Metamorphic Testing for Infrastructure-as-Code Engines

DAVID SPIELMANN, University of St. Gallen, Switzerland
GEORGE ZAKHOUR, University of St. Gallen, Switzerland
DOMINIK ARNOLD, University of Zurich, Switzerland
MATTEO BIAGIOLA, University of St. Gallen and Università della Svizzera italiana, Switzerland
ROLAND MEIER, armasuisse, Switzerland
GUIDO SALVANESCHI, University of St. Gallen, Switzerland

Infrastructure-as-Code (IaC) engines, such as Terraform, OpenTofu, and Pulumi, automate the provisioning and management of cloud resources. They parse IaC specifications and orchestrate the required actions, making them the backbone of modern clouds, and critical to the reliability of both the underlying infrastructure and the software that depends on it. Despite this importance, this class of systems has received little attention: prior work largely targets the correctness of IaC programs rather than the IaC engines themselves. Existing test suites rely on manually written oracles and struggle to expose faults that manifest across multiple executions, leaving a significant reliability gap.

We present EMIaC, a metamorphic testing framework for IaC engines. EMIaC defines metamorphic relations as graph-based transformations of IaC programs and checks invariants across executions of the original and transformed programs. A central novelty is our use of *e-graphs* in software testing, as both a test-input generator and an equivalence oracle. E-graphs compactly represent program equivalences, enabling the systematic generation of large spaces of equivalent IaC programs. To ground these relations, we analyze 43,593 real-world Terraform programs and show that IaC dependency graphs are typically small and sparse, making e-graphs a natural fit.

Evaluating EMIaC on Pulumi, Terraform, and OpenTofu, we show that it complements existing test suites by exercising engine-critical code paths and covering 98 previously untested statements in Terraform and 1,313 in Pulumi. EMIaC also uncovers previously unknown issues in all three test suites, improving their adequacy. Three test cases have been merged into Terraform's main branch, and Pulumi has merged a specification fix.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; *Formal methods*.

Additional Key Words and Phrases: Software Testing, Infrastructure as Code, E-Graphs

ACM Reference Format:
David Spielmann, George Zakhour, Dominik Arnold, Matteo Biagiola, Roland Meier, and Guido Salvaneschi. 2026. Metamorphic Testing for Infrastructure-as-Code Engines. *Proc. ACM Program. Lang.* 10, OOPSLA1, Article 118 (April 2026), 29 pages. https://doi.org/10.1145/3798226

1 INTRODUCTION

The widespread adoption of cloud computing has led to increasingly complex cloud environments, where even small configuration errors result in significant security, availability, or cost-related

Authors' addresses: David Spielmann, david.spielmann@unisg.ch, University of St. Gallen, St. Gallen, Switzerland; George Zakhour, george.zakhour@unisg.ch, University of St. Gallen, St. Gallen, Switzerland; Dominik Arnold, University of Zurich, Zurich, Switzerland; Matteo Biagiola, matteo.biagiola@{unisg,usi}.ch, University of St. Gallen and Università della Svizzera italiana, St. Gallen/Lugano, Switzerland; Roland Meier, roland.meier@ar.admin.ch, armasuisse, Thun, Switzerland; Guido Salvaneschi, guido.salvaneschi@unisg.ch, University of St. Gallen, St. Gallen, Switzerland.

ACM 2475-1421/2026/4-ART118
https://doi.org/10.1145/3798226

issues [21, 45, 73]. Infrastructure as Code (IaC) has emerged as a dominant paradigm for managing these environments, allowing developers to define cloud infrastructure as source code. As a result, infrastructure specifications go through a well-established software lifecycle not different from traditional software, including, e.g., version control and bug fixing, greatly simplifying the operation of cloud applications. IaC engines such as Pulumi [71], Terraform [33], and OpenTofu [53] automate the creation, updating, and deletion of infrastructure based on these specifications.

While IaC engines improve reproducibility and mitigate common human errors, the correctness of such engines remains a critical, under-tested risk surface. They interface with modular *providers*, which are platform-specific plugins responsible for interacting with the APIs of individual cloud service providers, e.g., Amazon Web Services (AWS). The core of each IaC engine orchestrates resource lifecycles in a platform-agnostic manner. In contrast, each provider implements the concrete operations required to provision, update, and delete resources on its respective platform. A single IaC engine supports dozens of providers, all of which build on the shared logic implemented by it. Thus, a bug in the engine propagates across all providers, potentially compromising infrastructure deployments on multiple platforms and affecting all software stacks running on such deployments. In summary, IaC engines have become critical to modern IT infrastructure, and their correctness should be regarded as being as vital to system operations as compilers and network protocols.

The central abstraction that enables this orchestration is the *resource graph*. An IaC program is compiled into a graph whose nodes are resources, e.g., a virtual machine, a database, or a storage bucket, and edges are dependencies, e.g., a database must be created after its VM. The engine maintains this graph as persistent state across deployments: rather than recreating the entire infrastructure, it compares the graph in the current state to the one implied by the new program and computes the minimal sequence of create, update, and delete operations needed to reconcile them. Correctness of this reconciliation is essential: even small errors in graph construction, traversal, or update logic, lead to inconsistencies that ripple through an entire deployment.

Such fragility has led to documented issues. Real-world reports from Terraform's issue tracker illustrate how delicate this reconciliation is [29, 30]. In one case, the engine inferred a spurious cycle in the resource graph when certain lifecycle options (such as create_before_destroy) were combined with resource removals, causing deployments to halt even though no real cycle existed [29]. In another case, the engine failed to maintain a necessary dependency edge between resources, leaving the order of operations undefined [30]. Such a fault can lead to situations where a resource is modified or removed before its dependents are safely removed, producing subtle misconfigurations. Both examples stem from small mistakes in reasoning about the resource graph and highlight that even mature engines are susceptible to such errors. Even worse, bugs in IaC engines occasionally manifest not as crashes, but as subtle deviations in the resource graph [12, 31]. For example, a missing edge may result in a resource being created too early, violating a dependency; a missing node may silently omit a resource altogether; or an incorrect diff may remove critical infrastructure. In all of these cases, the deployment may appear to succeed, producing no errors or warnings, while leaving the infrastructure in an incorrect or insecure state. Such silent faults undermine the very reproducibility and reliability that motivate the adoption of IaC in the first place.

In summary, ensuring the reliability of IaC engines is essential, but challenging for two main reasons. First, the *oracle problem* [3]: for a given IaC program there is rarely a ground-truth infrastructure state to compare against, especially under incremental updates where state evolution matters. Second, realistic tests often require interacting with real cloud services, which is slow, costly, flaky [54], and hard to scale.

In this paper, we address these challenges with EMIAC, a fully automated, tool-agnostic framework for IaC engines that reasons directly on resource graphs. We address the oracle problem using *metamorphic testing* [10], which exploits relations between multiple executions to validate

program behavior without relying on a conventional, single-execution oracle. Specifically, rather than validating a single sequence of IaC program deployments, we apply semantic-preserving transformations to this sequence and verify that a resulting sequence of IaC programs converges to the same resource graph when executed one after the other. If two semantically equivalent sequences produce divergent deployments then a bug is found. Additionally, we address the challenge of system test execution by making use of mock providers that simulate resource creation without invoking external cloud APIs. This design choice is appropriate because an IaC engine is agnostic w.r.t. the provider internals: its responsibility is to construct and reconcile the resource graph, not to perform the low-level API calls. As a result, mock providers are ideal for isolating and testing engine correctness without introducing the cost and flakiness of real cloud executions.

Two design decisions make EMIAC both practical and general. First, we introduce a compact *Intermediate Representation (IR)* for describing resource graphs and their transformations. The IR captures the core engine-level operations—add, remove, connect, and disconnect—which capture the manipulation of resource graphs and act as a core calculus with which we can reason about metamorphic testing formally. By abstracting away tool syntax and provider specificities, the IR allows us to reason about IaC engines uniformly while still being able to transpile programs back into concrete IaC programs suitable for Pulumi, Terraform or OpenTofu. Second, we use *e-graphs* [50], a data structure that compactly represents large spaces of equivalent programs. Not only can it serve as an equivalence oracle, i.e., two programs end up in the same equivalence class if and only if they compute the same graph, but it can also serve as a powerful test generator, enumerating wildly different equivalent variants from a set of rewrite rules. Our core design choice is to use as the metamorphic relation the *semantic equivalence of programs*. Equivalence is the most precise relation in this setting, because it requires identical resource graphs and therefore minimizes false positives and false negatives compared to weaker relations such as sub-/super-graph relations. Given a single test input, there are infinitely many programs that are equivalent w.r.t. the resource graph. This choice makes the oracle both precise and constructive, enabling us to generate diverse equivalent variants from a single seed while checking their convergence.

Crucially, IaC engines are stateful: they maintain a persistent resource graph across deployments and update it incrementally. To test this behavior, EMIAC extracts multiple equivalent variants from a canonical IR program, splitting each into batches that incrementally evolve the graph. Each update represents a realistic change that the engine must reconcile correctly. This lifecycle-aware testing enables validation of update sequences such as incorrect diffs or misordered operations.

We evaluate EMIAC on all the three major open-source IaC engines, Pulumi, Terraform, and OpenTofu [22]. Our analysis covers 43,593 public IaC programs from 7,283 repositories, revealing that their resource graphs are small, shallow, and sparse. We further demonstrate the effectiveness of our IR by showing that EMIAC is tool-agnostic. EMIAC complements official test suites by covering 98 previously untested statements in Terraform and 1,313 in Pulumi. Mutation analysis highlights its complementary strength, as EMIAC kills additional mutants that existing suites miss. EMIAC has already produced concrete impact: Terraform merged three of our test cases into the main branch after our mutation analysis revealed real gaps in their tests, and Pulumi fixed a specification error in its documentation. Finally, EMIAC generates tests faster than the baseline rewriting approach, achieving higher diversity in less time. As the number of generated tests increases, the diversity of EMIAC-generated tests remains consistently high, degrading far less than that of tests generated by the baseline approach.

Contributions. This paper makes the following contributions:

(1) We present the first testing framework for IaC engines, addressing the oracle problem via semantic equivalence over resource graphs.

(2) We introduce the use of *e-graphs* and *equality saturation* in software testing to model program equivalences and automatically generate test inputs.
(3) We design an IR for a theoretical framework connecting metamorphic testing, IaC, and e-graphs. The IR also allows expressing tool-independent resource graph transformations that we compile to major IaC engines.
(4) We provide the first empirical characterization of real-world IaC resource graphs by analyzing public IaC programs from the largest available dataset.
(5) We implement our technique in the tool EMIaC and evaluate it on the major IaC engines, demonstrating that it complements official test suites by covering 98 previously untested statements in Terraform and 1,313 in Pulumi, killing additional mutants missed by existing suites, and producing concrete impact with three test cases merged into Terraform and a specification fix in Pulumi.

2 BACKGROUND

In this section, we introduce the key concepts about metamorphic testing, Infrastructure as Code, and e-graphs that will be used in the rest of the paper.

2.1 Metamorphic Testing

A core challenge in software testing is the *oracle problem* [3]: distinguishing the desired, correct behavior from potentially incorrect behavior based on test outputs. In many domains, constructing a reliable oracle is costly, time-consuming, or even impossible. *Metamorphic Testing* (MT), first introduced in 1998 [10], addresses this challenge. The key idea of MT is that checking the relationship between the outputs of related inputs is possible even though the correctness of individual outputs may not be. *Metamorphic relations* (MRs) formalize such relationships and enable (1) transforming existing *source* test cases into new *follow-up* test cases, and (2) validating the program behavior without a conventional oracle. Following Chen et al. [10], an MR is defined as follows:

*Definition 2.1 (**Metamorphic Relation**).* Let f be the system under test, and $\mathcal{I}_R$ and $\mathcal{O}_R$ be input and output relations respectively. Then a *Metamorphic Relation* ($\mathcal{MR}$) is the property that for every test cases $t_1, \ldots, t_n$ such that $\mathcal{I}_R(t_1, \ldots, t_n)$ then $\mathcal{O}_R(f(t_1), \ldots, f(t_n))\}$.

A $\mathcal{MR}$ can be used as a test oracle since once an input relation $\mathcal{I}_R$ holds, the output relation $\mathcal{O}_R$ must hold as well: a faulty behavior of f is triggered whenever it is not the case. Given an input relation $\mathcal{I}_R$ and a source test case t_1, which could be given or generated, t_1 is transformed into a follow-up test t_2 using an input transformation [1]:

*Definition 2.2 (**Metamorphic Transformation**).* Given $\mathcal{I}_R$, t_1, t_2, and some transformation $\mathcal{T}$. Then $\mathcal{T}$ is said to be a *Metamorphic (Input) Transformation* whenever $t_1 \xrightarrow{\mathcal{T}} t_2$ satisfies $\mathcal{I}_R$.

Metamorphic tests are generated from source test cases by applying input transformations to produce follow-up tests. Many metamorphic transformations can be composed sequentially to transform a single source test case into a follow-up test case [56].

Example. A classic example of MT is testing the sine function. Instead of unit testing it using a sine table, we can use the trigonometric identity $\sin(t) = \sin(\pi - t)$ as a $\mathcal{MR}$. In particular, $\mathcal{I}_R = \{(t, \pi - t) : t \in \mathbb{R}\}$ and $\mathcal{O}_R$ is equality. The metamorphic transformation which satisfies this MR is simply $t_1 \xrightarrow{\mathcal{T}} \pi - t_1$. Thus, MT enables correctness checking without an explicit oracle.

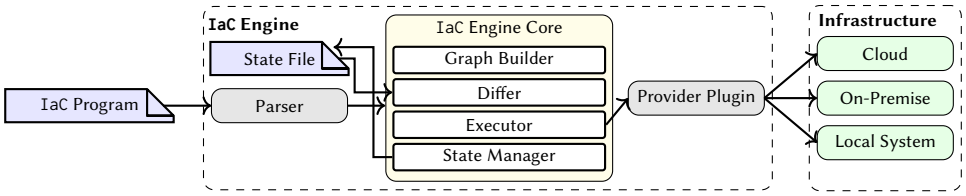


Fig. 1. Deployment workflow of an IaC program and the key components of an IaC engine.

