From Security Findings to Verified Fixes: Automated Repair of Infrastructure-as-Code Misconfigurations

Ying Li*
UCLA
Los Angeles, United States
yinglee@ucla.edu

Tamjid Al Rahat
Amazon Web Services
San Jose, USA
tamjr@amazon.com

Yingjun Lyu
Amazon Web Services
Los Angeles, USA
yingjunl@amazon.com

Omer Tripp
Amazon Web Services
San Jose, USA
omertrip@amazon.com

George Argyros
Amazon Web Services
New York, USA
yorgos@amazon.com

Abstract

Traditional Infrastructure as Code misconfiguration remediation requires manual analysis, a labor-intensive approach that cannot meet the scale and speed demands of modern cloud deployments. We present IacFix, an agentic tool that combines LLM-based agents with formal methods to enable automated IaC security remediation from security findings to validated patches. IacFix introduces end-to-end remediation capabilities across CDK and CloudFormation packages, leveraging historical fix patterns and iterative refinement to produce validated patches for complex misconfigurations. For example, IacFix achieves 80.9% end-to-end success rate with 97.7% fault localization precision and 7.12 minutes average processing time, demonstrating comparable effectiveness to manual fixes while delivering automated solutions significantly faster. Notably, IacFix outperforms LLM-only baselines by 45 percentage points in real-world scenarios such as dangling hostclass remediation. The system exhibits one-sided error: it either produces a validated patch or safely declines to patch, ensuring that no incorrect patch is surfaced to developers. Our approach shows promising potential for addressing diverse misconfiguration types at scale.

CCS Concepts

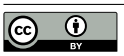
• **Security and privacy** → **Software security engineering**; • **Software and its engineering** → *Software verification and validation.*

Keywords

Infrastructure as Code, automated program repair

ACM Reference Format:
Ying Li, Tamjid Al Rahat, Yingjun Lyu, Omer Tripp, and George Argyros. 2026. From Security Findings to Verified Fixes: Automated Repair of Infrastructure-as-Code Misconfigurations. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3832783.3834509

*This work was done while the author was an intern at Amazon Web Services.



This work is licensed under a Creative Commons Attribution 4.0 International License.
ASE '26, Munich, Germany
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3834509

1 Introduction

Infrastructure as Code (IaC) has revolutionized cloud infrastructure management by enabling programmable provisioning and configuration through code rather than manual processes, significantly improving deployment reliability and reducing recovery time by up to 90% [5, 13, 31, 33]. Yet this automation comes at a cost. When security misconfigurations occur in IaC templates, they propagate automatically across entire cloud infrastructures, turning isolated errors into widespread issues [6, 8, 11]. The scale of this problem is significant: 90% of organizations have experienced cloud misconfigurations in recent years, with these incidents now representing 30% of all cloud security issues. These issues typically take 186 days to identify and another 65 days to contain, with misconfiguration-related incidents taking 20% longer to detect and contain than other security issues [34].

Current remediation relies on manual analysis by developers who must understand complex cloud architectures and craft precise fixes without breaking functionality or introduce new issues. This labor-intensive approach cannot scale with modern cloud deployment speeds, creating an urgent need for automated remediation solutions.

Recent advancements in Large Language Models (LLMs) have shown remarkable capabilities in automated program repair (APR) [1, 19, 22, 25, 36, 39]. However, existing approaches face critical limitations when applied to IaC security remediation. Unlike traditional program repair where failing tests or stack traces pinpoint the defect location, IaC remediation must bridge a fundamental abstraction gap between deployed resource states and source-level definitions. These limitations manifest in three key challenges: **(1)** ***Cross-abstraction traceability:*** resource-level violation reports (e.g., account ID and non-compliant items) must be traced back to the responsible source-code locations through multiple layers of abstraction (deployment pipelines, synthesis steps, dynamic code generation), **(2)** ***High-quality patch synthesis*** that translates abstract security guidance into concrete fixes for heterogeneous IaC codebases spanning multiple languages and frameworks, and **(3)** ***Rigorous validation*** that ensures patches eliminate misconfigurations without introducing new security issues and

operation side effects, where LLM-based validation alone is insufficient due to its inherent non-deterministic nature that provides no formal guarantees, and a single incorrect validation could leave more misconfigurations unaddressed or introduce new security risks.

To address these challenges, we propose IacFix, an agentic system for automated IaC misconfiguration repair that combines LLM-based agents with SMT-based validation to achieve reliable, safe remediation. The key insights of IacFix are: (1) We bridge the gap between security alerts and code fixes by systematically mapping account-level violations back to the specific code locations that caused them through multi-stage analysis of deployment pipelines. (2) We improve patch quality by extracting historical fix patterns from resolved findings, enabling LLMs to learn from real developer strategies rather than relying solely on abstract security guidance. (3) We ensure reliability through hybrid validation combining deterministic verification with semantic analysis. The SMT-based validation component provides soundness guarantees for the security resolution check (under a faithful encoding of CloudFormation intrinsics), and the overall system exhibits one-sided error: it either produces a validated patch or safely declines, ensuring no incorrect patch is surfaced. (4) We enable iterative refinement using validation feedback to progressively improve patch quality and handle complex cases requiring multiple iterations.

Our evaluation demonstrates IacFix's effectiveness across 64 IaC packages[1] , achieving 80.9% end-to-end success rate on 47 packages with confirmed violations and high precision (97.7%) in fault localization. Importantly, our formal verification-based guardrail caught 55% of incorrect patches that LLM-based validation would have missed, preventing potential service failures. IacFix exhibits one-sided error: among 17 packages without actual violations, the system correctly identified 16 as having no issues; the single localization false positive was caught by downstream validation (the generated patch did not remove any dangling resources and was therefore rejected), resulting in zero incorrect patches surfaced end-to-end across all 64 packages. Ablation studies show that historical patterns and iterative refinement work synergistically, achieving 45 percentage points improvement over LLM-only baselines while processing packages in 7.12 minutes average.

Contributions. We summarize our contributions as follows:

- **Agentic IaC Repair Tool:** We present the first end-to-end automated tool for IaC misconfiguration repair and instantiate it for dangling-hostclass violations that employs specialized LLM agents for fault localization, patch generation, and semantic validation, combined with formal verification methods to enable fully automated processing from security alerts to validated fixes.
- **Historical Specification-Guided Patch Localization and Generation:** We develop a systematic approach to automatically extract and leverage historical developer fix patterns, enabling LLMs to generate contextually appropriate patches that mirror real-world repair strategies rather than relying solely on abstract security guidance.
- **Multi-Layer Validation with One-Sided Error:** We design a multi-layered validation method that combines SMT-based security verification with LLM-based semantic analysis. The validation component provides soundness guarantees for security resolution under a faithful encoding of CloudFormation intrinsics, and the overall system ensures one-sided error: incorrect patches are rejected rather than surfaced.

[1] 33 CDK packages (pre-filtered to confirm violations) + 31 CFN packages (all evaluated since violations cannot be determined statically; 14 were found to contain actual violations). Packages were randomly sampled from our production monitoring system.

2 Motivating Example

In modern cloud environments, infrastructure is defined using two complementary IaC frameworks. *AWS CloudFormation* uses declarative JSON/YAML templates that directly specify resource configurations. *AWS CDK* (Cloud Development Kit) allows developers to define infrastructure in general-purpose programming languages (e.g., TypeScript, Java); CDK source code is then *synthesized* into CloudFormation templates (stored in a `cdk.out` directory), which are subsequently deployed to provision cloud resources. This synthesis pipeline introduces multiple abstraction layers between source code and deployed state.

A key security construct in this context is the IAM (Identity and Access Management) *trust policy*, which governs which entities may assume a given role. Trust policies reference *hostclasses*, service identity labels that represent groups of compute instances. A *dangling hostclass* occurs when a trust policy references a hostclass that no longer exists in the target environment, creating an exploitable gap: an adversary could register a new hostclass with the dangling name and thereby gain unauthorized access to the role's permissions.

