Automated Bug Discovery in Cloud Infrastructure-as-Code Updates with LLM Agents

Yiming Xiang*, Zhenning Yang*, Jingjia Peng, Hermann Bauer, Patrick Tser Jern Kon, Yiming Qiu, Ang Chen
University of Michigan
{kxiang, znyang, jingjia, hjbauer, patkon, yimingq, chenang}@umich.edu

***Abstract*—Cloud environments are increasingly managed by Infrastructure-as-Code (IaC) platforms (e.g., Terraform), which allow developers to define their desired infrastructure as a configuration program that describes cloud resources and their dependencies. This shields developers from low-level operations for creating and maintaining resources, since they are automatically performed by IaC platforms when compiling and deploying the configuration. However, while IaC platforms are rigorously tested for initial deployments, they exhibit myriad errors for *runtime updates*, e.g., adding/removing resources and dependencies. IaC updates are common because cloud infrastructures are long-lived but user requirements fluctuate over time. Unfortunately, our experience shows that updates often introduce subtle yet impactful bugs. The update logic in IaC frameworks is hard to test due to the vast and evolving search space, which includes diverse infrastructure setups and a wide range of provided resources with new ones frequently added. We introduce TerraFault, an automated, efficient, LLM-guided system for discovering update bugs, and report our findings with an initial prototype. TerraFault incorporates various optimizations to navigate the large search space efficiently and employs techniques to accelerate the testing process. Our prototype has successfully identified bugs even in simple IaC updates, showing early promise in systematically identifying update bugs in today's IaC frameworks to increase their reliability.**

***Index Terms*—Infrastructure-as-Code, Program update, Using LLMs for Cloud Ops, Reliability, Software testing and debugging.**

I. Introduction

Cloud infrastructure has become a critical part of many enterprises. Until recently, cloud tenants (e.g., Home Depot) relied on approaches like API scripting or web portal clicking to manage and deploy their cloud infrastructure, which are known to be unreliable, not scalable, and hard to manage. This motivated the development of modern cloud Infrastructure-as-Code (IaC) tools [1]–[3] which allow cloud tenants to define their intended infrastructure using high-level IaC programs, then realize the user intention automatically by executing sequences of API calls for resource provisioning (e.g., creating a VM), thus shielding cloud users away from low-level details. So far, Terraform [4] is leading the market, but tools like OpenTofu [5] and Pulumi [6] are also gaining popularity. Essentially, these tools follow a *state-centric* design philosophy, enabling them to track the current cloud infrastructure state and update it to bridge the "delta" between the desired and current states.

While a state-centric design is essential to modern IaC platforms, infrastructure states are in constant flux and they

```
resource VPC a { // define resource
  name = "a";    // specify attributes
  location = "US-west";
}
resource SUBNET b {
  name = "b";
  vpc_name = VPC.a.name;
}
```

↓ Change VPC Region & Delete Subnet

```
resource VPC a {
  name = "a";
  location = "US-east"; // field update
}
/* Removed subnet: structural update */
```

Fig. 1: A simplified IaC Update

are not easy to handle correctly. As shown in Figure 1, since the intention of cloud users could change over time (e.g., hosting new services, scaling in/out, migration to other clouds, or patching security issues), IaC platforms have to handle frequent *state transitions* during the management lifecycle. A strawman approach to realize such transitions is simply to *destroy and re-deploy* the entire infrastructure, which is by nature a very slow and disruptive process. As a result, IaC platforms attempt to do *in-place updates* whenever possible to reduce the deployment overhead. This introduces complex, case-by-case behaviors that vary from one IaC resource to another.

To navigate such complexity, modern IaC platforms adopt a two-layered system architecture. First, a provider-agnostic core compiler translates state transitions into deployment plans. Then a set of provider-specific plugins carry out the plans by executing cloud-level APIs, making decisions on how to perform in-place updates. These plugins, authored by developers from different providers, vary in their code quality and reliability. This is made worse by the fact that IaC frameworks like Terraform involve a very large provider plugin ecosystem, where more than 4,000 IaC providers each contribute and define their resource blocks. As an example, major providers like AWS could contain more than 200 services, each containing dozens of distinct resource types [2]. Since the overall IaC program comprises many different resources, any bug in any constituent resource could risk infrastructure stability.

