universität wien

MASTERARBEIT | MASTER'S THESIS

Titel | Title

Evaluating Infrastructure as Code Testing Practices in Open-Source GitHub Projects

verfasst von | submitted by

Elma Hamzabegovic BSc

angestrebter akademischer Grad | in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien | Vienna, 2026

Studienkennzahl lt. Studienblatt | Degree programme code as it appears on the student record sheet:	UA 066 926
Studienrichtung lt. Studienblatt | Degree programme as it appears on the student record sheet:	Masterstudium Wirtschaftsinformatik
Betreut von | Supervisor:	Univ.-Prof. Dr. Uwe Zdun

Abstract

Infrastructure as Code (IaC) is an approach to managing cloud infrastructure through version-controlled, machine-readable definition files rather than manual configuration. Infrastructure components such as servers, networks, and cloud services are expressed as declarative configurations that can be automatically provisioned, modified, and destroyed. Since infrastructure definitions are managed as code, they need to be tested for correctness and reliability, similar to traditional software systems. However, IaC testing differs from conventional software testing in several ways. Tests often require the provisioning of real cloud resources, which can make them slower to execute, more expensive, and dependent on external services. Additionally, the declarative nature of most IaC languages means that much of the behaviour cannot be meaningfully validated without actual deployment. While academic literature recommends testing IaC projects at a high level, there is limited empirical evidence on how testing is actually implemented in real-world IaC projects. This gap is also highlighted in several practitioner surveys, which report a discrepancy between the perceived value of testing and its actual adoption in practice. This thesis aims to address this gap through three contributions. First, an evaluation framework derived from academic literature is developed to systematically assess IaC testing maturity. Second, the framework is applied to twenty-five open-source GitHub repositories that implement IaC testing, producing detailed evaluations of each project. Third, cross-cutting findings are synthesised to identify areas of alignment and divergence between academic recommendations and observed practices. Based on these findings, the thesis derives specific practitioner recommendations outlining key testing priorities for improving testing practices in IaC projects.

Kurzfassung

Infrastructure as Code (IaC) ist ein Ansatz zur Verwaltung von Cloud-Infrastruktur mithilfe versionskontrollierter, maschinenlesbarer Definitionsdateien anstelle manueller Konfiguration. Infrastrukturkomponenten wie Server, Netzwerke und Cloud-Services werden dabei als deklarative Konfigurationen beschrieben, die automatisch bereitgestellt, geändert und entfernt werden können. Da Infrastrukturdefinitionen als Code verwaltet werden, müssen sie – ähnlich wie traditionelle Softwaresysteme – auf Korrektheit und Zuverlässigkeit getestet werden. Das Testen von IaC unterscheidet sich jedoch in mehreren Aspekten vom klassischen Softwaretesten. Tests erfordern häufig die Bereitstellung realer Cloud-Ressourcen, was sie zeitaufwändiger, kostenintensiver und abhängig von externen Diensten machen kann. Darüber hinaus bedeutet die deklarative Natur der meisten IaC-Sprachen, dass ein großer Teil des Systemverhaltens ohne tatsächliche Bereitstellung der Infrastruktur nur eingeschränkt überprüft werden kann. Obwohl die akademische Literatur das Testen von IaC-Projekten auf einer allgemeinen Ebene empfiehlt, gibt es bislang nur begrenzte empirische Erkenntnisse darüber, wie Testpraktiken in realen IaC-Projekten tatsächlich umgesetzt werden. Diese Lücke wird auch durch mehrere Praktikerbefragungen bestätigt, die eine Diskrepanz zwischen dem wahrgenommenen Stellenwert des Testens und dessen tatsächlicher Anwendung in der Praxis aufzeigen. Ziel dieser Arbeit ist es, diese Lücke durch drei Beiträge zu adressieren. Erstens wird ein auf der wissenschaftlichen Literatur basierendes Evaluierungsframework entwickelt, um die Testreife von IaC-Projekten systematisch zu bewerten. Zweitens wird dieses Framework auf fünfundzwanzig Open-Source-GitHub-Repositories angewendet, die IaC-Tests implementieren, wodurch detaillierte Evaluierungen der einzelnen Projekte entstehen. Drittens werden übergreifende Erkenntnisse zusammengeführt, um Bereiche der Übereinstimmung und Abweichung zwischen akademischen Empfehlungen und den beobachteten Praktiken zu identifizieren. Auf Grundlage dieser Ergebnisse werden spezifische praxisorientierte Empfehlungen abgeleitet, die zentrale Testprioritäten zur Verbesserung der Testpraktiken in IaC-Projekten aufzeigen.

Declaration on the Use of AI Tools

This thesis was authored by the student. During the preparation of this work, AI-based language models were used in the following capacities:

- **Language refinement and paraphrasing:** Improving the clarity, readability, and academic register of text drafted by the author, particularly for non-native English expression
- **Diagram and figure generation:** Generating TikZ and pgfplots code for the visual diagrams and charts included in this thesis
- **LaTeX formatting:** Assistance with table formatting, document structure, cross-referencing, and typesetting
- **Iterative drafting support:** Refining section structure, improving argumentation flow, and condensing content for conciseness

All literature reviews, repository analyses, and research conclusions are the original intellectual work of the author. The evaluation framework, repository selection and evaluation, and interpretation of findings were performed by the author, with AI tools used as a writing and formatting aid rather than as a source of analytical judgment. The author takes full responsibility for the content and accuracy of this thesis.

Contents

List of Tables

List of Figures

1. Introduction

1.1. Problem Definition

Infrastructure as Code (IaC) has become a widely adopted approach for managing cloud infrastructure in modern software development. By treating infrastructure specifications as version-controlled code, organisations achieve reproducibility, scalability, and automation of infrastructure provisioning. Despite these advantages, ensuring the reliability and correctness of IaC deployments remains difficult in practice.

Empirical evidence suggests systematic testing practices are not consistently applied in practice. Guerriero et al. conducted interviews with 44 IaC practitioners and found that while 90% rated testing frameworks as highly desirable and 82% rated static analysis tools as highly desirable, actual adoption of linters reached only 9% (Guerriero et al., 2019). This mismatch between perceived value and actual adoption illustrates a gap between literature recommendations and their implementation in practice.

Insufficient testing of IaC configurations can result in significant problems. Rahman et al. report that security vulnerabilities are common in IaC scripts, including misconfigured access controls and weak cryptographic configurations. When such scripts are deployed, these weaknesses can propagate to infrastructure environments at scale and become exploitable by attackers (Rahman et al., 2019b). This is evidenced in real-world incidents such as the Capital One data breach, where a misconfigured cloud resource allowed an attacker to exfiltrate sensitive data from over 100 million individuals (Khan et al., 2022).

Even though the academic literature has established testing recommendations including static analysis, unit testing, integration testing, and end-to-end validation (Morris, 2021), the extent to which practitioners actually implement these recommendations remains quite limited. Additionally, there is limited empirical evidence about testing practices in real-world IaC projects. While practitioner surveys offer insights into perceived challenges (Guerriero et al., 2019), they do not reveal actual implemented testing strategies. Existing repository mining studies focus primarily on security smells in IaC scripts (Rahman et al., 2019b), but lack systematic evaluation of testing practices across the full spectrum of validation techniques.

This gap between theoretical recommendations and observed real-world practices creates several problems:

1. Without empirical data on current testing practices, researchers have no way of knowing whether proposed testing frameworks address real practitioner needs. The reported gap between perceived usefulness and adoption suggests potential misalignment between existing solutions and development workflows (Guerriero et al., 2019).

2. Practitioners lack concrete examples and maturity benchmarks for evaluating their own testing strategies. Much of the existing guidance is conceptual (Morris, 2021), without evidence-based maturity models grounded in actual repository analysis.

3. There is currently no widely accepted framework for systematically assessing IaC testing maturity that connects academic recommendations with practices from the industry. Existing studies often focus on individual testing techniques in isolation (Chiari et al., 2022; Hummer et al., 2013) or provide high-level principles (Morris, 2021) without clearly defined evaluation criteria.

This thesis aims to address these challenges by developing an evaluation framework grounded in literature, conducting a systematic empirical analysis of IaC testing practices in open-source Git repositories, and identifying concrete gaps between academic recommendations and implemented practices.

1.2. Goals and Contributions

The goal of this thesis is to investigate how IaC testing is applied in practice and to what extent academic recommendations are reflected in real-world IaC projects. Based on these findings, the thesis aims to bridge the gap between recommended testing strategies and their actual adoption by proposing a set of practical guidelines for testing IaC projects.

The thesis makes the following contributions:

1. An **eight-dimension evaluation framework** derived from academic literature for systematically assessing IaC testing maturity across repositories

2. **Twenty-five detailed repository evaluations** of open-source IaC repositories spanning twelve project types (such as Terraform modules, Ansible collections, and IaC orchestration tools), four ecosystems (Terraform, Pulumi, Ansible, and CloudFormation), and three cloud providers (AWS, GCP, and Azure)

3. **Practitioner recommendations** derived from a cross-repository synthesis across eight thematic clusters, identifying the highest-impact testing investments for different IaC project types and cross-cutting practices applicable across the full corpus

To produce these contributions, the thesis pursues three objectives:

1. Conduct a literature review of existing academic research on IaC testing to identify recommended testing practices and derive an evaluation framework.

2. Apply the framework to open-source IaC repositories to assess how testing is implemented in practice.

3. Synthesise cross-cutting findings and derive practitioner recommendations grounded in both the literature and empirical observations.

1.3. Structure of the Thesis

Chapter 2 provides an introduction to the foundational concepts of Infrastructure as Code and outlines the relevant testing practices. The current state of research on IaC testing is examined in Chapter 3, where existing studies are analysed and classified. This review forms the basis for identifying the research gap and for deriving the evaluation framework used to assess the selected projects. The methodological approach adopted in this thesis, including the criteria used for repository selection, is described in Chapter 4. Chapter 5 provides a condensed overview of the twenty-five repository evaluations, highlighting the distinctive testing practices observed in each repository. The full detailed evaluations are provided in Appendix A. The cross-cutting findings and practitioner recommendations derived from this analysis are discussed in Chapter 6, together with the main limitations of the study and potential threats to validity. Finally, Chapter 7 summarises the key findings and outlines directions for future research.

2. Background

This chapter introduces Infrastructure as Code, the main tools and languages used in practice, general software testing concepts, and the specific challenges that arise when testing infrastructure code.

2.1. Infrastructure as Code

Infrastructure as Code is an approach to automating infrastructure using concepts and practices from software development (Morris, 2021). Instead of manually configuring infrastructure or using interactive configuration tools, IaC enables infrastructure to be provisioned and modified in a consistent and repeatable way through machine-readable definition files (Rahman et al., 2019a; Morris, 2021).

This approach represents a shift from traditionally reserving infrastructure knowledge for system administrators toward managing it as a shared, version-controlled asset accessible to the entire team (Artač et al., 2017). Morris (2021) describes this as part of a transition from static, hardware-centric infrastructure management to dynamic, cloud-based approaches enabled by virtualization technologies.

The main principles of IaC include:

- **Version control:** Infrastructure definitions are stored in source control systems (e.g., Git) to enable tracking of changes, collaboration, and rollback capabilities (Morris, 2021; Artač et al., 2017).
- **Reproducibility:** The same configuration can be applied multiple times to create identical environments, helping teams avoid the common problem where development, staging, and production systems gradually diverge over time (Morris, 2021).
- **Idempotence:** IaC scripts are expected to produce the same result regardless of how many times they are executed. Writing non-idempotent code is considered a bad practice, as it can lead to unpredictable system states (Guerriero et al., 2019).
- **Automation:** Infrastructure provisioning can be triggered automatically through CI/CD pipelines, enabling DevOps practices such as continuous testing and deployment (Artač et al., 2017; Morris, 2021).

2.2. IaC Tools and Languages

Existing studies indicate that the IaC ecosystem consists of a broad range of tools and languages rather than being centered around one dominant tool (Rahman et al., 2019a). Industry surveys reflect this diversity: most practitioners report using three or more tools in combination to address different infrastructure needs (Guerriero et al., 2019).

IaC tools can be divided into two categories based on how the infrastructure is specified:

Declarative IaC. In declarative tools, users specify what the infrastructure should look like, and the tool handles the steps to achieve that state. Practitioners often mention this as one of the main characteristics of working with IaC (Guerriero et al., 2019).

Popular declarative tools are:

- **Terraform** (HashiCorp): Manages infrastructure provisioning using a declarative approach (Nedeltcheva et al., 2023)
- **CloudFormation** (AWS): Amazon's native service for defining resources in JSON or YAML
- **Puppet** and **Chef**: Configuration management tools with dedicated domain-specific languages (Rahman et al., 2019a)

Imperative IaC. Some tools follow an imperative approach, allowing users to define infrastructure in general-purpose languages such as TypeScript, Python, or Go (Morris, 2021). This provides access to standard programming constructs like loops, conditionals, and reusable abstractions, that declarative languages lack. Some of the popular imperative IaC tools are:

- **Pulumi**: Supports TypeScript, Python, Go, or C#
- **AWS CDK** (Cloud Development Kit): Generates CloudFormation templates from general-purpose code

Hybrid IaC. Some tools combine both paradigms. **Ansible** (Red Hat) follows a hybrid declarative-imperative approach (modules declare desired state, but execution order can be controlled imperatively), making it well-suited for configuration management tasks (Nedeltcheva et al., 2023).

As discussed in Section 2.4, this distinction between declarative and imperative has direct implications for how IaC can be tested.

2.3. Software Testing Fundamentals

Software testing is the process of evaluating a system or component to verify that it satisfies specified requirements and to identify defects. In this thesis, testing practices are grouped into the following main categories:

- **Static analysis** analyses code without executing it and detects issues such as syntax errors, style violations, and potential security vulnerabilities. In the IaC context, static analysis tools can parse and detect more than just syntax errors without requiring connection to a cloud platform (Morris, 2021).
- **Unit testing** checks individual components in isolation, usually at the function or module level.
- **Integration testing** verifies that multiple components work correctly together.
- **End-to-end (E2E) testing** validates complete workflows by deploying real infrastructure, thereby verifying that it behaves as expected, often across multiple integrated systems.

The *testing pyramid* suggests that a well-structured test suite should have many fast unit tests at the base, followed by fewer integration tests in the middle, and a small number of slow E2E tests at the top (Morris, 2021). However, as discussed in the following section, this model applies differently to IaC.

Despite its importance, testing has received limited attention in IaC research, with only 4 of 32 publications in one systematic mapping study addressing testing topics (Rahman et al., 2019a).

2.4. Challenges of Testing Infrastructure Code

Testing infrastructure code presents unique challenges compared to traditional software testing. Research often identifies testing as one of the most significant pain points for IaC practitioners (Guerriero et al., 2019; Morris, 2021).

Low value of tests for declarative code. For declarative IaC, tests often provide limited value because they only restate the static configurations that the tool itself already validates (Morris, 2021). This creates a maintenance burden where every infrastructure code change requires a corresponding test update without providing meaningful validation. Unit tests become valuable only when infrastructure definitions include conditional logic or dynamically computed parameters. As a result, the traditional testing pyramid is less applicable to declarative infrastructure codebases, which may benefit from a diamond-shaped distribution with fewer low-level tests (Morris, 2021). In contrast to that, imperative IaC benefits more from traditional unit testing approaches due to its programmatic nature.

Slow feedback loops. Testing IaC generally involves deploying real infrastructure, which introduces delays due to resource provisioning. This time constraint is one of the main reasons why teams struggle to implement automated infrastructure testing (Morris, 2021). Dependencies on other infrastructure components add to the complexity of this problem, as shared test environments are often slow, expensive, or unreliable (Morris, 2021).

Lack of testing standards. Industry research identifies testability as one of the main challenges that practitioners face when working with IaC (Guerriero et al., 2019). The effort required to set up suitable testing environments, combined with the lack of established testing standards, makes it difficult for teams to implement quality assurance for their infrastructure code.

Debugging and tool heterogeneity. Debugging defects in IaC code is challenging due to the distributed nature of infrastructure deployments (Guerriero et al., 2019). Additionally, organisations often use multiple IaC tools with different syntax and testing capabilities, making it harder to enforce consistent testing practices across their full infrastructure codebase.

Security concerns. Security vulnerabilities in IaC can have serious consequences for both deployment and development environments (Rahman et al., 2019a). Despite these risks, security testing for IaC is still a relatively underresearched area and requires further attention.

These challenges mean that IaC testing practices differ from traditional software testing in several respects, and directly motivate the framework and repository analysis in the following chapters.

3. State of the Art

This chapter reviews the existing literature on Infrastructure as Code testing. It combines findings from academic literature together with Morris's book on infrastructure testing (Morris, 2021), to describe the recommended testing methods, their level of adoption, and the gaps in the area. The review covers the main categories of IaC testing: static analysis, unit testing, integration testing, and end-to-end testing. At the end of the chapter, a comparative discussion of all mentioned test types is presented and the research gaps that motivated the main contributions of this thesis are identified.

3.1. Static Analysis

3.1.1. Concept and Purpose

Static analysis is the first layer of IaC testing and runs before any code is executed or resources are deployed. It involves validating the syntax, structure, security, and compliance of infrastructure code without provisioning actual infrastructure. Because no resources are created, static analysis is fast and does not entail any cloud provider costs. Tools range from basic syntax checkers that catch typos and structural errors to more comprehensive linters that scan for security issues and enforce policy rules (Morris, 2021). Cankar et al. (2023) demonstrate a combined approach in which static analysis at design time is complemented by a machine learning-based runtime anomaly detection component, illustrating how IaC security tooling increasingly spans both phases of the DevSecOps lifecycle.

Morris (2021) differentiates between offline testing, which is performed entirely in a local environment, and online static analysis, which interacts with cloud provider APIs to validate configuration details such as instance types or image availability without actually provisioning resources.

3.1.2. Static Analysis in Practice

The importance of static analysis in IaC development is strongly supported with the industry examples. Guerriero et al. (2019) interviewed 44 practitioners in semi-structured interviews to evaluate their IaC development practices. When asked about the most desired features for IaC development, practitioners ranked static analysis tools as the second most desired feature (with a desirability score of 0.82 on a normalised scale, where 1.0 represents the highest possible score), right after testing frameworks (with a score of 0.90), suggesting that practitioners recognise the importance of automated code analysis as a way of identifying issues at an early stage of development.

There is, however, a significant gap between demand for static analysis and its use. According to Guerriero et al. (2019), only 4 of the 44 interviewees reported using static analysis tools, including `pylint` (Python linter) and `cfn-lint` (CloudFormation linter), as part of their standard development toolkit, which is an adoption rate of around 9%, despite 82% rating static analysis as highly desirable. Tool ecosystems also vary considerably across IaC technologies: Chiari et al. (2022) found that Puppet has the most developed ecosystem with six dedicated static analysis tools, followed by Ansible and TOSCA with four each, while Terraform, despite being widely adopted (Guerriero et al., 2019), has comparatively fewer dedicated tools.

3.1.3. Categories of Static Analysis Testing

In the literature, several categories of static analysis for IaC have been defined, with each of them addressing a different aspect of code quality.

Syntax Validation This is the simplest form of static analysis. It uses tools to check if the written code is syntactically correct, structurally valid and logical. Some examples are running `terraform validate` and `terraform fmt` to catch errors like missing brackets or undefined variables in Terraform scripts. According to Morris (2021), syntax checking is fast but can only identify a small number of defects.

Linting and Code Quality Analysis Linting is not limited to syntax checking and ensures that coding standards and best practices are followed. Linting tools, as described by Morris (2021), do not require infrastructure provisioning and can detect possible bugs, complexity problems, and style violations.

Hasan et al. (2020) point out that the prevention of anti-patterns in the code is a highly important testing practice, and linters are the main tool in the detection process. Some of the platform-specific linters are `ansible-lint` for Ansible, `tflint` for Terraform, and `Foodcritic` for Chef. These tools flag anti-patterns such as too long statements, missing default blocks in conditionals, and not following naming conventions.

30% of 50 internet artifacts analyzed by Hasan et al. (2020) report the use of linters to identify anti-patterns, which is much higher than the 9% among practitioners identified by Guerriero et al. (2019).

Security and Compliance Scanning The importance of static analysis has especially increased in response to high-profile infrastructure security incidents caused by IaC misconfigurations, such as publicly exposed cloud storage buckets and access policies that grant unnecessarily broad permissions. These incidents highlight the need to proactively check infrastructure configurations for security before deployment. Different studies have examined security-related weaknesses in IaC scripts, like Rahman et al. (2019b) who researched "security smells", i.e. recurring configuration patterns that may indicate potential security vulnerabilities. In practice, these issues are often addressed through static analysis tools such as `Checkov`, `Terrascan`, `TFSec`, which scan infrastructure code

prior to deployment in order to identify misconfigurations. Typical examples that can get flagged through these tools include publicly accessible storage resources, overly permissive IAM policies, or missing encryption for data stores.

The importance of security-focused static analysis is also reflected in Guerriero et al. (2019)'s findings that practitioners rated "tools and methods to support IaC security and privacy related issues" with a desirability score of 0.76, indicating very high demand. This represents what Morris (2021) describes as the "shift-left" approach to security, where security issues are identified and addressed early in the development cycle rather than discovered in production.

Policy-as-Code Validation Policy-as-code is a more recent development in the industry. Tools like HashiCorp Sentinel and Open Policy Agent (OPA) allow organisations to define infrastructure policies in code and validate that planned infrastructure changes comply with organisational standards. For example, an organisation might require that all S3 buckets have encryption enabled or that no security group allows inbound traffic from 0.0.0.0/0. This approach evaluates the output of planning commands (such as `terraform plan`) against custom-defined policies, enabling organisations to enforce governance requirements automatically (Joyce et al., 2025).

Static Code Analysis with API Some static analysis tools go beyond strictly offline checks by querying the cloud platform API to validate that resource configurations are compatible with what the provider actually supports (Morris, 2021). `TFLint`, for example, can verify that instance types and machine images referenced in Terraform code exist on the target platform. In contrast to previewing changes, which is discussed in the context of integration testing, it does not evaluate code against a specific stack instance, but it does require a live API connection to catch configuration errors that fully local tools would miss.

3.1.4. Integration into Development Workflows

Adoption of static analysis tools is strongly influenced by how well they are integrated into developers' existing workflows. Guerriero et al. (2019) found that, when asked about development tools, practitioners mentioned IDEs and text editors far more frequently than standalone static analysis tools, with 11 and 4 mentions respectively vs. 4 mentions for linters. This observation may suggest that static analysis is more likely to be adopted when it is embedded within the existing development environment rather than used as an independent tool.

At the same time, the perceived effectiveness of static analysis varies across roles and organisational contexts. Findings by Joyce et al. (2025) indicate that different stakeholders benefit from static analysis in different ways. While DevOps engineers reported a 60% reduction in runtime misconfigurations as a result of pre-deployment validation, security engineers observed an 85% decrease in secret exposure through automated scanning. This

difference implies that role-specific integration strategy, as opposed to a one-size-fits-all approach, is necessary for the effective usage of static analysis tools.

With regard to tooling, as illustrated in the tools discussed by Hasan et al. (2020), static analysis checks are often embedded within broader testing frameworks that support multiple validation steps rather than applied in isolation. This approach is also reflected in the most popular tools such as `Molecule` for Ansible, which integrates `ansible-lint` for syntax and style checks, as well as `Test Kitchen` for Chef and `Terratest` for Terraform, both of which include static analysis checks alongside other testing practices. These examples suggest a preference for unified testing environments that combine multiple validation activities within a single toolchain, rather than managing separate tools for individual validation steps, which is supported by findings from Guerriero et al. (2019), who report a high desirability score (0.74) for IaC-specific Integrated Development Environments (IDEs). Similar recommendations are made by Morris (2021), who argues that validation should be performed directly within developers' editors, for example via the Language Server Protocol (LSP).

3.1.5. Limitations of Current Tools

Although static analysis provides clear advantages in terms of efficiency and cost, the literature also points to some limitations. Chiari et al. (2022) distinguish between two broad categories of static analysis tools: syntactic approaches, such as linters and code-smell detectors, which operate on surface-level code features and cannot reason about runtime behaviour or issues that only arise during the actual provisioning of resources; and model-checking approaches, which can reason about deployment behaviour without requiring an actual deployment, but depend on sufficiently precise abstractions of the infrastructure. Similarly, Morris (2021) note that static analysis often performs only shallow validation. For example, a tool may confirm that code provisions a server without detecting that the referenced machine image does not exist or that the requested instance type exceeds available quotas.

Chiari et al. (2022) found several gaps in the current IaC tool landscape such as limited accuracy, no automated remediation (tools find problems but do not fix them), and using pattern matching instead of understanding the meaning of IaC code. This naturally also affects the feedback loop. Three practitioners in Guerriero et al. (2019)'s research noted that IaC has a longer feedback loop, since developers must deploy the entire infrastructure before they can determine whether their code is correct. The failure of static analysis to find bugs at the beginning will make developers deploy the infrastructure to ensure that it is correct, which defeats the purpose of static analysis in the first place.

War et al. (2025) quantified these concerns by evaluating traditional ML-based static analysis methods (TF-IDF and Bag of Words with Random Forest classifiers) for detecting security misconfigurations in Ansible and Puppet scripts. Precision was between 0.70 and 0.77, meaning that 23–30% of flagged issues were false positives. Recall was 0.73–0.84, indicating that the tools detected the majority of real misconfigurations. An ablation experiment from the same study further showed that when natural language context such as task descriptions and comments was removed from IaC scripts, precision dropped

from 0.92 to 0.46 for Ansible and from 0.87 to 0.55 for Puppet, suggesting that static analysis accuracy is highly sensitive to the availability of semantic context in the script. In practice, IaC scripts with sparse documentation may therefore produce noisier results and may limit the reliability of static analysis as a quality gate.

3.1.6. The Role of Static Analysis in the Testing Strategy

Despite its limitations, static analysis is an important testing step and can be seen as a first line of defense in testing IaC projects. As Morris (2021) emphasises in his discussion of progressive testing, faster tests with narrower scope should run first to provide quick feedback when issues are simple and localised. Static analysis fits this well because it runs in seconds, requires no infrastructure provisioning, and can catch a lot of errors before any deployment is attempted.

Guerriero et al. (2019)'s finding that practitioners want better static analysis tools (0.82 desirability) despite existing tool options suggests room for improvement. Nevertheless, as emphasised by both Morris (2021) and confirmed by Guerriero et al. (2019)'s findings on testing challenges, static analysis must be complemented by runtime testing to fully validate infrastructure code. The declarative nature of IaC, together with the complexity of cloud platforms and the importance of actual deployment behaviour after deployment, means that doing static analysis alone is not enough to ensure that the IaC code is correct. Therefore, static analysis should be viewed as the foundation of a comprehensive testing strategy rather than a complete solution.

3.2. Unit Testing

3.2.1. Concept and Purpose

Unit testing can be seen as the next layer of IaC testing after static analysis. It validates the logical behaviour and configuration semantics of individual units of IaC programs before they are integrated or deployed. While static analysis examines code structure and compliance, unit testing is used to verify that individual components such as modules, resources, or configuration functions behave as expected under different conditions (Morris, 2021).

For IaC projects, a unit can be regarded as the smallest independently testable component, such as a Terraform module, an Ansible role, or a Puppet class. Unlike traditional software units, these components express declarative configurations rather than executable logic, which makes them more difficult to test in isolation (Morris, 2021). Unit testing therefore tends to focus on the logic embedded in configuration code, such as conditionals, string interpolations, and parameter handling, rather than on static declarations, where tests often have limited value (Morris, 2021). Within a progressive testing strategy, unit tests are intended to run early and provide fast feedback, detecting errors before they reach slower, higher-fidelity integration or deployment stages (Morris, 2021).

3.2.2. The Testing Dilemma in IaC

Sokolowski and Salvaneschi (2023) describe the *testing dilemma* for IaC: integration testing provides realistic validation but is slow and costly, while unit testing provides faster feedback at the cost of high setup complexity. Unlike application code, IaC definitions interact with external, mutable cloud environments, which complicates the isolation of testable units and simulation of their dependencies. In their empirical investigation of over 12,000 public Pulumi repositories, Sokolowski et al. (2024) found that fewer than 1% of IaC projects implemented any form of testing — specifically, 118 out of 12,945 analysed projects had unit tests. This low adoption reflects structural obstacles: unit testing requires mocking all resource definitions, and effective mocks demand implementing validation logic that simulates complex cloud provider behaviour. Mocks must also be updated with every change to the infrastructure code. The authors call this a testing dilemma, meaning developers either invest a lot of effort in unit testing or resort to expensive integration testing, both of which slow down development speed.

Although unit testing is not extensively used in practice, empirical research from Drosos et al. (2024) shows that it may be far more valuable than its current adoption suggests. The authors analyzed 360 real-world defects across 45,000 artifacts from Ansible, Puppet, and Chef ecosystems and found that nearly 60 % of IaC bugs came from configuration units, i.e. exactly the modules that unit tests should verify. Most defects were small and localised and typically affected only a few lines of code (median of eight). Defects such as input-handling mistakes, state misinterpretations, and API inconsistencies are ideal candidates for detection through such lightweight module-level tests. Despite these findings, systematic unit testing is rarely implemented in industry, leaving many of these issues to be discovered only after deployment.

Morris (2021) warns that tests for purely declarative IaC often have limited value when they only restate static configurations already validated by the tool itself. Instead, unit tests gain significance when infrastructure definitions include conditional logic or dynamically computed parameters, where outputs may vary with different inputs.

Interviews by Guerriero et al. (2019) reinforce this point: practitioners in that study widely reported that deployment-time validation was the most reliable way to verify correctness, reflecting a tendency to treat real provisioning as the primary test. This suggests that the traditional testing-pyramid model does not transfer straightforwardly to IaC development (Morris, 2021).

3.2.3. Automated Configuration Testing (ACT) and ProTI

To reduce the high effort usually required for IaC unit testing, Sokolowski et al. (2024) introduced *Automated Configuration Testing* (ACT), a method for performing scalable unit testing of IaC programs without the need for developers to write the test code manually. ACT automatically mocks all resource definitions within an IaC program, uses a generator to produce different configuration inputs, and applies oracles to validate expected properties of each resource.

The authors implemented ACT as *ProTI* for Pulumi TypeScript. ProTI's pluggable

architecture allows external generators and oracles, supporting reuse across projects and experimentation with property-based or fuzzing approaches. In their evaluation, the authors demonstrate that ProTI can be applied to existing IaC programs without modification, identifies configuration defects within seconds and achieves reliability comparable to integration testing while scaling to hundreds of generated configurations. By treating unit testing as property-based configuration validation, ProTI emphasises invariant checking over example-based testing (e.g., "all storage buckets must enforce encryption" or "instance types must match region availability"), which is consistent with emerging trends in automated testing research.

3.2.4. The Role of Unit Testing in Continuous Testing Frameworks

Mascheroni et al. (2021) propose the *Continuous Testing Improvement Model* (CTIM) as a structured framework for gradually adopting continuous testing in organisations using Continuous Delivery or Deployment. The model defines four improvement levels, each targeting a specific set of quality problems. Level 1 (*Implementation*) establishes the foundational testing pipeline by introducing automated builds, automated unit test execution, and automated functional testing as mandatory CI stages. Level 2 (*Management*) adds test coverage measurement, multi-layered functional test frameworks, and mobile device testing. Level 3 (*Reliability*) addresses flaky tests through test doubles, headless functional testing, static code analysis, and deployment tests. Level 4 (*Continuity*) reduces test execution time through parallelisation, test selection, and continuous monitoring. Unit tests therefore form the first automated stage in this pipeline, providing immediate feedback on code correctness before integration. The authors also identify recurring pain points across levels, including flaky tests caused by non-deterministic mocks, time-consuming execution without efficient isolation. These observations align with the motivation behind ACT and ProTI, which aim to minimise manual effort and standardise test structure through automation.

3.2.5. Unit Testing in Practice

Integration of Unit Testing in Cloud-Native Pipelines Maliekal (2025) demonstrates how unit testing forms the first validation layer in automated Kubernetes pipelines. Each pull request triggers checks starting with unit tests that verify the correctness of individual code components before deployment. Tools such as `Terratest` and `Testkube` enable these tests to run in isolated namespaces, providing realistic yet safe execution environments. The study confirms that early-stage unit tests, combined with static and security checks, reduce deployment failures but also highlights maintainability challenges, as test suites require frequent updates and manual mocking remains effort-intensive.

Applied Example in Regulated Environments Kohler (2023) illustrates a cloud-based financial case where Terraform-managed infrastructure and AWS Lambda functions are jointly validated by automated unit tests before merge approval. In this CI/CD pipeline,

unit test success is a prerequisite for code integration, confirming early-stage automated testing as an essential quality gate even in highly regulated, IaC-driven systems.

