

NSync: Automated Cloud Infrastructure-as-Code Reconciliation with AI Agents

ZHENNING YANG, University of Michigan, USA

HUI GUAN, Amazon, USA

VICTOR NICOLET, Amazon, USA

BRANDON PAULSEN, Amazon, USA

JOEY DODDS, Amazon, USA

DANIEL KROENING, Amazon, USA

ANG CHEN, University of Michigan, USA

Cloud infrastructure is managed through a mix of interfaces—traditionally, cloud consoles, command-line interfaces (CLI), and SDKs are the tools of choice. Recently, Infrastructure-as-Code/IaC frameworks (e.g., Terraform) have quickly gained popularity. Unlike conventional tools, IaC frameworks encode the infrastructure in a “source-of-truth” configuration. They are capable of automatically carrying out modifications to the cloud—deploying, updating, or destroying resources—to bring the actual infrastructure into alignment with the IaC configuration. When IaC frameworks are used together with consoles, CLI, or SDKs, IaC is unaware of changes through these non-IaC interfaces, and the IaC configuration no longer captures the intended state. This is called *infrastructure drift*. IaC frameworks will revert non-IaC changes based on the outdated IaC configuration, leading to misconfigurations or failures.

We propose NSync, an automated system for *IaC reconciliation*, which aims to propagate out-of-band changes back to the IaC program in the form of an update. Our key insight is that infrastructure changes via IaC, consoles, CLI, or SDK eventually all occur via cloud API invocations—the lowest layer for cloud management operations. Hence, NSync gleans insights from API traces to detect drift (i.e., non-IaC changes) and reconcile it (i.e., update the IaC configuration to capture the changes). This is a challenging task—identifying the intended change from low-level, noisy API traces is not easy; moreover, because of the criticality of cloud infrastructure, NSync cannot directly test the synthesized updates in a live environment. NSync addresses these challenges using an agentic design. It infers high-level infrastructure change intent from cloud API sequences with the help of LLMs, and synthesizes targeted IaC updates using domain-specific context management with customized agent tooling; it further maintains an evolving knowledge base of past successful reconciliation runs, reusing prior insights to achieve higher accuracy on future tasks. In addition to system design, we contribute a novel evaluation pipeline for injecting drift into cloud infrastructure and assessing reconciliation attempts, by sourcing scenarios from authoritative cloud operation examples and transplanting them into an IaC-centric framework. Experiments across five real-world Terraform projects and 372 drift scenarios show that NSync outperforms the baseline both in terms of accuracy (from 0.71 to 0.97 pass@3) and token efficiency (1.47× improvement).

CCS Concepts: • **Software and its engineering** → **Cloud computing**; • **Computing methodologies** → **Intelligent agents**; • **General and reference** → *Reliability*.

Authors' Contact Information: Zhenning Yang, University of Michigan, Ann Arbor, USA, znyang@umich.edu; Hui Guan, Amazon, Arlington, USA, huiguan@amazon.com; Victor Nicolet, Amazon, Arlington, USA, victornl@amazon.com; Brandon Paulsen, Amazon, Arlington, USA, bpaulse@amazon.com; Joey Dodds, Amazon, Portland, USA, jldodds@amazon.com; Daniel Kroening, Amazon, Seattle, USA, dkr@amazon.co.uk; Ang Chen, University of Michigan, Ann Arbor, USA, chenang@umich.edu.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2026 Copyright held by the owner/author(s).

ACM 2994-970X/2026/10-ARTISSTA030

<https://doi.org/10.1145/3832121>

Additional Key Words and Phrases: Infrastructure-as-Code, Drift reconciliation, AI Agents

ACM Reference Format:

Zhenning Yang, Hui Guan, Victor Nicolet, Brandon Paulsen, Joey Dodds, Daniel Kroening, and Ang Chen. 2026. NSync: Automated Cloud Infrastructure-as-Code Reconciliation with AI Agents. *Proc. ACM Softw. Eng.* 3, ISSTA, Article ISSTA030 (October 2026), 23 pages. <https://doi.org/10.1145/3832121>

1 Introduction

Cloud management—creating, maintaining, and updating cloud resources—is a crucial task. Management tasks take place through a mix of approaches. Infrastructure-as-Code (IaC) frameworks are gaining popularity: Terraform [27] leads the market, and similar tools like Pulumi [40] and OpenTofu [37] are quickly rising. These frameworks provide declarative high-level specifications of the infrastructure in readable and reusable IaC configurations. This reliable “source of truth” offers DevOps teams a centralized view of resources, ensures reproducibility, and enables code maintenance through version control. In theory IaC frameworks should complement traditional imperative management methods such as cloud consoles, command-line interfaces (CLI), and SDKs; these imperative approaches perform direct mutative actions on infrastructure and enable quick scripting and issue remediation, but do not offer the benefits of a declarative interface [62]. DevOps teams would like the benefits of both: the abstract declarative view of IaC with the easily programmable, mutative power of imperative modifications.

Unfortunately, mixing IaC with imperative interfaces introduces a critical problem: *infrastructure drift*, where the live infrastructure state diverges from the IaC configuration. Drift arises when infrastructure provisioned by an IaC framework is modified via other interfaces. DevOps operators often make imperative cloud changes for rapid response, even in IaC-managed environments. For example, when an urgent permission update is needed, directly attaching a policy to a cloud identity can be faster than locating the IaC module that defines the identity, understanding its composition, and modifying the appropriate declarative description. Such out-of-band updates leave the cloud in a state inconsistent with the IaC specification. In practice, drift has contributed to major security breaches [49], billions of dollars in cloud waste from unmanaged and dangling resources [24], and remediation efforts that can take months to complete [20]. We propose to provide the best of both declarative and imperative worlds by automatically incorporating out-of-band changes into the IaC configuration to consolidate the central view. We call this the *IaC reconciliation* problem.

Currently, IaC reconciliation remains largely manual, as no existing system provides end-to-end solutions for detecting drift and patching the configuration. For *detection*, one class of tools aims at “drift management”, but only identifies changes to resources that are explicitly managed by IaC [25, 47], leaving externally induced drifts unaddressed. Another class of tools goes to the extreme of “lifting” a whole configuration [5, 15, 18, 19, 22, 36] from scratch, by exhaustively scanning the cloud via large numbers of costly API calls to generate the configuration; these tools suffer from limited resource coverage because the scanning rules need to be hand-crafted for each resource [39]. With manual intervention, DevOps engineers need to inspect the live infrastructure, identify discrepancies, and retrofit updates into existing IaC configurations [26]. This manual process is challenging, slow, and error-prone. Since drift management tools such as IaC frameworks cannot automatically discover new resources (e.g., a new subnet created via non-IaC tools), DevOps engineers need to identify these unmanaged resources themselves: this is not scalable and may overlook subtle yet critical changes, leave dangling resources, and can eventually cause failed deployments and misconfigurations. Even if DevOps engineers successfully identify all changes, they must then translate the changes into valid IaC updates. This requires understanding both the structure and logic of the existing codebase, and mastering the correspondence between deployed cloud resources and their declarative IaC representations.

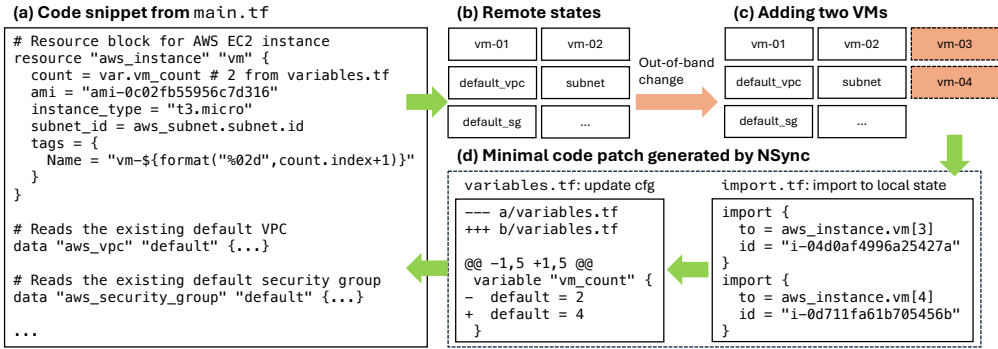


Fig. 1. A simplified IaC example written in Terraform. (a) Code snippets of a Terraform configuration file; (b) Cloud infrastructure provisioned by the configuration files; (c) Cloud infrastructure after out-of-band change that adds two VMs; (d) Code patch generated by NSync

We present NSync, the first LLM-powered agentic system to autonomously perform IaC reconciliation. The key insight behind our approach is to *operate on the cloud API call traces*. RESTful APIs are the lowest-level abstraction for cloud management, and all higher-level modalities (IaC, CLI, SDK, consoles) invoke these APIs to perform tasks [62]. Furthermore, cloud providers offer API monitoring services, such as AWS CloudTrail [3] and Azure Activity Logs [35]. NSync uses this central vantage point to capture complete traces of all infrastructure changes, IaC or imperative. It jointly analyzes API traces and the original IaC configuration to generate code patches that reconcile the drift, while preserving the IaC configuration's structure and abstractions.

To support a thorough evaluation for the task, we developed a novel evaluation framework. We sourced changes from AWS automation runbooks [7], which are a set of predefined API-based actions for typical DevOps tasks. They are then transformed into IaC-based drifts for evaluation. We have applied this methodology to generate 372 drift scenarios across five complex, real-world Terraform projects, ranging from tens to over a thousand resources. Our evaluation shows that NSync can successfully reconcile the IaC codebase, achieving 0.95 pass@3 accuracy, outperforming the off-the-shelf Claude agent (0.71) while being 1.72× more token-efficient. Continuous learning further improves the accuracy to 0.97 pass@3, while remaining 1.47× more efficient than baseline solutions. Under the stricter pass@1 metric, NSync achieves an average accuracy of 0.80, outperforming the baseline (0.49). To summarize, this paper makes the following key contributions:

- We present a new task: *trace-driven IaC reconciliation*, and devise a novel solution that gleans insights from cloud API call traces to repair IaC configurations.
- We develop NSync, an agentic system that reconciles out-of-band infrastructure drift from API traces, substantially outperforming the baseline.
- We propose an evaluation pipeline for generating realistic, assessable drift scenarios and curate the first IaC reconciliation dataset with 372 validated cases across five diverse Terraform projects.
- We perform a detailed empirical study to assess the effectiveness and robustness of our system across diverse drift scenarios.

We have open-sourced our implementation and benchmark at <https://github.com/zyang37/nsync>.

2 Motivation

To further motivate the problem, we provide background on IaC frameworks, and discuss typical DevOps engineers' workflows, how drift happens, and why existing tools are insufficient.

