Automated Infrastructure as Code Program Testing

Daniel Sokolowski, David Spielmann, and Guido Salvaneschi

***Abstract*—Infrastructure as Code (IaC) enables efficient deployment and operation, which are crucial to releasing software quickly. As setups can be complex, developers implement IaC programs in general-purpose programming languages like TypeScript and Python, using PL-IaC solutions like Pulumi and AWS CDK. The reliability of such IaC programs is even more relevant than in traditional software because a bug in IaC impacts the whole system. Yet, even though testing is a standard development practice, it is rarely used for IaC programs. For instance, in August 2022, less than 1 % of the public Pulumi IaC programs on GitHub implemented tests. Available IaC program testing techniques severely limit the development velocity or require much development effort. To solve these issues, we propose Automated Configuration Testing (ACT), a methodology to test IaC programs in many configurations quickly and with low effort. ACT automatically mocks all resource definitions in the IaC program and uses generator and oracle plugins for test generation and validation. We implement ACT in ProTI, a testing tool for Pulumi TypeScript with a type-based generator and oracle, and support for application specifications. Our evaluation with 6 081 programs from GitHub and artificial benchmarks shows that ProTI can directly be applied to existing IaC programs, quickly finds bugs where current techniques are infeasible, and enables reusing existing generators and oracles thanks to its pluggable architecture.**

***Index Terms*—Property-based testing, fuzzing, infrastructure as code, DevOps.**

I. Introduction

INFRASTRUCTURE as Code (IaC) automates software operations [1] and is a key tool in organizations that aim for reliable, high-throughput software development and deployment [2]. With IaC, developers specify provisioning, deployment, and configuration in text-based files that are amenable to well-known software engineering practices like version control, code review, and continuous integration. As a result, IaC enables faster, more reproducible software operations [3], [4], [5], [6].

IaC started with *imperative* scripts, but meanwhile, many more robust, *declarative* IaC solutions are available. With declarative IaC, developers describe the target state of the deployment instead of the deployment steps [7], [8], which are automatically derived. Deployments can be complex—a trend also driven by modern systems often consisting of several small components. For example, applications that have consisted of a monolithic web server and a database now may comprise tens or hundreds of serverless functions and microservices. This trend transfers complexity from components to their composition, resulting in long, structured IaC scripts. To cope with such complexity, developers implement IaC programs—in contrast to IaC scripts—with recent declarative IaC solutions that adopt general-purpose languages, e.g., TypeScript, Python, or Java, and not only configuration languages and DSLs with constrained expressivity like JSON and YAML. Such *Programming Languages IaC (PL-IaC)* solutions come with all abstractions (and tools) of well-known general-purpose programming languages. To the best of our knowledge, the industrial-strength PL-IaC solutions available today are Pulumi [9], the Cloud Development Kit (CDK) of Amazon Web Services (AWS CDK) [10], and the CDK for Terraform (CDKTF) [11]. They have existed since 2018–2020 with quickly growing communities. The NPM core packages of AWS CDK, CDKTF, and Pulumi alone grew from 11 M downloads in 2020 to 146 M downloads in 2023.[1] Pulumi reported growth from hundreds to 2 000 customers and tens of thousands to 150 000 end users in the same period [12], [13].

Testing IaC programs is an open research problem and critical in practice. For example, Rahman et al. [6] urge in their mapping study of IaC research for more work on testing, and Guerriero et al. [3] found that declarativity and "impossible testing" are the most mentioned differences between IaC and traditional software in 44 semi-structured interviews with senior developers. The lack of suitable testing techniques is especially apparent for PL-IaC: while studies found that more than 50 % of public software projects on GitHub use testing [14], [15], we found only 25 % of the PL-IaC programs use testing, dropping to 1 % for general PL-IaC [16], which only Pulumi implements. Because Pulumi provides more-advanced support and is more open than the CDKs (Section II-A), we focus on Pulumi.

Current testing techniques for PL-IaC (Section II-B) pose a dilemma (Section II-C): integration testing is notoriously slow and potentially causes high infrastructure costs. Unit testing is the only alternative without these issues, but insightful unit tests for IaC programs require high development effort. Every

Manuscript received 24 November 2023; revised 8 April 2024; accepted 13 April 2024. Date of publication 1 May 2024; date of current version 14 June 2024. This work is partially supported by the Swiss National Science Foundation (SNSF) under Grant 200429 and by Armasuisse S+T. Recommended for acceptance by P. Runeson. *(Corresponding author: Daniel Sokolowski.)*

The authors are with the University of St. Gallen, 9000 St. Gallen, Switzerland (e-mail: daniel.sokolowski@unisg.ch; david.spielmann@unisg.ch; guido.salvaneschi@unisg.ch).

Digital Object Identifier 10.1109/TSE.2024.3393070

[1]According to https://npm-stat.com/ for `aws-cdk-lib`, `@aws-cdk/core`, `cdktf`, and `@pulumi /pulumi`.

resource definition has to be replaced with a mock faithfully modeling the cloud resource and implementing configuration validation and generation logic. Mocking code easily becomes more complex than the IaC program under test itself, resulting in very few projects using systematic testing—despite testing being crucial for the high-velocity development of reliable software [17], [18].

To enable efficient testing of IaC programs, we propose Automated Configuration Testing (ACT). ACT is an automated framework allowing developers to rapidly unit-test IaC programs in hundreds of configurations without writing any code. ACT uses existing mechanisms to mock all resource definitions in the IaC program automatically. In the mocks, a generator provides test input, and oracles validate resource configurations. The approach is open and enables reuse across projects through pluggable third-party generators and oracles.

We implement ACT in the ProTI testing tool for Pulumi TypeScript with a default generator and oracle leveraging type information from Pulumi package schemas. The evaluation on 6 081 Pulumi TypeScript programs from GitHub and generated artificial benchmarks shows that (1) ProTI can find bugs reliably *and* quickly compared to existing testing techniques, (2) ProTI can be applied to IaC programs without any changes, (3) ProTI finds bugs often within seconds or tens of seconds, and (4) ProTI can leverage existing generator and oracle tools through simple plugins.

This work is the first investigation of testing for IaC programs. It is *relevant* because the popularity of PL-IaC is continuously increasing. Further, testing IaC is open research with a high impact on the security and reliability of software systems. ACT and ProTI are *novel* because they introduce efficient testing of IaC programs and *significant* because they can improve the velocity of correct IaC program development. Also, ProTI's pluggable architecture enables researchers to experiment with new oracles and generators specialized for PL-IaC. Lastly, our work follows scientific standards [19] *rigorously*, and we disclose all developed software, analysis scripts, and data to ensure *verifiability, transparency, reusability, and recoverability*. ProTI[2] is open source and publicly maintained on GitHub[3] with long-term archived releases [20]. The remaining evaluation material is published under the CC-BY-4.0 license on Zenodo [21]. We only exclude analyzed third-party code that does not permit re-distribution. In summary, this paper contributes:

1) A comparison of testing methods for IaC programs, establishing the testing dilemma of PL-IaC, which is backed by our previous repository mining study [16].
2) Automated Configuration Testing (ACT): A novel approach for efficient unit testing of IaC programs.
3) ProTI: A testing tool implementing ACT for Pulumi TypeScript that is pluggable with third-party generators and oracles and provides default type-based generators and oracles based on Pulumi package schemas.
4) An evaluation with 6 081 Pulumi TypeScript programs from GitHub and benchmarks, showing that ProTI applies to existing IaC programs, can efficiently find bugs, and can leverage existing tools as test generators and oracles through plugins.

[2]https://proti-iac.github.io.
[3]https://github.com/proti-iac/proti.

Fig. 1. High-level architecture of PL-IaC solutions.

II. PL-IaC and the Testing Problem

We introduce PL-IaC (Section II-A) and IaC program testing (Section II-B), discuss how existing testing techniques fall short (Section II-C), and outline our solution in Section II-D.

A. Programming Languages IaC (PL-IaC)

PL-IaC solutions adopt a general-purpose programming language, e.g., Python or TypeScript. IaC programs define the declarative *target state* of the deployment as a directed acyclic graph (DAG). Each node is a resource with its configuration, and the arcs are dependencies between resources that constrain the deployment order. For instance, a (virtual) server must be created before a web application on it. Similarly, the server should be created before its DNS record.

While IaC programs describe the target state, the *deployment engine* performs the actual deployment actions and maintains the deployment's state. The deployment engine receives the target state from the IaC program, compares it with the current state, and performs the required actions to fill the gap (Fig. 1). Further, it provides the IaC program with information about the deployment state such that the program can observe infrastructure information only available after a resource was created, e.g., a dynamically assigned IP.

To define the target state, PL-IaC solutions provide an embedded DSL that is available as libraries for many programming languages. These SDK libraries simply provide a class for each deployable resource type. In an IaC program, developers define a resource (i.e., a node in the target state) by instantiating an object of the resource type's class. The resource's *input configuration* is provided as an argument to the constructor. After the deployment engine deploys a resource, its post-deployment *output configuration* is available as properties on the resource's object. Developers explicitly define a dependency from a resource A to a resource B (i.e., an arc from node A to B in the target state) by referencing B or one of its output properties in A's input configuration. Alternatively, such a dependency can be defined implicitly by instantiating A in a program part that depends on an output property of B. Defined resources, their properties, and their dependencies are immutable. Thus, the target state is monotonically growing throughout the IaC program execution, and defined resources and dependencies can neither be changed nor removed.

Listing 1. RWW example: Pulumi TypeScript program that deploys a static website on AWS S3 showing a random word.[4]

```
1.1  import * as pulumi from '@pulumi/pulumi';
1.2  import * as aws    from '@pulumi/aws';
1.3  import * as random from '@pulumi/random';
1.4
1.5  const words = ['software', 'is', 'great'];
1.6  const bucket = new aws.s3.Bucket('website', {
1.7      website: { indexDocument: 'index.html' }
1.8  });
1.9  const range = { min: 0, max: words.length };
1.10 const rng = new random.RandomInteger('word-id', range);
1.11 rng.result.apply((wordId) => {
1.12     new aws.s3.BucketObject('index', {
1.13         bucket: bucket, key: 'index.html',
1.14         contentType: 'text/html; charset=utf-8',
1.15         content: '<!DOCTYPE html>' +
1.16                  words[wordId].toUpperCase()
1.17     });
1.18 });
1.19
1.20 export const url = bucket.websiteEndpoint;
```

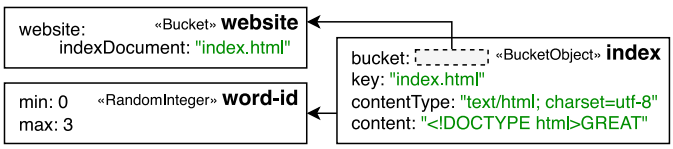

Fig. 2. Example of a target state described by Listing 1.[4]

As a running example, we introduce the Pulumi TypeScript IaC program of the Random Word Website (RWW) in Listing 1, which defines the target state in Fig. 2. It deploys a static website on AWS S3 [22] that displays a word randomly selected from the array in Line 1.5. Lines 1.6–1.8 define the S3 bucket and Line 1.10 the `word-id` resource. It receives `range` (Line 1.9) as input configuration and is assigned to `rng`. After `word-id` is deployed, the deployment engine provides a randomly drawn number as the `result` field of the resource's output configuration. Such output configuration values are available as properties of the resource objects, in this case as `rng.result`. To access the value, `apply` (Line 1.11) registers a callback (Lines 1.11–1.18), which executes as soon as the random number is available. The number is used to select a word from the `words` array in Line 1.16, which is capitalized and set as `content` in the input configuration of the `index` resource (Lines 1.12–1.17). The dependence of `index` on `word-id` is defined implicitly by defining `index` in the apply callback, a program part depending on `word-id`'s output configuration. The dependence on the S3 bucket is made explicit by referencing its object in the input configuration (Line 1.13). Finally, Line 1.20 exports the website's URL.

