# Towards Reliable Infrastructure as Code

Daniel Sokolowski
*University of St. Gallen, Switzerland*
daniel.sokolowski@unisg.ch
0000-0003-2911-8304

Guido Salvaneschi
*University of St. Gallen, Switzerland*
guido.salvaneschi@unisg.ch
0000-0002-9324-8894

***Abstract*—Modern Infrastructure as Code (IaC) programs are increasingly complex and much closer to traditional software than to simple configuration scripts. Their reliability is crucial because their failure prevents the deployment of applications, and incorrect behavior can introduce malfunction and severe security issues. Yet, software engineering tools to develop reliable programs, such as testing and verification, are barely used in IaC. In fact, we observed that developers mainly rely on integration testing, a slow and expensive practice that can increase confidence in end-to-end functionality but is infeasible to systematically test IaC programs in various configurations—which is required to ensure robustness. On the other hand, fast testing techniques, such as unit testing, are cumbersome with IaC programs because, today, they require significant coding overhead while only providing limited confidence.**

**To solve this issue, we envision the automated testing tool ProTI, reducing the manual overhead and boosting confidence in the test results. ProTI embraces modern unit testing techniques to test IaC programs in many different configurations. Out of the box, ProTI is a fuzzer for Pulumi TypeScript IaC programs, randomly testing the program in many different configurations for termination, configuration correctness, and existing policy compliance. Then developers can add specifications to their program to guide random-based value generation, test additional properties, and add further mocking, making ProTI a property-based testing tool. Lastly, we aim at automatically verifying IaC-specific properties, e.g., access paths between resources.**

***Index Terms*—Infrastructure as Code, Cloud Engineering, Fuzzing, Property-based Testing, Verification**

## I. Introduction

The setup of modern cloud applications is increasingly complex. Modern applications often comprise many small components, e.g., serverless functions, microservices, smaller databases, and blob storage, moving inherent complexity from inside big monolithic apps to the composition of these smaller components. This complexity has to be tackled by modern Infrastructure as Code (IaC) solutions, which are used to automate the applications' deployment. In IaC, developers configure systems by implementing programs that are similar to traditional software and much more complex than configuration scripts. This trend is likely to continue. For instance, current research and visions treat IaC programs already as long-running processes—like services in traditional software—instead of their common limitation today to be one-off tasks, which get executed over and over, e.g., by a CI/CD pipeline [1]. This idea enables automated coordination across IaC deployments [2] and implementing—not only configuring—resource orchestrators in IaC programs [3], solving the problem that today, there is no holistic view on static deployment configurations and their dynamic behavior [4].

With the growing complexity of IaC programs, their reliability is an increasingly important concern. Best case, faulty IaC programs do not deploy an application. Worst case, the faulty IaC program deploys the application such that it works correctly, but the error causes an insecure setup with vulnerabilities. Colloquially, *reliable* systems "just work." This aligns with the definition of Meyer [5] that we use in this paper: Reliability is a more general term encompassing *correctness* and *robustness*. Correctness describes that the program performs tasks as specified. Robustness describes that the program reacts appropriately to abnormal conditions. IaC should satisfy both—work as intended, even in a changed environment—and the developers have to achieve it.

Previous research on the reliability of IaC is limited to configuration as code solutions like Ansible, Chef, and Puppet [6], [7], [8], [9]. Also, in practice, tool support for developing modern IaC programs is limited, and developers lack quick and thorough feedback on the reliability of their IaC programs. In this paper, we envision filling this gap.

Section II highlights the lack of developer tooling for reliable IaC programs. Section III outlines our vision of ProTI, a tool for the semi-automated testing of IaC programs, making modern fuzzing and property-based testing techniques accessible to IaC developers. Finally, Section IV adds automated verification to ProTI, and Section V concludes.