2.1 IaC and Terraform

IaC frameworks operate on a declarative view of the resources in IaC configurations. These configurations are compiled to cloud APIs that eventually modify the infrastructure. We focus on Terraform, which is the dominant IaC solution with 62% market share as of 2025 [19]. Cloud infrastructure has three forms of representation. (1) In Terraform, DevOps engineers write *configuration files* (`.tf`), which is the *declarative state* that defines the desired infrastructure. Terraform codebases are typically structured like large software projects, with resources distributed across multiple files and directories. They also make extensive use of custom modules, which function as reusable building blocks that can be composed hierarchically. (2) After deployment, Terraform produces a *local state* file (`.tfstate`) recording the IaC-managed resources (e.g., compute instances) and their current attributes as retrieved from the cloud (e.g., machine image, ID). This is the *runtime state from the IaC point of view*, different from *configuration files*. (3) Finally, the cloud does not have a central view of the infrastructure state, but exposes resource-specific APIs to retrieve the *remote state* per resource.

A typical deployment workflow could be a DevOps engineer invoking a dry-run command: `terraform plan` to review changes, followed by `terraform apply` to modify the cloud. IaC as “source-of-truth” is one of its most valued properties, assuming that the infrastructure is entirely managed by IaC. Consider the example in Figure 1(a), which gives a Terraform configuration snippet (`main.tf`)—the declarative state. This configuration provisions a number of VM instances specified by `var.vm_count`, along with other networking components such as a Virtual Private Cloud (VPC), subnets, and a security group (SG) for the VM instances. Figure 1(b) illustrates the cloud’s remote state after deployment. Unlike IaC, the cloud lacks a central state file; each resource exposes its own state, yielding a decentralized and fragmented view. In order to construct the remote state, one has to use state retrieval APIs at the resource level—i.e., one has to know that a certain VM exists before calling state retrieval APIs on that VM.

2.2 Infrastructure Drift and IaC Reconciliation

Operations are easier if DevOps engineers only use IaC frameworks to manage their infrastructure. However, in real-world operations, they often make *valid* changes outside the IaC workflow via consoles, CLIs, or SDKs. Our interviews with a major cloud provider reveal several representative situations where engineers bypass IaC workflows, which can introduce drift. (1) *Incident response*: when troubleshooting production issues, engineers need to apply fixes immediately. To act quickly, engineers bypass IaC and use faster API-driven tools such as AWS Systems Manager Automation [4] or Azure Sentinel Playbooks [33]. (2) *Performance tuning*: engineers frequently experiment with configuration parameters (e.g., scaling factors). While such modifications can be performed with IaC, doing so requires writing and deploying code for every trial, which is far more cumbersome than clicking a button in a dashboard or running a single CLI command. To iterate quickly, they bypass IaC and modify resources interactively. (3) *Debugging and observability*: during live debugging, engineers often enable additional resources such as logging, tracing, or temporary dashboards. IaC provides little support for live diagnostics, so these are typically set up directly through consoles or APIs. In all three cases, drift arises when these changes are not synchronized in the IaC codebase.

When drift occurs, without direct operational contexts, the IaC team may miss relevant updates, struggle to identify which resources need to be managed and which are temporary, and face difficulties in determining the complete scope of resource changes across different cloud services. Unfortunately, there is no fully automated solution to address the IaC reconciliation problem today.

2.3 Limitations of Existing Tools

Prior work has explored automated program repair in other domains, such as system configurations. For instance, some of these projects [42, 56] infer the desired state from local system calls and use SMT-based synthesis to repair programs; these target a small and relatively stable set of file-system operations with well-defined semantics. In contrast, cloud platforms expose thousands of APIs across hundreds of resource types, with semantics that vary across services and evolve over time [48]. Encoding and maintaining symbolic repair rules for this space is, therefore, substantially more difficult.

A related class of tools is IaC lifting [5], which synthesizes IaC by scanning live cloud state and regenerating configurations. However, lifting tools are not drift-aware and do not solve the *reconciliation* task: 1) they translate cloud state from service API responses into flat resource definitions, creating maintenance challenges; 2) similar to prior symbolic approaches, they do not produce patches over an existing codebase, discarding the abstractions and conventions.

To illustrate the cost of ignoring this structure, we analyze five large, real-world AWS infrastructures deployed using user-written IaC. We selected `aws2tf` [5] because it has broad AWS resource coverage [39], but even so, we found its full-cloud scans (using handcrafted scanning rules) to be brittle, frequently failing due to resource coverage, API changes, and rate limiting. To obtain usable results, we restricted it to only scan supported resources. Table 1 shows that lifting cloud state from scratch substantially inflates the generated configuration, increasing LoC by more than 200% in most projects. Worse, lifting regenerates flat resource definitions from cloud state, duplicating shared components, inlining raw cloud resource identifiers instead of reusable variables or module abstractions. The resulting IaC program is therefore difficult to reuse, parameterize, and maintain. The ideal solution should address drifts through precise modifications to the existing codebase, to provide higher usability. Additional results are provided in the supplementary material.

Existing lifting tools inherently lack drift awareness: even localized drift can require re-synthesizing an entire dependency subgraph rooted at the affected resource, pulling in unrelated resources, inflating the codebase, and severely damaging maintainability. This is unsuitable for production IaC, where practitioners organize configurations using modules, hierarchical abstractions, shared variables, naming conventions, and project-specific patterns. Repairing drift requires integrating changes into arbitrary, user-written code structures, rather than editing flat attributes or regenerating standalone resource definitions. A purely symbolic approach would require rules for each cloud API, resource type, codebase organization, and abstraction pattern. We instead propose an API trace-driven approach that uses LLMs to reason over observed cloud API behavior, source code, documentation, and project context, producing reconciliation patches that preserve the existing IaC codebase structure.

3 Problem Formulation and Challenges

We cast IaC reconciliation as a *program repair* task for cloud management, which presents novel challenges. In traditional program repair, typically a specification or a set of test cases is given as explicit instructions for the repair. In the cloud, the repair instructions are implicitly captured in the API traces. The repair task needs to first perform *intent identification*, summarizing long and noisy cloud API traces into a concise description of the change. After identifying the intent, the *patch generation* step is another departure from conventional program repair. The stakes of

Table 1. Lifting tools cause severe LoC inflation, creating substantial challenges for code maintenance.

Projects	Lifted LoC	NSync LoC
lab12 [50]	1053 (↑254%)	317 (↑6.9%)
flask [43]	6095 (↑663%)	828 (↑3.8%)
ssm3 [28]	6144 (↑1529%)	413 (↑9.7%)
live-score [17]	5986 (↑278%)	1597 (↑1.0%)
mega-mesh [41]	24071 (↑239%)	7141 (↑0.8%)

cloud operations are high; NSync cannot push a synthesized patch into a live deployment to test correctness. Rather, it has to perform static evaluations, using a set of ReadOnly tools to assess the match; this further requires nuanced reasoning across both APIs and IaC, which is difficult to capture symbolically. We now describe the three key challenges and our insights and solutions.

Challenge #1: Inferring Intent for IaC Reconciliation. As mentioned in Section 2.3, IaC lifting tools fundamentally lack drift awareness and cannot perform reconciliation. Modern cloud platforms provide auditing services that record *API-level* infrastructure modifications, regardless of the interface used to perform them. Examples include AWS CloudTrail [3], Azure Event Hub [35], Google Cloud Audit [21], and Alibaba ActionTrail [2]. However, accurately and efficiently inferring intent from API traces remains challenging. First, the traces contain substantial irrelevant noise, such as read-only operations, retry attempts, and user-specific account information. Second, API structures vary significantly across different cloud services, using inconsistent naming conventions and parameter formats, making uniform analysis difficult; worse, vendors update the APIs very frequently [48]. Third, auditing services can populate API events out of order [6, 34] due to asynchronous collection across services and regions and low timestamp resolution. Hence, the system must reason about the semantics of cloud operations, infer the resulting infrastructure state from mutating API actions, and tolerate imprecise event ordering in audit logs. To address this challenge, we introduce an *Intent Identification* stage (Section 4.1) that preprocesses API traces by annotating and consolidating noisy low-level events into a small set of persistent drifts requiring reconciliation. Conceptually, this stage acts as a multi-step filtering pipeline over raw cloud API traces.

Challenge #2: Generating IaC Patches from Intent. Automatically bringing updates into an existing IaC codebase is challenging. For example, when an operator manually attaches a policy to a cloud identity, the reconciler must determine where this change belongs in the codebase. Adding the policy directly to the affected identity may duplicate existing abstractions, while modifying a shared module may grant the same permission to every identity that instantiates the module. Thus, generating a correct patch requires reasoning about cloud state, repository structure, resource ownership, and cross-resource dependencies. A practical solution must identify the drift, then adapt to user-written IaC codebases with arbitrary style and determine where the corresponding changes should be made within the existing code structure. We therefore use an LLM-based agent that can inspect the current cloud/IaC context, generalize across resource types, and synthesize reconciliation patches. However, we need to conquer two key barriers: (1) IaC patches cannot be safely executed for validation, as live testing is destructive and dry-run tools like `terraform plan` provide only partial feedback; and (2) IaC repositories can be large, while only a small fraction is relevant to any given drift—a naïve approach will overwhelm the LLM’s context window. These constraints sharply contrast with conventional code-generation agents, which rely on bounded contexts and rich execution feedback for iterative refinement [11, 32, 45, 65]. Without execution feedback, NSync relies on custom, read-only tools that we designed to check IaC syntax and preview prospective changes, enabling safe operation without modifying live infrastructure.

Challenge #3: Leveraging LLMs Efficiently for IaC Reconciliation. A third challenge lies in how to efficiently leverage LLMs for reconciliation. Traditionally, adapting LLMs to specialized tasks relies on fine-tuning, which is resource-intensive and poorly suited to cloud infrastructure operations. Since cloud providers and IaC frameworks evolve APIs and semantics rapidly [48], maintaining accuracy through fine-tuning would require frequent retraining. Recent research explores lifelong learning for agents, where systems continuously accumulate knowledge and improve through repeated interactions with their environment [51, 53]. These approaches have shown promise, but almost all operate in sandboxed domains, such as simulated games or controlled benchmarks, where agents can freely explore, fail, and recover at little cost. Cloud infrastructure presents a fundamentally harder setting: cloud infrastructure does not provide a

safe environment for experimentation and learning. Our approach to enabling continual learning is based on the observation that IaC reconciliation is a repeated process. Cloud infrastructure is long-lived, and agents may repeatedly encounter similar drift patterns as it evolves. Even without free exploration, knowledge from past reconciliation can be carried forward and reused. Specifically, the correspondence between API traces and IaC structures, or lessons from failed attempts, need not be rediscovered each time. Similarly, a codebase’s structure and conventions, once learned, can be reused across future reconciliations for the same infrastructure.

Together, these three challenges motivate the design of NSync, an agentic system that makes IaC reconciliation safe, efficient, and continually improving over time.

