# Local Expectation Testing for Terraform

Zach Gleason
University of Illinois, Urbana Champaign
Department of Computer Science
Urbana-Champaign, United States
zgleaso2@illinois.edu

Sreenidish Chinmayanilayam
University of Illinois, Urbana Champaign
Department of Computer Science
Urbana-Champaign, United States
sc49@illinois.edu

Yaroslav Buzko
University of Illinois, Urbana Champaign
Department of Computer Science
Urbana-Champaign, United States
ybuzko2@illinois.edu

## Abstract

With the rise of large, distributed systems, Infrastructure-as-code (IaC) has evolved from a best practice to a requirement in industry environments. HashiCorp's Terraform has become the go to IaC solution for many companies. In this paper, we explore potential solutions for testing Terraform locally using AWS-mocking tools LocalStack and Moto. Specifically, this paper proposes a framework for testing Terraform configurations that can (i) identify errors locally that were previously only detectable in the cloud, (ii) speed up test run time, and (iii) save costs. Our findings show the benefits of testing Terraform configurations locally but highlight the need for further research.

## I. Introduction

Cloud computing continues to be an area of rapid growth and evolution. Systems deployed in the cloud are becoming increasingly complex and interconnected, often supporting thousands of requests per second. In addition to these complexities, cloud-based solutions are expected to operate at nearly 100% uptime and auto-scale seamlessly. This puts Infrastructure as Code (IaC) into the spotlight. Managing these systems programmatically is critical, and declarative IaC tools like Terraform provide a solution. Terraform allows teams to break down their infrastructure stack into reusable code blocks called modules that can be individually tested. Still, testing Terraform configurations of large, industry systems remains a time and resource intensive task. Despite Terraform's built-in local unit and contract tests, further verification of the systems requires spinning up actual cloud resources, trial and error, and many minor code changes. Our team proposes a framework that mocks cloud APIs to enable rapid, local testing of Terraform configurations.

## II. State of the field

Testing is a key area of focus in IaC, and in the following section, we explore the most widely-used approaches to verify industry code.

HashiCorp provides a testing framework for Terraform. The framework guides developers through unit tests, contract tests, integration tests, and end-to-end tests to validate metadata values and built-in functions, input formatting, individual modules in the cloud, and entire cloud-based systems respectively.

Some cloud provider APIs expose a dry-run option that validates the potential resource request without fulfilling it. This is useful but not consistently available across cloud services. The dry-run option enables developers to explore how their created infrastructure will behave in a cloud environment prior to deployment.

Finally, there are local solutions like LocalStack and Moto that simulate a cloud provider locally. Both these solutions mock Amazon Web Services (AWS). Moto is built specifically to mock the AWS Python interface, boto, but also provides a standalone server mode. LocalStack, creates a local cloud provider instance in a Docker container and allows developers to explore how their resources would interact.

## III. Literature review

Our team performed a comprehensive review of existing research and literature on the subject of cloud IaC testing and Terraform in particular. For completeness, we included not only published papers but also articles on Medium and similar online resources. The review was conducted along the following broad themes:

- Current state of IaC
- General IaC testing
- Local simulation of cloud infrastructure
- Testing Terraform

### A. Current state of Infrastructure as Code

In *Adoption, Support, and Challenges of Infrastructure-as-Code: Insights from Industry* [1], Guerriero et al. conduct a set of interviews to state of the field, best practices, tooling, and challenges related to Infrastructure as Code. Despite the paper being as recent as 2019, the consensus among those interviewed is that testability of IaC is extremely limited. The authors concluded that the research on this subject is in early stages; there is opportunity to develop more frameworks and tooling to increase testability of IaC.

In *A systematic mapping study of infrastructure as code research* [2], Rahman et al. survey and categorize 30,000+ publications on IaC-related topics. The authors conclude that while framework/tooling for *developing* IaC is well-studied, there is need for more research related to defect identification.

### B. General IaC testing

In *Testing Idempotence for Infrastructure as Code* [3], the Hummer et al. endeavor to create an IaC middleware system that tests if a sequence of actions is idempotent. By extracting the state from Chef, they were able to construct a State Transition Graph (STG) and derive test cases that would identify idempotence during automation tasks. We believe that testing

for idempotence is an important step in IaC, but our focus is testing valid Terraform code prior to integration tests.

In *Code Smells in Infrastructure as Code* [4], Schwartz et al. adapted a way of determining code quality to Chef. By analyzing patterns from both object-oriented development and the paper *"Does your configuration code smell?"* [18], the authors were able to make two conclusions. First, Code Smell can adequately judge IaC code in general. Second, that there are still well-known programming smells that have yet to be adapted to IaC. In future work, they plan on adapting smells to Terraform, Salt, and other IaC frameworks. On the contrary, our paper focuses on testing validity. Code smell indicates poor quality in code, but it does not determine if the code will function.