2.2 Infrastructure as Code

IaC refers to automating the provisioning and the maintenance of computing infrastructures and systems via a code-based description of the infrastructure. The focus of this work is on provision-based IaC engines, which help provision, maintain, and update cloud infrastructure. This contrasts with configuration-based IaC tools, such as Ansible, Puppet, and Chef, which are primarily used to install and manage software within an already existing infrastructure [19]. Typically, provision-based IaC engines interact with cloud platforms such as AWS. The most popular provision-based IaC engines are Pulumi [71], OpenTofu [53], Terraform [33], and AWS CloudFormation [63].

IaC engines allow IaC programs to specify the target *state* of the deployment in a declarative way. All configuration-based engines and some provision-based ones (e.g., Terraform) only support configurations (e.g., JSON) or domain-specific languages (e.g., HCL for Terraform and YAML for Ansible) for the specification of IaC programs. On the other hand, Pulumi, AWS Cloud Development Kit (CDK) [62], and CDK for Terraform (CDKTF) [32] allow developers to specify IaC programs using general-purpose programming languages [65].

IaC Deployment Workflow. The typical deployment workflow of an IaC program is in Figure 1. An *IaC Program* declaratively specifies the desired infrastructure state. The *Parser* processes the program and produces an intermediate representation consumed by the *Core*. Within the core, the *Graph Builder* constructs a *resource graph*: a dependency graph of the resources. The *Differ* then compares this desired state with the current infrastructure state, typically recorded in a *State File*, and determines a set of changes. Based on this plan, the *Executor* issues dependency-respecting operations to transition the infrastructure into the target state. To apply these changes, the core interacts with *Provider Plugins*, which act as adapters to concrete platforms and perform operations via their APIs. These providers are responsible for translating the abstract operations into concrete API calls to the target environment. After execution, the *State Manager* updates the state file to reflect the new infrastructure state. While cloud platforms such as AWS are the most common, providers may also manage resources on premise or even on the local system.

Example. Section 2.2 shows a Terraform IaC program that provisions a storage service for a photo-sharing application. It begins with a provider block that configures the region to be us-east-1. Next, the IaC program declares an aws_s3_bucket resource named photo_bucket, an Amazon S3 bucket to store application images. Next, it declares an aws_s3_object resource called welcome_file, that depends on the bucket which specifies that the object must be placed in the S3 bucket. This object is initialized with the key welcome.txt and contains the text "Welcome!"

```
1 provider "aws" { region = "us-east-1" }
2 resource "aws_s3_bucket" "photo_bucket" {
3   bucket = "sample-photo-app-images" }
4 resource "aws_s3_object" "welcome_file" {
5   bucket = aws_s3_bucket.photo_bucket.id
6   key = "welcome.txt"
7   content = "Welcome!" }
```

Listing 1. Terraform example for a photo-sharing app: an S3 bucket with an initial object.

When executed, the core builds a resource graph with two nodes, one for each resource, with an edge from photo_bucket to welcome_file. If the state file is empty, the differ determines that both resources need to be created. The executor then invokes the AWS provider to create the S3 bucket and then the object in it. Finally, the state file is updated with the provider configuration and the existence of both resources, preventing future runs from recreating resources needlessly.

This example illustrates the key responsibilities of deployment engines in provision-based IaC engines: (1) providers mediate access to concrete platforms; (2) resources are declared in a high-level, declarative manner; (3) dependencies are inferred from references; and (4) the system maintains a state that enables reproducible and idempotent deployments.

2.3 E-Graphs

The e-graph [50] is a data-structure to efficiently store and query from the smallest reflexive, symmetric, transitive, and congruent closure of a set of equalities. It has been applied successfully to many domains such as automated theorem proving [18], SMT solving [16], compiler optimizations [39], and program synthesis [49]. It offers two operations: (1) *find*, that retrieves the equivalence class of a given expression, and (2) *merge*, sometimes called *union*, that merges two equivalence classes into a single one. Two expressions are equal if they have the same equivalence class, and two expressions can be made equal by merging their equivalence classes. The e-graph is a graph with two kinds of nodes: (1) e-classes, representing equivalence classes, and (2) e-nodes, representing expressions. E-graphs have also two kinds of edges: (1) class–node, representing membership in an equivalence class, and (2) node–class representing congruence.

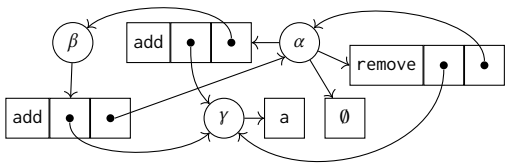
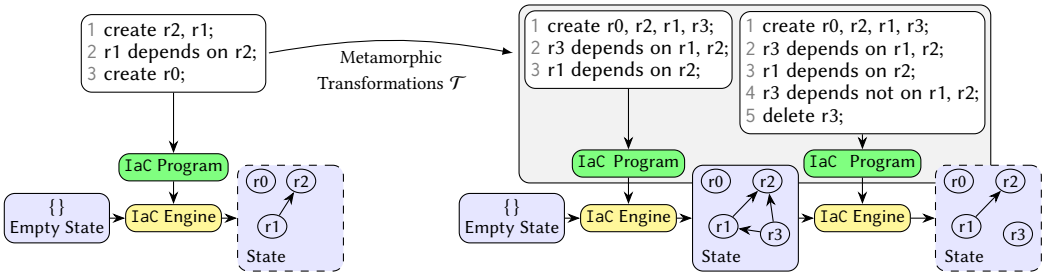


Fig. 2. The e-graph generated from: $\emptyset$=remove(a, $\emptyset$) and remove(a, add(a, $\emptyset$)) = $\emptyset$. E-classes are circles and e-nodes are multicell rectangles.

E-graph example. Figure 2 shows the e-graph built from the two equalities $\emptyset$ = remove(a, $\emptyset$) and remove(a, add(a, $\emptyset$)) = $\emptyset$. Expressions are drawn from a grammar of either terminal symbols, e.g. a and $\emptyset$, or function applications, e.g., add(a, $\emptyset$) and remove(a, $\emptyset$). E-classes are represented as circles labeled with Greek letters and e-nodes using multicell rectangles whose first cell is an identifier and remaining cells are holes. An e-node belongs to an e-class when the two are connected by an edge. For example, $\emptyset$ belongs to α and thus $find(\emptyset) = \alpha$. When an edge connects an e-class to a hole in an e-node, all expressions built by replacing the hole with any e-node belonging to the e-class are equal. For example, the two expressions add(a, $\emptyset$) and add(a, remove(a, $\emptyset$)) are equal since both $\emptyset$ and remove(a, $\emptyset$) belong to α, i.e., replacing the last two expressions in the e-node add(γ, α) produces the previous expressions. We also say that two e-nodes in an e-class are equal, thus $\emptyset$ and remove(a, $\emptyset$) are equal since they both belong to α. In the example, the e-graph has three equivalence classes: α, β, and γ. The equivalence class γ has a finite number of expressions, only a, while both α and β contain an infinite number of expressions.

Equality Saturation. In Figure 2, the functions add and remove are *uninterpreted* as nothing can be said about their definition besides simple equalities. To provide more meaning to the uninterpreted functions in the e-graph we may *saturate* the e-graph with respect to some rewrite rules [70]. For example, saturating the e-graph of Figure 2 with the rule add(x, add(y, g)) $\rightarrow$ add(y, add(x, g)) consists of three operations: (1) *e-matching* [15], which searches the e-graph for all expressions matching the left-hand-side and obtains bindings for the variables x, y, and g; (2) *rewriting*, which applies the bindings to the right-hand-side to obtain a new expression, and (3) *merging*, which equates the newly added expression to the match. Crucially, variables in a rewrite rule are bound

to e-classes and not e-nodes, this effectively applies a large (potentially infinite) amount of rewrites in one step. While this process may diverge, timeouts and amortized rebuilding [74] can make it practical. To the best of our knowledge, we are the first to apply e-graphs to testing. **David:** ►*Fix*◄



(a) Resource graph after source test case.

(b) Resource graphs after follow-up test case.

Fig. 3. Terraform's resource graphs after the execution of source (Figure 3a) and follow-up test cases (Figure 3b). The resource graphs in the dashed boxes are different, violating the **MR** hence killing the Terraform mutant.

3 EMIAC IN A NUTSHELL

We now present an example that demonstrates how EMIAC reveals subtle bugs in IaC engines. We consider a *mutant* of the Terraform source code, i.e., a version of Terraform where a small syntactic change has been introduced as part of our mutation analysis evaluation (Section 6.4). Mutation analysis is used to assess the quality (or adequacy) of a test suite, which is determined by the number of mutants it can kill [17], i.e., the number of mutants for which the test suite produces a failing execution. Specifically, the mutant consists of injecting a single `break` statement into the code[1]. Although this change is minimal, it alters the control flow, causing the deployment engine to behave incorrectly under certain conditions. EMIAC is able to uncover this bug (or kill the mutant), whereas the official Terraform test suite does not.

Figure 3 shows the resource graphs that Terraform produces when executing the source test case (Figure 3a) and the follow-up test cases (Figure 3b). EMIAC randomly generates the *source test case* in the IR, which is then transpiled into the IaC program (top part of Figure 3a). The source test case defines three resources (r0, r1, and r2), with one dependency (r1 depends on r2). After executing the corresponding transpiled IaC program from an empty state, the IaC engine produces the resource graph shown in the dashed box on the right-hand side of Figure 3a.

Then, EMIAC applies the metamorphic input transformations $\mathcal{T}$ to the source test case through its e-graph, generating a follow-up test case that is semantically equivalent to the source test case. In particular, the follow-up test case corresponds to the IR with five lines shown on the right-hand side of Figure 3b. This follow-up test introduces one additional resource w.r.t. the source test, namely r3, which depends on r1 and r2 (Line 2). However, Line 4 removes this dependency, and Line 5 deletes the resource itself; thus, the follow-up test remains equivalent to the source test.

To exercise all the core components of the IaC engine, including the Differ (Figure 1), EMIAC divides the follow-up test into two parts (or batches). The first batch is the IR with three lines shown on the left-hand side of Figure 3b, while the second batch is the whole follow-up test on the right-hand side. As IaC engines are stateful, they should be able to handle incremental updates that result in the same resource graph. Both batches are transpiled into IaC programs; the first batch produces the resource graph in the middle of Figure 3b (box with a solid line). Then, the whole

[1]In particular, the break statement replaces the following line of code: https://github.com/hashicorp/terraform/blob/33aa0f719b08aa42217777a43df597eefabc4448/internal/states/state.go#L362

follow-up test is transpiled and executed. In this case, the IaC engine first needs to compute the difference between the current state and the desired state specified in the transpiled version of the follow-up test. The expected outcome (i.e., the output relation O_R) is that the IaC engine produces the same resource graph as the source test (i.e., equivalence), as the source test is equivalent to the follow-up test (i.e., the input relation $\mathcal{I}_R$ is also an equivalence relation). However, as shown in Figure 3, the states in the dashed boxes differ: resource r3 is not deleted in Figure 3b.

This inconsistency reveals a bug in the IaC engine, effectively killing the mutant; the state after sequential executions diverges from the expected declarative semantics. Next, we describe how our approach generates source and follow-up test cases using e-graphs.

4 APPROACH

In this section, we elaborate on EMIAC's approach to test IaC programs. We describe the goals that guide the decisions taken in the design of EMIAC. Then, we provide an overview of EMIAC and each of its components.

4.1 Design Goals and Key Insights

Goal 1: Unifying theoretical framework for metamorphic testing of IaC. Both programming languages and metamorphic testing (Section 2.1) offer their own distinct theoretical frameworks. We wish to formally present EMIAC, which requires bridging the gap between the two frameworks.
Key insight: Intermediate Representation (IR). All IaC engines manipulate a resource graph. Accordingly, we define an IR for graph manipulations and elaborate on its properties in Section 4.3.2. This enables us to define a syntactic theory that connects resource graphs, metamorphic testing, and e-graphs in Section 4.6, which depends on a concise and sound equational system over graph expressions, which we exploit for the metamorphic transformation and the oracle (Section 4.4).

A pleasant side effect of adopting the IR is that it is IaC engine-agnostic: EMIAC targets virtually any provision-based IaC engine regardless of its features and the different programming languages it offers to its users. We use transpilers to target concrete languages, e.g., HCL and Python (Section 4.3).

Goal 2: Diverse tests from the get-go. Traditionally, metamorphic testing with an equational system starts from the source test program. It chains metamorphic transformations, each applied locally, one step at a time, to produce different follow-up test programs. Each local rewrite incrementally transforms the program, gradually moving it away from the original into a diverse test case. In contrast, EMIAC introduces diversity from the very beginning by exploring equivalence classes thanks to e-graphs. For example, consider testing an arithmetic calculator with the metamorphic transformations $x \times y = y \times x$, $x + 0 = x$, $y \times 1 = y$, and $(x + y) \times z = x \times z + y \times z$; the output relation states that the execution of the source and the follow-up programs should be the same. Given the expression 5 as a starting source test, we reach the follow-up test $5 \times 1 + 5 \times 0$ by chaining at least six metamorphic transformations. Our goal is to generate this follow-up test faster.
Key insight: Semantics-driven generation through random e-graph walks. We achieve this goal by using the source program semantics to guide a generator. In the example, the generator might choose to generate an equivalent expression of the form $\cdot + \cdot$, and choose a multiplication for each argument, producing an expression of the form $(\cdot \times \cdot) + (\cdot \times \cdot)$, thus reaching the desired shape in three steps instead of six. Thanks to e-graphs, the follow-up program is obtained by simply walking a graph, achieving this goal efficiently (Section 4.4).