4 Design of NSync

NSync is an agentic system that uses API call traces that record cloud infrastructure events, and updates the original IaC configuration to reflect the current infrastructure state.

Figure 2 illustrates the architecture of NSync. The **intent identification** component analyzes noisy API call traces from the cloud provider to determine if there are any infrastructure changes, and analyzes the nature of those changes. When infrastructure changes are detected, the second component, **patch generation**, synthesizes an update with the help of specialized IaC reconciliation tools, and is given access to the source configuration, the identified change intent, and a self-organized evolving knowledge base per IaC project. It outputs updated configurations that accurately reflect the current state of the infrastructure, ensuring alignment between the IaC code and the actual deployed resources. Finally, our agent uses **continuous learning** to record experience to achieve better performance over a longer time horizon.

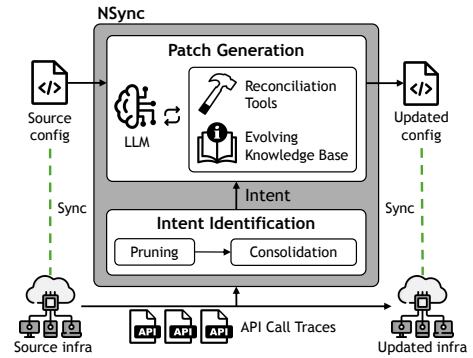


Fig. 2. Overview of NSync.

4.1 Intent Identification

This step identifies the infrastructure change intent from raw API traces as shown in Figure 3. Intent identification focuses on the content and relationships of events rather than their raw sequence in the trace. Before intent identification, we preprocess the traces to extract a clean sequence of mutating events—API calls that alter infrastructure state. We discard read-only calls (e.g., `Describe*`, `List*`), deduplicate retries by keeping only the final success, and remove irrelevant fields such as timestamps or request IDs, thereby reducing trace volume while preserving all state-changing operations. The intent identification process then proceeds in two stages: *annotation* and *consolidation*.

The key idea is to consolidate the API events and identify the corresponding IaC-level resource blocks. Recall that imperative API calls eventually trigger changes on cloud states, which need to be encoded as IaC configuration updates. Inferring IaC resource-level changes (create, delete, update) from the API calls is not easy. For instance, while the `CreateVpc` call corresponds to creating a new VPC block, a `CreateTags` call on a VPC does not create a new resource but rather updates the VPC. To capture such distinctions, we normalize API calls into a consistent schema using an LLM.

4.1.1 Step 1: Annotation. Our key contribution in this step is a neurosymbolic annotation procedure that uses LLMs to produce consistent, schema-aligned labels for heterogeneous API calls. We design a fixed schema that the LLMs must adhere to; further, we encourage consistent labeling across

Table 2. Annotation schema for API calls. Fields not applicable are set to null.

Field	Description
category	Operation type: create, delete, attach, detach, update or unknown
type	Primary resource type (e.g., Vpc, Instance) for create/delete/update
id	Primary resource identifier (e.g., vpc-123, i-456), else null
type1	First resource type in attach/detach relation, else null
id1	First resource identifier in attach/detach relation, else null
type2	Second resource type in attach/detach relation, else null
id2	Second resource identifier in attach/detach relation, else null

event categories and resource types using two mechanisms: (1) the annotator maintains memory (T) of previously inferred resource types to ensure coherence across a trace, and (2) annotation is executed in batches with retries to guard against occasional LLM errors.

Table 2 shows the standard schema that we define, which labels each API call into one of five canonical categories: *create*, *delete*, *update*, *attach*, and *detach*. Each annotated event captures only the essential information: operation type, resource type, and unique resource identifier(s). For example, an EC2 instance launch (RunInstances) is represented as *create*(instance, i-1234), while a volume attachment becomes *attach*(volume-1, instance-2).

Our LLM annotator reasons about the semantics of an event in context. For example, CreateTags may look like a new creation, but it is actually an *update*, and the target of the update depends on parameters—adding a tag could modify a VM, a VPC, or another resource type. Likewise, AuthorizeSecurityGroupIngress appears to authorize something new, but in fact it updates the configuration of an existing security group. By contrast, RunInstances might seem to update the state of a VM, yet it actually provisions a new instance and should be classified as a *create*. These examples show why simple regex or keyword matching would be brittle. LLMs, on the other hand, having been trained on cloud documentation including API usage examples and descriptions, understand these nuances; moreover, because cloud APIs continue to evolve, LLMs can adapt to these changes. To improve scalability, we annotate API calls in batches rather than individually. Algorithm 1 takes API calls C and processes them in batches B of size b , using the schema S and previously collected resource types T as context. For each batch, the LLM attempts up to r retries to produce annotations $\hat{B}$. The results are merged into the annotation set A , and the newly discovered types from $\hat{B}$ are added to T for subsequent batches.

Algorithm 1: LLM-Based API Annotation

Input: API calls C , schema S , batch size b , retries r

Output: Annotated calls A

Init:

$A \leftarrow \emptyset$ (annotations);

$T \leftarrow \emptyset$ (resource types);

foreach $B \subseteq C$, $|B| = b$ **do**

$k \leftarrow 0$;

while $k \leq r$ **do**

$\hat{B} \leftarrow \text{LLM}(B, S, T)$;

if $|\hat{B}| = |B|$ **then**

$\perp$ break

$k \leftarrow k + 1$;

if $|\hat{B}| \neq |B|$ **then**

$\hat{B} \leftarrow \{\emptyset\}^{|\hat{B}|}$

$A \leftarrow A \cup \text{MERGE}(B, \hat{B})$;

$T \leftarrow T \cup \text{TYPES}(\hat{B})$;

return A ;

4.1.2 Step 2: Consolidation. After annotation, each API call is in standardized form (e.g., *create*(instance, i-1234) or *attach*(volume-1, instance-2)), giving us a labeled trace of mutating events. Reconciliation only requires reasoning about *persistent drift*, i.e., the infrastructure changes that remain at the end of the trace. Relative to that goal, the trace still contains noise: there may be many transient operations, such as redundant or overwritten updates, or creation of temporary resources followed by deletion. Consolidation identifies the set of persistent drifts that need to be reconciled in IaC.

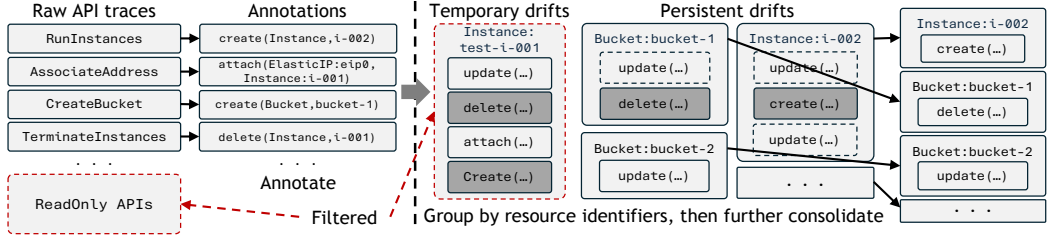


Fig. 3. Intent identification. Raw API traces are first filtered to remove read-only and irrelevant events, then annotated into canonical operation types. The annotations are grouped by resource identifiers and consolidated to remove transient operations, yielding the persistent drifts that must be reconciled in IaC.

Given an annotated API trace, we first organize the events by their resource identifiers. An identifier is a unique number generated by the cloud when a resource (e.g., VM) is created, and remains consistent throughout its lifecycle. For each *node identifier*, we collect all events that occurred on this node/resource—e.g., subnet-456: [create, update] means that a unique ID subnet-456 was created and then updated. For operations that involve a dependency or association between two resources, we group events by the ordered pair of resource identifiers, written as $\langle id1, id2 \rangle$. For instance, $\langle volume-1, instance-2 \rangle$: [attach, detach, attach] denotes that the volume volume-1 was attached to, detached from, and then reattached to instance-2. We then apply the following reduction rules independently to each node or edge identifier:

- **Persistent Create.** If a resource is created and never deleted, we retain only the create event and drop intermediate updates. Because reconciliation targets the final infrastructure state, subsequent updates are implicitly captured in the resource's current state, which can be retrieved through read-only cloud APIs given the resource type and identifier.
- **Persistent Delete.** If a resource was deleted, we retain only the delete and remove all associated updates prior to the deletion. Deletes, like creates, are higher-precedence persistent drifts, as they eliminate resources entirely from the infrastructure.
- **Persistent Update(s).** If a resource was only updated (with no create or delete), we conservatively retain the update. This ensures that updated attributes, even if they are not defined in the original IaC configuration, will be explicitly captured.
- **Persistent Attach/Detach.** Balanced attach/detach pairs cancel out, while any net imbalance is retained as a conservative signal of persistent drift.

The output of this reduction is a compact set of persistent drifts, organized by resource and edge IDs, and it is deliberately conservative to avoid discarding meaningful information.

4.2 Patch Generation

Generating IaC patches with LLMs presents unique challenges compared to traditional automated program repair (APR). In APR, repair correctness can be validated through test execution and runtime signals. However, direct execution of a patched IaC project is not feasible as it would modify live infrastructure. Furthermore, agents may hallucinate during long reconciliation sessions [31]. To mitigate this, we introduce two agent tools: *drift_report*, which is a safe, read-only operation that previews the alignment between the current patch and the cloud state, and *self_critique*, which prompts the agent to assess accumulated edits and API traces for correctness and necessity [32, 45]. Self-critique helps reduce hallucinated changes, scope creep, and untracked divergence from the IaC codebase. The patch generation workflow in NSync is an iterative patch-evaluate-refine loop, shown in Figure 4. This mirrors the trial-and-error workflow of program repair agents, but here it is realized entirely through safe, read-only operations tailored to IaC.

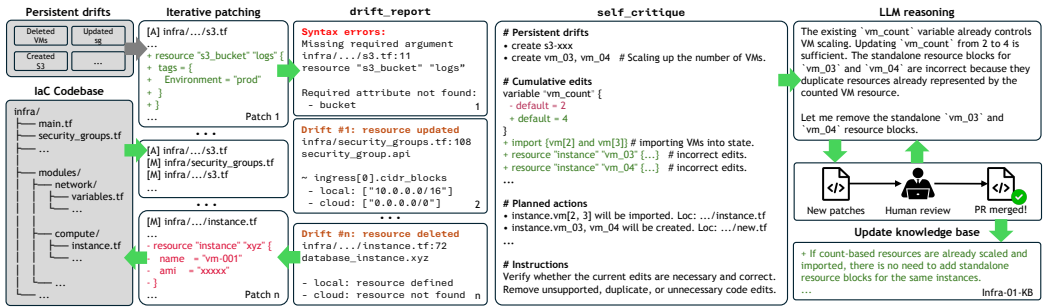


Fig. 4. Tool use during reconciliation. The agent can invoke `drift_report` and `self_critique` multiple times during iterative patching. The stacked `drift_report` boxes show outputs from separate calls after different patch attempts, detecting a syntax error, drift after a resource update, and issues after a resource deletion. These reports provide feedback for subsequent edits. The agent can also invoke `self_critique` to validate accumulated changes against API traces and guide patch refinement before human review.