Empirical Evidence from DevOps Pipelines Research by Nagineni (2025) identifies unit testing as the first automated quality gate in CI/CD pipelines. Mature teams usually keep execution times under five minutes and employ mutation testing to strengthen defect detection. In containerised environments, multi-stage Docker builds and Kubernetes-based parallelisation enable fast execution at scale, confirming the importance of early, automated unit testing in high-velocity DevOps practices.

3.2.6. Challenges and Research Gaps

Although unit testing offers clear advantages, its adoption in IaC projects remains limited. Prior studies point to several contributing factors:

- **Mocking complexity**: developers must replicate provider logic, leading to high implementation overhead. In some cases, the mocking logic may become more complex than the IaC program itself (Sokolowski et al., 2024).
- **Lack of standardised frameworks**: few open-source solutions exist for unit testing IaC programs, and even within ecosystems that provide testing support, adoption remains minimal — less than 1% of public Pulumi projects on GitHub implemented unit tests (Sokolowski et al., 2024). Tools such as `Molecule`, `Kitchen-Terraform`, and `InSpec` address specific aspects of unit or compliance testing but do not provide a unified, cross-platform framework.
- **Absence of common oracles**: there is no consensus on what constitutes a correct configuration, and implementing oracles that can reliably distinguish valid from invalid resource configurations requires significant effort and domain-specific knowledge (Sokolowski et al., 2024).
- **Flakiness and determinism**: imprecise mocks may generate unrealistic test inputs, producing false positives and reducing confidence in test results (Sokolowski et al., 2024).

3.3. Integration Testing

3.3.1. Concept and Purpose

Integration testing is generally one of the most commonly adopted forms of IaC testing, closing the gap between isolated code validation and full deployment. Unlike unit tests, which operate on mocked components, integration tests deploy actual infrastructure to verify that individual components interact correctly and that provisioned resources are configured as expected. This "deploy-verify-destroy" cycle, i.e. provisioning resources, validating behaviour and tearing down infrastructure, remains the foundation of practical

IaC validation (Hasan et al., 2020). Morris (2021) distinguishes between *verification testing*, which checks that deployed resources exist and are configured correctly, and *outcome testing*, which checks that the infrastructure behaves as expected, such as ensuring endpoint reachability or correct network routing. Integration tests therefore target both structural correctness and operational functionality, and are usually automated in CI/CD pipelines using frameworks like `Terratest`, `Test Kitchen`, and `InSpec` (Hasan et al., 2020).

3.3.2. Integration Testing in Practice

In the industry, integration testing is often integrated into CI/CD workflows as a routine quality gate. The most commonly adopted framework, `Terratest`, automates provisioning, validation, and cleanup in a single test sequence. Other frameworks like `Kitchen-Terraform`, `Serverspec`, `Awspec`, and `Molecule` apply this approach to different IaC tools, following a common practice of deployment, verification, and teardown. Thota (2024) highlight that integrating such tests into CI/CD workflows significantly reduces failed deployments and debugging time, particularly in multi-cloud and hybrid contexts, and position integration testing as part of a broader validation strategy combining static analysis, policy-as-code enforcement, and runtime monitoring.

Central to these practices is the principle of *test isolation*: ensuring that each test run operates in a clean, independent environment so that state left over from a previous run cannot affect subsequent results. Morris (2021) frames this as a prerequisite for reliable CI/CD integration, arguing that tests which share state or depend on execution order are inherently fragile. In IaC testing, isolation is typically achieved through ephemeral environments (temporary, isolated environments created for each test run), namespace or account separation, and explicit teardown hooks that execute even when tests fail. Maliekal (2025) illustrates this in practice: each pipeline execution provisions a temporary, isolated Kubernetes namespace, runs functional regression and performance tests, and removes all resources automatically upon completion, demonstrating how namespace isolation enables reproducible validation without affecting live workloads. Without such isolation, test suites become prone to flakiness, where failures reflect residual state rather than genuine defects.

Kohler (2023) provides a further example in a financial services context where integration tests validate Terraform-managed AWS Lambda deployments before code integration, serving as both a technical quality gate and a compliance safeguard.

3.3.3. Idempotence and State Evolution Testing

Idempotence testing verifies that applying a configuration multiple times yields the same result as applying it once, which is an essential requirement for infrastructure that must reliably converge to a desired state regardless of its starting point (Hummer et al., 2013). Hummer et al. (2013) introduced a model-based framework that constructs State Transition Graphs (STGs) to systematically generate test cases validating idempotence from various initial states, using lightweight Linux Containers for efficient execution. Applying their

framework to 298 Chef cookbooks drawn from publicly available repositories, they found that approximately 31% exhibited non-idempotent behaviour, highlighting how common idempotence violations are in practice, particularly when imperative commands are mixed with declarative code or when scripts depend on external state. Morris (2021) notes that modern declarative tools like Terraform and CloudFormation are designed to be idempotent by default, but recommends explicitly testing both fresh deployments and updates to existing infrastructure to verify that changes can be safely applied without unintended side effects.

Beyond repeated application of the same configuration, infrastructure code must also correctly handle *changes* to existing deployments. State evolution testing extends the concerns of idempotence testing to cover planned modifications: adding, updating, or removing resources from a running infrastructure stack. This distinction matters because many IaC defects only manifest when code changes are applied to infrastructure that is already in a specific state, rather than being provisioned from scratch. Drosos et al. (2024) provide empirical grounding for this concern: in their study of 360 real-world IaC bugs across Ansible, Puppet, and Chef ecosystems, more than half were state-dependent, meaning they manifested only under specific initial system conditions. For configuration unit bugs in particular, 65% required such pre-existing state to trigger, and in the majority of those cases that state was not managed by the configuration unit under test. This finding indicates that testing IaC code only from a clean initial state (as most integration test suites do) is insufficient to expose the majority of real-world defects. Validating that planned configuration changes can be safely applied to existing infrastructure therefore represents a distinct and underexplored testing concern beyond idempotence alone.

3.3.4. Negative Testing

While most IaC tests verify that infrastructure is correctly provisioned when valid configurations are provided, *negative testing* validates the complementary concern: that invalid, insecure, or policy-violating inputs are correctly rejected before deployment. In the IaC context, this includes verifying that misconfigured resources trigger expected errors, that policy-as-code rules block non-compliant configurations, and that security constraints are enforced under adversarial or boundary-condition inputs.

Morris (2021) connects this to outcome-based validation at the integration testing level: beyond verifying that resources are correctly provisioned, tests should also confirm that the system rejects invalid inputs and handles unexpected conditions gracefully. Despite its relevance, negative testing receives limited dedicated attention in the IaC testing literature, and systematic approaches to testing error conditions, rejection behaviour, and policy enforcement remain underexplored relative to positive validation.

3.3.5. Challenges and Limitations

Although integration testing provides stronger assurance of correctness than static analysis or unit testing, it is also considerably more costly and complex. Sokolowski et al. (2023) report that in programming-language-based IaC systems, developers depend almost

exclusively on integration testing due to the lack of practical alternatives, resulting in extended feedback cycles and delayed defect discovery. Running full deployments for every test cycle consumes significant time and cloud resources (Hasan et al., 2020; Morris, 2021), and Sokolowski et al. (2024) identify this overhead as a key reason for low testing adoption across open-source IaC projects. Beyond cost, integration test suites are also prone to flakiness caused by environmental instability, state drift between runs, and the complexity of managing credentials, cleanup, and parallel execution in shared environments (Morris, 2021).

Recommended mitigation strategies include progressive test layering, which defers expensive integration tests until faster checks have passed, and ephemeral environments that ensure each run starts from a clean state (Morris, 2021). Tighter CI/CD integration can further reduce overhead through automated teardown and environment isolation (Maliekal, 2025).

3.4. End-to-End Testing

3.4.1. Concept and Purpose

End-to-end (E2E) testing validates complete workflows across the entire infrastructure and application stack, verifying that the system functions correctly as an integrated whole under realistic conditions. Unlike integration testing, which focuses on component interaction and resource configuration, E2E testing targets user-facing outcomes: whether applications are reachable, whether scaling mechanisms respond correctly, and whether the system handles failures gracefully. Morris (2021) argues that E2E testing should focus on validating such outcomes rather than simply verifying resource existence — for example, confirming that users can successfully access an application through a load balancer rather than just checking that the load balancer was provisioned.

3.4.2. Testing in Production and Operational Realism

Morris (2021) provides further context for E2E testing through what he calls *testing in production*. He argues that as systems increase in complexity, the scope of risks that can be practically checked outside production shrinks, since production environments have characteristics that cannot be realistically replicated elsewhere: actual user behaviour, production-scale data volumes, real traffic patterns, and true concurrency levels. To mitigate the risks this brings, he outlines strategies for safe, incremental validation, including progressive deployment techniques, zero-downtime rollout patterns, canary releases, feature flags, and chaos engineering, all of which validate system resilience under realistic conditions rather than in isolated test environments.

3.4.3. Practice and Research Gap

Despite its conceptual importance, E2E testing is rarely implemented in practice. The high cost, long setup time, and operational risk associated with full-stack validation

appear to drive organisations to rely on integration tests, accepting a lower level of confidence in exchange for faster execution and lower maintenance overhead. Academic research also reflects this: most existing studies focus on static analysis, unit testing, or integration testing, with only indirect references to full-stack or production-level validation. Sokolowski et al. (2024), for instance, explicitly position their unit testing framework as a response to the prohibitive cost of integration and end-to-end testing, highlighting the absence of practical E2E support rather than addressing it. How to systematically validate entire infrastructure systems under realistic operational conditions therefore remains an underexplored area in the IaC testing literature.

3.5. Cross-Type Comparison

Table 3.1.: Comparison of IaC testing types across key dimensions

Test Type	Deployment Required	Speed	Cost	Confidence	Primary Purpose / Notes
Static Analysis	No	Seconds	None	Low–Medium	Detect syntax errors, security smells, policy violations; limited semantic insight.
Unit Testing	No / Minimal	Seconds–Minutes	Low	Medium	Validate logic in modules or imperative IaC; limited usefulness for declarative IaC with little isolated logic.
Integration Testing	Yes	Minutes–Hours	Medium–High	High	Validate actual resource creation and interactions; dominant practice in industry.
E2E Testing	Yes	Hours	High	Very High	Validate full system behaviour; rarely practised due to cost and complexity.

3.6. Research Gap

The literature review identifies several gaps that this thesis aims to address.

Limited empirical data on implemented practices. Surveys such as Guerriero et al. (2019) provide insights into practitioner perceptions and challenges, but they rely on self-

reported data rather than on direct observation of implemented practices. The reported gap between high demand for testing tools (82% desirability for static analysis) and low actual adoption (9% usage of linters) points to a disconnect that surveys alone cannot fully explain. Repository-level analysis can reveal what testing practices are actually implemented, independent of what practitioners say they do or plan to do.

Focus on individual testing types. Existing research tends to examine testing categories in isolation. Chiari et al. (2022) provide a comprehensive survey of static analysis tools but do not examine how static analysis fits into broader testing strategies. Hummer et al. (2013) focus specifically on idempotence testing for Chef scripts. Sokolowski and Salvaneschi (2023) address unit testing for Pulumi. What remains unclear is how projects combine different testing types and whether comprehensive testing strategies exist in practice.

Absence of evaluation frameworks. The literature provides recommendations for what should be tested but offers limited guidance on how to assess testing maturity. Morris (2021) describes progressive testing and various validation techniques, but does not define measurable criteria for evaluating whether a project has adequate testing. Practitioners currently have limited benchmarks against which to compare their own practices.

Underexplored testing types. End-to-end testing for IaC receives limited attention in the academic literature. As noted in the review, most studies focus on static analysis, unit testing, or integration testing, with E2E testing mentioned only briefly or discussed alongside production monitoring rather than as a distinct concern. While idempotence is recognised as a fundamental property of IaC, systematic approaches to testing idempotence beyond Hummer et al. (2013)'s work on Chef remain limited. More broadly, state evolution testing has received limited attention as a distinct concern separate from idempotence, despite empirical findings that many real-world IaC defects are state-dependent (Drosos et al., 2024). Negative testing and test isolation are similarly underrepresented in the reviewed literature, leaving practitioners with limited guidance on how to incorporate these practices into a comprehensive testing strategy.

Declarative versus imperative. The distinction between declarative and imperative IaC tools has significant implications for testing, yet most studies focus on one paradigm or the other. Sokolowski and Salvaneschi (2023) examine Pulumi specifically, while Hummer et al. (2013) focus on Chef. Cross-tool comparisons of testing practices are largely absent.

These gaps point to the need for empirical analysis of testing practices in real IaC repositories. Such analysis can provide evidence of what testing approaches are actually used, how they are combined, and where the largest gaps between recommendations and practice exist. This thesis addresses these gaps through systematic repository analysis, as described in the following chapters.

4. Methods

4.1. Research Methodology

To answer the research questions, this thesis combines literature review with repository analysis. The process begins with a literature review to establish the current State of the Art in IaC testing, identifying recommended practices, reported adoption rates, and research gaps. Based on the findings from the literature, an evaluation framework is developed that translates theoretical insights into measurable dimensions for assessing IaC testing maturity. Repositories are then selected for analysis using explicit search and filtering criteria, targeting projects with non-trivial testing implementations. The evaluation framework is applied to each selected repository to assess testing practices across the dimensions. Finally, findings are synthesised across repositories and compared against the literature to identify cross-cutting observations, strengths, and gaps in current IaC testing practices.

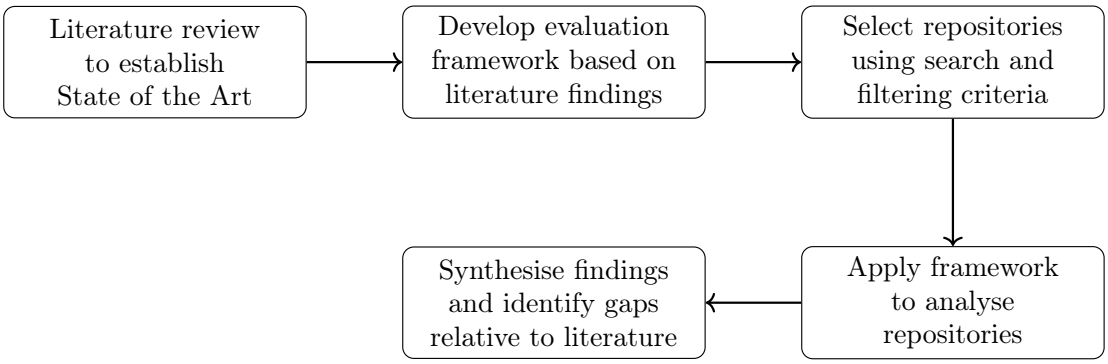

Figure 4.1.: Overview of the research approach.

In the following sections, the repository selection process and evaluation framework are described in detail.

4.2. Repository Search and Selection Criteria

Repositories were searched for on GitHub,[1] as it is the predominant hosting platform for open-source software and hosts the majority of publicly available IaC projects, including

[1] `https://github.com`

those from major cloud vendors, independent open-source organisations, and community-maintained collections. Since most publicly available IaC repositories contain little or no testing, random sampling would not be an efficient approach for finding relevant repositories as the results would be very limited. Instead, manual inspection targeted repositories that implement some form of IaC testing, then filtered for diversity and depth. The process combined keyword-based search, exploration of IaC tool ecosystems, and cross-referencing from practitioner literature, followed by screening based on repository metadata and explicit inclusion/exclusion criteria.

4.2.1. Search Strategy

IaC repositories vary widely in structure, purpose, and maturity. Repositories with meaningful testing are not centralised anywhere, making them difficult to locate through a single search approach. A multi-step strategy was therefore used:

1. **Keyword-based GitHub searches**. Queries such as "terraform terratest", "terraform integration tests", "terraform e2e tests", "iac testing", "ansible pytest", "terraform conftest", "helm testing", "infrastructure testing github actions", and related combinations were used to identify candidate repositories.

2. **Test-tool-targeted searches**. Rather than searching for IaC code directly, queries targeted the presence of testing infrastructure itself, for example `path:molecule` for Ansible role testing, `path:test filename:_test.go` for Terratest-style Go tests, and `path:.github/workflows filename:test` for CI-integrated testing. This ensured candidate repositories contained actual test implementations before deeper inspection.

3. **Exploration of established IaC organisations**. Certain organisations maintain consistently high testing standards across their repositories. Cloud Posse (for Terraform modules with standardised BATS/Terratest harnesses), Google Cloud Foundation Toolkit (for Blueprint Test integration), Ansible Collections under `ansible-collections/` (for `ansible-test` and `Molecule`), and Pulumi's provider repositories were systematically explored. Targeting these "gold standard" organisations efficiently surfaced repositories with mature, non-trivial testing practices.

4. **Exploration of official IaC ecosystems**. Many IaC tools publish official or community-supported examples that include testing. Repositories linked from the Terraform Registry, Terratest documentation, AWS/GCP/Azure sample solutions, and Ansible collections were manually inspected for testing practices.

5. **Cross-referencing from practitioner literature**. Blog posts, conference talks, and grey literature discussing IaC testing frequently reference public repositories; these were examined and added to the candidate pool when relevant.

6. **Repositories referenced in academic studies**. The reviewed academic papers were also examined for references to specific open-source repositories used as evaluation subjects or illustrative examples. Where these repositories met the activity

and testing evidence thresholds described below, they were added to the candidate pool.

Combining keyword search with manual inspection was necessary because many well-tested repositories are often hard to find through keyword search alone as they may use non-standard directory names, lack descriptive READMEs, or do not rank highly in GitHub's search results.

4.2.2. Preliminary Screening Based on Repository Metadata

After initial discovery, candidate repositories were screened based on repository metadata. Open-source ecosystems often contain substantial noise: abandoned projects, forks without meaningful changes, tutorial-level examples, and repositories with broken or incomplete test suites. Metadata screening filtered out these cases before deeper analysis.

The following metadata indicators were examined:

- **Activity**: Repositories with no commits in several years or showing no maintenance activity were excluded, as testing practices evolve over time and outdated repositories do not reflect current IaC workflows.
- **Structure**: Repositories with very small codebases, or no directory structure (e.g., no `modules/`, `test/`, or CI directories) were filtered out. Projects implementing non-trivial IaC usually contain a recognizable structure.
- **Testing evidence**: The presence of `test/`, `tests/`, or Terratest Go files (e.g., `_test.go`) was treated as an initial indicator of testing relevance.
- **CI/CD configuration**: Existence of GitHub Actions, GitLab CI, or other automated workflows was checked. This does not guarantee the presence of IaC tests, but it often signals more mature engineering practices.
- **Originality**: Forks were excluded unless they contained independent testing logic diverging from the upstream project.
- **Community adoption**: GitHub star counts served as a rough filter to exclude trivial projects and student work, with repositories below approximately 50 stars deprioritised. However, star count does not reliably indicate code quality or testing maturity, so exceptions were made for repositories demonstrating distinctive testing practices not observed elsewhere in the corpus. Above this threshold, repositories with evidence of active community engagement such as issues, pull requests, downstream dependents, and release history were prioritised over those with passive star counts alone.

Metadata screening reduced the initial set of approximately 80 candidates to around 30 repositories needing further evaluation. This filtering likely introduces a selection bias, i.e. favouring repositories that are actively maintained, have better IaC practices,

and are more likely to have testing, which is not representative of all IaC repositories on GitHub. However, this limitation is acknowledged and acceptable given the study's goal of understanding how testing is done when it is present, rather than estimating how common testing is across all IaC projects.

4.2.3. Inclusion Criteria

Repositories were included only if they met all of the following criteria:

- **Contains Infrastructure as Code.** Projects must include IaC configurations. Given Terraform's dominant position in the IaC ecosystem, its mature testing toolchain, and the better availability of well-maintained projects with meaningful test suites compared to other IaC tools, Terraform-based repositories were prioritised. However, to avoid limiting findings to a single tool's paradigm, projects combining multiple IaC tools (such as Terraform with Ansible, Helm, or CDK) and alternative testing approaches were deliberately included.
- **Contains IaC testing.** At least one form of testing must be present: static analysis, unit tests, integration tests or end-to-end tests. Testing could be implemented via established frameworks (such as `Terratest`, `pytest`, Go testing, etc.) or custom tooling.
- **Implements real-world infrastructure practices.** Repositories were expected to manage production-relevant infrastructure concerns: deploying cloud resources, configuring networking or security, implementing logging or monitoring pipelines, orchestrating multi-service environments, or building provider integrations. Trivial examples, tutorials, and student projects were excluded.
- **Publicly accessible.** Full access to IaC code, configuration, and test suites was required for analysis.
- **Sufficient complexity for framework application.** Repositories needed enough information and structure (modules, variables, CI workflows) to meaningfully apply the evaluation framework.

4.2.4. Exclusion Criteria

Repositories were excluded if they:

- lacked any IaC testing
- contained only trivial IaC examples or single-file modules
- were archived, inactive, or abandoned or
- were forks or mirrors without independent testing logic

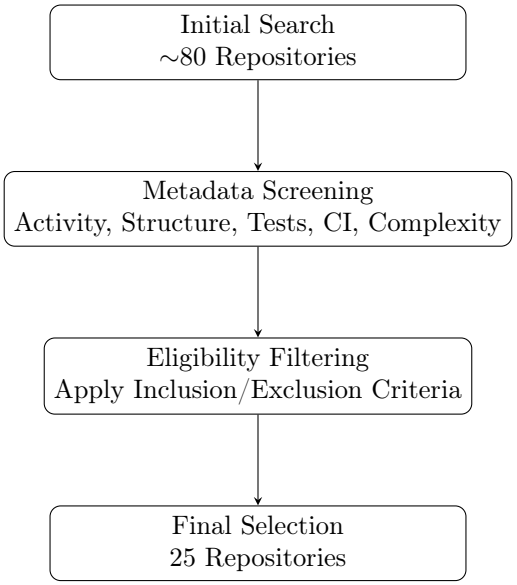

Figure 4.2.: Repository search and selection process.

4.2.5. Final Selection

Twenty-five repositories were selected to represent different project types, testing approaches, IaC tools, and complexity levels. These repositories span twelve distinct project types, where project type refers to the primary role a repository plays in the IaC ecosystem (for example, whether it provides reusable infrastructure modules, orchestrates deployments across multiple stacks, or implements a testing or transformation tool):

- **Terraform modules** (10): terraform-sumologic-sumo-logic-integrations, terraform-null-label, terraform-aws-eks-cluster, terraform-aws-rds, terraform-aws-vpc, terraform-azurerm-caf-enterprise-scale, terraform-google-address, terraform-google-bootstrap, terraform-google-kubernetes-engine, terraform-google-network
- **Terraform provider** (1): terraform-provider-commercetools
- **Platform generator** (1): Nebari
- **Organisation platform** (1): modernisation-platform
- **IaC orchestrators** (2): Atmos, Terragrunt
- **Multi-language IaC platform** (1): Pulumi
- **Multi-language providers** (2): pulumi-cloudflare, pulumi-gcp
- **IaC interop bridge** (1): pulumi-cdk

- **Ansible collections** (3): ansible-collections/amazon.aws, ansible-collections/community.zabbix, ansible-collections/ansible.netcommon
- **IaC file transformation tool** (1): yor
- **Kubernetes control plane** (1): Crossplane
- **IaC provider testing framework** (1): cloudformation-cli

This diversity enables comparison across the declarative–imperative spectrum (Terraform, Ansible, Pulumi, CDK), across cloud providers (AWS, GCP, Azure), across project scopes (utility modules to full platforms), and across implementation languages (Go, Python, TypeScript). The corpus includes projects from major cloud vendors (Google Cloud Foundation Toolkit modules, AWS CloudFormation CLI), independent open-source organisations (Cloud Posse, Gruntwork, Bridgecrew), CNCF graduated projects (Crossplane), and community-maintained Ansible collections. This breadth reduces the risk that observed practices reflect the conventions of a single organisation or tool ecosystem.

The selected repositories are not statistically representative of all IaC projects. They were chosen precisely because they implement testing, which most IaC repositories do not. Instead, they represent a curated selection of projects where testing practices can be meaningfully analysed and compared against recommendations from the literature.

Table 4.1 summarises the selection rationale for each repository, showing how the corpus was designed to maximise diversity across ecosystems, project types, test languages, and testing approaches.

Table 4.1.: Selection rationale for the twenty-five repositories.

Repository	**Ecosystem**	**Project Type**	**Test Lang.**	**Selection Rationale**
terraform-sumologic	Terraform	Terraform module	Go	Cross-platform validation: tests query Sumo Logic API after AWS provisioning; multi-apply state evolution
terraform-null-label	Terraform	Terraform module	Go	Purely functional logic with zero cloud dependencies; testing without provisioning
terraform-aws-eks	Terraform	Terraform module	Go	ChatOps-triggered testing; K8s operational readiness validation; dual TF/OpenTofu compatibility
terraform-aws-rds	Terraform	Terraform module	Go	Cloud Posse standardised harness (BATS + Terratest); organisational framework comparison
terraform-aws-vpc	Terraform	Terraform module	Go	Same Cloud Posse framework; adds Gateway and Interface VPC endpoint testing
terraform-azurerm-caf	Terraform	Terraform module	Go, Rego	Only Azure repository; OPA/Conftest baseline regression; security-separated CI

Continued on next page

Repository	Ecosystem	Project Type	Test Lang.	Selection Rationale
terraform-google-address	Terraform	Terraform module	Go	Google CFT Blueprint Test; dual verification (TF state + `gcloud` API)
terraform-google-bootstrap	Terraform	Terraform module	Go	Genuine E2E of provisioned CI/CD pipelines; validation of Cloud Build pipeline operational behaviour
terraform-google-gke	Terraform	Terraform module	Go	36 scenarios with golden file validation; application-level E2E with Nginx
terraform-google-network	Terraform	Terraform module	Go	Integration-only strategy (no unit tests); unvalidated migration tooling
commercetools	Terraform	Terraform provider	Go	Only provider in corpus; procedural diff logic; mock server for credential-free testing
Nebari	TF + Helm	Platform generator	Python	Most comprehensive layered testing: static, unit, integration, deployment, Playwright E2E
modernisation-platform	TF + OPA	Organisation platform	Go, Rego	Strongest policy-as-code negative testing via paired Rego unit tests
Atmos	Terraform	IaC orchestrators	Go	Custom AST-based `lintroller` plugin; 1,000+ test files; 80% coverage gate
Terragrunt	TF + OpenTofu	IaC orchestrators	Go	Most testing-intensive repo: 45 linters, fuzz testing, flaky detection, 179 fixture directories
Pulumi	Go/TS/Python	Multi-language IaC platform	Go	Largest multi-language IaC platform; property-based fuzz testing; 40k+ CI runs
pulumi-cloudflare	Pulumi bridge	Multi-language providers	Go	Provider upgrade testing with no-replacement assertions; daily upstream monitoring
pulumi-gcp	Pulumi bridge	Multi-language providers	Go	Drift detection testing; exponential backoff E2E; provider upgrade validation
pulumi-cdk	Pulumi + CDK	IaC interop bridge	TS, Go	Only CDK–Pulumi bridge; dual-language tests; strongest parameterised negative testing
amazon.aws	Ansible	Ansible collections	Python	API recording via `placebo` (unique practice); check mode validation
community.zabbix	Ansible	Ansible collections	YAML	Molecule + ansible-test dual architecture; multi-dimensional CI matrix; `testinfra`
ansible.netcommon	Ansible	Ansible collections	Python	Tests against real virtual network devices (NX-OS, IOS-XR, IOS-XE)
yor	TF + CFN	IaC file transformation tool	Go	Multi-format tagger; Go plugin architecture; comprehensive security scanning

Continued on next page

Repository	Ecosystem	Project Type	Test Lang.	Selection Rationale
Crossplane	Kubernetes	Kubernetes control plane	Go	CNCF graduated; Nix hermetic CI; OSS-Fuzz; `depguard` assertion banning
cloudformation-cli	CloudFormation	IaC provider testing framework	Python	Only CFN testing framework; hypothesis property testing; contract suite as product

The complete list of analysed repositories including their URLs and reference commit hashes (`repository_corpus_list.csv`), as well as a snapshot of all repositories at the inspected commits (`repository_corpus_snapshot.zip`), are available at `https://uclo ud.univie.ac.at/index.php/s/9bapYXLPfQPS6bk`.

4.3. Evaluation Framework Design

To assess IaC testing practices consistently across repositories, an evaluation framework was developed based on the testing practices identified in the literature review. The framework evolved during the analysis as certain dimensions proved more observable and relevant than others.

4.3.1. Evaluation Dimensions

The literature review identified several factors that influence the quality, effectiveness, and reliability of IaC testing. These were consolidated into eight evaluation dimensions, each corresponding to practices recommended in academic or practitioner sources. For each dimension, specific indicators were used to distinguish maturity levels during assessment.

Each repository was assessed against the following dimensions:

1. **Static analysis and syntax checks.** Presence and breadth of tools that validate code without provisioning infrastructure. Indicators of maturity include:
   - Multiple complementary tool types (syntax validation, linting, type checking, security scanning) rather than a single tool
   - Security-focused scanning (`Checkov`, `tfsec`, `Terrascan`, `gosec`) beyond basic syntax validation
   - Policy-as-code enforcement (`OPA`, `Conftest`, `Sentinel`) for organisation-specific rules
   - CI integration as a mandatory gate rather than optional or local-only execution
2. **Logic-level unit tests.** Tests that validate IaC logic in isolation without provisioning infrastructure. Indicators of maturity include:
   - Tests targeting actual logic (conditionals, string processing, type conversions) rather than trivial assertions on static values

- Use of mocks, stubs, or recorded API responses to simulate cloud provider behaviour offline
- Coverage of parameterised variations (e.g., table-driven tests across input combinations)
- Distinction between unit and integration scope (e.g., build tags or separate directories preventing accidental cloud provisioning)

3. **Integration testing.** Tests that deploy actual infrastructure and validate results. Indicators of maturity include:
   - Assertions that verify provisioned resource properties beyond IaC tool exit codes (e.g., querying cloud APIs, checking connectivity)
   - Coverage across module variants and input combinations, not just a single default scenario
   - Automated teardown via `defer`, cleanup hooks, or `always:` blocks registered before assertions
   - Handling of real-world concerns: timeouts, retries, and eventual consistency in cloud API responses
4. **End-to-end and outcome-based testing.** Validation that deployed infrastructure functions correctly as a system, beyond individual resource existence. Indicators of maturity include:
   - Application-level verification (HTTP endpoints respond, data flows through pipelines, users can complete workflows)
   - Validation across service boundaries (e.g., provisioned CI/CD pipeline triggers builds, deployed functions handle requests)
   - Use of dedicated E2E frameworks (`Playwright`, `Selenium`, custom HTTP clients)
5. **State evolution and migration testing.** Tests that verify infrastructure handles configuration changes across multiple apply cycles. Indicators of maturity include:
   - Explicit multi-apply sequences that modify, add, or remove resources and assert correct outcomes at each stage
   - Provider or schema upgrade testing (e.g., validating that version upgrades do not trigger unintended resource replacements)
   - Assertions distinguishing in-place updates from destructive replacements
   - Testing of migration paths and backward compatibility for breaking changes
6. **Test fixtures and isolation.** Mechanisms ensuring test reproducibility and independence between runs. Indicators of maturity include:
   - Resource isolation via unique naming, dedicated accounts, or namespace separation preventing cross-test interference

- Cleanup registered before assertions so that failures do not leak cloud resources
- Offline-capable testing through mocked APIs, recorded responses, or hermetic environments
- Controlled test state (temporary directories, pinned dependency versions, deterministic configuration)

7. **Negative and robustness testing.** Tests that deliberately exercise error conditions and edge cases. Indicators of maturity include:
   - Explicit tests for invalid inputs, malformed configurations, and missing required parameters
   - Verification that error messages, exit codes, or plan outputs are correct (not just that operations fail)
   - Boundary testing: quota limits, permission denials, conflicting resource configurations
   - Fuzz testing or property-based testing that generates randomised inputs to discover unexpected failures
8. **CI/CD integration.** Automation and sophistication of test execution in pipelines. Indicators of maturity include:
   - Tests triggered automatically on pull requests or commits with results gating merge approval
   - Separation of fast checks (static analysis, unit tests) from slow checks (integration, E2E) through pipeline stages or build tags
   - Cost and resource management strategies (ChatOps gating, cron-scheduled expensive tests, parallel execution)
   - Reproducible CI environments with managed credentials, pinned tool versions, and caching

4.3.2. Assessment Approach

Each dimension was assessed qualitatively based on evidence observed in the repository: test files, CI/CD configurations, helper code, and documentation. Assessments used the following ratings:

- **High maturity:** The practice is implemented comprehensively, covering the relevant scope of the project and in line with recommendations in the literature.
- **Moderate maturity:** The practice is present but incomplete, for example integration tests exist for some modules but not others, or CI runs tests but with limited automation.
- **Low maturity:** The practice is minimal or absent. Evidence may exist (e.g., a single linter configured) but coverage is limited.