Goal 3: Fast generation of diverse tests. Test programs can be efficiently produced by walking the e-graph. However, the e-graph must first be constructed. Our goal is to amortize this cost.
Key insight: Eager e-graph saturation and caching. As we will discuss in Section 4.4.2, this construction step is performed once as a pre-processing phase. The key insight is that a saturated

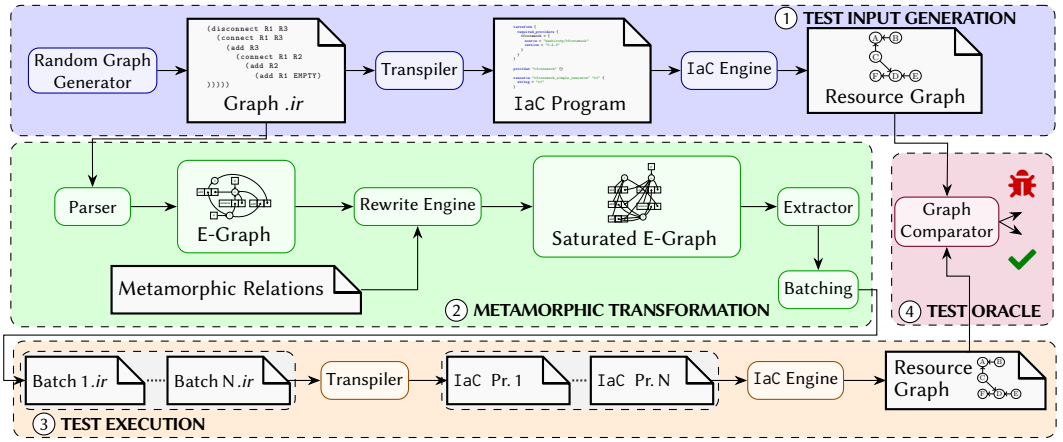


Fig. 4. Overview of EMIaC, which consists of four main phases, namely ① Test Input Generation, ② Metamorphic Transformation, ③ Test Execution and ④ Test Oracle.

e-graph *already intensionally models all possible test programs.* As such the e-graph incorporates all the relevant metamorphic transformations; it only needs to be built once and after this initial cost, arbitrarily many diverse follow-up test programs can be generated efficiently by walking it.

4.2 Overview of EMIaC

EMIaC encompasses four phases (Figure 4): (1) *Test Input Generation* (Section 4.3) takes as input graph parameters, e.g., a bound on the number of nodes and edges or a random seed, and generates a IaC source test input in our IR and transpiles it; (2) *Metamorphic Transformation* (Section 4.4) uses a saturated e-graph to transform this input into multiple equivalent variants that are split into deployment batches; (3) *Test Execution* (Section 4.5) runs each batched variant on the IaC engine to obtain its corresponding resource graphs and (4) *Test Oracle* compares the resource graphs of each variant to that of the initial test input to identify potential bugs in the IaC engine.

4.3 Test Input Generation

4.3.1 Random Graph Generator. The first step in metamorphic testing is the generation of the source test case. In the context of IaC engine testing, a test case is a resource graph. Thus, we employ a *random graph generator* to generate test cases, which ensures exploration of different graph topologies and produces resource graphs with controlled characteristics. Given the number of resources and the maximum number of edges, the generator constructs nodes and randomly connects them with directed edges. To guarantee acyclicity, all edges are directed from lower-indexed nodes to higher-indexed nodes. Randomization is central to this process. By sampling both the number of edges and their placement subject to DAG constraints, the generator produces a broad spectrum of graph topologies, from sparse linear chains to denser structures with multiple fan-in and fan-out patterns. This enables large-scale and efficient test generation: even with fixed parameters, repeated runs yield structurally distinct graphs. As a result, the generator mitigates overfitting to specific graph structures and improves coverage of potential edge cases.

4.3.2 Resource Graph Intermediate Representation. We now introduce the intermediate representation (IR) of graphs. The IR is central to our treatment of IaC, metamorphic testing, and e-graphs in

the same theoretical framework. Pleasantly, it also abstracts the specifics of each IaC engine. The IR is defined by the following syntactic rules where node identifiers range over n and m.

$$G ::= \texttt{empty} \mid (\texttt{add}\ n\ G) \mid (\texttt{rem}\ n\ G) \mid (\texttt{con}\ n\ m\ G) \mid (\texttt{disc}\ n\ m\ G)$$

The empty expression represents the empty resource graph, add expresses adding a node n to a resource graph G while rem expresses removing a node (and all its in- and out-edges). con expresses adding an edge from n to m while disc expresses removing such an edge. Any graph $G = (N, E)$ with a total ordering on its nodes N inducing a total lexicographic order on its edges $E \subseteq N \times N$ can be canonically expressed using $(\!|\cdot|\!) : N \times E \to G$ defined below:

$$\frac{}{(\!|\emptyset, \emptyset|\!) = \texttt{empty}} \qquad \frac{(\!|N, E|\!) = G \qquad (n, m) = \max E}{(\!|N, E \uplus \{(n, m)\}|\!) = (\texttt{con n m G})} \qquad \frac{(\!|N, \emptyset|\!) = G \qquad n = \max N}{(\!|N \uplus \{n\}, \emptyset|\!) = (\texttt{add n G})}$$

IR expressions can be evaluated into graphs using $[\![\cdot]\!] : G \rightharpoonup V \times E$ defined as follows:

$$\frac{}{[\![\texttt{empty}]\!] = (\emptyset, \emptyset)} \qquad \frac{[\![G]\!] = (N, E)}{[\![(\texttt{add}\ n\ G)]\!] = (N \cup \{n\}, E)}$$

$$\frac{[\![G]\!] = (N, E) \qquad n \in N}{[\![(\texttt{rem}\ n\ G)]\!] = (N \setminus \{n\}, E \setminus (\{(n, m)\ :\ m \in N\} \cup \{(m, n)\ :\ m \in N\}))}$$

$$\frac{[\![G]\!] = (N, E) \qquad n, m \in N}{[\![(\texttt{con}\ n\ m\ G)]\!] = (N, E \cup \{(n, m)\})} \qquad \frac{[\![G]\!] = (N, E) \qquad n, m \in N}{[\![(\texttt{disc}\ n\ m\ G)]\!] = (N, E \setminus \{(n, m)\})}$$

The only invariant that holds on a graph $[\![G]\!] = (N, E)$ constructed with the IR is that $E \subseteq N \times N$. A simple inductive argument on $[\![\cdot]\!]$ shows that, whenever it is defined, the invariant holds. Moreover, it is again easy to show by induction that $[\![\cdot]\!]$ is a left-inverse of $(\!|\cdot|\!)$. The latter is not a left-inverse of the former because $[\![\cdot]\!]$ is not injective. For example, $[\![(\texttt{add}\ a\ (\texttt{add}\ a\ \texttt{empty}))]\!] = [\![(\texttt{add}\ a\ \texttt{empty})]\!]$.

Since $[\![\cdot]\!]$ is a partial-function, we define *well-formed* expressions as those expressions g for which $[\![g]\!]$ is well-defined.

4.3.3 Transpiler. To deploy an IaC program written in the IR, it must be transpiled into the language consumed by the IaC engine. The transpiler starts by interpreting the IR according to the semantics in Section 4.3.2, producing a graph, and checking that it is a DAG. Then every node in this graph is transpiled into a resource definition statement and every edge between two nodes is transpiled into a dependency statement. Crucially, we make use of custom providers that are free of side-effects.

4.4 Metamorphic Transformation via E-Graphs

At the heart of EMIAC is an e-graph that plays four important roles. (1) It internalizes the rewrite rules with the saturation engine, (2) it acts as an oracle that efficiently checks if two programs compute the same resource graph, (3) it efficiently represents equivalence classes of IR programs, and, (4) it allows efficient extraction of equivalent IR programs. In the next sections we describe in detail these aspects and finally we formally define the metamorphic relation and transformation.

4.4.1 Rewrite Rules. To define the metamorphic transformation, we design a system which describes equalities between well-formed expressions when they evaluate to the same graph under $[\![\cdot]\!]$. For every equality $a = b$ we can produce two rewrite rules $a \to b$ and $b \to a$. Rewrite rules are generally quantified, e.g., the rewrite rule $\forall r, g.\ (\texttt{add}\ r\ g) \to g$ eliminates every occurrence of add. To increase expressivity, rules can be conditional, e.g., the rule $\forall r_1, r_2, g.\ r_1 \neq r_2 \Rightarrow (\texttt{rem}\ r_1\ (\texttt{add}\ r_2\ g)) \to (\texttt{add}\ r_2\ (\texttt{rem}\ r_1\ g))$ allows swapping an add and a remove operation only when the resources being added and removed are not equal. In the following, conditions are restricted to being conjunctions of resource disequalities in order to keep the system purely-syntactic, i.e. it does not rely on the interpretation of the graph expression.

In the following we implicitly quantify over resources r, r_1, r_2, r_3, r_4 and a graph g.

Idempotent. Three rules state when operations are idempotent:

$$(\texttt{add}\ r\ (\texttt{add}\ r\ g)) \equiv (\texttt{add}\ r\ g)$$

$$(\texttt{con}\ r_1\ r_2\ (\texttt{con}\ r_1\ r_2\ g)) \equiv (\texttt{con}\ r_1\ r_2\ g) \qquad (\texttt{disc}\ r_1\ r_2\ (\texttt{disc}\ r_1\ r_2\ g)) \equiv (\texttt{disc}\ r_1\ r_2\ g)$$

Inverse. Two rules state when some operations are inverses of others:

$$(\texttt{rem}\ r\ (\texttt{add}\ r\ \texttt{empty})) \equiv \texttt{empty}$$

$$(\texttt{disc}\ r_1\ r_2\ (\texttt{con}\ r_1\ r_2\ (\texttt{add}\ r_1\ (\texttt{add}\ r_2\ \texttt{empty})))) \equiv (\texttt{add}\ r_1\ (\texttt{add}\ r_2\ \texttt{empty}))$$

Commutative. Four rules state when an operation is commutative:

$$(\texttt{add}\ r_1\ (\texttt{add}\ r_2\ g)) \equiv (\texttt{add}\ r_2\ (\texttt{add}\ r_1\ g)) \qquad r_1 \neq r_2 \Rightarrow (\texttt{rem}\ r_1\ (\texttt{rem}\ r_2\ g)) \equiv (\texttt{rem}\ r_2\ (\texttt{rem}\ r_1\ g))$$

$$(\texttt{con}\ r_1\ r_2\ (\texttt{con}\ r_3\ r_4\ g)) \equiv (\texttt{con}\ r_3\ r_4\ (\texttt{con}\ r_1\ r_2\ g))$$

$$(\texttt{disc}\ r_1\ r_2\ (\texttt{disc}\ r_3\ r_4\ g)) \equiv (\texttt{disc}\ r_3\ r_4\ (\texttt{disc}\ r_1\ r_2\ g))$$

Commute. These six rules state when two operations commute:

$$r_1 \neq r_2 \Rightarrow (\texttt{add}\ r_1\ (\texttt{rem}\ r_2\ g)) \equiv (\texttt{rem}\ r_2\ (\texttt{add}\ r_1\ g))$$

$$r_1 \neq r_2 \wedge r_1 \neq r_3 \Rightarrow (\texttt{add}\ r_1\ (\texttt{con}\ r_2\ r_3\ g)) \equiv (\texttt{con}\ r_2\ r_3\ (\texttt{add}\ r_1\ g))$$

$$r_1 \neq r_2 \wedge r_1 \neq r_3 \Rightarrow (\texttt{rem}\ r_1\ (\texttt{con}\ r_2\ r_3\ g)) \equiv (\texttt{con}\ r_2\ r_3\ (\texttt{rem}\ r_1\ g))$$

$$r_1 \neq r_2 \wedge r_1 \neq r_3 \Rightarrow (\texttt{add}\ r_1\ (\texttt{disc}\ r_2\ r_3\ g)) \equiv (\texttt{disc}\ r_2\ r_3\ (\texttt{add}\ r_1\ g))$$

$$r_1 \neq r_2 \wedge r_1 \neq r_3 \Rightarrow (\texttt{rem}\ r_1\ (\texttt{disc}\ r_2\ r_3\ g)) \equiv (\texttt{disc}\ r_2\ r_3\ (\texttt{rem}\ r_1\ g))$$

$$r_1 \neq r_3 \wedge r_2 \neq r_4 \Rightarrow (\texttt{con}\ r_1\ r_2\ (\texttt{disc}\ r_3\ r_4\ g)) \equiv (\texttt{disc}\ r_3\ r_4\ (\texttt{con}\ r_1\ r_2\ g))$$

Right-Absorption. Six rules state when a nested operation can be omitted:

$$(\texttt{rem}\ r_1\ (\texttt{con}\ r_1\ r_2\ g)) \equiv (\texttt{rem}\ r_1\ g) \qquad (\texttt{rem}\ r_2\ (\texttt{con}\ r_1\ r_2\ g)) \equiv (\texttt{rem}\ r_2\ g)$$

$$(\texttt{rem}\ r_1\ (\texttt{disc}\ r_1\ r_2\ g)) \equiv (\texttt{rem}\ r_1\ g) \qquad (\texttt{rem}\ r_2\ (\texttt{disc}\ r_1\ r_2\ g)) \equiv (\texttt{rem}\ r_2\ g)$$

$$(\texttt{con}\ r_1\ r_2\ (\texttt{disc}\ r_1\ r_2\ g)) \equiv (\texttt{con}\ r_1\ r_2\ g) \qquad (\texttt{disc}\ r_1\ r_2\ (\texttt{con}\ r_1\ r_2\ g)) \equiv (\texttt{disc}\ r_1\ r_2\ g)$$

Left-Absorption. Four rules state when a parent operation can be omitted:

$$(\texttt{add}\ r_1\ (\texttt{con}\ r_1\ r_2\ g)) \equiv (\texttt{con}\ r_1\ r_2\ g) \qquad (\texttt{add}\ r_2\ (\texttt{con}\ r_1\ r_2\ g)) \equiv (\texttt{con}\ r_1\ r_2\ g)$$

$$(\texttt{add}\ r_1\ (\texttt{disc}\ r_1\ r_2\ g)) \equiv (\texttt{disc}\ r_1\ r_2\ g) \qquad (\texttt{add}\ r_2\ (\texttt{disc}\ r_1\ r_2\ g)) \equiv (\texttt{disc}\ r_1\ r_2\ g)$$

The system is designed to preserve well-formedness under the assumption that they are applied to well-formed expressions. For example, the general idempotent rule for `rem` is missing since $(\texttt{rem}\ r\ (\texttt{rem}\ r\ g))$ is not well-formed, i.e. $[\![(\texttt{rem}\ r\ (\texttt{rem}\ r\ g))]\!]$ is undefined.