Indeed, we have found IaC updates [7] to be a frequent source of reliability problems, leading to subtle yet impactful bugs that could hang the infrastructure and disrupt normal

*Equal contribution.

operation. Update bugs manifest in two forms [8], [9]. Certain failures halt updates midway, leaving infrastructure in unintended states that require a full teardown and rebuild to fix. Others, known as state drift, complete without errors but deviate from user intent [10]. Both are challenging and time-consuming to resolve, placing the burden of fixing on the user and worsened by lengthy cloud update processes [8].

To the best of our knowledge, no systematic approach exists for finding IaC update bugs. Today, IaC users encounter these bugs unexpectedly and then rely on manual troubleshooting to correct the problem. While creating "unit tests" is a standard software engineering practice, creating test cases for *updates* is much harder because of the combinatorial space of plausible changes. One could create test cases for each IaC resource to validate initial deployment, but update tests involve state transitions, where valid start and end states reveal bugs during the update process. Manually authoring and validating these transition test cases at high coverage is difficult and time-consuming. Software testing [11], [12] can be automated using fuzzing tools [13], [14], but these focus on static code behavior. IaC updates, however, require tests that capture both initial configurations and state transitions, significantly increasing complexity compared to traditional testing methods.

To this end, we proposed TerraFault, a tool that automatically generates IaC updates and validates the resulting state transitions to discover update bugs in Terraform. The key challenge that TerraFault faces is the vast search space due to the complex topological structure of IaC programs and the need to reason about state transitions. Our insight is that the capabilities of Large Language Models (LLMs) provide effective assistance in generating and mutating IaC programs, enabling the creation of diverse and robust update test cases. An LLM-powered search assistant also guides the system in navigating the large and complex search space, enhancing efficiency and coverage in identifying potential issues across varied state transitions. We report the technical roadmap of TerraFault and present initial results in identifying impactful bugs even for simple IaC updates.

II. Motivation

We present case studies of update failures to highlight the need for IaC-specific testing tools. Finally, we outline our roadmap for TerraFault, addressing challenges in automated bug discovery and the role of LLM agents.

A. Even Simple IaC Updates Can Produce Bugs

Our experience with Terraform on the Google Cloud Platform (GCP) shows that even simple updates could fail. We classify these updates as structural updates, field updates, and combinations of the two. Notably, all IaC programs in our case study deploy successfully on their own but fail if we update one program to another.

Case 1: Structural updates Update bugs can arise from structural changes to the resource dependency graph, as shown in Figure 2. In this example, the initial configuration only contains a VPC, but the update adds a firewall and a virtual machine (VM) to the cloud infrastructure. While simple,

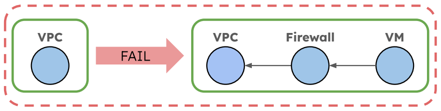


Fig. 2: Case 1: Structural update

this update caused a bug in the provider plugin which is written in Go, where the network attribute was resolved inconsistently between the plan and apply phases. Specifically, it was resolved as a shorthand identifier "projects/..." during the planning phase but expanded to a full URL "https://..." during the apply phase, causing a mismatch and resulting in the error. In order to clean up the failed update, this requires destroying all existing resources and re-deploying them from scratch. Therefore, IaC update bugs have drastic impacts and once triggered, are time-consuming to clean up. Examining the error message manually indicated that this inconsistency is due to a bug in the provider plugin, which needs to roll out a proper fix for their update logic.