- **N/A:** The dimension does not apply to the project. For example, end-to-end testing is not applicable to a utility module that provisions no infrastructure.

Intermediate ratings (e.g., "moderate-high" or "low-moderate") were used when evidence fell between categories.

A numeric scoring model was also considered but not adopted. Numeric scores imply that dimensions are comparable and additive, whereas in practice this is often not the case. Qualitative assessment allows each repository to be evaluated on its own terms while still enabling comparison. This thesis does not compute any statistical measures, as the focus is on understanding testing practices rather than deriving generalisable results. For each repository, findings are presented in evaluation tables that pair evidence with ratings, making the basis for each assessment transparent and traceable to specific code evidence.

Ratings are assigned relative to what is feasible and appropriate for the project type, not against a single fixed standard. The same rating can therefore reflect different tools, mechanisms, or levels of depth across repositories, as long as the practice is implemented well given the nature and scope of the project. Two repositories sharing the same rating within an evaluation category may differ in how that practice is realised; the per-repository appendix sections document the specific evidence for each rating.

4.3.3. Application of the Framework

The framework is applied in Chapter 5, which provides condensed overviews of each repository evaluation, and in Chapter 6, which synthesises cross-cutting findings. Full detailed evaluations are provided in Appendix A.

5. Evaluation

This chapter provides a condensed overview of the twenty-five repository evaluations analysed in this thesis, organised by project type. Each repository was evaluated against the eight-dimension framework described in Section 4.3.1. Full detailed evaluations, including testing architecture descriptions, test suite coverage tables, architectural diagrams, and dimension-by-dimension assessments, are provided in Appendix A for the first eight repositories. The remaining seventeen repositories follow a condensed format covering testing architecture, dimension ratings, coverage tables, and overall assessment, reflecting the scope constraints of a master's thesis.

Table 5.1 presents the assessment ratings across all twenty-five repositories and dimensions. Each rating reflects a qualitative judgement based on evidence observed in test files, CI/CD configurations, helper code, and documentation, as described in Section 4.3.2. **H** (High) indicates comprehensive implementation covering the project's relevant scope; **MH** (Moderate–High) and **M** (Moderate) indicate partial implementation with gaps in coverage or automation; **LM** (Low–Moderate) and **L** (Low) indicate minimal or limited presence; **N** (None) indicates complete absence; and **N/A** indicates the dimension does not apply to the project type.

Table 5.1.: Cross-repository comparison of IaC testing maturity, grouped by project type.

Repository	Static	Unit	Integ.	E2E	State	Isol.	Neg.	CI/CD
Project type: Terraform modules								
terraform-sumologic	MH	L	H	H	H	MH	L	H
terraform-null-label	M	H	MH	N/A	N/A	H	L	H
terraform-aws-eks	M	N/A	H	MH	LM	H	L	H
terraform-aws-rds	M	N	M	L	N	M	N	M
terraform-aws-vpc	M	L	H	M	LM	H	L	M
terraform-azurerm-caf	M	LM	H	MH	M	H	LM	H
terraform-google-address	MH	N	H	M	N	H	L	H
terraform-google-bootstrap	M	N	H	H	N	H	LM	H
terraform-google-gke	LM	N	H	H	N	H	L	H
terraform-google-network	MH	N	H	L	L	H	L	H
Project type: Terraform provider								
commercetools	M	MH	H	LM	MH	H	LM	H
Project type: Platform generator								

Continued on next page

Table 5.1.: Cross-repository comparison of IaC testing maturity (continued).

Repository	Static	Unit	Integ.	E2E	State	Isol.	Neg.	CI/CD
nebari	H	H	H	H	M	H	M	H
Project type: Organisation platform								
modernisation-platform	H	MH	H	LM	L	M	M	H
Project type: IaC orchestrators								
atmos	H	H	H	LM	L	H	LM	H
terragrunt	H	H	H	MH	L	H	M	H
Project type: Multi-language IaC platform								
pulumi/pulumi	H	H	H	N/A	MH	MH	M	H
Project type: Multi-language providers								
pulumi-cloudflare	H	M	H	LM	H	M	L	H
pulumi-gcp	H	H	H	H	H	H	M	H
Project type: IaC interop bridge								
pulumi-cdk	M	H	H	MH	M	H	MH	H
Project type: Ansible collections								
ansible-collections/aws	H	MH	H	L	MH	M	MH	H
ansible-collections/zabbix	M	L	H	L	MH	MH	LM	H
ansible-collections/netcommon	H	MH	H	M	LM	M	M	H
Project type: Kubernetes control plane								
crossplane	H	H	H	H	MH	H	MH	H
Project type: IaC file transformation tool								
yor	H	H	H	M	M	MH	LM	H
Project type: IaC provider testing framework								
cloudformation-cli	H	H	M	M	LM	H	MH	H

The following subsections summarise the distinctive testing characteristics of each repository, grouped by project type. Table 5.2 indicates which dimensions are most relevant for each category. Relevance ratings were derived from the repository evaluation findings: *Critical* indicates the dimension addresses a primary risk area for that project type (e.g., state evolution for modules managing long-lived infrastructure); *High* indicates significant benefit observed in repositories of that type; *Moderate* and *Low* indicate diminishing returns due to project characteristics (e.g., unit testing for declarative modules with little testable logic); *Universal* indicates the dimension applies universally without type-specific emphasis; and *N/A* indicates the dimension does not apply (e.g., E2E testing for utility modules that provision no infrastructure).

Table 5.2.: Relevance of testing dimensions by project type.

Project type	Static analysis	Unit testing	Integration testing	E2E testing	State evolution	Fixtures / isolation	Negative testing	CI/CD integration
Terraform modules	Universal	Low	Critical	Universal	Critical	Universal	Universal	Universal
Terraform provider	Universal	High	Critical	Low	Universal	Universal	Universal	Universal
Platform generator	Universal	High	Universal	Critical	Universal	Universal	Universal	Universal
Organisation platform	Universal	Moderate	Universal	Low	Universal	Universal	High	Universal
IaC orchestrators	Universal	High	Critical	Low	Low	High	Universal	Universal
Multi-language IaC platform	Universal	High	Critical	Universal	Critical	Critical	Universal	Universal
Multi-language providers	Universal	Moderate	Critical	Moderate	Critical	Moderate	Universal	Universal
IaC interop bridge	Universal	High	Critical	Moderate	Universal	High	High	Universal
Ansible collections	Universal	Moderate	Critical	Low	Universal	Universal	Universal	Universal
IaC file transf. tool	Universal	High	Universal	Low	N/A	Universal	Universal	Universal
Kubernetes control plane	Universal	High	Critical	High	Universal	Critical	Universal	Universal
IaC provider test framework	Universal	High	Moderate	Low	Universal	High	Critical	Universal

5.1. Terraform Modules

terraform-sumologic-sumo-logic-integrations stands out for its cross-platform validation: after provisioning AWS resources, tests query the Sumo Logic API to confirm that logs and metrics actually flow through the ingestion pipeline. State evolution is tested via `TestUpdates` in every module, toggling configuration flags across multiple apply cycles.

terraform-null-label validates purely functional transformations (naming conventions, tag generation) without cloud dependencies. A single comprehensive test covers dozens of input/output combinations, reflecting a focused strategy appropriate for a utility module.

terraform-aws-eks-cluster introduces ChatOps-triggered testing (`/terratest` comments) to control CI costs for expensive EKS provisioning. Its tests extend beyond infrastructure creation to validate Kubernetes operational readiness by monitoring worker node joins using Kubernetes event listeners (shared informers) that watch for nodes becoming ready, and it tests dual-platform compatibility across Terraform and OpenTofu.

terraform-aws-rds and **terraform-aws-vpc** follow Cloud Posse's standardised test-harness framework with BATS-based static analysis and Terratest integration testing. Both use ChatOps gating for cost-conscious cloud provisioning. terraform-aws-vpc adds comprehensive endpoint testing covering both Gateway and Interface VPC endpoint types.

terraform-azurerm-caf-enterprise-scale exhibits distinctive practices for large-scale enterprise modules: OPA/Conftest baseline regression testing validates plans against stored JSON baselines, a security-conscious CI architecture separates GitHub Actions linting from Azure Pipelines infrastructure tests, and CI workflows automatically detect available Terraform and provider versions by querying the HashiCorp and Azure APIs,

then generate a test matrix across those versions, with each job running in a dedicated Azure subscription to prevent resource conflicts.

terraform-google-address, **terraform-google-bootstrap**, **terraform-google-kubernetes-engine**, and **terraform-google-network** share the Cloud Foundation Toolkit (CFT) Blueprint Test framework, implementing a systematic dual verification strategy that combines Terraform state validation with `gcloud` API queries. terraform-google-bootstrap is notable for genuine E2E testing of provisioned CI/CD pipelines: triggering Cloud Scheduler workflows, polling builds, and verifying artifacts. terraform-google-kubernetes-engine covers 36 GKE scenarios with golden file validation and application-level E2E testing deploying Nginx services. terraform-google-network exemplifies a deliberate integration-only strategy with 17 scenarios but no unit tests, and includes migration tooling (`helpers/migrate.py`) that lacks automated validation, which is a practice common across multiple Terraform modules.

5.2. Terraform Provider

terraform-provider-commercetools illustrates how provider repositories can benefit from unit testing due to their procedural nature: unit tests cover custom diff logic, type handling, and state migration functions. A mock server architecture enables acceptance testing without real Commercetools credentials.

5.3. Platform Generator

Nebari exhibits the most comprehensive layered testing observed: extensive static analysis via pre-commit (`Ruff`, `Black`, `mypy`), unit tests with mocked cloud APIs, integration tests validating rendered artifacts across cloud providers, deployment tests against live services, and Playwright-based browser automation. This breadth reflects its role as a platform that generates Terraform and Helm artifacts rather than being IaC itself.

5.4. Organisation Platform

modernisation-platform combines Terraform integration tests with the strongest policy-as-code validation observed: OPA/Conftest policies with paired Rego unit tests covering both positive and negative scenarios (missing keys, invalid values, type mismatches). This systematic negative testing through policies is not observed in other analysed repositories.

5.5. IaC Orchestrators

Atmos contributes a custom `lintroller` plugin with lint rules that analyse the Go source code's syntax tree (AST) to enforce project-specific practices, integrated into `golangci-lint`, plus a `TestKit` wrapper that saves and restores global CLI settings

(environment variables, configuration files) before and after each test, preventing tests from interfering with each other.

Terragrunt exhibits the most testing-intensive architecture observed: a five-level strategy spanning static analysis (seven tools with 45 linters including seven test-quality-specific linters), unit testing (40+ packages), integration testing isolated via Go build tags (so unit and integration tests can be run separately) covering 179 fixture directories across 19 CI scenarios, Go native fuzz testing with auto-discovery, and dedicated flaky test detection tooling. It uniquely tests both Terraform and OpenTofu compatibility via build tags.

5.6. Multi-Language IaC Platform

Pulumi employs testing practices specific to multi-language platforms: dual-directory isolation preventing plugin version conflicts during parallel testing, a test provider architecture where each test registers custom handler functions for specific operations (allowing fine-grained control over simulated cloud behaviour), and property-based fuzz testing that generates 10,000 random sequences of create, update, and delete operations and verifies that internal state remains consistent after each sequence. 118 integration test scenarios and 40,000+ CI workflow runs reflect exceptional testing activity.

5.7. Multi-Language Providers

pulumi-cloudflare and **pulumi-gcp** represent multi-language providers bridging Terraform providers to Pulumi. Both implement provider upgrade testing with assertions that upgrading from a baseline provider version does not trigger unintended resource replacements (i.e., resources are updated in-place rather than destroyed and recreated), and scheduled daily upstream monitoring with automated PR creation. pulumi-gcp adds distinctive drift detection testing (modifying infrastructure via `gcloud` then validating Pulumi reconciliation) and E2E tests that call deployed Cloud Functions with increasing wait times between retries (exponential backoff) to handle startup delays.

5.8. IaC Interop Bridge

pulumi-cdk bridges AWS CDK and Pulumi ecosystems with a dual-language testing architecture: TypeScript unit tests validate CDK-to-Pulumi translation correctness (19 Jest test files with extensive parameterised testing of CloudFormation intrinsic functions such as Fn::Join, Fn::Select, and Ref via `test.each`), while Go integration tests validate deployment of translated constructs as real AWS infrastructure. Negative testing is notably strong, with parameterised error cases at both levels.

5.9. Ansible Collections

ansible-collections/amazon.aws introduces an innovative API recording practice: the `placebo` library records real AWS API responses as JSON fixtures, enabling deterministic, credential-free unit tests against actual API response structures. Integration testing covers 160 targets with explicit check mode validation ensuring dry-run operations have no side effects.

ansible-collections/community.zabbix implements a dual testing architecture: `Molecule` for role testing and `ansible-test` for module testing, with a multi-dimensional CI matrix (8 OS $\times$ 4 Zabbix versions $\times$ 2 databases). Its use of `testinfra` for infrastructure-level assertions (service status, file permissions, log content) goes beyond Ansible task-level success. Explicit idempotency validation by running each playbook twice and asserting the second run reports no changes, confirming the configuration is already in the desired state, appears in every integration test.

ansible-collections/ansible.netcommon provides network automation testing against real virtual devices provisioned through Cisco Modeling Labs: NX-OS, IOS-XR, and IOS-XE appliances testing four connection protocols (CLI, HTTPAPI, NETCONF, RESTCONF). A two-tier CI workflow delegation practice combines organisation-wide and team-specific reusable workflows.

5.10. IaC File Transformation Tool

yor tests a multi-format IaC tagger (Terraform, CloudFormation, Serverless) with golden file comparison (testing output against stored reference files) to ensure that tagging operations preserve the original file formatting, Go plugin architecture testing where custom tagger plugins are compiled as shared libraries during the test run and loaded dynamically, and multi-commit git repository testing for tag evolution. Its security scanning pipeline (`gosec`, `TruffleHog`, `Checkov`, `CodeQL`) is among the most comprehensive observed.

5.11. Kubernetes Control Plane

Crossplane implements a three-tier testing architecture: unit tests (138 files in the main codebase, with a further 38 files in the temporarily vendored `tmp/crossplane-runtime` module) with third-party assertion libraries explicitly banned via `depguard`; fuzz tests (9 functions across 8 files, with two targets in `internal/xcrd/fuzz_test.go`) with OSS-Fuzz continuous integration; and E2E tests deploying into temporary `Kind` clusters (created for the test and destroyed afterwards). Distinctively, Crossplane uses the Nix package manager as its primary build system: all unit tests run inside a sandboxed environment with no network or filesystem access, meaning that if a test passes on one machine it will pass on any other with the same inputs. Build artifacts are cached via `Cachix` to avoid redundant rebuilds.

5.12. IaC Provider Testing Framework

cloudformation-cli occupies a unique position: its contract test suite is both internal test code and a shipped product for downstream CloudFormation provider developers. Distinctive practices include property testing using the `Hypothesis` library, which automatically generates random but valid test inputs from each provider's JSON Schema definition rather than relying on hand-written test cases, an assertion framework where individual checks (e.g., "resource exists", "fields match input") are defined as decorators that can be composed together to build complex contract validations from simple reusable pieces, and the highest coverage threshold observed (98%) with context-aware exclusion of the shipped contract suite.

6. Results

This section synthesises cross-cutting findings from the twenty-five repository evaluations presented in Chapter 5. Rather than cataloguing individual practices, the analysis identifies eight thematic clusters that emerge across the corpus, each integrating evidence from multiple repositories with practitioner implications. Table 6.1 maps these findings to literature recommendations, and Table 6.3 summarises practitioner recommendations with adoption data. Figure 6.1 provides a visual overview of adoption rates across key practices.

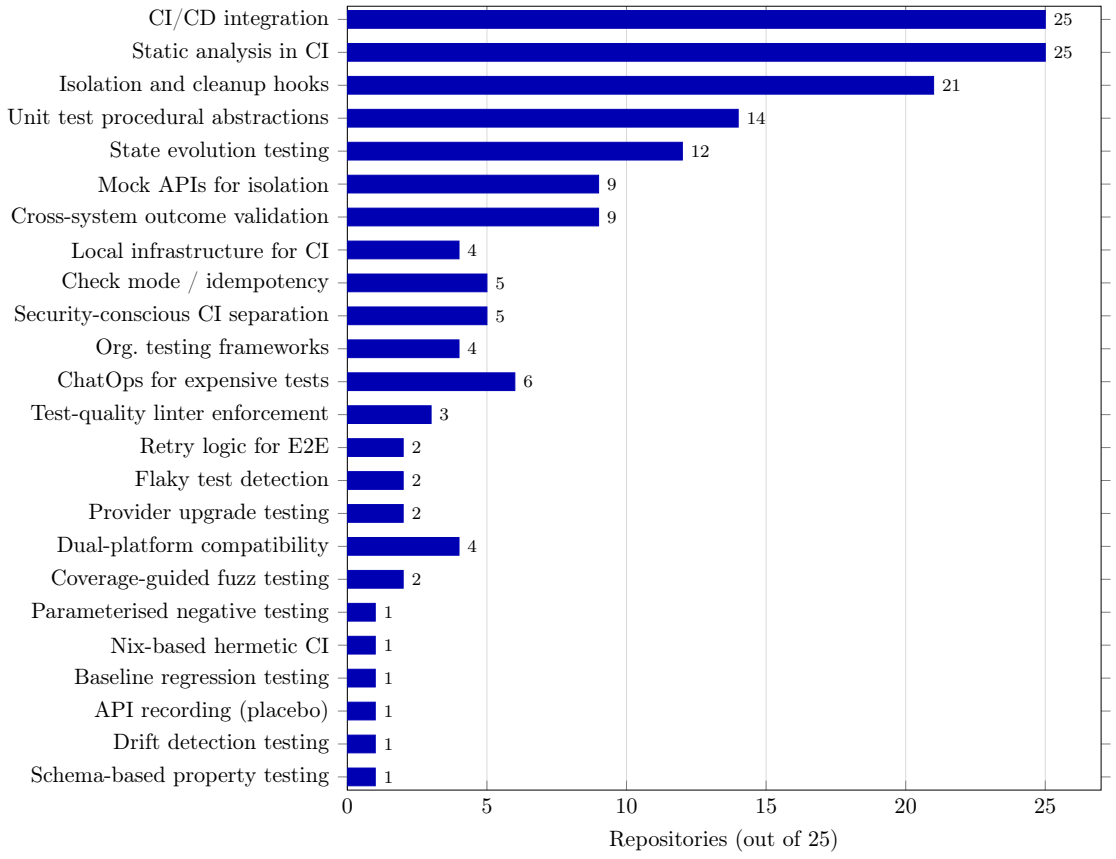

Figure 6.1.: Adoption of key testing practices across the twenty-five repositories ordered by adoption count. See Table 6.3 for the complete list of practices.

6.1. Testing Strategy by Project Type

Across the twenty-five repositories, integration testing is the dominant validation mechanism. Even terraform-null-label, which provisions no cloud resources, uses Terraform's evaluation engine as an integration layer. cloudformation-cli is a partial exception: while its contract test framework exercises the complete resource provider lifecycle (CREATE → READ → UPDATE → DELETE → LIST), it tests the framework's execution machinery rather than provisioning actual cloud infrastructure. More significantly, however, the distribution of testing effort across dimensions appears to vary with project type, in ways that reflect the underlying structure of each project rather than arbitrary choices.

Unit test investment tends to be higher in projects with substantial procedural code. Repositories implementing non-declarative logic, including Nebari (Python rendering pipeline), Commercetools (provider diff logic), modernisation-platform (Rego policies), Atmos and Terragrunt (Go CLI orchestrators), Pulumi (Go engine and SDKs), pulumi-cloudflare, pulumi-gcp, and pulumi-cdk (provider bridge and translation logic), Crossplane (Go reconciler controllers), cloudformation-cli (Python contract framework), and the Ansible collections (modules, connection plugins, filter plugins), all implement meaningful unit test suites. By contrast, pure declarative Terraform repositories (terraform-aws-rds, terraform-aws-vpc, terraform-google-address, terraform-google-bootstrap, terraform-google-kubernetes-engine, terraform-google-network) and collections focused on declarative automation (ansible-collections/community.zabbix) have minimal or no unit tests. terraform-null-label is a notable exception: despite being pure Terraform HCL, its utility-module nature (string processing, label generation, truncation) enables comprehensive output-based validation through assertions on naming conventions, tag propagation, and chaining behaviour.

The terraform-google-network repository illustrates the integration-only test strategies most clearly. With 16 submodules and production-grade maturity, it implements zero unit tests while investing heavily in 17 integration scenarios with real GCP resource provisioning. This reflects a conscious trade-off: slower test execution, cloud resource costs, dependency on provider availability, and complex environment setup are accepted in exchange for confidence in real-world module behaviour, validation of actual provider interactions, and detection of provider bugs and API inconsistencies. The pulumi-cdk repository represents the opposite case. As a library with substantial procedural logic, it implements 19 unit test files covering its translation engine, using Pulumi runtime mocks and filesystem mocking via mock-fs to validate translation correctness without cloud credentials.

End-to-end testing maturity follows a similar direction, scaling with project scope. Sumo Logic validates data flows from HTTP traffic through AWS into the Sumo Logic ingestion pipeline. Nebari tests deployed services via APIs and Playwright browser automation. terraform-google-bootstrap validates provisioned CI/CD pipelines by triggering Cloud Scheduler workflows, polling Cloud Build jobs, and verifying that artifacts are pushed to Artifact Registry. pulumi-gcp validates deployed Cloud Functions with HTTP requests and response assertions. pulumi-cdk validates deployed infrastructure with

HTTP endpoint checks and direct AWS SDK assertions. Crossplane validates complete user workflows (resource creation, package installation, composition changes, and lifecycle upgrades) though its E2E tests use a mock provider rather than provisioning actual cloud infrastructure.

Repositories focused on narrower concerns appropriately limit or omit E2E testing. Pulumi's architecture focuses on infrastructure provisioning correctness rather than application workflows, making full-stack E2E testing inapplicable to the platform itself. Similarly, pulumi-cloudflare focuses on control-plane validation (resource creation and configuration correctness) rather than data-plane behaviour such as DNS resolution or worker execution. It is also worth noting that within the High maturity evaluation for E2E testing, the nature of validation varies considerably across repositories. Crossplane and Nebari validate full deployment workflows against live infrastructure; terraform-sumologic validates API-level responses from provisioned cloud resources. Both satisfy the criterion for High but represent different depths of outcome coverage.

These observations suggest that testing strategy tends to be shaped by project type rather than applied uniformly. Across the repositories analysed, unit testing investment appears higher where procedural logic is present, integration testing serves as the baseline across all project types, and E2E testing scope appears to scale with the breadth of user-facing or multi-service behaviour a project delivers.

6.2. Static Analysis Maturity and Enforcement Practices

Static analysis is the most consistently implemented dimension across the analysed repositories, with twenty-four of twenty-five repositories achieving at least Moderate maturity and all twenty-five embedding tools in CI pipelines as mandatory PR checks. The breadth of tooling varies across ecosystems: Terraform repositories typically combine `terraform fmt`, `TFLint`, and `terraform validate`, with mature repositories adding security scanners (`Checkov`, `Trivy`, `TFSec`). Ansible collections integrate extensive tooling: ansible-collections/amazon.aws uses ten tools (`black`, `isort`, `flynt`, `flake8`, `pylint`, `ruff`, `mypy`, `ansible-lint`, `yamllint`, `ansible-test` sanity), while Go repositories rely primarily on `golangci-lint` with varying numbers of enabled linters, from 7 in pulumi-cloudflare and pulumi-gcp to 45 in Terragrunt. Security scanning is less consistently adopted: eleven of twenty-five repositories include dedicated security tools (Sumo Logic: `Checkov`; modernisation-platform: `Trivy`, `Checkov`, `TFSec`; yor: `gosec`, `TruffleHog`, `Checkov` secrets, `CodeQL`; Pulumi, Atmos, and Crossplane: `gosec` via `golangci-lint`; pulumi-cloudflare and pulumi-gcp: `gosec`; Nebari: `Trivy`; Terragrunt: `CodeQL`; cloudformation-cli: `bandit`). A more distinctive finding is that the analysed Go-based repositories exhibit three levels of enforcement for testing conventions, visualised in Figure 6.2.

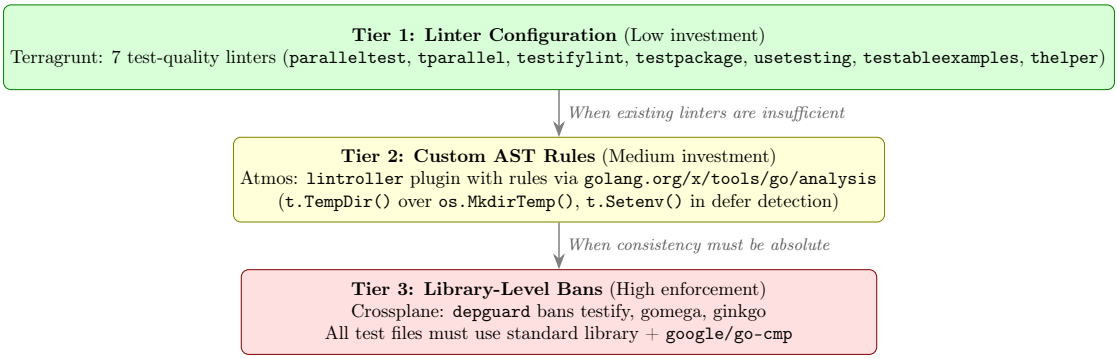

Figure 6.2.: Three-tier enforcement spectrum for testing conventions observed in the analysed Go-based IaC repositories. Each tier represents increasing investment and enforcement strength.

At the first tier, Terragrunt configures seven existing test-quality-specific linters within `golangci-lint`. These collectively enforce that all tests call `t.Parallel()` for concurrent execution, that testify assertions use correct syntax (e.g., `assert.Equal(t, expected, actual)` rather than `assert.True(t, expected == actual)`), that tests reside in external test packages (`_test` suffix), that test helper functions receive Go's test context (`testing.T`) so failures are reported at the correct location, and that tests use `t.Context()` (which is automatically cancelled when the test ends) rather than `context.Background()` (which is not). This approach requires no custom code and is reusable across Go projects.

At the second tier, Atmos implements a `lintroller` plugin that encodes four project-specific testing conventions as custom AST-based rules using `golang.org/x/tools/go/analysis`. These rules systematically prevent test isolation bugs: requiring `t.TempDir()` over `os.MkdirTemp()` (because the test framework automatically cleans up the former), requiring `t.Setenv()` over `os.Setenv()` (because the test framework automatically restores the original value), and detecting `t.Setenv()` inside defer blocks (which causes a panic because Go's test framework has already finished). The plugin integrates into `golangci-lint` as a module plugin and runs in pre-commit hooks.

At the third tier, Crossplane uses `depguard` to ban entire test assertion libraries (testify, gomega, ginkgo), mandating that all test code uses the standard library `testing` package with `google/go-cmp`. This ensures a single consistent assertion style across all test files, eliminating the possibility of inconsistent assertion styles that complicate code review and refactoring. The Go ecosystem's `golang.org/x/tools/go/analysis` framework makes custom linters relatively accessible; similar frameworks exist for other languages (ESLint rules for JavaScript, pylint plugins for Python, RuboCop for Ruby).

The observed progression suggests that existing linter configuration may be sufficient for common conventions, with custom AST rules and library-level bans representing progressively higher investment for project-specific requirements.

6.3. Test Isolation and Environment Management

Effective IaC testing requires deliberate isolation strategies that go beyond standard software testing practices. Unlike conventional software tests that can run in a single process with in-memory state, IaC tests interact with external systems such as cloud providers, configuration management agents, Kubernetes clusters, where shared state and global side effects create interference between parallel test executions. The analysed repositories exhibit a spectrum of isolation approaches, visualised in Figure 6.3, ranging from lightweight mocking to heavyweight cloud-level isolation.

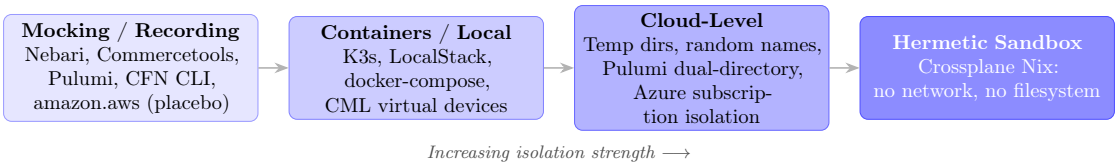

Figure 6.3.: Spectrum of test isolation strategies observed across the twenty-five repositories, from lightweight mocking to hermetic sandbox execution.

Mocking and API recording. At the lightweight end, several repositories use mocking to eliminate external dependencies entirely. Nebari uses pytest `monkeypatch` fixtures (which temporarily replace functions or values during a test and restore them afterwards) to simulate available Kubernetes versions, instance types, regions, and credential checks without calling real APIs. Commercetools uses a mock server as a GitHub Actions service container, enabling acceptance testing without real Commercetools credentials. The cloudformation-cli repository relies extensively on `unittest.mock` to isolate boto3 sessions, CloudFormation API calls, STS credentials, S3 uploads, Docker containers, and filesystem operations. Notably, it sets AWS credentials to empty strings in the pytest-local hook to prevent accidental cloud API calls which is a credential protection practice not observed in other Python repositories. Pulumi implements a lightweight mocking approach where test doubles are plain structs with function fields and each test assigns its own implementation to these fields, enabling test-specific behaviour without a mocking framework. The pulumi-cdk repository layers Pulumi runtime mocks (`pulumi.runtime.setMocks`) simulating AWS API responses (account IDs, regions, partitions, availability zones, SSM, Secrets Manager, ECR) with mock-fs for CDK assembly manifests, enabling the entire unit test suite to run without cloud credentials. The pulumi-gcp repository uses `httptest.NewServer` to mock GCP Compute API responses for region validation testing.

The ansible-collections/amazon.aws repository introduces a notable practice: the `placebo` library records real AWS API responses as JSON fixtures during initial test development (with credentials), then replays these recordings for subsequent test runs. This enables deterministic, fast, credential-free unit testing while validating against actual API response structures. The repository couples this with automatic `time.sleep()` patching during playback, accelerating tests that would otherwise wait for eventual consistency,

thereby reducing test duration from minutes to seconds. This addresses a challenge in cloud automation testing: achieving realistic validation without infrastructure costs, credentials, or network dependencies. For projects testing cloud provider interactions, API recording libraries (placebo for boto3, VCR.py for general HTTP, equivalent tools for other SDKs) provide a middle ground between manual mocking and requiring credentials for all test runs. Recording against multiple API versions may help catch compatibility issues.

Local infrastructure. Commercetools spins up a mock Commercetools server as a service container. Atmos uses K3s for Helmfile testing and LocalStack for S3 backend validation. ansible-collections/community.zabbix uses docker-compose to provision a complete Zabbix stack (PostgreSQL + Zabbix server + web UI) for integration testing. At the more resource-intensive end of local infrastructure, ansible-collections/ansible.netcommon provisions real Cisco NX-OS 9000v, IOS-XR 9000v, and Catalyst 8000v/IOS-XE virtual devices connected via an unmanaged switch through Cisco Modeling Labs. Dynamic inventory is generated from DHCP-assigned IPs with computed port-forwarding offsets. This represents the opposite end of the local infrastructure spectrum: where Atmos uses lightweight containers, CML provisions full vendor network operating systems when protocol fidelity (CLI, HTTPAPI, NETCONF, RESTCONF) demands it, at significantly higher resource cost. These practices enable integration testing without cloud provider accounts, reducing one of the key barriers to CI/CD integration identified across repositories.

Cloud-level isolation. Most repositories use temporary directories (`CopyTerraformFolderToTemp`, `t.TempDir()`) and randomised resource naming (`random.UniqueId()`). Pulumi implements a more sophisticated dual-directory approach separating HomePath for plugin installations, configuration files, and credentials from RootPath for workspace files and program execution. This addresses a specific challenge in multi-version testing: when tests run in parallel with different provider versions, a shared plugin cache causes version conflicts. The isolation is achieved through environment variable injection (PULUMI_HOME, PULUMI_CREDENTIALS_PATH) rather than modifying global filesystem locations.

