connect. The code is valid, so syntax checks pass, and only a test that exercises the intended behavior reveals the problem.

A3. State and idempotency breaks. Because infrastructure files are stateful, a fix that is correct on a clean slate can conflict with resources that already exist. A change to an immutable field can force a resource to be destroyed and recreated, which a plan preview will show as replacement rather than update. This mode is specific to infrastructure and rarely appears in studies of ordinary code [10].

4.3 Family B: Fabrication Failures

B1. Resource and identifier hallucination. The fix inserts an identifier that does not exist in the target, such as a machine image that is not published in the destination region, an instance size that the region does not offer, a resource name that was never defined, or a module source that cannot be found. This is the infrastructure form of the package hallucination measured in general code, where models confidently produce names that point to nothing [4,5].

B2. Attribute and API hallucination. The fix uses a provider argument, attribute, or data source that the model has misremembered or invented. The surrounding structure looks plausible, which makes the error easy to miss on a quick read, although a plan command usually rejects an unknown argument. The low first attempt pass rates reported for model generated infrastructure are consistent with frequent invention of this kind [11,12].

4.4 Family C: Contextual Failures

C1. Security regression. The fix removes the reported weakness but introduces a new one, or it chooses an insecure default such as an over permissive access rule or disabled encryption. This mirrors the broad finding that a large share of model written code carries weaknesses even when it satisfies the immediate request [1,2,3].

C2. Compliance and context misalignment. The fix is valid in general but violates a rule that applies to the specific deployment, most often a data residency or retention requirement that differs by region. A change that enables logging to a storage location in one country can break the residency rules of another. A static tool that applies one rule set everywhere cannot judge this, because the correct answer depends on context the template does not state.

C3. Scope and over correction. The fix changes more than the targeted defect. It edits adjacent settings, alters shared values, or updates a field that other modules depend on, so that repairing one resource quietly breaks another. This connects to the documented difficulty models have with dependencies that cross module boundaries [9,10].

4.5 Comparative Analysis

Table 1 rates each leaf mode on the three lenses and names the control best suited to catch it. Figure 2 summarizes the same ratings at the family level. A clear pattern emerges. Correctness failures are mostly caught early, since a linter or a plan preview rejects bad syntax, replacements, and many broken references before anything is applied. Fabrication failures are caught somewhat later but still by automation, because an unknown argument or a missing resource usually fails the plan step, with the dangerous exception of an invented value that happens to resolve to a real but wrong target. Contextual failures sit apart. They tend to compile, to pass a dry run, and to satisfy a generic policy scan, yet they carry the highest impact when they reach production. This combination of low automated detectability and high severity is the central finding of the comparison.

Failure mode	Static tools	Plan time	Production severity	Best suited control
A1 Syntactic and version	High	High	Low	Linter in the pipeline; reject and retry loop
A2 Functional non remediation	Low	Medium	Medium	Behavior test that checks the issue is truly closed
A3 State and idempotency	Low	High	High	Plan review for replacement of existing resources
B1 Resource and identifier hallucination	Medium	High	Medium	Plan against real state; lookup rather than literals
B2 Attribute and API hallucination	Medium	High	Low	Schema validation; retrieval of provider docs
C1 Security regression	Medium	Low	High	Policy as code re scan after the fix
C2 Compliance and context misalignment	Low	Low	High	Region aware policy; retrieval of local rules; human review
C3 Scope and over correction	Low	Medium	High	Diff review limited to the intended resource; human review

Table 1. Failure modes rated by detectability and production severity, with the control best suited to each. Ratings are analytical assessments synthesized from the cited literature, not measured frequencies.

International Journal for Innovative Engineering and Management Research
PEER REVIEWED OPEN ACCESS INTERNATIONAL JOURNAL
www.ijiemr.org

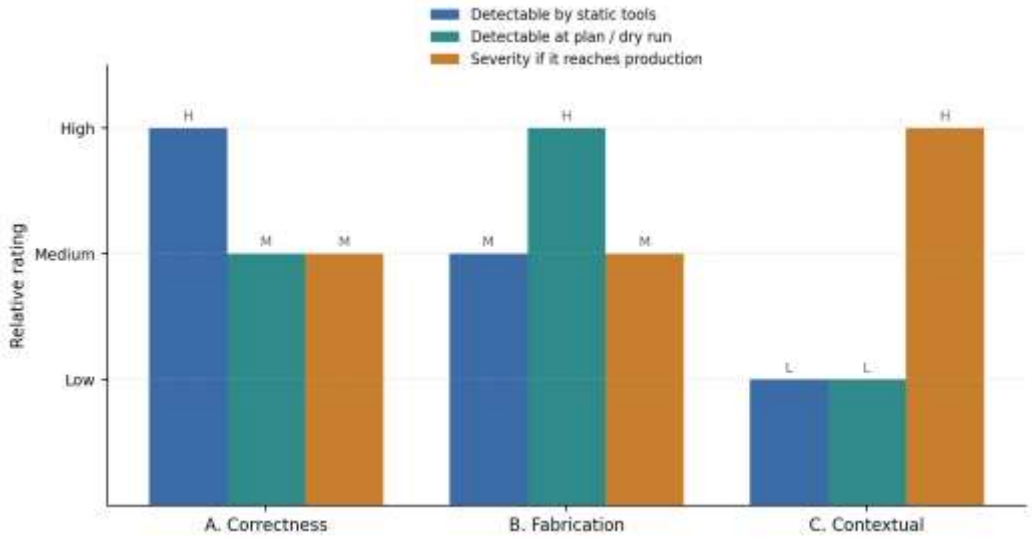

Figure 2. Family level comparison. Contextual failures combine the lowest automated detectability with the highest production severity.