Fig. 3 shows the PL-IaC architecture in detail. The IaC program (e.g., Listing 1) uses the IaC solutions' SDK (e.g., Pulumi's SDK is imported in Line 1.1) to deploy an application in the cloud. A cloud is a collection of resources controllable through an API, e.g., the random number generator and AWS public cloud in Listing 1. For each cloud, there is a *provider*,

[4]For brevity, we omit the bucket's ownership controls, public access block, and policy resources that are required to allow public access from the Internet.

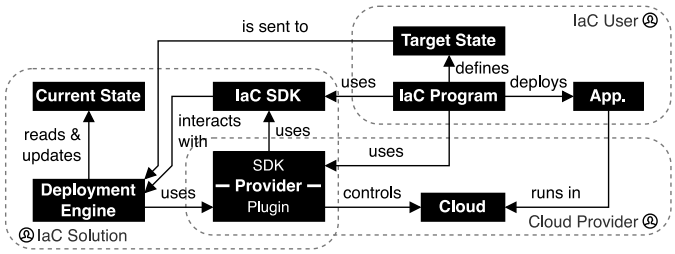

Fig. 3. Roles and relationships in PL-IaC solutions.

the component in the PL-IaC architecture that implements the cloud-specific SDK library and deployment engine plugin to define and control the cloud's resources (for Pulumi, e.g., there are providers for AWS, Azure, Google Cloud, etc.). IaC programs use the provider SDK libraries (e.g., Pulumi's AWS and random provider SDK libraries are imported in Line) and instantiate objects of their resource types (e.g., the S3 bucket in Lines 1.6–1.8) to define resources in the target state. The deployment engine receives the target state from the IaC program and compares it to the current state, which it maintains. To reach the target state, the deployment engine uses the provider plugins to control the specific clouds, i.e., to perform actions to create, read, update, delete, and list the resources (CRUDL actions).

To our knowledge, Pulumi is the only industrial-grade IaC solution implementing this PL-IaC approach to its full extent. Pulumi features a CLI that orchestrates the concurrent execution of the IaC program and the deployment engine. Both other available PL-IaC solutions—AWS CDK and CDKTF—support a weaker approach, *two-phase* PL-IaC (Section VI). Hence, we focus on Pulumi and TypeScript, which is the most popular programming language in PL-IaC [16]. We do not investigate cloud SDKs, e.g., AWS SDK [23] and Azure SDK [24], because they target the imperative, low-level management of resources. PL-IaC abstracts the complexities of CRUD operations for IaC program developers and hides such SDKs in the provider plugins.

B. Testing IaC Programs

Fig. 4 shows the PL-IaC testing techniques available for Pulumi programs [25], ordered top-to-bottom by time consumption. *Unit Testing* IaC programs is like in traditional software: IaC users run (parts of) the program with a unit testing framework, mock objects with side effects, i.e., every resource definition in an IaC program, and add checks. Even with runtime mocking—like supported by Pulumi—developers still have to provide the mocking logic. *Dry Running* simply executes the IaC program without executing deployment actions, providing a quick indication of whether the program terminates and a preview of the target state. Yet, the preview of dry running is incomplete and neither supports specific checks nor ensures sufficient coverage. Dry running does not execute code paths that depend on values available only after a resource was created. *Resource Property Testing* and *Stack Property Testing*, e.g., with CrossGuard [26], solve these issues by performing the deployment, making them integration testing techniques. They

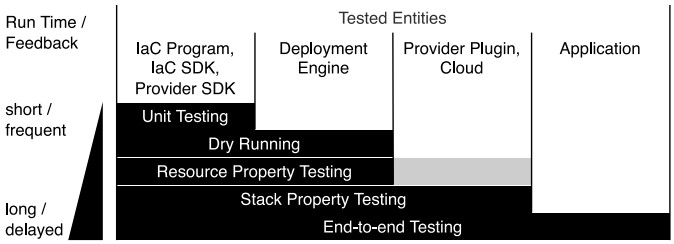

Fig. 4. PL-IaC testing techniques and their coverage, ordered by expected run time and feedback cycle frequency.

TABLE I
TESTING TECHNIQUES IN PUBLIC PULUMI PROGRAMS ON GITHUB (AUGUST 2022) [16]

	# Projects	(% of Total)
Pulumi projects	12 945	(100 %)
with unit testing	118	(1 %)
with property testing	33	(0 %)
with end-to-end testing	22	(0 %)

check resource configurations against policies before their deployment and the final observed post-deployment state. *End-to-end Testing*, e.g., Pulumi's integration testing framework [27], runs the IaC program and validates the resulting deployment—not only its observed state.

To shed light on the adoption of these testing techniques, we analyzed all public Pulumi projects on GitHub in August 2022 in our PIPr dataset. For the detailed method, results, and replication scripts, we refer to the PIPr dataset paper [16]. Briefly, we extracted specific keywords and the file extensions of all files in the projects' directory and mapped this data to testing techniques. Table I summarizes the results, showing that less than 1 % of all 12 945 projects implement systematic testing. This minuscule share indicates that developers perceive systematic testing impractical for IaC programs.

C. The Dilemma of IaC Program Testing

Even though neglected by PL-IaC developers, systematic testing is crucial for the high-velocity development of IaC—no less than for traditional software [17], [18]. Without testing, e.g., it is easy to miss the bug in Listing 1: The random number ranges from zero to three (Line 1.9), but the `words` array index only from zero to two. If three is drawn, Line 1.16 calls `toUpperCase()` on `undefined`, causing an error. We now consider today's PL-IaC testing techniques for Listing 1.

For integration testing IaC programs, including end-to-end and property testing, a single run of Listing 1 takes at least seconds. Programs with more complex resources may require hours and cause high infrastructure costs. Testing only a few configurations can miss corner-case bugs, like in Listing 1.

Dry running is fast and does not require coding. Yet, it cannot find many errors, including the one in Listing 1, because it does not execute code depending on output configuration that is only available post-deployment, e.g., the `apply` callback in Lines 1.11–1.18.

Unit testing PL-IaC is labor-intensive and error-prone compared to developing the program. First, one has to mock *all* resource definitions—three in Listing 1. This step is not problematic per se, e.g., by adopting the runtime mocking Pulumi provides. Yet, to create effective mocks, developers must implement validation logic for the input configuration and generate output configurations as test inputs for the rest of the program. Such code simulates the logic of cloud configuration, which is complex and requires a correct model. Lastly, developers must ensure the tests cover all relevant cases and may need to update mocks with every change.

In summary, developers face a dilemma when testing IaC programs: They either invest excessive programming effort for efficient unit testing or resort to expensive integration testing, hampering development velocity.

D. Automated IaC Testing to the Rescue

To solve this issue, we propose Automated Configuration Testing (ACT), a novel unit testing methodology for fast testing of IaC programs with low development effort. ACT automatically mocks all resource definitions. Each mock implements a test oracle to validate the resources' input configurations and a test generator that provides the resources' output configurations. With this level of automation, the PL-IaC program can be tested in many different configurations without writing testing code.

We implement ACT in ProTI, a testing tool for Pulumi TypeScript. ProTI's results depend on oracles and generators, which are pluggable to foster reuse across programs. ProTI is equipped with a default oracle and generator based on types of resource configurations from Pulumi package schemas. Further, ProTI provides a specification mechanism to refine oracles and generators in the IaC program where needed. ProTI tests the example in Listing 1 in hundreds of different configurations in a short time with no changes to its code. In each test, ProTI validates all resource configurations, including different number values for `wordId`. ProTI will likely test a case where Line 1.16 fails, detecting the bug in seconds.

III. AUTOMATED CONFIGURATION TESTING

We now introduce Automated Configuration Testing (ACT), a novel testing methodology for IaC programs. To effectively address the testing dilemma (Section II-C), ACT is a *unit testing* technique because the core issue of integration testing, being slow and resource-intensive, is caused by the cloud providers, e.g., AWS and Azure, and cannot be significantly improved at the side of IaC developers. Thus, we aim to understand and minimize the developer's unit testing effort.

A. Why Unit Testing IaC Programs is Effortful: Mocks

Efficient unit testing requires eliminating of integration with external, slow, and resource-intensive components. For IaC programs, this means mocking the interaction with the cloud, which is encapsulated in resource definitions. To this end, all resource object instantiations, a substantial part of the IaC program's code, must be mocked—most of the code in the RWW example (Listing 1).

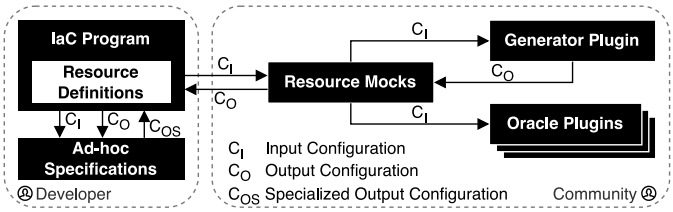

Fig. 5. Overview of ACT.

Mocking all resource definitions with a naïve mock is trivial, requiring, e.g., in Pulumi TypeScript, only a couple of lines of code—independent of the IaC program's size. Yet, for effective unit testing, the mocks have to implement the cloud logic in two crucial aspects. (1) The mocks have to return an output configuration for each resource input configuration they receive. This is because, in a real deployment, the cloud provides the resource's output configuration to the IaC program after the resource deployment. As the output configurations are accessible in the remaining IaC program, they indeed constitute *test input*. Thus, the returned output configurations have to be realistic to test the remaining IaC program precisely. Further, to cover all paths, it may be necessary to return different output configurations across test executions. (2) To test the declarative target state the IaC program defines, i.e., the cloud configuration to set up and not only the imperative IaC program execution, the mocks have to validate the received resource input configurations (i.e., they have to implement *test oracles*). This is because the cloud provides feedback to the IaC program on the resource input configurations by reporting an error when an invalid configuration is deployed.

Such oracles should be *intentless*, i.e., they reject configurations that are generally invalid, independent of the IaC program's context. Ideally, they are further *intentful*, i.e., they also reject configurations that violate the IaC program's application-specific goals.

Finally, the significant challenge is that mocks have to implement both suitable test generators *and* oracles. Suitable test generators ensure coverage and minimize false positives because they do not generate unrealistic test inputs that trigger issues that would never occur in practice. Suitable test oracles verify the cloud configuration the IaC program defines. Both are non-trivial and require a significant amount of code—likely a multiple of the IaC program's code. Further, such mocks mirror the logic of the IaC program under test and the cloud it uses, leading to code tightly coupled with the IaC program, ultimately slowing down any future changes.

B. Automating Unit Testing With ACT

To solve these issues, we propose ACT (Fig. 5). ACT automatically mocks all resource definitions by intercepting the constructors of resource classes, e.g., the constructor of `aws.s3.Bucket` in Lines 1.6–1.8 of Listing 1. The ACT resource mocks receive the input configuration of each resource and return suitable output configurations. The resource mocks implement both a test generator and a set of test oracles.

A test generator provides a separate value producer for each test case. During the execution of a test case, its value producer receives the resources' input configurations and returns for each an output configuration. As the output configurations are test input, the test case is ultimately defined as the sequence of output configurations its value producer returns. An oracle is a predicate function that decides whether the resource's input configurations are valid. We distinguish between two kinds of oracles. *Resource oracles* receive individual resource input configurations during the IaC program execution. *Deployment oracles* receive the input configuration of all resources after the IaC program completed, enabling holistic validations.

In ACT, both the generator and the oracles are plugins, allowing for exchange, adoption, and experimentation with test generators and oracles. Ideally, these plugins implement generalized, reusable generation and validation strategies decoupled from a specific IaC program. ACT solves the issue of unit testing IaC programs by moving the development effort of testing code from the developers of an individual IaC program to the community. Once the community instantiates ACT for a specific platform (e.g., .Net or Python; our reference implementation covers Pulumi Typescript) and provides suitable plugins, developers can test the basic correctness of the imperative IaC program and its target state without implementing any code.

ACT's approach fosters the reuse of plugins across different applications. However, to ensure that testing is also based on application-specific knowledge (e.g., *intentful* oracles, Section III-A), a mechanism to augment the community-provided generators and oracles with application-specific generation and validation specifications is needed. For this, ACT implementations can leverage various approaches, e.g., specification DSLs separated from or embedded into the IaC program code. ProTI features *ad-hoc specifications*, an embedded DSL integrated into the IaC program code (Section IV-C).