Case 2: Field updates These updates modify specific resource attributes, without altering the underlying resource dependency structure, as shown in Figure 3. The initial state includes a VPC network and a route policy, with a long tail dependency chain between them. During our testing,

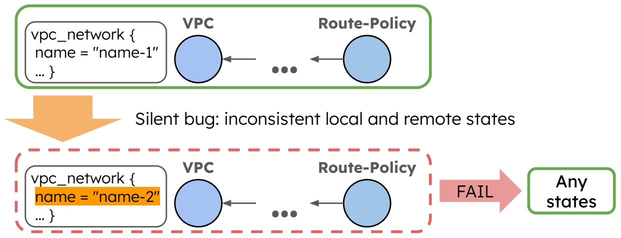
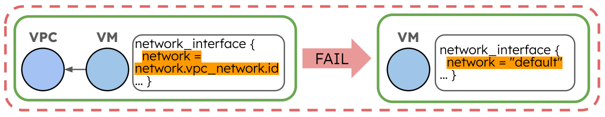


Fig. 3: Case 2: Field update

we observed that simply renaming the VPC can lead to error. Specifically, such a change introduces an inconsistency between the local and remote states: Terraform destroys and recreates all dependent resources except the route policy, disregarding the fact that the policy is one of the dependents. But Google Cloud deletes the route policy when the VPC is re-created, and Terraform is left unaware, resulting in local and cloud states falling out of sync. This inconsistency prevents any further state transitions including rollbacks, trapping the infrastructure in an inconsistent state. Worse still, users are unable to rectify the issue within Terraform, instead needing to manually delete resources through the GCP portal. Notably, while the initial update completes without triggering system alerts, any subsequent update attempts expose the underlying inconsistency, causing them to fail.

Case 3: Field & structural updates Last, we explored a common and complex update type that combines both field and structural changes, which can similarly lead to errors, as shown in Figure 4. We begin with an initial setup where a VM is connected to a VPC. We perform a structural update by removing the VPC and a field update by detaching the VM from the VPC and changing its network interface to "default." This results in an error indicating that the network cannot be



```
network_interface {
  network =
network.vpc_network.id
... }
```

```
network_interface {
  network = "default"
... }
```

Fig. 4: Case 3: Field & structural updates

deleted because it is still in use by the VM, even though the default VPC should exist and handle such cases. To resolve the issue, the users had to destroy all existing resources and re-deploy the final desired state.

These case studies show that, despite significant advancements in IaC frameworks, they remain more robust for initial deployments than for state transitions. To the best of our knowledge, there is no existing tool for testing IaC updates. Consequently, IaC updates remain challenging and understudied, accentuating the need for tools to specifically expose, address, and mitigate IaC update bugs. In this work, we build an automated system to effectively uncover state transition bugs as an initial step.

B. Roadmap, Technical Challenges, and Ideas

Roadmap. TerraFault aims to generate realistic IaC updates and validate them by deploying them to the cloud. For a specific IaC program g, TerraFault generates a family of updates to g, obtaining a set of programs $P(g)$, referred to as "snapshots." We ensure that all programs in $P(g)$ are valid and deployable, and because they are mutations to the same initial g, any two programs in this set would give us a plausible update from one to the other. To test an update, TerraFault first deploys the first program to the cloud, then modifies the program to the second, attempting an update. If the update triggers any bugs, this update is labeled as buggy; we record this update alongside the cloud error message in our bug store. In naïve testing, the process involves either exhaustively iterating through all enumerated updates or randomly sampling updates until a timeout is reached. Instead, we use an LLM-guided search to focus on relevant and promising updates, exposing bugs more effectively. Additionally, we generate a range of representative programs G, and repeat the above process for each $g \in G$. This workflow translates to three technical challenges, each benefiting from the power of LLMs.

Challenge 1 involves generating realistic state transition sequences. To simulate transitions between IaC states, it is essential to understand plausible intermediate configurations. TerraFault models this as a graph mutation problem, where resource dependency graphs are systematically modified to produce a wide range of intermediate states. Specifically, we design mutation operators that modify resource dependency graphs at both the structural and field levels. Structural mutations alter the graph topology by adding, removing, or modifying resources and dependencies, while field-level mutations adjust specific attributes of individual resources, such as configuration parameters, resource names, or versions. To further expand the diversity of the generated states, we also leverage LLMs to suggest updates, utilizing their capability to propose diverse and generalized mutations. This integration significantly expands the range of intermediate states, improving our ability to identify bugs and thoroughly test IaC updates.