4.2.1 Drift Report. Terraform plan is not suitable for patching because it assumes the outdated IaC configuration is the “source of truth” and generates update plans that propose reverting cloud-side changes to match stale configurations. Worse, Terraform plan exposes attribute-level differences for each managed resource, including runtime fields marked as “known after apply,” resulting in verbose outputs that inflate context length and overwhelm LLMs. This combination of misleading guidance and excessive verbosity effectively poisons the agent’s reasoning, pushing it toward incorrect or irrelevant patches. `drift_report` provides detailed, reconciliation-oriented feedback. It compiles the configuration and issues read-only cloud API calls without mutating cloud resources, enabling the detection of both syntax errors and drifted resources. Figure 4 shows representative outputs. This targeted feedback helps NSync generate and refine patches more effectively. The agent analyzes the condensed API trace to create an initial patch, then uses `drift_report` in an iterative patch-evaluate-refine loop to improve the solution. `drift_report` treats all detected infrastructure changes as candidates for reconciliation into the IaC patch. This does not make the reconciliation final: the patch is still reviewed by a human through the pull-request process. At that stage, DevOps engineers can decide whether a detected change should be preserved in IaC or reverted by editing or rejecting the corresponding patch.

4.2.2 Self Critique. Another challenge in the agentic workflow is maintaining focus across multiple edits [31]. Prior work suggests that reflection can improve agent reliability [32, 45, 65], but existing methods depend on execution feedback or test cases—signals unavailable in IaC reconciliation, where live execution is unsafe. We design `self_critique`, a safe reflection mechanism that requires no execution and operates at a coarser granularity than `drift_report`. Whereas `drift_report` helps the agent verify whether a candidate patch has fully reconciled the infrastructure by checking if any drift remains, `self_critique` periodically reviews the accumulated edits and prompts the agent to reason about whether its progress still aligns with the drift intent. By surfacing reconciliation at this higher level, the tool enforces consistency and helps the agent stay focused on the reconciliation objective. Figure 4 demonstrates how the agent can iteratively invoke `drift_report` and `self_critique` during reconciliation to diagnose persistent drifts, validate edits against API traces, and refine patches before merge.

4.3 Continual Learning

Cloud infrastructure is long-lived, and reconciliation tasks repeatedly arise. The lack of memory would force the agent to repeatedly rediscover mappings between API traces and IaC resources,

relearn project-specific conventions, and re-synthesize fixes for error cases it has already encountered. To overcome this limitation, we introduce a lightweight, domain-specific knowledge base (KB) that enables continual learning [51, 53] across reconciliation sessions. The KB captures information about prior reconciliations, such as effective patching strategies, frequently observed drift patterns, and project-specific semantics (e.g., naming conventions, module structures, or dependency layouts). This KB is continuously updated after a reconciliation task completes; during the next reconciliation, the agent retrieves relevant knowledge on demand to guide patch generation, thereby grounding its reasoning in accumulated experience rather than starting from scratch.

We design the KB using the following principles. (1) Updates are *selective*: only reconciliations that succeed contribute new entries, while failed sessions are discarded to avoid propagating errors. (2) The KB is *scoped per project* rather than shared globally, since much of the knowledge (e.g., naming conventions, module structure) is project-specific; this prevents cross-project contamination while allowing repeated reconciliations on the same project to benefit from accumulated insights, shown in Figure 4. The learning process is *agent-driven*: the agent invokes the tools `knowledge_update` and `knowledge_retrieval` when it judges that the current context is complete and accurate enough to warrant persistence for reuse, as opposed to always logging every intermediate action. Constructing a useful KB entry requires the complete and up-to-date context of a reconciliation attempt; otherwise, the agent risks recording incomplete or misleading knowledge. We employ a ReAct-style [64] prompting strategy that enforces an *Observation–Thought–Action* loop on every iteration, instructing the agent to ground its reasoning in the latest observations before deciding whether to update or retrieve knowledge. For example, if the agent repeatedly fails to apply a patch because an S3 bucket update requires adjusting its versioning setting, once it discovers the correct fix it can invoke `knowledge_update` to record this resolution for future runs. Over time, this mechanism enables a form of experience-guided reconciliation, where the agent progressively adapts to each IaC project.

The KB is a lightweight, project-specific text file. It can be extended with vector database [30] support for larger or more complex knowledge. The `knowledge_retrieval` tool queries the KB to access previously recorded reconciliation insights, while `knowledge_update` writes new successful reconciliation experiences to the KB. This lightweight implementation ensures that knowledge persistence and retrieval remain simple yet effective, while maintaining the per-project scoping principle. The agent edits or discards outdated KB entries when previously successful strategies fail under evolving external systems, ensuring that knowledge remains accurate over time. As shown later in Table 5 and Figure 9, continual learning improves reconciliation accuracy by reusing prior experience about tool usage, API-to-IaC mappings, import formats, and project-specific code organization.

5 Evaluation

We evaluate NSync on drift-reconciliation tasks across real-world Terraform projects. This section first describes the evaluation protocol (Sec 5.1) used to inject drifts and validate generated patches. We then explain how we construct the benchmark dataset (Sec 5.2), summarize the experimental setup and metrics (Sec 5.3), and finally present the main results and analysis (Sec 5.4).

5.1 Evaluation Protocol

The same infrastructure can be represented by many different IaC codebases. Thus, directly comparing Terraform code is not a reliable way to compare infrastructure. But Terraform maintains a structured state that encodes its current belief about the live infrastructure. Building on this mechanism, we design a Terraform-centric evaluation protocol. The protocol mirrors the information available to a DevOps engineer during drift reconciliation, while preserving a hidden oracle for automated grading. The *base workspace* contains the information visible to the agent: the original Terraform project directory, its Terraform state, and the raw CloudTrail trace of out-of-band API

Table 3. Benchmark statistics: configuration size, mutation space, and event counts.

Benchmark	#Resources	LoC	#Scenarios	#API Events		#Mutating Events	
				Avg	(Min, Max)	Avg	(Min, Max)
lab12 [50]	47	289	96	183.2	(90, 441)	9.5	(1, 47)
flask [43]	74	799	94	280.8	(158, 671)	10.0	(1, 56)
ssm3 [28]	66	377	103	268.0	(149, 737)	8.0	(1, 48)
live-score [17]	193	1582	61	612.3	(373, 1163)	8.5	(1, 42)
mega-mesh [41]	1930	7087	18	3735.9	(3692, 3920)	7.8	(1, 31)

events. The *drift workspace* is used only by the evaluation harness to inject drift and construct ground truth; it is never exposed to the agent. Evaluation proceeds as follows:

- (1) **Deploy the base infrastructure.** We deploy the base Terraform project, creating real cloud infrastructure and generating the initial Terraform state in the base workspace.
- (2) **Inject drift.** We copy the initial Terraform state from the base workspace into the separate drift workspace. Terraform executes the required delta and updates the drift workspace state from the initial state to a drifted state, which is later used as ground truth for evaluation.
- (3) **Collect traces.** We use CloudTrail to collect the resulting raw API events. The agent is given only the base Terraform project directory, the base workspace state, and the CloudTrail trace, and is asked to generate a patch to the IaC configuration.
- (4) **Evaluate the patch.** We evaluate the patched configuration with `terraform plan` against the drift workspace state, which serves as the ground-truth infrastructure state after drift injection.
- (5) **Reset the infrastructure.** After evaluation, we apply the initial Terraform configuration against the ground-truth state, allowing Terraform to compute and execute the revert actions needed to restore the base configuration before evaluating the next reconciliation task.

Patch correctness is evaluated by comparing the patched configuration against the ground-truth infrastructure state with `terraform plan`. First, resources introduced by drift, which are absent from the agent’s observable Terraform state, must be defined in the patched configuration and importable into the base workspace state through `import` actions. Second, existing Terraform-managed resources must match their desired state: the patched configuration and state must be consistent, and any resources deleted by drift must be removed from the configuration.

Concretely, if `terraform plan` reports no changes other than the required `import` actions, the patch is considered correct. Any additional additions, updates, or deletions indicate that the patch has not been fully reconciled yet. One edge case arises from Terraform resource addresses. A resource address is a Terraform-language identifier for a managed resource, separate from the identity of the underlying cloud resource. Since Terraform state records resource addresses, an agent may generate a correct resource definition for a newly introduced cloud resource but assign it a different address from the one used in the ground-truth state. In such cases, `terraform plan` may report a deletion of the ground-truth address and a creation of the agent-generated address, even though both definitions refer to the same cloud resource. Treating this plan as incorrect would produce a false negative. To handle this case, we add a post-processing check over planned actions. When the plan contains a delete–create pair with the same resource type, we compare the corresponding resources at the attribute level. If the attributes match, we treat the pair as a spurious address mismatch and filter it from the plan. After this filtering step, the patch is considered correct if no planned actions remain beyond the required `import` actions.

5.2 Benchmark Dataset

Our benchmark construction procedure consists of four steps: selecting base deployable Terraform projects, sampling drift scenarios, creating project-specific Terraform mutations conditioned on

those scenarios, and validating the mutations through deployment and reset. Because the benchmark operates on real cloud infrastructure, we use explicit selection criteria, manual expert review, and deployment-and-revert validation. Below, we describe how projects and drifts were sampled, how expert validation was conducted, and how failed candidate cases were excluded.

Base project selection. Setting up the evaluation required live cloud infrastructure; we sourced candidate projects from TerraDS [10], a dataset of Terraform code snippets, and traced snippets back to their original public repositories because the snippets themselves were small and not directly deployable. From the recovered repositories, we sampled 100 candidates that appeared to provision at least 30 cloud resources. Many were forks, near-duplicates, severely outdated, depended on unavailable resources, or were not deployable without substantial rewriting. A DevOps expert spent over 40 hours deduplicating, repairing, and test-deploying candidates, yielding five unique, nontrivial, deployable Terraform projects suitable for systematic drift injection. These projects vary in size, domain, and structure, from tens to thousands of resources (Table 3): **lab12** [50] for multi-VPC networking, **ssm3** [28] for zero-downtime patching, **live-score** [17] for an event-driven service, **flask** [43] for a DynamoDB-backed REST API, and **mega-mesh** [41] for a multi-region VPC mesh. These projects also span flat resource layouts and nested custom-module structures, making them suitable for evaluating drift localization under diverse Terraform code organizations.