We now present a concrete example of this misconfiguration and discuss the challenges of automating its repair.

```
export class ToyIamStack extends ToyStack {
   ...
    const testRoleArn = `arn:${this.partition}:iam::${props.test.
          account}:role/IaCTestRole`;
   ...
    testRole.attachInlinePolicy(servicePermissions);
    const testHostclass = new Array<string>();
    if (Conditions.isProdStack(stage)) {
     testHostclass.push(`TEST-ABC-XYZ-${airportCode.toUpperCase()}`);

     testHostclass.push(`TEST-ABC-PROD-${airportCode.toUpperCase()}-
          SECONDARY`);
    } else {
        testHostclass.push(`TEST-ABC-ALPHA-${airportCode.toUpperCase
             ()}`);
        testHostclass.push(`TEST-ABC-BETA-${airportCode.toUpperCase
             ()}`);
        testHostclass.push(`TEST-ABC-GAMMA-${airportCode.toUpperCase
             ()}`);
      }
   ...
}
```

Figure 1: Motivating Example of Dangling Hostclass References in IAM Role Configuration

Motivating Example. As shown in Figure 1, this CDK code defines an IAM deployment stack that creates test roles and configures hostclass arrays (i.e., adding trust relationships that allow the listed hostclasses to assume the role) for different deployment environments. The code dynamically generates hostclass names based on the deployment stage (i.e., production vs non-production) and airport codes. In production

environments (line 7 to line 9), it creates hostclasses with consistent naming patterns (`TEST-ABC-XTZ-${airportCode}$` and `TEST-ABC-PROD-${airportCode}$`). While in non-production environment (line 11 to line 13), it generates another three different hostclass variants.

However, when the code is deployed, not all generated hostclass names correspond to actually provisioned hostclasses in the target environment. For example, the non-production branch generates hostclasses such as `TEST-ABC-ALPHA-BAH`, `TEST-ABC-BETA-KIX`, `TEST-ABC-BETA-IAD`, `TEST-ABC-BETA-BAH`, and `TEST-ABC-ALPHA-KIX`, which are included in the IAM role's trust policy despite being non-existent. Such dangling references pose a security risk: an unauthorized user could create new hostclasses with these names and gain unauthorized access to the role's permissions. Therefore, these dangling references must be removed to prevent inadvertent access to sensitive resources.

Challenges. While the misconfiguration may appear straightforward, automatically fixing it poses several challenges. **(C1) Cross-Abstraction Traceability.** The violation report provides only an account ID and non-compliant items. Tracing these back to source locations requires resolving indirect account-to-package mappings and understanding dynamic code generation (loops, conditionals, string interpolation) as in Figure 1. **(C2) Patch Generation.** IaC codebases span multiple languages and frameworks, and official remediation guidance offers only high-level principles rather than actionable code fixes, making LLM-generated patches unreliable without grounding in proven repair strategies. **(C3) Patch Verification.** A correct patch must precisely remove all dangling hostclasses without over-removal (breaking functionality) or under-removal (leaving violations). Source-level changes pass through intermediate synthesis steps (e.g., CDK to CloudFormation templates), complicating correctness checking.

Our Method in a Nutshell. To address these challenges, we propose IacFix, an automated agentic system that generates verified patches for IaC security misconfigurations (Figure 2). First, IacFix derives fix specifications from historical developer repairs, providing concrete guidance for patch generation. Second, IacFix traces scanner alerts back to specific code locations through deployment pipeline graph traversal and agent-based fault localization. Third, guided by the fix specifications and the localized root causes, IacFix generates patches using an LLM agent. Finally, IacFix validates patches through static analysis and SMT-based formal verification, and iteratively refines failed patches using validation feedback.

3 Methodology

In this section, we present the methodology of IacFix. We first provide a high-level overview of the system workflow, illustrating how IacFix transforms account-level security alerts into verified source-code patches. We then describe the root cause localization process, which traces from scanner findings to specific code locations through deployment pipeline analysis and agent-based fault localization. Next, we detail the patch generation approach, which derives fix specifications from historical developer repairs to guide misconfiguration repair. Finally, we present the multi-layered patch validation framework, which combines static analysis and formal verification to ensure precise misconfiguration resolution, along with the iterative refinement mechanism that leverages validation feedback to refine failed patches.

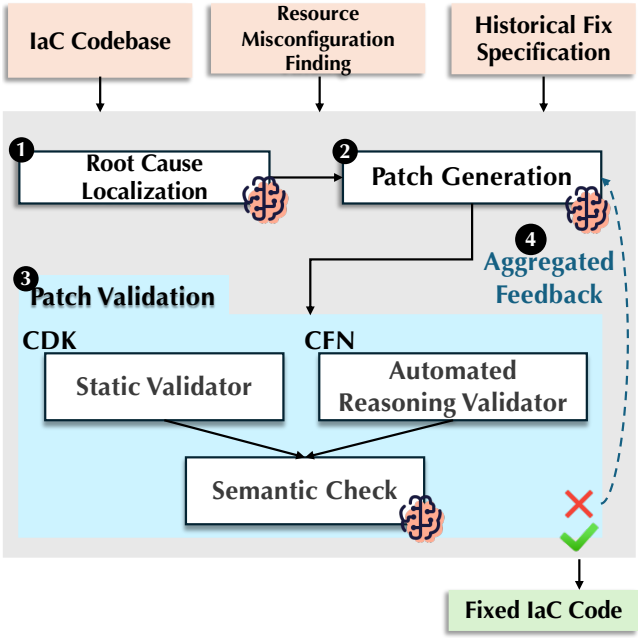


Figure 2: System Architecture of IacFix. The system takes three inputs: IaC codebase, resource-misconfiguration finding, and historical fix specifications. (1) Root cause localization traces findings to specific code locations; (2) Patch generation produces targeted fixes guided by fix specifications; (3) Multi-layered patch validation including build testing, static/automated reasoning validators, and semantic check; (4) Aggregated feedback refines failed patches iteratively.

3.1 Overview

Figure 2 presents the overall workflow of IacFix. The system takes three inputs: an IaC codebase, a resource-misconfiguration finding (containing violated policy title, account ID, and non-compliant items), and historical fix specifications derived from previously resolved findings.

Given these inputs, IacFix operates through four stages. ❶ *Root Cause Localization* traces from the resource-misconfiguration finding back to the specific code location responsible for the misconfiguration. It first identifies candidate packages via deployment pipeline graph traversal, then employs an AI agent to pinpoint the exact root cause within the code. ❷ *Patch Generation* takes the localized root cause and historical fix specifications as input, and generates targeted code modifications using an LLM agent. If root cause localization fails, the process terminates, as generating patches without a precise root cause lacks practical value. ❸ *Patch Validation* verifies the generated patch through multiple layers. It first performs build testing to ensure the patched code compiles successfully. Then, for CDK packages, a static validator compares pre- and post-patch synthesized templates; for CloudFormation (CFN) packages, an automated reasoning validator uses SMT solvers [4] to formally verify

that only the intended misconfigurations are eliminated. Finally, both paths undergo a semantic check by an AI agent to ensure no unintended functional changes are introduced. ❹ If validation fails, *aggregated feedback* from the validation components is used to iteratively refine the patch with the counter-example, looping back to the patch generation stage. This process continues until the patch passes all validation checks or a maximum number of refinement attempts is reached. Upon successful validation, IacFix outputs the fixed IaC code.

3.2 Root Cause Localization

Root cause localization addresses the cross-abstraction traceability problem: given a resource-level violation report (account ID, non-compliant items), identify the source-code location responsible for generating the non-compliant resources. IacFix decomposes this into two phases: (1) *package identification*, which narrows the search space to candidate packages, and (2) *fault localization*, which pinpoints the exact source within a package.

Package Identification. As a platform-specific preprocessing step, IacFix resolves the account ID to candidate code packages by querying deployment pipeline metadata. In typical cloud environments, deployment pipelines orchestrate the deployment of code packages through multiple stages, each targeting specific accounts. By leveraging this account-to-pipeline-to-package relationship, IacFix narrows the search space from all packages in the organization to a tractable set of candidates (18 packages per finding on average).

Fault Localization. The core localization challenge is determining, within a candidate package, which code is responsible for generating a given non-compliant resource. This is a form of *dynamic provenance* problem [7]: the relationship between source definitions and generated resources is mediated by synthesis steps (CDK compilation, template expansion) and runtime parameters, making static rule-based matching insufficient.

We address this with an AI agent that provides two key capabilities: (1) *adaptive search*, which decomposes non-compliant items into candidate patterns (e.g., deriving `"TEST-ABC-*"` or `"*-BETA-*"` from `"TEST-ABC-BETA-BAH"`) and iteratively refines the search strategy based on intermediate results, and (2) *contextual judgment*, which reads surrounding code to distinguish actual resource definitions from unrelated references. This discrimination requires understanding of the code's generative semantics (e.g., recognizing that a loop over airport codes with string interpolation produces the reported hostclass names).