The terraform-azurerm-caf-enterprise-scale repository achieves the strongest cloud-level isolation observed: a PowerShell script uses the Azure Subscription Aliases API (`Microsoft.Subscription/aliases@2021-10-01`) to allocate dedicated Azure subscriptions to each matrix job, providing complete tenant-level separation. This gives each job dedicated subscriptions for resource deployment, preventing resource naming conflicts and avoiding Azure API rate limiting, with certificate-based authentication through dedicated Service Principals. Crossplane achieves the strongest build-level isolation: all unit tests run inside the Nix sandbox with no network or filesystem access, ensuring that if a test passes in the sandbox, it will pass on any machine with the same Nix inputs.

Cleanup should be registered immediately after creating resources (before any assertions) to ensure it runs even if tests fail. All Ansible collections use `always:` blocks in playbooks; Go repositories use `defer` and `t.Cleanup()` hooks; terraform-azurerm-caf-enterprise-scale

includes dedicated cleanup jobs with `always()` conditions and 120-minute timeouts; and ansible-collections/ansible.netcommon implements CML lab lifecycle management with lab-create → integration-tests → lab-destroy using `always()` conditions.

6.4. Negative Testing and Robustness Validation

Negative testing, i.e. verifying that invalid inputs are rejected and failure modes are handled correctly, is the most consistent gap across the analysed repositories. While a few repositories test specific error paths, such as Terragrunt validating credential error explanations, ansible.netcommon asserting on RESTCONF/NETCONF protocol errors, and Pulumi using specialised test providers that deliberately fail, no repository in the analysed corpus systematically tests for cloud API failures, quota exhaustion, or permission denials as part of its regular test suite. Retry logic for transient failures is more common (present in Sumo Logic, terraform-google-bootstrap, terraform-google-gke, terraform-azurerm-caf, terraform-aws-eks-cluster, and pulumi-gcp) but handles eventual consistency rather than exercising failure scenarios. Despite this overall gap, five distinct approaches were observed for partially addressing negative testing, each suited to different project types.

Policy-as-code. In the modernisation-platform, OPA/Conftest policies are used to systematically test misconfiguration scenarios for declarative configurations. Each Rego unit test constructs malformed input like missing keys, invalid values, type mismatches, out-of-range parameters, and asserts the expected denial message. This approach may explain why negative testing appears strongest in the repository with a policy layer: at the Terraform level, errors are hard to test because they often appear as cryptic provider failures rather than structured error responses. The modernisation-platform case suggests that a policy-as-code layer can provide a testable abstraction over infrastructure validation rules for declarative configurations, where errors are otherwise difficult to test in a structured way.

Parameterised error cases. The pulumi-cdk repository exhibits the most structured dual-level negative testing observed. Unit tests use `test.each` to systematically cover missing CloudFormation mappings, absent mapping sections, incorrect parameter counts, missing keys at multiple nesting levels, and unresolvable cross-stack references. Integration tests include dedicated error scenarios (`errors-test`, `unsupported-error`) with `ExpectFailure` and stderr verification of specific error messages for unsupported CloudFormation types. This dual-level approach is feasible because the translation layer produces structured, predictable errors rather than unclear provider failures.

Contract-level error validation. The cloudformation-cli repository implements systematic negative tests for all lifecycle operations, validating expected error codes: update-without-create → NotFound, delete-already-deleted → NotFound, read/update/list after delete → NotFound, duplicate create with writable identifiers → AlreadyExists, and

unsupported hook target → UnsupportedTarget. Project-level negative tests additionally cover invalid schemas, path traversal blocking, credential exposure prevention, and invalid payloads.

Reconciler error path coverage. Crossplane exhibits extensive error path coverage across all 20 reconciler controllers (the components responsible for continuously aligning actual infrastructure with desired state), covering resource-not-found, retrieval failure, and update failure scenarios. Tests also verify that errors propagate correctly through composition functions (e.g., malformed protocol buffer inputs, failed credential lookups, unparseable responses), that the admission webhook rejects invalid or protected resource definitions, and that the package manager handles dependency errors (incompatible versions, missing packages, insufficient permissions). This represents among the most thorough error path coverage in the Go repositories in this study, though it remains focused on controller-level errors rather than infrastructure failure modes. Pulumi complements this with specialised test providers (`fails_on_create`, `fails_on_delete`) and event stream validation ensuring error messages contain specific guidance for users, i.e. testing not just that errors occur, but that error messages are actionable.

Coverage-guided fuzz testing. Fuzz testing validates robustness against arbitrary inputs, catching issues that hand-written test cases miss. Terragrunt implements Go native fuzz testing for three parser components (`FuzzParse`, `FuzzLexer`, `FuzzParser`) with one seed input per target (three seeds total) covering paths, attributes, operators, graph expressions, and edge cases. A distinctive aspect is automatic fuzz function discovery in CI: the Makefile iterates all packages, lists `Fuzz*` functions via `go test -list`, and runs each with configurable fuzz time (30 seconds per target), so new fuzz functions are automatically included when added to any package. Crossplane extends this with nine fuzz functions across eight files registered with Google's OSS-Fuzz continuous fuzzing service, providing ongoing automated testing beyond CI runs on dedicated infrastructure. These complement Pulumi's property-based fuzz testing using the Rapid library, which validates that internal state remains logically consistent across 10,000 random operation sequences, rather than testing only whether inputs parse without crashing.

The approach to negative testing in the analysed repositories depends on project type: policy-as-code appeared effective for declarative configurations, parameterised error testing for procedural logic with well-defined error conditions, contract-level validation for framework and provider testing, and fuzz testing for parsers and interpreters.

6.5. State Evolution and Migration Testing

State evolution testing, i.e. validating that infrastructure handles configuration changes, provider upgrades, and migrations correctly, shows the widest variation across repositories: twelve of twenty-five implement meaningful coverage, while the remainder lack explicit state evolution testing. Figure 6.4 shows the distribution of state evolution coverage by ecosystem.

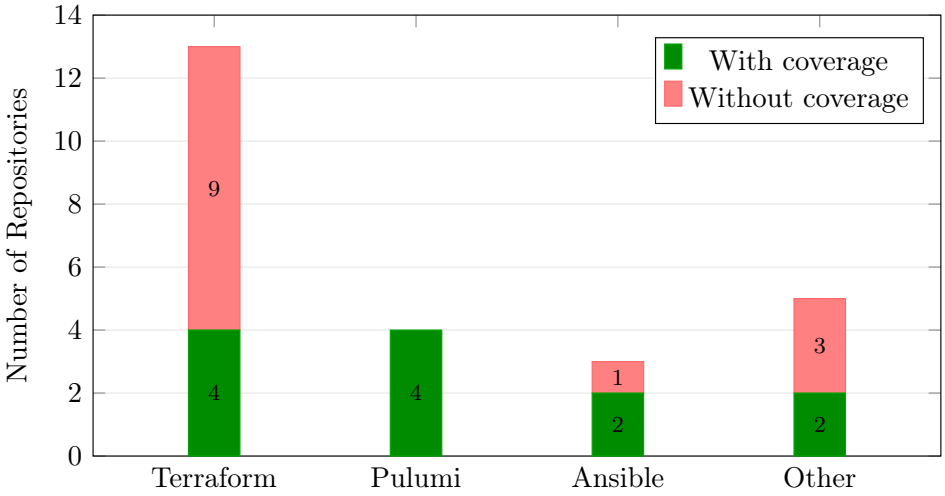

Figure 6.4.: Observed state evolution testing coverage by ecosystem in the analysed repositories. Pulumi shows the strongest coverage (4/4), followed by Ansible (2/3) and Terraform-ecosystem repositories (4/13). 'Other' includes IaC orchestrators (Atmos, Terragrunt), IaC tooling (yor), the Kubernetes control plane (Crossplane), and the CloudFormation testing framework (cloudformation-cli).

Strong implementations are different depending on the IaC ecosystem. In the Terraform ecosystem, Sumo Logic implements `TestUpdates` tests in every module, executing multiple `terraform apply` cycles with changing configurations (toggling `create_*` flags, switching between new and existing resources, modifying tags) and asserting expected add/change/destroy counts after each step. Commercetools includes explicit state migration tests for provider schema upgrades. The terraform-azurerm-caf-enterprise-scale repository implements sequential E2E testing with shared state across three test modules, validating incremental deployment workflows typical of enterprise landing zone updates.

In the Pulumi ecosystem, state evolution testing is particularly mature. Pulumi implements property-based fuzz testing with 10,000 iterations validating state consistency across random operation sequences, plus an EditDirs pattern where each test step is represented as a directory containing the modified program: the test framework applies each directory in sequence, running validation after each step. Steps can be flagged as expected failures, enabling tests for partial failures, rollbacks, and incremental fixes. Pulumi also loads exported stack states from production users for reproduction testing, which pulumi-cloudflare extends with recorded state files from three provider versions (v4, v5, v6) validated through import and preview operations. pulumi-gcp implements systematic provider upgrade testing with a default baseline version of 6.67.0, ensuring schema changes do not trigger unintended resource replacements. pulumi-cdk reuses the EditDirs pattern for multi-step lifecycle tests validating removal policies (retain-then-destroy with intermediate AWS SDK verification), resource replacement, and SSM

parameter updates.

In the Ansible ecosystem, ansible-collections/amazon.aws implements check mode testing: creating resources in check mode, querying cloud providers to confirm no actual resources were created, then running without check mode and verifying resources exist. This validates an important safety property: planning operations have no side effects. ansible-collections/community.zabbix validates idempotency through dual execution in every integration test. ansible-collections/ansible.netcommon implements RESTCONF lifecycle testing covering create, read, update, and delete operations with idempotent deletion assertions (first delete reports a change, second delete confirms nothing remains to change) and NETCONF transactions with confirmed commit and automatic rollback on error. However, this coverage is limited in scope relative to the other Ansible collections, reflecting its Low–Moderate rating on this dimension. In the broader ecosystem, yor implements multi-commit tag evolution testing verifying that `yor_trace` identifiers persist across re-tagging while git metadata tags update correctly, plus idempotency validation confirming that tagging twice without committing produces stable results. Crossplane implements lifecycle upgrade E2E testing (installs a prior version via Helm, creates claims, upgrades to the current build, verifies claims survive), composition revision propagation, and package dependency upgrade scenarios (version, digest, transitive, downgrade, already-exists), though it lacks CRD schema versioning evolution testing. The cloudformation-cli repository addresses state evolution through contract test lifecycle sequences (create → read → update → delete → list) with model consistency validation after property pruning and asynchronous state transition testing via IN_PROGRESS callback polling.

Two specialised sub-practices address specific state evolution challenges. **Provider upgrade testing** validates schema evolution across provider versions rather than configuration changes within a single version. The pulumi-cloudflare repository tests upgrades from baseline version 5.49.0 using `providertest.PreviewProviderUpgrade`: tests create infrastructure using the baseline provider, preview upgrade to the current version, and assert `HasNoReplacements` and `HasNoDeletes`. Version-specific test program directories (`test-programs/zone/zonev5`) contain configurations compatible with the baseline provider. The `TestRuleSetNonEmptyRulesUpgrade` test extends this by importing recorded state files from three provider versions and validating that preview operations succeed, addressing a specific issue where state schema changes caused preview failures in production. pulumi-gcp follows a similar approach with `TestUpgradeCoverage` generating reports showing which resources have upgrade test coverage.

Baseline regression testing validates plan consistency against stored known-good baselines. The terraform-azurerm-caf-enterprise-scale repository maintains `baseline_values.json` files containing JSON representations of known-good Terraform plans. During unit tests, a shell script dynamically updates these baselines by substituting test-specific values using `jq`, then uses Conftest to validate current plans against baselines across six resource types (management groups, policy definitions, policy set definitions, policy assignments, role definitions, role assignments). A dedicated baseline maintenance pipeline (`tests-update.yml`) triggered via `/azp run update` regenerates baselines when intentional changes occur, with `git diff` providing visibility into how module changes

affect the plan.

The most notable gap is the presence of migration tooling without automated validation. The terraform-google-network repository includes a Python migration script (`helpers/migrate.py`) generating `terraform state mv` commands for module refactoring, handling count-to-for_each migrations with `-dryrun` preview mode, yet no automated tests validate migration correctness. This practice extends across multiple repositories: terraform-aws-eks-cluster, terraform-aws-rds, terraform-aws-vpc, terraform-google-address, terraform-google-bootstrap, and terraform-google-kubernetes-engine all include migration tooling or manual upgrade documentation without automated validation. The terraform-google-kubernetes-engine case is particularly notable: despite manual upgrade documentation covering 30+ major versions, no automated tests validate these upgrade paths.

Across the analysed repositories, state evolution testing appears most mature in the Pulumi ecosystem, where provider upgrade testing and multi-step lifecycle validation are consistently implemented, and least developed in the Terraform ecosystem, where migration tooling is common but automated validation of migration correctness is largely absent.

6.6. Cross-System and Outcome-Based Validation

The most mature repositories validate beyond IaC tool success by querying external systems, verifying actual cloud state through provider APIs, and testing application-level behaviour. Figure 6.5 illustrates the progression of validation maturity observed across the analysed repositories.

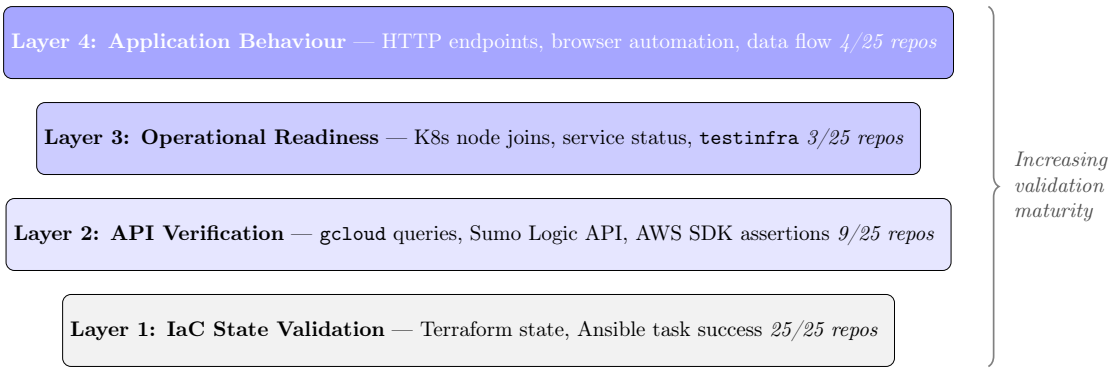

Figure 6.5.: Four layers of validation maturity observed in analysed repositories. Most repositories validate only at Layer 1 (IaC state). The most mature repositories reach Layer 4 (application behaviour), catching issues invisible to lower layers.

Dual verification. The strongest systematic practice is combining IaC state validation with independent API queries. The four Google CFT repositories (terraform-google-

address, terraform-google-bootstrap, terraform-google-kubernetes-engine, terraform-google-network) implement this via the Blueprint Test framework: `DefaultVerify()` validates Terraform state consistency, followed by `gcloud` CLI commands querying actual GCP resource state via compute and DNS APIs. For example, `TestDnsForwardExample` validates that DNS records not only exist in Terraform state but also contain the correct IP addresses when queried through `gcloud dns record-sets list`. This catches issues where Terraform reports success but actual infrastructure state differs due to eventual consistency delays, IAM permission failures, or provider bugs. Sumo Logic extends validation further: after provisioning AWS infrastructure, tests query the Sumo Logic API to confirm that logs and metrics actually flow through the ingestion pipeline. The ELB test creates infrastructure, sends HTTP traffic, and verifies access logs appear in both S3 and Sumo Logic. pulumi-cdk validates deployed infrastructure through HTTP endpoint checks (`AssertHTTPResultWithRetry` for custom resources and nested stacks) and direct AWS SDK assertions verifying S3 bucket existence and deletion for removal policy lifecycle tests.

Operational readiness validation. Beyond verifying that resources exist, the terraform-aws-eks-cluster repository validates cluster operational readiness by monitoring worker node joins using Kubernetes event watchers (shared informers from `k8s.io/client-go`). Tests authenticate to the cluster using AWS IAM tokens, set up Kubernetes event watchers (informers) to observe node resources, and use thread-safe counters to track how many nodes have joined. A 5-minute timeout ensures that at least 2 nodes successfully join the cluster. This catches issues invisible to Terraform state: IAM permission problems preventing node registration, VPC networking misconfigurations blocking kubelet communication, or EKS addon failures preventing node readiness. The ansible-collections/community.zabbix repository uses `testinfra` to validate actual system state after role execution: service status (`assert zabbix.is_running`), file ownership and permissions (`assert conf.mode == 0o640`), configuration content (`assert conf.contains("ListenPort=10051")`), and log content (`assert not logfile.contains("Access denied")`). This catches issues where Ansible reports success but the deployed system is misconfigured such as services that start but immediately crash, configuration files with incorrect permissions, or database connection failures not surfaced during role execution.

Drift detection testing. The pulumi-gcp repository implements drift detection: creating infrastructure through Pulumi, modifying it externally using `gcloud` commands, then validating that subsequent Pulumi operations correctly detect and reconcile the drift. The `TestCloudrunServicePanicRegress2622` test creates a Cloud Run service, uses `gcloud run deploy` to change its CPU configuration, then runs `pulumi up` to verify the provider detects the external modification and reconciles state correctly. This tests a critical but often overlooked scenario: infrastructure that changes outside the IaC tool's control, whether through manual operations, other automation, or external systems. Providers must correctly detect such drift, determine required changes, and apply corrections

without data loss or unnecessary resource recreation. For IaC providers and platforms, drift detection testing can be especially useful in brownfield environments (existing infrastructure that was set up manually or by other tools before IaC adoption), during incident response, and in organisations with multiple automation systems.

Exponential backoff. The pulumi-gcp repository implements a systematic approach to eventual consistency in E2E testing. The `validateAPITest` helper implements exponential backoff with configurable retry schedules (5s, 10s, 20s, 40s, 80s, 160s) and an `isRetryable` function that distinguishes transient failures (5xx errors, 404s during provisioning) from permanent failures. This prevents flaky tests where services report as created but take additional time to become operational. terraform-google-bootstrap extends this by polling Cloud Build jobs and workflow executions with similar retry logic (up to 100 iterations, 33-minute maximum) and handling eventually consistent IAM through workflow retry on `FAILED` status. This exponential backoff with configurable retry schedules can provide more reliable E2E validation than single-attempt checks for eventually consistent cloud services, avoiding false failures caused by services that are provisioned but not yet fully operational.

6.7. CI/CD Optimisation Practices

CI/CD integration is strong across the corpus: all twenty-five repositories rate High, with all using GitHub Actions (some additionally using Azure Pipelines or Cloud Build). Beyond this baseline, mature repositories develop sophisticated practices addressing four challenges specific to infrastructure testing, visualised in Figure 6.6.

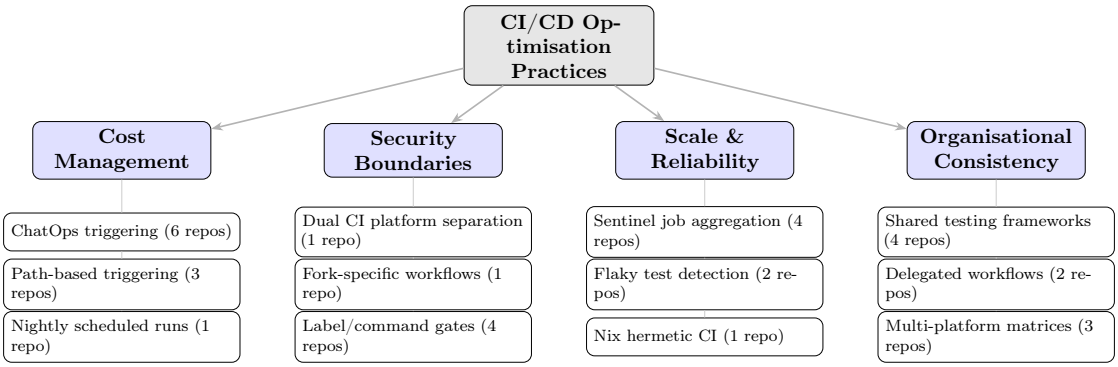

Figure 6.6.: Taxonomy of CI/CD optimisation practices observed across the twenty-five repositories, organised by the challenge they address.

Cost management. Infrastructure integration tests incur real cloud costs, and several repositories implement different practices to manage this. ChatOps triggering, i.e. on-demand test execution via pull request comments, is the most commonly observed practice.

The terraform-aws-eks-cluster repository implements `/terratest` comments with access control enforcement (requiring `run_terratest` permission), pending GitHub status checks before execution, status updates on completion, and reactions to triggering comments for user feedback. This addresses the economic reality that provisioning complete EKS clusters is expensive (120-minute test timeout); automatic execution on every commit would incur significant AWS costs. terraform-aws-rds and terraform-aws-vpc follow the same approach. pulumi-cdk extends it with an additional security dimension: `/run-acceptance-tests` slash commands restrict cloud tests to trusted contributors, while a separate workflow notifies external fork contributors that acceptance tests require maintainer approval. Complementary practices include path-based triggering, i.e. running workflows only when relevant files change (ansible-collections/community.zabbix, terraform-aws-rds, terraform-aws-vpc), and nightly scheduled runs separating expensive tests from PR feedback loops (pulumi-cdk). Parallel test splitting distributes tests across multiple CI runners: pulumi-cdk uses `go-test-split-action` across 3 runners, and Terragrunt partitions integration tests into granular CI scenarios via build tags.

Security boundaries. Open-source IaC projects face a security challenge: accepting PRs from untrusted forks while protecting cloud credentials from malicious code. Five distinct approaches emerge. terraform-azurerm-caf-enterprise-scale uses different CI platforms for different trust levels: GitHub Actions workflows run automatically (including from forks) but only perform static analysis without Azure credentials, while Azure Pipelines workflows require manual approval via comment triggers (`/azp run unit`, `/azp run e2e`) by maintainers. The repository's documentation explicitly explains this security model, ensuring future maintainers understand the architecture. pulumi-cdk runs a dedicated `pull-request.yml` workflow for fork PRs providing lint/build feedback while restricting cloud test execution to maintainer-approved commands. pulumi-cloudflare and pulumi-gcp use repository_dispatch requiring explicit `/run-acceptance-tests` commands before running tests. ansible-collections/ansible.netcommon uses a "safe to test" label gate before exposing CML credentials. yor implements `pull_request_target` with a GitHub environment approval gate. Terragrunt isolates OIDC authentication tests into a separate workflow with elevated `id-token: write` permissions. All share a common principle: separate automated linting (safe for untrusted code) from credential-requiring tests (manual approval required).

Distribution integrity testing. Terragrunt extends CI security beyond source code to the binary distribution pipeline. A dedicated `install-script-test.yml` workflow validates the installation script across Ubuntu and macOS, verifying GPG signature checking and Cosign verification of released binaries. This tests not the IaC tool's functionality but the integrity of what end users receive, which is something that other analysed repositories do not present. Separate signing workflows for macOS and Windows binaries complete the supply chain, though only the installation verification is automated as a test.

Scale and reliability. Sentinel jobs (a single CI job that only passes when all other required checks succeed) prevent race conditions where branch protection rules might allow merging before all parallel checks complete (pulumi-cloudflare, pulumi-gcp, pulumi-cdk, ansible-collections/ansible.netcommon). Programmatic matrix generation in terraform-azurerm-caf-enterprise-scale queries publisher APIs for latest Terraform and AzureRM provider versions via PowerShell, constructing test matrices combining minimum and latest versions and allocating dedicated Azure subscriptions per job, thereby eliminating manual version tracking. Scheduled upstream monitoring with automated PR creation tracks dependency changes daily (pulumi-cloudflare, pulumi-gcp detect new upstream Terraform provider versions at 3 AM UTC). For test reliability at scale, Terragrunt implements custom flaky test detection tooling (a standalone Go module in `test/flake/` with dedicated components for test result analysis, GitHub API integration, output parsing, and type definitions), while Crossplane publishes results to BuildPulse, those being the only two repositories with systematic flake detection. Crossplane's Nix-based hermetic CI provides the strongest reproducibility guarantees: all unit tests run inside the Nix sandbox with no network or filesystem access, complemented by Cachix binary caching at individual package granularity. Since Nix identifies build outputs by the hash of their inputs (content-addressing), unchanged packages are fetched from cache rather than rebuilt, significantly reducing CI execution time. The trade-off is increased complexity: Nix's functional language and derivation model have a steep learning curve.

Performance regression as an untapped CI dimension. Five repositories implement Go benchmark tests measuring configuration parsing throughput (Atmos), stack operation latency (Pulumi, Terragrunt), provider SDK performance (pulumi-gcp), and tagging speed (yor). However, none uses benchmarks as a CI gate. None of the analysed repositories runs `benchstat` comparisons against a baseline, defines performance budgets, or fails builds on regressions. In these repositories, benchmarks exist as developer tools but are disconnected from CI enforcement. For IaC orchestration tools where configuration parsing speed directly affects developer feedback loops, and for providers where API translation latency accumulates across hundreds of resources, this represents an opportunity to catch performance regressions before they reach users.

Organisational consistency. The four Google CFT repositories share the Cloud Foundation Toolkit Blueprint Test framework, which provides standardised testing practices including lifecycle management (init, apply, verify, teardown) handled automatically and built-in helpers like `GetStringOutput()` and `GetTFOptions()`. This embeds organisational testing conventions (dual verification) into the framework itself, reducing per-project configuration and ensuring consistency. ansible-collections/amazon.aws and ansible-collections/ansible.netcommon use delegated workflows: shared, reusable CI workflow definitions maintained at the organisation level so that individual repositories call a central template rather than duplicating pipeline logic. ansible-collections/ansible.netcommon extends this with two-tier delegation to both organisation-wide (ansible/ansible-content-actions) and team-specific (ansible-network/github_actions) reusable workflows. Multi-

platform CI matrices validate cross-OS compatibility for CLI tools: Atmos (Linux/Windows/macOS, 140,000+ workflow runs), Pulumi (platform-specific optimisations including Windows parallelism tuning and coverage instrumentation workarounds, 40,000+ runs), and Terragrunt (Ubuntu/macOS/Windows with build-tag-isolated scenarios, JUnit reporting across all workflows, and disk space optimisation reclaiming runner space by removing unused toolchains).

6.8. Multi-Language and Cross-Ecosystem Testing

Projects spanning multiple languages, providers, or IaC ecosystems face testing challenges not present in single-tool projects, requiring specialised practices for interoperability validation.

Cross-language validation. Multi-language platforms must ensure that components written in one language work correctly when consumed from others. Pulumi implements systematic cross-language testing where component providers implemented in Go, Python, and Node.js are each tested for consumption from all three languages. Tests validate that resource counts match across language implementations, that parent-child relationships are preserved, and that outputs propagate correctly. The pulumi-cloudflare and pulumi-gcp providers extend this by testing SDK equivalence across five to six languages: identical test scenarios execute through each language's SDK using a CI matrix strategy, ensuring that code generation produces functionally equivalent implementations and catching language-specific bugs such as TypeScript async handling errors or .NET serialisation failures. This level of interoperability testing is specific to multi-language platforms and not applicable to single-language tools, but for platforms supporting multiple language frontends, cross-language validation can be beneficial to catch serialisation bugs, type mapping errors, and SDK inconsistencies.

Dual-platform compatibility. Tool ecosystem fragmentation following the OpenTofu fork from Terraform in 2023 requires cross-platform validation. The terraform-aws-eks-cluster repository executes tests systematically against both Terraform and OpenTofu via CI matrix execution, using Linux's `update-alternatives` command to switch which binary the `terraform` command points to, with separate GitHub status checks per platform. Terragrunt extends this to orchestration tooling: integration tests run against both Terraform and OpenTofu via build tags (`tofu`), verifying that the orchestration layer correctly coordinates with both underlying IaC engines, which is particularly critical for orchestrators that must handle engine-specific behaviours (command syntax, output formats, state file handling). The automated version detection in terraform-aws-eks-cluster (`terraform-config-inspect` and `vert` for version resolution) further reduces maintenance by ensuring tests run against versions declared in the module itself, complemented by pull request labels for override testing of specific versions. Name-based secret injection conditionally injects API keys for multiple external providers (GitHub,

Opsgenie, Datadog, Spotinst) based on repository naming conventions, reducing credential configuration overhead across organisational repositories.

Interop bridge testing. The pulumi-cdk repository bridges two ecosystems (AWS CDK and Pulumi) with a dual-language testing architecture: TypeScript unit tests validate CDK-to-Pulumi translation correctness (19 Jest test files with extensive parameterised testing of CloudFormation intrinsic functions such as Fn::Join, Fn::Select, and Ref via `test.each`), while Go integration tests validate deployment of translated constructs as real AWS infrastructure. The EditDirs pattern enables multi-step lifecycle tests validating removal policies (retain-then-destroy with intermediate AWS SDK verification), resource replacement, and SSM parameter updates, suggesting that Pulumi's testing approaches can transfer to downstream libraries. Negative testing is notably strong at both levels, with parameterised error cases covering missing mappings, absent sections, and unresolvable references.

Contract testing as shipped product. The cloudformation-cli repository has a unique distinction: its contract test suite in `src/rpdk/core/contract/suite/` serves a dual role as both internal test code and a shipped product for downstream CloudFormation provider developers. Resource provider developers use this shipped suite to validate that their implementations conform to the CloudFormation contract (CREATE, READ, UPDATE, DELETE, LIST lifecycle compliance). This duality creates meta-testing requirements: the test framework itself must be extensively tested, and changes to the contract suite affect correctness guarantees for the entire CloudFormation provider ecosystem.

The repository's coverage configuration provides direct evidence of this tension: `.coveragerc` sets `fail_under = 98` (the highest threshold observed in this study) while simultaneously excluding the shipped contract suite from measurement (`omit = */contract/suite/*`). The suite is "product code" that ships to consumers but is excluded from the CLI's own coverage because it is exercised by the contract test runner against provider implementations, not by the CLI's unit tests. The `tests/test_test.py` file provides meta-testing of the `cfn test` command, validating pytest configuration generation, JSON override loading with Jinja2 templating, marker options for selective test execution, and security concerns (path traversal blocking, credential exposure prevention).

Two testing practices from this repository are worth mentioning: first, hypothesis-based property testing converts JSON Schema definitions into hypothesis strategies via `resource_generator.py`. The generator handles type-specific strategies (integers with exclusive bounds, floats excluding NaN/infinity, text with Unicode category blacklisting), string format support (ARN, URI, date-time via regex-based generation), JSON Schema combiners (`allOf` via schema merge, `oneOf`/`anyOf` via `one_of(*strategies)`), and `$ref` resolution. This means the schema simultaneously serves as the infrastructure definition and the test specification, which is a practice not observed in any other analysed repository. Second, a decorator-based assertion composition framework (`contract_asserts_commons.py`) separates *what to check* (assertion decorators) from *what to execute* (handler test functions). The `decorate()` meta-function accepts an `after` parameter

controlling pre/post-condition execution, and resource-specific assertions compose: `@response_does_not_contain_write_only_properties`, `@response_contains_primary_identifier`, and skip decorators (`@skip_not_writable_identifier`) can be stacked on a single handler test function. This differs from table-driven tests with explicit assertions (Crossplane, Terragrunt), parameterised matrices with inline assertions (pulumi-cdk), and custom assertion helpers (Pulumi's `AssertHTTPResultWithRetry`), providing cleaner separation of concerns for contract test frameworks.

The cloudformation-cli case suggests that projects building testing frameworks or compliance suites consumed by others may benefit from explicit coverage boundaries between the framework's own tests and its shipped products, to avoid circular dependencies and ensure the orchestration layer is independently validated.

6.9. Mapping Findings to Literature

Table 6.1 maps observed practices from repository analysis to recommendations from the literature review.

Table 6.1.: Comparison of observed practices with literature recommendations.