Drift scenario sampling. For each selected base project, we generated drift scenarios from two complementary sources: AWS Systems Manager (SSM) Automation runbooks [7] and manually crafted project-specific scenarios. Here, we describe how we created drift scenarios from the 763 publicly available SSM Automation runbooks. These runbooks operate at the API level and are not specific to any Terraform projects, requiring manual inspection. For each project, we compared each scenario against the resources present in the project’s Terraform state and retained scenarios whose required resources existed or could be added consistently with the project’s architecture. We inspected the runbook scripts and descriptions to determine whether each runbook would inject meaningful drift into at least one of the five projects. For example, a runbook that updates an existing CloudWatch alarm does not apply to a project that does not provision any CloudWatch alarms. Because our evaluation repeatedly modifies live cloud infrastructure, the experts also filtered out runbooks that were operationally unsuitable for frequent execution. For instance, some AWS resources impose mutation-rate limits. EBS volumes, for example, cannot be modified repeatedly within a short time window. We removed some such cases based on expert knowledge, and identified additional cases later through deployment testing. This process resulted in 90 unique runbook scenarios that perform mutating infrastructure operations.

Drift creation. We constructed each drift through an expert-led manual process, using an LLM only as a coding assistant. Each final mutation was validated using a fixed checklist covering whether: (1) it matched the natural-language scenario; (2) the affected resources were relevant and compatible with the project state; (3) applying it changed the live infrastructure; (4) the diff contained no unrelated edits; and (5) it could be deployed and reverted without residual infrastructure changes. The experts inspected the Terraform code, and when needed, cross-checked the affected resources against the Terraform state and deployment logs. When a case was complex or ambiguous, the experts discussed it jointly and either manually repaired the mutation when the intended drift was clear or excluded it when the scenario could not be mapped unambiguously to the project. We observed that the LLM agent introduced configuration changes unrelated to the scenario, which were later removed or adjusted manually. To mitigate bias, more than 80% of the scenarios came from AWS-provided runbooks rather than author-designed examples, the same validation checklist was applied across all projects, and final inclusion depended on deployment-and-revert testing rather than expert judgment alone. We supplemented the runbook-derived mutations with manually crafted project-specific mutations to cover realistic drift cases not represented in the SSM runbooks.

Drift validation. To support continuous evaluation, each mutation was validated by deploying the mutated configuration and then reverting the infrastructure to the original configuration. This validation was costly: each deployment-and-revert cycle could take minutes to hours, depending on project size and the cloud resources involved. Many failures required manual inspection and repair, often followed by partial infrastructure teardown and redeployment. The revertibility check was necessary because some infrastructure changes are not bidirectional; for example, an EBS volume can be expanded but not shrunk back to its original size. We also observed other failure modes, such as drift altering account permissions and causing cascading failures in subsequent tests. Before exclusion, we attempted to diagnose and repair failed cases when the intended drift remained clear, with debugging effort ranging from minutes to several hours per case. During validation, we further excluded 14 from lab12, 18 from flask, 17 from ssm3, 21 from live-score, and 7 from mega-mesh. Overall, this process produced **372 drift cases**, and we spent more than **1,200 hours of DevOps expert effort** on scenario filtering, mutation inspection, manual correction, and validating deployment-revert cycles. The complete list of SSM-sourced scenarios and project-specific scenarios is included in the supplementary material.

5.3 Experimental Setup

We implemented NSync in 11k lines of code in Python, and it targets Terraform-based IaC for the AWS cloud, using Boto3 to interface with CloudTrail for obtaining API traces. To our knowledge, NSync is the first domain-specific agentic system for IaC reconciliation. Our evaluation focuses on system design, and we use Claude 3.7 Sonnet, one of the strongest coding models at the time of writing, across all variants, including the baseline, NSync-NL, and NSync, which we describe next.

Baselines. We evaluate three agents, all built using the Strands agent framework [1] and deployed via the AWS Bedrock service with Claude 3.7 Sonnet as the backbone LLM. The Baseline solution is an LLM agent equipped with pre-built file editing tools and shell access, but without domain specialization. The second method, NSync-NL, is our system without continual learning, where each reconciliation run is treated independently. Our full system, NSync, builds on the same framework and backbone, but with all domain-specific mechanisms detailed in Section 4—including intent identification, patch generation (e.g., `drift_report` and `self_critique`), as well as a project-level knowledge base (KB) that enables continual learning across runs. The annotation process in both variants relies on the same Claude 3.7 Sonnet model. All three systems operate on identical inputs (mutating API traces and the IaC project directory) and share the same core editing tools shown in Table 4. All experiments are conducted against realistic deployments on AWS to ensure results reflect practical operating conditions. For each system, we perform three independent runs, with drift scenarios presented in arbitrary order to reduce any bias from experiment ordering.

Metrics. We focus on correctness and efficiency, using the $\text{pass}@k$ metric. A reconciliation attempt is considered correct if the agent’s patched configuration, when evaluated with terraform plan against the *ground truth state* and the *live infrastructure* created by the mutation, produces no differences—for example, the out-of-band creation of a new EC2 instance has been captured by the patch, i.e., it includes the corresponding resource block (e.g., `aws_instance`). Efficiency is measured in terms of two complementary metrics: tokens processed (computational cost) and agent steps. Since multiple candidate patches may be generated for each scenario, we report results using the $\text{pass}@k$ metric, which measures whether at least one of the top- k patches is correct while accounting for the associated efficiency cost. We report both $\text{pass}@1$ and $\text{pass}@3$ results to evaluate the single-attempt accuracy of the system and its robustness under limited retries.

Table 4. Tools available to all agents.

Agent Tools	Description
<code>file_read</code>	Read a file
<code>file_write</code>	Overwrite a file
<code>editor</code>	Fine-grained edits
<code>shell</code>	Run shell commands

Table 5. Accuracy and Efficiency Comparison between Baseline, NSync-NL, and NSync. M: Millions.

Benchmark	# Exp.	Pass@3 Accuracy			Avg Tokens (M)			Avg Steps		
		Baseline	NSync-NL	NSync	Baseline	NSync-NL	NSync	Baseline	NSync-NL	NSync
lab12	96	0.89	0.98	0.99	0.45	0.26	0.36	21.7	14.5	16.2
flask	94	0.64	0.95	0.97	0.73	0.57	0.66	26.1	18.2	19.4
ssm3	103	0.74	0.92	0.97	0.52	0.32	0.33	21.0	15.3	15.5
live-score	61	0.57	0.93	0.98	0.70	0.39	0.51	22.4	17.3	21.6
mega-mesh	18	0.72	0.94	0.94	1.05	0.44	0.50	18.9	18.1	15.6
Overall	372	0.71	0.95	0.97	0.69	0.40	0.47	22.0	16.7	17.7

5.4 Results and Analysis

We conduct a comprehensive evaluation of NSync on real-world Terraform projects and realistic drifts. We design our experiments to answer four key research questions:

- **RQ1 (Overall Performance):** How effective is NSync at reconciling infrastructure drifts, as measured by pass@k accuracy, compared to a baseline agent?
- **RQ2 (Intent Identification):** How well does API trace consolidation help identify the intent, thus improving performance, and what are the trade-offs in accuracy and efficiency?
- **RQ3 (Patch Generation):** How does NSync leverage custom tools during reconciliation, and what do the frequency and timing of tool usage reveal about its agentic problem-solving strategy?
- **RQ4 (Continual Learning):** How reliably does the learning algorithm improve outcomes?

In the rest of this section, we answer these research questions.

5.4.1 RQ1: Overall Performance. We present Table 5 summarizing the detailed accuracy and efficiency results of NSync, NSync-NL, and the baseline. NSync proves highly effective, significantly outperforming the baseline agent. Across 372 scenarios in five diverse real-world projects, NSync achieves 0.97 accuracy, far exceeding the baseline. At the same time, it is more efficient, consuming on average 0.47M tokens per reconciliation, a 1.47× improvement in token efficiency. Furthermore, both NSync-NL and NSync significantly outperform the baseline across all benchmarks, requiring fewer reasoning cycles and less computation overall. We report additional runtime and LoC results in the supplementary material. On simpler projects (e.g., lab12), which have a flat structure without modules, all methods perform well. The performance gap widens on projects with custom or prebuilt modules, where the code structure is more complex. And the performance of NSync-NL and NSync scales more gracefully to project complexity.

A closer inspection shows that the baseline consumes more tokens and fails more often, because it relies heavily on raw terraform plan outputs. Its context becomes cluttered with verbose and often irrelevant information produced by this Terraform-native tool. The context window overflows, triggering summarization that discards useful details. This forces the agent into repeated reasoning and revisiting Terraform files, further increasing token usage. Even worse, the baseline hallucinates: due to the bloated context dominated by plan noise, it incorrectly reasons that applying the current configuration will fix the drift (i.e., undoing the out-of-band change), rather than updating the configuration to reflect the changes. In contrast, NSync’s specialized drift_report tool feeds only the relevant drifted resources into the model’s context. This focused signal not only improves efficiency but also yields higher accuracy by avoiding context dilution.

NSync has a success rate close to 100%, and our analysis of the failure cases shows that the remaining failures typically arise in import handling. In Terraform, import is used to bring externally created resources under IaC management by linking them with corresponding configuration blocks. In one case from the flask project, when importing a resource, the agent mistakenly attempted “RootAccUsage:/aws/cloudtrail/flask-micros”, which incorrectly reversed the

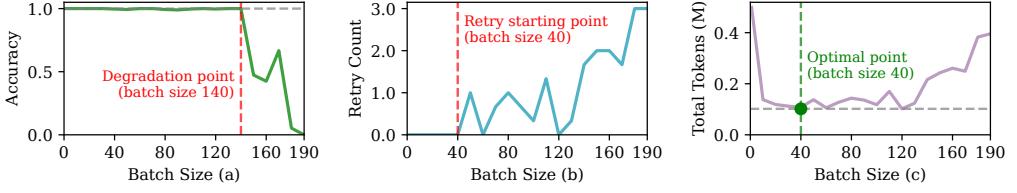


Fig. 5. LLM annotation of a synthetic API trace with 190 mutating events.

name and identifier, and consequently failed to discover the correct format. The correct format is “/aws/cloudtrail/flask-micros:RootAccUsage.” Such cases could be addressed with retrieval-augmented generation, where the precise import syntax is supplied directly to the agent. In summary, these results demonstrate that NSync yields substantial accuracy and efficiency gains.

5.4.2 RQ2: Intent Identification. We observed that intent identification improves efficiency while maintaining accuracy. NSync correctly handled common AWS APIs. NSync can be readily extended with an API-documentation lookup step during annotation to further ensure correct interpretation of API semantics (e.g., via AWS’s MCP documentation server [44]), but we did not include this component in the current design because it was unnecessary for the scenarios evaluated.

The small annotation overhead from LLM is amortized by the much larger savings during reconciliation. Table 6 compares NSync with and without intent identification (IID) on the live-score benchmark. Both NSync and NSync-NL maintain high accuracy, while IID reduces token usage and agent steps by approximately 23% in token usage and steps. Accuracy remains unchanged for NSync (0.98) and even slightly improves for NSync-NL (0.93 vs. 0.92).