Concretely, the agent operates in an iterative loop: it decomposes the non-compliant items into candidate search patterns, uses shell tools (e.g., `find`, `grep`) to locate matches, and reads surrounding context to confirm whether each match corresponds to the actual resource definition. If the current pattern yields no confirmed match, the agent refines its strategy (e.g., broadening substrings or switching to different segments of the non-compliant item) and repeats until the root cause is localized or no further patterns can be derived. Upon successful localization, IacFix advances to patch generation; when localization fails, the system terminates since generating patches without a confirmed root cause lacks practical value.

3.3 Security Patch Generation

After localizing the root cause, IacFix generates patches to address the security issue. However, official remediation guidance typically provides only high-level security principles (e.g., "remove dangling references") without specifying the concrete code-level strategies needed to implement a fix, such as filtering conditions, array modifications, or configuration changes. To bridge this gap, IacFix derives *historical fix specifications* from previously resolved findings, which capture proven, code-level remediation strategies that developers have applied in practice. The workflow operates in two phases: (1) *Fix Specification Extraction* analyzes resolved findings to extract reusable remediation strategies, and (2) *Patch Generation* leverages these specifications to guide patch creation.

Fix Specification Extraction. To provide concrete remediation guidance for patch generation, IacFix extracts fix specifications offline from previously resolved findings before the repair pipeline runs.

The extraction follows two steps. First, *fix commit identification* locates commits that resolved each finding: for each resolved finding, IacFix retrieves commits made after the finding's last observed date and uses an LLM to determine whether each commit addresses the finding by jointly evaluating whether the commit message references the security issue, whether the code diff modifies the reported non-compliant items, and whether associated code review descriptions mention the finding. Second, *fix specification clustering* analyzes the code diffs from identified fix commits using an LLM to abstract away package-specific details (e.g., concrete hostclass names, variable names) and group changes by their underlying remediation strategy. For example, for dangling hostclass issues, the extracted specifications include removing specific hostclass entries from hardcoded arrays, adding conditional logic to exclude certain parameter values during dynamic generation (e.g., filtering out specific airport codes in loops), removing entire hostclass template entries that only produce non-existent hostclasses, etc.

The clustering step abstracts each code diff into a *canonical remediation action* by replacing package-specific identifiers (hostclass names, variable names) with typed placeholders, then groups the abstracted diffs by structural similarity. From 63 historical fixes, clustering yielded 6 distinct specification types, including *array entry deletion* (removing literals from hardcoded arrays), *conditional filtering* (adding `if/Fn::If` guards to exclude parameter values in dynamic templates), *template removal* (deleting entries whose instantiations are entirely dangling), and *loop exclusion* (adding skip conditions in hostclass-generating iterations).

Each specification is annotated with an *applicability predicate* describing its code-structural context (e.g., array deletion applies to static arrays; conditional filtering applies to parameterized templates). During generation, IacFix matches the root cause's code structure against these predicates and provides all matching specifications with representative example diffs. When multiple specifications match, the agent selects the most conservative strategy that preserves legitimate configurations, with correctness enforced by downstream validation.

We employ saturation-based extraction: since clustering abstracts over package-specific details, fixes sharing the same remediation logic map to a single specification. Extraction terminates

when no new types emerge; in our dataset, the set stabilized after approximately 40 fixes. We evaluate specification quality extrinsically through downstream patch success rate (Section 4.2), following established practice in template-guided program repair [17].

Patch Generation. With extracted fix specifications and localized root causes, IacFix generates targeted patches through a structured three-step process. The agent first *analyzes* how the violation items are defined and used in the codebase (e.g., in configuration files, templates, or source code) and identifies the organizational patterns across different contexts such as environments and regions. It then performs a *security assessment* to evaluate existing security controls, cross-environment dependencies, and whether complete removal or conditional access is more appropriate. Finally, it *generates code modifications* guided by the fix specifications, each of which includes detailed remediation strategies and example diffs from historical fixes. As a safety constraint, the agent is instructed to make no modifications if it cannot confirm the presence of the violation items in the codebase.

Additionally, to ensure comprehensive fixes, the agent searches for related references and dependencies that might be affected by the primary patch. This includes: (1) *call graph analysis* to identify classes that inherit from common base classes or use shared constants, (2) *reference tracking* through grep-based searches to locate all usages of violation items across the codebase, and (3) *pattern-based identification* to find similar code structures that require consistent modifications. This prevents incomplete fixes where the root cause is addressed but residual references remain.

3.4 Patch Validation

After the patch candidate is generated, it must be validated before deployment, as a patch that compiles successfully may still fail to resolve the security issue or introduce new problems. LLM-based validation alone is insufficient for this task, as LLM judges can incorrectly approve patches that contain over-removal or under-removal errors (Section 4.1). Even though recent LLMs demonstrate strong code generation and repair capabilities, they can still produce invalid or deceptively successful code and make inconsistent correctness judgments across model families [2, 15]. IacFix therefore employs a multi-layered validation framework combining deterministic methods with semantic analysis: (1) *Security Resolution Validation* confirms the patch precisely addresses the security issue without introducing new problems, and (2) *Semantic Correctness Analysis* verifies the patch maintains intended code logic.

Security Resolution Validation. Security resolution validation ensures that patches achieve *exact resolution*: eliminating all non-compliant items without over-removal (removing legitimate configurations that should be preserved) or under-removal (leaving dangling references unaddressed). For example, when fixing the dangling hostclass issue in Figure 1, a patch that removes the entire template `TEST-ABC-ALPHA-${airportCode}$` would eliminate the dangling `TEST-ABC-ALPHA-BAH` but also remove valid hostclasses, constituting over-removal.

Directly analyzing the source code for exact resolution is complex due to dynamic logic and variable dependencies. Instead, IacFix validates at the CloudFormation template level. For CDK packages, the build process synthesizes the source code (e.g., TypeScript, Java) into CloudFormation templates in the `cdk.out` directory; for CloudFormation packages, the templates are the deployment artifacts themselves. In both cases, complex source-level logic is reduced to straightforward template declarations (Figure 3) that can be analyzed deterministically. These templates contain resources defined as either hardcoded arrays like `["TEST-ABC-PROD-BAH", "TEST-ABC-BETA-BAH"]` (Figure 3a), or dynamically generated using templates such as '`TEST-ABC-ALPHA-${AirportCode}`' with parameters (Figure 3b).

```
--- a/cdk.out/TestStack.template.json
+++ b/cdk.out/TestStack.template.json
@@ -15,8 +15,7 @@
  "Condition": {
    "ForAnyValue:StringLike": {
      "iam.example.com:hostclass": [
        "TEST-ABC-PROD-BAH",
-       "TEST-ABC-BETA-BAH"
-       "TEST-ABC-ALPHA-BAH"
      ]
    }
  }
```

(a) Hardcoded Configurations

```
--- a/cloudFormation/template.yaml
+++ b/cloudFormation/template.yaml
@@ -8,7 +8,15 @@
ForAnyValue:StringLike:
  "iam.example.com:hostclass":
-   {"Fn::Sub": "TEST-ABC-ALPHA-${airportCode}"}
+   {"Fn::If": [
+     "IsInValidAirportCode",
+     {"Ref": "AWS::NoValue"},
+     {"Fn::Sub": "TEST-ABC-ALPHA-${airportCode}"}
+   ]}
+Conditions:
+ IsInValidAirportCode: {"Fn::Or": [
+  {"Fn::Equals":[{"Ref": "airportCode"}, "KIX"]},
+  {"Fn::Equals":[{"Ref": "airportCode"}, "BAH"]}
+ ]}
```

(b) Configurations with templates

Figure 3: Static vs. Parameterized Hostclass Configuration Patch Examples

Accordingly, IacFix employs *direct template comparison* for the former and *constraint-based template validation* using an SMT solver for the latter.

Direct Template Comparison. When the synthesized templates contain concrete resource values rather than parameterized variables (as shown in Figure 3a), IacFix extracts the set of resource values from the relevant policy sections in both the pre-patch and post-patch templates. It then computes the set difference: the removed set ($V_{pre} \setminus V_{post}$) must exactly match the non-compliant items from the finding, and the added set ($V_{post} \setminus V_{pre}$) must be empty.