## II. Modern Infrastructur as Code Solutions and the Reliability Tooling Issue

The core abstraction of declarative IaC solutions is the directed, acyclic resource graph [10]. Each node is a resource, and arcs are dependencies between them. In recent, modern IaC solutions, i.e., Pulumi [11], Amazon Web Services (AWS) Cloud Development Kit (CDK) [12], and CDK for Terraform (CDKTF) [13], which are the focus of our research and this paper, such resource graphs are not defined in JSON, YAML, or similar DSLs, but as programs in (imperative) general-purpose programming languages, e.g., TypeScript, Python, or Java.

Listing 1 is an example IaC program written in Pulumi TypeScript. While this example is relatively trivial and tiny compared to realistic IaC programs, it suffices to illustrate our ideas in this paper. The program deploys a static website on AWS S3 by describing the resource graph in Figure 1.

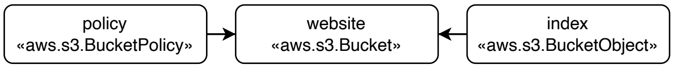

Fig. 1. Resource graph of a static website on AWS S3 described by Listing 1.

Listing 1
Pulumi TypeScript IaC program of a static website on AWS S3.

```
 1 import * as aws from "@pulumi/aws";
 2
 3 const bucket = new aws.s3.Bucket("website", {
 4     website: { indexDocument: "index.html" }
 5 });
 6 const index = new aws.s3.BucketObject("index", {
 7     bucket: bucket,
 8     key: "index.html",
 9     contentType: "text/html; charset=utf-8",
10     content: `<!DOCTYPE html>Hello World!`
11 });
12 new aws.s3.BucketPolicy("policy", {
13     bucket: bucket.bucket,
14     policy: bucket.arn.apply(bucketArn => ({
15         Version: "2012-10-17",
16         Statement: [{
17             Effect: "Allow",
18             Principal: "*",
19             Action: [ "s3:GetObject" ],
20             Resource: [ `${bucketArn}/*` ]
21 }]})) });
22
23 export const url = bucket.websiteEndpoint;
```

Object instantiation defines resources. Lines 3 to 5 define the S3 bucket, Lines 6 to 11 the static HTML document in it, and Lines 12 to 21 configure the access policy that grants public access from the Internet. Both the document and the policy reference the bucket which defines the dependency arcs.

A bug in Listing 1 may cause a crash such that some or all resources are not deployed and the website does not work. However, even without crashing, an error in the policy could cause the webpage is not accessible from the Internet. Vice versa, if the bucket would host sensitive data, e.g., invoices with customers' data, a faulty policy could permit public access, posing a severe data breach.

Testing and verification are traditional software engineering techniques to improve program reliability. To our knowledge, verification has not yet been applied to modern IaC programs. In contrast, various testing techniques are available, i.e., unit testing, dry running, integration testing, and end-to-end testing. Unit testing can provide fast feedback during development. The other techniques either rely on deploying the infrastructure in every test run—which is slow and expensive—or can only execute parts of the program. Yet, we observed that unit testing is barely used for IaC programs, and developers instead seem to rely on slower, more expensive integration testing.

We conjecture that today's unit testing for IaC programs is (a) too cumbersome and (b) generates too less confidence in the program's reliability. Unit testing is cumbersome because it requires mocking all side effects. And modern IaC programs are based on side effects, e.g., in Listing 1 all three resource objects would need to be mocked. By default, each mock either does not provide value because it does not perform any checks or value generation, or it is a significant coding effort, which is cumbersome and error-prone. Further, standard example-based testing is not thorough enough. It can only provide confidence that the program works correctly for a single configuration. Integration testing methods also test only a single configuration per test run. However, in this case, it is at least ensured that the tested example is realistic.

We argue that testing can only provide confidence regarding a program's robustness when it is sufficiently fast such that many different configurations are tested—ideally, covering all relevant cases. This is not possible with integration testing due to time and cost, but it is feasible for unit testing. Thus, it must become easy to unit test IaC programs, enabling automated random testing techniques (i.e., fuzzing and property-based testing), generating quick and thorough feedback, and therefore confidence in the reliability of IaC programs.