Intent identification proceeds in two steps: annotation and consolidation. We next examine the scalability of its annotation stage—specifically, whether LLMs can annotate large batches of API events effectively. We find that LLM annotation consumes on average 5.6K tokens per drift—only about 1% of the overall reconciliation cost, which is on the order of millions of tokens. To stress-test the system, we construct a single trace containing 190 mutating API calls. Figure 5 shows the effect of varying batch size on this trace. Accuracy drops sharply beyond 140, marking a clear degradation point. The number of retries increases once the batch size exceeds 40, indicating that the model struggles to process overly large batches in a single pass. This effect is also visible in token usage: it is minimized around a batch size of 40, which we identify as the optimal operating point. Beyond this point, additional retries required to handle larger batches lead to higher overall token consumption. These results show that event consolidation is essential, and that batching around 40 mutating API calls is sufficient in practice, providing a good balance of accuracy, efficiency, and robustness. We therefore adopt a batch size of 40 for all experiments.

Table 6. Ablation study of intent identification (IID) on the live-score benchmark.

Method	Pass@3	Tokens (M)	Steps
baseline	0.57	0.70	22.4
NSync-NL	0.93	0.39	17.3
NSync-NL w/o IID	0.92	0.51	22.5
NSync	0.98	0.51	21.6
NSync w/o IID	0.98	0.66	28.1

5.4.3 RQ3: Patch Generation. Table 7 shows that specialized tools substantially improve reconciliation performance, with each removal lowering accuracy and the absence of drift_report causing the steepest drop. We perform an ablation study on the live-score benchmark. Removing any tool or knowledge-base (KB) operation reduces reconciliation accuracy. Without learning—i.e., in NSync-NL, where the KB operations knowledge_update and knowledge_retrieval are disabled—accuracy drops from 0.98 to 0.93. Excluding self_critique reduces it further to 0.81, while excluding drift_report has the most severe impact, lowering accuracy to 0.60. This shows

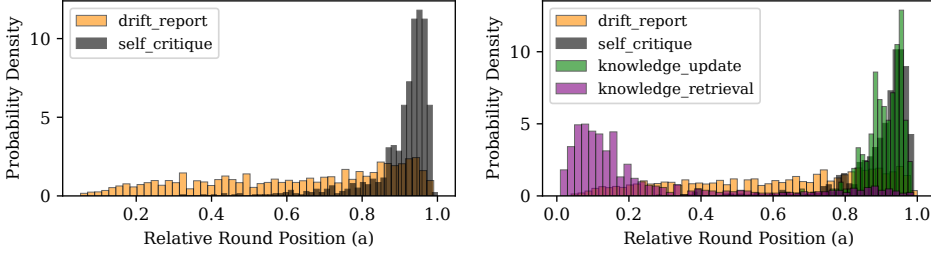


Fig. 6. Distribution of tool usage across reconciliation progress. The x-axis denotes the normalized round position ($\text{tooluse_round} / \text{total_round}$) for each run. In Figure 6(a), NSync-NL uses `drift_report` uniformly throughout the process, while `self_critique` is concentrated near the end. In Figure 6(b), NSync has similar tool-use patterns, but knowledge tools exhibit distinct timing: `knowledge_retrieval` is typically invoked early to supply prior context, while `knowledge_update` appears near the end to record new insights.

that each tool plays an important role, with `drift_report` and `self_critique` being especially critical for achieving high performance.

We next analyze the timing of tool use during reconciliation. As shown in Figure 6, `drift_report` is invoked by the agent throughout the process, `self_critique` is concentrated near the end, and in NSync, knowledge tools follow a clear temporal structure (retrieval at earlier stages, update at later stages). This confirms that the agent is able to use our tools effectively, in a structured and phase-specific manner.

We take a closer look at tool usage frequency across the three methods (Figure 7). While the baseline relies heavily on generic `file_read` and `shell` operations, NSync-NL shifts its usage toward domain-specific tools, `drift_report` and `self_critique`. The frequency of `editor` and `file_write` calls remains similar across agents, since these tools are still required to generate patches. By contrast, `shell` usage decreases substantially, as the baseline often used `shell` to locate files and invoke native Terraform commands—tasks that are largely replaced by domain-specific tools in NSync. `file_read` usage also decreases, since the baseline repeatedly re-scans the codebase to rediscover context, whereas NSync makes more efficient use of retained context and KB. NSync further incorporates KB operations, though at relatively low frequency compared to other tools; this is also reflected in Figure 6 as KB operations only happen at the beginning and the end.

5.4.4 RQ4: Continual Learning. We find that NSync remains the most robust under stricter pass@1 evaluation when using our methods, as shown in Table 8; it also maintains reliable performance across experiment runs where drifts are presented in arbitrary order. Figure 9 presents a snippet of the evolving reconciliation knowledge base learned from experience. The knowledge spans diverse categories, including how to inspect resource state with Terraform commands, map API parameters to Terraform attributes, resolve formatting issues, and apply the custom tool `drift_report` to pinpoint attribute-level drifts more effectively.

Table 7. Performance ablation study on the live-score benchmark.

Method	Pass@3
NSync	0.98
NSync-NL	0.93
NSync w/o <code>drift_report</code>	0.60
NSync w/o <code>self_critique</code>	0.81

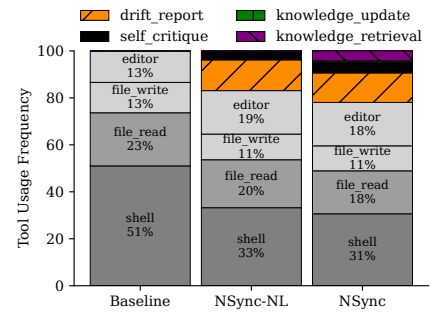


Fig. 7. Tool usage frequency breakdown.

Table 8. Performance across runs under the pass@1 metric

Benchmark	Baseline		NSync-NL		NSync	
	Pass@1	range $\pm$ std	Pass@1	range $\pm$ std	Pass@1	range $\pm$ std
lab12	0.63	$(0.52, 0.78) \pm 0.11$	0.88	$(0.88, 0.89) \pm 0.01$	0.90	$(0.86, 0.95) \pm 0.04$
flask	0.40	$(0.31, 0.47) \pm 0.07$	0.70	$(0.69, 0.72) \pm 0.01$	0.65	$(0.55, 0.74) \pm 0.08$
ssm3	0.51	$(0.49, 0.52) \pm 0.01$	0.73	$(0.65, 0.80) \pm 0.06$	0.80	$(0.76, 0.84) \pm 0.03$
live-score	0.38	$(0.34, 0.39) \pm 0.03$	0.70	$(0.66, 0.75) \pm 0.04$	0.84	$(0.80, 0.93) \pm 0.06$
mega-mesh	0.53	$(0.47, 0.59) \pm 0.05$	0.76	$(0.59, 0.88) \pm 0.13$	0.78	$(0.71, 0.82) \pm 0.06$
Overall	0.49	$(0.43, 0.55) \pm 0.05$	0.76	$(0.69, 0.81) \pm 0.05$	0.80	$(0.73, 0.86) \pm 0.05$

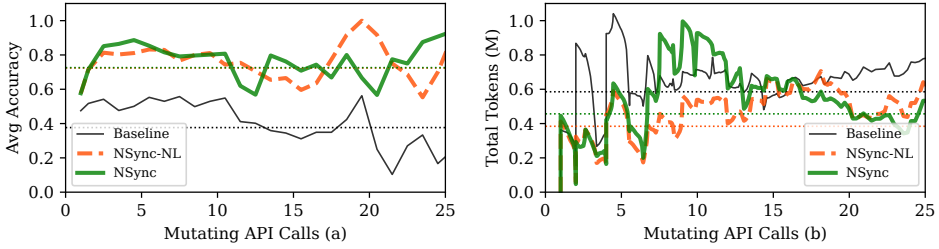


Fig. 8. Performance comparison across mutating API calls. Figure 8(a) reports average accuracy across all drift scenarios from five projects, reorganized by the number of mutating API calls. Figure 8(b) presents the corresponding token cost for reconciliation as drift complexity increases.

Table 8 evaluates robustness under the pass@1 metric, where no retries are allowed, and results are averaged across three runs. The baseline performs poorly in this setting, with an overall accuracy of only 0.49, highlighting its unreliability when a single attempt must succeed. NSync-NL improves substantially to 0.76, and NSync further increases robustness to 0.80 by reusing knowledge from prior reconciliations. Importantly, the drift scenarios are presented in arbitrary order across runs, ensuring that improvements are not due to any favorable ordering of the data. This confirms that the learning method is agnostic to ordering and that there is no hidden bias or structure in our benchmarks that guarantees improvement. While all methods see a drop compared to pass@3, the degradation is smaller for NSync-NL and especially for NSync, which remains consistently strong across runs. These results demonstrate that learning not only improves average accuracy but also reduces variance, yielding more reliable reconciliation outcomes.

Figure 8 shows that, as drift complexity increases (more mutating API calls), the baseline shows a clear downward trend in accuracy, while NSync-NL remains stable. In terms of efficiency, the baseline’s token usage steadily grows with complexity, whereas NSync-NL consistently requires fewer tokens, and NSync adds only a modest overhead due to knowledge retrieval. These results highlight that our methods are more resilient under increasing drift complexity, maintaining accuracy and efficiency where the baseline degrades. As the number of mutating API calls increases, the baseline token usage eventually surpasses NSync.

6 Related Work

AI Agent for Cloud Management. Recent advances have explored the use of AI agents to automate diverse aspects of cloud operations, or “AIOps”. However, much of this work only focuses on incident detection, root cause diagnosis, and remediation [12, 13, 38, 54, 59]—e.g., via log mining, causal inference, or LLM-powered analysis to recommend or execute corrective actions. Meanwhile, existing work also highlights the potential of AI agents to expand beyond diagnostic tasks to handle provisioning, continuous monitoring, and tool synthesis [8, 14, 16, 62, 63]. NSync is an AIOps design but is specialized for detecting and reconciling drift in IaC frameworks.