Challenge 2 concerns efficiently testing a large number of state transition candidates. Testing these transitions demands an approach that minimizes unnecessary testing to reduce deployment overhead and cost. Our prototype optimizes testing by overlapping initial and final states to reduce deployments and leveraging parallel workers for faster execution. We also employ an LLM agent to dynamically guide the search during testing. By constructing a "bug store," the agent reorders test sequences to prioritize specific transitions, enabling more efficient testing and reducing the time to uncover bugs.

Challenge 3 involves generating comprehensive IaC programs to serve as the initial input for testing. Our idea is to bootstrap the process with a set of programs that are hand-crafted or from online repositories but to use LLMs to enhance their representativeness. By leveraging LLMs, we aim to create diverse IaC programs that include resources not readily available from crawled online repositories, expanding coverage. LLM-assisted generation allows us to explore a broader range of configurations and enhances the system's ability to detect issues across various cloud resources and scenarios.

In the rest of this paper, we describe our ongoing work in tackling these challenges and report initial findings.

III. System Sketch

Figure 5 illustrates the automated bug discovery workflow. In §III-A and §III-B, we describe how TerraFault generates and tests updates for a specific IaC program (addressing Challenge 1 and 2 respectively); in §III-C, we describe how TerraFault generates a wide range of representative IaC programs, which tackles Challenge 3.

A. Generating Program Mutations as Updates

Our goal is to mutate a given IaC program g (e.g., one that is obtained from an online repository) to obtain a family of updates: a pool of realistic programs $P(g)$ that are mutants of g. This allows us to then construct a *transition sequence* by sampling intermediate states from $P(g)$. We define a set of operators for both structural and field mutations, and use LLM-based heuristics to drive the search and obtain realistic updates. We also validate each mutated program to ensure that it successfully compiles and is deployable.

Structural. Given an initial program $g = g_0$, our structural mutation generates a sequence of program variants that we call snapshots: $s(g_0) = \{g_0, g_1, ..., g_k\}$. We currently use a basic mutation operator $del(v)$, which removes the vertex v and its edges from g; in other words, this mutation deletes one cloud resource and its dependencies from the IaC program. To ensure validity, we perform a topological sort so that a removed v does not have other resources that depend on it—i.e., v has an in-degree of zero. The intuition for this method is that the sequence $s(g)$ is obtained by gradual deletion from a representative program g. Therefore, the intermediate states

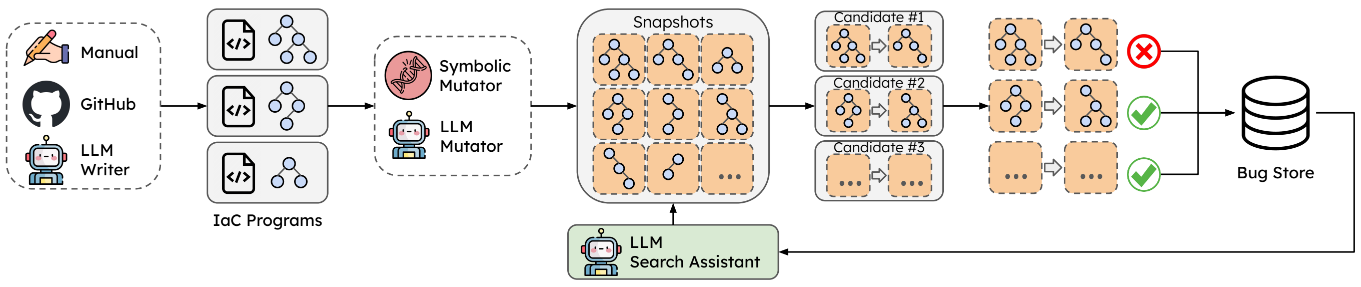


Fig. 5: TerraFault workflow. (§III-C) IaC programs are transformed into resource dependency graphs, (§III-A) mutated into graph variants (snapshots), and sampled to generate state transition candidates. (§III-B) TerraFault tests these candidates, logs failures in the bug store, and uses an LLM agent to reorder untested candidates, enhancing bug discovery during runtime.

in $s(g)$ are not conjured up from thin air; rather, each g_i represents a plausible program that could have occurred in a realistic update. $s(g)$ also forms the basis for field updates, as discussed next.