## III. Quick and Thorough IaC Unit Testing

To achieve quick and thorough IaC unit testing, we propose the design of ProTI. It will automate the tedious mocking of resources to simplify unit testing IaC programs. Further, it will combine state-of-the-art random testing techniques, specializing them for IaC programs. In summary, ProTI will be a fuzzer and a property-based testing tool for IaC programs.

### A. Fuzz Testing with ProTI

Fuzz testing (fuzzing) refers to techniques where whole programs are tested with many randomly generated input configurations to find bugs [14], [15]. The fewer resources a single test run requires, i.e., time and compute, the more test runs are possible. Therefore, fuzzing is only suitable for IaC program unit testing—not for integration testing, where a single test run takes at least tens of seconds and can realistically take up to multiple hours.

Fuzzing IaC programs with ProTI based on whole-program unit tests requires mocking, which is problematic (cf. Section II). However, based on the type definitions of the resource classes, ProTI can automate their mocking. Specifically, the constructors shall check whether the provided configuration values satisfy the expected format. Further, all outputs, i.e., the properties and functions of resources, can be replaced with random-based value generators for the respective type. For instance, in Listing 1, ProTI will automatically mock the policy definition in Lines 12 to 21, checking whether the provided configuration is valid in each test run. The accessed properties `bucket` and `arn` in Lines 13 and 14 return values from value generators of the properties' types.

With these features, ProTI can quickly run the IaC program many times without any adjustments or input from the developer. The initial implementation of ProTI will focus on Pulumi TypeScript. In each run, its random-based value generators will provide different values, hopefully covering many edge cases early and quickly finding errors regarding execution, termination, and resource configuration—if present. Value generation strategies will be at the core of our evaluation, which shall be performed on public open-source Pulumi TypeScript programs from GitHub. Besides naïve uniformly-distributed and biased random value generation, literature discussed feedback-directed [16], search-based [17], coverage-guided [14], and combinatorial-coverage-guided [18] strategies.

### B. *Property-based Testing with ProTI*

Property-based testing (PBT) [19], pioneered by QuickCheck [20], is another random testing technique. In contrast to fuzzing, the program is not treated as an opaque box, but developers leverage application domain knowledge to (a) guide the value generation and (b) check against application-specific properties specified as a predicate on the test outcomes. ProTI will seamlessly blend this idea into the above-presented fuzzer by providing developers functions, which they can use to (a) provide more specific value generators and (b) define custom properties to check during each test run.

ProTI will offer a `gen().with()` syntax to provide custom value generators. It can be applied to any property or function `X` by replacing it with `gen(X).with(Y)`, where `Y` defines a value generator, which is defined like arbitraries in the property-based testing framework fast-check [21]. E.g., in Listing 1, `bucket.arn` (Line 14) could be replaced with ``gen(bucket.arn).with(base64().map(s => `arn:aws:s3:::${s.replace(/[+/]/g, '-')}`))`` so that realistic ARN strings are generated instead of strings of any format during testing. In non-ProTI execution, e.g., production, the statement preserves the original semantics of `bucket.arn`.

Custom property checks will be added with widely-known `expect()` assertion syntax as available, e.g., in Jest [22], providing user-friendly short hands while supporting custom predicates to define whether an observed value is valid. E.g., `` `${bucketArn}/*` `` in Line 20 could be replaced with ``expect(`${bucketArn}/*`).toMatch(/^arn:aws:s3:::[a-zA-Z0-9-]+$/)`` to ensure all observed values are valid ARNs. To preserve the programs' semantics, the syntax will return the value provided as a parameter in `expect()` and ignore defined assertion in regular, non-ProTI execution.