*Infrastructure as Code: Towards Dynamic and Programmable IT systems* [5] by S.Naziris talks about Software Defined Infrastructure, Computing, Storage and Networking, but is not providing any deeper insights into how specifically the IaC can be tested either locally or remotely.

In *Smart-Cloud: A Framework for Cloud Native Applications Development* [6], the author develops a simplified cloud application development portal. He was able to develop a system that generates a UI, app logic, and service-to-service communication with a set of available libraries and APIs to expand upon the system. While helpful in code-less cloud development, the paper only handles testing applications created with Smart-Cloud. Additionally, Terraform isn't discussed. While the paper does provide helpful indications of the state of the field and an alternative development method, it doesn't dive into testing IaC.

In *Toward a Catalog of Software Quality Metrics for Infrastructure Code* [10], Dalla Palma et al. pinpointed 46 different properties inherent to Ansible code that increase the complexity of a blueprint. These properties tie directly to IaC structures, and, similar to *Code Smells in Infrastructure as Code* [4], these properties help evaluate IaC quality. While some of the properties defined in the paper can be applied directly to Terraform, they are limited to determining code quality and fail to test if code will run. Our paper expands upon the current state of the field by providing more comprehensive testing suites in Terraform.

In *Source Code Properties of Defective Infrastructure as Code Scripts* [11], Rahman and Williams define 12 common practices found in defective Puppet IaC commits. These practices range from lines of code to inclusion of URLs. With their sophisticated statistical techniques, they were able to show correlation between 10 of the properties and defective IaC scripts. This paper provides a method of applying stats in an attempt to determine if code is valid. Our paper looks to expand upon this by accurately determining if Terraform code is valid prior to the integration tests and identifying bugs.

In *Tortoise: interactive system configuration repair* [12], Weiss et al. define an approach to repairing system configurations called imperative configuration repair. This approach applies specifically to Puppet and infrastructure configuration. When an error surfaces, a user debugs the error and reconfigures the system in the correct manner. With this system, Tortoise will then apply the change to the Puppet manifest to prevent environment drift. Our paper will be focused on detecting errors prior to Terraform even running; however, developing a system similar to Tortoise to identify and correct drift in Terraform environments could provide a useful debugging method.

In *Continuous cloud infrastructure with Ansible, Molecule & TravisCI on AWS* [14], Hecht establishes Molecule as an important testing tool for infrastructure as code. Building on existing best practices for Docker and Vagrant, Hecht proposes an approach to test Ansible code for AWS using Molecule. This framework is potentially complementary to the solution we are proposing; Molecule can be configured to execute tests against "mock" AWS, or "real" AWS, or both, depending on testing requirements.

In *Test Suite Reduction in Idempotence Testing of Infrastructure as Code* [15], Ikeshita et al. proposed an approach to reduce the amount of test cases to generate the test cases needed for testing of idempotence that combined testing and static verification to generate test cases. This paper is informative and has relevance for what we proposed but focuses on approach/framework rather than actual tooling.

In *Asserting reliable convergence for configuration management scripts* [16], Hanappi et al. establish the importance of eventual convergence for cloud configuration frameworks with loose ordering. The paper proposes a framework, implemented on Puppet, to detect convergence and idempotence issues using state transition graphs.

In *Automated testing of chef automation scripts* [17], Hummer et al. explore issues of convergence and idempotence as related to Chef and propose a framework/tool for automated testing of Chef configurations. This paper also leverages state transition graphs. Notably, the paper calls out the categorical gap in testing functionality between full-on cloud-based tests and "dry runs", which is the area our paper is meant to address.

### C. Local simulation of cloud infrastructure

In *LocalStack - A fully functional local AWS cloud stack* [8], authors introduce LocalStack as a functional, easy-to-use test/mocking framework for developing Cloud applications. The framework relies on Docker and provides feature-rich local simulations of 30+ AWS cloud services. As mentioned in the motivation above, our paper is not aiming at replicating LocalStack, but instead on providing high-fidelity "stubs" to validate IaC code without spinning up any resources (even locally). This has potential to greatly improve testing speed and minimize resource usage.

In *What is LocalStack?* [21], the authors further establish the main use cases for local AWS simulation: local development and continuous integration.

In *LocalStack for Local AWS Dev* [9], N. Gibbon explores the tradeoffs between different options for testing DevOps code and microservices. The author then makes a case for "local mock" testing using LocalStack, which mitigates most of the inherent drawbacks of local testing. The author also provides demo/POC code. This paper is useful for overall understanding of testing approaches; however it is focused on one technology and does not discuss applicability for Terraform.