Constraint-Based Template Validation. When templates contain parameterized variables rather than concrete values (Figure 3b), direct comparison is insufficient because the actual resource values depend on runtime parameters. IacFix instead uses an SMT solver to formally verify that the patch removes exactly the intended non-compliant items without introducing new ones (Figure 4).

As illustrated in Figure 4, IacFix first extracts the *precondition* under which each resource template is active from both the pre- and post-patch templates. A precondition is a Boolean

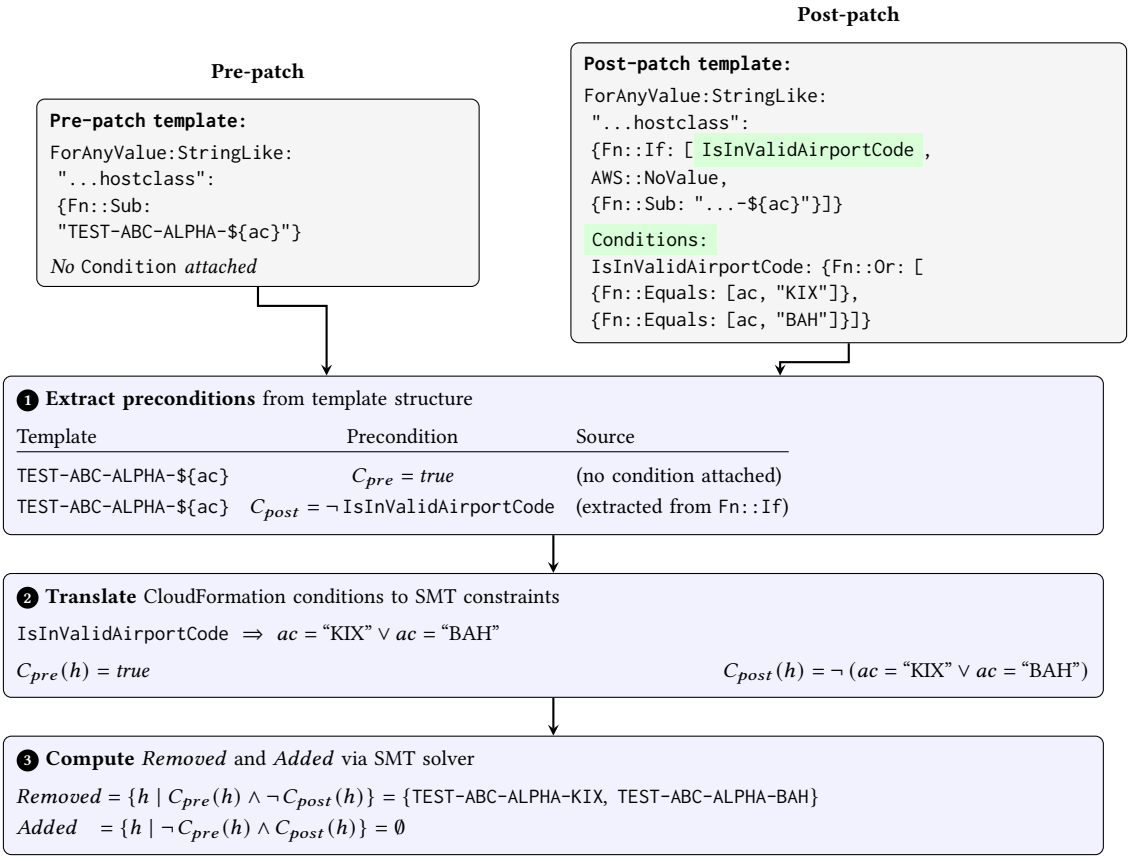


Figure 4: Precondition extraction and constraint-based validation. IacFix extracts the precondition under which each resource template is active from both the pre- and post-patch CloudFormation templates, translates CloudFormation intrinsic functions (e.g., `Fn::Or`, `Fn::Equals`) into SMT constraints, and uses an SMT solver to compute the sets of removed and added resources.

formula over template parameters: under a given parameter assignment, the resource is generated if the formula evaluates to true and is omitted otherwise. For each resource h, let $C_{pre}(h)$ and $C_{post}(h)$ denote its pre- and post-patch preconditions, respectively. IacFix derives these formulas from the conditional structure surrounding the resource. A resource with no attached condition has precondition *true*, whereas a resource inside an `Fn::If` expression inherits either the guard or its negation, depending on which branch contains `AWS::NoValue`. In Figure 4, the template `TEST-ABC-ALPHA-${airportCode}$` is unconditional before patching, so $C_{pre}(h) = true$. After patching, the `Fn::If` expression suppresses the resource with `AWS::NoValue` when `IsInValidAirportCode` holds and retains it otherwise. Therefore, the resource is active only when this condition is false, yielding $C_{post}(h) = \neg$ `IsInValidAirportCode`.

IacFix then translates CloudFormation intrinsic functions into SMT constraints. The condition `IsInValidAirportCode` is resolved to $ac =$ "KIX" $\vee$ $ac =$ "BAH", so $C_{post}(h) = \neg(ac =$ "KIX" $\vee$ $ac =$ "BAH"). Finally, the solver computes the sets of removed and added resources: $Removed = \{h : C_{pre}(h) \wedge \neg C_{post}(h)\}$, $Added = \{h : \neg C_{pre}(h) \wedge C_{post}(h)\}$

In this example, $C_{pre}(h) \wedge \neg C_{post}(h)$ is satisfied when $ac \in$ {"KIX", "BAH"}, yielding $Removed =$ {`TEST-ABC-ALPHA-KIX`, `TEST-ABC-ALPHA-BAH`} and $Added = \emptyset$. The validation succeeds if and only if *Removed* exactly matches the non-compliant items from the resource-misconfiguration finding and $Added = \emptyset$.

To formally verify patch correctness, IacFix defines a translation function $\mathcal{T}[\![\cdot]\!]$ that maps CloudFormation intrinsic expressions to formulae in the quantifier-free theory of strings (QF_S). Template parameters (e.g., `airportCode`) are modeled as free variables of string sort, ensuring verification is sound over *all* possible parameter assignments without requiring knowledge of the deployment-time parameter space. The translation is defined recursively:

$$\mathcal{T}[\![\texttt{Fn::Equals}[x, \text{“}v\text{”}]]\!] = (x = v)$$
$$\mathcal{T}[\![\texttt{Fn::Or}[e_1, \ldots, e_n]]\!] = \mathcal{T}[\![e_1]\!] \vee \cdots \vee \mathcal{T}[\![e_n]\!]$$
$$\mathcal{T}[\![\texttt{Fn::And}[e_1, \ldots, e_n]]\!] = \mathcal{T}[\![e_1]\!] \wedge \cdots \wedge \mathcal{T}[\![e_n]\!]$$
$$\mathcal{T}[\![\texttt{Fn::Not}[e]]\!] = \neg\, \mathcal{T}[\![e]\!]$$
$$\mathcal{T}[\![\texttt{Fn::If}[c, \mathit{val}, \texttt{NoValue}]]\!] \Rightarrow \mathit{active}(h) := \mathcal{T}[\![c]\!]$$
$$\mathcal{T}[\![\texttt{Fn::If}[c, \texttt{NoValue}, \mathit{val}]]\!] \Rightarrow \mathit{active}(h) := \neg\, \mathcal{T}[\![c]\!]$$
$$\mathcal{T}[\![\texttt{Fn::Sub}[\text{“}s_1\texttt{\$\{p\}\$}s_2\text{”}]]\!] = s_1 \cdot p \cdot s_2$$

where $active(h)$ denotes the precondition under which resource h is generated (i.e., not suppressed by `AWS::NoValue`).

Applying this translation to both pre- and post-patch templates yields the preconditions $C_{pre}(h) = \mathcal{T}[\![e_{pre}]\!]$ and $C_{post}(h) = \mathcal{T}[\![e_{post}]\!]$. Given the set of non-compliant items V from the misconfiguration finding, the solver decides: for each $v \in V$, whether the parameter assignment producing v satisfies $C_{pre}(h) \wedge \neg C_{post}(h)$ (confirming removal); and whether any assignment satisfies $\neg C_{pre}(h) \wedge C_{post}(h)$ (detecting spurious additions).