Dimension	**Literature Recommendation**	**Observed Practice**
Static analysis	Security scanning, policy validation, and linting recommended; shift-left approach advocated	24/25 at Moderate or above; all 25 enforce in CI; security scanning present in 11/25
Unit testing	Noted as limited for purely declarative IaC; more applicable where procedural logic is present	Absent in declarative HCL repositories; present where procedural logic exists (Go, Python, TypeScript); API recording and mock-based approaches observed
Integration testing	Primary validation mechanism for IaC	Dominant across all twenty-five repositories; deploy-verify-destroy cycle universally adopted
E2E testing	Validates system behaviour beyond resource existence; rarely implemented in practice	Present in platform and multi-service projects; absent or minimal in narrower-scope repositories
State evolution	Migration and idempotence testing address common defect sources	Meaningful coverage in 12/25 repositories; absent or minimal in the remaining 13; migration tooling present without automated validation in several Terraform repositories

Continued on next page

Dimension	Literature Recommendation	Observed Practice
Test fixtures and isolation	Ephemeral environments and cleanup hooks recommended for reliable CI	Cleanup hooks present in 21/25; isolation strategies range from lightweight mocking to hermetic Nix sandboxes; API recording observed as a distinctive credential-free approach
Negative testing	Robustness and security testing recommended; largely underexplored in literature	Consistently the weakest dimension; partial coverage through OPA policies, fuzz testing, contract-level validation, and parameterised error cases in a minority of repositories
CI/CD integration	Continuous testing integration advocated; cost management strategies recommended	Strong across all twenty-five repositories; 25/25 rate High; ChatOps gating, security boundaries, and flaky test detection observed as mature extensions

6.10. Practitioner Recommendations

This section translates findings from the repository analysis into actionable guidance for IaC practitioners, organised to support three questions: *What should I focus on given my project type? What can any project adopt immediately?* and *Where are the biggest opportunities for improvement?* Given that the analysed corpus includes twenty-five manually selected repositories, the sample size is too small to derive statistically grounded conclusions or broadly applicable recommendations. The guidance below should therefore be read as preliminary observations and possible directions grounded in the analysed repositories, rather than established best practices.

6.10.1. Priority Actions by Project Type

An important insight from this study is that testing strategy should be calibrated to project type rather than applied uniformly. Table 6.2 identifies the highest-impact testing investments for each project category, based on practices observed across the twenty-five repositories.

Table 6.2.: Highest-impact testing actions by project type. Actions are ordered by priority within each row.

Project Type	**Priority Actions**
Terraform modules	1. Integration tests with real cloud resources for every module variant. 2. Dual verification: validate Terraform state *and* query cloud APIs independently. 3. State evolution tests: multi-step `apply` cycles with configuration changes and add/change/destroy count assertions. 4. Automate migration script validation if upgrade documentation exists.
Terraform provider	1. Unit tests for provider diff logic with mock server for acceptance testing. 2. State migration tests for schema upgrades across provider versions. 3. Drift detection tests: create resources, modify externally, verify reconciliation.
Platform generator	1. Unit tests for rendering pipeline and configuration generation logic. 2. E2E tests validating deployed services via API and browser automation. 3. API recording (e.g., placebo) for deterministic cloud testing without credentials.
Organisation platform	1. Policy-as-code (OPA/Conftest) for systematic negative testing of misconfiguration scenarios. 2. Baseline regression testing: store known-good plans, validate current plans match. 3. Sequential E2E testing with shared state simulating incremental deployment.
IaC orchestrators	1. Unit tests for CLI logic with state management (TestKit pattern or subprocess isolation). 2. Fuzz testing for parsers and expression languages. 3. Dual-platform CI matrix testing Terraform and OpenTofu compatibility. 4. Test-quality linters enforcing `t.Parallel()`, correct assertion syntax, and cleanup.
Multi-language IaC platform	1. Cross-language interoperability tests: validate components work when consumed from all supported languages. 2. Property-based fuzz testing for state consistency across random operation sequences. 3. EditDirs-style multi-step tests with failure injection for state evolution. 4. Router-based test providers for positive and negative scenario coverage.

Continued on next page

Project Type	Priority Actions
Multi-language providers	1. Provider upgrade testing with no-replacement/no-delete assertions from baseline versions. 2. Scheduled upstream monitoring with automated PR creation for dependency tracking. 3. Multi-language SDK equivalence testing via CI matrix across all supported languages.
IaC interop bridge	1. Dual-language testing: unit tests for translation correctness, integration tests for deployment. 2. Parameterised negative testing covering missing mappings, invalid inputs, and unresolvable references. 3. Multi-step lifecycle tests validating removal policies, resource replacement, and state updates.
Ansible collections	1. Check mode testing: validate dry-run predicts changes without applying them. 2. Idempotency testing: dual execution in every integration test. 3. testinfra or equivalent for infrastructure-level assertions beyond task success. 4. Delegated workflows for organisational CI consistency.
Kubernetes control plane	1. Reconciler error path coverage across all controllers. 2. Lifecycle upgrade E2E testing: install prior version, create resources, upgrade, verify survival. 3. Nix-based hermetic CI or equivalent for reproducible builds. 4. Register fuzz targets with OSS-Fuzz for continuous robustness testing.
IaC file transformation tool	1. Golden file comparison to verify output correctness and format preservation. 2. Plugin architecture testing: compile and load custom plugins during the test run. 3. Multi-commit repository testing for tag evolution and idempotency validation. 4. Comprehensive security scanning (`gosec`, `TruffleHog`, `CodeQL`) as a CI gate.
IaC provider testing framework	1. Meta-testing: test the test orchestration layer itself (command generation, override loading, security). 2. Schema-based property testing: generate test inputs from declarative schemas. 3. Explicit coverage boundaries between framework tests and shipped test products.

6.10.2. Cross-Cutting Practices

Regardless of project type, the following practices were consistently observed in repositories with higher testing maturity and appeared applicable across project types regardless of

ecosystem or scope.

Embed static analysis in CI as a mandatory gate. All twenty-five repositories enforce static analysis on every pull request. At minimum, this means language-specific formatting (`terraform fmt`, `Black`, `Prettier`), linting (`TFLint`, `golangci-lint`, `ansible-lint`), and validation (`terraform validate`). Security scanning (`Checkov`, `Trivy`, `gosec`) was present in fewer than half of the repositories (eleven of twenty-five), representing a gap relative to the consistent adoption of linting and formatting.

Register cleanup before assertions. Test failures that leak cloud resources can create interference between runs and incur unnecessary costs. The consistent practice across mature repositories is to register cleanup immediately after resource creation (before any assertions), using `defer terraform.Destroy()`, `t.Cleanup()`, or Ansible `always:` blocks. Twenty-one of twenty-five repositories implement this.

Separate trusted from untrusted CI. Open-source projects accepting contributions must protect cloud credentials from malicious fork PRs. The consistent principle across five distinct implementations is: run linting automatically on all PRs (safe for untrusted code), but gate infrastructure tests behind manual approval (ChatOps commands, label gates, or separate CI platforms).

Use ChatOps for expensive tests. Rather than running costly cloud integration tests on every commit, six repositories implement on-demand execution via pull request comments (e.g., `/terratest`, `/run-acceptance-tests`). This balances automation benefits with cost constraints and naturally integrates with the security boundary practice above.

Validate beyond IaC tool success. Terraform reporting "Apply complete" or Ansible reporting "ok" does not guarantee infrastructure works. Nine of twenty-five repositories query external APIs after provisioning rather than relying on IaC tool success alone, suggesting that state validation may not be sufficient to confirm correct infrastructure behaviour.

6.10.3. Gap-Closing Priorities

The analysis reveals three areas where current practice falls furthest short of what literature recommends and what production infrastructure requires. These represent the highest-value improvement opportunities across the analysed repositories.

Priority 1: State evolution testing. Only twelve of twenty-five repositories implement state evolution testing, despite configuration state complexity being a documented source of hard-to-reproduce IaC defects. The practices observed suggest the following entry points for different project types: executing multiple `apply` cycles with changing variables and asserting expected add/change/destroy counts (as in Sumo Logic), running playbooks

twice and asserting the second run reports no changes (as in the Ansible collections), or testing provider upgrades from a pinned baseline version with no-replacement assertions (as in pulumi-cloudflare and pulumi-gcp).

Priority 2: Negative testing. Only ten of twenty-five repositories test error conditions at even Moderate maturity, making this the most consistent gap across the corpus. The practices observed suggest the following entry points for different project types: adding OPA/Conftest policies that test rejection of invalid inputs (as in modernisation-platform), parameterised error cases with `test.each` or table-driven tests covering missing inputs, invalid types, and boundary conditions (as in pulumi-cdk), or enabling Go native fuzz testing for parsers and interpreters (as in Terragrunt, where adding a `Fuzz*` function is sufficient for automatic CI inclusion).

Priority 3: Migration validation. Multiple repositories include migration scripts and extensive upgrade documentation but no automated tests for these procedures, meaning users may encounter migration failures in production rather than in CI. The terraform-google-network and terraform-google-kubernetes-engine repositories illustrate this gap most clearly: both include migration tooling and manual upgrade documentation covering multiple major versions, yet neither validates these procedures automatically. Adding a CI job that executes the documented upgrade path on a test configuration, as the Pulumi ecosystem does through its EditDirs pattern and provider upgrade testing, represents a practice observed to close this gap in repositories that implement it.

6.10.4. Evidence Mapping

Table 6.3 provides the complete mapping of recommendations to repository sources, literature references, and adoption counts in analysed repositories (n/25). Implementation effort varies across these recommendations, as illustrated in Figure 6.7. The upper-left quadrant (high impact, low effort) represents the most accessible starting points, while the upper-right quadrant (high impact, high effort) contains practices worth planning for as testing maturity develops.

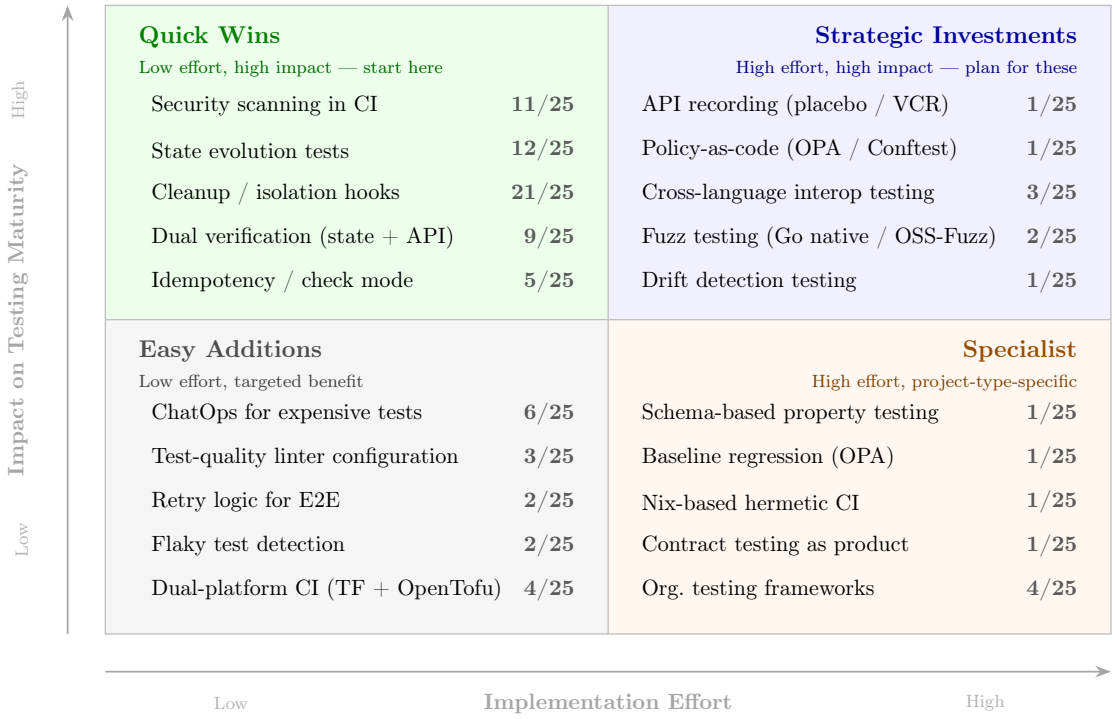

Figure 6.7.: Effort–impact classification of recommended testing practices.

Table 6.3.: Practitioner recommendations with evidence sources and adoption gaps.

Recommendation	**Repository Source**	**Literature**	**Adoption (n=25)**
	Cluster 1: Testing Strategy		
Match strategy to project type	All (varying practices)	Hasan et al.	—
Unit test procedural abstractions	Nebari, Commercetools, MoJ, Atmos, Terragrunt, Pulumi, Pulumi CF/GCP/CDK, Crossplane, CFN CLI, Ansible AWS/Netcommon, Yor	Hasan et al.	14/25
	Cluster 2: Static Analysis		
Static analysis in CI	All 25 repositories	Chiari et al., Rahman et al.	25/25
Test-quality linter enforcement	Terragrunt, Atmos, Crossplane	—	3/25
	Cluster 3: Test Isolation		
API recording for cloud testing	ansible-collections/amazon.aws	—	1/25

Continued on next page

Recommendation	Repository Source	Literature	Adoption (n=25)
Mock APIs for isolation	Nebari, Commercetools, Atmos, Pulumi, Pulumi CDK/GCP, CFN CLI, Ansible AWS, Yor	Hasan et al.	9/25
Local infrastructure for CI	Atmos, community.zabbix, Ansible Netcommon, Crossplane	Mascheroni et al.	4/25
Isolation and cleanup hooks	21 repositories	Guerriero et al., Mascheroni et al.	21/25
	Cluster 4: Negative Testing		
Policy-as-code for negative testing	modernisation-platform	Rahman et al., Joyce et al.	1/25
Parameterised negative testing	pulumi-cdk	—	1/25
Coverage-guided fuzz testing	Terragrunt, Crossplane (OSS-Fuzz)	—	2/25
	Cluster 5: State Evolution		
State evolution testing	Sumo Logic, Nebari, Commercetools, Pulumi, Pulumi CF/GCP/CDK, Crossplane, Ansible AWS, Zabbix, CAF, Yor	Drosos et al.	12/25
Provider upgrade testing	pulumi-cloudflare, pulumi-gcp	—	2/25
Baseline regression testing	terraform-azurerm-caf-enterprise-scale	—	1/25
Check mode / idempotency testing	Ansible AWS, Zabbix, GKE, Ansible Netcommon, Yor	Morris, Hummer et al.	5/25
	Cluster 6: Cross-System Validation		
Cross-system outcome validation	Sumo Logic, Nebari, Pulumi GCP/CDK, TF EKS, TF Address/Bootstrap/GKE/Network	Morris	9/25
Drift detection testing	pulumi-gcp	—	1/25
testinfra for infrastructure validation	ansible-collections/community.zabbix	—	1/25
Retry logic for E2E validation	pulumi-gcp, terraform-google-bootstrap	—	2/25
	Cluster 7: CI/CD Optimisation		
ChatOps for expensive tests	TF null-label, TF EKS/RDS/VPC, pulumi-cloudflare, pulumi-cdk	—	6/25
Security-conscious CI separation	CAF, Pulumi CDK, Terragrunt, Ansible Netcommon, Yor	—	5/25
Flaky test detection	Terragrunt, Crossplane	—	2/25
Nix-based hermetic CI	Crossplane	—	1/25

Continued on next page

Recommendation	Repository Source	Literature	Adoption (n=25)
Organisational testing frameworks	TF Address/Bootstrap/GKE/Network	—	4/25
	Cluster 8: Multi-Language		
Cross-language interoperability testing	Pulumi, Pulumi CF/GCP	—	3/25
Dual-platform compatibility	TF EKS/RDS/VPC, Terragrunt	—	4/25
Schema-based property testing	cloudformation-cli	—	1/25
Contract testing as shipped product	cloudformation-cli	—	1/25

6.11. Threats to Validity

Several limitations affect the generalisability of these findings.

Selection bias. Repositories in this thesis were selected specifically because they implement non-trivial testing practices. The findings describe practices in projects that have invested in testing, and not typical IaC projects. Most open-source IaC repositories contain limited or no testing, so conclusions about testing practices cannot be generalised to the broader population of IaC repositories.

Tool ecosystem balance. Although the expanded corpus of twenty-five repositories includes Terraform modules (10), Ansible collections (3), Pulumi repositories (4), two IaC orchestrators, a Kubernetes control plane, and a CloudFormation testing framework, Terraform-based repositories still constitute the largest single group. Testing practices for less-represented tools (Chef, Puppet, TOSCA, Bicep) may differ in ways not captured by this study. The inclusion of Pulumi, Ansible, Crossplane, and cloudformation-cli mitigates this bias compared to a purely Terraform-focused analysis, but the findings should not be assumed to generalise to all IaC ecosystems equally.

Snapshot analysis. Repositories were analysed at a single point in time. Testing practices can evolve over time, so the findings represent a snapshot rather than the current state of these repositories or how their testing evolved.

Qualitative assessment. Maturity ratings (High, Moderate, Low) involve judgment about what constitutes each level. Different assessors might reach different conclusions. The evidence presented in each repository evaluation provides transparency, but subjectivity remains inherent in qualitative evaluation.

Limited negative evidence. The absence of certain practices (e.g., security scanning) could reflect undocumented practices, private CI configurations, or deliberate decisions not captured in the public repository. The analysis can only assess what is visible in the codebase.

Sample size. Twenty-five repositories provide analysis of cross-cutting practices across twelve project types, four IaC ecosystems, and three cloud providers. However, statistical generalisation is not possible with this sample size. The practices identified should be treated as well-evidenced observations across a diverse corpus rather than definitive conclusions about all IaC testing practice.

7. Conclusion

This thesis investigated how Infrastructure as Code testing is practised in real-world open-source projects and to what extent academic recommendations are reflected in actual implementations. Through a systematic analysis of twenty-five repositories spanning twelve project types, four IaC ecosystems, and three cloud providers, the study addressed three objectives: identifying recommended practices through literature review, assessing alignment between recommendations and practice, and deriving practical guidelines for IaC testing.

7.1. Summary of Contributions

The thesis makes three primary contributions. First, it presents an eight-dimension evaluation framework grounded in academic literature that provides a structured, reproducible approach for assessing IaC testing maturity. The framework covers the full testing spectrum from static analysis through end-to-end validation, including cross-cutting concerns such as state evolution, negative testing, and CI/CD integration. Second, it provides twenty-five detailed project evaluations documented in the appendix. Third, the cross-repository synthesis identifies eight thematic clusters with concrete practitioner recommendations based on the project type, bridging the gap between academic recommendations and practitioner-oriented guidance.

7.2. Key Findings

The analysis reveals significant variation in testing maturity. Integration testing is present in all twenty-five repositories, static analysis is almost universal (24/25 at Moderate or above, with all 25 enforcing checks on every PR), and CI/CD integration is consistently at High maturity across the corpus (25/25). These represent established practices where the analysed projects show consistent adoption.

However, substantial gaps remain in three areas. Negative testing is the weakest dimension across the corpus: while a few repositories test specific error paths, no repository in the corpus systematically tests for cloud API failures, quota exhaustion, or permission denials, though five distinct partial approaches were identified: policy-as-code, parameterised error cases, contract-level validation, reconciler error paths, and fuzz testing. State evolution testing is present in only twelve of twenty-five repositories despite migration failures being a documented source of IaC defects, with a notable ecosystem disparity: all Pulumi and Ansible repositories implement it, while the majority of Terraform modules do not. Migration tooling without automated validation emerged as a recurring gap,

with multiple repositories providing upgrade scripts or documentation but no CI-based verification that these procedures work correctly.

The analysis also identified several testing practices not identified in the academic literature reviewed for this thesis: API recording for deterministic cloud testing, three-tier static analysis enforcement spectrum, drift detection testing, baseline regression with OPA/Conftest, Nix-based hermetic CI, schema-based property testing from JSON Schema definitions, and decorator-based assertion composition for contract test frameworks.

A central observation is that testing strategy appears to vary systematically by project type rather than following a single template. In the analysed repositories, unit test investment tends to be higher in projects with more procedural code, E2E testing maturity tends to be higher in projects with broader scope, and specialised practices such as cross-language interoperability testing, provider upgrade testing, and dual-platform compatibility were observed only in specific project categories. This observation underpins the project-type-specific practitioner recommendations presented in the Results chapter.

7.3. Future Work

Several directions emerge from this research. First, the evaluation framework and findings could inform the development of automated tools that assess IaC testing maturity from repository metadata and CI configurations, reducing the manual effort currently required for such analysis. Second, a longitudinal study tracking how testing practices evolve within repositories over time would complement this snapshot analysis, particularly for understanding how projects adopt new testing dimensions as they mature. Third, practitioner validation through surveys or interviews would assess whether the recommended priority actions align with real-world constraints such as team size, budget, and organisational context. Fourth, extending the corpus to include IaC tools not well-represented in this study, particularly Chef, Puppet, TOSCA, Bicep, and CDK for Terraform, would improve generalisability. Finally, the novel practices identified (API recording, drift detection testing, schema-based property testing) warrant deeper investigation to understand their effectiveness and the conditions under which they provide the most value.

Bibliography

Artač, M., Borovšak, T., Di Nitto, E., Guerriero, M., and Tamburri, D. A. (2017). Devops: Introducing infrastructure-as-code. In *Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 497–498, Buenos Aires, Argentina. IEEE.

Cankar, M., Petrović, N., Costa, J. P., Černivec, A., Antić, J., Martinčič, T., and Štepec, D. (2023). Security in devsecops: Applying tools and machine learning to verification and monitoring steps. In *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering (ICPE '23 Companion)*, Coimbra, Portugal. ACM.

Chiari, M., De Pascalis, M., and Pradella, M. (2022). Static analysis of infrastructure as code: a survey. In *IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*, pages 218–225. IEEE.

Drosos, G.-P., Sotiropoulos, T., Alexopoulos, G., Mitropoulos, D., and Su, Z. (2024). When your infrastructure is a buggy program: Understanding faults in infrastructure as code ecosystems. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA2):2490–2520.

Guerriero, M., Garriga, M., Tamburri, D. A., and Palomba, F. (2019). Adoption, support, and challenges of infrastructure-as-code: Insights from industry. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 580–589. IEEE.

Hasan, M. M., Bhuiyan, F. A., and Rahman, A. (2020). Testing practices for infrastructure as code. In *Proceedings of the 1st ACM SIGSOFT International Workshop on Languages and Tools for Next Generation Testing*, LANGETI '20. ACM.

Hummer, W., Rosenberg, F., Oliveira, F., and Eilam, T. (2013). Testing idempotence for infrastructure as code. In *Middleware 2013: ACM/IFIP/USENIX 14th International Middleware Conference*, volume 8275 of *Lecture Notes in Computer Science*, pages 386–405. Springer.

Joyce, J. E., Maheshwari, S., and Kothamasu, D. (2025). Secure by design: Strategic approaches to infrastructure as code. In *2025 Third International Conference on Networks, Multimedia and Information Technology (NMITCON)*, pages 1–6. IEEE.

Khan, S., Kabanov, I., Hua, Y., and Madnick, S. (2022). A systematic analysis of the capital one data breach: Critical lessons learned. *ACM Transactions on Privacy and Security*, 26(1).

Kohler, J. (2023). Comparison of an automated FaaS- to a SaaS-based ETL process in a cloud environment based on an infrastructure as code approach. In *2023 International Conference on Cloud Computing Technologies and Applications (CloudTech)*, pages 1–7. IEEE.

Maliekal, K. S. (2025). Automated testing in kubernetes: A comprehensive framework for code quality and deployment assurance. *International Research Journal of Modernization in Engineering Technology and Science*, 7(5):2927–2941.

Mascheroni, M. A., Irrazábal, E., and Rossi, G. (2021). Continuous testing improvement model. In *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*. IEEE.

Morris, K. (2021). *Infrastructure as Code: Dynamic Systems for the Cloud Age*. O'Reilly Media, 2nd edition.

Nagineni, S. (2025). Integrated development lifecycle: QA engineers and DevOps teams in collaborative agile workflow. *Sarcouncil Journal of Multidisciplinary*, 5(8):159–174.

Nedeltcheva, G. N., Xiang, B., Niculut, L., and Benedetto, D. (2023). Challenges towards modeling and generating infrastructure-as-code. In *Companion of the 14th ACM/SPEC International Conference of Performance Engineering (ICPE '23 Companion)*, pages 189–193, Coimbra, Portugal. ACM.

Rahman, A., Mahdavi-Hezaveh, R., and Williams, L. (2019a). A systematic mapping study of infrastructure as code research. *Information and Software Technology*, 108:65–77.

Rahman, A., Parnin, C., and Williams, L. (2019b). The seven sins: Security smells in infrastructure as code scripts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 164–175. IEEE.

Sokolowski, D. and Salvaneschi, G. (2023). Towards reliable infrastructure as code. In *2023 IEEE 20th International Conference on Software Architecture Companion (ICSA-C)*, pages 318–321, L'Aquila, Italy. IEEE.

Sokolowski, D., Spielmann, D., and Salvaneschi, G. (2023). Creed for speed: Comprehensive infrastructure as code testing. In *Proceedings of CONFLANG 2023, co-located with SPLASH 2023*, Cascais, Portugal. ACM.

Sokolowski, D., Spielmann, D., and Salvaneschi, G. (2024). Automated infrastructure as code program testing. *IEEE Transactions on Software Engineering*, 50(6):1585–1599.

Thota, R. C. (2024). Enhancing infrastructure as code (IaC) with automated validation for reliable and error-free deployments. *Journal of Artificial Intelligence Research and Applications*, 4(1):827–847.

War, A., Rawass, A. A., Kabore, A. K., Samhi, J., Klein, J., and Bissyande, T. F. (2025). Detection of security smells in IaC scripts through semantics-aware code and language processing. Submitted 23 September 2025.

A. Detailed Repository Evaluations

This appendix contains detailed evaluations of all twenty-five repositories analysed in this thesis. The first eight repositories (Sections A.1–A.8) are presented in full detail, with dedicated subsections for each testing dimension, architectural diagrams, and test suite coverage tables. The remaining seventeen repositories (Sections A.9–A.25) follow a condensed format comprising repository overview, testing architecture, test suite coverage table, dimension ratings table, and overall assessment. This structure reflects the practical length constraints of a master's thesis.

A.1. Evaluation of IaC Testing Practices in terraform-sumologic-sumo-logic-integrations

A.1.1. Repository Overview

The `terraform-sumologic-sumo-logic-integrations`[1] repository provides Terraform modules for integrating AWS services with the Sumo Logic observability platform. The modules provision AWS resources (S3 buckets, Lambda functions, IAM roles, CloudTrail configurations) along with corresponding Sumo Logic collectors and sources, enabling automated log and metric ingestion pipelines.

The repository was selected because it demonstrates cross-platform integration testing: tests must validate not only AWS resource creation but also that data flows correctly from AWS into Sumo Logic. This represents a more complex testing challenge than single-platform IaC modules.

A.1.2. Testing Architecture

Static checks and security scanning are orchestrated through GitHub Actions workflows (`.github/workflows/validate-terraform.yml`), which run `terraform validate` and Checkov scans in parallel jobs. A shell script (`run_tests.sh`) provides additional local orchestration for running `terraform fmt` and TFLint.

Integration tests use Terratest (Go) and are organised in `terratest/aws` and `terratest/sumologic`. Each module's tests follow a JSON-driven approach where expected outputs are defined in files like `TestWithDefaultValues.json`, which enables declarative test specification.

Helper functions in `terratest/common` (e.g., `assertsumologic.go`) provide reusable utilities for API validation through direct HTTP calls to verify Sumo Logic resource

[1] `https://github.com/SumoLogic/terraform-sumologic-sumo-logic-integrations`

creation. These helpers support integration testing rather than isolated unit testing, as no unit tests exist for the helper logic itself.

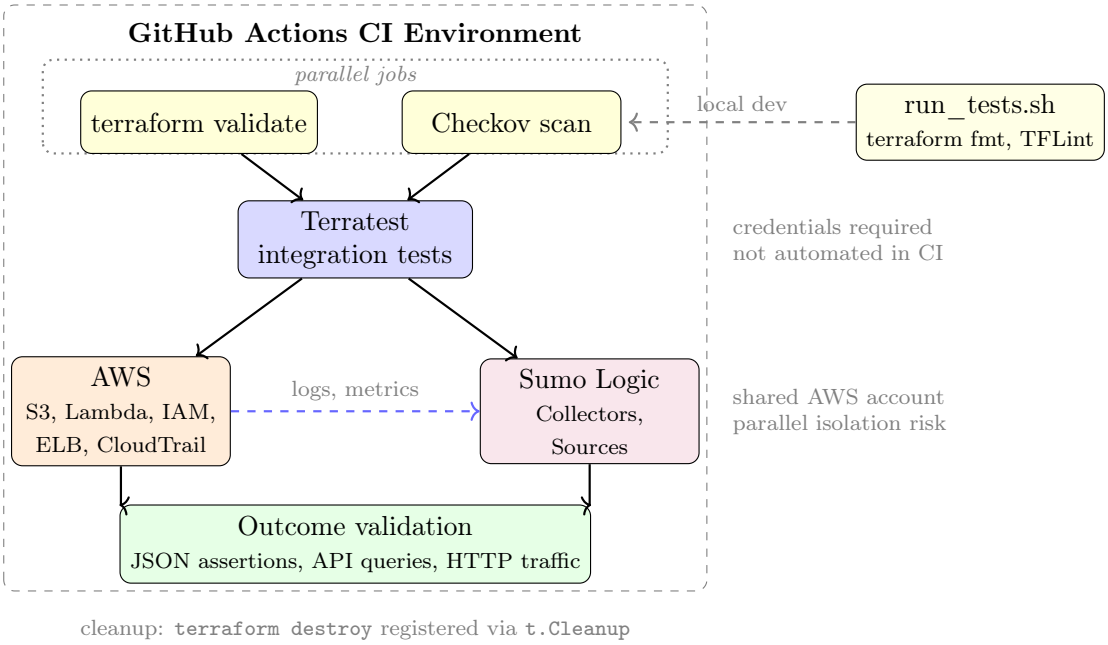

Figure A.1.: Sumo Logic cross-platform testing architecture

A.1.3. Test Suite Coverage

Table A.1 summarises the test suite coverage.

Table A.1.: Test coverage in `terraform-sumologic-sumo-logic-integrations`.

Module Category	What Is Tested	Validation Approach
AWS integrations (CloudTrail, ELB, CloudWatch)	Resource creation, IAM permissions, S3 bucket configuration	Terraform output assertions
Sumo Logic sources	Collector and source creation, endpoint configuration	API queries via helpers
Data pipelines	Log flow from AWS to Sumo Logic	Cross-platform outcome validation
Update scenarios	Configuration changes, tag updates, flag toggling	Multi-step TestUpdates

A.1.4. Evaluation Against IaC Testing Dimensions

Static analysis and syntax checks. The repository implements comprehensive static analysis. GitHub Actions workflows (`validate-terraform.yml`) run `terraform validate` and Checkov security scans in parallel jobs across all modules. The `run_tests.sh` script provides additional checks including `terraform fmt` and TFLint for local development. This combination covers syntax validation, formatting, linting, and security scanning.

Logic-level unit tests. No unit tests exist for helper functions or module logic. The helpers in `terratest/common` (e.g., `assertsumologic.go`) perform direct HTTP calls to external APIs, making them unsuitable for isolated unit testing without mocking. This reflects a tendency observed across several repositories where IaC helper code is validated indirectly through integration tests rather than tested in isolation.

Integration testing. Integration tests cover all modules. Each module has Terratest suites that:

- Deploy real AWS and Sumo Logic resources.
- Validate Terraform outputs against expected JSON files.
- Query the Sumo Logic API to confirm resource creation.
- Execute cleanup via `terraform destroy`.

The JSON-driven approach (e.g., `TestWithDefaultValues.json`) enables declarative test specification and clear output contracts.

End-to-end and outcome-based testing. This repository implements strong outcome validation. Tests verify not just resource existence but actual data flow: the ELB test sends HTTP traffic and confirms access logs reach both S3 and Sumo Logic. This cross-platform validation catches integration failures that Terraform state alone cannot detect.

State evolution and migration. The repository stands out with state evolution testing. `TestUpdates` functions in every module execute multi-step apply cycles, modifying configuration flags and resource names across sequential apply calls and asserting exact add/change/destroy counts at each step. Separate parameterised tests cover the full range of deployment scenarios: fresh resource creation, integration with fully pre-existing resources, and mixed states where some resources exist and others do not.

Test fixtures and isolation. Tests use temporary directories via `CopyTerraformFolderToTemp` and register cleanup hooks via `t.Cleanup`. Resource names include unique identifiers to avoid collisions. However, tests share AWS and Sumo Logic accounts, which creates potential for interference in parallel execution.

Negative and robustness testing. No explicit negative tests exist. Tests do not validate behaviour with invalid inputs, missing permissions, or API failures. Retry logic handles eventual consistency but does not exercise failure scenarios.

CI/CD integration. CI/CD integration uses GitHub Actions. The `validate-terraform.yml` workflow runs `terraform validate` and Checkov scans in parallel jobs, using modern CI practices (`setup-terraform`, `checkout@v4`). This decouples validation from the shell script and enables parallel execution of linting and security scans. Integration tests require AWS and Sumo Logic credentials, constraining full automation.