In *Testing AWS Python code with Moto* [22], Brandes explores the machinery under the covers of Moto and sets the basic ground rules of using a mock. In particular, the paper explores safeguards against incidental modification of live cloud infrastructure in cases when the "mock" credentials were not provided.

### D. Testing Terraform

In *Testing HashiCorp Terraform* [7], Wang proposes Hashicorp's own classification of IaC testing practices: unit tests, contract test, integration tests, and end-to-end tests. Like other papers attempting to systematize the testing field, Wang concludes that anything beyond contract tests (i.e. dry runs using Terraform's own validator) necessitate instantiation of cloud resources. Our paper aims to propose a testing layer that does not rely on Terraform's own validation but can still be run without incurring cloud provisioning expenses.

In *Terraform Up and Running* [13], Brikman describes general best practices in Terraform. These best practices range from getting started to testing Terraform Code. In the How to Test Terraform Code, the author describes the current state of automated and manual tests and specifically defines that there is no local environment for testing Terraform. This book defines exactly what we are trying to achieve and the novelty of our solutions while also providing helpful Terraform implementation details that we will use to create our test Docker image.

In *A Jump Start on Terraform Testing* [19], Smith describes different types of testing one can perform on Terraform code to ensure correctness, security and reliability. Like other papers surveying the spectrum of testing techniques, this paper highlights a gap between static tests and unit/integration tests that deploy resources.

## IV. Proposal

### A. Hypothesis

In our paper, we aim to create a new tier of local Terraform testing that we call expectation tests. We define expectation tests as a dry-run that verifies the behavior of cloud resources in concert with each other. In other words, while contract and unit testing ensure that individual resources are created with the correct input values, expectation testing provides a framework for developers to validate the functionality of their entire system by verifying that resources will be properly created together, validating dynamic entities, and catching AWS specific constraints. This is all done without requiring any extra test blocks.

By building upon LocalStack and Moto, we endeavor to show that expectation testing provides developers an accurate, local testing framework that saves time and money. We also show that expectation testing provides value by catching errors locally that previously only surfaced in cloud-based tests.

Expectation tests sit between contract and integration tests in the HashiCorp-provided testing hierarchy in Fig 1.

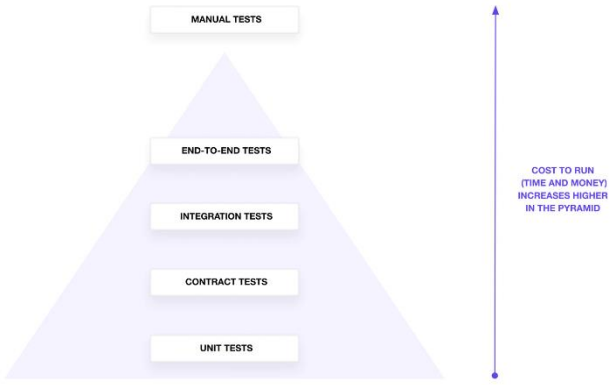


Fig.1. *Testing hierarchy according to HashiCorp.*

We use LocalStack and Moto to mimic the behavior (API responses) of the cloud provider (AWS) with sufficient accuracy to satisfy typical Terraform calls. Moto and LocalStack make up the backbone of our proposed solution, allowing us to test how resources will act in a cloud environment while testing only on local machines. We test if LocalStack and Moto speed up Terraform apply runs in our experiments to prove expectation tests provide developers a way to quickly and accurately test for free locally.

Not all errors can be caught using a mock provider. A trivial counter-example is spinning up an EC2 instance of a non-existent type. AWS EC2 would fail on such an attempt, while both LocalStack and Moto carry on and report success. We explore such situations in our paper in an attempt to both prove the usefulness of expectation testing and uncover its limitations.

### B. Scope

For the purposes of this project, we focus on analyzing AWS. Additionally, we narrow down the scope to several AWS services (EC2, SNS, S3, DynamoDB). We review the key APIs used by Terraform and explore only the core functionality. We allow Terraform to “create”, “describe”, “list” and “terminate” instances, but do not consider complex spot fleet-related scenarios or advanced network configurations. The underlying assumption is that if we prove expectation testing is viable and useful, it can be further generalized to other APIs, other AWS services, and potentially other cloud providers (e.g. Azure) in future work.

### C. Expected behavior and outcomes

We apply the following design principles to our implementation:

- If a test fails locally (i.e. using LocalStack or Moto), it should not pass remotely (i.e. in the cloud).