As IaC programs may use all TypeScript features, including slow, expensive, or only-for-production code, ProTI will also provide widely-known mock functions syntax. Developers can use it to replace, e.g., to fasten up, parts of their code or check further properties. Functions can be mocked by using a `spy().with()` syntax, e.g., `fn = spy(fn).with(mock)`. Such mocks will be ignored in non-ProTI executions.

## IV. Automated Verification

Testing only shows the presence of errors. In contrast, verification can provide ultimate confidence that a program is reliable, but the required effort is often too high. Yet, automatically verifying certain domain-specific properties can be (a) technically feasible and (b) critical enough to justify the automation effort. We now outline how ProTI could be extended to provide IaC-specific property verification.

TABLE I
IaC shared responsibility model.

| **Responsibilities** | IaC Solution Developers | Cloud Resource Providers | Developers |
|---|---|---|---|
| **IaC Today** | IaC Solution | Resource Plugins | IaC Programs |
| **Automated Verification** | | Verification Models and Automation | Specifications to Verify |

To reduce the effort of verification, we will limit the scope to specific properties, making it amenable to full automation. We plan to leverage the shared responsibility model shown in Table I. The verification model and automation technique shall be part of the resource plugins, which are developed by the resource providers and the IaC solution developers and communities. Developers only add specifications to their IaC programs, which are automatically verified. This centralizes the tedious, resource-intensive task of developing the verification model and automation, making them usable with low effort for all customers of the resource providers.

Initial evidence that such a model can work is automated verification at AWS [23], [24]. AWS developed fully-automated verification services. AWS Tiros analyzes network configurations regarding reachability queries [25]. AWS Zelkova verifies access configured by AWS IAM role-based access policies [26]. Both services drive features in various services, including AWS Config [26], Amazon Inspector [25], and AWS S3 [26], [27]. However, these verification techniques are currently only used to verify already set-up cloud resources. We argue that IaC programs encode all required information for such verification. E.g., in Listing 1, an accessibility query could be added like `new proti.Accessibility('website-public', { source: proti.internet, target: index })` specifying and triggering verification that the index document is accessible from the Internet. Such constraints could be verified offline—before deployment—based on the target configuration state in IaC programs if the resource providers automate the verification similar to Tiros and Zelkova, e.g., using SMT solving with Z3 [28] and MonoSAT [29].

## V. Conclusion

Modern IaC programs have become similar to traditional software. This trend is driven by increasingly complex deployments and novel IaC solutions. However, developers lack support for writing reliable IaC programs. To fill this gap, we propose the automated unit-testing technique ProTI that will combine fuzzing, property-based testing, and automated verification. ProTI will provide quick *and* thorough feedback, enabling rapid development with frequent feedback cycles while providing high confidence in IaC programs' reliability.

## Acknowledgements

This work has been co-funded by the Swiss National Science Foundation (SNSF, No. 200429) and by the University of St. Gallen (GFF, No. 1025525 and No. 1025526).

# References

[1] D. Sokolowski, “Infrastructure as Code for Dynamic Deployments,” in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, A. Roychoudhury, C. Cadar, and M. Kim, Eds. ACM, 2022, pp. 1775–1779. [Online]. Available: https://doi.org/10.1145/3540250.3558912

[2] D. Sokolowski, P. Weisenburger, and G. Salvaneschi, “Automating Serverless Deployments for DevOps Organizations,” in *ESEC/FSE ’21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, D. Spinellis, G. Gousios, M. Chechik, and M. D. Penta, Eds. ACM, 2021, pp. 57–69. [Online]. Available: https://doi.org/10.1145/3468264.3468575

[3] D. Weerasiri, M. C. Barukh, B. Benatallah, Q. Z. Sheng, and R. Ranjan, “A Taxonomy and Survey of Cloud Resource Orchestration Techniques,” *ACM Comput. Surv.*, vol. 50, no. 2, pp. 26:1–26:41, 2017. [Online]. Available: https://doi.org/10.1145/3054177

[4] M. Guerriero, M. Garriga, D. A. Tamburri, and F. Palomba, “Adoption, Support, and Challenges of Infrastructure-as-Code: Insights from Industry,” in *2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019, Cleveland, OH, USA, September 29 - October 4, 2019*. IEEE, 2019, pp. 580–589. [Online]. Available: https://doi.org/10.1109/ICSME.2019.00092

[5] B. Meyer, *Object-Oriented Software Construction, 2nd Edition*. Prentice-Hall, 1997.