C. Running Test Sequences With ACT

With automated test execution, generation, and validation, ACT can execute the IaC program in many different configurations. For a sequence of tests, the generator plugin provides a different value producer for each test case. The test case selection it performs is crucial, i.e., which value producer instances it chooses, as it determines which and how parts of the IaC program are tested. ACT terminates once an oracle finds a bug, the program under test crashes, or, if no bug is found, after a defined amount of runs or a timeout. Thus, a generator's prioritization and selection of test cases is crucial to ensure relevant bugs are triggered (early).

Conceptually, ACT combines property-based testing (PBT) [28], [29] and fuzzing [30] techniques for IaC programs. Both systematically test a program s in many configurations $c \in C$, which are put into relation by a property p, leading to $\forall c \in C.\, p(c, s(c))$ if s is correct. However, the pessimistic assumption is that s contains a bug, yielding the goal to find and test a configuration e leading to $\neg p(e, s(e))$ as early as possible in a sequence of tests. As the generator plugin is exchangeable, ACT is amenable to new state-of-the-art fuzzing and PBT test case

selection strategies, e.g., based on testing feedback [31], search-based techniques [32], code coverage [33], and combinatorial coverage [34].

D. Discussion

We now discuss bugs in IaC programs, ACT's design, its relation to cloud models, and the resulting limitations.

1) IaC Program Bugs: We propose a bug taxonomy for IaC programs. In contrast to previous, more fine-grained bug taxonomies, e.g., for IaC defects by Rahman et al. [35], we focus purely on the required oracle to find a bug. Recent fuzzing literature, e.g., Su et al. [36] and Li et al. [33], commonly distinguishes *crash bugs* that cause the program to crash and non-crashing *logic bugs*, which require a more precise oracle than crash detection to identify erroneous computations. We add two categories for bugs where the program logic may be correct, but the resulting resource configuration is faulty. *Configuration bugs* are the wrong configuration of an isolated resource, e.g., setting an IPv4 address to the invalid value `400.0.0.1`. With *configuration interaction bugs*, the configuration of the individual resources is valid but invalid in combination. For example, there is a subnet `192.168.0.1/24` and a server in it has the IP address `192.168.1.2`, which is invalid in this subnet. In contrast to crash and logic bugs, configuration bugs require oracles that can identify invalid cloud configurations and, for configuration interaction bugs, even across multiple resources.

Crash bugs and logic bugs are related to "traditional" code, while configuration (interaction) bugs are related to the embedded DSL code in IaC programs that defines the target state of the deployment through instantiating objects of the resource types' classes. However, IaC programs mix traditional code (Lines 1.1–1.5, Line 1.9, and Line 1.20 of Listing 1) with the embedded DSL code (Lines 1.6–1.8 and Lines 1.10–1.18). This mixing prevents testing the kinds of code in isolation and causes existing testing methods to be only applicable with a huge mocking effort (Section III-A).

2) ACT's Approach: ACT focuses on finding configuration (interaction) bugs. To this end, static analysis is a suitable alternative; for example, it can easily find the bug in Listing 1. Yet, we base ACT on automated testing because it does not incur the limitations of static analysis when covering complex dynamic behavior of the IaC program code and supporting all features of the host language. In such systematic testing, the generator has to exercise the IaC program in different configurations to find crash and logic bugs that yield wrong configurations effectively. We argue that covering such configuration-related crash and logic bugs is sufficient because IaC programs focus on the configuration, and all relevant logic drives this purpose. If an IaC program implements complicated configuration-unrelated logic, it should be separated from the embedded DSL code and specifically checked with existing, well-established testing techniques.

3) Cloud Configuration Models: Generators and oracles implicitly define models of cloud resource configuration. Such models could be derived from specifications, be hand-crafted, or, more realistically, be derived from existing approximate models, including *types*. For instance, Pulumi providers, i.e., vendor-specific plugins (cf. Section II-A) used by Pulumi to interact with the cloud, are distributed as packages that contain a schema JSON file defining the types of the resources' target and output configuration. Such type definitions are a configuration model that is by design available for all resources Pulumi supports—even for dynamically typed languages—and they can be leveraged for *type-based* generators and oracles [29]. ACT's open architecture ensures that developers can adopt and combine available models and plug in domain-specific optimizations. ACT is not limited to functional properties. For instance, models of cloud performance and security, predicting bad performance and insecure setups based on resource configurations, can be embedded in ACT oracle plugins to cover such non-functional aspects.

Ideally, models for ACT generators and oracles are (1) *complete*, i.e., they can produce all valid configurations, and (2) *correct*, i.e., they include only valid configurations. Incomplete models in a generator systematically prevent generating test cases that may be needed to find bugs, and incorrect models can yield test cases that never occur in practice. Incomplete models in oracles can trigger false positives (i.e., alerts in the absence of a bug) and incorrect models false negatives (i.e., missing bugs). In practice, cloud models are not perfect. For instance, Pulumi package schema types are complete but not fully correct. In RWW (Listing 1), a correct generator should generate integers in the range (Line 1.9) for `RandomInteger`'s `result` field (Line 1.11). Yet, a type-based generator provides any number, including outside the range and fractions, because the type of `RandomInteger.result` is number. Similarly, a correct oracle only accepts valid HTML for the `content` field (Line 1.16), but a type-based one accepts any string.

In practice, useful test generators and oracles may still generate irrelevant tests or miss bugs. Even if application-specific knowledge can further limit the configuration space, correcting the model in generator and oracle plugins may overfit the plugins to the specific program, reducing reusability or slowing down development. ACT addresses these issues by enabling fine-tuning of test generation and oracles for a specific application, e.g., ProTI provides an ad-hoc specifications syntax (Section IV-C).

IV. ProTI: ACT for Pulumi TypeScript

We present ProTI, an instantiation of ACT for Pulumi TypeScript. ProTI is built upon the popular JavaScript testing tool Jest [37], fast-check [38] for the test execution strategy and arbitraries, and Pulumi's runtime mocking. ProTI comprises six TypeScript packages (Table II). The first four packages implement the core abstractions and Jest plugins for a Jest runner, test runner, and reporter. `@proti-iac/pulumi-packages-schema` is a Pulumi-packages-schema-based oracle and a generator plugin. `@proti-iac/spec` implements the ad-hoc specification syntax. ProTI is used through Jest's CLI, which's configuration it facilitates with a preset. ProTI preserves Jest's pre-test features and optimizations, e.g., an in-memory file system for the code.

TABLE II
PROTI PACKAGES: NON-BLANK, NON-COMMENT SLOC

Package	Description	Source SLOC	Test SLOC
@proti-iac/core	Core abstractions	758	863
@proti-iac/runner	Jest runner	26	51
@proti-iac/test-runner	Jest test runner	429	90
@proti-iac/reporter	Jest reporter for check results	149	19
@proti-iac/spec	Ad-hoc specifications	12	74
@proti-iac/pulumi-packages-schema	Pulumi packages schema infrastructure, oracle, and generator	1 334	1 960
Total		2 708	3 057

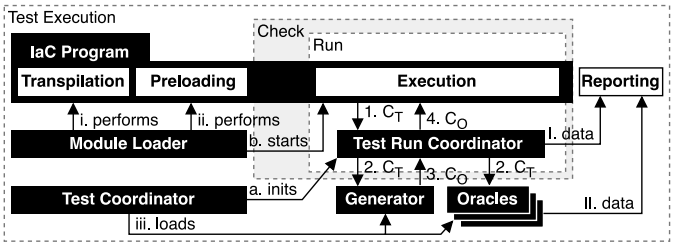

Fig. 6. ProTI test execution: i. – iii. initialization, a. – b. run initialization, 1. – 4. resource mocking, I. – II. reporting.

A. Test Execution With ProTI

Jest *runners* distribute tests over multiple workers. They invoke a *test runner* for each test suite. ProTI's runner extends Jest's default by (1) verifying the test configuration and (2) forwarding file system and module resolution information to ProTI's test runners, which they had to re-generate otherwise.

ProTI's test runner is invoked once on the `Pulumi.yaml` of each IaC program and implements ACT (Section III). Fig. 6 details the test execution. First, the IaC program and its dependencies are transpiled to JavaScript (i) and a configured set of dependencies is preloaded (ii). Preloaded modules are shared among all IaC program runs, breaking isolation but reducing overhead. For technical reasons, Pulumi's SDK must be preloaded. Further, the test coordinator loads the generator and oracle plugins (iii). ProTI *checks* the IaC program in several *runs*, each configured with its own test run coordinator (a), managing isolated run states for the generator and oracles. Each run executes the IaC program once (b). ProTI mocks all resource definitions by intercepting the constructors of all resource classes with Pulumi's runtime mocking feature. This way, each resource input configuration C_I is run through validations and transformations that the provider's SDK may implement in the resources' constructors. We call the checked and potentially transformed C_I target configuration C_T. For instance, a resource's C_I may contain additional fields, which is valid in TypeScript's structural type system, but the resource constructor does not add them to C_T. ProTI uses C_T instead of C_I in the remaining ACT workflow. The mock provides all resources' target configuration C_T to the generator and oracle plugins (1 – 2), and receives the output configurations C_O to use in the remaining program execution (3 – 4). Finally, ProTI reports the test results (I – II).

B. Test Generator and Oracle Plugins

In an execution, ProTI loads exactly one generator plugin and a variable number of oracle plugins, which are invoked in parallel. We do not provide an explicit mechanism to compose different plugins; however, when developers write a plugin's code, they can also combine other plugins programmatically. ProTI plugins are implemented as NodeJS modules, exporting the respective plugin as default and, optionally, an `init` function of ProTI's `TestModuleInitFn` type that can implement initialization code called by ProTI when loading the plugin and also implements a plugin configuration interface. `@proti-iac/core` implements all plugin-related types.

Generator plugins are implemented as fast-check value generators of ProTI's `Generator` type, i.e., type `Arbitrary<Generator>`. The arbitrary is called once for each test run to provide a `Generator` and may implement shrinking, a technique from property-based testing where, once an error is found, simplified versions are tested and presented to the developer as an easier-to-understand alternative if they still trigger the bug [29]. The test run's generator is invoked for each resource with its target configuration and returns its output configuration for the run. Further, the generator is invoked with the arbitrary of each ad-hoc generator specification, guiding its execution to enable deterministic test generation strategies, including shrinking.

Oracle plugins are implemented as a class inheriting from ProTI's `Oracle<S>` type and can leverage state of type `S` that is initialized for every test run through a function they implement and passed to all invocations of the oracle in the run. For these invocations, oracles implement at least one out of four resource input configuration validation interfaces, which are separately called for each resource or once with all resources, both available synchronously and asynchronously.

For now, ProTI provides default generator and oracle plugins based on Pulumi packages schema types in `@proti-iac/pulumi-packages-schema`. The package implements the infrastructure to automatically retrieve the schemas of all resources in the IaC program under test. The oracle translates the schemas' resource types to validation functions to dynamically check each resource input configuration. The generator composes fast-check arbitraries to generate output configurations, inheriting fast-check's random value generation strategy, which is biased towards generating extremes, e.g., instead of using an even distribution, it prioritizes generating small and big values. However, ProTI can be easily extended with oracles and generator arbitraries based on other model sources, e.g., codified policies and cloud specifications.