Proposition 1 (Soundness of Exact Resolution). *Let $\mathcal{T}$ be faithful (i.e., each supported intrinsic is translated per the rules above). If the solver reports Removed $= V$ and Added $= \emptyset$, then for every parameter assignment σ over the string domain, the post-patch template generates resource h under σ if and only if the pre-patch template generates h under σ and $h \notin V$.*

Proposition 1 establishes that, under a faithful encoding of the supported CloudFormation intrinsics, the validation component provides *exact* security resolution: it guarantees neither over-removal (revoking legitimate access) nor under-removal (leaving violations unresolved). Within this supported subset, upstream stages may fail, but no patch violating this criterion passes validation or is surfaced to users.

Semantic Correctness Analysis. Security resolution validation confirms that the intended non-compliant items are removed, but does not check whether the patch introduces unintended functional changes (e.g., altering deployment logic or breaking resource dependencies). IacFix therefore employs an AI agent that performs a graded semantic analysis. The agent first analyzes the patch and returns one of three assessments: "yes" (semantically safe), "no" (semantic violations detected), or "maybe" (requires additional context). For "maybe" responses, the agent automatically expands its analysis scope, examining broader code context, resource dependencies, or related configurations, and then provides a definitive assessment. This continues until reaching a definitive assessment or hitting the iteration limit, which escalates to human review.

3.5 Feedback Guided Patch Refinement

When validation fails, IacFix refines the patch using a counter-example-guided approach inspired by CEGIS (Counter-Example Guided Inductive Synthesis) [30]. At each refinement round i, the validator analyzes patch p_i and produces a structured feedback f_i that characterizes the failure. The agent then generates an improved patch p_{i+1} conditioned on both the failed patch and its feedback: $p_{i+1} = Generate(p_i, f_i)$. This loop continues until the patch passes all validation checks or a maximum number of rounds k is reached.

The feedback f_i differs by failure type, with security validation failures receiving the most detailed counter-examples.

Security Validation Failures. For security validation failures, the feedback f_i is a structured counter-example derived from the *Removed* and *Added* sets computed by the validator (Section 3.4). Let V denote the set of non-compliant items from the finding. The counter-example characterizes the error using four sets: $R_i^{actual} = Removed_i$ (items actually removed by p_i), $R_i^{over} = R_i^{actual} \setminus V$ (over-removed legitimate configurations), $R_i^{under} = V \setminus R_i^{actual}$ (under-removed non-compliant items still present), and $A_i = Added_i$ (newly introduced items). This decomposition tells the agent exactly which items were incorrectly handled.

Over-removal ($R_i^{over} \neq \emptyset$) indicates that the patch applies overly broad modifications that eliminate legitimate configurations alongside the intended targets, such as removing the entire template `TEST-ABC-ALPHA-${airportCode}$` in Figure 3b instead of filtering only the specific invalid airport codes. Under-removal ($R_i^{under} \neq \emptyset$) creates "fake fixes" where source code is modified but violations persist in the generated templates. For example, Figure 5 shows a patch that appears to fix the issue in Figure 1, but the counter-example reveals $R_i^{under} = V$ (all dangling hostclasses remain) and highlights the flawed logic from the generation log.

The refinement prompt provides the agent with the failed patch p_i, a summary of its approach, and the full counter-example f_i, enabling targeted correction rather than regeneration from scratch.

```
--- a/lib/stack/rs/rsIamStack.ts
+++ b/lib/stack/rs/rsIamStack.ts
export class ToyIamStack extends DeploymentStack {
    ...
     if (Conditions.isProdStack(stage)) {
       testHostclass.push(`TEST-ABC-XYZ-${airportCode.toUpperCase()}`);
       testHostclass.push(`TEST-ABC-PROD-${airportCode.toUpperCase()}-
       SECONDARY`);
     } else {
+       const isDanglingCombination =
+           (airportCode.toUpperCase() === 'BAH' && (stage === Stage.Alpha
       || stage === Stage.Beta))
+           (airportCode.toUpperCase() === 'KIX' && (stage === Stage.Alpha
       || stage === Stage.Beta))
+           (airportCode.toUpperCase() === 'IAD' && stage === Stage.Beta);
+       if (!isDanglingCombination){
+           if (stage === Satge.Alpha || stage === Stage.Gamma) {
               testHostclass.push(`TEST-ABC-ALPHA-${airportCode.
      toUpperCase()}`);
+           }
+           if (stage === Stage.Beta || stage === Stage.Gamma) {
               testHostclass.push(`TEST-ABC-BETA-${airportCode.toUpperCase
      ()}`);
+           }
+           if (stage === Stage.Gamma) {
              testHostclass.push(`TEST-ABC-GAMMA-${airportCode.
     toUpperCase()}`);
           }
+       }
+   }
    ...
}
```

Figure 5: Incorrect Fix Generated by AI Agent for the Issue in Figure 1

Other Validation Failures. For build failures, the feedback f_i contains the compilation errors, guiding the agent to fix syntax or type issues. For semantic validation failures, f_i includes the functional change analysis from the semantic check agent, identifying specific unintended modifications to avoid. For service-level failures (e.g., LLM throttling), IacFix employs simple retry mechanisms.

4 Evaluation

In this section, we evaluate the performance of IacFix through various experiments. Specifically, we assess IacFix with the following research questions:

- **RQ1 (Effectiveness):** How effectively can IacFix automatically locate, fix, and validate patches for security issues in CDK/CloudFormation packages?

- **RQ2 (Ablation):** What is the contribution of key design choices to IacFix's performance?
- **RQ3 (Efficiency):** How efficient is IacFix in terms of time cost for repairing security issues?

Dataset. We evaluate IacFix on production IaC code to capture real-world complexity in code structure, dependencies, and deployment configurations. We randomly sampled 356 accounts from a production monitoring system that have attached hostclasses and contain CDK/CFN resources. Through our internal deployment pipeline, we mapped these accounts to 85 code packages: 54 CDK packages and 31 CloudFormation template packages. The codebases and deployment mappings are real production artifacts, but the end-to-end benchmark uses controlled dangling-hostclass violations to obtain precise ground truth. We pre-filtered CDK packages, retained all CFN packages, and report repair success over the 47 packages confirmed to contain violations. Historical fix specifications, in contrast, come from naturally resolved production findings.

Evaluation Criteria. We evaluate IacFix using several effectiveness metrics. For Root Cause Localization (RCL), we use standard binary classification metrics where True Positives (TP) represent correctly identified misconfigured code locations, False Positives (FP) are incorrect localizations, False Negatives (FN) are missed violations that exist in the code, and True Negatives (TN) are cases where no violation exists and none is found. For *historical finding identification*, we use standard precision, recall, and F1-score based on manual validation of a random sample. *Overall system performance* is measured by end-to-end success rate.

Implementation. We implemented IacFix in Python 3.12. The root cause localization, patch generation, and semantic check agents are built using Strands Agents 1.1.0 [32]. Historical fix identification and clustering leverage Kiro CLI [3] enabled with internal MCP tools. In all LLM usage, we used `claude-sonnet-3.7`. For constraint-based template validation, we use Z3 [10] as the SMT constraint solver to verify security resolution.

Experiment Settings. We ran IacFix on a server running Amazon Linux 2 (kernel 5.10.239), equipped with 16-core Intel Xeon Platinum 8259CL CPUs @ 2.50GHz, 64 GiB of RAM, and 500 GB NVMe SSD storage.

4.1 RQ1: IacFix Performance

Root Cause Localization Performance. For CDK packages, we used pre-patch `cdk.out` files (the CloudFormation templates synthesized from CDK source code) to identify 33 out of 54 packages that actually contain the designated dangling hostclasses. For CloudFormation packages, we evaluated all 31 packages since we cannot determine violations statically due to their parameterized nature. This filtering process demonstrates our system's ability to exclude packages without actual violations, reducing the evaluation set from 85 to 64 packages. As shown in Table 1, IacFix achieved 91.49% recall (43/47) and 97.7% precision (43/44) across all candidate packages. The high precision indicates that IacFix rarely generates unnecessary patches, while the strong recall demonstrates effective identification of actual security violations localization. Two of the FNs occurred because the violations were defined in JSON configuration files, which fall outside the agent's modification scope that focuses on source code files. The other two localized successfully during refinement.