  **Reasoning:** under no circumstance we want the local solution to introduce new failures that do not occur in the cloud. That would force the developer to spend time debugging errors that only occur locally and this is counterproductive.

- If a test passes locally, it most likely passes remotely - with limitations (e.g. we ignore capacity availability and assume truly infinite resources).

  **Reasoning:** we strive to replicate cloud behavior as closely as possible but realize that 100% fidelity is not practically achievable. Even partial failure detection in a local test saves time and cost and is therefore beneficial. Any issues not detected during local testing will surface during the next testing phase (integration testing) that occurs in the cloud.

- Our proposed solution creates 0 cloud resources.

  **Reasoning:** the proposed approach is supposed to save costs (especially in the long run/at large scale). Tests run on a local, simulated cloud stack (either truly local on a developer's laptop, or dockerized as part of a CI/CD pipeline) can be run in air-gapped/firewalled conditions. This enables more flexible implementations such as instances with substantial security restrictions, e.g., financial services or federal.

## V. Proposed Solution

### A. Environment setup and code

To test our hypothesis, we developed a set of scripts that serves as a wrapper around the tests we intend to run. The scripts allow for timed, sequential execution of the same test(s) across multiple infrastructure providers. (Out of the box, we test across AWS, LocalStack and Moto, but other providers can be added.)

Installation and configuration instructions are provided in the "readme" section of the GitHub repository, available at https://github.com/Zman94/Local-Expectation-Testing-for-Terraform. In our framework, we created a template for each of our tests that mirrors the same Terraform code across Moto, LocalStack, and AWS environments.

Each mocked error has a separate "`.tool_versions`" file to control our environment's version of terraform using `asdf-vm` (asdf-vm.com), a symlinked `error.tf` and `fixed.tf` to consistently replicate the error and the error resolution across our three environments, and a script to run our experiments. The directory structure is shown in Fig. 2.

For each mocked error, the single `error.tf` contains that test's broken Terraform code. This allows us to execute the exact same test in 3 environments: AWS, LocalStack, and Moto. We can verify the consistency of the tests across the infrastructures and show the speed and cost improvements from using expectation testing. Each mocked error also has a similar `fixed.tf` HCL file containing the necessary adjustments to fix the error. This acts as a control to verify that our local tests provide results consistent with AWS. Thus, we are testing both "positive" and "negative" cases for each test case.

The `run_tests.sh` script first verifies the Terraform versioning is accurate before running our experiments. It then navigates to the target test's subfolders, in sequence. The script runs `terraform init` to initialize Terraform then times `terraform apply` and `terraform destroy` separately using the bash `time` command. This runs for each environment for both `error.tf` and `fixed.tf` and logs the results. The output of all runs is logged to verify whether we get consistent output from the three providers.

```
.
├ test_case_1
│  ├ error.tf ← "bad" code
│  ├ fixed.tf ← "fixed" code
│  ├ error
│  │  ├ aws
│  │  │  ├ error.tf ← symlink to ../../error.tf
│  │  │  └ provider.tf ← AWS defined as provider
│  │  ├ localstack
│  │  │  ├ error.tf ← symlink to ../../error.tf
│  │  │  └ provider.tf ← localstack defined as provider
│  │  └ moto
│  │     ├ error.tf ← symlink to ../../error.tf
│  │     └ provider.tf ← moto defined as provider
│  └ fixed
│  │  ├ aws
│  │  │  ├ fixed.tf ← symlink to ../../fixed.tf
│  │  │  └ provider.tf ← AWS defined as provider
│  │  ├ localstack
│  │  │  ├ fixed.tf ← symlink to ../../fixed.tf
│  │  │  └ provider.tf ← localstack defined as provider
│  │  └ moto
│  │     ├ fixed.tf ← symlink to ../../fixed.tf
│  │     └ provider.tf ← moto defined as provider
├ test_case_2
│  ├ ...
...
└ test_case_N
```

Fig.2. *Directory structure of the test framework.*

Outside of the mock error directories, we have a `run_all_tests.sh` script that iterates through all our test cases and individually logs the results. We also have the scripts `clean_tf_files.sh` and `format.sh`. These are helper scripts that are not run in the experiments.

Note: the set of tests included in the repository is not intended to cover every service or use case. These examples were selected to allow us to assess the feasibility of our proposed solution on a wide array of AWS services, as well as give developers an idea of how to apply the framework to their own IaC stack.

## VI. Experiments and Key Findings

### A. Data collection framework

To ensure accurate results, we ran our tests on three separate machines at varying times of the week to account for possible local machine or AWS delays. We also ran our tests five times on each machine to isolate outliers. Across our tests, we recorded if the LocalStack and Moto results were consistent with AWS, the times in seconds to run `terraform apply` on failure and success tests and the AWS cost per minute.