C. Ad-Hoc Specifications in ProTI

To fine-tune generators and oracles, ProTI provides *ad-hoc specification* syntax. `generate(e).with(a)` defines an ad-hoc specification to replace values returned by `e` with values from a fast-check arbitrary `a`. For ad-hoc oracles, `expect(e).to(p)` applies an oracle predicate function to an expression `e`. In a regular execution, the ad-hoc syntax only returns the evaluation of the wrapped expression `e`, with no

Listing 2. Listing 1 with ProTI ad-hoc specifications (orange).

```
2.1  import * as ps from '@proti-iac/spec';
...  // Same as Lines 1.1 to 1.8
2.10 const range = { min: 0, max: words.length - 1 };
2.11 const rng = new random.RandomInteger('word-id', range);
2.12 ps.generate(rng.result).with(ps.integer(range))
2.13     .apply((wordId) => {
2.14         new aws.s3.BucketObject('index', {
2.15             bucket: bucket, key: 'index.html',
2.16             contentType: 'text/html; charset=utf-8',
2.17             content: '<!DOCTYPE html>' +
2.18                 ps.expect(words[wordId].toUpperCase())
2.19                     .to((s) => s.length > 0)
2.20         });
2.21     });
2.22 export const url = bucket.websiteEndpoint; // Like Line 1.20
```

change in the semantics of the IaC program. When running with ProTI, however, `generate(e).with(a)` calls the generator plugin with `a` and returns a value from `a`. The oracle syntax still returns the evaluation of `e`, but it introduces a check, reporting an error if `p(e)` is `false` or fails.

Listing 2 fixes the indexing bug in Listing 1 and is fine-tuned with ad-hoc specifications. In ProTI executions, the ad-hoc generator specification in Line 2.12 addresses the imprecision of the type-based oracle in Line 1.11, which generates any number, not only realistic output configuration values. Instead, Line 2.12 specifies to only generate integer values in the correct `range` interval. Further, Lines 2.18–2.19 specify an oracle that checks that the webpage's content is not empty, encoding the developers' application-specific intent to show a non-empty webpage.

While ProTI implements this embedded specification DSL for application-specific generator and oracle directives, ACT implementations could use external DSLs or encourage separating the specification code into other files, as common with most testing frameworks today. Such separation is also possible with ProTI's ad-hoc specifications but would require restructuring the IaC program code to improve testability. For instance, the code augmented with specifications in Listing 2 could be wrapped in functions that separate testing files mock during testing. We support inlining in ProTI for simplicity, assuming that few ad-hoc specifications are required with good plugins. Yet, if a lot of ad-hoc specifications are required, separation is preferable to avoid the added complexity of mixing concerns, obfuscating the IaC program code, and potentially introducing new error sources.

V. Evaluation

We evaluate ACT's effectiveness, applicability, performance, and extensibility by answering the following research questions about its ProTI implementation.

- **RQ1:** *Can ProTI find bugs reliably?* We determined whether ProTI can find bugs quickly and reliably compared to existing PL-IaC testing techniques (cf. Section II-B).
- **RQ2:** *Is ProTI applicable to real-world open-source code?* We explored whether ProTI can be applied to existing, real-world IaC programs.
- **RQ3:** *How long does ProTI run, and how does the run time scale?* We measured ProTI's execution duration and scalability to ensure it is fast enough for realistic IaC programs.
- **RQ4:** *Can existing test generation and oracle tools be integrated into ProTI?* We investigated whether ACT allows to leverage third-party oracles and generators.

The following four subsections present our experiments and Section V-E discusses their results and threats to validity. We ran all experiments on serverless AWS Fargate [39] containers with 1 vCPU and 4 GB of memory on AWS Elastic Container Service (ECS) [40] in the eu-west-1 region (Ireland).

A. Finding Errors in IaC Programs

We compared ProTI with the available testing techniques for Pulumi TypeScript programs (cf. Section II-B) on nine variants of the RWW example (Section II-A). The variants are the following. VC is correct, VS is Listing 2, i.e., VC with ProTI ad-hoc specifications, and VSDB adds the deployment of a serverless database to VS. Most remaining variants have a crash bug according to our bug taxonomy (Section III-D): VNT has syntax errors, VE always throws an error, which VAE throws asynchronously, VO is Listing 1, i.e., it has a one-off bug in asynchronous code that leads to a crash, which we combine with the ad-hoc specifications of Listing 2 in VSO. VSB is VS with a configuration bug, setting a string instead of an object for the bucket's website property (cf. Line 1.7).

ProTI was configured with the type-based oracle and generator (Section IV) and up to 100 runs. Unit testing used Jest [37] and ran the program once with a naïve mock that returned empty configurations. Dry running executed `pulumi preview`. (Dry) property testing executed Pulumi CrossGuard [26] via (`pulumi preview`) `pulumi up` with the AWSGuard policy pack [41]. All Pulumi commands were non-interactive with skipped previews. End-to-end testing used Pulumi's Go integration testing framework [27], checking the content of the deployed website. We executed each experiment 10 times after warmup. Table III reports whether an error was (always) found and the minimum and average run time.

As expected, dry running did not find asynchronous errors (VAE, VO, and VSO) as it does not run code depending on unknown output configurations. Property testing and end-to-end testing found the one-off bugs (VO and VSO) only occasionally. ProTI was the only technique that spotted all errors reliably. However, the imprecision of the type-based generator, i.e., generating any number for `rng.result` and not only integer values in the defined range, increased the likelihood of finding the error in VO, but also caused that ProTI identified VC as faulty; a false positive. This imprecision is resolved in VS, VSO, VSB, and VSDB with ProTI ad-hoc specifications (cf. Section IV-B). ProTI always identified bugs in the first test run, except in VSO, where it required 2 to 6 tests, causing a slightly longer run time compared to VO.

> RQ1: ProTI can find bugs reliably and is able to uncover errors in edge cases without explicitly testing for them.

TABLE III
PL-IAC TESTING TECHNIQUES ON VARIANTS OF THE RWW EXAMPLE (LISTING 1). ∗ FAULTY VARIANT. ERROR FOUND ∗ (ALWAYS ⊛), MINIMUM (AVERAGE) RUN TIME OVER 10 REPETITIONS

		ProTI Unit Test		Dry Run Dry Property Test		Property Test End-to-end Test	
*	VNT: Non-transpilable	⊛ 16.7 s	(16.8 s)	⊛ 10.0 s	(10.3 s)	⊛ 12.4 s	(12.4 s)
		⊛ 1.9 s	(2.0 s)	⊛ 11.6 s	(11.7 s)	⊛ 47.9 s	(65.7 s)
*	VE: Error	⊛ 7.0 s	(7.2 s)	⊛ 2.3 s	(2.4 s)	⊛ 4.4 s	(4.5 s)
		⊛ 2.2 s	(2.2 s)	⊛ 3.7 s	(3.8 s)	⊛ 52.5 s	(59.6 s)
*	VAE: Async Error	⊛ 7.4 s	(7.6 s)	3.4 s	(3.5 s)	⊛ 9.4 s	(9.6 s)
		⊛ 2.4 s	(2.5 s)	4.8 s	(4.9 s)	⊛ 50.8 s	(60.7 s)
	VC: Correct	⊛ 7.5 s	(7.6 s)	3.4 s	(3.4 s)	9.5 s	(9.7 s)
		2.7 s	(2.7 s)	4.8 s	(4.9 s)	53.5 s	(59.0 s)
	VS: Listing 2 (ad-hoc specs.)	21.0 s	(21.1 s)	3.5 s	(3.5 s)	9.5 s	(9.7 s)
		2.8 s	(2.9 s)	5.0 s	(5.0 s)	52.6 s	(62.3 s)
*	VO: Listing 1 (one-off bug)	⊛ 7.4 s	(7.6 s)	3.4 s	(3.4 s)	∗ 9.4 s	(9.6 s)
		2.7 s	(2.7 s)	4.8 s	(4.9 s)	∗ 51.9 s	(58.4 s)
*	VSO: Listing 2 with one-off bug	⊛ 8.1 s	(8.3 s)	3.5 s	(3.6 s)	∗ 9.5 s	(9.7 s)
		2.8 s	(2.9 s)	4.9 s	(5.0 s)	∗ 59.5 s	(66.6 s)
*	VSB: Listing 2 with config. bug	⊛ 7.6 s	(7.8 s)	⊛ 3.5 s	(3.5 s)	⊛ 5.6 s	(5.7 s)
		2.8 s	(2.9 s)	⊛ 4.8 s	(4.9 s)	⊛ 48.4 s	(57.4 s)
	VSDB: Listing 2 with AWS RDS	39.2 s	(39.6 s)	8.1 s	(8.4 s)	163.4 s	(189.9 s)
		3.1 s	(3.1 s)	8.0 s	(8.1 s)	212.5 s	(265.7 s)

TABLE IV
EXECUTION TIME AND RESULT CLASSIFICATION OF PROTI EXECUTIONS ON 6 081 PULUMI TYPESCRIPT PROGRAMS

Category # programs.	Error Reason [# Programs. (% in Category)]	Execution Time average (std)
Project 2 (0 %)	invalid Pulumi.yaml 2 (100 %)	1.6 s (0.1 s)
Transpilation 2 649 (44 %)	module resolution 1 335 (50 %), type checking 984 (37 %), program resolution 324 (12 %), legacy NodeJS 5 (0 %), JSX 1 (0 %)	8.9 s (5.6 s)
Preloading 482 (8 %)	module resolution 410 (85 %), legacy NodeJS/Pulumi 20 (4 %), unknown 18 (4 %), syntax error 18 (4 %), config 16 (3 %)	7.8 s (5.9 s)
Checking 1 633 (27 %)	setup 659 (40 %), mocking 468 (29 %), missing type definition 416 (25 %), application 86 (5 %), other 64 (4 %), oracle 58 (4 %)	17.2 s (17.2 s)
Passed 772 (13 %)		23.4 s (11.4 s)
Crashed 543 (9 %)	out of memory 473 (87 %), unknown 70 (13 %)	25.9 s (38.9 s)
Total 6 081 (100 %)		14.4 s (17.0 s)

B. Applicability to Real-World Programs

We executed ProTI on all 6 081 Pulumi TypeScript programs in the PIPr dataset [16] of all public IaC programs on GitHub in August 2022. PIPr contains examples, toy projects, and production projects in unknown shares and is only filtered by the relevance criteria inherent to the GitHub Code Search API [42] we used for the evaluation and that we discuss in detail in the dataset's paper [16]. PNPM was used to install dependencies and TypeScript version 5.1.6 for the execution.

The first two columns of Table IV show the results. We categorized the executions by the phase in the ProTI run where a problem was detected: invalid *project* files that prevent execution, failures during *transpilation*, failures during module *preloading*, failures during *checking*, successfully *passed*, and *crashed* executions. Within each category, we grouped errors by common causes and report their frequency. Both the categorization and error labeling are based on string matching on the execution logs, and the error grouping by open coding. This process was incrementally performed and implemented by the first author and reviewed by the second author. The authors know Pulumi and ProTI well through their research.

On a technical level, ProTI was able to test 40 % of the IaC programs out of the box. This share is extremely remarkable and exceeds our initial expectations because (1) we did not filter for buggy or non-functional programs, (2) ran all programs with current NodeJS and TypeScript versions, and (3) did neither look into nor provide any program-specific environments. We suspect that ProTI can be used for most of the remaining IaC programs, too, after little effort is invested to understand their expected execution environment or bug.

The most common reasons why ProTI could not test a program are module resolution and type checking, failing 1 745 (29 %) and 984 (16 %) executions. The causes include incompatibility with PNPM, the TypeScript version, unmet environment assumptions, and incomplete, broken setups. Among the programs ProTI was able to test, it found issues in 68 %. The tests found 659 (11 %) executions where the setup was incomplete, e.g., missing configuration or programs. Mocking failed in 468 (8 %) executions, which can be caused by incompatible, outdated Pulumi versions. Our type-based oracle and generator failed to find type definitions in 416 (7 %) executions because they are dynamic resources, stack references, or missing in the provider's schema. Our oracle identified invalid resource configurations in 58 (1 %) executions. ProTI ran only an unknown number of tests in crashed executions, 100 tests in the passing ones, and only a single test in 98 % of the executions under checking. In the other 26 checking executions, ProTI ran between 2 and 38 tests until an error was found. Due to a lack of ground truth, we cannot determine the precision and recall of the experiment.

> RQ2: ProTI can be applied to existing IaC programs.

C. Execution Duration and Scaling Behavior

We performed time measurements on Pulumi programs that define 0, 1, 10, 50, and 100 AWS S3 bucket resources. The experiment considered two program variants, one defining the resources *independently* for parallel deployment, and one in a dependency *chain* for sequential deployment. We ran ProTI three times on each program and repeated the experiment five times. As the programs are correct, ProTI runs them 100 times in each execution without identifying a bug. Table V and Fig. 7 report the average execution time in total and separated by phase. Table V shows the absolute values separately for the first and consecutive runs. Fig. 7 separates the first, second, and third runs and also shows results as relative values.