Table A.2 lists the ratings across all eight dimensions.

Table A.2.: Evaluation of `terraform-sumologic-sumo-logic-integrations` against IaC testing dimensions.

Dimension	Evidence	Assessment
Static analysis	`terraform validate` and Checkov security scanning via GitHub Actions; no `terraform fmt` check or TFLint configuration found.	Moderate–High.
Logic-level unit tests	No unit tests; helpers validated indirectly through integration tests.	Low.
Integration testing	Terratest suites with JSON-driven expected outputs and API validation.	High.
End-to-end / outcome-based	Cross-platform data flow validation (AWS to Sumo Logic); HTTP traffic tests.	High.
State evolution	`TestUpdates` in every module; multi-step apply cycles with assertions.	High.
Test fixtures and isolation	Temp directories, cleanup hooks, unique identifiers; shared accounts limit isolation.	Moderate–High.
Negative / robustness	No error scenario or invalid input tests.	Low.
CI/CD integration	GitHub Actions with parallel validation and security scanning; integration tests need credentials.	High.

A.1.5. Overall Assessment

What distinguishes this repository is its outcome-based validation: rather than stopping at Terraform state checks, tests verify that logs and metrics actually flow from AWS through the Sumo Logic ingestion pipeline, including HTTP traffic tests for ELB access logs. State evolution testing via `TestUpdates` in every module is another strength, with multi-step apply cycles that toggle flags, modify tags, and assert expected resource change counts. Static analysis includes Checkov security scanning alongside `terraform validate`. The absence of unit tests for helper logic and negative testing for error scenarios are the main gaps, and the shared AWS and Sumo Logic account model limits isolation for parallel execution.

A.2. Evaluation of IaC Testing Practices in terraform-provider-commercetools

A.2.1. Repository Overview

The `terraform-provider-commercetools`[2] repository implements a Terraform provider for the Commercetools e-commerce platform. Unlike Terraform modules that compose existing providers, this project implements the provider itself. It translates Terraform

[2] `https://github.com/labd/terraform-provider-commercetools`

resource definitions into Commercetools API calls and manages state synchronisation between Terraform and the platform.

The repository was selected because provider development involves different testing concerns than module development: unit testing of type conversion logic, acceptance testing against real or mocked APIs, and state migration testing for schema upgrades. This provides insights into testing practices for a distinct category of IaC artifact.

A.2.2. Testing Architecture

Unit tests validate internal logic (error handling, state migration, utility functions) in isolation. Acceptance tests use the Terraform provider testing framework to validate resource lifecycle operations against the Commercetools API.

A key architectural feature is the use of a mock server in CI. The GitHub Actions workflow (`tests.yaml`) spins up a service container using `labdigital/commercetools-mock-server` on port 8989, enabling tests to run in an isolated environment without requiring real Commercetools credentials. This decouples CI execution from external dependencies.

The provider combines the Terraform Plugin Framework with legacy SDK components through a multiplexed architecture, requiring tests to cover both implementation paths.

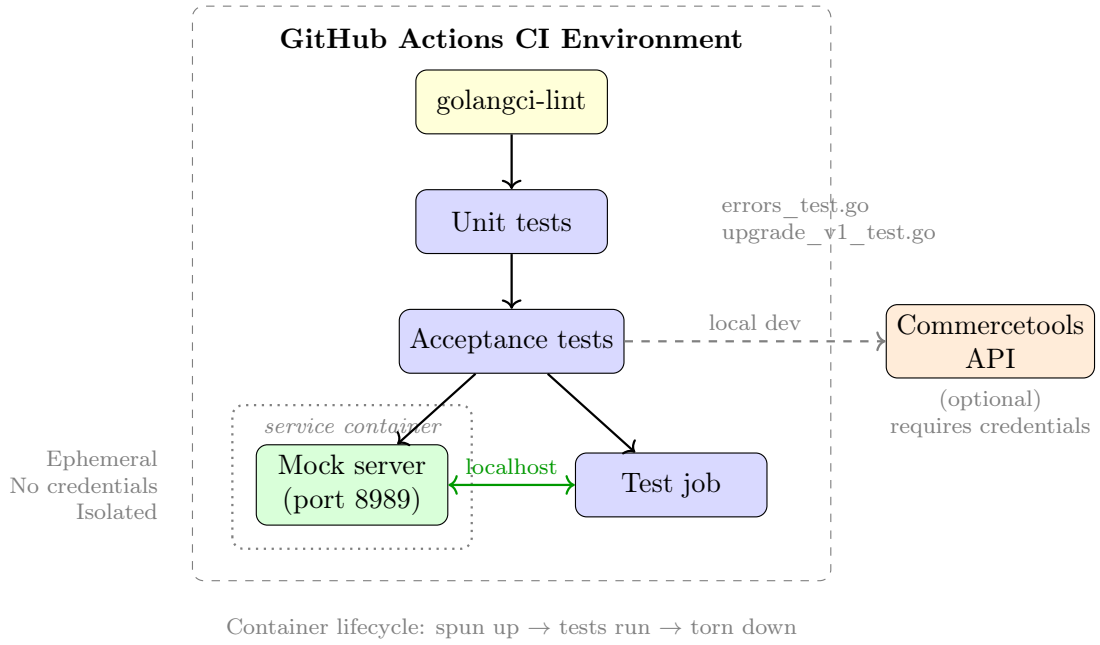

Figure A.2.: Commercetools ephemeral test environment

A.2.3. Test Suite Coverage

Table A.3 summarises the test suite coverage.

Table A.3.: Test coverage in `terraform-provider-commercetools`.

Test Category	What Is Tested	Location
Unit tests	Error classification, state migration (V0→V1), utility functions, diff computation	`internal/utils/errors_test.go`, `internal/resources/project/upgrade_v1_test.go`, `internal/resources/subscription/model_test.go`
Acceptance tests	Resource CRUD operations, attribute validation, API synchronisation, import verification	`internal/**/resource_*_test.go`
State migration	Schema upgrade logic across multiple resources	`internal/resources/project/upgrade_v1_test.go`, `internal/resources/subscription/upgrade_v0_test.go`

A.2.4. Evaluation Against IaC Testing Dimensions

Static analysis and syntax checks. The `tests.yaml` workflow runs `golangci-lint-action@v9` with a curated configuration defined in `.golangci.yml`, which explicitly disables all default linters and enables seventeen specific linters covering correctness (`errcheck`, `govet`), complexity (`cyclop`), and safety (`forcetypeassert`). However, the linting step is configured as non-blocking: `continue-on-error: true` and `-issues-exit-code=0` mean lint findings are reported but do not prevent merging. Terraform-specific linting is not included, though this is less relevant for provider code than for modules. No security scanning tool is present.

Logic-level unit tests. The repository includes pure Go unit tests for:

- Error classification logic (`internal/utils/errors_test.go`)
- State migration from V0 to V1 (`internal/resources/project/upgrade_v1_test.go`)
- Type conversion and expansion/flattening utilities, including write-only secret handling across all destination types (`internal/resources/subscription/model_test.go`)
- Diff computation logic (`internal/resources/subscription/model_test.go`)

This reflects the provider's nature as a Go program with substantial logic beyond declarative resource definitions.

Integration testing. Acceptance tests validate resource lifecycle operations using the Terraform provider testing framework. Tests exercise create, read, update, and delete operations, verifying that Terraform state matches API state. Import verification is also tested via `ImportState: true` with `ImportStateVerify`, confirming that existing resources can be brought under Terraform management. The mock server in CI enables

these tests to run without external dependencies, though tests can also run against a real Commercetools project for full validation.

End-to-end and outcome-based testing. Testing focuses on control-plane validation, confirming that resources are created with correct attributes rather than application-level workflows. The `examples/` directory contains basic resource definitions but no complex workflow scripts. This is appropriate for a provider, where the concern is API interaction correctness rather than business process validation.

State evolution and migration. The repository includes explicit state migration testing. The `upgrade_v1_test.go` file in `internal/resources/project/` and `upgrade_v0_test.go` in `internal/resources/subscription/` each test their respective state upgrade functions, verifying that existing state is correctly transformed when the schema changes. Known values are preserved, and newly introduced fields are correctly marked as unknown rather than assigned arbitrary defaults. This practice across multiple resources indicates systematic attention to upgrade path correctness, addressing a common source of provider upgrade failures. State migration testing of this kind was not observed in the Terraform module repositories in this corpus.

Test fixtures and isolation. The CI pipeline uses a mock server architecture for isolation. The GitHub Actions workflow spins up `labdigital/commercetools-mock-server` as a service container, allowing tests to run against a local, isolated environment. This eliminates dependencies on shared cloud projects and external credentials for CI execution. Tests can also run against real Commercetools projects for additional validation when credentials are available.

Negative and robustness testing. Some negative testing exists: `TestAccSubscription_sqs_failure` deliberately configures an invalid SQS destination and asserts that the apply fails with a specific error message via `ExpectError`. However, systematic testing of API failure scenarios, permission errors, or rate limiting is not present.

CI/CD integration. GitHub Actions workflows run linting, unit tests, and acceptance tests against the mock server in a single job. The repository also uses Dependabot with Changie (`dependabot-changie.yaml`) to automate dependency updates and changelog generation, maintaining dependency hygiene without manual intervention.

Table A.4 lists the ratings across all eight dimensions.

Table A.4.: Evaluation of `terraform-provider-commercetools` against IaC testing dimensions.

Dimension	Evidence	Assessment
Static analysis	`golangci-lint-action@v9` with curated `.golangci.yml` (17 linters, `disable-all: true`); non-blocking (`continue-on-error: true`, `-issues-exit-code=0`); no security scanning.	Moderate.
Logic-level unit tests	Unit tests for error handling, state migration, write-only secret handling across all destination types, diff computation logic.	Moderate–High.
Integration testing	Acceptance tests via Terraform provider framework; mock server in CI; import state verification.	High.
End-to-end / outcome-based	Control-plane validation only; no application workflow tests.	Low–Moderate.
State evolution	Upgrade tests in `internal/resources/-project/upgrade_v1_test.go` and `internal/resources/subscription/upgrade_v0_-test.go`; known values preserved; new fields marked unknown.	Moderate–High.
Test fixtures and isolation	Mock server as CI service container eliminates external dependencies; isolated execution per job.	High.
Negative / robustness	`TestAccSubscription_sqs_failure` with `ExpectError`; no systematic API failure or permission error tests.	Low–Moderate.
CI/CD integration	GitHub Actions with linting, unit and acceptance tests, mock server; Dependabot + Changie for automated dependency maintenance.	High.

A.2.5. Overall Assessment

The repository combines unit tests for internal Go logic (error handling, state migration, write-only secret handling, diff computation) with a mock server architecture that enables acceptance testing without real Commercetools credentials. State migration is tested across multiple resources (`project` and `subscription`), covering both value preservation and correct handling of newly introduced fields. Static analysis uses a curated `golangci-lint` configuration, though findings are non-blocking. The testing scope is appropriately limited to control-plane validation rather than end-to-end business workflows. Gaps include the non-enforced linting configuration and limited systematic negative testing.

A.3. Evaluation of IaC Testing Practices in the Nebari Platform

A.3.1. Repository Overview

Nebari[3] is an open-source platform that deploys data science environments on Kubernetes across multiple cloud providers (AWS, GCP, Azure, DigitalOcean) and on-premises infrastructure. Nebari generates Terraform and Helm configurations from a Python-based configuration system, then orchestrates their deployment to produce a complete data science platform including JupyterHub, Dask, monitoring, and authentication services.

The repository was selected because it represents platform-level IaC: testing must validate not only generated configurations but also that deployed infrastructure provides functional services. This requires testing at multiple layers, from configuration rendering to live service validation.

A.3.2. Testing Architecture

Nebari implements a five-layer testing architecture:

1. **Static analysis**: Linting and formatting enforced via pre-commit hooks, including Ruff, Black, isort, Codespell, Typos, and `terraform_fmt`. Type checking is configured via mypy in `pyproject.toml`, though it runs as non-blocking (`continue-on-error: true`) in CI.

2. **Unit tests**: Validate configuration logic, schema validation, CLI behaviour, and upgrade registry using mocked cloud and Kubernetes APIs.

3. **Integration tests**: Verify rendered Terraform and Helm artifacts across cloud providers without deployment.

4. **Deployment tests**: Validate live service APIs against a running Nebari cluster, covering JupyterHub, Dask, Grafana, Keycloak, Loki, NFS, and conda-store.

5. **E2E tests**: Playwright-based browser automation against live Nebari instances, validating user workflows.

The pre-commit configuration includes a CI section with monthly autoupdate scheduling, ensuring static checks are integrated into automated maintenance workflows rather than relying solely on local developer execution.

[3] `https://github.com/nebari-dev/nebari`

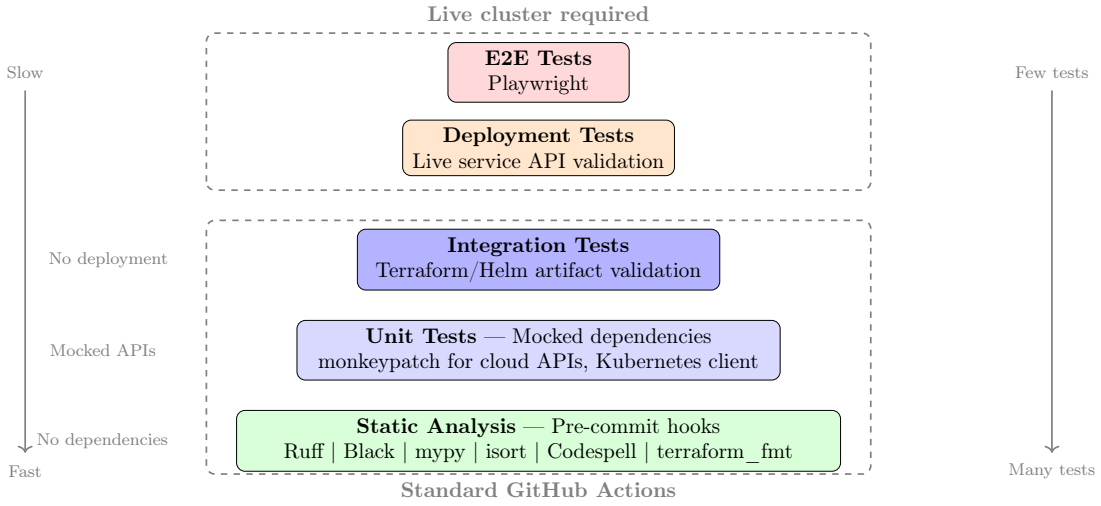

Figure A.3.: Nebari testing pyramid

A.3.3. Test Suite Coverage

Table A.5 summarises the test suite coverage.

Table A.5.: Test coverage in Nebari.

Layer	What Is Tested	Approach
Static analysis	Code style, formatting, imports, spelling, Terraform formatting, type hints	Pre-commit (Ruff, Black, isort, mypy, Codespell, terraform_fmt)
Unit tests	Schema validation, CLI commands, upgrade registry, cloud config	Mocked Kubernetes and cloud APIs via monkeypatch
Integration tests	Terraform/Helm artifact rendering across providers	Validation without deployment
Deployment tests	Live service APIs: JupyterHub, Dask, Grafana, Keycloak, Loki, NFS, conda-store	API calls against running Nebari cluster
E2E tests	User login, notebook execution, service accessibility	Playwright against live instances

A.3.4. Evaluation Against IaC Testing Dimensions

Static analysis and syntax checks. Static analysis runs through pre-commit hooks. The configuration specifies Ruff for linting, Black for formatting, isort for import sorting, Codespell and Typos for spell checking, and `terraform_fmt` for formatting the Terraform files Nebari generates. Type checking is configured via mypy in `pyproject.toml`; the corresponding `typing.yaml` workflow runs mypy in CI as non-blocking (`continue-on-error: true`). The pre-commit CI integration includes monthly autoupdates, ensuring the tool-

chain remains current. Trivy security scanning runs on every PR in both configuration and filesystem scan modes, uploading results to the GitHub Security tab.

Logic-level unit tests. The repository includes extensive unit tests with mocked dependencies across 20 test files in `tests/tests_unit/`:

- Schema validation and configuration rendering
- CLI command handling (`test_cli_plugin.py`)
- Upgrade registry logic (`test_upgrade.py`)
- Cloud provider configuration validation

Tests use `monkeypatch` to mock Kubernetes clients and cloud APIs, and `unittest.mock.patch` to simulate plugin loading and version retrieval. This enables fast, deterministic testing without cloud credentials.

Integration testing. Integration tests validate that configuration rendering produces correct Terraform and Helm artifacts across cloud providers. Tests verify artifact structure and content without actual deployment, catching configuration errors before expensive provisioning.

End-to-end and outcome-based testing. Playwright-based E2E tests in `tests/tests_e2e/playwright` validate user workflows against live Nebari instances:

- Authentication flows (login/logout)
- JupyterHub notebook execution
- Service accessibility (Argo, monitoring, user management)
- conda-store namespace and environment management

A separate `tests/tests_deployment/` layer validates live service APIs directly, covering JupyterHub role synchronisation with Keycloak, Dask gateway availability, Grafana, Loki log aggregation, and NFS storage. This API-level validation layer sits between artifact rendering and full browser automation.

State evolution and migration. The upgrade registry is tested in `test_upgrade.py`, validating the framework mechanics for version migrations using a `MockKeycloakAdmin` class to simulate the authentication service. Tests focus on the upgrade logic itself rather than actual state migrations through deployment cycles.

Test fixtures and isolation. Isolation is achieved through multiple mechanisms:

- Cloud APIs mocked via `monkeypatch`
- Kubernetes client mocked for upgrade tests
- Temporary directories for configuration rendering
- Session-scoped tokens for deployment and E2E tests

Unit and integration tests run without cloud credentials; deployment and E2E tests require a running Nebari cluster.

Negative and robustness testing. Negative testing at the CLI validation layer is systematic. The `test_cli_validate.py` file uses a fixture-based approach: six YAML files named with the `.error.` convention — covering invalid project names, extra inputs, invalid Kubernetes versions, and unsupported authentication types — are automatically discovered and parameterised at test collection time. Additional parameterised functions cover invalid environment variable overrides (three cases) and missing cloud credentials across AWS, Azure, and GCP (five cases). Negative testing of deployment failures or cloud API errors during provisioning is not present.

CI/CD integration. The repository has 15 GitHub Actions workflows. Static analysis runs via pre-commit on every PR with monthly autoupdates. Unit tests run across a four-version Python matrix (3.10–3.13) with coverage reporting. Local integration and E2E tests run on PRs via `test_local_integration.yaml`, which provisions a local Kubernetes cluster, deploys a full Nebari instance, runs deployment pytests, Playwright browser automation, and conda-store user journey tests. Cloud-provider integration tests (AWS, Azure, GCP) run on weekly schedules. Trivy security scanning runs on every PR. Concurrency management with `cancel-in-progress` prevents resource waste on rapid PR updates.

Table A.6 lists the ratings across all eight dimensions.

Table A.6.: Evaluation of Nebari against IaC testing dimensions.

Dimension	**Evidence**	**Assessment**
Static analysis	Pre-commit with Ruff, Black, isort, Codespell, Typos, terraform_fmt; mypy for type checking (non-blocking); Trivy security scanning on every PR.	High.
Logic-level unit tests	20 test files covering schema, CLI, upgrades, cloud config; mocked cloud and Kubernetes APIs via monkeypatch.	High.
Integration testing	Artifact rendering validation across cloud providers without deployment.	High.
End-to-end / outcome-based	Playwright tests for user workflows; parameterised across service configurations; deployment tests validate live service APIs across eight services.	High.
State evolution	Upgrade registry tested against historical config files using MockKeycloakAdmin; covers configuration format migration between versions but not infrastructure state preservation across deployment upgrade cycles.	Moderate.
Test fixtures and isolation	Mocked APIs, temp directories, session-scoped tokens; deployment and E2E tests require live cluster.	High.
Negative / robustness	Fixture-based validation error tests (6 error YAML files); parameterised invalid environment variable and missing credential tests; no deployment failure tests.	Moderate.
CI/CD integration	15 GitHub Actions workflows; pre-commit CI with monthly autoupdates; unit tests across Python 3.10–3.13 matrix; local integration and Playwright E2E on every PR; cloud-provider tests on weekly schedules; Trivy on PRs; concurrency management.	High.

A.3.5. Overall Assessment

Nebari's testing reflects the platform's nature as a generator that produces and deploys IaC rather than being IaC directly. A five-layer architecture spans pre-commit static analysis, mocked unit tests, artifact rendering validation, live service API testing across eight services, and Playwright browser automation for end-user workflows. Negative testing at the CLI validation layer is systematic, using fixture files and parameterised functions to cover invalid configurations and missing credentials. Gaps include non-enforced mypy type checking and the absence of deployment failure testing.

A.4. Evaluation of IaC Testing Practices in terraform-null-label

A.4.1. Repository Overview

The `terraform-null-label`[4] module is a widely adopted Terraform utility that provides deterministic, composable naming and tagging conventions for cloud resources. Rather than provisioning infrastructure, the module performs functional transformations: normalising strings, composing resource identifiers, generating tag maps, and encoding naming conventions.

The repository was selected because it represents a different category of IaC artifact: a utility module that provisions no resources but encodes critical organisational logic. Most IaC testing research focuses on cloud deployments and resource lifecycles whereas this module expands the analysis to functional correctness of reusable abstractions.

A.4.2. Testing Architecture

The repository contains a single Terratest-based Go test (`TestExamplesComplete`) located in `test/src/examples_complete_test.go`. The test suite has a dedicated `go.mod` file in `test/src/`, isolating test dependencies from the main module. The repository has also adopted the Atmos toolchain (`atmos.yaml`) for infrastructure orchestration, reflecting Cloud Posse's newer architectural approach.

The test workflow is:

1. Copy example configuration to a temporary folder via `CopyTerraformFolderToTemp`.
2. Execute `terraform init` and `terraform apply`.
3. Read Terraform outputs (IDs, tag maps, normalised contexts, truncated identifiers).
4. Compare each output against hardcoded expected values via `assert.Equal`.
5. Clean up via `defer terraform.Destroy`.

[4] `https://github.com/cloudposse/terraform-null-label`

Although technically integration tests (invoking Terraform end-to-end), the validation target is entirely functional. Because no external systems are provisioned, the suite is fast, hermetic, and reproducible.

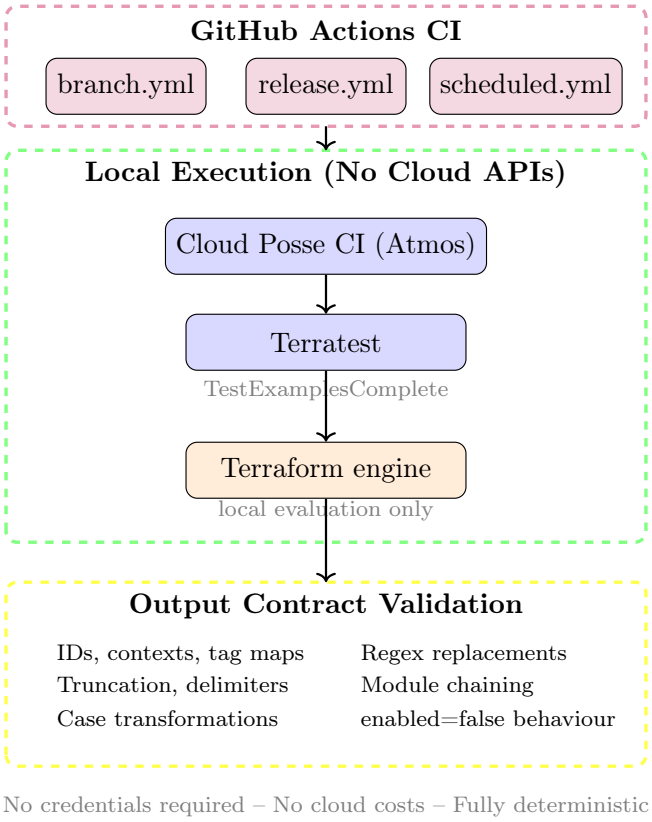

Figure A.4.: `terraform-null-label` functional testing

A.4.3. Test Suite Coverage

The test validates functional transformations across different input combinations:

- Case transformations (upper, lower, title, none)
- Character replacement via regular expressions
- Delimiter insertion and suppression
- ID construction and hashing behaviour
- Tag-generation semantics
- Inheritance and propagation in chained module calls

- Truncation behaviour and length limits
- Disabled module behaviour (`enabled = false` producing empty output)

A.4.4. Evaluation Against IaC Testing Dimensions

Static analysis and syntax checks. Static analysis is enforced through the shared CloudPosse CI workflow (`shared-ci-terraform.yml`), which runs three jobs on every pull request: `terraform fmt` formatting checks via `dflook/terraform-fmt`, TFLint with the AWS ruleset via `reviewdog/action-tflint`, and BATS-based structural checks covering input and output descriptions, module pinning, and provider pinning. No security scanning tools (Checkov, Trivy, tfsec) are included, though these are less relevant for a utility module with no cloud resource definitions.

Logic-level unit tests. The test suite validates functional transformations through 62 `assert.Equal` calls covering:

- String normalisation and case handling
- Delimiter semantics and regex-based replacement
- Tag map generation from input attributes
- Module chaining and context propagation
- Truncation rules respecting length limits

This is effectively unit testing implemented through Terraform's evaluation engine rather than direct function calls, as the module has no testable logic outside of Terraform.

Integration testing. Tests exercise Terraform's module evaluation, variable interpretation, and output propagation. This represents integration testing within the IaC engine: validating that the module behaves correctly when processed by Terraform, without involving cloud APIs.

End-to-end and outcome-based testing. Not applicable. The module produces no infrastructure, so there is no deployed system to validate. This absence reflects the module's nature, not a testing gap.

State evolution and migration. Not applicable. The module has no stateful behaviour — it computes outputs from inputs without maintaining state between runs.

Test fixtures and isolation. Isolation is achieved through several mechanisms:

- `CopyTerraformFolderToTemp` copies the example configuration to a temporary directory, preventing state file pollution of the source tree.
- `defer terraform.Destroy` ensures cleanup runs even if the test fails midway.
- No environment variables or credentials required.
- No external dependencies.
- Test dependencies isolated via dedicated `go.mod` in `test/src`.

Negative and robustness testing. No tests for malformed inputs, invalid delimiters, empty attributes, or edge cases. Only happy-path scenarios are covered. This is a gap, as a utility module used across many configurations would benefit from validation of boundary and invalid input handling.

CI/CD integration. The repository uses GitHub Actions with four workflow files. All three primary workflows — `branch.yml` (pull requests), `release.yml` (published releases), and `scheduled.yml` (daily at 03:00 UTC) — delegate entirely to shared reusable workflows in the central CloudPosse `.github` repository. A fourth workflow, `chatops.yml`, triggers Terratest runs on demand via `/terratest` pull request comments. Because the actual CI steps are defined in the external shared workflows, the specific validation steps cannot be fully evaluated from this repository alone.

Table A.7 lists the ratings across all eight dimensions.

Table A.7.: Evaluation of `terraform-null-label` against IaC testing dimensions.

Dimension	Evidence	Assessment
Static analysis	`terraform fmt`, TFLint (AWS ruleset), and BATS structural checks enforced via shared CloudPosse reusable CI workflow; no security scanning tools.	Moderate.
Logic-level unit tests	62 `assert.Equal` calls validating string processing, naming, tags, truncation, chaining, and `enabled=false` behaviour.	High.
Integration testing	`terraform apply` + output evaluation; tests module within Terraform engine without cloud APIs.	Moderate–High.
End-to-end / outcome-based	Not applicable; module produces no infrastructure.	N/A.
State evolution	Not applicable; no stateful behaviour.	N/A.
Test fixtures and isolation	Temporary directories via `CopyTerraformFolderToTemp`; deferred destroy; isolated `go.mod`; no credentials required.	High.
Negative / robustness	No malformed-input or edge-case tests; only happy-path scenarios present.	Low.
CI/CD integration	Four GitHub Actions workflows delegating to shared CloudPosse CI; every PR runs `terraform fmt`, BATS structural checks, and TFLint (AWS ruleset); chatops trigger for on-demand Terratest runs; daily scheduled runs.	High.

A.4.5. Overall Assessment

As a utility module encoding organisational naming conventions rather than provisioning resources, functional correctness is the primary testing concern. The single `TestExamplesComplete` function acts as a contract specification, encoding each naming rule as a Terratest assertion against real `terraform apply` output — 62 assertions in total. Isolation is strong: temporary directories, deferred cleanup, and isolated test dependencies ensure reproducibility without credentials. The main gaps are the absence of negative testing for malformed inputs and the delegation of CI steps to external shared workflows, which limits visibility into the full static analysis posture.

A.5. Evaluation of IaC Testing Practices in modernisation-platform

A.5.1. Repository Overview

The `modernisation-platform`[5] repository is the main IaC codebase for the UK Ministry of Justice, managing a multi-account AWS platform. It contains Terraform configurations for core networking, logging, security, and shared services, together with governance and security policies implemented using Open Policy Agent (OPA).

The repository was selected because it combines infrastructure testing with policy-as-code validation in a production platform. Unlike module or provider repositories, this represents organisation-wide IaC with two complementary testing layers: Terratest for Terraform modules and Conftest/OPA for policy enforcement. This addresses a gap in IaC testing literature where governance testing receives less attention than execution-level tests.

A.5.2. Testing Architecture

- **Static analysis:** Security scanning via GitHub Actions workflows including TFLint, Trivy, and Checkov, plus Regal linting for Rego policies
- **Policy-as-code:** OPA/Rego policies with unit tests validating governance rules for environment definitions, network configurations, and collaborator access
- **Terraform integration tests:** Terratest suites validating core environment configurations via `refresh` and `plan` against live infrastructure

[5] `https://github.com/ministryofjustice/modernisation-platform`

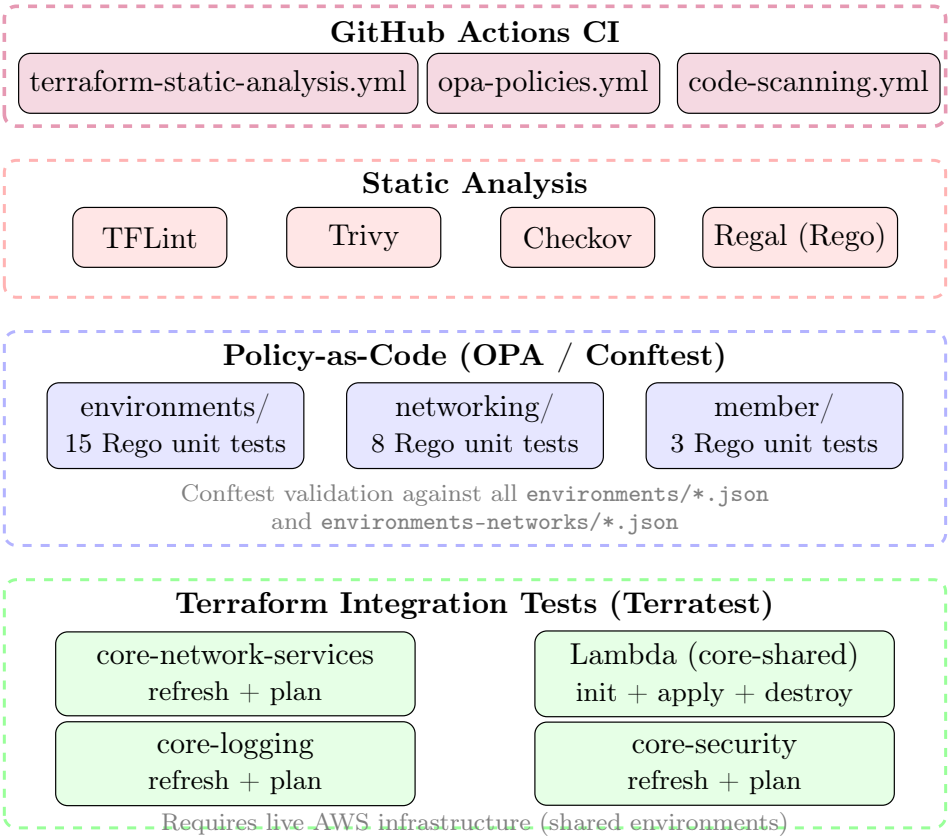

Figure A.5.: `modernisation-platform` testing architecture

A.5.3. Test Suite Coverage

Table A.8 summarises the test suite coverage.

Table A.8.: Test coverage in the modernisation-platform repository.