Historical Fix Identification. To evaluate the effectiveness of our historical fix pattern extraction for dangling hostclass remediation, we analyzed 96 packages with 63 identified historical fixes. We randomly sampled 20 packages with resolved dangling hostclass findings and analyzed 165 commits from these packages. IacFix's commit identification component achieved 88.2% precision (15/17) and 100% recall (15/15) in identifying commits that specifically remediated dangling hostclass violations, with an F1-score of 93.8%. These extracted patterns are used to guide our LLM-based patch generation, where the quality of generated patches directly depends on the correctness of learned remediation patterns. The 2 false positives occurred because: (1) one commit involved architectural refactoring that removed properly implemented hostclass usage rather than fixing a violation, and (2) the other was part of a multi-commit fix where this particular commit only addressed dependencies rather than directly removing the dangling hostclass violation. IacFix extracted 6 distinct fix specifications from the identified security commits. The most common patterns included: (1) removing specific hostclass entries from arrays, (2) adding conditional logic to filter invalid hostclasses, and (3) removing entire related policies.

End-to-End Performance of IacFix. Table 3 presents the comprehensive end-to-end results across all stages of IacFix's pipeline. Overall, IacFix achieved an 80.9% success rate (38/47) when evaluated against ground truth packages that actually contain security violations.

IacFix demonstrated strong performance across both package types. CDK packages achieved a 75.8% final success rate (25/33), improving from 48.5% after initial generation through iterative refinement (27.3% improvement). CFN packages performed even better with a 92.9% success rate (13/14), dramatically improving from 50.0% to 92.9% after just the first refinement round. These consistent improvements across both package types validate the effectiveness of our feedback-guided refinement approach .

All 9 failures were correctly identified by our system: 6 incorrect patches detected by our validator, 2 root cause localization failures, and 1 build failure. The patch-related failures included under-removal (2 cases), both under- and over-removal (3 cases), and pure over-removal (1 case). Crucially, IacFix exhibits one-sided error: across all 64 packages, zero incorrect patches were surfaced to users. Every failure case was either caught by validation and rejected, or the system declined to generate a patch. **Importantly**, our deterministic validation caught 11 incorrect patches in the first generation round that LLM-based validation would have incorrectly approved, preventing potential service failures if these patches had been merged. This demonstrates the critical importance of SMT-based validation guardrails in automated patch generation, where LLM judgment alone proved insufficient for ensuring patch correctness in security-critical scenarios.

As shown in Table 2, patch quality correlates strongly with violation complexity: packages with fewer violation items (1-5) achieved 94.1% perfect patches (32/34), while those with more violations (10+) only achieved 43% (3/7). Packages with numerous violations often led to patches that removed entire code blocks rather than

Table 1: RCL Performance

Package Type	TP	TN	FP	FN
CDK	30	0	0	3
CFN	13	16	1	1
Overall	43	16	1	4

Table 2: Patch Quality Analysis vs. Violation Count[a]

Type	# Violations	# Pkgs	# Over-removed	# Under-removed	# Both	# Perfect
CDK (30)	1-5	22	1	0	1	20
	6-10	2	0	0	0	2
	10+	6	1	0	2	3
CFN (14)	1-5	12	0	0	0	12
	6-10	1	0	0	0	1
	10+	1	1	0	0	0

[a]Analysis includes only packages with successful patch generation and confirmed violations.Three CDK packages excluded in this table: one due to build failure (10+ violation), two failed in localization.

Table 3: End-to-End Patch Generation and Validation Results. Refinement stages process validation failures from previous stage; generation failures are retried only once in the next stage. Security Failed format: Total (Under-removed/Over-removed/Both). Success Rate based on Ground Truth packages.

Type	Stage	Attempted	Generated	Build Failed	Security Failed	Semantic Failed	Success	Success Rate
CDK (GT:33)	Initial	33	30	1	13 (8/1/4)	0	16	48.5%
	Refine @1	17	15	1	9 (5/1/3)	0	5	63.6% ↑
	Refine @2	10	10	1	7 (4/0/3)	0	2	69.7% ↑
	Refine @3	7	7	1	5 (2/0/3)	0	2	75.8% ↑
	Final	**33**	**31**	**1**	**5 (2/0/3)**	**0**	**25**	**75.8%**
CFN (GT:14)	Initial	31	14	0	7 (4/2/1)	0	7	50.0%
	Refine @1	24	7	0	1 (0/1/0)	0	6	92.9% ↑
	Refine @2	2	1	0	1 (0/1/0)	0	0	92.9%
	Refine @3	1	1	0	1 (0/1/0)	0	0	92.9%
	Final	**31**	**14**	**0**	**1 (0/1/0)**	**0**	**13**	**92.9%**
Total (GT:47)		**64**	**45**	**1**	**6 (2/1/3)**	**0**	**38**	**80.9%**

implementing precise conditional logic. While refinement progressively improved patches, it could not always achieve the granularity needed for complex dynamically generated configurations. Patches that survived earlier validation stages all passed the final semantic validation step.

4.2 RQ2: Ablation Study

Existing IaC repair tools either target different IaC technologies [27], require human intervention [23], or address only simple scenarios [12], making direct experimental comparison infeasible. We instead conduct an ablation study to evaluate the contribution of each key component. The baseline variant (LLM agent without domain-specific enhancements) represents the state-of-practice approach of directly prompting an LLM to fix security issues.

We compared four system variants on a randomly sampled subset of 20 packages (10 CDK and 10 CFN) from the packages confirmed with security issues: (1) Baseline: Agent-only patch generation without historical patterns or refinement, (2) + Historical: baseline enhanced with historical fix patterns, (3) + Refinement: baseline with feedback-guided refinement but no historical patterns, and (4) Full System: complete IacFix with both components.

Table 4 reveals significant performance improvements from each component. The baseline achieved only 40% success rate (8/20), while adding historical patterns improved this to 55% (11/20) and refinement alone reached 65% (13/20). The full system combining both components achieved 85% success rate (17/20), demonstrating their synergistic effect.

The improvement was most significant for CFN packages: the full system achieved a 100% success rate (10/10), compared to the baseline's 30% (3/10). CFN repair requires complex conditional logic to exclude dangling hostclasses; while historical specifications provide a necessary structural foundation, they are often incomplete. Our refinement process then iteratively completes these structures upon validation failure. This synergy is vital, as historical patterns provide the initial quality, while refinement enables recovery from early failures.

Table 4: Ablation Study Results on 20 Randomly Sampled Packages

Variant	CDK Success	CFN Success	Overall Success
Baseline	5/10	3/10	8/20
+ Historical	6/10	5/10	11/20
+ Refinement	7/10	6/10	13/20
Full System	7/10	10/10	17/20

4.3 RQ3: Efficiency of IacFix

Table 5 presents the time efficiency analysis of IacFix. IacFix completed the end-to-end process in an average of 7.12 minutes per package, with CDK packages requiring 8.33 minutes and CFN packages 5.90 minutes.

Package build after fixing was the most time-intensive component (2.83 minutes average), with CDK packages taking significantly

longer (3.98 minutes) than CFN packages (1.68 minutes). Root cause localization (2.09 minutes) and patch generation (1.06 minutes) were both efficient across package types. Security validation and semantic analysis required minimal overhead at 0.07 and 1.08 minutes respectively. Note that the reported times represent single-round processing; refinement rounds require similar time overhead per iteration.

Table 5: Time Efficiency Analysis by Package Type (minutes)

Package Type	RCL	Generation	Validation			Total
			Build	Security	Semantic	
CDK	2.23	0.92	3.98	0.08	1.12	8.33
CFN	1.94	1.19	1.68	0.06	1.03	5.90
Average	**2.09**	**1.06**	**2.83**	**0.07**	**1.08**	**7.12**

5 Discussion

In this section, we discuss the practical implications of IacFix. , including productivity improvements, key lessons learned from development and evaluation, and limitations.

Practical Impact. IacFix is deployed within a large cloud infrastructure that manages tens of thousands of IaC packages across hundreds of components, where security misconfigurations carry a direct operational and compliance risk. Manual remediation is a significant bottleneck: industry reports indicate that misconfiguration incidents typically take 186 days to identify and 65 days to contain [34]. IacFix integrates directly into the existing remediation workflow: when a scanner flags a misconfiguration, IacFix automatically localizes the fault, generates a patch, validates it against formal correctness constraints, and surfaces a code review for developer approval—completing end-to-end in 7.12 minutes per package (Section 4.3). The one-sided error property is critical for production adoption: developers can trust that every surfaced patch has passed formal verification, and no incorrect patch is silently merged. This guarantee enabled integration into existing remediation workflows, where IacFix generates validated patches that developers review and apply, rather than requiring them to manually interpret security findings, locate root causes, and craft fixes from scratch.