Execution times are higher for first runs because the transpilation overhead is significant and, on average, 76 % lower in subsequent runs (Fig. 7). Test runs, transpilation, and module

TABLE V
TOTAL AVERAGE EXECUTION TIME OF PROTI OVER 5 REPETITIONS OF THE DURATION EXPERIMENTS BY FIRST/CONSECUTIVE EXECUTION AND PHASE FOR IAC PROGRAMS WITH 0, 10, 50, AND 100 RESOURCES WITH BOTH INDEPENDENT AND CHAINED DEPENDENCIES

	Resources:	0	1	10		50		100	
	Phase			indep.	chain	indep.	chain	indep.	chain
Run 1	Remaining	1.7 s	1.6 s	2.2 s	2.2 s	6.3 s	4.6 s	14.8 s	15.9 s
	100 Runs	1.0 s	9.3 s	57.4 s	69.5 s	274.5 s	262.8 s	563.3 s	535.8 s
	Preloading	0.7 s	0.7 s	0.7 s	0.7 s	0.7 s	0.7 s	0.7 s	0.7 s
	Transpilation	15.1 s	15.2 s	15.3 s	15.3 s	15.2 s	15.3 s	15.3 s	15.3 s
	Total	18.5 s	26.8 s	75.6 s	87.7 s	296.6 s	283.4 s	594.2 s	567.8 s
Run 2 & 3	Remaining	1.6 s	1.6 s	2.2 s	2.3 s	6.3 s	6.3 s	11.1 s	10.0 s
	100 Runs	1.4 s	7.4 s	51.7 s	50.3 s	260.8 s	243.1 s	520.2 s	493.6 s
	Preloading	0.8 s	0.8 s	0.8 s	0.8 s	0.7 s	0.8 s	0.8 s	0.8 s
	Transpilation	3.7 s	3.7 s	3.7 s	3.7 s	3.7 s	3.7 s	3.7 s	3.7 s
	Total	7.5 s	13.5 s	58.3 s	57.1 s	271.6 s	253.9 s	535.7 s	508.1 s

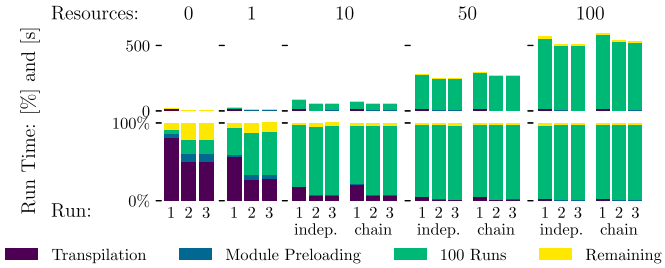

Fig. 7. Average execution time of ProTI over 5 repetitions of the duration experiments (Table V) by phase, resource count, and dependency. Results for three consecutive executions (1, 2, 3). In total (top row) and relative (bottom row).

preloading are the only actions of the test runner taking significant time. The remaining execution time was consumed outside the test runner, including Jest's setup and reporting. A single test run in the experiments took 10 ms to 5.9 s, and the duration scales linearly with the resource number.

We found similar execution times in the IaC programs from GitHub (Table IV). Conservatively approximating a single test duration by dividing the total run time of all *passed* ProTI executions by 100 (the number of runs), we measured test run durations from 34 ms to 1.0 s; 234 ms on average. It is an approximation because the total run time also includes overhead like setup and reporting, and it is conservative because we assume these contributors are instant, i.e., the test run duration is likely a bit lower. The RWW experiments (Table III) confirm these durations, too. Lastly, our experiments show that ProTI is quicker when it finds a bug because of early termination.

The experiments passing RWW experiments with 6 resources (VS) and 25 resources (VSDB) in Table III confirm that test time grows with the number of resources (on average, 21 s and 40 s including overhead). They further show that the performance of integration testing heavily depends on the deployment time of the resources—which ProTI is independent of. Deploying AWS RDS databases takes longer than AWS S3 resources, yielding testing VSDB takes 20× and 4× longer than VS with property testing and end-to-end testing, respectively, while ProTI was only 2× slower.

RQ3: A single test run of ProTI typically takes hundreds of milliseconds and test duration scales with the number of resources—not with their deployment time—permitting to quickly check hundreds of configurations.

D. Integrating Existing Tools Into ProTI

ACT's effectiveness is crucially dependent on the quality of its plugins. Many techniques have been developed for test generation and oracles (cf. Section III-C). To leverage advanced techniques from related work, ProTI must be open to extension with them. To demonstrate ProTI's extendability, we implemented ProTI plugins using the Radamsa fuzzer [43] and the Daikon invariant detector [44] with a generator and an oracle plugin based on existing tools. This experiment assesses the feasibility of integrating existing approaches; optimizing them and evaluating their effectiveness and efficiency is the subject of future work focusing on test generation and oracle techniques, while this paper focuses on the overall approach.

Radamsa [43] is a fuzzing tool that derives fresh test inputs from an example. We adopted it for a ProTI generator plugin that, separately for each resource type, uses the type-based generator to generate an output configuration example, which is passed to Radamsa as JSON to generate a list of derived test inputs. We filter non-parsable configurations from Radamsa's results and use the remaining ones as test input in ProTI. Whenever ProTI runs out of Radamsa-generated inputs, we repeat the procedure. The generator implementation required 83 SLOC, of which only 48 differ from a naïve generator returning empty configurations.

Daikon [44] is a dynamic invariant detector that identifies application invariants in a set of program traces. We used it for an invariant regression oracle that detects behavior changes across different versions of an IaC program. In the first ProTI execution, the oracle records all resources' target and output states and invokes Daikon on them to find resource configuration invariants over all runs, e.g., a particular bucket's id equals a field of a policy, independent of the concrete value. In consecutive ProTI executions, we repeat the procedure and additionally compare the obtained invariants with the previously generated ones, issuing a warning if an invariant cannot be found anymore, i.e., it may be violated in the new program version. The oracle plugin comprises only 120 SLOC, mainly for converting resource configurations between ProTI and Daikon and managing state across executions.

RQ4: Existing tools can be integrated into ProTI by implementing a plugin, demonstrating ProTI's openness to third-party techniques.

E. Limitations, Threats to Validity, and Implications

Our experiments on ProTI show that ACT can find bugs quickly and reliably in IaC programs, even in edge cases (RQ1),

can be applied to IaC programs without adjustments (RQ2), can be fast enough to run hundreds of tests in a short time (RQ3), and can be extended with existing tools through generator and oracle plugins (RQ4). Yet, our experiments do not provide quantitative insight into ACT's effectiveness, i.e., the likelihood that all bugs and no false positives are found and after which time. Such insights require an IaC program dataset *with correctness annotations*, i.e., precise knowledge about bugs in them. Such evaluation is planned in future work to assess advanced generator and oracle plugins. This paper focuses on the feasibility of the ACT approach to test IaC, not on the precision and recall of a specific testing technique.

Relevant threats to validity in this work include that we evaluate ACT through ProTI, a single instantiation for one specific PL-IaC solution and language. Yet, we expect that implementations for other languages and PL-IaC solutions yield similar results because IaC programs for other tools and other languages, i.e., the embedded PL-IaC DSL, are, technically, analogous. The IaC program selection in our experiments is also a threat. For RQ1, the set of variants in RWW suffices to demonstrate the behavioral differences of ACT compared to other techniques; yet, more experiments are needed to show with statistical significance that these differences are relevant in practice such that ACT is beneficial on other IaC programs. For RQ2, we inherit the limitations and validity threats of the PIPr dataset [16]—including generalizability—but, based on our experience, we expect the qualitative insight to apply to other IaC programs. For RQ3, we focused on the number of resources and their dependencies in IaC programs, showing how they influence performance. We rely on our experience that resource number and dependency are the factors that most significantly impact performance, but other factors can be studied with a more comprehensive sensitivity analysis. The categorization, as well as the error labeling and grouping in RQ2, may be subjective, an issue we limited through the review of a second author. Another potential issue is that ProTI is a random-based testing tool, which, in case of a bug, may cause the bug to be inconsistently (not) caught by different test cases across executions. Hence, we apply 10 repetitions for RQ1. For RQ2, we saw negligible variance in tests. As the programs in RQ3 are correct, they are not impacted by this threat. RQ4 is also not affected because it only demonstrates that existing tools can be leveraged in ACT. RQ4 does not measure ProTI executions to quantify the effectiveness of specific tools in the context of IaC. This aspect must be evaluated for each plugin and crucially depends on the implemented method.

For practitioners, ACT and ProTI are new techniques whose effectiveness depends, in the long run, on a community effort to maintain the framework and the test generation and oracle plugins. Practitioners can now try out ACT with low effort on existing Pulumi TypeScript IaC programs. This solution can already reduce the development time through earlier bug detection and increase the reliability of IaC programs, supporting faster evolving, functional, secure systems. A user study assessing user acceptance of ACT and ProTI is left to future work. For researchers, ACT and ProTI are novel testbeds that facilitate exploring advanced test generation and oracle techniques for IaC programs and correct and secure cloud configuration.

VI. Related Work

We summarize the limitations of two-phase PL-IaC solutions and related work on infrastructure deployment quality, automated mocking, and related software testing techniques.

A. Limitations of Two-Phase PL-IaC

General PL-IaC solutions like Pulumi can observe a resource's state after deployment, the output configuration, and process the values in the general-purpose language. In contrast, two-phase PL-IaC solutions like AWS CDK and CDKTF prohibit IaC programs from accessing the deployment state. Two-phase PL-IaC solutions (1) execute the IaC program to generate the target state as a JSON file and (2) provide it to the deployment engine, i.e., AWS CloudFormation or Terraform. Such exchange is uni-directional, i.e., with no arrow from the deployment engine to the IaC program in Fig. 1. Due to this approach, two-phase PL-IaC can only compute on resource state that can be expressed in the deployment engine's DSL—practically limited to referencing values, string interpolation, and simple value processing. Yet, using an expressive language to process the externally generated state is the reason for using general-purpose languages in IaC programs in the first place. Accordingly, two-phase PL-IaC only provides a subset of PL-IaC's capabilities. In fact, AWS CDK code can be embedded into Pulumi programs, but not vice versa [45].

Unit testing two-phase PL-IaC, i.e., for CDKs [46], [47], is simpler than for general PL-IaC and does not require mocking as two-phase IaC programs do not interact with the deployment engine. We believe this simplification is the reason unit testing is much more common in CDK projects than Pulumi [16]. Also, template projects set up through the CLI include a unit testing setup with a simple test for CDKs (commented in the templates by default), but not for Pulumi.

B. Infrastructure as Code Quality

Previous work discussed IaC quality and how to improve it, but it is mainly focused on Puppet, Ansible, and Chef. These *Configuration as Code* (CaC) tools have been designed to configure existing, mutable infrastructure—even though they also support provisioning. In contrast, PL-IaC does not only employ general-purpose programming languages instead of DSLs; it also focuses on infrastructure provisioning (like, e.g., Terraform and AWS CloudFormation), typically implementing immutable infrastructure management. Research on PL-IaC has been limited to deployment coordination [48], [49].

Hummer et al. [50] proposed an idempotency testing approach for Chef scripts, which Ikeshita et al. [51] augmented with verification techniques to minimize the size of the required test suite. Shambaugh et al. [52] proposed Rehearsal to verify the determinacy and idempotency of Puppet scripts. Yet, declarative IaC ensures idempotency by design.

Sharma et al. [53] were the first to identify code smells in Puppet scripts. Later studies confirmed them for Chef [54]. Rahman et al. surveyed CaC research [6] and identified source code properties correlating with defects in Puppet scripts [55], such as hard-coded strings. They further recognized security

smells and proposed linters for Puppet, Ansible, and Chef [56], [57]. Saavedra and Ferreira [58] introduced GLITCH for linters on a CaC-solution-agnostic intermediate representation. Reis et al. [59] found that such linters are too imprecise but can be improved through user feedback.