[6] S. D. Palma, D. D. Nucci, F. Palomba, and D. A. Tamburri, “Within-Project Defect Prediction of Infrastructure-as-Code Using Product and Process Metrics,” *IEEE Trans. Software Eng.*, vol. 48, no. 6, pp. 2086–2104, 2022. [Online]. Available: https://doi.org/10.1109/TSE.2021.3051492

[7] I. Kumara, M. Garriga, A. U. Romeu, D. D. Nucci, F. Palomba, D. A. Tamburri, and W. van den Heuvel, “The Do’s and Don’ts of Infrastructure Code: A Systematic Gray Literature Review,” *Inf. Softw. Technol.*, vol. 137, p. 106593, 2021. [Online]. Available: https://doi.org/10.1016/j.infsof.2021.106593

[8] R. Opdebeeck, A. Zerouali, and C. D. Roover, “Smelly Variables in Ansible Infrastructure Code: Detection, Prevalence, and Lifetime,” in *19th IEEE/ACM International Conference on Mining Software Repositories, MSR 2022, Pittsburgh, PA, USA, May 23-24, 2022*. ACM, 2022, pp. 61–72. [Online]. Available: https://doi.org/10.1145/3524842.3527964

[9] 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. [Online]. Available: https://doi.org/10.1145/3408897

[10] M. Wurster, U. Breitenbücher, M. Falkenthal, C. Krieger, F. Leymann, K. Saatkamp, and J. Soldani, “The Essential Deployment Metamodel: A Systematic Review of Deployment Automation Technologies,” *SICS Softw.-Intensive Cyber Phys. Syst.*, vol. 35, no. 1-2, pp. 63–75, 2020. [Online]. Available: https://doi.org/10.1007/s00450-019-00412-x

[11] Pulumi, “Pulumi: Universal Infrastructure as Code,” 2022, https://github.com/pulumi/pulumi (Accessed: 2022-07-12).

[12] Amazon Web Services, “AWS Cloud Development Kit,” 2022, https://aws.amazon.com/cdk/ (Accessed: 2022-07-12).

[13] HashiCorp, “CDK for Terraform,” 2022, https://www.terraform.io/cdktf (Accessed: 2022-07-12).

[14] J. Li, B. Zhao, and C. Zhang, “Fuzzing: A Survey,” *Cybersecur.*, vol. 1, no. 1, p. 6, 2018. [Online]. Available: https://doi.org/10.1186/s42400-018-0002-y

[15] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler, “Fuzzing: Breaking Things with Random Inputs,” in *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2022, retrieved 2022-07-25 12:07:43+02:00. [Online]. Available: https://www.fuzzingbook.org/html/Fuzzer.html

[16] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, “Feedback-directed random test generation,” in *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*. IEEE Computer Society, 2007, pp. 75–84. [Online]. Available: https://doi.org/10.1109/ICSE.2007.37

[17] A. Löscher and K. Sagonas, “Targeted Property-based Testing,” in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, T. Bultan and K. Sen, Eds. ACM, 2017, pp. 46–56. [Online]. Available: https://doi.org/10.1145/3092703.3092711