Lessons Learned. Our development and deployment revealed several insights with implications beyond our specific system. (1) *LLM validation is insufficient for security-critical patches.* LLM-based validation incorrectly approved 55% of the patches that our deterministic validator rejected (Section 4.1). This has broad implications for the growing body of work applying LLMs to security-critical code repair: without formal guardrails, over half of incorrect patches would reach production, potentially introducing new vulnerabilities rather than remediating existing ones. For any organization considering LLM-based automated patching, this result argues strongly for coupling generation with deterministic validation. (2) *Historical fix specifications and iterative refinement are synergistic.* Neither component alone matches the full system (Section 4.2): the combination achieved 85% success versus 55% and 65% individually. Across our corpus, this 20–30 point improvement translates to dozens of additional packages remediated without manual developer intervention. (3) *Package type significantly affects repair difficulty.* CFN packages achieved 92.9% versus 75.8% for CDK. CDK's richer constructs (loops, conditionals, class hierarchies) challenge both fault localization and fix generation, highlighting that as organizations increasingly adopt programmatic IaC frameworks over declarative templates, repair tools must evolve accordingly.

Generalizability, Scalability, and Limitations. The architecture generalizes across misconfiguration types: localization, generation, and validation are misconfiguration-agnostic, and only the SMT correctness constraints require adaptation. Adding a new violation type (e.g., overly permissive IAM policies) requires authoring an SMT constraint template—hours of effort versus building a dedicated fixer. The approach extends beyond our ecosystem: Terraform's plan output provides an analogous intermediate artifact to offer a potential path to broader adoption. For scalability, packages are processed independently to enable parallelization, and cost per package is bounded by refinement iterations. Finally, we also ran the complete IacFix pipeline on naturally occurring scanner findings in production. Due to compliance constraints, however, we used artificially injected security issues in the quantitative evaluation. Our validation currently relies on CFN templates; frameworks without comparable deployment artifacts would require alternative validation strategies.

LLM Generalizability. Although we instantiate all LLM-based roles with Claude Sonnet 3.7, IacFix's methodology is model-agnostic. Existing multi-model studies provide evidence that other capable LLMs can support the same workflow. Feedback-driven repair improves performance across GPT, Claude, DeepSeek, GLM, and Qwen models [9], while verifier-guided refinement improves IaC generation across multiple proprietary and open-weight models [14, 42]. Moreover, incorrect Terraform repairs have been observed at statistically similar rates across GPT-4o, Claude 3.5 Sonnet, and Gemini 2.5 Flash Lite (57.1–71.4%) [2], indicating that the need for external validation is not specific to Claude. Together, these findings suggest that other capable code LLMs can be substituted into IacFix's localize–generate–validate–refine workflow and benefit from its structured feedback. Model choice may affect the number of successfully generated patches, but it does not alter the deterministic exact-resolution criterion enforced by the validator within the supported SMT encoding.

6 Related Work

Root Cause Localization. Accurately locating fix point is crucial for automated program repair, as crash points often differ from actual fix locations. Early approaches include static slicing, dynamic slicing, and program state-based methods like delta debugging, but these are computationally expensive [24, 26, 29]. Spectrum-based methods efficiently locate faults using test coverage information [38], while recent LLM approaches achieve 82.0% F1 score by transforming localization into sequence labeling [41]. However, these methods are not directly applicable to IaC localization, which must trace from high-level account violations to specific code locations across complex deployment pipelines without execution traces or test coverage information.

LLM-Powered Patch Generation. Recent LLM advancements enable automated patch generation for memory bugs [19, 25, 28, 39,

43]. These works typically assume known patch locations or rely on sanitizer reports for root cause analysis. However, automated repair for IaC misconfigurations remains underexplored [12, 23, 27, 35]. Existing approaches either use formal methods with SMT-based repair [27], require human intervention [23], provide only remediation steps [35], or focus on simple scenarios like spelling errors [12]. The most closely related work is InfraFix [27], which normalizes multiple declarative IaC languages into a shared intermediate representation and uses an inferred desired system state to guide SMT-based repair synthesis. In contrast, IacFix starts from deployed security findings that identify non-compliant resources but provide neither the responsible source location nor a complete desired configuration. IacFix therefore performs cross-abstraction fault localization and historical-fix-guided source patch generation. Moreover, InfraFix uses SMT to synthesize repairs, whereas IacFix uses SMT after patch generation to verify that the reported violations are removed without over-removal or newly introduced configurations.

Patch Validation. Li et al. [21] found approximately 12% of security patches are defective, with 7% failing to fully fix security issues while 5% introduce new ones. Validation approaches include static analysis [20], dynamic testing [16, 37], and formal methods [18]. Yu et al. [40] showed that over 40% of test-validated patches are actually incorrect, confirming that test suite-based validation fundamentally overestimates patch correctness. Unlike these test-dependent approaches, IacFix provides formal security guarantees via SMT-based verification without requiring pre-existing test suites.

7 Conclusion

In this work, we introduced IacFix, a multi-agent tool designed to automate the end-to-end process of repairing security misconfigurations in Infrastructure as Code. Using the capabilities of LLMs and enhancing them with specialized agents for fault localization, patch generation, and validation, IacFix combines agentic approaches with formal verification methods. Our rigorous evaluation on real-world IaC packages, including comprehensive ablation studies, demonstrates the effectiveness of our approach in addressing the key challenges of automated IaC security repair. IacFix represents a significant advancement in automated security remediation, providing a practical solution for the growing challenge of cloud infrastructure security management.

Data Availability Statement

Our evaluation uses production IaC code packages from AWS's internal deployment system, ensuring that the results reflect real-world code complexity and deployment configurations. Due to the proprietary nature of the production code and internal security policies governing production infrastructure, the dataset and associated artifacts cannot be made publicly available.

References

[1] Baleegh Ahmad, Shailja Thakur, Benjamin Tan, Ramesh Karri, and Hammond Pearce. 2023. Fixing hardware security bugs with large language models. *arXiv preprint arXiv:2302.01215* (2023).

[2] Manar Alsaid, Chimdumebi Nebolisa, and Faris Abbas. 2026. Empirical Software Engineering TerraProbe: A Layered-Oracle Framework for Detecting Deceptive Fixes in LLM-Assisted Terraform. *arXiv preprint arXiv:2606.26590* (2026).

[3] Amazon Web Services. 2026. Kiro CLI. https://kiro.dev/cli/. Accessed: 2026-07-31.

[4] Clark Barrett and Cesare Tinelli. 2018. Satisfiability modulo theories. In *Handbook of model checking*. Springer, 305–343. doi:10.1007/978-3-319-10575-8_11

[5] Tony Bradley. 2025. How Infrastructure as Code is Revolutionizing Cloud Disaster Recovery. https://www.forbes.com/sites/tonybradley/2025/03/25/how-infrastructure-as-code-is-revolutionizing-cloud-disaster-recovery/. Accessed: August 26, 2026.

[6] Checkmarx. [n. d.]. IaC Security Best Practices: How to Secure Infrastructure as Code. https://checkmarx.com/learn/iac-security/iac-security-best-practices-how-to-secure-infrastructure-as-code/. Accessed: August 26, 2026.

[7] James Cheney, Laura Chiticariu, and Wang-Chiew Tan. 2009. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases* 1, 4 (2009), 379–474. doi:10.1561/1900000006

[8] Configr. [n. d.]. Understanding and Mitigating Misconfigurations in Cloud Computing. https://configr.medium.com/understanding-and-mitigating-misconfigurations-in-cloud-computing-6a25e7932156. Accessed: August 26, 2026.

[9] Dekun Dai, MingWei Liu, Anji Li, Jialun Cao, Yanlin Wang, Chong Wang, Xin Peng, and Zibin Zheng. 2025. FeedbackEval: A Benchmark for Evaluating Large Language Models in Feedback-Driven Code Repair Tasks. *arXiv preprint arXiv:2504.06939* (2025).