Opdebeeck et al. [60], [61] analyzed the quality of semantic versioning and variable-precedence-related code smells in Ansible. Further, they applied program dependence graph analysis to Ansible scripts, motivating control- and data-flow analysis for IaC security smell detection techniques [62]. Dalla Palma et al. [63], [64], [65] proposed various quality metrics and an AI defect prediction framework for Ansible scripts. Kumara et al. [4] and Guerriero et al. [3] explored IaC best practices and issues in the industry through a grey literature survey and practitioner interviews. Hassan and Rahman [66] studied bugs in open-source Ansible test scripts. Borovits et al. [67] proposed FindICI, an AI-based tool to identify linguistic inconsistency between documentation, comments, and code in Ansible scripts, and Chiari et al. [68] surveyed work on static analysis for IaC, focusing mainly on CaC.

This paper is the first about quality in PL-IaC, which focuses—unlike CaC—on declarative infrastructure provisioning through programs in popular imperative programming languages. Further, we propose ACT and implement it in ProTI, enabling efficient unit testing of IaC programs.

C. Correctness of Infrastructure and Architecture Modeling

Modeling languages are textual or graphical languages that express a system's structure. The Topology and Orchestration Specification for Cloud Applications (TOSCA) is a modeling language for the topology of cloud applications, resources, and their orchestration [69]. Bellendorf and Mann [70] surveyed existing literature on TOSCA. Wurster et al. [71] presented a systematic review of declarative deployment technologies and introduced a metamodel for their common core. A follow-up work [72] leverages such core to define TOSCA Light, a subset of TOSCA, aiming to reconcile research modeling and industrial practice. TOSCA Light enables the transformation of compliant deployment models to technology-specific models.

Architecture description languages (ADL) define application components and their relationships. For example, ArchJava [73] embeds such specifications in Java source code, enabling architecture compliance checks at compile time. Krüger et al. [74] introduced ORS for the compositional specification, deployment, and dynamic reconfiguration of systems of services. In contrast to other established ADLs, ORS separates the application from infrastructure concerns. Terra and de Oliveira Valente [75] proposed specifying and statically enforcing dependencies in the software architectures to avoid erosion. Placement types are a language approach where the type system checks architectural conformance [76], [77].

ProTI verifies the correct composition of infrastructure configuration, e.g., through type-based oracles, and enables application-specific checks through ad-hoc specifications. On top, ProTI tests the imperative IaC program generating it.

D. Infrastructure Verification

Ensuring infrastructure correctness has been extensively studied. AWS investigated automatically verifying infrastructure properties [78], [79], [80], leading to at least two automated services in production: AWS Tiros verifies reachability queries on virtual networks [81] and AWS Zelkova performs access verification on role-based AWS IAM policies [82]. These solutions verify already deployed setups, but their techniques should be applicable pre-deployment on IaC programs, which encode the infrastructure's configuration. Such pre-deployment infrastructure verification could also leverage more foundational techniques. E.g., Alloy [83] is a language and analysis tool to verify structural properties of software. Ahrens et al. [84] developed a proof system for invariants on reconfigurable distributed systems. Evangelidis et al. [85] proposed probabilistic verification of performance properties of rule-based auto-scaling policies. Lastly, Abu Jabal et al. [86] gave a comprehensive overview of techniques for policy verification, focused on access control and network management.

Program verification remains an open challenge, either requiring significant manual effort or being limited to specific properties [87]. Augmenting ACT with automated verification of domain-specific properties, e.g., network access constraints, is a promising direction, orthogonal to ACT's contribution to the testing of IaC programs.

E. Automated Mocking

In a study on mocking in open source systems, Spadini et al. [88] found that developers mock components that are difficult to handle and that mocking code increases the coupling between system and test code. According to the authors, the results motivate the need for mock synthesis. Taneja et al. [89] proposed MODA, using an efficient, SQL-aware mock and advanced test generation techniques to automatically test database applications. Solms and Marshall [90] automatically generated mocks from explicit component contracts. Various works synthesize mocks from interaction traces of components [91], [92], [93]. In contrast, Zhu et al.'s StubCoder [94] synthesizes mocks for regression testing solely from the tests' code, without running the mocked component.

Mocking resource definitions in IaC programs is trivial because PL-IaC solutions provide an interface to intercept them, eliminating the need for advanced mocking techniques. Yet, the mocks' test generation and validation logic are complex. ACT encapsulates them into plugins, enabling the integration of mocking techniques from literature into ProTI.

F. Fuzz and Property-Based Testing

Fuzz testing (fuzzing) discovers software vulnerabilities, typically by treating the program as a closed box and testing it for hangs and crashes. Yet, input-value-generation-guided approaches exist; for example, grammar-based fuzzing is an active research field [95], [96]. Li et al. [33] and Zeller et al. [30] provided an overview of state-of-the-art fuzzing techniques. Property-based testing (PBT) [28], [29] is a related approach,

where code is exercised on randomly generated tests, and results are checked against invariants—the properties.

Various works investigate effective PBT test generators. Lampropoulos et al. proposed Luck, a language for PBT generators [97], and coverage-guided PBT [98]. Löscher and Sagonas introduced targeted PBT [32] and automated it [99] using search-based techniques to guide the generation. Kuhn et al. [100] found that most bugs are caused by the interaction of only a few parameters, motivating combinatorial testing [101], which Goldstein et al. [34] applied to PBT generators by modifying the random generator distributions. On the intersection with formal methods, Paraskevopoulou et al. [102] integrated PBT into a proof assistant to verify tests, and Lampropoulos et al. [103] compiled logical conditions (inductive relations) to generators and to their soundness and completeness proofs. De Angelis et al. [104] leveraged symbolic execution and constraint logic programming to automatically derive generators.

ACT is fuzzing and PBT for IaC programs. For ProTI, type-based generators and oracles, prototypes demonstrating third-party tool integration, and an ad-hoc specification syntax are available. The approaches above can be integrated or implemented in ProTI plugins to use them for IaC programs.

VII. Conclusion

Testing is rarely used for IaC programs, and available techniques either hinder development velocity or require much programming effort. We present Automated Configuration Testing (ACT) for quick IaC program testing at low effort and implement it for Pulumi TypeScript in ProTI. ProTI is effective on existing IaC programs, and its modular architecture enables the use of existing third-party and novel test generators and oracles, breaking ground for future research on effective test generators and oracles for IaC programs.

References

[1] K. Morris, *Infrastructure as Code: Dynamic Systems for the Cloud Age*, 2nd ed. Sebastopol, CA, USA: O'Reilly Media, Inc., 2021.

[2] G. Kim, J. Humble, P. Debois, J. Willis, and N. Forsgren, *The DevOps hHandbook: How to Create World-class Agility, Rel., & Secur. Technol. Organizations*, 2nd ed. IT Revolution Press, 11 2021.

[3] M. Guerriero, M. Garriga, D. A. Tamburri, and F. Palomba, "Adoption, support, and challenges of Infrastructure-as-Code: Insights from industry," in *Proc. IEEE Int. Conf. Softw. Maintenance Evolution (ICSME)*, Cleveland, OH, USA. Piscataway, NJ, USA: IEEE Press, 2019, pp. 580–589, doi: 10.1109/ICSME.2019.00092.

[4] I. Kumara et al., "The do's and don'ts of infrastructure code: A systematic gray literature review," *Inf. Softw. Technol.*, vol. 137, 2021, Art. no. 106593. doi: 10.1016/J.INFSOF.2021.106593.

[5] L. A. F. Leite, C. Rocha, F. Kon, D. S. Milojicic, and P. Meirelles, "A survey of DevOps concepts and challenges," *ACM Comput. Surv.*, vol. 52, no. 6, pp. 127: 1–127:35, 2020, doi: 10.1145/3359981.

[6] A. Rahman, R. Mahdavi-Hezaveh, and L. A. Williams, "A systematic mapping study of infrastructure as code research," *Inf. Softw. Technol.*, vol. 108, pp. 65–77, 2019, doi: 10.1016/j.infsof.2018.12.004.

[7] C. Endres, U. Breitenbücher, M. Falkenthal, O. Kopp, F. Leymann, and J. Wettinger, "Declarative vs. imperative: Two modeling patterns for the automated deployment of applications," Accessed: Nov. 30, 2023. [Online]. Available: https://www.iaas.uni-stuttgart.de/publications/INPROC-2017-12-Declarative-vs-Imperative-Modeling-Patterns.pdf

[8] U. Breitenbücher, T. Binz, K. Képes, O. Kopp, F. Leymann, and J. Wettinger, "Combining declarative and imperative cloud application provisioning based on TOSCA," in *Proc. IEEE Int. Conf. Cloud Eng.*, Boston, MA, USA. Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 2014, pp. 87–96, doi: 10.1109/IC2E.2014.56.

[9] "Pulumi: Infrastructure as code in any programming language." Pulumi. Accessed: Nov. 29, 2023. [Online]. Available: https://github.com/pulumi/pulumi

[10] "Cloud development framework: AWS Cloud Development Kit." Amazon Web Services. Accessed: Nov. 29, 2023. [Online]. Available: https://aws.amazon.com/cdk/

[11] "CDK for Terraform." HashiCorp. Accessed: Nov. 29, 2023. [Online]. Available: https://developer.hashicorp.com/terraform/cdktf

[12] J. Duffy, "Pulumi raises Series B to build the future of cloud engineering." Pulumi Blog. Accessed: Nov. 30, 2023. [Online]. Available: https://www.pulumi.com/blog/series-b/

[13] J. Duffy, "Building the best infrastructure as code with $41M Series C funding." Pulumi Blog. Accessed: Nov. 30, 2023. [Online]. Available: https://www.pulumi.com/blog/series-c/

[14] M. Madeja, J. Porubän, S. Chodarev, M. Sulír, and F. GurbáĽ, "Empirical study of test case and test framework presence in public projects on GitHub," *Appl. Sci.*, vol. 11, no. 16, pp. 1–22, 2021, doi:10.3390/app11167250.

[15] P. Singh Kochhar, T. F. Bissyandé, D. Lo, and L. Jiang, "An empirical study of adoption of software testing in open source projects," in *Proc. 13th Int. Conf. Qual. Softw.*, Najing, China. Piscataway, NJ, USA: IEEE Press, 2013, pp. 103–112, doi: 10.1109/QSIC.2013.57.

[16] D. Sokolowski, D. Spielmann, and G. Salvaneschi, "The PIPr dataset of public infrastructure as code programs," in *Proc. 21st IEEE/ACM Int. Conf. Mining Softw. Repositories (MSR)*, Lisbon, Portugal, 2024, pp. 498–503, doi: 10.1145/3643991.3644888.

[17] H. Holmström Olsson, H. Allahyari, and J. Bosch, "Climbing the "stairway to heaven" - A mulitiple-case study exploring barriers in the transition from agile development towards continuous deployment of software," in *Proc. 38th Euromicro Conf. Softw. Eng. Adv. Appl. (SEAA)*, Cesme, Izmir, Turkey, V. Cortellessa, H. Muccini, and O. Demirörs, Eds., Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 2012, pp. 392–399, doi: 10.1109/SEAA.2012.54.

[18] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Reading, MA, USA: Addison-Wesley, 2010.

[19] P. Ralph et al., "Empirical standards for software engineering research," 2021, *arXiv:2010.03525*.

[20] D. Sokolowski, D. Spielmann, and G. Salvaneschi, "ProTI: Automated unit testing of Pulumi TypeScript infrastructure as code programs," 2023, doi: 10.5281/zenodo.10028479.

[21] D. Sokolowski, D. Spielmann, and G. Salvaneschi, "Evaluation of automated infrastructure as code program testing," 2024. doi: 10.5281/zenodo.10908273.