4.6 Discussion: Placing the Right Guardrail

The taxonomy suggests a layered defense in which each control is matched to the failures it can actually see. A linter in the pipeline removes syntactic and version errors cheaply, so these should never reach a human. A plan preview against current state is the natural place to catch state breaks and most invented references, since both surface as rejected or unexpected changes. A policy scan repeated after the fix, rather than only before it, catches a security regression that the repair introduced. None of these automated layers can resolve the contextual family on its own. A generic policy rule cannot know that a region forbids a storage location, and a diff tool cannot know which adjacent edits were intended. Two approaches narrow this residual gap. Retrieval based grounding supplies the model with the specific rules, identifiers, and provider documentation for the target before it writes the fix, which reduces both invention and context misalignment by replacing guesses with looked up facts [7]. Focused human review then handles what remains, concentrated on the contextual family rather than spread evenly across every change.

4.7 Limitations

The taxonomy is synthesized from published literature and has not yet been validated against a freshly labeled corpus of model generated fixes, so the comparative ratings are qualitative and relative. The boundaries between modes can overlap, since a single fix may be both a hallucinated attribute and a security regression. The tooling landscape also evolves, which can shift a failure from hard to detect toward easy to detect as scanners improve. These limits point directly to the most useful next step, described in the conclusion.

5. Conclusion

Automated repair of infrastructure as code can shorten the path from a detected problem to a working change, but it shifts the hard question from detection to trust. This paper organized the ways a model generated fix can fail into three families, correctness, fabrication, and context, and compared them by how readily each is caught and how much each costs when missed. The comparison shows that the failures easiest to detect are also the least damaging, while the failures that carry the most risk, those tied to regional rules and to the scope of a change, are the ones automated tooling is least able to see. The practical message is to align controls with this map: let linters and plan previews dispatch the cheap failures, repeat policy scans after the fix, ground the model with retrieved facts, and reserve human attention for the contextual family. The clearest direction for future work is an empirical labeling study that applies this taxonomy to a corpus of real model generated fixes, turning the qualitative ratings here into measured frequencies and detection rates.

References

1. Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., & Karri, R. (2022). Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. In 2022 IEEE Symposium on Security and Privacy (SP), pp. 754-768.
2. Fu, Y., Liang, P., Tahir, A., Li, Z., Shahin, M., Yu, J., & Chen, J. (2023). Security Weaknesses of Copilot Generated Code in GitHub. arXiv preprint arXiv:2310.02059.
3. Perry, N., Srivastava, M., Kumar, D., & Boneh, D. (2023). Do Users Write More Insecure Code with AI Assistants? In Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS).
4. Siddiq, M. L., & Santos, J. C. S. (2023). Generate and Pray: Using SALLMs to Evaluate the Security of LLM Generated Code. arXiv preprint arXiv:2311.00889.
5. Ji, Z., Lee, N., Frieske, R., Yu, T., Su, D., Xu, Y., Ishii, E., Bang, Y. J., Madotto, A., & Fung, P. (2023). Survey of Hallucination in Natural Language Generation. ACM Computing Surveys, 55(12), Article 248.
6. Spracklen, J., Wijewickrama, R., Sakib, A. H. M. N., Maiti, A., Viswanath, B., & Jadliwala, M. (2025). We Have a Package for You! A Comprehensive Analysis of Package Hallucinations by Code Generating LLMs. In 34th USENIX Security Symposium, pp. 3687-3706.
7. Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., & Cao, Y. (2023). ReAct: Synergizing Reasoning and Acting in Language Models. In International Conference on Learning Representations (ICLR).
8. Rahman, A., Parnin, C., & Williams, L. (2019). The Seven Sins: Security Smells in Infrastructure as Code Scripts. In 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pp. 164-175.
9. Rahman, A., Farhana, E., & Williams, L. (2020). The 'as Code' Activities: Development Antipatterns for Infrastructure as Code. Empirical Software Engineering, 25(5), pp. 3430-3467.
10. Rahman, A., & Parnin, C. (2023). Detecting and Characterizing Propagation of Security Weaknesses in Puppet-based Infrastructure Management. IEEE Transactions on Software Engineering, 49(6), pp. 3536-3553.
11. Srivatsa, K. G., Mukhopadhyay, S., Katrapati, G., & Shrivastava, M. (2023). A Survey of using Large Language Models for Generating Infrastructure as Code. In Proceedings of the 20th International Conference on Natural Language Processing (ICON), pp. 523-533.
12. Kon, P. T. J., Liu, J., Qiu, Y., Fan, W., He, T., Lin, L., Zhang, H., Park, O. M., Elengikal, G. S., Kang, Y., Chen, A., Chowdhury, M., Lee, M., & Wang, X. (2024). IaC-Eval: A Code Generation Benchmark for Cloud Infrastructure-as-Code Programs. In Advances in Neural Information Processing Systems (NeurIPS) 37, Datasets and Benchmarks Track, pp. 134488-134506.

13. Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Kuttler, H., Lewis, M., Yih, W., Rocktaschel, T., Riedel, S., & Kiela, D. (2020). Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In Advances in Neural Information Processing Systems (NeurIPS) 33, pp. 9459-9474.