We curated a set of 10 errors across the AWS services EC2, SNS, S3, and DynamoDB. We chose our errors by searching for common errors related to each service in GitHub issues. We took that error, replicated it in an isolated environment, and implemented the proposed solution. All our tests are based on errors found in live infrastructure stacks.

We compiled the output of all the runs into a set of tables. Table 1 illustrates the fidelity of the tests across stacks and across cloud services. In each cell, we record a check if Moto or LocalStack has the same result as AWS and an "x" if they do not. That is, in the failure case, we expect Moto, LocalStack and AWS to throw an error, and, in the success case, we expect Moto, LocalStack, and AWS to run successfully.

| Test case | Failure | | Success | |
|---|---|---|---|---|
| | LocalStack output consistent with AWS? | Moto output consistent with AWS? | LocalStack output consistent with AWS? | Moto output consistent with AWS? |
| EC2: create instance of bad type | ✗ | ✗ | ✓ | ✓ |
| EC2: create instance with bad AMI | ✗ | ✗ | ✓ | ✓ |
| EC2: create instance with no AMI at all | ✓ | ✓ | ✓ | ✓ |
| EC2: bad local exec code | ✓ | ✓ | ✓ | ✓ |
| SNS: create with bad IAM policy | ✓ | ✓ | ✓ | ✓ |
| SNS: create with bad delivery policy | ✓ | ✓ | ✓ | ✓ |
| S3: create bucket with bad ACL | ✓ | ✓ | ✓ | ✓ |
| DynamoDB: insert rows into nonexistent table | ✓ | ✓ | ✓ | ✓ |
| DynamoDB: create table with hyphen in name | ✓ | ✗ | ✓ | ✓ |
| EC2: bad index in local exec | ✓ | ✓ | ✓ | ✓ |

Table 1. *Analysis of outcome fidelity for positive and negative test executions across AWS, LocalStack, and Moto.*

There are three test cases that are inconsistent across AWS and local testing tools:

- Test 1, "EC2: create instance of bad type", where we create an AWS instance with a non-existent instance type "superhuge". While the run with AWS as the cloud target failed, both LocalStack and Moto executions passed.
- Test 2, "EC2: create instance with bad AMI", where we create an AWS instance type with a non-existent AMI. Similar to test1, it failed on AWS but passed on both LocalStack and Moto.
- Test 9, "DynamoDB: create table with hyphen in name", where we create a DynamoDB with a hyphen in its name. As indicated in AWS docs, DynamoDB table names cannot have hyphens ('-') in them. The failure results are consistent across AWS and LocalStack but Moto fails to throw an error.

The root cause of tests 1 and 2 can be attributed to the inability for LocalStack and Moto to validate the presence or absence of a given instance type or AMI. These lists are updated frequently by the cloud provider without notice, so it is not practical for the local simulation to keep up.

Finally, the inconsistency related to honoring DynamoDB table naming conventions seems to be a legitimate bug in Moto that needs to be fixed. There is a GitHub issue explaining that Moto does not perform any validation of the table name before using it.

Across all our tests, there were no occurrences of Moto or LocalStack introducing errors that would not be observed in the cloud. When we fixed the errors in each test case, our framework executed exactly as AWS did.

We also captured the timing of all tests across multiple runs on multiple machines. Table 2 shows summarized results. We created a heatmap overlay to highlight the most time-consuming operations. Since columns 1 and 4, representing AWS runs, are by far the slowest in most of the tests, we can clearly observe time savings from running the tests locally.

| Row Labels | Average of Failure AWS timing, s | Average of Failure LocalStack timing, s | Average of Failure Moto timing, s | Average of Success AWS timing, s | Average of Success LocalStack timing, s | Average of Success Moto timing, s |
|---|---|---|---|---|---|---|
| EC2: create instance of bad type | 9.29 | 7.27 | 11.96 | 40.51 | 12.16 | 11.96 |
| EC2: create instance with bad AMI | 12.19 | 9.56 | 10.06 | 50.66 | 9.56 | 9.31 |
| EC2: create instance with no AMI at all | 2.34 | 4.32 | 4.26 | 53.53 | 12.14 | 11.96 |
| EC2: bad local exec code | 7.91 | 9.10 | 8.92 | 9.03 | 12.30 | 12.10 |
| SNS: create with bad IAM policy | 15.93 | 5.43 | 4.91 | 14.85 | 5.25 | 7.24 |
| SNS: create with bad delivery policy | 78.56 | 2.34 | 2.20 | 11.67 | 2.31 | 2.25 |
| S3: create bucket with bad ACL | 36.30 | 1.48 | 1.45 | 7.96 | 2.33 | 2.24 |
| DynamoDB: insert rows into nonexistent table | 13.99 | 2.33 | 2.21 | 19.57 | 2.74 | 2.55 |
| DynamoDB: create table with hyphen in name | 16.39 | 2.52 | 2.67 | 23.11 | 2.78 | 2.95 |
| EC2: bad index in local exec | 6.50 | 2.78 | 3.18 | 9.99 | 2.74 | 2.47 |