[10] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340. doi:10.1007/978-3-540-78800-3_24

[11] Startup Defense. [n. d.]. IaC Misconfiguration Exploitation. https://www.startupdefense.io/cyberattacks/iac-misconfiguration-exploitation. Accessed: August 26, 2026.

[12] Josu Diaz-de Arcaya, Juan López-De-Armentia, Gorka Zárate, and Ana I Torre-Bastida. 2024. Towards the self-healing of Infrastructure as Code projects using constrained LLM technologies. In *Proceedings of the 5th ACM/IEEE International Workshop on Automated Program Repair*. 22–25. doi:10.1145/3643788.3648014

[13] Benny FMO. [n. d.]. What is Infrastructure as Code (IaC) and Why It's Transforming DevOps. https://dev.to/bennyfmo_237/what-is-infrastructure-as-code-iac-and-why-its-transforming-devops-1l2p. Accessed: August 26, 2026.

[14] Prithwish Jana, Sam Davidson, Bhavana Bhasker, Andrey Kan, Anoop Deoras, and Laurent Callot. 2026. TerraFormer: Automated Infrastructure-as-Code with LLMs Fine-Tuned via Policy-Guided Verifier Feedback. In *Proceedings of the IEEE/ACM 48th International Conference on Software Engineering: Software Engineering in Practice*. 578–589. doi:10.1145/3786583.3786898

[15] Hongchao Jiang, Yiming Chen, Yushi Cao, Hung-yi Lee, and Robby T Tan. 2026. Codejudgebench: Benchmarking llm-as-a-judge for coding tasks. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 19416–19448. doi:10.18653/v1/2026.acl-long.888

[16] Wei Jin, Alessandro Orso, and Tao Xie. 2010. Automated behavioral regression testing. In *2010 Third international conference on software testing, verification and validation*. IEEE, 137–146. doi:10.1109/ICST.2010.64

[17] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *Proceedings of the 35th International Conference on Software Engineering*. 802–811. doi:10.1109/ICSE.2013.6606626

[18] Hyungsub Kim, Muslum Ozgur Ozmen, Z Berkay Celik, Antonio Bianchi, and Dongyan Xu. 2023. {PatchVerif}: Discovering Faulty Patches in Robotic Vehicles. In *32nd USENIX Security Symposium (USENIX Security 23)*. 3011–3028.

[19] Youngjoon Kim, Sunguk Shin, Hyoungshick Kim, and Jiwon Yoon. 2025. Logs In, Patches Out: Automated Vulnerability Repair via Tree-of-Thought LLM Analysis. *Usenix Security 2025* (2025).

[20] Xuan-Bach D Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: syntax-and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 593–604. doi:10.1145/3106237.3106309

[21] Frank Li and Vern Paxson. 2017. A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2201–2215. doi:10.1145/3133956.3134072

[22] Ying Li, Faysal hossain Shezan, Bomin Wei, Gang Wang, and Yuan Tian. 2025. SoK: Towards Effective Automated Vulnerability Repair. *Usenix Security 2025* (2025).

[23] En Low, Carmen Cheh, and Binbin Chen. 2024. Repairing infrastructure-as-code using large language models. In *2024 IEEE Secure Development Conference (SecDev)*. IEEE, 20–27. doi:10.1109/SecDev61143.2024.00008

[24] Yisroel Mirsky, George Macon, Michael Brown, Carter Yagemann, Matthew Pruett, Evan Downing, Sukarno Mertoguno, and Wenke Lee. 2023. {VulChecker}: Graph-based vulnerability localization in source code. In *32nd USENIX Security Symposium (USENIX Security 23)*. 6557–6574.

[25] Yu Nong, Haoran Yang, Long Cheng, Hongxin Hu, and Haipeng Cai. 2025. APPATCH: Automated Adaptive Prompting Large Language Models for Real-World Software Vulnerability Patching. *Usenix Security 2025* (2025).

[26] Prashant Hari Narayan Rajput, Constantine Doumanidis, and Michail Maniatakos. 2023. {ICSPatch}: Automated Vulnerability Localization and {Non-Intrusive} Hotpatching in Industrial Control Systems using Data Dependence Graphs. In

32nd USENIX Security Symposium (USENIX Security 23). 6861–6876.

[27] Nuno Saavedra, João F Ferreira, and Alexandra Mendes. 2025. InfraFix: Technology-Agnostic Repair of Infrastructure as Code. In *Proceedings of the 34th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 41–45. doi:10.1145/3713081.3731735

[28] Ridwan Shariffdeen, Christopher S Timperley, Yannic Noller, Claire Le Goues, and Abhik Roychoudhury. 2025. Vulnerability repair via concolic execution and code mutations. *ACM Transactions on Software Engineering and Methodology* 34, 4 (2025), 1–27. doi:10.1145/3707454

[29] Shiqi Shen, Aashish Kolluri, Zhen Dong, Prateek Saxena, and Abhik Roychoudhury. 2021. Localizing vulnerabilities statistically from one exploit. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*. 537–549. doi:10.1145/3433210.3437528

[30] Armando Solar-Lezama. 2008. Program synthesis by sketching.

[31] Spacelift. [n. d.]. Business Benefits of Infrastructure as Code. https://spacelift.io/blog/business-benefits-of-iac. Accessed: August 26, 2026.

[32] Strands. [n. d.]. Strands Agents. https://strandsagents.com/. Accessed: August 26, 2026.

[33] StratusGrid. [n. d.]. IaC Helps Reduce Infrastructure Costs. https://stratusgrid.com/blog/iac-helps-reduce-infrastructure-costs. Accessed: August 26, 2026.

[34] TechUnity, Inc. 2024. Cloud Misconfigurations: The Silent Killer of Enterprise Security. Accessed: August 26, 2026. https://www.linkedin.com/pulse/cloud-misconfigurations-silent-killer-enterprise-security-c44jc/

[35] Dheer Toprani and Vijay K Madisetti. 2025. LLM Agentic Workflow for Automated Vulnerability Detection and Remediation in Infrastructure-as-Code. *IEEE Access* (2025). doi:10.1109/ACCESS.2025.3560911

[36] Yi Wu, Nan Jiang, Hung Viet Pham, Thibaud Lutellier, Jordan Davis, Lin Tan, Petr Babkin, and Sameena Shah. 2023. How effective are neural networks for fixing security vulnerabilities. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1282–1294. doi:10.1145/3597926.3598135

[37] Qi Xin and Steven P Reiss. 2017. Identifying test-suite-overfitted patches through test case generation. In *Proceedings of the 26th ACM SIGSOFT international symposium on software testing and analysis*. 226–236. doi:10.1145/3092703.3092718

[38] Dandan Xu, Di Tang, Yi Chen, XiaoFeng Wang, Kai Chen, Haixu Tang, and Longxing Li. 2024. Racing on the Negative Force: Efficient Vulnerability {Root-Cause} Analysis through Reinforcement Learning on Counterexamples. In *33rd USENIX Security Symposium (USENIX Security 24)*. 4229–4246.

[39] Zheng Yu, Ziyi Guo, Yuhang Wu, Jiahao Yu, Meng Xu, Dongliang Mu, Yan Chen, and Xinyu Xing. 2025. PATCHAGENT: A Practical Program Repair Agent Mimicking Human Expertise. *Usenix Security 2025* (2025).

[40] Zheng Yu, Wenxuan Shi, Xinqian Sun, Zheyun Feng, Meng Xu, and Xinyu Xing. 2026. Patch Validation in Automated Vulnerability Repair. *arXiv preprint arXiv:2603.06858* (2026).

[41] Jian Zhang, Chong Wang, Anran Li, Weisong Sun, Cen Zhang, Wei Ma, and Yang Liu. 2024. An empirical study of automated vulnerability localization with large language models. *arXiv preprint arXiv:2404.00287* (2024).

[42] Tianyi Zhang, Shidong Pan, Zejun Zhang, Zhenchang Xing, and Xiaoyu Sun. 2026. Deployability-Centric Infrastructure-as-Code Generation: Fail, Learn, Refine, and Succeed through LLM-Empowered DevOps Simulation. *Proceedings of the ACM on Software Engineering* 3, FSE (2026), 321–343. doi:10.1145/3797142

[43] Yuntong Zhang, Xiang Gao, Gregory J Duck, and Abhik Roychoudhury. 2022. Program vulnerability repair via inductive inference. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 691–702. doi:10.1145/3533767.3534387

Received 2026-05-01; accepted 2026-07-01