Moreover, the system is sound and complete: the application of an equality does not modify the resource graph an expression computes, and given two expressions that compute the same resource graph we can always apply some equalities to transform one into the other. Formally:

Lemma 4.1 (Soundness). *Given well-formed expressions g_1 and g_2, if $g_1 \equiv g_2$ then $[\![g_1]\!] = [\![g_2]\!]$.*

Proof Sketch. By structural induction on $\equiv$. The mechanized proof is in [68]. □

Lemma 4.2 (Completeness). *Given well-formed expressions g_1 and g_{n+1} such that $[\![g_1]\!] = [\![g_{n+1}]\!]$ then there exists $g_2, \cdots, g_n$ such that $g_i \equiv g_{i+1}$ for every $1 \leq i \leq n+1$.*

Proof Sketch. Every program can be rewritten into a canonical form: all removes/disconnects can be pushed next to matching adds/connects with commutative rules, all these pairs can be pushed towards the `empty` and eliminated, all duplicates are removed with idempotent rules, and commutative rules are applied to sort the result into the canonical form. By transitivity of $\equiv$ and Lemma 4.1 the canonical acts as a bridge between g_1 and g_{n+1}. The mechanized proof is in [68]. □

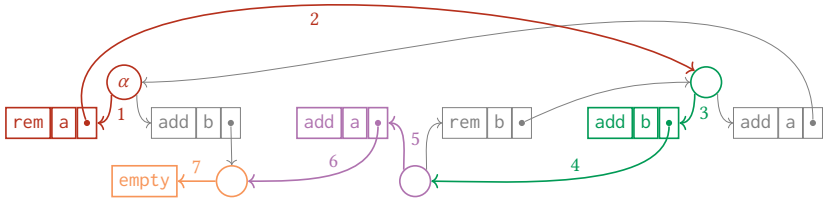


Fig. 5. Walking the e-graph along the path 1–7 starting from the e-class α of resource graphs containing only the resource b produces the program: `(rem a (add b (add a empty)))`

For testing, soundness is crucial: without it the e-graph may generate two expressions that compute different resource graphs. Completeness improves diversity: the more equivalences EMIAC discovers, the more diverse the set of follow up test expressions is.

4.4.2 Rewrite Engine, Saturation, and E-Graph Caching. To apply the metamorphic relation we use the e-graph saturation engine. The process starts with *e-matching* [15] which consists of searching the e-graph for terms that match one side of a rule to find assignments for quantified variables. Substituting these assignments into the other side produces a new term that is added and equated in the e-graph. *Saturation* involves repeatedly e-matching and substituting over all the rules until no new equalities are discovered.

In general, saturation does not terminate. This occurs when new e-classes are discovered at every iteration. But in EMIAC the number of resources is bounded and hence the number of possible resource graphs is too. By completeness, two distinct e-classes cannot contain expressions that compute the same resource graph. This bounds the number of e-classes in the e-graph, and therefore also the number of e-nodes, to be the number of graphs. Nevertheless this upper bound is the number of graphs, 2^{n^2} for n nodes, which is too large for all practical purposes. Therefore, even though saturation eventually converges in EMIAC, it practically never does – and by a similar argument neither would any other complete set of rules. Hence, as is common, saturation is limited by a timeout or by a number of e-classes.

When the set of resources is fixed a priori, each test run wastefully re-generates the same e-graph. Thus, saturating the e-graph can be done once in advance and cached. Our testing framework is parameterized by an upper bound on the number of resources. The framework caches the corresponding e-graph and reuses it for new test inputs. As a result, the most expensive e-graph operations are avoided during generation.

4.4.3 Extraction from the E-Graph. To generate tests we must process the saturated e-graph. We start by describing a naive e-graph traversal algorithm that extracts a random IR expression and discuss its weakness. Given a resource graph, we extract an IR expression by retrieving the e-class containing its canonical expression. Once the e-class is found, we select a random e-node in the e-class and recurse on the e-class argument, or terminate if the random e-node is `empty`. Such a walk is illustrated in Figure 5.

Alas, this walk generates expressions that are either too large or highly repetitive since the chance of terminating, i.e. of hitting the `empty` e-node, is small. On the other extreme, instead of randomly walking the e-graph, we can always follow the shortest path towards `empty`. While this generates short expressions, it may not generate diverse ones after multiple executions, i.e. it may not divert from the shortest path towards `empty`. EMIAC adopts a middle-ground presented in Algorithm 1 which constrains the walk by identifying the nodes that take a long path towards `empty` and removing them from the e-graph. It does so by treating the e-graph as a traditional

graph with all edges weighted by 1, and adapting the classical Dijkstra shortest-path algorithm to probabilistically mark a node as visited after having visited it. The probabilistic check at line 14 marks an e-class as visited with probability ε, therefore the shortest-path walk is recovered by setting ε to 0, and the naive random walk is recovered by setting ε close to 1. In the following we describe the algorithm in more detail.

To identify all relevant e-nodes, we start at the e-class of `empty`, mapping it to the singleton set containing only the `empty` e-node. This e-class is then placed in the queue of e-classes awaiting processing (line 2—line 5). As long as the queue is non-empty, we repeatedly select an e-class to process. For the chosen e-class, we collect all e-nodes that reference it and belong to an as-yet unprocessed e-class (line 11). These e-nodes are added into their parent's path set (line 12), and their corresponding e-classes are enqueued for future processing (line 13). Importantly, we do not immediately mark these e-classes as visited, since doing so would reduce diversity. Instead, we do a random check to mark them as visited (line 14).

Algorithm 1: E-Graph Filtering

```
   input : G – The e-graph to extract from.
           ε – The probability of not following the
           shortest path.
   output: P – An e-graph that can be walked to
           produce expressions.
1  P ← Map.empty()
2  empty_eclass ← G.find(empty)
3  P[empty_eclass] = { empty }
4  visited = { empty_eclass }
5  queue = [ empty_eclass ]
6  while |queue| > 0 do
7      eclass ← queue.pop()
8      for enode ∈ G.ref_to(eclass) do
9          parent ← G.find(enode)
10         if parent ∈ visited then
11             continue
12         P[parent] ← P[parent] ∪ {enode}
13         queue.push(parent)
14         if rand01() < 1 − ε then
15             visited ← visited ∪ { parent }
```

Finally, walking the e-graph generated by Algorithm 1 randomly as in Figure 5 is sufficient to generate diverse yet small graph expressions. We note that since Algorithm 1 only depends on the (cached) saturated e-graph and a probability parameter, it can be done once preemptively and cached for future test generation.

4.4.4 Batching. To test the stateful behavior of provision-based `IaC` engines we must deploy multiple programs in succession. We do so by deploying the generated test program in batches. For example, consider the program `(con a b (add c (add a (add b empty))))`. We may split it into the following three batches: first `(add b empty)`, second `(add a (add b empty))`, and third the full program.

Care must be taken when splitting the program. To be deployable, an `IaC` program's resource graph must be a DAG. Even though the full program produces a DAG, a batch could still be cyclic. For example, the program `(disc a b (con a b (con b a empty)))` computes a DAG, yet if it is split into two batches at the `disc`, then the first batch contains a cycle and cannot be deployed.

To make sure we select acyclic batches we avoid traversing e-classes whose corresponding resource graph contains a cycle. In fact, we do so by preventing cyclic graphs from ever being represented in the e-graph by modifying all the rules from Section 4.4.1 that introduce a `(con a b g)` by adding the constraint that `a < b` according to some total-order on the resource nodes.

Therefore, as we walk P to generate a program, it is guaranteed that splitting the program at any point generates two DAGs.

4.5 Test Execution and Oracles

To improve the performance of EMIAC, we designed it to generate multiple tests in parallel. Since walking the e-graph is a read-only operation and therefore easily parallelizable, EMIAC can generate one test per thread. Moreover, each program can be batched in parallel. If the number of test inputs is n and each produces b batches, then transpiling the $n \cdot b$ programs can also be done in parallel

and interleaved with batching. Finally, once all $n \cdot b$ programs are transpiled and written to disk, we spawn n threads, each deploying its b batches sequentially, thereby producing n resource graphs.

Each of the resource graphs generated by the test execution is compared for equality against the resource graph from the source test, i.e., the one from the blue box in Figure 4. This source resource graph is produced during test input generation before any metamorphic transformation is applied. It acts as the oracle for the n resource graphs produced by the n batches of the metamorphic transformation. Crucially, testing the equivalence of resource graphs is *not* graph homomorphism. For example, a deployment that creates a single AWS S3 bucket is not the same as a deployment that creates a single AWS EC2 instance. The resource graph equality test between two graphs is done by checking the equality of the two node sets and the two edge sets. Finally, a bug is found when the equality test fails and the batch sequence is exhibited as the bug witness.

4.6 Formalism

In this section we present EMIAC formally in line with the definitions of Section 2.1.

Let $\mathcal{G}$ be the set of all resource graphs and $0_{\mathcal{G}} \in \mathcal{G}$ be the empty resource graph. Let $\mathcal{P}$ be the set of all programs consumed by a stateful `IaC` deployment engine modeled as $D_1 : \mathcal{G} \times \mathcal{P} \to \mathcal{G}$, a function that deploys a program on top of an existing resource graph. We assume the deployment engine is *correct*, i.e. that no matter the state it produces the same result. Finally, we assume we are given a sound transpiler $T : G \to \mathcal{P}$, one that satisfies: $D_1(T(g), 0_{\mathcal{G}}) = [\![g]\!]$. The system under test is defined as follows:

Definition 4.3. The system under test is the function $D^* : G^* \to \mathcal{G}$ that deploys a sequence of IR programs such that: $D^*(\emptyset) = 0_{\mathcal{G}}$ and $D^*(t_1, \cdots, t_n, t_{n+1}) = D_1(T(t_{n+1}), D^*(t_1, \cdots, t_n))$

To define the metamorphic relation $\mathcal{MR}$ we must define the input relation $\mathcal{I}_{\mathcal{R}}$ and output relation $\mathcal{O}_{\mathcal{R}}$. Two sequences of input IR programs are in $\mathcal{I}_{\mathcal{R}}$ if they aim to deploy the same resource graph in the final program regardless of the intermediary deployments, i.e., if $\mathcal{I}_{\mathcal{R}}(\{g_1, \cdots, g_n\}, \{g'_1, \cdots, g'_m\})$ whenever $[\![g_n]\!] = [\![g'_m]\!]$ and $\mathcal{O}_{\mathcal{R}}$ is graph equality. To define the metamorphic transformation $\mathcal{T}$ we require a definition of e-graphs which we adapt from [76]:

Definition 4.4. An e-graph $\mathcal{E}$ is a triple (V_c, V_n, E) such that: (1) V_c is a set of e-classes, (2) V_n is a set of e-nodes, and (3) $E : V_n \to V_c$ is a total function representing e-class membership edges.
E-classes are drawn from some countably infinite set, i.e., $V_c \subseteq C$ and e-nodes are tuples whose first element is a function from an alphabet Σ with an associated arity and the remaining elements are e-classes, i.e., $V_n \subseteq \bigcup_{f \in \Sigma} \{f\} \times V_c^{arity(f)}$.

To simplify, let $R = \{r_1, \ldots, r_n\}$ be the set of resources and choose $\Sigma = \{\texttt{empty}\} \cup \{\texttt{add}, \texttt{rem}\} \times R \cup \{\texttt{con}, \texttt{disc}\} \times R^2$. All symbols in Σ have arity 1 except `empty`, which has arity 0. By abuse of notation, every program in the IR, except for `empty`, can be rewritten as $f(g)$ where $f \in \Sigma$ and g is another program, for example $(\texttt{add}\ \texttt{r}_1\ \texttt{empty})$ can be written as $(\texttt{add}, \texttt{r}_1)(\texttt{empty})$ with $(\texttt{add}, \texttt{r}_1) \in \Sigma$. In the following we first define how to find the e-class of an expression with $[\cdot]$, which we use to define an equivalence relation $\equiv_{\mathcal{E}}$ over G, and then define a saturated e-graph.

Definition 4.5. If $g = \texttt{empty}$ then $[g] = E(\texttt{empty})$, otherwise let $g = f(g')$ then $[g] = E(f([g']))$.

Definition 4.6. Given two IR programs g_1 and g_2, then $g_1 \equiv_{\mathcal{E}} g_2$ iff $[g_1] = [g_2]$.

Definition 4.7. An e-graph $\mathcal{E}$ is saturated w.r.t. an equational system E whenever $(g_1, g_2) \in E^*$ implies $g_1 \equiv_{\mathcal{E}} g_2$ where E^* is the reflexive, symmetric, transitive, and congruent closure of E.

We saturate our e-graph w.r.t. the equational system in Section 4.4.1. Thanks to Lemma 4.1, when $[g_1] = [g_2]$, i.e. $g_1 \equiv_{\mathcal{E}} g_2$, then $[\![g_1]\!] = [\![g_2]\!]$. Now we define the metamorphic transformation $\mathcal{T}$ that

is given a sequence of IR programs and produces a new sequence of programs. The transformation is defined in two parts: as a walk (Section 4.4.3) and a batcher (Section 4.4.4). First, we define $Walk : V_c \to G$ which maps an e-class to a (possibly infinite) set of all IR programs in it:

Definition 4.8. Let g be an IR program and c be an e-class. If $g = \texttt{empty}$ then $g \in Walk(c)$ iff $E(\texttt{empty}) = c$. Otherwise let $g = f(g')$ then $g \in Walk(c)$ iff $g' \in Walk([g'])$ and $E(f([g'])) = c$.