[22] "Cloud object storage: Amazon S3," Amazon Web Services. Accessed: Nov. 29, 2023. [Online]. Available: https://aws.amazon.com/s3/

[23] "Developer tools: SDKs and programming toolkits for building on AWS: SDKs," Amazon Web Services. Accessed: Nov. 29, 2023. [Online]. Available: https://aws.amazon.com/developer/tools/#SDKs

[24] "Azure SDK releases." Microsoft Azure. Accessed: Nov. 29, 2023. [Online]. Available: https://azure.github.io/azure-sdk/

[25] "Testing of Pulumi programs." Pulumi. Accessed: Nov. 29, 2023. [Online]. Available: https://www.pulumi.com/docs/using-pulumi/testing/

[26] "Policy as code for any cloud with Pulumi: Pulumi CrossGuard." Pulumi. Accessed: Nov. 29, 2023. [Online]. Available: https://www.pulumi.com/crossguard/

[27] "Integration testing for Pulumi programs." Pulumi. Accessed: Nov. 29, 2023. [Online]. Available: https://www.pulumi.com/docs/using-pulumi/testing/integration/

[28] G. Fink and M. Bishop, "Property-based testing: A new approach to testing for assurance," *ACM SIGSOFT Softw. Eng. Notes*, vol. 22, no. 4, pp. 74–80, 1997, doi: 10.1145/263244.263267.

[29] K. Claessen and J. Hughes, "A lightweight tool for random testing of Haskell programs," in *Proc. Fifth ACM SIGPLAN Int. Conf. Functional Program. (ICFP '00)*, Montreal, Canada, M. Odersky and P. Wadler, Eds., New York, NY, USA: ACM, 2000, pp. 268–279, doi: 10.1145/351240.351266.

[30] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler, "Fuzzing: Breaking things with random inputs." Fuzzing. Accessed: Nov. 30, 2023. [Online]. Available: https://www.fuzzingbook.org/html/Fuzzer.html

[31] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *Proc. 29th Int. Conf. Softw. Eng. (ICSE 2007)*, Minneapolis, MN, USA. Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 2007, pp. 75–84, doi: 10.1109/ICSE.2007.37.

[32] A. Löscher and K. Sagonas, "Targeted property-based testing," in *Proc. 26th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, Santa Barbara, CA, USA, T. Bultan and K. Sen, Eds., New York, NY, USA: ACM, 2017, pp. 46–56, doi: 10.1145/3092703.3092711.

[33] J. Li, B. Zhao, and C. Zhang, "Fuzzing: A survey," *Cybersecurity*, vol. 1, no. 1, 2018, Art. no. 6, doi: 10.1186/S42400-018-0002-Y.

[34] H. Goldstein, J. Hughes, L. Lampropoulos, and B. C. Pierce, "Do judge a test by its cover - combining combinatorial and property-based testing," in *Proc. Program. Lang. Syst. 30th Eur. Symp. Program. (ESOP)*, Luxembourg City, Luxembourg, N. Yoshida, Ed., vol. 12648, Cham, Switzerland: Springer-Verlag, 2021, pp. 264–291, doi: 10.1007/978-3-030-72019-3_10.

[35] A. Rahman, E. Farhana, C. Parnin, and L. A. Williams, "Gang of Eight: A defect taxonomy for infrastructure as code scripts," in *Proc. 42nd Int. Conf. Softw. Eng.,* Seoul, South Korea, G. Rothermel and D. Bae, Eds., New York, NY, USA: ACM, 2020, pp. 752–764, doi: 10.1145/3377811.3380409.

[36] T. Su et al., "Fully automated functional fuzzing of Android apps for detecting non-crashing logic bugs," in *Proc. ACM Program. Lang.,* vol. 5, no. OOPSLA, 2021, pp. 1–31, doi: 10.1145/3485533.

[37] "Jest: Delightful JavaScript testing." Meta Platforms. Accessed: Nov. 29, 2023. [Online]. Available: https://jestjs.io/

[38] N. Dubien, "fast-check official documentation." fast-check. Accessed: Nov. 29, 2023. [Online]. Available: https://fast-check.dev/

[39] "Serverless compute: AWS Fargate." Amazon Web Services. Accessed: Nov. 29, 2023. [Online]. Available: https://aws.amazon.com/fargate/

[40] "Amazon elastic container service." Amazon Web Services. Accessed: Nov. 29, 2023. [Online]. Available: https://aws.amazon.com/ecs/

[41] "Policies for AWS (AWSGuard)." Pulumi. Accessed: Nov. 29, 2023. [Online]. Available: https://www.pulumi.com/docs/using-pulumi/crossguard/awsguard/

[42] "Github docs: Searching code (legacy)." GitHub. Accessed: Nov. 30, 2023. [Online]. Available: https://docs.github.com/en/search-github/searching-on-github/searching-code

[43] A. Helin, "Radamsa: A general-purpose fuzzer." GitHub. Accessed: Nov. 30, 2023. [Online]. Available: https://gitlab.com/akihe/radamsa

[44] M. D. Ernst et al., "The Daikon system for dynamic detection of likely invariants," *Sci. Comput. Program.*, vol. 69, nos. 1–3, pp. 35–45, 2007, doi: 10.1016/j.scico.2007.01.015.

[45] L. Hoban, "Introducing AWS CDK on Pulumi." Pulumi Blogs. Accessed: Nov. 30, 2023. [Online]. Available: https://www.pulumi.com/blog/aws-cdk-on-pulumi/

[46] "Testing constructs: AWS Cloud Development Kit (AWS CDK) v2." Amazon Web Services. Accessed: Nov. 29, 2023. [Online]. Available: https://docs.aws.amazon.com/cdk/v2/guide/testing.html

[47] "Unit tests: CDK for terraform." HashiCorp. Accessed: Nov. 29, 2023. [Online]. Available: https://developer.hashicorp.com/terraform/cdktf/test/unit-tests

[48] D. Sokolowski, P. Weisenburger, and G. Salvaneschi, "Automating serverless deployments for DevOps organizations," in *Proc. 29th ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, Athens, Greece, D. Spinellis, G. Gousios, M. Chechik, and M. D. Penta, Eds., New York, NY, USA: ACM, 2021, pp. 57–69, doi: 10.1145/3468264.3468575.

[49] D. Sokolowski, P. Weisenburger, and G. Salvaneschi, "Decentralizing infrastructure as code," *IEEE Softw.*, vol. 40, no. 1, pp. 50–55, 2023, doi: 10.1109/MS.2022.3192968.

[50] W. Hummer, F. Rosenberg, F. Oliveira, and T. Eilam, "Testing idempotence for infrastructure as code," in *Proc. Middleware ACM/IFIP/USENIX 14th Int. Middleware Conf.*, Beijing, China, D. M. Eyers and K. Schwan, Eds., vol. 8275, Berlin, Heidelberg, Germany: Springer-Verlag, 2013, pp. 368–388, doi: 10.1007/978-3-642-45065-5_19.

[51] K. Ikeshita, F. Ishikawa, and S. Honiden, "Test suite reduction in idempotence testing of infrastructure as code," in *Proc. Tests Proofs - 11th Int. Conf.* Marburg, Germany, S. Gabmeyer and E. B. Johnsen, Eds., vol. 10375, Springer-Verlag, 2017, pp. 98–115, doi: 10.1007/978-3-319-61467-0_6.

[52] R. Shambaugh, A. Weiss, and A. Guha, "Rehearsal: A configuration verification tool for Puppet," in *Proc. 37th ACM SIGPLAN Conf. Program. Lang. Des. Implementation (PLDI)* Santa Barbara, CA, USA, C. Krintz and E. D. Berger, Eds., New York, NY, USA: ACM, 2016, pp. 416–430, doi: 10.1145/2908080.2908083.

[53] T. Sharma, M. Fragkoulis, and D. Spinellis, "Does your configuration code smell?" in *Proc. 13th Int. Conf. Mining Softw. Repositories (MSR),* Austin, TX, USA, M. Kim, R. Robbes, and C. Bird, Eds., New York, NY, USA: ACM, 2016, pp. 189–200, doi: 10.1145/2901739.2901761.

[54] J. Schwarz, A. Steffens, and H. Lichter, "Code smells in infrastructure as code," in *Proc. 11th Int. Conf. Qual. Inf. Commun. Technol. (QUATIC)*, Coimbra, Portugal, A. Bertolino, V. Amaral, P. Rupino, and M. Vieira, Eds., Los Alamitos, CA, USA: IEEE Comput. Soc., 2018, pp. 220–228, doi: 10.1109/QUATIC.2018.00040.

[55] A. Rahman and L. A. Williams, "Source code properties of defective infrastructure as code scripts," *Inf. Softw. Technol.*, vol. 112, pp. 148–163, 2019. doi: 10.1016/j.infsof.2019.04.013.

[56] A. Rahman, C. Parnin, and L. A. Williams, "The seven sins: Security smells in infrastructure as code scripts," in *Proc. IEEE / ACM* 41st Int. Conf. Softw. Eng. (ICSE), Montreal, QC, Canada, J. M. Atlee, T. Bultan, and J. Whittle, Eds., 2019, pp. 164–175, doi: 10.1109/ICSE.2019.00033.

[57] A. Rahman, M. R. Rahman, C. Parnin, and L. A. Williams, "Security smells in Ansible and Chef scripts: A replication study," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 1, pp. 3: 1–3:31, 2021, doi: 10.1145/3408897.

[58] N. Saavedra and J. F. Ferreira, "GLITCH: Automated polyglot security smell detection in infrastructure as code," in *Porc. 37th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Rochester, MI, USA. New York, NY, USA: ACM, 2022, pp. 47: 1–47:12, doi: 10.1145/3551349.3556945.

[59] S. Reis, R. Abreu, M. d'Amorim, and D. Fortunato, "Leveraging practitioners' feedback to improve a security linter," in *Proc. 37th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Rochester, MI, USA. New York, NY, USA: ACM, 2022, pp. 66: 1–66:12, doi: 10.1145/3551349.3560419.

[60] R. Opdebeeck, A. Zerouali, C. Velázquez-Rodríguez, and C. De Roover, "On the practice of semantic versioning for Ansible Galaxy roles: An empirical study and a change classification model," *J. Syst. Softw.*, vol. 182, 2021, Art. no. 111059, doi: 10.1016/j.jss.2021.111059.

[61] R. Opdebeeck, A. Zerouali, and C. De Roover, "Smelly variables in Ansible infrastructure code: Detection, prevalence, and lifetime," in *Proc. 19th IEEE/ACM Int. Conf. Mining Softw. Repositories (MSR)*, Pittsburgh, PA, USA. New York, NY, USA: ACM, 2022, pp. 61–72, doi: 10.1145/3524842.3527964.

[62] R. Opdebeeck, A. Zerouali, and C. De Roover, "Control and data flow in security smell detection for infrastructure as code: Is it worth the effort? " in *Proc. 20th IEEE/ACM Int. Conf. Mining Softw. Repositories (MSR),* Melbourne, Australia. Piscataway, NJ, USA: IEEE Press, 2023, pp. 534–545, doi: 10.1109/MSR59073.2023.00079.

[63] S. Dalla Palma, D. Di Nucci, F. Palomba, and D. A. Tamburri, "Within-project defect prediction of Infrastructure-as-Code using product and process metrics," *IEEE Trans. Softw. Eng.*, vol. 48, no. 6, pp. 2086–2104, Jun. 2022, doi: 10.1109/TSE.2021.3051492.

[64] S. Dalla Palma, D. Di Nucci, F. Palomba, and D. A. Tamburri, "Toward a catalog of software quality metrics for infrastructure code," *J. Syst. Softw.*, vol. 170, 2020, Art. no. 110726, doi: 10.1016/J.JSS.2020.110726.

[65] S. Dalla Palma, D. Di Nucci, and D. A. Tamburri, "AnsibleMetrics: A Python library for measuring Infrastructure-as-Code blueprints in Ansible," *SoftwareX*, vol. 12, 2020, Art. no. 100633, doi: 10.1016/J.SOFTX.2020.100633.

[66] M. M. Hassan and A. Rahman, "As code testing: Characterizing test quality in open source Ansible development," in *Proc. 15th IEEE Conf. Softw. Testing, Verification Validation (ICST)*, Valencia, Spain. Piscataway, NJ, USA: IEEE Press, 2022, pp. 208–219, doi: 10.1109/ICST53961.2022.00031.