Field. For each intermediate state in $g_i \in s(g)$, we also perform field updates using an operator $mod(v)$, which modifies some attribute of the vertex v, such as the 'location' of a VM resource from 'us-east' to 'us-west'. Such modifications do not impact resource dependency and therefore, do not change the edges. For each g_i, therefore, field updates produce another set of snapshots: $g_{i0}(= g_i), g_{i1}, ..., g_{il}$ where the programs share the same structure but have mutated fields. When applying this methodology to the sequence of snapshots in $s(g)$, we effectively produce a matrix of programs that represent a family of updates to the program g:

$$\texttt{Mut}\ (g = g_0) = \begin{bmatrix} g_{00} & g_{10} & \cdots & g_{k0} \\ g_{01} & g_{11} & \cdots & g_{k1} \\ \vdots & \vdots & \vdots & \vdots \\ g_{0t} & g_{1t} & \cdots & g_{kt} \end{bmatrix}$$

where $g_{00} = g$ and $\{g_{00}, g_{10}, \cdots, g_{k0}\} = s(g)$. The horizontal dimension represents structural updates and the vertical dimension represents field updates. A hybrid (i.e., field & structural) update can be simulated by picking two snapshots g_{im} and g_{jn} from different rows and columns, or equivalently, $i \neq j \wedge m \neq n$.

LLM mutator. For a large program, an exhaustive search on all possible updates would be prohibitive; at the same time, it may duplicate updates of the same nature leading to inefficiency. Our goal is to exploit LLMs that could suggest diverse mutations as well as better candidates to test, without necessarily materializing the entire matrix symbolically as shown above. We approximate the above workflow by prompting the LLM for structural and field update suggestions, obtaining a partial set of programs in the matrix that are believed to be the most fruitful to test.

B. Efficient Testing of IaC Updates

From the above step, we have obtained a set of updates to be tested in the cloud, represented as $g' \to g''$ where g' and g'' are two elements in the matrix. A naïve solution is to test each update individually, without considering their relation

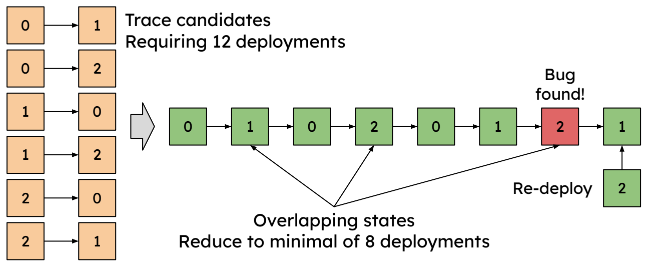


Fig. 6: Leveraging overlap to reduce deployment overheads.

to each other and without any particular order. However, IaC testing is costly and time-consuming, since cloud bills are quite high and resource deployments could take minutes or hours [8]. We describe our solution for efficient testing based upon LLM-powered search.