Layer	Scope	Validation Style
Static analysis	Terraform linting, security vulnerabilities, compliance checks, Rego linting	TFLint, Trivy, Checkov via GitHub Actions; Regal for Rego
Terraform (core VPC environments)	Route tables, subnet counts, transit gateway IDs, firewall ARNs, VPC CIDRs	Terratest `refresh`/`plan` + output assertions
Terraform (Lambda)	Instance-scheduler function name and execution result	Terratest `apply`/`destroy` with dedicated fixture

Table A.8.: Test coverage in the modernisation-platform repository. (continued)

Layer	Scope	Validation Style
OPA (environment policies)	Required keys, business-unit names, email validation, access levels, CNI flag	Rego unit tests with synthetic inputs (15 tests)
OPA (network policies)	CIDR consistency, subnet structures, naming conventions	Rego unit tests + Conftest validation (8 tests)
OPA (member policies)	Filename character/length rules, allowed environment names	Rego unit tests (3 tests)

A.5.4. Evaluation Against IaC Testing Dimensions

Static analysis and syntax checks. Static analysis is enforced through two GitHub Actions workflows. The `code-scanning.yml` workflow includes a dedicated TFLint job using `terraform-linters/setup-tflint`. The `terraform-static-analysis.yml` workflow orchestrates a broader security suite via the `modernisation-platform-github-actions/terraform-static-analysis` shared action, running Trivy for vulnerability scanning (with `tfsec_trivy: trivy`, indicating Trivy has replaced TFSec as the security scanner), Checkov for compliance checks, and TFLint for Terraform linting. The `opa-policies.yml` workflow additionally runs Regal, a dedicated linter for Rego code itself, before executing Conftest tests. Three distinct scan modes are configured: a changed-files scan on PRs, a full-codebase scan on manual dispatch, and a scheduled full scan on weekday mornings with Slack failure notification.

Logic-level unit tests. Rego unit tests provide genuine unit-level coverage for policy logic. Tests use the `with input as {...}` construct to inject synthetic inputs directly into policy rules, asserting on specific error messages in the `deny` set. A representative test injects a synthetic JSON object with an invalid `business-unit` value and asserts that the deny set contains the exact error message naming the invalid value and listing all allowed values. Policy and test files are paired: `environment-definitions.rego` with `environment-definitions_test.rego`, and `core-vpc.rego` with `core-vpc_test.rego`. Tests cover helper functions (`has_field`, type checks, regex validation), required-key presence, allowed enumeration values, email format validation, and the `critical-national-infrastructure` boolean field. The 26 Rego unit tests across three policy packages constitute genuine unit-level testing for governance logic.

Integration testing. Terratest suites validate Terraform modules for four core environments. Three tests (`core-logging`, `core-network-services`, `core-security`) follow

the same approach: run `terraform refresh` then `terraform plan` against existing infrastructure, then read named outputs via `terraform.Output` and assert on their values using `assert.Equal` or `assert.Contains`. This validates infrastructure shape without reprovisioning.

The fourth test (`TestInstanceSchedulerLambda` in `core-shared-services`) uses a full `InitAndApply`/`Destroy` cycle against a dedicated test fixture in `test/test_terraform/`, which invokes the Lambda function and asserts on the HTTP status code of the response.

End-to-end and outcome-based testing. Tests validate infrastructure shape (VPCs, subnets, gateways, Lambda functions) but do not exercise application-level workloads or user traffic. This is control-plane validation rather than full system testing, appropriate for a shared government platform where application-layer testing is the responsibility of tenant teams.

State evolution and migration. No explicit state migration tests exist. Operational workflows handle monitoring and maintenance (`reusable-scheduled-baselines.yml`, `access-keys-monitoring.yml`), but these are not designed for testing infrastructure code evolution in a controlled test environment. No dedicated test scenarios exist for validating infrastructure state transitions.

Test fixtures and isolation. The approach is pragmatic for a shared government platform. Terraform tests operate against shared, long-lived environments using `refresh`/`plan` only, avoiding resource creation and destruction costs. The Lambda test is the exception: it uses a dedicated `test_terraform/` fixture with a full `apply`/`destroy` cycle. Rego tests are fully hermetic, executing pure logic on synthetic inputs with no AWS access. Recreating core multi-account AWS infrastructure for each test run would be impractical in this context, making the shared-environment model an intentional design choice rather than a gap.

Negative and robustness testing. Negative testing is strong in the policy layer. The 26 Rego unit tests are predominantly negative tests: missing required keys (`test_empty_file`, `test_no_environments`, `test_no_tags`), empty values (`test_empty_values`), out-of-range enumeration values (`test_unexpected_business_units`, `test_unexpected_access`), invalid formats (`test_invalid_email`), and type mismatches (`test_cidr_types`, `test_option_types`). Terraform tests focus on happy-path validation, but the policy layer compensates by catching configuration errors before they reach Terraform execution.

CI/CD integration. Full CI/CD automation runs through GitHub Actions. Static analysis workflows (`code-scanning.yml`, `terraform-static-analysis.yml`) run TFLint, Trivy, and Checkov. OPA tests are automated via `run-opa-tests.sh` and `conftest verify` in `opa-policies.yml`, which also runs Regal lint on the policy directory. The OPA workflow triggers on changes to `.json`, `.rego`, or the test runner script, keeping

CI execution targeted. Some Terraform tests execute in CI, though full automation is constrained by reliance on shared environments that cannot be safely reprovisioned in a pipeline.

Table A.9 lists the ratings across all eight dimensions.

Table A.9.: Evaluation of modernisation-platform against IaC testing dimensions.

Dimension	Evidence	Assessment
Static analysis	TFLint, Trivy, and Checkov via GitHub Actions workflows; Regal linting for Rego policies.	High.
Logic-level unit tests	26 Rego unit tests across three policy packages; direct invocation of policy rules via `with input as {...}`; helper functions tested.	Moderate–High.
Integration testing	Terratest for four core environments; `refresh`/`plan` + output assertions; Lambda test uses full `apply`/`destroy`.	High.
End-to-end / outcome-based	Infrastructure shape validation; no application workload tests.	Low–moderate.
State evolution	No explicit migration tests.	Low.
Test fixtures and isolation	Shared environments for Terraform (refresh/plan only); dedicated fixture for Lambda test; hermetic Rego tests.	Moderate.
Negative / robustness	Extensive in Rego tests (26 negative scenarios); OPA/Conftest policies enforce governance; limited negative coverage in Terraform tests.	Moderate.
CI/CD integration	GitHub Actions with static analysis (TFLint, Trivy, Checkov) and OPA automation (Regal lint + Conftest verify); targeted triggers on JSON/Rego changes.	High.

A.5.5. Overall Assessment

The most distinctive aspect of this repository is its policy-as-code layer: OPA/Conftest policies with paired Rego unit tests covering positive and negative scenarios alike (missing keys, invalid values, type mismatches). This makes it one of the few repositories in the corpus with systematic negative testing, achieved through a testable policy abstraction

rather than at the Terraform level where invalid configurations surface as cryptic provider failures. Static analysis is comprehensive, with TFLint, Trivy, and Checkov complemented by Regal linting for Rego policies. Terraform integration testing is pragmatic: three tests use `refresh`/`plan` against shared environments to avoid provisioning costs, while the Lambda test uses a full `apply`/`destroy` cycle against a dedicated fixture. The main gap is the absence of explicit state evolution testing.

A.6. Evaluation of IaC Testing Practices in ansible-collections/amazon.aws

A.6.1. Repository Overview

The `ansible-collections/amazon.aws`[6] repository is the official Ansible collection for automating AWS service management, maintained by the Ansible AWS community and certified by Red Hat. It provides 141 modules and plugins for managing AWS resources including EC2, S3, RDS, VPC, IAM, Lambda, and CloudFormation, serving as the foundation for AWS automation in enterprise Ansible environments.

This repository was selected because it represents an Ansible collection providing declarative resource management through playbooks rather than code-based IaC (Terraform, Pulumi), offering insight into testing practices within the configuration management paradigm.

A.6.2. Testing Architecture

Static analysis uses ten tools (black, isort, flynt, flake8, pylint, ruff, mypy, ansible-lint, yamllint, ansible-test sanity) across a multi-version matrix (Ansible 2.17–2.20, Python 3.10–3.14). Unit tests employ a hybrid approach: pure mocking via `unittest.mock.MagicMock` and pytest `monkeypatch` for module logic, and the `placebo` library for recording and replaying AWS API calls as JSON fixtures, enabling credential-free testing against real API response structures. Integration tests cover 160 targets provisioning real AWS resources via the `ansible-test` framework, with check mode validation ensuring dry-run operations produce no side effects.

[6] `https://github.com/ansible-collections/amazon.aws`

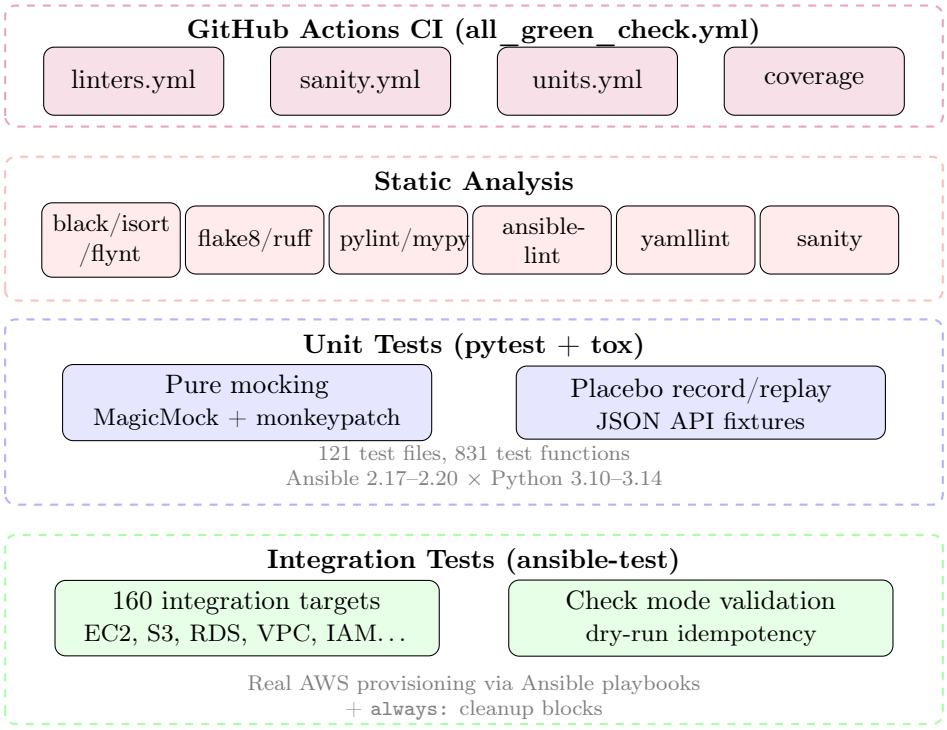

Figure A.6.: `ansible-collections/amazon.aws` testing architecture

A.6.3. Test Suite Coverage

Table A.10 summarises the test suite coverage.

Table A.10.: Test suite coverage for ansible-collections/amazon.aws.

Layer	Scope	Approach
Static analysis	Code style, formatting, type hints, YAML syntax, Ansible best practices	10 tools: black, isort, flynt, flake8, pylint, ruff, mypy, ansible-lint, yamllint, ansible-test sanity
Unit tests (pure mocking)	Module logic, parameter validation, control flow, error handling	`unittest.mock.MagicMock` + pytest `monkeypatch`; 121 test files, 831 test functions
Unit tests (placebo)	AWS API interactions, request/response handling	Placebo library recording/replaying real API calls as JSON fixtures; `time.sleep` patching for retry loops

Table A.10.: Test suite coverage for ansible-collections/amazon.aws. (continued)

Layer	Scope	Approach
Integration tests	160 AWS service targets: EC2, S3, RDS, VPC, IAM, Lambda, CloudFormation	Real AWS provisioning via Ansible playbooks; `ansible-test` framework; check mode validation; `always:` cleanup blocks
Sanity tests	Module syntax, imports, documentation, version compatibility	`ansible-test sanity` across Ansible 2.17–2.20 × Python 3.10–3.14

A.6.4. Evaluation Against IaC Testing Dimensions

Static analysis and syntax checks. Static analysis is enforced through three GitHub Actions workflows delegating to shared reusable workflows from the `ansible-network/github_actions` library. `linters.yml` runs the full tox linting suite (black, isort, flynt, flake8, pylint, ruff, mypy, ansible-lint, yamllint). `sanity.yml` runs `ansible-test sanity` across the full version matrix. The shared-workflow delegation approach means the collection inherits upstream improvements to the CI toolchain automatically.

Logic-level unit tests. Unit tests employ a hybrid mocking strategy across 121 test files containing 831 test functions. For module logic and control flow, tests use `unittest.mock.MagicMock` and pytest `monkeypatch`. For AWS API interactions, the `placebo` library intercepts `boto3_conn` (the collection's AWS connection factory) and replays pre-recorded JSON API responses, enabling credential-free testing against real API response structures rather than hand-written mock data. The `maybe_sleep` fixture patches `time.sleep` to return immediately during replay, ensuring retry and polling logic is exercised without wall-clock delay. Tests are run across a version matrix via tox (Ansible 2.17–2.20 × Python 3.10–3.14).

Integration testing. Integration tests cover 160 targets provisioning real AWS resources via the `ansible-test` framework. Each target is an Ansible playbook role with an `aliases` file declaring its CI classification. Targets use `always:` task blocks for cleanup, and select targets include explicit check mode runs confirming that dry-run operations produce no side effects. Integration tests run on Ansible's external CI infrastructure rather than the repository's own GitHub Actions workflows.

End-to-end and outcome-based testing. No end-to-end scenario tests exist that chain multiple modules together into a complete infrastructure workflow. Integration tests validate individual module behaviour in isolation.

State evolution and migration. No systematic framework for testing infrastructure state transitions exists. Some unit tests exercise multi-step state sequences (stop, start, terminate practices) and tag updates, but these are not structured as migration tests.

Test fixtures and isolation. Placebo fixtures provide strong isolation for unit tests: AWS credentials are never required and API responses are deterministic. Integration tests use unique resource naming and `always:` cleanup blocks, but run against shared AWS environments without containerised isolation.

Negative and robustness testing. Negative testing is systematic. Dedicated error handler test files exist for each major service area (autoscaling, RDS, waiter, retry, cloud retry), using `pytest.raises` and `botocore.exceptions.ClientError` to verify correct error propagation, graceful suppression of expected errors, and retry backoff behaviour. Parametrized test tables cover multiple operation-name/error-code combinations. Of 121 unit test files, 70 contain explicit error assertions.

CI/CD integration. `all_green_check.yml` orchestrates CI across PRs and pushes to `main` and `stable-*` branches. A final `all_green` job asserts that all required jobs succeeded via a Python script, acting as a strict gate. Coverage is measured on every run using `tox -e ansible2.20-py314-with_constraints`, with the resulting `coverage.xml` uploaded to SonarCloud for quality tracking.

A.6.5. Overall Assessment

The `placebo` record/replay approach is the most distinctive testing technique in this repository: it captures real AWS API responses as JSON fixtures and replays them for deterministic, credential-free unit testing, bridging the gap between pure mocking (which may diverge from real API behaviour) and full integration testing (which requires live credentials). Static analysis is broad with ten complementary tools, and integration coverage spans 160 AWS module targets with check mode validation. Unit test coverage is substantial at 121 test files and 831 test functions across a systematic error handler implementation. Gaps include the absence of end-to-end scenario testing, containerised isolation for integration tests, and a state evolution framework.

Table A.11.: Evaluation of ansible-collections/amazon.aws against IaC testing dimensions.

Dimension	Evidence	Assessment
Static analysis	10 tools configured: black, isort, flynt, flake8, pylint, ruff, mypy, ansible-lint, yamllint, ansible-test sanity; multi-version matrix.	High.

Table A.11.: Evaluation of amazon.aws against IaC testing dimensions. (continued)

Dimension	Evidence	Assessment
Logic-level unit tests	Hybrid approach: pure mocking (MagicMock + monkeypatch) for logic; placebo library for recording/replaying AWS API calls; 121 test files, 831 test functions.	Moderate–High.
Integration testing	160 test targets provisioning real AWS resources; ansible-test framework; check mode validation; cleanup via `always:` blocks.	High.
End-to-end / outcome-based	Infrastructure control-plane validation only; no application workflows or multi-module scenario chains.	Low.
State evolution	Some multi-step state sequences in unit tests (stop/start/terminate, tag updates); no systematic migration framework.	Moderate–High.
Test fixtures and isolation	Placebo fixtures for unit tests (no AWS access); unique naming and `always:` cleanup for integration; no containerisation.	Moderate.
Negative / robustness	Dedicated error handler test files per service area; parametrized ClientError/WaiterError cases; 70 of 121 test files contain explicit error assertions.	Moderate–High.
CI/CD integration	GitHub Actions with delegated shared workflows; multi-version matrix (Ansible 2.17–2.20, Python 3.10–3.14); `all_green` gate job; SonarCloud coverage integration.	High.

A.7. Evaluation of IaC Testing Practices in Atmos

A.7.1. Repository Overview

Atmos[7] is a Terraform orchestration framework implemented as a command-line tool written in Go. The repository provides a CLI tool and Go library for managing component

[7] `https://github.com/cloudposse/atmos`

orchestration, stack inheritance, and workflow automation across multi-account, multi-cloud environments.

This repository was selected because it represents testing practices for IaC tooling rather than IaC configurations themselves, providing insight into how orchestration frameworks validate CLI behaviour, configuration processing logic, and integration with backend tools. As infrastructure tooling written in Go, Atmos's testing approach differs fundamentally from the Terraform-based and Ansible-based repositories previously analysed.

A.7.2. Testing Architecture

Static analysis uses a custom `lintroller` plugin with six AST-based rules enforcing test isolation practices (discussed below), alongside golangci-lint with 30 enabled linters (including gosec, errorlint, depguard, forbidigo, revive, tparallel) and pre-commit hooks (`go-fumpt`, `go-build-mod`, `go-mod-tidy`).

Unit tests span over 1,000 `_test.go` files across the `pkg/`, `internal/`, `cmd/`, and `errors/` packages, comprising 11,304 test functions and 90 benchmark functions. A `TestKit` wrapper (defined in `cmd/testkit_test.go`) provides automatic `RootCmd` state cleanup before and after each test, preventing global CLI state from leaking between tests. Mocks are generated with `go.uber.org/mock` using a registry pattern for dependency injection. Coverage is enforced at 80% on new and changed lines via Codecov.

Acceptance tests in the `tests/` package execute the full compiled Atmos binary as a subprocess using `os.Args` manipulation, validating complete CLI workflows. The suite includes 412 golden snapshot files (in `tests/snapshots/`) covering the expected output of `atmos describe`, `atmos terraform`, and related commands across structured fixture scenarios in `tests/fixtures/scenarios/`.

Integration tests use `LocalStack` for S3 backend validation, `miniredis` for Redis state across hook invocations, and `K3s` for Helmfile operations. CI comprises nine specialised jobs running across Linux, Windows, and macOS, accumulating over 140,000 total workflow runs.

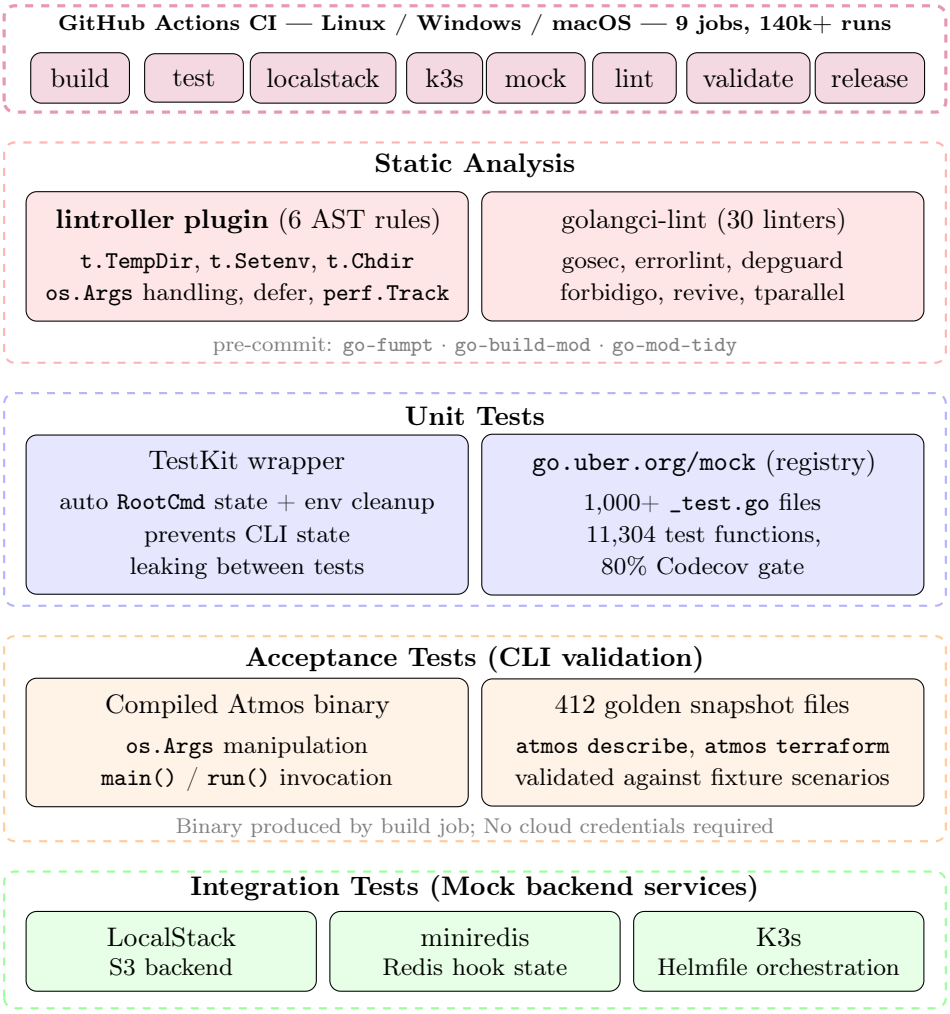

Figure A.7.: Testing architecture of Atmos: four-tier pipeline from AST-level static analysis through mock-backend integration testing, with no cloud deployment required at any tier.

A.7.3. Test Suite Coverage

Table A.12 summarises the test suite coverage.

Table A.12.: Test suite coverage for Atmos.

Layer	Scope	Approach
Static analysis	Test isolation practices, architecture compliance, security, error handling, code quality	Custom `lintroller` plugin (6 AST-based rules) + golangci-lint (30 linters: gosec, errorlint, depguard, forbidigo, revive, tparallel) + pre-commit hooks
Unit tests	Config parsing, path resolution, templates, auth, CLI, schema validation, store integration	TestKit for RootCmd isolation + `go.uber.org/mock` with registry pattern; 1,000+ `_test.go` files, 11,304 test functions; 80% coverage gate (Codecov)
Acceptance tests	Complete CLI workflows, command output correctness, multi-command cross-invocation state	Direct `main()`/`run()` execution with `os.Args` manipulation; 412 golden snapshot files in `tests/snapshots/`
Integration tests	Backend operations, Redis hook state, Helmfile orchestration, multi-platform compatibility	miniredis for Redis; LocalStack for S3 backend; K3s for Helmfile; Linux/Windows/macOS CI matrix

A.7.4. Evaluation Against IaC Testing Dimensions

Static analysis and syntax checks. Static analysis operates at three complementary levels. At the toolchain level, golangci-lint enables 30 linters enforcing security (gosec), error wrapping (errorlint), import restrictions (depguard), forbidden identifiers (forbidigo), code complexity (revive, gocognit, cyclop), and parallel test conventions (tparallel). At the project-specific level, a custom `lintroller` plugin encodes six AST-based rules using `golang.org/x/tools/go/analysis`: `TSetenvInDeferRule` detects `t.Setenv()` inside `defer` blocks (which panics because the test framework has already finished); `OsSetenvInTestRule` requires `t.Setenv()` over `os.Setenv()` in test files (so the framework auto-restores original values); `OsMkdirTempInTestRule` requires `t.TempDir()` over `os.MkdirTemp()` (so temporary directories are cleaned up automatically); `OsChdirInTestRule` requires `t.Chdir()` over `os.Chdir()` in tests (ensuring directory state is restored after each test); `OsArgsInTestRule` prevents direct `os.Args` mutation in test files (which can leak state between tests); and `PerfTrackRule` enforces the use of `defer perf.Track()` in public functions per the project's performance instrumentation

guidelines. The plugin integrates into golangci-lint as a module plugin and runs in pre-commit hooks. At the commit level, pre-commit hooks enforce formatting (`go-fumpt`), compilation (`go-build-mod`), and module hygiene (`go-mod-tidy`).

Logic-level unit tests. The repository contains over 1,000 `_test.go` files with 11,304 test functions and 90 benchmark functions spanning the `pkg/`, `internal/`, `cmd/`, and `errors/` packages. The `TestKit` wrapper (in `cmd/testkit_test.go`) captures and restores the global `RootCmd` state and environment variables before and after each test, preventing CLI state from leaking between test cases. Mocks are generated with `go.uber.org/mock` using a `go:generate` registry pattern, enabling dependency injection without concrete implementations. Coverage is tracked via Codecov with an 80% target on new and changed lines, and coverage documentation is maintained in code comments. The test suite validates configuration parsing, path resolution, template processing, authentication, CLI argument handling, schema validation, and store integrations.

Integration testing. Acceptance tests in the `tests/` package execute the full compiled Atmos binary as a subprocess by manipulating `os.Args` and calling `main()` or `run()`, exercising complete CLI workflows end-to-end. The suite includes 412 golden snapshot files covering the expected output of commands such as `atmos describe component`, `atmos describe stacks`, and `atmos terraform`, validated against structured fixture scenarios in `tests/fixtures/scenarios/`. Root-level integration tests (for example `main_hooks_-and_store_integration_test.go`) test cross-invocation state by running sequences of `run()` calls and asserting state persisted across invocations via miniredis. The CI pipeline runs these tests across Linux, Windows, and macOS, accumulating over 140,000 workflow runs.

End-to-end and outcome-based testing. End-to-end coverage is limited to backend integration: LocalStack provides a mock AWS S3 backend for validating Terraform state storage, and K3s provides a local Kubernetes cluster for Helmfile orchestration tests. No tests deploy real infrastructure or validate provisioning outcomes; this is appropriate scope for an orchestration tool whose primary responsibility is configuration resolution and command delegation rather than resource lifecycle management.

State evolution and migration. No migration or upgrade tests exist. There are no version compatibility matrices, no schema evolution tests validating that stack configurations from an older Atmos version parse correctly under a newer one, and no tests for hierarchical configuration inheritance changes across versions. This is a notable gap for a tool whose core value proposition is managing complex stack inheritance chains.

Test fixtures and isolation. The `lintroller` plugin actively enforces isolation practices at the AST level, requiring `t.TempDir()`, `t.Setenv()`, `t.Chdir()`, and `t.Cleanup()` throughout the test suite. The `TestKit` wrapper provides structured RootCmd state cleanup. Backend isolation uses in-process mock services (miniredis, LocalStack) rather

than shared external infrastructure. Fixture scenarios in `tests/fixtures/scenarios/` provide self-contained input configurations for acceptance tests.

Negative and robustness testing. Error path tests exist with inline documentation and table-driven error validation for configuration parsing failures and invalid command arguments. However, boundary and edge case testing is limited: there are no tests for configuration corruption recovery, resource exhaustion, or deployment failure propagation. Negative coverage is adequate for a CLI tool but does not reach the systematic error handler testing observed in repositories such as `ansible-collections/amazon.aws`.

CI/CD integration. The nine-job CI pipeline (build, test, docker, localstack, k3s, mock, lint, validate, release) enforces a strict dependency graph: the build job produces platform-specific binaries that all downstream test jobs download before execution, ensuring tests run against the compiled artefact rather than source code. The release job requires all other jobs to pass. Coverage is uploaded to Codecov on every run with the 80% patch target enforced. The multi-platform matrix (Linux/Windows/macOS) with over 140,000 workflow runs provides strong cross-OS compatibility confidence.

A.7.5. Overall Assessment

Atmos demonstrates mature testing practices for a CLI tool: an extensive unit test suite of over 1,000 `_test.go` files with 11,304 test functions, a custom AST-level linting plugin that prevents the most common test isolation bugs before code is committed, golden snapshot testing across 412 expected outputs, and multi-platform CI across three operating systems. The `lintroller` plugin in particular represents a distinctive engineering investment: encoding six project-specific testing rules as AST-based analysers ensures that isolation violations are caught automatically rather than through code review, representing the middle tier of the three-tier enforcement spectrum documented in Chapter 5.

The primary gaps are state evolution testing and boundary robustness. For an orchestration tool managing complex hierarchical configuration inheritance, the absence of tests validating schema compatibility across versions or stack inheritance changes across upgrades represents a meaningful coverage gap. Negative testing is functional but not systematic.

Table A.13.: Evaluation of Atmos against IaC testing dimensions.

Dimension	**Evidence**	**Assessment**
Static analysis	Custom `lintroller` plugin with 6 AST-based rules enforcing test isolation (`t.TempDir()`, `t.Setenv()`, `t.Chdir()`, `os.Args` handling, defer usage, performance instrumentation); golangci-lint with 30 linters (gosec, errorlint, depguard, forbidigo, revive, tparallel); pre-commit hooks; multi-layered enforcement.	High.
Logic-level unit tests	1,000+ `_test.go` files; 11,304 test functions; 90 benchmark functions; TestKit for automatic RootCmd state cleanup; `go.uber.org/mock` with registry pattern; table-driven tests; 80% coverage target (Codecov); validates config, paths, auth, CLI, schemas, templates, stores.	High.
Integration testing	Acceptance tests calling `main()`/`run()` with `os.Args` manipulation; 412 golden snapshot files; miniredis for Redis state; multi-platform CI (Linux/Windows/macOS); LocalStack for S3 backend; K3s for Helmfile; 9 specialised CI jobs; 140,000+ workflow runs.	High.
End-to-end / outcome-based	Backend integration only (LocalStack S3, K3s Helmfile); no infrastructure deployment testing; validates tool invocation correctness, not provisioning outcomes; appropriate scope for an orchestration tool.	Low–moderate.
State evolution	No migration or upgrade testing; no version compatibility matrices; no schema evolution tests; significant gap for a hierarchical configuration tool.	Low.
Test fixtures and isolation	TestKit automatic cleanup; `t.TempDir()`, `t.Setenv()`, `t.Chdir()`, `t.Cleanup()` enforced by `lintroller`; miniredis/LocalStack/K3s mock services; structured fixture scenarios.	High.

Table A.13.: Evaluation of Atmos against IaC testing dimensions. (continued)

Dimension	Evidence	Assessment
Negative / robustness	Error path tests with inline documentation; table-driven error validation; limited boundary and edge case coverage; no corruption recovery or resource exhaustion tests.	Low–moderate.
CI/CD integration	9-job workflow (build, test, docker, localstack, k3s, mock, lint, validate, release); multi-platform matrix; Codecov 80% patch target; pre-commit integration; artefact management; 140,000+ workflow runs.	High.

A.8. Evaluation of IaC Testing Practices in ansible-collections/community.zabbix

A.8.1. Repository Overview

The `ansible-collections/community.zabbix`[8] repository is an Ansible collection for managing Zabbix monitoring infrastructure. It provides six roles (`zabbix_agent`, `zabbix_server`, `zabbix_proxy`, `zabbix_web`, `zabbix_javagateway`, `zabbix_repo`) for deploying Zabbix components across multiple operating systems, and 47 modules for managing Zabbix configuration resources including hosts, host groups, templates, actions, triggers, maintenance windows, and authentication settings.

The repository was selected because it represents a different Ansible collection use case than `amazon.aws`: while `amazon.aws` manages cloud infrastructure through AWS API calls, `community.zabbix` manages monitoring infrastructure deployment and configuration. This distinction extends the analysis to configuration management and monitoring-as-code practices, where roles handle system-level deployment and modules handle day-two configuration of a running Zabbix instance.

A.8.2. Testing Architecture

Molecule orchestrates role deployment tests using Docker containers (geerlingguy/docker-*-ansible images) with testinfra verification, while `ansible-test integration` executes module CRUD tests against a docker-compose Zabbix stack (PostgreSQL, Zabbix server, and web UI). GitHub Actions workflows use path-based triggering so that changes to a role only trigger that role's workflow.