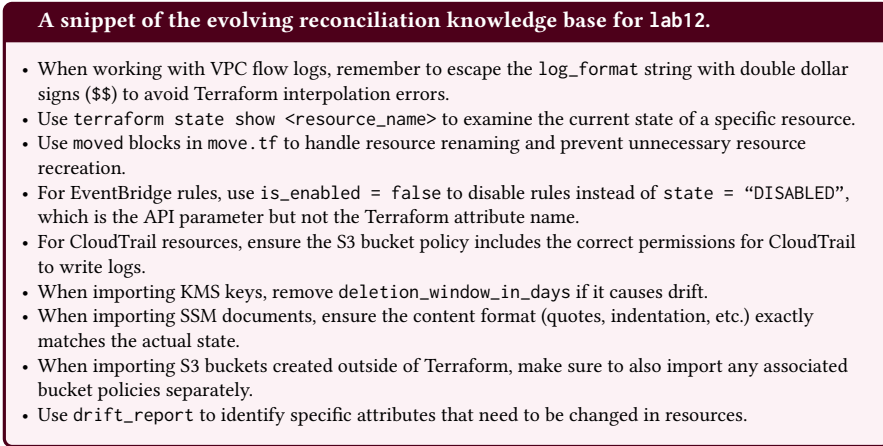


Fig. 9. An example of the evolving reconciliation knowledge base. All entries for the lab12 project represent accumulated knowledge, continually updated by the agent from its prior reconciliation experience.

AI Agent for Program Repair. Automated Program Repair (APR) has been extensively studied, with recent surveys showing that LLMs have transformed the field by enabling repair through fine-tuning, prompting, procedural workflows, and agentic frameworks [60, 66]. Fine-tuned repair systems [29, 46, 68] leverage domain-specific bug corpora for training, while prompting approaches [23, 58] offer lightweight zero/few-shot deployment. More advanced pipelines [55, 57] integrate retrieval or analysis feedback to guide iterative repair. Agentic systems [9, 52, 61, 67] orchestrate external tools, testing, and multi-step reasoning. These works primarily target source code bugs in conventional software projects, with correctness validated against unit tests or benchmarks. NSync builds on the program-repair perspective but applies it to cloud Infrastructure-as-Code. Unlike traditional APR, where explicit failing tests provide repair oracles, IaC reconciliation operates without runnable test cases: its “specification” is implicitly encoded in cloud API traces. This shifts the repair challenge from patching faulty logic based on test cases to synthesizing declarative updates that represent API traces, ensuring that the IaC codebase reflects the true deployed state.

7 Threats to Validity

Internal Validity. Our correctness metric relies on `terraform plan` output and resource identity matching to determine whether a reconciliation is successful. This may classify some semantically correct but non-canonical patches as failures (e.g., patches that differ syntactically but result in equivalent cloud states), leading to conservative false negatives. As a result, our reported success rates should be interpreted as lower bounds on actual reconciliation correctness.

Another internal threat arises from manual validation and filtering of benchmark ground truth. Although each drift case was inspected by DevOps experts, expert review can involve subjective judgment, especially when a scenario admits multiple plausible infrastructure changes or when a generated mutation only partially matches the intended operation. Candidate drifts were retained only if they could be generated, validated, deployed, and reverted; experts attempted to manually repair failed cases before exclusion. We mitigated this threat by using a fixed validation checklist, involving two experts for ambiguous cases, inspecting both mutation reports and Terraform diffs, requiring deployment-and-revert validation, and checking final correctness with `terraform plan` against the ground-truth state. Nevertheless, this process may underrepresent drifts with difficult-to-synthesize ground truth and bias the benchmark toward more structured or easier-to-reconcile cases.

External Validity. Our evaluation uses open-source Terraform projects from GitHub. We made a best-effort attempt to select projects that vary in scale, application type, and resource diversity. Nevertheless, open-source Terraform repositories may not fully capture the characteristics of production cloud infrastructure. In particular, production IaC deployments are often proprietary, substantially larger, and more expensive to deploy and evaluate. Drift scenarios are primarily sourced from runbooks, which encode common DevOps operations on AWS. This improves realism by grounding the benchmark in operationally meaningful actions. However, real-world drift may also arise from undocumented ad hoc mutations that remain silent until later and potentially cause cascading failures. Such cases are difficult to systematically observe, reproduce, and validate. Thus, while our benchmark reflects realistic projects and common operational drifts, it may underrepresent rare, latent, or highly organization-specific drift scenarios.

Our evaluation focuses on Terraform and AWS. However, NSync depends only on abstractions common across IaC frameworks, namely a planning phase that previews intended actions and API audit logs that record executed operations. These abstractions are available in other IaC frameworks and cloud providers such as Azure and Google Cloud—the core design of NSync generalizes beyond the specific platforms studied. To isolate the impact of our agent design, we fix the underlying LLM to Claude 3.7 Sonnet, one of the strongest coding models at the time of writing. Performance may vary with other LLMs, and evaluating additional models is left to future work.

Construct Validity. To ensure realism, we implement NSync within a production-grade agent framework [1] and validate all reconciliation scenarios using real cloud deployments rather than simulated environments. While this strengthens ecological validity, it may limit the scale of evaluation due to cost and operational constraints.

8 Conclusion

Infrastructure-as-Code (IaC) frameworks are gaining popularity, and they encode the infrastructure in a declarative configuration as the source of truth. However, when IaC frameworks are used together with other management interfaces (e.g., cloud consoles, CLI, SDK), the IaC configuration no longer captures the de facto state, leading to infrastructure drift. We presented NSync, the first agentic system for automated IaC reconciliation. NSync relies on cloud APIs to identify and reconcile drift, and it works by cleaning and consolidating noisy API traces, generating patches to update configurations, and continuously learning from experience. To support systematic evaluation, we introduced a pipeline for generating realistic and assessable drift scenarios, and curated the first IaC reconciliation dataset containing 372 validated cases across five real-world Terraform projects. Experiments show that NSync achieves 0.97 accuracy, a 26-percentage-point improvement over the baseline, while improving token efficiency by approximately 1.5×. These results highlight that IaC reconciliation is both feasible and practical with modern agentic systems.

Future Work. Our evaluation is on the leading IaC platform—Terraform—and the most popular cloud—AWS. NSync’s architecture is generalizable to other IaC frameworks (e.g., Pulumi [40], OpenTofu [37]) and cloud providers (e.g., Azure, GCP). Key changes may include different API monitoring tools across clouds and resource management practices. For instance, Azure’s and GCP’s event logs may use different event schemas compared to AWS CloudTrail. Each IaC framework also has its own local state scheme and planning operations. Hence, an interesting avenue of future work is to generalize NSync to other IaC frameworks and clouds.

9 Data Availability

The implementation of NSync and the curated IaC reconciliation dataset, comprising 372 validated drift cases, are available in the supplementary material and at <https://github.com/zyang37/nsync>.

References

- [1] Strands Agents. 2025. Strands Agents. <https://strandsagents.com/latest/>
- [2] Alibaba Cloud. 2025. *Alibaba Cloud ActionTrail*. ActionTrail monitors and records your Alibaba Cloud account activities; latest documentation updated February 12, 2025. <https://www.alibabacloud.com/help/en/actiontrail/>
- [3] Amazon Web Services. 2024. AWS CloudTrail Documentation. <https://docs.aws.amazon.com/awscloudtrail>.
- [4] Amazon Web Services. 2024. *AWS Systems Manager Automation*. Amazon Web Services. AWS Systems Manager Documentation. <https://docs.aws.amazon.com/systems-manager/latest/userguide/systems-manager-automation.html>
- [5] Amazon Web Services. 2024. *aws2tf*. AWS Samples. GitHub Repository. <https://github.com/aws-samples/aws2tf>
- [6] Amazon Web Services. 2024. Understanding CloudTrail events. AWS Documentation. <https://docs.aws.amazon.com/awscloudtrail/latest/userguide/cloudtrail-events.html>
- [7] AWS. 2024. AWS Systems Manager Automation runbook reference. <https://docs.aws.amazon.com/systems-manager-automation-runbooks/latest/userguide/automation-runbook-reference.html>.
- [8] Archit Bhatnagar, Yiming Qiu, Sarah McClure, Sylvia Ratnasamy, and Ang Chen. 2025. A Case for Learned Cloud Emulators. In *Proceedings of the 24th ACM Workshop on Hot Topics in Networks (HotNets '25)*. doi:10.1145/3772356.3772404
- [9] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2024. Repairagent: An autonomous, llm-based agent for program repair. *arXiv preprint arXiv:2403.17134* (2024). doi:10.1109/ICSE55347.2025.00157
- [10] Christoph Bühler, David Spielmann, Roland Meier, and Guido Salvaneschi. 2025. TerraDS: A Dataset for Terraform HCL Programs. In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*. IEEE Press. doi:10.1109/MSR66628.2025.00101
- [11] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. CodeT: Code Generation with Generated Tests. <https://arxiv.org/abs/2207.10397>
- [12] Yinfang Chen, Manish Shetty, Gagan Somashekar, Minghua Ma, Yogesh Simmhan, Jonathan Mace, Chetan Bansal, Rujia Wang, and Saravan Rajmohan. 2025. AIOpsLab: A Holistic Framework to Evaluate AI Agents for Enabling Autonomous Clouds. <https://arxiv.org/abs/2501.06706>
- [13] Yinfang Chen, Huaibing Xie, Minghua Ma, Yu Kang, Xin Gao, Liu Shi, Yunjie Cao, Xuedong Gao, Hao Fan, Ming Wen, Jun Zeng, Supriyo Ghosh, Xuchao Zhang, Chaoyun Zhang, Qingwei Lin, Saravan Rajmohan, Dongmei Zhang, and Tianyin Xu. 2024. Automatic Root Cause Analysis via Large Language Models for Cloud Incidents. In *Proceedings of the Nineteenth European Conference on Computer Systems (EuroSys '24)*. doi:10.1145/3627703.3629553
- [14] Qian Cheng, Doyen Sahoo, Amrita Saha, Wenzhuo Yang, Chenghao Liu, Gerald Woo, Manpreet Singh, Silvio Saverese, and Steven C. H. Hoi. 2023. AI for IT Operations (AIOps) on Cloud Platforms: Reviews, Opportunities and Challenges. <https://arxiv.org/abs/2304.04661>
- [15] Cycloid. 2025. terracognita. <https://www.cycloid.io/open-source/terracognita>
- [16] Yingnong Dang, Qingwei Lin, and Peng Huang. 2019. AIOps: Real-World Challenges and Research Innovations. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. doi:10.1109/ICSE-Companion.2019.00023
- [17] Chris Dobson. 2023. live-scores. <https://github.com/ChrisDobby/live-scores>
- [18] env0. 2025. Meet Cloud Compass: AI-Assisted IaC Coverage, Audit and Risk Mitigation. <https://www.env0.com/blog/meet-cloud-compass-ai-assisted-iac-coverage-audit-and-risk-mitigation>. Accessed: 2025-10-06. <https://www.env0.com/blog/meet-cloud-compass-ai-assisted-iac-coverage-audit-and-risk-mitigation>
- [19] Firefly. 2025. *State of Infrastructure as Code 2025*. Firefly.ai. <https://www.firefly.ai/state-of-iac-2025>
- [20] Gomboc. 2025. Why 70% of Cloud Breaches Start with IaC Misconfigurations (and How to Fix Them). Gomboc. <https://www.gomboc.ai/blog/why-70-of-cloud-breaches-start-with-iac-misconfigurations-and-how-to-fix-them>
- [21] Google Cloud. 2024. Cloud Audit Logs Overview. <https://cloud.google.com/logging/docs/audit>.
- [22] Google Cloud. 2024. *Terraformer*. Google Cloud. GitHub Repository. <https://github.com/GoogleCloudPlatform/terraformer>
- [23] Mirazul Haque, Petr Babkin, Farima Farmahinifarahani, and Manuela Veloso. 2025. Towards Effectively Leveraging Execution Traces for Program Repair with Code LLMs. *arXiv preprint arXiv:2505.04441* (2025). doi:10.18653/v1/2025.knowledgenlp-1.17
- [24] Harness. 2025. \$44.5 Billion in Infrastructure Cloud Waste Projected for 2025 Due to FinOps and Developer Disconnect, Finds "FinOps in Focus" Report from Harness. PR Newswire. <https://www.prnewswire.com/news-releases/44-5-billion-in-infrastructure-cloud-waste-projected-for-2025-due-to-finops-and-developer-disconnect-finds-finops-in-focus-report-from-harness-302385580.html>
- [25] HashiCorp. 2024. *Generating Configuration from Import*. HashiCorp. Terraform Documentation. <https://developer.hashicorp.com/terraform/language/import/generating-configuration>
- [26] Hashicorp. 2025. Manage resource drift. Terraform Documentation. <https://developer.hashicorp.com/terraform/tutorials/state/resource-drift#introduce-drift>
- [27] HashiCorp. 2025. Terraform. <https://developer.hashicorp.com/terraform>