Table 2. *Average execution time of Terraform tests across AWS, LocalStack, and Moto.*

Chart 1 illustrates the execution times aggregated up to the service level and highlights the variability of execution times across test runs.

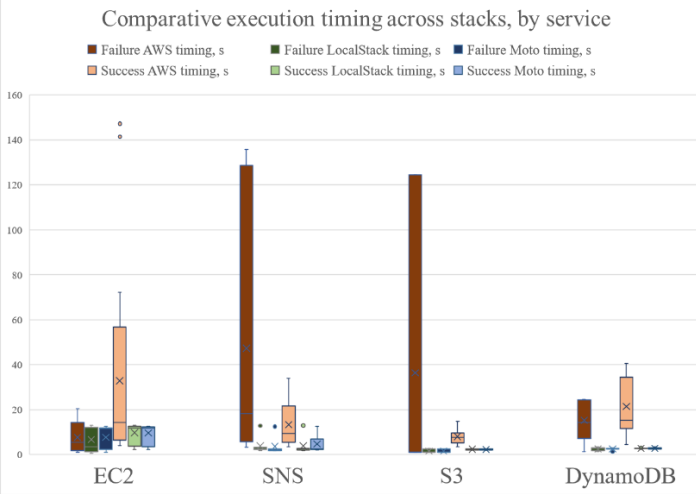


Chart 1. *Comparative execution timing across stacks, by service.*

Finally, we attempted to capture AWS cost for the operations carried out in the cloud to estimate potential cost savings from local testing. Our findings are provided in Table 3. Capturing cost of atomic operations with fine granularity is quite challenging on AWS, so these numbers are approximate.

| Test case | Failure AWS cost, $/m | Success AWS cost, $/m |
|---|---|---|
| EC2: create instance of bad type | $ - | $ 0.000193 |
| EC2: create instance with bad AMI | $ - | $ 0.000193 |
| EC2: create instance with no AMI at all | $ - | $ 0.000193 |
| EC2: bad local exec code | $ - | $ - |
| SNS: create with bad IAM policy | $ 0.000002 | $ 0.000002 |
| SNS: create with bad delivery policy | $ - | $ 0.000005 |
| S3: create bucket with bad ACL | $ - | $ - |
| DynamoDB: insert rows into nonexistent table | $ - | $ - |
| DynamoDB: create table with hyphen in name | $ - | $ 0.000006 |
| EC2: bad index in local exec | $ - | $ - |

Table 2. *Cost of running tests on AWS (failure and success).*

### B. *Analysis and key insights*

For our framework to be useful, LocalStack and Moto need to generally produce the same test outcomes as the cloud provider they simulate (AWS). The goal of our research is to provide a local testing option in Terraform that catches more errors than contract tests without introducing new errors not seen in integration tests. As seen in Table 1, the majority of our tests were consistent with AWS. The exceptions were analyzed and surmised to be the result of errors that could not be uncovered locally and should be discovered with integration tests.

This consistency is not uniform across AWS services. For example, in SNS all our tests behaved identically across stacks, while for EC2 certain tests were inconsistent. Importantly, all test runs conformed to our design principles, i.e. in no case did expectation testing produce an error that the cloud stack did not. Thus, in our selection of tests, we were able to confirm the first part of our hypothesis: **local stacks can replicate cloud stack behavior with sufficient fidelity**.

Next, it is important that our framework provides time benefits. As shown in Table 2 and Chart 1, the vast majority of tests recorded faster times locally than on AWS. The results varied across AWS service, network speeds, and local machine specs. On average, the DynamoDB tests performed comparatively the best locally. Interestingly, EC2 instance creation took longer to mock on LocalStack/Moto than to actually spin up in the cloud. This is an opportunity for further profiling and potential optimizations. However overall, because we were able to fail faster in 70+% of our tests, we confirmed the second part of our hypothesis: **tests on local stacks generally execute faster than in the cloud**.