The batcher splits an IR program into parts with the invariant that any earlier program is a subterm of a later program. We define $Batch : G \to \mathcal{P}(G^*)$ that generates all possible batchings:

Definition 4.9. $Batch(\texttt{empty}) = \{\texttt{empty}\}$ and $Batch(f(g)) = (Batch(g) \cup Batch(g) \setminus \{g\}) \times \{f(g)\}$.

Finally, we can define the metamorphic transformation $\mathcal{T}$ as follows:

Definition 4.10. $g_1, \cdots, g_n \xrightarrow{\mathcal{T}} g'_1, \cdots, g'_m$ iff $\exists g'' \in Walk([g_n]), (g'_1, \cdots, g'_m) \in Batch(g'')$.

We state the correctness of our framework in the following two theorems.

Theorem 4.11 (Metamorphic Relation). *D^*, $\mathcal{I}_R$, and O_R satisfy Definition 2.1*

Proof Sketch. Let $b_0 = \{g_1, \ldots, g_n\}$ and $b_1 = \{g'_1, \ldots, g'_m\}$ be given. The hypothesis $H : \mathcal{I}_R(b_0, b_1)$ is by definition $[\![g_n]\!] = [\![g'_m]\!]$. By definition of D^*: $D^*(b_0) = D_1(g_n, D^*(g_1, \ldots, g_{n-1}))$. By the correctness of D_1: $D^*(b_0) = D_1(T(g_n), 0_{\mathcal{G}})$. By the soundness of T: $D^*(b_0) = [\![g_n]\!]$. By the same argument $D^*(b_1) = [\![g'_m]\!]$. By definition of O_R and H it follows that $O_R(D^*(b_0), D^*(b_1))$. □

Theorem 4.12 (Metamorphic Transformation). *$\mathcal{I}_R$ and $\mathcal{T}$ satisfy Definition 2.2*

Proof Sketch. Assume $g_1, \cdots, g_n \xrightarrow{\mathcal{T}} g'_1, \cdots, g'_m$ then there exists $g'' \in Walk([g_n])$ and $g'_1, \ldots, g'_m \in Batch(g'')$. By the definition of $Batch$: $g'_m = g''$ so $g'_m \in Walk([g_n])$. By induction on $Walk$ it's easy to show that for every $g \in Walk(c)$ then $[g] = c$. Thus $[g_n] = [g'_m]$. By Lemma 4.1 and Definition 4.7 then $[\![g_n]\!] = [\![g'_m]\!]$ or $\mathcal{I}_R(\{g_1, \cdots, g_n\}, \{g'_1, \cdots, g'_m\})$. □

5 IMPLEMENTATION

We implemented EMIaC primarily in Rust. The implementation comprises about 2,900 lines of code including tests. A key component is a custom e-graph with a custom saturation mechanism. Our implementation supports deferred rebuilding as introduced by egg [74]. However, the rewrite rules handled by egg cannot support the rules from Section 4.4.1: while egg's e-matcher supports conditional e-matching, it only supports conjunction of equalities and not conjunction of disequalities. While it is technically possible to convert disequalities into equalities by embedding them [76], this would inflate the size of the e-graph and incur a runtime cost.

Supporting components are lightweight. The *random graph generator* and *graph comparator* (see Figure 4) are written in Python and each requires fewer than 100 lines of code. Execution of the generated IaC programs is orchestrated via the native IaC tool CLIs (e.g., terraform apply).

6 EXPERIMENTAL EVALUATION

In this section, we address the following research questions:

RQ_1 (Characterization of Resource Graphs): *What is the structure of the resource graphs derived from real-world* IaC *programs?*

RQ_2 (Generalizability): *Does EMIaC generalize to different* IaC *engines?*

RQ_3 (Coverage Comparison): *How does the code coverage achieved by EMIaC compare to that of the official test suites of Terraform and Pulumi?*

RQ_4 (Mutation Analysis): *How effective is EMIaC in detecting bugs in* IaC *engines, as measured by mutation testing?*

RQ_5 (Test Quality): *How does the e-graph influence the quality of follow-up tests compared to a baseline rewriting-based approach?*

6.1 Characterization of Resource Graphs (RQ_1)

Understanding the size and shape of `IaC` resource graphs is essential for evaluating `IaC` engines, as these graphs are the inputs that engines process to execute infrastructure changes. We conduct the first large-scale empirical study of resource graphs derived from real-world `IaC` programs. Our analysis establishes graph properties such as size, density, connectivity, and depth. These properties help design test cases covering both common and challenging edge cases for `IaC` engines.

Procedure. To answer RQ_1, we analyze resource graphs extracted from a curated subset of the TerraDS dataset [5], which originally comprises 279,344 `IaC` programs across 62,406 GitHub repositories. We applied the following filtering steps to TerraDS: (1) we removed two extreme outlier repositories—one with more than 14,000 resources in a single program and another with over 26,000 edges—that disrupt graph computation;[2][3] (2) we excluded 2,329 archived repositories, as these projects are unlikely to represent actively maintained `IaC` programs; and (3) we excluded 52,792 repositories with no commits after July 1, 2024, to focus on recently updated projects. After these steps, the dataset contains 7,283 repositories with 43,593 `IaC` programs.

A practical challenge with real-world `IaC` programs is to extract the resource graph. In practice, the graph is obtained by running CLI commands such as Terraform *plan* or *apply*. Both approaches require a fully configured environment with concrete input values, provider credentials, state backends or workspaces, and access to external data sources. Automating the environment creation is unreliable, and running *plan* or *apply* is slow and may incur cloud costs. To avoid these overheads, we relied on static analysis. Specifically, we employed Checkov [9], an open-source static analyzer for `IaC` programs. Checkov statically parses Terraform programs and internally constructs the resource graph without deploying or interacting with cloud providers. We forked Checkov's repository and extended it to export the resource graph in a machine-readable format. We executed Checkov on each of the 43,593 `IaC` programs and produced the corresponding resource graphs.

Metrics. We measure the *number of nodes* (resources) and the *number of edges* (dependencies) of resource graphs. We assess connectivity using the *maximum in-degree*, i.e., the largest number of incoming dependencies of any resource. Similarly, the *maximum out-degree* reflects how many other resources depend on a given one. Dependency depth is quantified by the *longest path length*, indicating the longest chain of resources that must be created sequentially. Overall connectivity is described by the *graph density*, defined for a directed graph with N nodes and E edges as $D = |E|/(2\binom{|N|}{2})$, where the denominator is the maximum number of possible directed edges without self-loops, yielding a maximal density of 0.5. Finally, we compute the *% of isolated nodes*, i.e., resources without any dependencies that can typically be created in parallel. We aggregate these metrics across the dataset and report descriptive statistics: minimum, maximum, mean, median, standard deviation, and the 95th percentile.

Results. Table 1 summarizes the computed metrics for the 43,593 `IaC` programs across 7,283 repositories. Our analysis shows that real-world `IaC` resource graphs are generally small and sparse. Programs define an average of 11 nodes and 9 edges, with medians of 5 and 2, respectively. Most nodes exhibit low connectivity: the median out-degree is 1, and 25% of nodes are isolated. The average maximum in-degree is 2.8, but some nodes reach up to 339 incoming edges. Resource graphs are shallow, with a median longest path length of 1, and sparse, with a mean density of 0.1.

[2] https://github.com/zedwerks
[3] https://github.com/Azure

Table 1. Summary statistics of IaC programs in TerraDS [5].

Metric	min	max	mean	median	std dev	95%ile
Number of nodes	1	801	10.9	5	21.9	40
Number of edges	0	1,172	8.9	2	23.7	40
Max in-degree	0	339	2.8	1	5.4	11
Max out-degree	0	51	1.6	1	1.9	5
Longest path length	0	23	1.5	1	1.6	5
Graph density	0	0.5	0.1	0.05	0.2	0.5
% isolated nodes	0%	100%	40%	25%	40%	100%

RQ_1 (Resource Graphs): Overall, real-world IaC resource graphs are small, sparse, and shallow with a median of 5 nodes and 2 edges, an average of 11 nodes and 9 edges.

6.2 Generalizability (RQ_2)

We aim to investigate whether EMIAC can be applied across different IaC engines without requiring engine-specific adaptations. To be practical, EMIAC should be capable of generating and executing test cases for major provisioning-based IaC engines, such as Terraform, Pulumi, and OpenTofu [22].

Procedure. Based on the insights from RQ_1, we randomly generated a resource graph with 5 nodes and 2 edges, matching the median values observed in real-world IaC programs. We then split the IR into two batches to also exercise the code responsible for diffing the state file against the updated IaC program. Figure 6 depicts the two resource graphs used in this experiment.

The IR is transpiled into HCL for Terraform and OpenTofu, and into Python for Pulumi.[4] We excluded AWS CloudFormation because it is closed-source and primarily targets AWS resources. The resulting programs are then executed using the three IaC engines across equivalent stages of a typical IaC workflow: *init*, *plan*, *apply-1*, *apply-2*, *graph*, and *destroy*. The *init* phase sets up the working directory and downloads the required provider plugins. The *plan* phase analyzes the program and computes the changes needed to reach the target state, without modifying any resources. The *apply-1* phase executes the first batch by creating all resources in the initial resource graph (left-hand side of Figure 6). In the *apply-2* phase, the second batch is executed (right-hand side of Figure 6). Unlike *apply-1*, this step does not start from an empty state but must compute the diff against the resources deployed in the first run, exercising different execution paths w.r.t. the first batch. The *graph* command exports the resource graph of the current state. Finally, the *destroy* command removes all previously created resources. The terminology varies across tools (e.g., Pulumi's `preview` corresponds to Terraform's `plan`), but the steps are functionally the same.

Metrics. We measure the absolute *statement coverage*; the number of source statements executed for each core command in each IaC engine. Our goal is to focus on understanding the overall coverage trends across the core phases of an IaC workflow, rather than directly comparing the coverage of different IaC engines. Measuring statement coverage at each stage of a typical workflow helps understand which and how much of the engine's code is exercised by common operations. All three IaC engines are written in Go, which provides native support for statement coverage [25, 26]. This ensures a uniform measurement procedure: the Go toolchain instruments the source code and groups source statements into minimal code blocks, counts the number of statements per block, and records how often each block is executed. We use this mechanism to quantify coverage per command consistently across Pulumi, Terraform, and OpenTofu (all the source code is instrumented).

[4]Throughout our evaluation, we use Terraform v1.12.1, Pulumi v3.171.0, and OpenTofu v1.10.5.

Results. Figure 7 reports statement coverage for Pulumi, Terraform, and OpenTofu across the main phases of an IaC workflow. Across all IaC engines, coverage increases sharply from *plan* to *apply*, as the plan computation is a strict subset of what is executed during apply. The second apply step reaches the highest coverage, since it must reconcile the updated program with the state created in the first run. As expected, Terraform and OpenTofu exhibit similar statement coverage, as OpenTofu was forked from Terraform in 2023. Pulumi shows a lower absolute coverage, but follows the same overall trend.

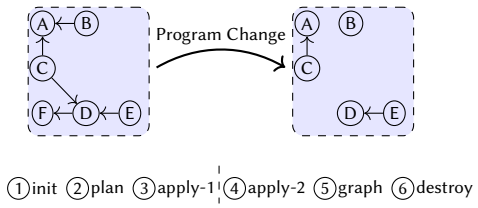


Fig. 6. Resource graphs of the first batch (left) and the second batch (right).

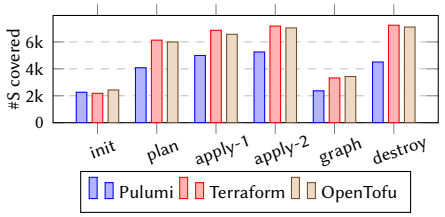


Fig. 7. Statement coverage of IaC programs generated from the same graph.

RQ_2 (Generalizability): Overall, EMIAC generalizes across Terraform, OpenTofu, and Pulumi, as it is applicable without engine-specific modifications. Statement coverage results show that the same generated test case triggers a similar coverage trend across a typical IaC workflow in the considered IaC engines, with peaks during the second apply (state diff).

6.3 Coverage Comparison (RQ_3)

Test suites maintained by tool developers serve as a natural baseline for coverage comparison [6, 23]. These tests are usually carefully crafted to validate the correctness of the codebase. Moreover, they are designed at the function level to exercise individual units. By comparing EMIAC's coverage against developer-written test suites, we aim at identifying complementary strengths.

Procedure. To answer RQ_3, we considered Terraform and Pulumi. We omitted OpenTofu, as Section 6.2 already shows that its coverage closely matches that of Terraform. For Terraform, the repository contains a root Go module in terraform, with the majority of source code located under terraform/ internal. Terraform comprises 168 Go packages, 165 of which under terraform/ internal. Pulumi, in comparison, consists of 131 packages. For both IaC engines, we executed the official test suites with coverage enabled. For Terraform, these are primarily the packages under terraform/ internal /terraform and terraform/ internal / states, which implement the core engine functionality. For Pulumi, the relevant code primarily resides in pulumi/pkg/engine and pulumi/pkg/graph. We then computed the coverage achieved by EMIAC for comparison. To configure EMIAC, we relied on the insights from RQ_1, setting the number of resources to 11 and the maximum number of dependencies to 9, consistent with the average graph characteristics (Section 6.1). We extracted five distinct graphs from the e-graph, corresponding to five different runs, to deal with the randomness of the e-graph walk. Each of these graphs was then split into four batches that we executed sequentially.

Metrics. We measure *statement coverage* using the same mechanism as in RQ_2. Since our focus is on comparing EMIAC with the developer-written test suite, we normalize statement coverage w.r.t. the total number of statements in the source code. For both Terraform and Pulumi, we report the (i) overall coverage of the test suite, (ii) overall coverage of EMIAC, averaged across five runs, (iii) a breakdown across packages, and (iv) the number of statements uniquely covered by EMIAC. This

Table 2. Statement coverage in /terraform/ internal comparing Terraform test suite and EMIAC.