[67] N. Borovits et al., "FindICI: Using machine learning to detect linguistic inconsistencies between code and natural language descriptions in Infrastructure-as-Code," *Empir. Softw. Eng.*, vol. 27, no. 7, 2022, Art. no. 178, doi: 10.1007/s10664-022-10215-5.

[68] M. Chiari, M. De Pascalis, and M. Pradella, "Static analysis of infrastructure as code: A survey," in *Proc. IEEE 19th Int. Conf. Softw. Archit. Companion (ICSA)*, Honolulu, HI, USA. Piscataway, NJ, USA: IEEE Press, 2022, pp. 218–225, doi: 10.1109/ICSA-C54293.2022.00049.

[69] "Topology and Orchestration Specification for Cloud Applications versin 1.0," OASIS. Accessed: Nov. 29, 2023. [Online]. Available: http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html

[70] J. Bellendorf and Z. Á. Mann, "Specification of cloud topologies and orchestration using TOSCA: A survey," *Computing*, vol. 102, no. 8, pp. 1793–1815, 2020, doi: 10.1007/S00607-019-00750-3.

[71] M. Wurster et al., "The essential deployment metamodel: A systematic review of deployment automation technologies," *SICS Softw.Intensive Cyber Phys. Syst.*, vol. 35, nos. 1–2, pp. 63–75, 2020, doi: 10.1007/S00450-019-00412-X.

[72] M. Wurster, U. Breitenbücher, L. Harzenetter, F. Leymann, J. Soldani, and V. Yussupov, "TOSCA Light: Bridging the gap between the TOSCA specification and production-ready deployment technologies," in *Proc. 10th Int. Conf. Cloud Comput. Services Sci., (CLOSER)*, Prague, Czech Republic, D. Ferguson, M. Helfert, and C. Pahl, Eds., Rijeka, Croatia: SciTech, 2020, pp. 216–226, doi: 10.5220/0009794302160226.

[73] J. Aldrich, C. Chambers, and D. Notkin, "ArchJava: Connecting software architecture to implementation," in *Proc. 24th Int. Conf. Softw. Eng. (ICSE)* , Orlando, Florida, USA, W. Tracz, M. Young, and J. Magee, Eds., New York, NY, USA: ACM, 2002, pp. 187–197, doi: 10.1145/581339.581365.

[74] I. Krüger, B. Demchak, and M. Menarini, "Dynamic service composition and deployment with OpenRichServices," in *Software Service and Application Engineering - Essays Dedicated to Bernd Krämer on the Occasion of His 65th Birthday*, M. Heisel, Ed., vol. 7365, Springer-Verlag, 2012, pp. 120–146, doi: 10.1007/978-3-642-30835-2_9.

[75] R. Terra and M. T. de Oliveira Valente, "A dependency constraint language to manage object-oriented software architectures," *Softw. Pract. Exp.*, vol. 39, no. 12, pp. 1073–1094, 2009, doi: 10.1002/SPE.931.

[76] P. Weisenburger, M. Köhler, and G. Salvaneschi, "Distributed system development with ScalaLoci," in *Proc. ACM Program. Lang.,* vol. 2, no. OOPSLA, pp. 129: 1–129:30, 2018, doi: 10.1145/3276499.

[77] G. Zakhour, P. Weisenburger, and G. Salvaneschi, "Type-safe dynamic placement with first-class placed values," in *Proc. ACM Program. Lang.,* vol. 7, no. OOPSLA2, Oct. 2023, doi: 10.1145/3622873.

[78] B. Cook, "Formal reasoning about the security of Amazon Web Services" in *Proc. Comput. Aided Verification 30th Int. Conf., (CAV),* Oxford, UK, H. Chockler and G. Weissenbacher, Eds., vol. 10981, Cham, Switzerland: Springer-Verlag, 2018, pp. 38–47, doi: 10.1007/978-3-319-96145-3_3.

[79] J. Backes et al., "One-click formal methods," *IEEE Softw.*, vol. 36, no. 6, pp. 61–65, Nov./Dec. 2019, doi: 10.1109/MS.2019.2930609.

[80] M. Bouchet et al., "Block public access: Trust safety verification of access control policies," in *Proc. 28th ACM Joint Eur. Softw. Eng. Conf. Symp. Foundations Softw. Eng. (ESEC/FSE '20)*, Virtual Event, USA, P. Devanbu, M. B. Cohen, and T. Zimmermann, Eds., New York, NY, USA: ACM, 2020, pp. 281–291, doi: 10.1145/3368089.3409728.

[81] J. Backes et al., "Reachability analysis for AWS-based networks," in *Proc. Comput. Aided Verification 31st Int. Conf. (CAV)* New York City, NY, USA, I. Dillig and S. Tasiran, Eds., vol. 11562, Cham, Switzerland: Springer-Verlag, 2019, pp. 231–241, doi: 10.1007/978-3-030-25543-5_14.

[82] J. Backes et al., "Semantic-based automated reasoning for AWS access policies using SMT," in *Proc. Formal Methods Comput. Aided Des. (FMCAD)*, Austin, TX, USA, N. S. Bjørner and A. Gurfinkel, Eds., Piscataway, NJ, USA: IEEE Press, 2018, pp. 1–9, doi: 10.23919/FMCAD.2018.8602994.

[83] D. Jackson, "Alloy: A lightweight object modelling notation," *ACM Trans. Softw. Eng. Methodol.*, vol. 11, no. 2, pp. 256–290, 2002, doi: 10.1145/505145.505149.

[84] E. Ahrens, M. Bozga, R. Iosif, and J. Katoen, "Reasoning about distributed reconfigurable systems," in *Proc. ACM Program. Lang.,* vol. 6, no. OOPSLA2, pp. 145–174, 2022, doi: 10.1145/3563293.

[85] A. Evangelidis, D. Parker, and R. Bahsoon, "Performance modelling and verification of cloud-based auto-scaling policies," *Future Gener. Comput. Syst.*, vol. 87, pp. 629–638, Oct. 2018, doi: 10.1016/j.future.2017.12.047.

[86] A. Abu Jabal et al., "Methods and tools for policy analysis," *ACM Comput. Surv.*, vol. 51, no. 6, pp. 121: 1–121:35, 2019. doi: 10.1145/3295749.

[87] G. Zakhour, P. Weisenburger, and G. Salvaneschi, "Type-checking CRDT convergence," vol. 7, no. PLDI, 2023, pp. 1365–1388, doi: 10.1145/3591276.

[88] D. Spadini, M. F. Aniche, M. Bruntink, and A. Bacchelli, "To mock or not to mock?: An empirical study on mocking practices," in *Proc. 14th Int. Conf. Mining Softw. Repositories (MSR),* Buenos Aires, Argentina, J. M. González-Barahona, A. Hindle, and L. Tan, Eds., Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 2017, pp. 402–412, doi: 10.1109/MSR.2017.61.

[89] K. Taneja, Y. Zhang, and T. Xie, "MODA: automated test generation for database applications via mock objects," in *Proc. 25th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Antwerp, Belgium, C. Pecheur, J. Andrews, and E. D. Nitto, Eds., New York, NY, USA: ACM, 2010, pp. 289–292, doi: 10.1145/1858996.1859053.

[90] F. Solms and L. Marshall, "Contract-based mocking for services-oriented development," in *Proc. Annu. Conf. South Afr. Inst. Comput. Sci. Inf. Technol. (SAICSIT),* Johannesburg, South Africa., F. F. Blauw, M. Coetzee, D. A. Coulter, E. M. Ehlers, W. S. Leung, C. Marnewick, and D. van der Haar, Eds., New York, NY, USA: ACM, 2016, pp. 40: 1–40:8, doi: 10.1145/2987491.2987534.

[91] D. Saff and M. D. Ernst, "Mock object creation for test factoring," in *Proc. ACM SIGPLAN-SIGSOFT Workshop Program Anal. Softw. Tools Eng. (PASTE'04)*, Washington, DC, USA, C. Flanagan and A. Zeller, Eds., New York, NY, USA: ACM, 2004, pp. 49–51, doi: 10.1145/996821.996838.

[92] S. Joshi and A. Orso, "SCARPE: A technique and tool for selective capture and replay of program executions," in *Proc. 23rd IEEE Int. Conf. Softw. Maintenance (ICSM 2007),* Paris, France. Los Alamitos, CA, USA: IEEE Comput. Soc., 2007, pp. 234–243, doi: 10.1109/ICSM.2007.4362636.

[93] M. Fazzini, A. Gorla, and A. Orso, "A framework for automated test mocking of mobile apps," in *Proc. 35th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Melbourne, Australia. Piscataway, NJ, USA: IEEE Press, 2020, pp. 1204–1208, doi: 10.1145/3324884.3418927.

[94] H. Zhu et al., "StubCoder: Automated generation and repair of stub code for mock objects," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 1, pp. 1–31, Aug. 2023, doi: 10.1145/3617171.

[95] V. Vikram, R. Padhye, and K. Sen, "Growing a test corpus with bonsai fuzzing," in *Proc. 43rd IEEE/ACM Int. Conf. Softw. Eng., ICSE 2021*, Madrid, Spain. Piscataway, NJ, USA: IEEE Press, 2021, pp. 723–735, doi: 10.1109/ICSE43902.2021.00072.

[96] D. Steinhöfel and A. Zeller, "Input invariants," in *Proc. 30th ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE)* Singapore, Singapore,A. Roychoudhury, C. Cadar, and M. Kim, Eds., New York, NY, USA: ACM, 2022, pp. 583–594, doi: 10.1145/3540250.3549139.

[97] L. Lampropoulos, D. Gallois-Wong, C. Hritcu, J. Hughes, B. C. Pierce, and L. Xia, "Beginner's Luck: A language for property-based generators," in *Proc. 44th ACM SIGPLAN Symp. Princ. Program. Lang. (POPL 2017)*, Paris, France, G. Castagna and A. D. Gordon, Eds., New York, NY, USA: ACM, 2017, pp. 114–129, doi: 10.1145/3009837.3009868.

[98] L. Lampropoulos, M. Hicks, and B. C. Pierce, "Coverage guided, property-based testing," in *Proc. ACM Program. Lang.,* vol. 3, no. OOPSLA, pp. 181: 1–181:29, 2019, doi: 10.1145/3360607.

[99] A. Löscher and K. Sagonas, "Automating targeted property-based testing," in *Proc. 11th IEEE Int. Conf. Softw. Testing, Verification Validation (ICST)*, Västerås, Sweden. Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 2018, pp. 70–80, doi: 10.1109/ICST.2018.00017.

[100] R. Kuhn, D. R. Wallace, and A. M. Gallo, "Software fault interactions and implications for software testing," *IEEE Trans. Softw. Eng.*, vol. 30, no. 6, pp. 418–421, 2004, doi: 10.1109/TSE.2004.24.

[101] R. Kuhn, Y. Lei, and R. Kacker, "Practical combinatorial testing: Beyond pairwise," *IT Prof.*, vol. 10, no. 3, pp. 19–23, May/Jun. 2008, doi: 10.1109/MITP.2008.54.

[102] Z. Paraskevopoulou, C. Hritcu, M. Dénes, L. Lampropoulos, and B. C. Pierce, "Foundational property-based testing," in *Interactive Theorem Proving - 6th Int. Conf. (ITP)*, Nanjing, China, C. Urban and X. Zhang, Eds., vol. 9236, Cham, Switzerland: Springer-Verlag, 2015, pp. 325–343, doi: 10.1007/978-3-319-22102-1_22.

[103] L. Lampropoulos, Z. Paraskevopoulou, and B. C. Pierce, "Generating good generators for inductive relations," in *Proc. ACM Program. Lang.,* vol. 2, no. POPL, pp. 45: 1–45:30, 2018, doi: 10.1145/3158133.

[104] E. De Angelis, F. Fioravanti, A. Palacios, A. Pettorossi, and M. Proietti, "Property-based test case generators for free," in *Tests Proofs 13th Int. Conf.,*Porto, Portugal, D. Beyer and C. Keller, Eds., vol. 11823, Cham, Switzerland: Springer-Verlag, 2019, pp. 186–206, doi: 10.1007/978-3-030-31157-5_12.