Finally, our framework provides cost benefits. We have recorded the cost per minute of each test in AWS. All LocalStack and Moto tests are free because they create no resources in AWS. All AWS tests that error out before creating any cloud resources are also free. The cost per run for each test can be seen in Table 3. In our tests, we are running small recreations of the errors we found. In industry environments, it is not uncommon to have much larger infrastructure deployments, and errors can occur at any point during creation. AWS costs are typically much higher than our individual tests. Thus, despite modest cost values in our simulations, such cost savings will accumulate over time. This is especially true for large CI/CD deployments with continuous tests on a daily (or hourly) basis. Overall, we confirmed the third part of our hypothesis: **running expectation tests can save costs**.

## VII. Conclusion

With the rise of Infrastructure as Code and Terraform specifically, industry code bases are getting larger and more complex. The added complexities require equally complex and robust test suites. Developers need to quickly test their code and stake-holders need to have confidence that those tests are accurate.

In this paper, we explore a solution to local infrastructure testing that we call expectation testing. We use AWS API mocking tools Moto and LocalStack to create a local testing solution that requires no new test suites. Our findings reveal both the potential of such a system as well as the need to investigate further. In our paper, we documented our experiments, results, and insights into those results. Ultimately, our results are promising; we see time improvements of local testing across the majority of our test cases and the cost improvements across

every test case. Additionally, all of our results follow our initial requirements: any test that fails locally has to fail on AWS, but if a test passes locally, it can pass on AWS, too.

Though our results are encouraging, more testing is required. In future research, we expect to expand the number of errors we test across multiple services. We have shown that our framework is viable as a proof of concept, but to prove its full viability for production use, our framework needs to be tested on full, industry AWS infrastructure stacks. This will allow us to provide a more definitive recommendation regarding cloud services that are a strong fit for local expectation testing. We also plan to expand our research to other cloud providers. To be more useful, our tool should test Azure and Google Cloud IaC as well as AWS.

We hope that practitioners can use our framework as is to test their AWS IaC stacks. With our results, we believe users can achieve faster, more cost-effective tests. Users can then integrate our framework into their own CI/CD pipelines to fail faster, iterate faster, and to gain additional assurances that their code works as expected.

## REFERENCES


[1] M. Guerriero, M. Garriga, D. A. Tamburri, F. Palomba, *Adoption, Support, and Challenges of Infrastructure-as-Code: Insights from Industry*, 2019, ISBN 978-1-7281-3095-8

[2] A. Rahman, R. Mahdavi-Hezaveh, L. Williams, *A systematic mapping study of infrastructure as code research*, Information and Software Technology, vol. 108, April 2019, p. 65-77

[3] W. Hummer, F. Rosenberg, F. Oliveira, T. Eilam, *Testing Idempotence for Infrastructure as Code*

[4] J. Schwarz, A. Steffens, H. Lichter, *Code Smells in Infrastructure as Code*

[5] S. Naziris, *Infrastructure as Code: Towards Dynamic and Programmable IT systems*, http://purl.utwente.nl/essays/80190 , November, 2019

[6] J. H. A. Al Kiswani, *Smart-Cloud: A Framework for Cloud Native Applications Development*, http://hdl.handle.net/11714/5691, May, 2019

[7] R. Wang, *Testing HashiCorp Terraform*, https://www.hashicorp.com/blog/testing-hashicorp-terraform

[8] *LocalStack - A fully functional local AWS cloud stack*, https://github.com/localstack/localstack

[9] N. Gibbon, *LocalStack for Local AWS Dev*, https://medium.com/pareture/localstack-for-local-aws-dev-22775e483e3d

[10] S.Dalla Palma, D.Di Nuccia, F.Palomba, D. A. Tamburri, *Toward a catalog of software quality metrics for infrastructure code*, https://www.sciencedirect.com/science/article/pii/S0164121220301618

[11] A. Rahman, L. Williams, *Source code properties of defective infrastructure as code scripts*, arXiv:1810.09605

[12] A.Weiss, A.Guha, Y.Brun, *Tortoise: interactive system configuration repair*, ASE 2017: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, October 2017

[13] Y. Brikman, *Terraform Up and Running: Writing Infrastructure as Code Second Edition*, O'Reilly, ISBN 9781492046905

[14] J.Hecht, *Continuous cloud infrastructure with Ansible, Molecule & TravisCI on AWS*, https://blog.codecentric.de/en/2019/01/ansible-molecule-travisci-aws/

[15] Ikeshita, K., Ishikawa, F., Honiden, S., Gabmeyer, S., Johnsen, E., *Test Suite Reduction in Idempotence Testing of Infrastructure as Code*, In Proceedings of the International Conference on Tests and Proofs (TAP'17), 2017

[16] Oliver Hanappi, Waldemar Hummer, Schahram Dustdar, *Asserting reliable convergence for configuration management scripts*, SIGPLAN Not. 51, 10 (October 2016)

[17] W. Hummer, F. Rosenberg, F. Oliveira, T. Eilam, *Automated testing of chef automation scripts*, In Proceedings Demo & Poster Track of ACM/IFIP/USENIX International Middleware Conference (MiddlewareDPT '13). ACM, New York, NY

[18] T. Sharma, M. Fragkoulis, D. Spinellis, *Does your configuration code smell?*, in: Proceedings of the 13th International Conference on Mining Software Repositories, MSR '16, ACM, New York, NY, USA, 2016, pp.189–200. doi:10.1145/2901739.2901761.