Scheduling. The first technique is to schedule updates so that each test case builds upon the previous IaC infrastructure, minimizing the amount of required deployments. For instance, a naïve testing that considers $g' \to g''$ and $g'' \to g'$ in isolation would require four deployments. Both g' and g'' are deployed twice, once in each update test case. However, a better strategy is to schedule these updates together, identifying a better sequence $g' \to g'' \to g'$ with only three deployments. This strategy builds upon the insight that a family of updates have a high degree of resource sharing, presenting an opportunity to reuse a deployed IaC infrastructure as much as possible. Figure 6 depicts a more complex example where better scheduling can reduce the testing overhead by one-third. More generally, if we want to test all two-state transitions in n snapshots exhaustively, the total number of deployments $\texttt{NumDep}(n)$ would be $\frac{2 \cdot n!}{(n-2)!}$. Leveraging overlaps would give $\frac{n!}{(n-2)!} + (n-1) \leq \texttt{NumDep}(n) \leq \frac{2 \cdot n!}{(n-2)!}$. The lower bound is achieved when all tests are successful, and each test case builds upon the previous. However, if any update results in the identification of a bug, this interrupts the scheduling and we need to clean up the state and rerun the scheduling algorithm for the remaining test cases. The worst case scenario is upper bounded by the naïve solution. Moreover, although we obtain a family of updates for an initial program p, not all snapshots have resource overlaps. Another optimization is to leverage multiple workers in parallel, with each worker assigned to test a specific subset of the updates. Each worker operates in

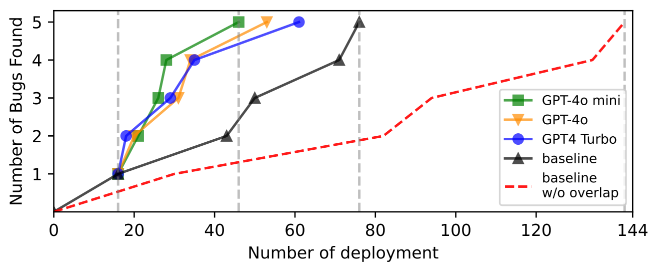


Fig. 7: Bug discovery w/ and w/o the LLM agent.

its own "namespace," so that deployments and failures in one worker's domain are isolated from others.

LLM search assistant. At runtime, when a bug is detected, TerraFault records the failed update, cleans up the environment, and continues testing. Our goal is to uncover as many buggy updates as possible, while ideally avoiding tests that are likely to succeed. We leverage an LLM agent that digests known buggy updates from the bug store and reorders the test case sequences to prioritize the updates that are likely to trigger bugs. The agent analyzes affected resources and error messages to guide testing. It prioritizes high-risk transitions, scoring tests by bug likelihood while maximizing overlaps to optimize discovery within a limited time budget.

C. Generating Initial IaC Programs

Having described how TerraFault systematically generates update test cases for a given IaC program, we now discuss how it curates the initial input IaC programs, which form the basis for the mutation process. The quality of these initial programs is crucial, as they influence the breadth and effectiveness of subsequent testing. To automate their generation, we aim to develop a pipeline capable of producing targeted programs that meet two key criteria: representativeness and high resource coverage.

To ensure representativeness, we plan to develop a web crawler to scrape Terraform GitHub repositories, creating a corpus that captures real-world usage of various cloud resources. This will help identify typical use cases for the most popular resources. Additionally, since online IaC configurations can be extremely large, we aim to prune redundant subgraphs, retaining unique structures. This approach not only preserves diversity for mutation testing but also reduces computational costs by eliminating unnecessary duplications.

LLM writer. For high resource coverage, particularly for newer or less common resources with limited online examples, we propose leveraging LLMs for their advanced code generation capabilities. By conditioning LLMs on documentation [15], we aim to generate minimal yet valid IaC programs that incorporate these resources, thereby improving overall coverage.

IV. Initial Results

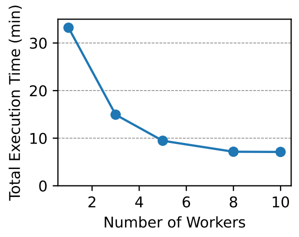
We have built a preliminary prototype for GCP (Google Cloud Platform), which has successfully discovered bugs even in simple IaC updates. We report the initial results using the three IaC programs described in §II-A: Case 1 (Firewall), Case 2 (Router Policy), and Case 3 (Disk). Our evaluation focuses on the efficiency of bug discovery, and we refer readers back to §II-A for the discovered bugs. Table I presents the statistics for the three tested IaC programs, detailing the resources, mutated programs (snapshots) generated by TerraFault, tested updates, and identified bugs for each program. For each program, TerraFault has identified specific updates that trigger the bugs in §II-A.