[18] H. Goldstein, J. Hughes, L. Lampropoulos, and B. C. Pierce, “Do Judge a Test by its Cover - Combining Combinatorial and Property-Based Testing,” in *Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings*, ser. Lecture Notes in Computer Science, N. Yoshida, Ed., vol. 12648. Springer, 2021, pp. 264–291. [Online]. Available: https://doi.org/10.1007/978-3-030-72019-3_10

[19] G. Fink and M. Bishop, “Property-Based Testing: A New Approach to Testing for Assurance,” *SIGSOFT Softw. Eng. Notes*, vol. 22, no. 4, p. 74–80, jul 1997. [Online]. Available: https://doi.org/10.1145/263244.263267

[20] K. Claessen and J. Hughes, “QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs,” in *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP ’00), Montreal, Canada, September 18-21, 2000*, M. Odersky and P. Wadler, Eds. ACM, 2000, pp. 268–279. [Online]. Available: https://doi.org/10.1145/351240.351266

[21] “fast-check: Property based testing for JavaScript and TypeScript,” 2023, https://dubzzz.github.io/fast-check.github.com/ (Accessed: 2023-01-26).

[22] Facebook, “Jest: Delightful JavaScript Testing,” 2023, https://jestjs.io/ (Accessed: 2023-01-26).

[23] B. Cook, “Formal Reasoning About the Security of Amazon Web Services,” in *Computer Aided Verification*, H. Chockler and G. Weissenbacher, Eds. Cham: Springer International Publishing, 2018, pp. 38–47. [Online]. Available: https://doi.org/10.1007/978-3-319-96145-3_3

[24] J. Backes, P. Bolignano, B. Cook, A. Gacek, K. S. Luckow, N. Rungta, M. Schaef, C. Schlesinger, R. Tanash, C. Varming, and M. Whalen, “One-Click Formal Methods,” *IEEE Software*, vol. 36, no. 6, pp. 61–65, 2019. [Online]. Available: https://doi.org/10.1109/MS.2019.2930609

[25] J. Backes, S. Bayless, B. Cook, C. Dodge, A. Gacek, A. J. Hu, T. Kahsai, B. Kocik, E. Kotelnikov, J. Kukovec, S. McLaughlin, J. Reed, N. Rungta, J. Sizemore, M. Stalzer, P. Srinivasan, P. Subotić, C. Varming, and B. Whaley, “Reachability Analysis for AWS-Based Networks,” in *Computer Aided Verification*, I. Dillig and S. Tasiran, Eds. Cham: Springer International Publishing, 2019, pp. 231–241. [Online]. Available: https://doi.org/10.1007/978-3-030-25543-5_14

[26] J. Backes, P. Bolignano, B. Cook, C. Dodge, A. Gacek, K. Luckow, N. Rungta, O. Tkachuk, and C. Varming, “Semantic-based Automated Reasoning for AWS Access Policies using SMT,” in *2018 Formal Methods in Computer Aided Design (FMCAD)*, 2018, pp. 1–9. [Online]. Available: https://doi.org/10.23919/FMCAD.2018.8602994

[27] M. Bouchet, B. Cook, B. Cutler, A. Druzkina, A. Gacek, L. Hadarean, R. Jhala, B. Marshall, D. Peebles, N. Rungta, C. Schlesinger, C. Stephens, C. Varming, and A. Warfield, “Block Public Access: Trust Safety Verification of Access Control Policies,” in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 281––291. [Online]. Available: https://doi.org/10.1145/3368089.3409728

[28] L. M. de Moura and N. S. Bjørner, “Z3: An Efficient SMT Solver,” in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, ser. Lecture Notes in Computer Science, C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, 2008, pp. 337–340. [Online]. Available: https://doi.org/10.1007/978-3-540-78800-3_24

[29] S. Bayless, N. Bayless, H. H. Hoos, and A. J. Hu, “SAT Modulo Monotonic Theories,” in *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*, B. Bonet and S. Koenig, Eds. AAAI Press, 2015, pp. 3702–3709. [Online]. Available: http://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/view/9951