		Coverage (%)		
	#S	Test Suite	EMIAC	ΔS
Overall	83,551	61.14%	10.95%	98
/terraform	8,208	85.12%	31.27%	1
/states	2,170	79.35%	26.22%	0
/plugin6	1,581	62.81%	19.43%	6
/configs	4,602	86.27%	14.69%	1
/command	12,335	74.29%	14.06%	14
/tfplugin6	3,056	10.50%	7.59%	69

Table 3. Statement coverage in /pulumi/pkg comparing Pulumi test suite and EMIAC.

		Coverage (%)		
	#S	Test Suite	EMIAC	ΔS
Overall	49,259	32.68%	14.82%	1,313
/graph	80	57.50%	58.75%	3
/engine	1,614	39.84%	40.39%	179
/resource	7,073	58.83%	36.36%	451
/util	330	34.55%	28.89%	20
/backend	7,322	49.99%	21.55%	333
/cmd	10,768	39.04%	17.58%	327

		Terraform		Pulumi	
Input Size ($\|V\|, \|E\|$)	Seed Source	#S	S Cov. (%)	#S	S Cov. (%)
Average (11, 9)	Generated	9,148	10.95%	7,300	14.82%
	TerraDS	9,142	10.95%	7,297	14.81%
95%ile (40, 40)	Generated	9,196	11.01%	7,403	15.04%
	TerraDS	9,190	11.00%	7,410	15.04%

Table 4. Comparison of coverage using randomly generated seeds vs. real-world graph shapes from TerraDS for average and large input size.

enables us to assess how much of the code exercised by EMIAC overlaps with or extends beyond the coverage achieved by the official test suites.

Results. The statement coverage metrics for Terraform and Pulumi are shown in Table 2 and Table 3, respectively. Due to space restrictions, we only report the most relevant subdirectories that are related to the deployment mechanism of the IaC engine, which is the target of EMIAC; nonetheless, the overall coverage reported in the first row of each table considers the whole codebase. For Terraform, EMIAC achieves notable coverage in deployment-related subsystems: 31.27% in /terraform, and 26.22% in /states. For Pulumi, EMIAC achieves 58.75% coverage in /pkg/graph and 40.39% in engine, slightly exceeding the coverage attained by the existing test suite.

Beyond coverage percentages, EMIAC executes *previously untested statements* (ΔS) that the official suites miss. For Terraform, EMIAC adds 98 statements overall, with most of the unique coverage in /tfplugin6 (69), and additional contributions in /command (14) and /plugin6 (6). For Pulumi, the effect is more pronounced: EMIAC contributes 1,313 extra statements, primarily in /resource (451), /backend (333), and /cmd (327).

In Table 4, we compare seeds that are randomly generated by our random graph generator (see Section 4.3.1) against graph shapes taken directly from real Terraform programs in TerraDS, for two representative input sizes: an average-sized graph (11 resources, 9 dependencies) and a large graph at the 95th percentile of TerraDS (40 resources, 40 dependencies). The results are consistent across Terraform and Pulumi, showing that the size of the input graphs has little impact on the coverage achieved by EMIAC. This shows that EMIAC produces inputs that stress the engine similarly to real-world graphs of the same size, and that EMIAC's coverage is not an artifact of a particular seed selection strategy. Our random graph generator is parameterized to match key structural characteristics of real IaC resource graphs. Nevertheless, it can produce shapes that differ from those observed in TerraDS. For example, it may generate very long dependency chains or nodes with very high in-degree/out-degree, which are unlikely to occur in practice, as resources typically

depend on a limited set of other resources. We view this as complementary: real-derived graphs ground testing in realistic structures, while random graphs expand exploration to stress edge cases.

RQ_3 (Coverage): Overall, EMIAC complements the official test suites of Terraform and Pulumi by exercising additional statements in core components of their deployment mechanisms, and this gain is stable across input sizes. In particular, EMIAC covers 98 extra statements in Terraform and 1,313 in Pulumi.

6.4 Mutation Analysis (RQ_4)

Code coverage is a computationally efficient criterion to assess the adequacy of a test suite. Yet, it may give a false sense of confidence on the correctness of the program under test, especially when the behaviors exercised by the test are not properly asserted [38]. Mutation analysis [17] addresses this problem by systematically injecting syntactic changes into the program under test (i.e., *mutants*) to mimic programmers' mistakes, and by assessing whether the test suite is able to catch them. In testing parlance, a mutant is said to be *killed* if the test suite fails when the mutant is executed. Several mutation analysis tools have been proposed by researchers and practitioners [11, 44, 72], and mutation analysis has also seen adoption in industry [28, 55]. In this research question, we aim to investigate whether EMIAC is effective in catching injected faults that are not killed by developer-written test suites. Such cases indicate that EMIAC complements the existing tests and strengthens its bug detection capability.

Procedure. We considered the major IaC engines, namely Terraform, Pulumi, and OpenTofu, for this RQ. To ensure a focused and efficient analysis, we selected one representative package per engine, specifically the one most relevant to its deployment mechanism. For Terraform, we analyzed the `internal/states` package, for Pulumi the `pkg/engine`, and for OpenTofu we considered the `internal/tofu` package. For each engine and package, we executed the Go mutation analysis tool `go-mutesting` [72], a maintained and widely used framework,[5] with its default configuration to inject mutants. We then processed the mutants by (1) filtering those that did not compile. To make the comparison with the developer-written test suites fair, we (2) retained the mutants that are executed by both the developer-written test suite and EMIAC. Indeed, a necessary condition for mutation killing is coverage, i.e., the line that contains the mutation must be excercised by at least one test case. Additionally, we (3) filtered the mutants killed by the developer-written test suite, and (4) executed EMIAC on the remaining mutants.

Metrics. We count the number of mutants killed by EMIAC that are not by the official test suites.

Results. Regarding Terraform, `go-mutesting` generated 564 mutants across 11 files in the `internal/states` package; 442 compile and are retained. Removing mutants not executed by both the official test suite and EMIAC, as well as those already killed by the suite, leaves 93 mutants. EMIAC killed four additional mutants. Two map to the same underlying defect, i.e., a missing test for a specific edge case. Therefore, we reported three concrete test gaps to the maintainers (IDs 2-4 in Table 5). The issues were acknowledged by the Terraform developers and our fixes were merged into Terraform's main branch[6]. Regarding Pulumi, the mutation analysis tool

Table 5. Previously unknown bugs and test gaps detected by EMIAC in major IaC engines.

ID	IaC Engine	Issue	Status
1	Pulumi	wrong specs	fixed
2	Terraform	missing test	fixed
3	Terraform	missing test	fixed
4	Terraform	weak oracle	fixed
5	Pulumi	missing test	ack.
6–10	OpenTofu	under review	unreported

[5] Last commit 6 months ago at the time of writing and 188 stars on GitHub.
[6] See pull request: https://github.com/hashicorp/terraform/pull/37650.

generated 996 mutants across 15 files in the pkg/engine package; 837 compile and are retained. After the filtering of the code coverage and test suite, 39 mutants remained. EMIAC killed one additional mutant. We reported the gap to the maintainers (ID 5 in Table 5), which has been acknowledged. We also identified and reported a specification error in the documentation (ID 1 in Table 5), which was promptly fixed[7].

Regarding OpenTofu, the mutation analysis tool generated 5,216 mutants across 120 files in the internal/tofu package. After removing non-compiling mutants, we randomly sampled 10 files for detailed analysis and retained the mutants that are executed by the test suite and EMIAC, yielding 273 mutants. The official suite killed 171 of 273. EMIAC killed 5 additional mutants. Due to the time-consuming task of manually inspecting and reporting each case, we are still in the process of reporting them to the maintainers.

False negatives. We estimate false negatives using mutation analysis. A false negative is a mutant that survives after running EMIAC. Since we only keep mutants that (i) compile, (ii) are executed by both EMIAC and the official test suite, and (iii) are not killed by the official suite, each survivor is a limitation of *both* oracles. These survivors therefore point to concrete opportunities to improve future metamorphic or unit-level checks. Table 6 groups false negatives by mutation operator, while Figure 8 groups them by engine component (see Figure 1).

Across engines, most false negatives come from *statement omission* and *boolean-term masking* (Table 6). Such changes can leave the high-level provisioning outcome unchanged for our workloads. For Terraform, 89/442 mutants survive (20.1%). The highest survival rates are for boolean-term masking (20/63, 31.7%) and statement omission (30/112, 26.8%). In contrast, branch-body deletion is rarely missed (7/140, 5.0%). The component breakdown in Figure 8 also reflects our setup. We mutate one representative engine-core package per tool (Terraform: internal/states; Pulumi: pkg/engine; OpenTofu: internal/tofu). It is therefore expected that Terraform's survivors appear in the *State Manager*. Pulumi's survivors are spread across components, suggesting remaining gaps across reconciliation and execution. OpenTofu's survivors appear mainly in the *Graph Builder*. This suggests that many remaining gaps are in code that constructs or updates dependency graphs. These cases are good candidates for future work, for example by adding checks on exported graphs and other intermediate representations. Overall, these results show that future work could, for example, focus on infrastructure drift by injecting controlled drift between runs and checking that reconciliation produces the expected plan and final state.

RQ_4 (Mutation Analysis): Overall, EMIAC kills 24 mutants that the official suites of Terraform, Pulumi, and OpenTofu miss, demonstrating its ability to identify real-world test deficiencies and bugs in the major IaC engines. Three such deficiencies have already been fixed and merged in the main branch of Terraform.

6.5 Test Quality (RQ_5)

RQ_5 investigates EMIAC's e-graph impact on follow-up test quality, and compares it to a baseline that applies random rewrites to generate follow-up tests according to the metamorphic rules.

Procedure. To assess the quality of the tests generated by EMIAC, we considered the following assignments to each variable: resource graphs with 5, 10, 20, 30, 40, 60, 80, 100 and 120 resources, escape probabilities of 1%, 10%, 25%, 40%, and 50%, and timeouts of 30 seconds, 2, 7, 15, and 30 minutes. For each triple, we generated 50 random source test cases, resulting in a total of 11,250 configurations. For each configuration, we used EMIAC to generate 100 equivalent follow-up test

[7]See issue: https://github.com/pulumi/docs/issues/15490.

Mutation operator	Terraform	Pulumi	OpenTofu
statement omission `x++` ⟶ ∅	30/112 (26.8%)	17/256 (6.6%)	25/346 (7.2%)
boolean-term masking `a&&b` ⟶ `true`	20/63 (31.7%)	15/93 (16.1%)	22/139 (15.8%)
loop control change `break` ⟶ `continue`	15/55 (27.3%)	1/65 (1.5%)	3/125 (2.4%)
branch-body deletion `if(c){s}` ⟶ `if(c){}`	7/140 (5.0%)	1/240 (0.4%)	17/404 (4.2%)
numeric literal ±1 `n` ⟶ `n+1`	16/65 (24.6%)	4/155 (2.6%)	16/94 (17.0%)
boundary shift `x<k` ⟶ `x≤k`	1/6 (16.7%)	0/14 (0.0%)	10/21 (47.6%)
arithmetic operator swap `a+b` ⟶ `a-b`	0/1 (0.0%)	0/14 (0.0%)	4/9 (44.4%)
Total	89/442 (20.1%)	38/837 (4.5%)	97/1,138 (8.5%)

Table 6. Surviving mutants (false negatives) by mutation class. Each cell reports *FN/compiled* and the corresponding FN rate.

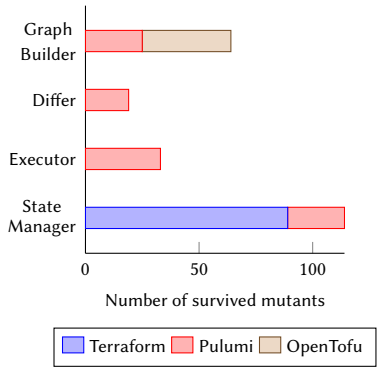


Fig. 8. False negatives per engine component.

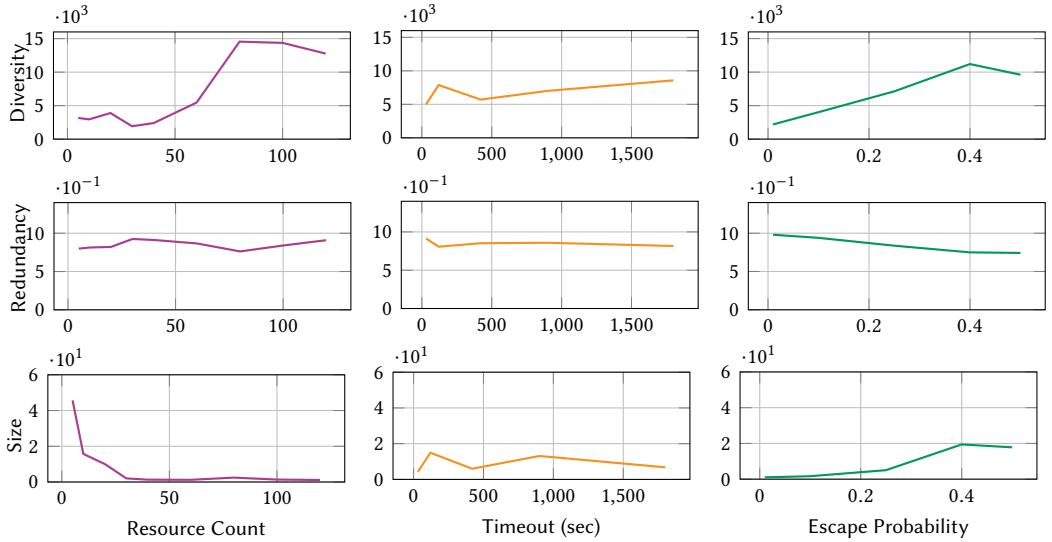


Fig. 9. The diversity, redundancy, and size of the tests by resource count, timeout, and escape probability.