TABLE I: Summary of bug finding results.

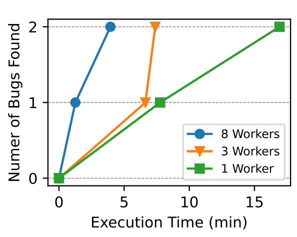
TF Program	#Resources	#Snapshots	#Updates	#Bugs
Firewall	3	10	63	13
Router Policy	4	12	143	5
Disk	3	11	121	5

Figure 7 demonstrates the effectiveness of TerraFault in scheduling IaC testing. We conducted two baseline experiments using exhaustive search: one with overlapping enabled and the other without. By enabling overlapping between initial and final states, it reduces $\texttt{NumDep}(n)$ across the board, achieving a 47% reduction in deployments and resulting in significant cost savings in both cloud expenses and testing time, especially for larger IaC configurations. Additionally, we evaluated the impact of the LLM search assistant. Upon detecting the first bug, the bug store is updated, and the LLM agent analyzes state deltas and reorders untested transitions. When both the LLM search assistant and overlapping are enabled, the number of deployments was reduced even further, achieving a 68% reduction. We also assessed the performance of various LLMs, including GPT-4o, GPT-4 Turbo, and GPT-4o Mini, observing similar testing efficiency across models. Smaller models achieve similar reductions in deployments and bug discovery rates as larger ones, offering significant savings in LLM API costs. Based on manual inspection, we observed that bugs with similar root causes can vary in blast radius; for example, some, like in Case 1, have a broader impact, while others, like in Cases 2 and 3, are more localized. Experimenting with a wider range of programs and providers, along with bug categorization and root cause analysis, is deferred to future work.

Figure 8a highlights the benefits of parallelization using multiple workers in testing IaC updates. We ran the experiment on a transition sequence of length 35, and we used execution time and bug discovery rate as key metrics. Using more than eight workers, we observed a 5x performance improvement compared to a single worker, with the average time to deploy one snapshot reduced to approximately 12.6 seconds. However, the difference between eight and ten workers was minimal, as the maximum sequence length among all workers became nearly identical, creating a bottleneck. Figure 8b illustrates the execution time to discover bugs with varying numbers of workers. With eight parallel workers, the time to detect bugs was reduced significantly, approximately four times faster than using a single worker. This demonstrates the efficiency of parallel execution in accelerating bug discovery.



(a)



(b)

Fig. 8: Parallelis testing speeds up bug discovery.

V. Related Work

Program/configuration testing. Software testing is a well-studied area, but most of these tools do not focus on update testing. Instead, existing work [11], [12] focuses on code correctness, overlooking state transitions that likewise could be a source of bugs. Similar tools [16], [17] exist for configuration testing but focus on validating specific setups rather than updates. Fuzzing [13], [18] is a common bug detection technique but is not typically specialized for updates.

Update testing. Acto [19] considers state transitions for Kubernetes operators, which operate within a more confined and predictable search space than IaC programs. TCP-Fuzz [20] focuses on TCP state transition testing, which is also substantially different from IaC updates. Cloud IaC environments require custom techniques due to the complexity of the programs, and the high costs and time demands of comprehensive testing—these are the goals of TerraFault.

IaC testing. Existing tools can scan IaC programs to detect bad coding practices and also detect configuration drift [8], [9], [19]. However, they cannot detect, analyze, and address update bugs, which could manifest in different ways depending on the IaC provider. Zodiac [8] discovers deployment bugs of IaC programs but does not address updates. Recognizing the critical importance of infrastructure updates, TerraFault is the first work to generate and validate IaC updates in an automated manner.

VI. Conclusion

We have described TerraFault, an automated tool for discovering and analyzing bugs in Infrastructure-as-Code (IaC) updates. These update bugs pose risks to cloud reliability, and they are hard to discover due to complexity and user-specific update scenarios. TerraFault leverages LLM agents for program mutation, test scheduling, and high-coverage IaC program generation. Our initial experience shows promising results and a full development of TerraFault will help IaC developers and cloud providers mitigate update pitfalls, enhancing cloud reliability.