- [28] Joseph Heyburn. 2023. terraform-examples: AWS SSM Automation Examples. <https://github.com/jdheyburn/terraform-examples/tree/main/aws-ssm-automation-3>
- [29] Nan Jiang and Yi Wu. 2024. RepairCAT: applying large language model to fix bugs in AI-generated programs. In *Proceedings of the 5th ACM/IEEE international workshop on automated program Repair*. doi:10.1145/3643788.3648020
- [30] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2017. Billion-scale similarity search with GPUs. arXiv:1702.08734 [cs.CV] <https://arxiv.org/abs/1702.08734>
- [31] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the Middle: How Language Models Use Long Contexts. *Transactions of the Association for Computational Linguistics* (2024). doi:10.1162/tacl_a_00638
- [32] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegrefe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welbeck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-Refine: Iterative Refinement with Self-Feedback. arXiv:2303.17651 [cs.CL] <https://arxiv.org/abs/2303.17651>
- [33] Microsoft. 2024. *Automation in Microsoft Sentinel*. Microsoft. Microsoft Azure Documentation. <https://learn.microsoft.com/en-us/azure/sentinel/automation/automation>
- [34] Microsoft Azure. 2022. Configuring event ordering policies for Azure Stream Analytics. Azure Documentation. <https://learn.microsoft.com/en-us/azure/stream-analytics/event-ordering>
- [35] Microsoft Azure. 2024. Azure Activity Logs and Resource Manager Logs. <https://learn.microsoft.com/en-us/azure/azure-monitor/platform/activity-log>.
- [36] Microsoft Azure. 2025. aztfexport. <https://github.com/Azure/aztfexport>
- [37] OpenTofu Project. 2024. OpenTofu: The Open Source Infrastructure as Code Tool. Accessed: August 28, 2026. <https://opentofu.org/>
- [38] Changhua Pei, Zexin Wang, Fengrui Liu, Zeyan Li, Yang Liu, Xiao He, Rong Kang, Tieying Zhang, Jianjun Chen, Jianhui Li, Gaogang Xie, and Dan Pei. 2025. Flow-of-Action: SOP Enhanced LLM-Based Multi-Agent System for Root Cause Analysis (WWW '25). Association for Computing Machinery. doi:10.1145/3701716.3715225
- [39] Jingjia Peng, Yiming Qiu, Patrick Tser Jern Kon, Pinhan Zhao, Yibo Huang, Zheng Guo, Xinyu Wang, and Ang Chen. 2025. Automated Lifting for Cloud Infrastructure-as-Code Programs. In *2025 IEEE/ACM International Workshop on Cloud Intelligence & AIOps (AIOps)*. doi:10.1109/AIOps66738.2025.00007
- [40] Pulumi Corporation. 2024. Pulumi: Modern Infrastructure as Code. Accessed: August 28, 2026. <https://www.pulumi.com/>
- [41] Jude Quintana. 2023. terraform-aws-mega-mesh. <https://github.com/JudeQuintana/terraform-aws-mega-mesh>
- [42] Nuno Saavedra, João F. Ferreira, and Alexandra Mendes. 2025. InfraFix: Technology-Agnostic Repair of Infrastructure as Code. arXiv:2503.17220 [cs.SE] <https://arxiv.org/abs/2503.17220>
- [43] AWS Samples. 2023. deploy-python-flask-microservices-to-aws-using-open-source-tools. <https://github.com/aws-samples/deploy-python-flask-microservices-to-aws-using-open-source-tools>
- [44] Amazon Web Services. 2025. AWS Documentation MCP Server. <https://awslabs.github.io/mcp/servers/aws-documentation-mcp-server>
- [45] Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language Agents with Verbal Reinforcement Learning. arXiv:2303.11366 [cs.AI] <https://arxiv.org/abs/2303.11366>
- [46] André Silva, Sen Fang, and Martin Monperrus. [n. d.]. *RepairLLaMA: Efficient Representations and Fine-Tuned Adapters for Program Repair*. Technical Report 2312.15698. arXiv. <http://arxiv.org/abs/2312.15698>
- [47] Spacelift. 2025. Drift Detection - Spacelift Documentation. <https://docs.spacelift.io/concepts/stack/drift-detection>. Accessed: 2025-10-06. <https://docs.spacelift.io/concepts/stack/drift-detection>
- [48] Stacklet. 2026. Changelog of AWS APIs. Powered by Stacklet. <https://awsapichanges.info/>
- [49] Josh Stella. 2019. A technical analysis of the Capital One cloud misconfiguration breach. Snyk Blog. <https://snyk.io/blog/a-technical-analysis-of-the-capital-one-cloud-misconfiguration-breach/>
- [50] Its-All-About-the Journey. 2024. AWS Advanced Networking. <https://github.com/Its-All-About-the-Journey/AWS-Advanced-Networking>.
- [51] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2023. Voyager: An Open-Ended Embodied Agent with Large Language Models. <https://arxiv.org/abs/2305.16291>
- [52] Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. 2025. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. arXiv:2407.16741 [cs.SE] <https://arxiv.org/abs/2407.16741>
- [53] Zhiruo Wang, Daniel Fried, and Graham Neubig. 2024. TroVE: Inducing Verifiable and Efficient Toolboxes for Solving Programmatic Tasks. <https://arxiv.org/abs/2401.12869>

- [54] Zefan Wang, Zichuan Liu, Yingying Zhang, Aoxiao Zhong, Jihong Wang, Fengbin Yin, Lunting Fan, Lingfei Wu, and Qingsong Wen. 2024. RCAgent: Cloud Root Cause Analysis by Autonomous Agents with Tool-Augmented Large Language Models (*CIKM '24*). doi:10.1145/3627673.3680016
- [55] Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. 2023. Copiloting the copilots: Fusing large language models with completion engines for automated program repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 172–184. doi:10.1145/3611643.3616271
- [56] Aaron Weiss, Arjun Guha, and Yuriy Brun. 2017. Tortoise: Interactive system configuration repair. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 625–636. doi:10.1109/ase.2017.8115673
- [57] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying LLM-based Software Engineering Agents. arXiv:2407.01489 [cs.SE] <https://arxiv.org/abs/2407.01489>
- [58] Chunqiu Steven Xia and Lingming Zhang. 2022. Less training, more repairing please: revisiting automated program repair via zero-shot learning (*ESEC/FSE 2022*). Association for Computing Machinery. <https://doi.org/10.1145/3540250.3549101>
- [59] Yiming Xiang, Zhenning Yang, Jingjia Peng, Hermann Bauer, Patrick Tser Jern Kon, Yiming Qiu, and Ang Chen. 2025. Automated Bug Discovery in Cloud Infrastructure-as-Code Updates with LLM Agents. In *2025 IEEE/ACM International Workshop on Cloud Intelligence & AIOps (AIOps)*. doi:10.1109/AIOps66738.2025.00011
- [60] Boyang Yang, Zijian Cai, Fengling Liu, Bach Le, Lingming Zhang, Tegawendé F. Bissyandé, Yang Liu, and Haoye Tian. 2025. A Survey of LLM-based Automated Program Repair: Taxonomies, Design Paradigms, and Applications. arXiv:2506.23749 [cs.SE] <https://arxiv.org/abs/2506.23749>
- [61] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. arXiv:2405.15793 [cs.SE] <https://arxiv.org/abs/2405.15793>
- [62] Zhenning Yang, Archit Bhatnagar, Yiming Qiu, Tongyuan Miao, Patrick Tser Jern Kon, Yunming Xiao, Yibo Huang, Martin Casado, and Ang Chen. 2025. Cloud Infrastructure Management in the Age of AI Agents. *SIGOPS Operating Systems Review* (2025). doi:10.1145/3759441.3759443
- [63] Zhenning Yang, Kaden Gruizenga, Tongyuan Miao, Patrick Tser Jern Kon, Hui Guan, and Ang Chen. 2026. Ambig-IaC: Multi-level Disambiguation for Interactive Cloud Infrastructure-as-Code Synthesis. arXiv:2604.02382 [cs.SE] <https://arxiv.org/abs/2604.02382>
- [64] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *Proceedings of the International Conference on Learning Representations (ICLR)*. https://openreview.net/forum?id=WE_vluYUL-X
- [65] Siyu Yuan, Zehui Chen, Zhiheng Xi, Junjie Ye, Zhengyin Du, and Jiecao Chen. 2025. Agent-R: Training Language Model Agents to Reflect via Iterative Self-Training. arXiv:2501.11425 [cs.AI] <https://arxiv.org/abs/2501.11425>
- [66] Qianjun Zhang, Chunrong Fang, Yang Xie, Yuxiang Ma, Weisong Sun, Yun Yang, and Zhenyu Chen. 2026. A Systematic Literature Review on Large Language Models for Automated Program Repair. *ACM Trans. Softw. Eng. Methodol.* (2026). doi:10.1145/3799693
- [67] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2024)*. Association for Computing Machinery. doi:10.1145/3650212.3680384
- [68] Xin Zhou, Kisub Kim, Bowen Xu, DongGyun Han, and David Lo. 2024. Out of sight, out of mind: Better automatic vulnerability repair by broadening input ranges and sources. In *Proceedings of the IEEE/ACM 46th international conference on software engineering (ICSE '24)*. doi:10.1145/3597503.3639222

Received 2026-01-29; accepted 2026-06-25