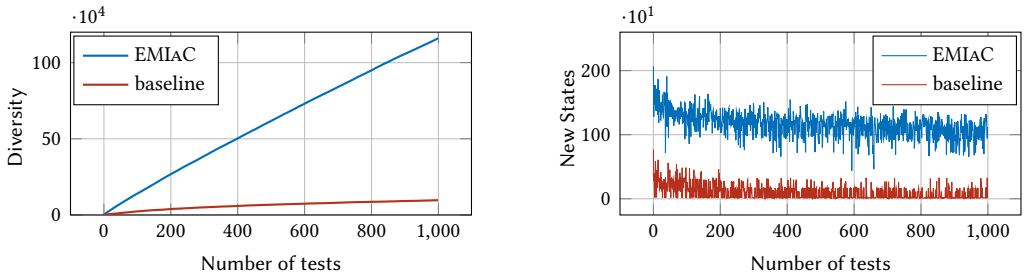


Fig. 10. The diversity and the number of new states discovered with the number of tests generated.

cases, resulting in a total of 1,125,000 tests.

For the baseline, we generated 10 source test cases and then applied EMIAC to generate 1,000 equivalent follow-up tests for each with a resource count of 15, a timeout of 900 seconds, and an

escape probability of 50%. We found that, on average, EMIAC generated 1 test per 3.9 ms. Then, we used the baseline approach that keeps searching and applying rewrite rules for 3.9 ms for each of the 10,000 follow-up tests. This time budget was equivalent to applying 381 rewrite rules, on average, per test. This comparison allows us to measure the difference in diversity and its evolution between follow-up tests generated by EMIAC and those generated by the baseline when both are given the same time budget as set by EMIAC.

Metrics. We use three metrics to judge the quality of follow-up tests. The first is a measure of *test diversity*: we count the number of *distinct intermediary* graphs across all 100 follow-up tests per configuration. We choose this metric to penalize test cases with repeating patterns. The second measure is *test redundancy*, the ratio of follow-up tests that do not discover any new resource graph while executing. Finally, *test size* measures the average size of a generated follow-up test per configuration compared to the canonical representation of the resource graph of the configuration.

Results. In the first row of Figure 9, we plot the average diversity of the 1,250,000 follow-up tests by the resource count first, then the timeout, then the escape probability. In other words, we count the average number of unique graphs visited by each of the 100 test cases of the 11,250 configurations. The diversity increases with the resource count since the space of directed graphs of a given size grows exponentially. Similarly, the diversity increases with the escape probability as the e-graph walker is allowed to choose more e-nodes that may take longer paths to the empty graph, and thus cover more intermediary states. However, the timeout's influence on the diversity seems weak but consistently positive. Therefore, as expected, increasing all three parameters seems to increase the test diversity.

In the second row of Figure 9, we show the ratio of tests that only revisit previously explored graphs. For the same reasons as before, the ratio decreases as the escape probability increase, whereas it remains relatively flat with the timeout and number of resources.

In the third row of Figure 9, we show the relative increase in test size w.r.t. the smallest test generated by the canonical graph representation. Thanks to the filtering performed before the e-graph walk in Algorithm 1, the increase in test size grows with higher escape probabilities as expected, while it remains relatively unchanged w.r.t. the timeout and number of resources.

Based on this analysis, we select an escape probability of 25%; a resource count of 50, more than the 95%ile of nodes (Table 1), and a timeout of 15 minutes to compare the diversity of the tests generated by EMIAC with that of the baseline rewrite approach (left-hand side) in Figure 10. We observe that EMIAC produces substantially more diverse tests than the baseline. On the right-hand side of the figure, we compare the average number of new states discovered per 10 source test inputs after executing each follow-up test. Although both approaches exhibit a natural decrease, EMIAC maintains a higher rate of new state discovery throughout the generation of follow-up tests. These results support the effectiveness of EMIAC's strategy of caching and selectively traversing the e-graph only when generating new tests: the quality of the generated tests does not degrade significantly as the number of tests increases.

RQ_5 (Test Diversity): Overall, EMIAC produces high-quality follow-up tests with substantially greater diversity than the baseline approach, and its diversity degrades far less as the number of generated tests increases.

7 THREATS TO VALIDITY

EMIAC targets the `IaC` engine core using mock, side-effect-free providers. Consequently, it does not cover provider-induced failures (e.g., authentication, rate limiting, or provider-specific API semantics). Moreover, our metamorphic oracle checks equivalence at the level of computed resource

graphs, missing faults that preserve the graph while changing other externally relevant outcomes (e.g., logs). Concurrency bugs are in scope but appear rare in practice [19]; they may arise when independent resources are deployed in parallel, which EMIAC tests. Although IaC engines reconcile drift, we do not simulate arbitrary perturbations of live infrastructure or the state file; systematically testing drift-handling would require controlled state perturbations, which is beyond this work.

Our evaluation uses randomly generated resource graphs. To improve representativeness, we parameterize the generator using size/depth/sparsity statistics from a large dataset of open-source IaC programs, but open-source programs may under-represent some industrial configurations. Our results indicate that coverage is similar across different graph sizes and between randomly generated graphs and graphs derived from real-world programs.

Finally, because test generation and e-graph exploration are randomized, individual runs may differ. To mitigate this, we repeat each experiment multiple times and report averages. The evaluation shows that the differences between runs are small.

8 RELATED WORK

Metamorphic Testing. MT has been successfully applied to many domains. Mansur et al. [46, 47] applied MT to test Datalog engines, revealing several bugs in state-of-the-art engines. Moreover, MT has been applied to test compilers [42], machine learning applications [75], or web systems [8]. Notably, Rigger and Su [58, 59] applied MT to database management systems and uncovered numerous bugs in popular systems. Segura et al. [61] provide a comprehensive survey of MT techniques and applications. Sun et al. [69] validate SMT solver rewrite systems by exploring the rewrite space via e-graphs and equality saturation. Compared to prior work that tests programs or domain-specific systems, we define MRs directly over resource graphs expressed in an IR, which makes the approach general and engine-agnostic.

Infrastructure as Code. IaC engines are commonly classified as either *configuration-based* or *provision-based* [19]. The former, such as Ansible [37], Puppet [36], and Chef [13], manage the configuration state of existing infrastructure components (e.g., files) on already provisioned machines. Most prior work has focused on analyzing IaC programs, whereas the deployment engines that execute them have received comparatively little attention. For example, Hummer et al. [35] were among the first to investigate deployment behavior, proposing an approach to test idempotency in Chef and revealing a bug in its implementation. Later, Drosos et al. [19] provided a large-scale study of bugs in Ansible, Puppet, and Chef, classifying issues across both programs and deployment engines. Research on Ansible has emphasized defect detection and maintainability: Opdebeeck et al. [51, 52] examined code smells and control-flow anomalies, while Hassan and Rahman [34] analyzed bugs in Ansible scripts. For Chef, security has been the primary concern, with Rahman et al. [57] identifying vulnerabilities in Chef scripts. For Puppet, Sotiropoulos et al. [67] addressed the lack of fault-detection for real-world manifests, introducing a trace-based method that identified missing ordering and uncovered numerous previously unknown issues in widely used modules.

On the other hand, *provision-based engines*, such as Pulumi [71], OpenTofu [53], Terraform [33], and AWS CloudFormation [63], manage cloud resources. Most studies on provision-based engines focus on testing methodologies and program analysis. Sokolowski et al. [65] introduced ProTI, an automated testing framework for efficient unit testing of Pulumi TypeScript programs. ProTI was evaluated on the PIPr dataset [66]. Similarly, Bühler et al. [5] compiled a dataset of Terraform programs. Compared to prior work, EMIAC is the first to test provision-based engines.

E-Graphs. E-Graphs were introduced in the PhD thesis by Nelson [50] for program verification. They extend the union-find data structure [24], which represents an equivalence relation, i.e., one which is reflexive, symmetric, and transitive, with congruence among uninterpreted functions

(UF). The e-graph is now a defacto standard for the implementation of the core UF theory in SMT solvers such as Z3 [16], CVC [2], and OpenSMT [4]. Beyond using e-graphs, Z3 also contributed the standard efficient e-matching algorithm [15]. Solvers beyond SMT such as Simplify [18] and CCLemma [40] have successfully employed e-graphs for the application of theorem proving.

E-Graphs have also found success in program optimization. Denali [39] is a super-optimizer that produces nearly optimal binaries using an e-graph. Denali later inspired Tate et al. [70] to equip e-graphs with a saturation engine for the purpose of optimization. In the real world, Cranelift [20] is a WebAssembly optimizing compiler that uses an (acyclic) e-graph for its optimization phase. Beyond programs, e-graphs have been used by Coward et al. [14] to optimize circuits. Cao et al. [7] introduce Babble, which uses e-graphs, equality saturation, and anti-unification to learn reusable library abstractions and compress program corpora.

After the introduction of the influential deferred rebuilding and e-class analysis extensions in egg [74], there has been active research into making e-graphs more expressive. For example, Zakhour et al. [76] extend e-graphs to reason about disequality as well as equality. EasterEgg [64] and Assume nodes [14] extend the e-graph to reason about conditional equalities. And finally, Slotted E-graphs [60] extend the e-graph to account for α-equivalence for expressions with binders.

Compared to prior work, we employ e-graphs as a test generator and equivalence oracle for `IaC` programs. To the best of our knowledge, EMIAC is the first to use e-graphs for software testing.

Automated Test Generation Techniques. Prior work on automated test generation often relies on coverage-guided generation [41], evolutionary search [27], or fuzzing [48]. We do not adopt these techniques in EMIAC for several reasons. Coverage-guided generation typically requires instrumenting the system under test, which ties the technique to a specific engine, programming language, and build setup, undermining portability [41]. Fuzzing spends substantial effort producing ill-formed or type-invalid configurations that are rejected early, whereas EMIAC generates well-formed programs by construction and focuses testing on the engine core [43]. Evolutionary search repeatedly mutates tests and evaluates candidates using a fitness metric such as coverage, which requires executing the engine inside the generation loop and is expensive [23]. By contrast, EMIAC can generate diverse candidates by walking a saturated e-graph without executing the engine. These techniques are complementary: when instrumentation is available, coverage-guided generation could be integrated with EMIAC. In this work, however, we prioritize portability and evaluate fault-finding capability via mutation analysis rather than optimizing for specific coverage metrics.

9 CONCLUSION

We present EMIAC, the first metamorphic testing framework for `IaC` engines. EMIAC addresses the oracle problem by encoding semantic equivalences over resource graphs in a dedicated IR and verifying them using equality saturation on e-graphs. Based on these equivalences, it systematically generates large families of diverse yet equivalent tests that exercise engine-critical paths.

Our evaluation across Pulumi, Terraform, and OpenTofu shows that EMIAC is practical: it covers 98 and 1,313 previously untested statements in Terraform and Pulumi respectively, and kills 3, 1, and 20 additional mutants in Terraform, Pulumi and OpenTofu. The three test deficiencies have already been merged by Terraform, and a specification fix has been accepted by Pulumi. Beyond immediate testing gains in `IaC`, e-graphs emerge as a general substrate for reasoning about test generation, paving the way for their application in other domains.

ACKNOWLEDGMENTS

This work has been co-funded by the Swiss National Science Foundation (SNSF, Grant No. 10001777), by armasuisse Science and Technology, and by European Union's Horizon research and innovation

programme (CAPE Project, Grant No. 101189899). Matteo Biagiola is partially supported by Fondo Istituzionale per la Ricerca granted by Università della Svizzera italiana (USI).

DATA-AVAILABILITY STATEMENT

The artifact is available on Zenodo [68]. It includes the implementation of EMIaC as well as the scripts which can be executed to reproduce our evaluation results.

REFERENCES

[1] Jon Ayerdi, Valerio Terragni, Gunel Jahangirova, Aitor Arrieta, and Paolo Tonella. 2024. GenMorph: Automatically Generating Metamorphic Relations via Genetic Programming. *IEEE Trans. Software Eng.* 50, 7 (2024), 1888–1900. https://doi.org/10.1109/TSE.2024.3407840

[2] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, et al. 2022. cvc5: A versatile and industrial-strength SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 415–442.

[3] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Trans. Software Eng.* 41, 5 (2015), 507–525. https://doi.org/10.1109/TSE.2014.2372785

[4] Roberto Bruttomesso, Edgar Pek, Natasha Sharygina, and Aliaksei Tsitovich. 2010. The opensmt solver. In *International Conference on TOOLS and Algorithms for the Construction and Analysis of Systems*. Springer, 150–153.

[5] Christoph Bühler, David Spielmann, Roland Meier, and Guido Salvaneschi. 2025. TerraDS: A Dataset for Terraform HCL Programs. In *Proceedings of the IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)* (Ottawa, Canada) *(MSR '25)*. IEEE, 5 pages.

[6] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 209–224.

[7] David Cao, Rose Kunkel, Chandrakana Nandi, Max Willsey, Zachary Tatlock, and Nadia Polikarpova. 2023. babble: Learning Better Abstractions with E-Graphs and Anti-unification. *Proc. ACM Program. Lang.* 7, POPL, Article 14 (Jan. 2023), 29 pages. https://doi.org/10.1145/3571207

[8] Nazanin Bayati Chaleshtari, Fabrizio Pastore, Arda Goknil, and Lionel C. Briand. 2023. Metamorphic Testing for Web System Security. *IEEE Trans. Software Eng.* 49, 6 (2023), 3430–3471. https://doi.org/10.1109/TSE.2023.3256322

[9] Checkov by Prisma Cloud. 2025. Checkov. https://www.checkov.io/

[10] T.Y. Chen, S.C. Cheung, and S.M. Yiu. 1998. *Metamorphic testing: A new approach for generating next test cases.* Technical Report HKUST-CS98-01. Department of Computer Science, The Hong Kong University of Science and Technology, Hong Kong. https://arxiv.org/abs/2002.12543 Technical Report, arXiv:2002.12543.