Acknowledgment

We thank the anonymous reviewers for their helpful reviews. This work was partially supported by NSF grants CNS-1942219, CNS-2106751, CNS-2106388, CNS-2214272, and a VMware Early Career Faculty Grant.

References

[1] K. Morris., *Infrastructure as Code: Managing Servers in the Cloud.* O'Reilly Media, Incorporated, 2016.

[2] P. T. J. Kon, J. Liu, Y. Qiu, W. Fan, T. He, L. Lin, H. Zhang, O. M. Park, G. S. Elengikal, Y. Kang, *et al.*, "Iac-eval: A code generation benchmark for cloud infrastructure-as-code programs," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[3] Y. Qiu, P. T. J. Kon, J. Xing, Y. Huang, H. Liu, X. Wang, P. Huang, M. Chowdhury, and A. Chen, "Simplifying cloud management with cloudless computing," in *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*, pp. 95–101, 2023.

[4] "Terraform by Hashicorp," https://www.terraform.io/.

[5] "Opentofu," https://opentofu.org/.

[6] "Pulumi," https://www.pulumi.com/.

[7] J. Lepiller, R. Piskac, M. Schäf, and M. Santolucito, *Analyzing Infrastructure as Code to Prevent Intra-update Sniping Vulnerabilities*, pp. 105–123. 03 2021.

[8] Y. Qiu, P. T. J. Kon, R. Beckett, and A. Chen, "Unearthing semantic checks for cloud infrastructure-as-code programs," in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2024.

[9] "Terrascan," https://runterrascan.io/.

[10] Snyk, "How to detect and prevent configuration drift," 2024.

[11] O. Crameri, N. Knezevic, D. Kostic, R. Bianchini, and W. Zwaenepoel, "Staged deployment in mirage, an integrated software upgrade testing and distribution system," in *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, (New York, NY, USA), p. 221–236, Association for Computing Machinery, 2007.

[12] Z. Li, Q. Cheng, K. Hsieh, Y. Dang, P. Huang, P. Singh, X. Yang, Q. Lin, Y. Wu, S. Levy, and M. Chintalapati, "Gandalf: An intelligent, End-To-End analytics service for safe deployment in Large-Scale cloud infrastructure," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, (Santa Clara, CA), pp. 389–402, USENIX Association, Feb. 2020.

[13] K. Even-Mendoza, A. Sharma, A. F. Donaldson, and C. Cadar, "Grayc: Greybox fuzzing of compilers and analysers for c," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, (New York, NY, USA), p. 1219–1231, Association for Computing Machinery, 2023.

[14] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *21st USENIX Security Symposium (USENIX Security 12)*, (Bellevue, WA), pp. 445–458, USENIX Association, Aug. 2012.

[15] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. tau Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive nlp tasks," 2021.

[16] S. Ma, F. Zhou, M. D. Bond, and Y. Wang, "Finding heterogeneous-unsafe configuration parameters in cloud systems," in *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, (New York, NY, USA), p. 410–425, Association for Computing Machinery, 2021.

[17] X. Sun, R. Cheng, J. Chen, E. Ang, O. Legunsen, and T. Xu, "Testing configuration changes in context to prevent production failures," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 735–751, USENIX Association, Nov. 2020.

[18] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2312–2331, 2021.

[19] J. T. Gu, X. Sun, W. Zhang, Y. Jiang, C. Wang, M. Vaziri, O. Legunsen, and T. Xu, "Acto: Automatic end-to-end testing for operation correctness of cloud system management," in *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, (New York, NY, USA), p. 96–112, Association for Computing Machinery, 2023.

[20] Y.-H. Zou, J.-J. Bai, J. Zhou, J. Tan, C. Qin, and S.-M. Hu, "TCP-Fuzz: Detecting memory and semantic bugs in TCP stacks with fuzzing," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pp. 489–502, USENIX Association, July 2021.