[19] M. Smith, *A Jump Start on Terraform Testing*, https://medium.com/swlh/a-jump-start-on-terraform-testing-295522cdb407

[20] TerraTest by GruntWorks

[21] *What is LocalStack?*, https://localstack.cloud/

[22] K. Brandes, *Testing AWS Python code with Moto*, https://blog.codecentric.de/en/2020/01/testing-aws-python-code-with-moto/


## VIII. APPENDIX

Below is a list of conferences that we are either planning to submit to or have already submitted.

*A. SPLASH 2022 Mon 14 - Sat 19 November 2022 Auckland, New Zealand - CONFLANG Workshop, https://2021.splashcon.org/home/conflang-2021*

**Submission date:** Aug 6, 2021 (submitted)

**Format requirement:** 600 word abstract

We believe this conference is specifically tailored for our paper because it focuses primarily on configuration languages, such as Terraform, testing and verification which are the core findings of our paper. Note, the submission deadline was Friday the 6th. To make this deadline, we have already submitted our findings.

*B. SREcon22 Americas Planned but not accepting papers yet. March 14, 2022–March 16, 2022 San Francisco, CA, United States Hyatt Regency San Francisco, https://www.usenix.org/conference/srecon21*

**Submission date:** 2022 (date not specified yet)

We believe our paper advances the cause of the low cost and low latency development cycle for Infrastructure-As-Code (IAC), which will have a material impact on productivity of practitioners. Our paper focuses specifically on the needs of SREs. Terraform is a tool widely used among SREs, and our paper brings novel local testing improvements to Terraform and IaC in general.

*C. ACM Middleware 2022, https://middleware-conf.github.io/2022*

**Submission date:** 2022 (date not specified yet)

**Format requirement:** Your submission must be made within the due date specified above for the specific rounds. Submitted papers must have at most 12 pages of technical content, including text, figures, and appendices, but excluding any number of additional pages for bibliographic references. Note that submissions must be double-blind: authors' names must not appear, and authors must make a good faith attempt to anonymize their submissions. Submitted papers must adhere to the formatting instructions of the ACM SIGPLAN style, which

can be found on the ACM template page. The font size has to be set to 10pt.

We believe our paper advances the cause of the low cost and low latency development cycle for Infrastructure-As-Code(IAC), which will have a material impact on productivity of practitioners. Our paper falls under the conference's category "deployment of distributed systems."

*D. ICSME 2022 (North America): International Conference on Software Maintenance and Evolution, https://icsme2021.github.io/cfp/ImportantDates.html*

**Submission date:** 2022 (date not specified yet)

**Format requirement:** Papers must not exceed 4 pages, including all text, figures, and appendices + 1 extra page for references only. All submissions must be in PDF and submitted online by the deadline via the ICSME 2021 EasyChair conference management system; authors should choose "ICSME 2021 Tool Demonstrations Track Papers". Submissions must strictly adhere to the two-column IEEE conference proceedings format.

We believe our paper advances the cause of the low cost and low latency development cycle for Infrastructure-As-Code (IAC), which will have a material impact on productivity of practitioners. Additionally, our paper provides a novel improvement on IaC software maintenance, a key area of focus in this conference.

*E. IEEE Cloud Summit 2021, https://icsme2021.github.io/cfp/ImportantDates.html*

**Submission date:** Aug 10, 2021 (submitted)

**Format requirement:** Submitted papers must be original, unpublished work and not currently under review for any other conference or journal. Authors may submit full papers not longer than 6 pages (up to 8 pages with extra page charges) or extended abstracts for posters not longer than 4 pages. All papers/extended abstracts should follow the standard IEEE double-column template. Please submit papers/extended abstracts via the online submission system.

We believe our paper advances the cause of the low cost and low latency development cycle for Infrastructure-As-Code (IAC), which will have a material impact on productivity of practitioners. This conference focuses on all topics related to cloud computing. At cloud computing's core is infrastructure configuration. Our paper focuses on testing the creation of cloud resources, which the conference focuses on.