[11] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. 2016. PIT: a practical mutation testing tool for Java (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 449–452. https://doi.org/10.1145/2931037.2948707

[12] Pulumi Corporation. 2023. Resources deleted for no apparent reason. GitHub issue #2593, pulumi/pulumi-aws. https://github.com/pulumi/pulumi-aws/issues/2593

[13] Progress Software Corporation. 2025. Chef Documentation. https://docs.chef.io Accessed: 2025-02-25.

[14] Samuel Coward, Theo Drane, and George A Constantinides. 2024. Constraint-Aware E-Graph Rewriting for Hardware Performance Optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2024).

[15] Leonardo De Moura and Nikolaj Bjørner. 2007. Efficient E-matching for SMT solvers. In *Automated Deduction–CADE-21: 21st International Conference on Automated Deduction Bremen, Germany, July 17-20, 2007 Proceedings 21*. Springer, 183–198.

[16] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: an efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

[17] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer* 11, 4 (1978), 34–41. https://doi.org/10.1109/C-M.1978.218136

[18] David Detlefs, Greg Nelson, and James B. Saxe. 2005. Simplify: a theorem prover for program checking. *J. ACM* 52, 3 (May 2005), 365–473. https://doi.org/10.1145/1066100.1066102

[19] Georgios-Petros Drosos, Thodoris Sotiropoulos, Georgios Alexopoulos, Dimitris Mitropoulos, and Zhendong Su. 2024. When Your Infrastructure Is a Buggy Program: Understanding Faults in Infrastructure as Code Ecosystems. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 359 (Oct. 2024), 31 pages. https://doi.org/10.1145/3689799

[20] Chris Fallin. 2023. ægraphs: Acyclic E-graphs for Efficient Optimization in a Production Compiler. In *EGRAPHS Workshop*. https://pldi23.sigplan.org/details/egraphs-2023-papers/2/-graphs-Acyclic-E-graphs-for-Efficient-Optimization-in-a-Production-Compiler Invited Talk.
[21] Kate Fazzini. [n. d.]. A Technical Slip-up Exposes Cloud Collaboration Risks. https://www.wsj.com/articles/a-technical-slip-up-exposes-cloud-collaboration-risks-1497353313. Accessed: 2025-05-20.
[22] Firefly. 2025. The State of IaC 2025. https://www.firefly.ai/state-of-iac-2025 Accessed: 2025-10-07.
[23] Gordon Fraser and Andrea Arcuri. 2014. A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite. *ACM Trans. Softw. Eng. Methodol.* 24, 2, Article 8 (Dec. 2014), 42 pages. https://doi.org/10.1145/2685612
[24] Bernard A. Galler and Michael J. Fisher. 1964. An improved equivalence algorithm. *Commun. ACM* 7, 5 (May 1964), 301–303. https://doi.org/10.1145/364099.364331
[25] Google. 2025. Go Coverage Profiling. https://go.dev/doc/build-cover. Accessed 2025-09-15.
[26] Google. 2025. The Go Programming Language. https://go.dev/. Accessed 2025-09-15.
[27] Mark Harman and Phil McMinn. 2009. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Transactions on Software Engineering* 36, 2 (2009), 226–247.
[28] Mark Harman, Jillian Ritchey, Inna Harper, Shubho Sengupta, Ke Mao, Abhishek Gulati, Christopher Foster, and Hervé Robert. 2025. *Mutation-Guided LLM-based Test Generation at Meta*. Association for Computing Machinery, New York, NY, USA, 180–191. https://doi.org/10.1145/3696630.3728544
[29] HashiCorp. 2020. Cycle error when removing a resource along with create_before_destroy. https://github.com/hashicorp/terraform/issues/26226. Accessed: 2025-09-26.
[30] HashiCorp. 2021. Graph missing dependency from resource update to resource destroy. https://github.com/hashicorp/terraform/issues/28150. Accessed: 2025-09-26.
[31] HashiCorp. 2022. Resources disappearing from state after successful apply. GitHub issue #27671, hashicorp/terraform-provider-aws. https://github.com/hashicorp/terraform-provider-aws/issues/27671
[32] HashiCorp. 2025. CDK for Terraform. https://developer.hashicorp.com/terraform/cdktf Accessed: 2025-02-25.
[33] HashiCorp. 2025. Terraform Docs Overview. https://developer.hashicorp.com/terraform/docs Accessed: 2025-02-25.
[34] Mohammad Mehedi Hassan and Akond Rahman. 2022. As Code Testing: Characterizing Test Quality in Open Source Ansible Development. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 208–219. https://doi.org/10.1109/ICST53961.2022.00031
[35] Waldemar Hummer, Florian Rosenberg, Fábio Oliveira, and Tamar Eilam. 2013. Testing Idempotence for Infrastructure as Code. In *Middleware 2013*, David Eyers and Karsten Schwan (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 368–388.
[36] Puppet Inc. 2025. Puppet Documentation. https://www.puppet.com/docs/index.html Accessed: 2025-02-25.
[37] RedHat Inc. 2025. Ansible Community Documentation. https://docs.ansible.com Accessed: 2025-02-25.
[38] Laura Inozemtseva and Reid Holmes. 2014. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th international conference on software engineering*. 435–445.
[39] Rajeev Joshi, Greg Nelson, and Keith Randall. 2002. Denali: a goal-directed superoptimizer. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (Berlin, Germany) *(PLDI '02)*. Association for Computing Machinery, New York, NY, USA, 304–314. https://doi.org/10.1145/512529.512566
[40] Cole Kurashige, Ruyi Ji, Aditya Giridharan, Mark Barbone, Daniel Noor, Shachar Itzhaky, Ranjit Jhala, and Nadia Polikarpova. 2024. CCLemma: E-Graph Guided Lemma Discovery for Inductive Equational Proofs. *Proc. ACM Program. Lang.* 8, ICFP, Article 264 (Aug. 2024), 27 pages. https://doi.org/10.1145/3674653
[41] Leonidas Lampropoulos, Michael Hicks, and Benjamin C Pierce. 2019. Coverage guided, property based testing. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–29.
[42] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 216–226. https://doi.org/10.1145/2594291.2594334
[43] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random testing for C and C++ compilers with YARPGen. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 196 (Nov. 2020), 25 pages. https://doi.org/10.1145/3428264
[44] Stephan Lukasczyk and Gordon Fraser. 2022. Pynguin: automated unit test generation for Python. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 168–172. https://doi.org/10.1145/3510454.3516829
[45] Kurt Mackie. [n. d.]. Configuration Glitch Caused Microsoft's Jan. 25 Exchange Online Disruption. https://redmondmag.com/articles/2023/01/26/configuration-glitch-caused-microsoft-jan-25-exchange-online-disruption.aspx. Accessed: 2025-05-20.

[46] Muhammad Numair Mansur, Maria Christakis, and Valentin Wüstholz. 2021. Metamorphic testing of Datalog engines. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 639–650.

[47] Muhammad Numair Mansur, Valentin Wüstholz, and Maria Christakis. 2023. Dependency-Aware Metamorphic Testing of Datalog Engines. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, René Just and Gordon Fraser (Eds.). ACM, 236–247. https://doi.org/10.1145/3597926.3598052

[48] Barton P. Miller, Lars Fredriksen, and Bryan So. 1990. An empirical study of the reliability of UNIX utilities. *Commun. ACM* 33, 12 (Dec. 1990), 32–44. https://doi.org/10.1145/96267.96279

[49] Chandrakana Nandi, Max Willsey, Adam Anderson, James R. Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. 2020. Synthesizing structured CAD models with equality saturation and inverse transformations. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 31–44. https://doi.org/10.1145/3385412.3386012

[50] Charles Gregory Nelson. 1980. *Techniques for program verification.* Stanford University.

[51] Ruben Opdebeeck, Ahmed Zerouali, and Coen De Roover. 2022. Smelly Variables in Ansible Infrastructure Code: Detection, Prevalence, and Lifetime. In *Proceedings of the 19th International Conference on Mining Software Repositories* (Pittsburgh, Pennsylvania) *(MSR '22)*. Association for Computing Machinery, New York, NY, USA, 61–72. https://doi.org/10.1145/3524842.3527964

[52] Ruben Opdebeeck, Ahmed Zerouali, and Coen De Roover. 2023. Control and Data Flow in Security Smell Detection for Infrastructure as Code: Is It Worth the Effort?. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE, Melbourne, Australia, 534–545. https://doi.org/10.1109/MSR59073.2023.00079

[53] OpenTofu. 2025. OpenTofu Documentation. https://opentofu.org/docs/ Accessed: 2025-03-03.

[54] Owain Parry, Gregory M. Kapfhammer, Michael Hilton, and Phil McMinn. 2022. A Survey of Flaky Tests. *ACM Trans. Softw. Eng. Methodol.* 31, 1 (2022), 17:1–17:74. https://doi.org/10.1145/3476105

[55] Goran Petrović, Marko Ivanković, Gordon Fraser, and René Just. 2022. Practical Mutation Testing at Scale: A view from Google. *IEEE Transactions on Software Engineering* 48, 10 (2022), 3900–3912. https://doi.org/10.1109/TSE.2021.3107634

[56] Kun Qiu, Zheng Zheng, Tsong Yueh Chen, and Pak-Lok Poon. 2022. Theoretical and Empirical Analyses of the Effectiveness of Metamorphic Relation Composition. *IEEE Trans. Software Eng.* 48, 3 (2022), 1001–1017. https://doi.org/10.1109/TSE.2020.3009698

[57] Akond Rahman, Md Rayhanur Rahman, Chris Parnin, and Laurie Williams. 2021. Security Smells in Ansible and Chef Scripts: A Replication Study. *ACM Trans. Softw. Eng. Methodol.* 30, 1, Article 3 (jan 2021), 31 pages. https://doi.org/10.1145/3408897

[58] Manuel Rigger and Zhendong Su. 2020. Detecting optimization bugs in database engines via non-optimizing reference engine construction. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1140–1152. https://doi.org/10.1145/3368089.3409710

[59] Manuel Rigger and Zhendong Su. 2020. Finding bugs in database systems via query partitioning. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 211 (Nov. 2020), 30 pages. https://doi.org/10.1145/3428279

[60] Rudi Schneider, Marcus Rossel, Amir Shaikhha, Andrés Goens, Thomas Kœhler, and Michel Steuwer. 2025. Slotted E-Graphs: First-Class Support for (Bound) Variables in E-Graphs. *Proceedings of the ACM on Programming Languages* 9, PLDI (2025), 1888–1910.

[61] Sergio Segura, Gordon Fraser, Ana Belén Sánchez, and Antonio Ruiz Cortés. 2016. A Survey on Metamorphic Testing. *IEEE Trans. Software Eng.* 42, 9 (2016), 805–824. https://doi.org/10.1109/TSE.2016.2532875

[62] Amazon Web Services. 2025. AWS CDK. https://docs.aws.amazon.com/cdk/v2/guide/home.html Accessed: 2025-02-25.

[63] Amazon Web Services. 2025. AWS CloudFormation. https://aws.amazon.com/cloudformation/ Accessed: 2025-10-08.

[64] Eytan Singher and Shachar Itzhaky. 2024. Easter egg: Equality reasoning based on E-graphs with multiple assumptions. In *2024 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 70–83.

[65] Daniel Sokolowski, David Spielmann, and Guido Salvaneschi. 2024. Automated Infrastructure as Code Program Testing. *IEEE Trans. Software Eng.* 50, 6 (2024), 1585–1599. https://doi.org/10.1109/TSE.2024.3393070

[66] Daniel Sokolowski, David Spielmann, and Guido Salvaneschi. 2024. The PIPr Dataset of Public Infrastructure as Code Programs. In *Proceedings of the 21st International Conference on Mining Software Repositories* (Lisbon, Portugal) *(MSR '24)*. Association for Computing Machinery, New York, NY, USA, 498–503. https://doi.org/10.1145/3643991.3644888

[67] Thodoris Sotiropoulos, Dimitris Mitropoulos, and Diomidis Spinellis. 2020. Practical fault detection in puppet programs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 26–37. https://doi.org/10.1145/3377811.3380384

[68] David Spielmann, George Zakhour, Dominik Arnold, Matteo Biagiola, Roland Meier, and Guido Salvaneschi. 2026. Artifact for Metamorphic Testing for Infrastructure-as-Code Engines. https://doi.org/10.5281/zenodo.18755966

Software artifact.
[69] Maolin Sun, Yibiao Yang, Jiangchang Wu, and Yuming Zhou. 2025. Validating SMT Rewriters via Rewrite Space Exploration Supported by Generative Equality Saturation. *Proceedings of the ACM on Programming Languages* 9, OOPSLA2 (2025), 1205–1231.
[70] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality saturation: a new approach to optimization. *SIGPLAN Not.* 44, 1 (Jan. 2009), 264–276. https://doi.org/10.1145/1594834.1480915
[71] Pulumi Team. 2025. Pulumi Docs. https://www.pulumi.com/docs/ Accessed: 2025-02-24.
[72] Avito Tech. 2024. go-mutesting: Mutation testing for Go source code. https://github.com/avito-tech/go-mutesting. Accessed: 2025-08-05.
[73] Vpn Monitor Research Team. 2019. Report: Travel Reservations Platform Leaks US Government Personnel Data. https://www.vpnmentor.com/blog/us-travel-military-leak/. Accessed: 2025-05-20.
[74] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.* 5, POPL, Article 23 (Jan. 2021), 29 pages. https://doi.org/10.1145/3434304
[75] Xiaoyuan Xie, Joshua Ho, Christian Murphy, Gail Kaiser, Baowen Xu, and Tsong Yueh Chen. 2009. Application of Metamorphic Testing to Supervised Classifiers. In *Proceedings of the 2009 Ninth International Conference on Quality Software (QSIC '09)*. IEEE Computer Society, USA, 135–144. https://doi.org/10.1109/QSIC.2009.26
[76] George Zakhour, Pascal Weisenburger, Jahrim Gabriele Cesario, and Guido Salvaneschi. 2025. Dis/Equality Graphs. *Proc. ACM Program. Lang.* 9, POPL, Article 77 (Jan. 2025), 24 pages. https://doi.org/10.1145/3704913

Received 2025-10-10; accepted 2026-02-17