[8] `https://github.com/ansible-collections/community.zabbix`

Multi-dimensional matrix testing covers OS images (8: Rocky Linux 10/9/8, Ubuntu 24.04/22.04, Debian 12/11, openSUSE Leap 15), Zabbix versions (4: v7.4, v7.2, v7.0, v6.0), database backends (2: MySQL and PostgreSQL), Ansible versions (5: stable-2.16 through stable-2.19 and `devel`), and Python versions (4: 3.10, 3.11, 3.12, 3.13). Explicit exclusions in the matrix reduce unsupported combinations (for example, Rocky Linux 10 excludes Zabbix versions 6.0–7.2, and Debian 11 excludes Zabbix 7.2 and above).

Static analysis is provided through tox linters (`flake8`, `antsibull-changelog`) and `ansible-test sanity` across Ansible versions 2.16–2.19 and `devel`, with explicit skip rules in `.ansible-lint` for pragmatic exceptions (`command-instead-of-shell`, `experimental`, `meta-no-info`, `no-handler`, `unnamed-task`). YAML formatting is enforced via `yamllint`.

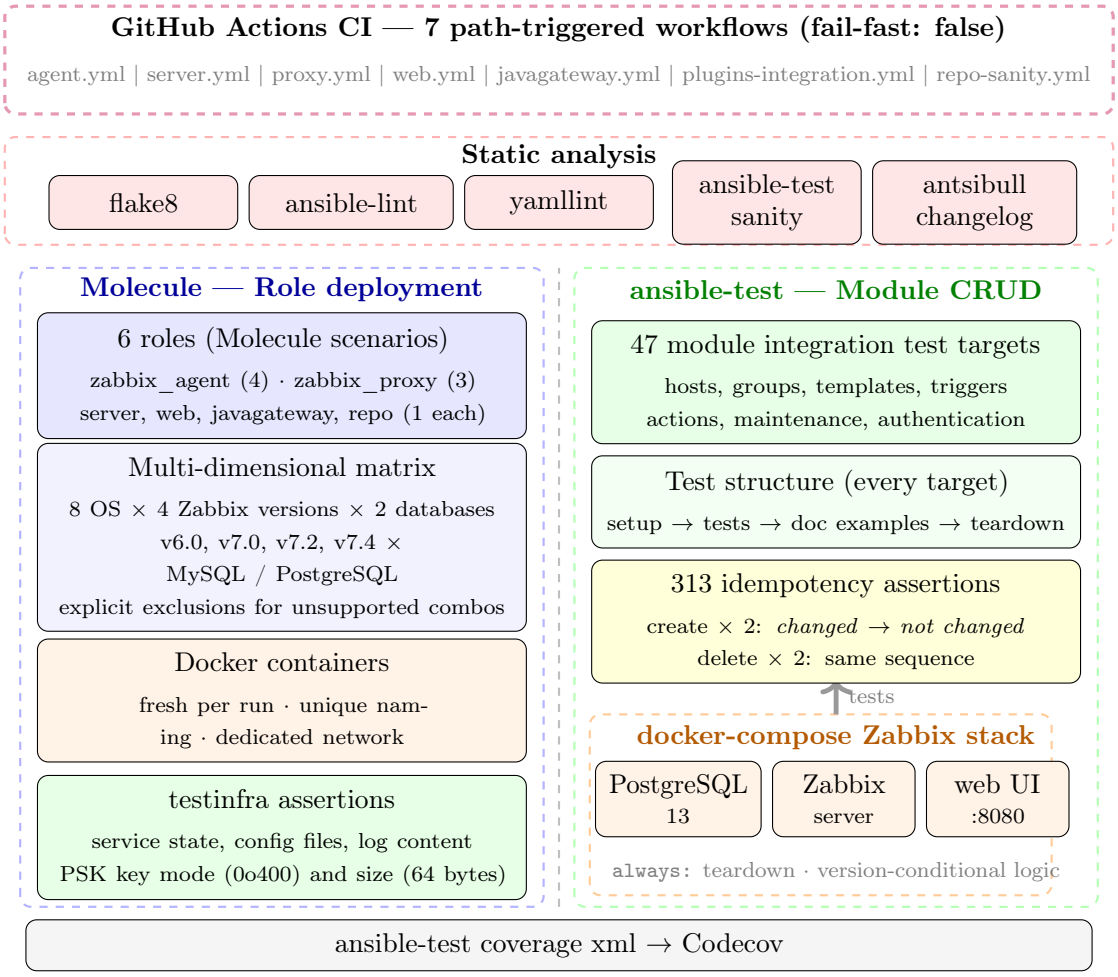

Figure A.8.: Dual testing architecture of ansible-collections/community.zabbix: Molecule for role deployment validation across a multi-dimensional OS/version/database matrix (left), and ansible-test integration for module CRUD and idempotency testing against a live docker-compose Zabbix stack (right).

A.8.3. Test Suite Coverage

Table A.14 summarises the test suite coverage.

Table A.14.: Test suite coverage for ansible-collections/community.zabbix.

Layer	Scope	Approach
Static analysis	Python code, YAML syntax, Ansible best practices, changelog	`flake8`, `ansible-lint` (5 skip rules), `yamllint`, `ansible-test sanity`, `antsibull-changelog`
Molecule role testing	6 roles; 1–4 scenarios per role (`zabbix_agent`: 4; `zabbix_proxy`: 3; remaining four roles: 1 each)	Docker containers (8 OS × 4 Zabbix versions × 2 database backends); testinfra verification (service state, config files, log content)
ansible-test integration	47 module test targets covering all collection modules	CRUD operations with explicit idempotency validation against docker-compose Zabbix stack (PostgreSQL, Zabbix server, web UI)
Coverage reporting	Code coverage for integration tests	`ansible-test coverage xml` → Codecov

A.8.4. Evaluation Against IaC Testing Dimensions

Static analysis and syntax checks. Static analysis is layered across three tools. `flake8` (configured via `tox.ini`) enforces Python style on plugin code with a 160-character line length limit and explicit per-file ignore rules. `ansible-lint` targets YAML task quality but carries five skip rules acknowledging pragmatic exceptions inherent in Ansible collection development (`command-instead-of-shell` for commands requiring shell expansion, `unnamed-task` for reusable task fragments). `yamllint` enforces YAML formatting (brace spacing, indentation, Unix line endings). The `repo-sanity.yml` workflow additionally runs `ansible-test sanity` in a Docker container across five Ansible versions, checking documentation fragments, module argument specs, return documentation, and import structure. There is no security scanning (no Trivy, Bandit, or equivalent), and no type checking of Python module code.

Logic-level unit tests. No unit tests exist. All tests provision real resources: Molecule tests deploy against Docker containers; module integration tests communicate with a live Zabbix server instance via the HTTP API. The `module_utils/` code (`api_request.py`, `base.py`, `helpers.py`), which implements all authentication and API communication logic, is exercised only indirectly through the integration tests. There is no API mocking

comparable to `amazon.aws`'s Placebo approach. This means that testing the collection without a running Zabbix instance is not possible.

Integration testing. Integration testing operates at two levels. Molecule role tests deploy each role into freshly provisioned Docker containers and use testinfra to verify the resulting state: service running and enabled, package version matching the matrix variable, configuration file ownership and permissions, correct configuration file contents, and log file assertions confirming successful startup (for example, asserting that the server log contains "`server #0 started [main process]`" and does not contain "`database is down: reconnecting`"). The agent role additionally tests PSK auto-generation, verifying that the generated key file has mode 0o400 and is exactly 64 bytes.

Module integration tests use 47 test targets, each exercising a single module through a structured task file (`setup` → `tests` → `doc examples` → `teardown`), with an `always:` block ensuring cleanup even on failure. Tests run against a docker-compose stack (PostgreSQL 13, Zabbix server, Zabbix web UI) provisioned by the CI runner. Version-conditional logic using `when: zabbix_version is version('7.0', '>=')` allows the same test targets to cover Zabbix 6.0 through 7.4 without separate test files per version.

End-to-end and outcome-based testing. The docker-compose stack provisions a full Zabbix installation including the web UI on port 8080. The `test_zabbix_web` Molecule test calls the Zabbix JSON-RPC API directly via `curl` and asserts a valid authentication token in the response, confirming that the web UI is functional. However, no tests validate end-to-end monitoring workflows (data collection, alerting, graphing, or notification delivery); tests validate API CRUD operations only. This is an appropriate scope for a configuration management collection.

State evolution and migration. Idempotency testing is explicit and systematic: every integration test applies a resource twice (asserting `is changed` on the first call and `is not changed` on the second) and deletes it twice (asserting the same sequence). The corpus contains 313 such idempotency assertions across the 47 test targets. Version-conditional logic handles feature differences between Zabbix versions. However, there are no upgrade or migration tests between Zabbix versions (for example, no test validates that a host configuration created under Zabbix 6.0 continues to function correctly under Zabbix 7.4), and no database backend migration tests.

Test fixtures and isolation. Molecule tests use fresh Docker containers per run, named using the template `zabbix-server-${VERSION}-${DB}-${OS}` to prevent naming collisions across matrix dimensions. A dedicated Docker network (`zabbix`) isolates the container networking. Module integration tests share a single docker-compose Zabbix instance within each CI run, relying on unique resource naming and `always:` cleanup blocks to prevent state pollution between individual test targets. This isolation model is weaker than fresh-per-test provisioning but is practical for a shared API server.

Negative and robustness testing. Negative testing is limited. Testinfra log assertions verify the absence of error signatures ("`Access denied for user`", "`database is down: reconnecting`", "`Both are missing in the system.`"), providing a lightweight form of negative validation after role deployment. Some integration tests use `ignore_errors: true` to confirm that operations on resources with dependencies (such as deleting a host group with assigned hosts) fail gracefully. There is no systematic API failure injection, no boundary value testing, and no testing of module behaviour under network errors or Zabbix API timeouts.

CI/CD integration. Seven GitHub Actions workflows are triggered by path-based filters, so that changes to `roles/zabbix_server/` trigger only `server.yml` and changes to `plugins/` trigger only `plugins-integration.yml`. The role workflows use fail-fast: false so that all matrix combinations are reported rather than stopping at the first failure. The `plugins-integration.yml` workflow runs `ansible-test integration` with `-continue-on-error` and `-coverage`, then generates `coverage.xml` via `ansible-test coverage xml` and uploads it to Codecov. Dependency versions are pinned at build time via `sed` substitutions in `galaxy.yml` to ensure reproducible CI runs. There is no security scanning workflow.

A.8.5. Overall Assessment

The dual testing architecture combining Molecule role tests with `ansible-test` module integration is well suited to a collection that provides both deployment roles and configuration modules. The Molecule tier validates system-level behaviour (services, packages, configuration files, logs) across a broad compatibility matrix; the `ansible-test` tier validates module logic (CRUD correctness, idempotency, version-conditional behaviour) against a live Zabbix API. Explicit idempotency assertions in every integration test target and path-based CI triggers are consistent engineering practices. The 313 idempotency assertions across 47 test targets represent a higher density of state-correctness checks than most repositories in the corpus.

The primary gaps are the absence of unit tests for `module_utils/` code (which handles all API communication logic), no API mocking, limited negative testing, and no security scanning in CI. The lack of unit tests means that bugs in the authentication or request-handling layer can only be caught by running a full Zabbix stack, increasing the feedback cycle for module developers.

Table A.15.: Evaluation of ansible-collections/community.zabbix against IaC testing dimensions.

Dimension	Evidence	Assessment
Static analysis	`flake8` (Python, 160-char limit), `ansible-lint` (5 skip rules: `command-instead-of-shell`, `experimental`, `meta-no-info`, `no-handler`, `unnamed-task`), `yamllint`, `ansible-test sanity` (Ansible 2.16–2.19 + `devel`), `antsibull-changelog`; no security scanning or type checking.	Moderate.
Logic-level unit tests	No unit tests; all tests provision real resources (Docker containers via Molecule, Zabbix server via docker-compose); `module_utils/` code validated indirectly through integration tests; no API mocking.	Low.
Integration testing	Dual architecture: (1) Molecule role tests with testinfra verification across 8 OS × 4 Zabbix versions × 2 database backends × 1–4 scenarios; (2) ansible-test integration for 47 module test targets with CRUD and explicit idempotency validation (313 idempotency assertions) against docker-compose Zabbix stack (PostgreSQL, server, web UI).	High.
End-to-end / outcome-based	docker-compose provisions Zabbix stack with web UI on port 8080; JSON-RPC API login validated via `curl` in testinfra; no end-to-end monitoring workflow validation (data collection, alerting, graphing); appropriate scope for a configuration management collection.	Low.

Table A.15.: Evaluation of ansible-collections/community.zabbix against IaC testing dimensions. (continued)

Dimension	Evidence	Assessment
State evolution	Explicit idempotency testing in every integration test (313 assertions; create twice, delete twice); version-conditional logic for feature testing (`when: zabbix_version is version('7.0', '>=')`); no upgrade or migration tests between Zabbix versions or database backends.	Moderate–High.
Test fixtures and isolation	Docker containers for role tests (fresh per run; unique naming: `zabbix-server-${VERSION}-${DB}-${OS}`; dedicated network); docker-compose for module integration (PostgreSQL + Zabbix per workflow); `always:` cleanup blocks; tests share Zabbix instance within a run.	Moderate–High.
Negative / robustness	Testinfra log assertions verify absence of error signatures (`Access denied`, `database is down`); some `ignore_errors: true` tests for dependency-constrained deletion; no systematic API failure injection, boundary value testing, or timeout handling.	Low–moderate.
CI/CD integration	7 path-triggered workflows (agent, server, proxy, web, javagateway, plugins, sanity); fail-fast: false; multi-dimensional matrices: 8 OS × 4 Zabbix × 2 databases for roles; 4 Zabbix × 5 Ansible × 4 Python for modules; pinned dependencies; coverage via Codecov; no security scanning.	High.

A.9. Evaluation of IaC Testing Practices in Pulumi

A.9.1. Repository Overview

Pulumi[9] is a multi-language infrastructure as code platform enabling users to define cloud infrastructure using general-purpose programming languages (TypeScript, JavaScript, Python, Go, .NET, Java, and YAML). It consists of a core engine written in Go, language-specific SDKs, and a provider ecosystem managing cloud resources across AWS, Azure, GCP, Kubernetes, and 150+ services.

The repository was selected because it represents a modern programmatic IaC approach with unique testing challenges around multi-language interoperability, resource lifecycle management, and state consistency. As a platform generating and executing infrastructure code across multiple language runtimes, Pulumi provides insights into testing practices for code generation systems, multi-language tooling, and stateful orchestration platforms.

A.9.2. Testing Architecture

The custom integration testing framework in `pkg/testing/integration` provides a declarative API for complex lifecycle testing across 118 integration scenarios. A dual-directory isolation approach (`HomePath` for plugins, `RootPath` for workspace) prevents conflicts when tests execute in parallel with different provider versions. Router-based test providers in `tests/testprovider` serve multiple resource types with different behaviours (including `fails_on_create`, `fails_on_delete`, and `flaky_create`) from a single binary. Property-based testing using Rapid validates state consistency with a configurable number of checks (Makefile default: 10,000 via `-rapid.checks`).

Static analysis uses golangci-lint with 22 explicitly enabled linters plus `requiredfield` as a standalone tool, mypy for Python type checking, actionlint for workflow validation, and JSON schema validation for `pulumi.json`. GitHub Actions workflows implement platform-specific optimisations, multi-cloud credential injection, and detailed test result analysis across 29 workflow files with more than 40,000 total workflow runs.

A.9.3. Test Suite Coverage

Table A.16 summarises the test suite coverage.

Table A.16.: Test suite coverage for Pulumi.

Layer	Scope	Approach
Static analysis	Go code, Python types, JSON schemas, workflows, copyright headers	golangci-lint (22 linters), requiredfield, mypy, actionlint, JSON schema validation, custom lint rules

[9] `https://github.com/pulumi/pulumi`

Table A.16.: Test suite coverage for Pulumi. (continued)

Layer	Scope	Approach
Unit testing	Backend operations, provider protocols, configuration, workspace management	Function-based mocks; 308 `_test.go` files in `pkg/`; no external dependencies
Integration testing	118 scenarios covering resource lifecycles, multi-language programs, component resources	Custom ProgramTest API; language-specific suites (approx. 1,796 lines Go, 3,546 lines Node.js); EditDirs for multi-step testing
Property-based testing	State consistency, snapshot integrity	Rapid library; configurable checks (Makefile default: 10,000 via `-rapid.checks`); state file-based fuzzing
Cross-language testing	Component provider interoperability	Go, Python, Node.js implementations validated across languages

A.9.4. Evaluation Against IaC Testing Dimensions

Table A.17 lists the ratings across all eight dimensions.

Table A.17.: Evaluation of Pulumi against IaC testing dimensions.

Dimension	Evidence	Assessment
Static analysis	golangci-lint with 22 enabled linters (gosec, exhaustive, forbidigo, importas, paralleltest, gocritic, goheader, etc.); custom rules enforcing `require.NoError` over `assert.NoError` and standardised import aliases; requiredfield as a standalone tool; mypy for Python SDK; actionlint for workflow files; JSON schema validation of `pulumi.json` via `jv` and Biome; gofumpt formatting.	High.

Table A.17.: Evaluation of Pulumi against IaC testing dimensions. (continued)

Dimension	Evidence	Assessment
Logic-level unit tests	Function-based mocking (`MockBackend`, `MockStack`, `MockProvider`); dedicated mock files covering backend operations, provider protocol validation, configuration handling, workspace management, and secrets; 308 `_test.go` files in `pkg/`; no external dependencies; isolated fast execution.	High.
Integration testing	Custom framework with 118 scenarios in `tests/integration`; ProgramTest API with `EditDirs` for multi-step lifecycle testing; language-specific suites (approx. 1,796 lines Go, 3,546 lines Node.js); cross-language component validation (Node.js, Python, Go); event stream validation; router-based test providers with per-operation handler registration.	High.
End-to-end / outcome-based	Not applicable; platform validates infrastructure provisioning correctness, not application workflows; appropriate scope for an IaC orchestration platform.	N/A.
State evolution	Property-based testing with Rapid (Makefile default: 10,000 checks) generating random sequences of create, update, and delete operations against the lifecycle engine; state file fuzzing (`TestFuzzFromStateFile`) for production issue reproduction; operational tests covering protect/unprotect, taint/untaint, refresh, and rename; no schema upgrade or version migration tests.	Moderate–High.
Test fixtures and isolation	Dual-directory approach (`HomePath` for plugins, `RootPath` for workspace); environment variable injection; local file-based backends; asymmetric cleanup (preserve on failure); global YarnInstallMutex for concurrency control; NoParallel constraints for tests requiring exclusive resources; no containerisation (Docker, LocalStack).	Moderate–High.

Table A.17.: Evaluation of Pulumi against IaC testing dimensions. (continued)

Dimension	Evidence	Assessment
Negative / robustness	Specialised test providers (`fails_on_create`, `fails_on_delete`, `flaky_create`); ExpectFailure approach with event stream validation; panic handling; error message validation for user-facing guidance; not systematic across all operations; no chaos engineering or fault injection.	Moderate.
CI/CD integration	29 workflows; 40,000+ runs; platform-specific optimisations (Windows parallelism tuning); coverage instrumentation with `PULUMI_GOCOVERDIR` workaround; multi-cloud credentials (AWS, Azure, GCP); goteststats analysis; matrix testing across OS, runtime, and versions; Codecov with 80% patch coverage target; 200+ Makefile targets.	High.

A.9.5. Overall Assessment

As the largest multi-language IaC platform in this study, Pulumi's testing infrastructure reflects its complexity. The router-based test provider architecture allows each test to register custom handler functions for specific operations, enabling fine-grained simulation of cloud behaviour without a full mocking framework. Property-based testing generates random sequences of create, update, and delete operations to verify internal state consistency. Dual-directory isolation separates plugin installations from workspace files, preventing version conflicts during parallel testing. Static analysis uses 22 explicitly enabled golangci-lint linters with custom rules, and over 40,000 CI workflow runs reflect sustained testing investment. The 118 integration scenarios prioritise system-level validation over unit testing, which is appropriate for an orchestration platform. Gaps include the absence of schema migration testing (a long-term state compatibility risk), no containerisation for cloud service simulation, and unsystematic negative testing.

A.10. Evaluation of IaC Testing Practices in pulumi-cloudflare

A.10.1. Repository Overview

The `pulumi-cloudflare`[10] repository is a Pulumi provider for managing Cloudflare infrastructure resources. It bridges the Terraform Cloudflare provider to Pulumi's multi-language platform, enabling users to define Cloudflare resources (zones, DNS records, workers, rulesets, access policies) using TypeScript, Python, Go, .NET, and Java.

The repository was selected because it demonstrates multi-language provider testing with distinctive challenges: SDK generation and validation across five languages, provider upgrade compatibility testing across versions, and integration testing against live cloud APIs. As a provider bridging two ecosystems (Terraform and Pulumi), it requires validation at multiple abstraction layers.

A.10.2. Testing Architecture

Static analysis uses golangci-lint (7 linters including security scanning), unit tests for provider logic (schema version handling, ID delegation), integration tests using Pulumi's ProgramTest framework across five language SDKs, and provider upgrade tests validating compatibility from baseline version 5.49.0. A distinctive feature is the state compatibility testing where recorded state files from provider versions v4, v5, and v6 are imported and validated to ensure preview operations succeed without errors. The CI configuration runs tests in a matrix across five languages with separate jobs for provider and example tests, requiring Cloudflare API credentials injected via Pulumi ESC (Environment, Secrets, Configuration) system.

A.10.3. Test Suite Coverage

Table A.18 summarises the test suite coverage.

Table A.18.: Test suite coverage for pulumi-cloudflare.

Layer	**Scope**	**Approach**
Static analysis	Go code quality, security, formatting	golangci-lint with 7 linters including gosec; go:embed workaround in lint Makefile target
Unit testing	Schema version tracking, ID delegation, migration reminders	Pure logic tests; no external dependencies

[10] `https://github.com/pulumi/pulumi-cloudflare`

Table A.18.: Test suite coverage for pulumi-cloudflare. (continued)

Layer	Scope	Approach
Integration testing	Resource creation across Node.js, Python, Go, .NET, Java	ProgramTest framework; live Cloudflare API; matrix testing across 5 languages
Provider upgrade	Compatibility from v5.49.0 to v6 for zones, records, rulesets, policies	PreviewProviderUpgrade; no-replacement/no-deletion assertions; version-specific test programs
State compatibility	Import v4/v5/v6 state files and validate preview	Recorded state import (`testdata/ruleset_state_*.json`); regression test for issue #1213

A.10.4. Evaluation Against IaC Testing Dimensions

Table A.19 lists the ratings across all eight dimensions.

Table A.19.: Evaluation of pulumi-cloudflare against IaC testing dimensions.

Dimension	Evidence	Assessment
Static analysis	golangci-lint with 7 explicitly enabled linters (goconst, gosec, lll, misspell, nakedret, revive, unconvert) plus gci and gofmt formatters; `.golangci.yml` excludes vendored code and generated files (`schema.go`, `pulumiManifest.go`); go:embed workaround in the lint Makefile target (temporarily renames `go:embed` directives before running golangci-lint, then restores them).	High.
Logic-level unit tests	Schema version validation (`TestSchemaVersionsForResetResources`); ID delegation for string and numeric fields (`Test_delegateID`); migration tracking reminder (`TestRuleSetVersionReminder`); no external API calls; fast, deterministic execution.	Moderate.

Table A.19.: Evaluation of pulumi-cloudflare against IaC testing dimensions. (continued)

Dimension	Evidence	Assessment
Integration testing	ProgramTest framework across 5 languages (Node.js, Python, Go, .NET, Java); language-specific test files (`examples_nodejs_test.go`, `examples_py_test.go`, etc.); live Cloudflare API with credentials from environment variables; regression tests (TestRegress444, TestRegress554); test.yml matrix with `-parallel 4` and 2-hour timeout.	High.
End-to-end / outcome-based	Control-plane validation only; `TestAccRecordGo` includes refresh with `optrefresh.ExpectNoChanges()` and `optrefresh.Diff()`; no application-level functionality (DNS resolution, worker execution, access policy enforcement); appropriate scope for a provider.	Low–moderate.
State evolution	PreviewProviderUpgrade from baseline v5.49.0 (`TestZoneUpgrade`, `TestRecordUpgrade`, `TestPageRuleUpgrade`); `assertpreview.HasNoReplacements` and `assertpreview.HasNoDeletes`; recorded state imports for v4/v5/v6 (`TestRuleSetNonEmptyRulesUpgrade`); version-specific test programs (`test-programs/zone/zonev5`, `test-programs/record/recordv5`).	High.
Test fixtures and isolation	pulumitest framework (`testProgram()` helper, `pulumitest.NewPulumiTest`); `opttest.AttachProvider` injects local implementation; configuration injection via `pt.SetConfig()`; `testing.Short()` skipping; `optnewstack.DisableAutoDestroy()` for debugging; requires live Cloudflare credentials; no containerisation or API mocking.	Moderate.

Table A.19.: Evaluation of pulumi-cloudflare against IaC testing dimensions. (continued)

Dimension	Evidence	Assessment
Negative / robustness	Skipped tests for known issues (`TestWorkersRoute`: destroy does not work, issue #1130); schema version mismatch validation; no systematic API failure testing, permission errors, rate limiting, or invalid configurations; error scenarios discovered through regression rather than proactive validation.	Low.
CI/CD integration	Matrix across 5 languages (Node.js, Python, .NET, Go, Java); run-acceptance-tests.yml with path filtering; manual approval for community PRs (`/run-acceptance-tests` command via repository_dispatch); upgrade-provider.yml scheduled daily at 3 AM UTC; Pulumi ESC with OIDC for secrets; Sentinel job (final job consolidating required checks); `-parallel 4` with 2-hour timeout.	High.

A.10.5. Overall Assessment

Provider upgrade testing is the standout feature: recorded state files from three provider versions (v4, v5, v6) are used to verify that upgrades produce no unexpected resource replacements or deletions, a critical safety property for users migrating between provider generations. Matrix integration testing validates five SDK implementations simultaneously, catching language-specific bugs in code-generated SDKs. Static analysis includes security scanning (gosec), and CI automation monitors upstream Terraform provider changes daily at 3 AM UTC with automated PR creation. The strategy appropriately focuses on control-plane validation rather than data-plane behaviour (DNS resolution, worker execution). Limitations include reliance on live Cloudflare credentials without containerised isolation, minimal negative testing for API failure scenarios, and several skipped tests indicating unresolved resource lifecycle issues.

A.11. Evaluation of IaC Testing Practices in pulumi-gcp

A.11.1. Repository Overview

The `pulumi-gcp`[11] repository is the Google Cloud Platform (GCP) resource provider for Pulumi, enabling infrastructure management across six programming languages:

[11] `https://github.com/pulumi/pulumi-gcp`

TypeScript, Python, Go, .NET, Java, and YAML. As a provider bridging the Terraform GCP provider to Pulumi's multi-language platform, it translates resource definitions into GCP API calls and manages state synchronisation between Pulumi and the cloud platform.

The repository was selected because it represents a production-grade multi-language provider with testing practices spanning all eight evaluation dimensions, requiring validation at multiple abstraction layers including SDK generation, provider upgrade compatibility, integration testing against live GCP APIs, and performance benchmarking.

A.11.2. Testing Architecture

Static analysis uses golangci-lint with 7 linters including gosec; provider unit tests (label handling, configuration validation with mocked HTTP servers), inline gRPC replay tests (provider RPC sequences recorded as JSON and replayed in-process without live credentials), provider integration tests (50 test programs using pulumitest framework), example integration tests (full-stack validation across six languages with real GCP provisioning), and provider upgrade tests (systematic validation against a default baseline of 6.67.0 with individual test overrides to 7.0.0, 7.2.1, and 7.17.0). Performance benchmark infrastructure exists but is unconditionally skipped in the current codebase. GitHub Actions workflows orchestrate matrix testing across five languages with parallel execution (`-parallel 4`) and secure GCP authentication via Workload Identity Federation.

A.11.3. Test Suite Coverage

Table A.20 summarises the test suite coverage.

Table A.20.: Test suite coverage for pulumi-gcp.

Layer	**Scope**	**Approach**
Static analysis	Go code quality, security, formatting	golangci-lint with 7 linters including gosec; go:embed workaround; 20m timeout
Provider unit tests	Label handling, provider config, region validation	Mocked HTTP servers (`httptest.NewServer`); `testutil.InitLogging` for log assertions
Inline replay tests	Provider RPC correctness without live credentials	`replay.Replay` / `replay.ReplaySequence` with JSON-embedded gRPC request/response sequences

Table A.20.: Test suite coverage for pulumi-gcp. (continued)

Layer	Scope	Approach
Provider integration tests	Regression scenarios, secrets, Cloud Run/Functions, drift detection	pulumitest framework; 50 test programs with `LocalProviderPath` and `YarnLink`
Example integration tests	Full workflows across 6 languages	`integration.ProgramTest` with real GCP provisioning; matrix CI across 5 languages
Provider upgrade tests	Schema evolution, backward compatibility	PreviewProviderUpgrade with default baseline 6.67.0 (individual tests override to 7.0.0, 7.2.1, 7.17.0); HasNoReplacements/HasNoDeletes assertions; upgrade coverage reporting
Performance benchmarks	Provider initialisation time	Benchmark suite across 21 YAML programs; unconditionally skipped in current codebase

A.11.4. Evaluation Against IaC Testing Dimensions

Table A.21 lists the ratings across all eight dimensions.

Table A.21.: Evaluation of pulumi-gcp against IaC testing dimensions.

Dimension	Evidence	Assessment
Static analysis	golangci-lint with 7 explicitly enabled linters (goconst, gosec, lll, misspell, nakedret, revive, unconvert) plus gci and gofmt formatters; go:embed workaround technique; gci import organisation; excludes generated files; lint_provider.fix target; 20m timeout.	High.

Table A.21.: Evaluation of pulumi-gcp against IaC testing dimensions. (continued)

Dimension	Evidence	Assessment
Logic-level unit tests	Label expansion with glob patterns; provider configuration with mocked HTTP servers (`httptest.NewServer`); region validation; custom `expectLogs` type for diagnostic message validation; schema transformation tests; no actual resource provisioning. Inline replay tests (`replay.Replay`, `replay.ReplaySequence`) execute full gRPC sequences (Configure, Diff, Create, Check) against an in-process provider server using JSON-embedded request/response fixtures, requiring no live GCP credentials.	High.
Integration testing	Dual architecture: (1) provider integration via pulumitest with 50 test programs covering secrets, Cloud Run panic regressions (TestCloudrunServicePanicRegress2155/2622), Cloud Functions with E2E HTTP validation, drift detection via `gcloud` modifications; (2) example integration via `integration.ProgramTest` across 6 languages with real GCP APIs; RetryFailedSteps for eventual consistency; matrix testing across 5 languages in CI.	High.
End-to-end / outcome-based	`validateAPITest` helper with exponential backoff (5 s–160 s) for HTTP GET requests; `isRetryable` function recognises transient 5xx/404 states; validates Cloud Function response content and deployment artefacts; tests runtime Node.js version detection and automatic runtime selection.	High.

Table A.21.: Evaluation of pulumi-gcp against IaC testing dimensions. (continued)

Dimension	Evidence	Assessment
State evolution	Systematic provider upgrade testing: `testUpgrade` creates infrastructure with default baseline 6.67.0 and previews upgrade to current version; individual tests override to 7.0.0 (IAM resources), 7.2.1 (GKE cluster), and 7.17.0; `assertpreview.HasNoReplacements` / `HasNoDeletes` ensure schema stability; `TestUpgradeCoverage` generates coverage reports; multi-step EditDirs for incremental changes; individual upgrade tests for DNS, PubSub, storage, and connection profile.	High.
Test fixtures and isolation	pulumitest framework with `t.TempDir()` and `t.Cleanup()` hooks; `t.Setenv()` for test-scoped environment variables; DestroyOnCleanup: true (default); Pulumi auto-naming for unique resource names; `testing.Short()` for conditional skipping; mocked HTTP servers for unit tests; inline replay tests require no credentials; no shared state between tests.	High.
Negative / robustness	Error message validation (`TestCloudfunctionWrongType`); configuration edge case tests (mismatched regions, API failures via `RegionListErrorHandler`); `expectLogs` type for diagnostic quality; `TestCheckConfigNoCredentials` validates the no-credentials error path; several upgrade tests permanently skipped due to flakiness (TestCloudFunctionUpgrade, TestNetworkUpgrade, TestClusterUpgrade, TestIamBinding, TestTopicIamBinding – all with open issues); no systematic fault injection for rate limits, network partitions, or quota exhaustion.	Moderate.

Table A.21.: Evaluation of pulumi-gcp against IaC testing dimensions. (continued)

Dimension	Evidence	Assessment
CI/CD integration	Matrix testing across 5 languages (Node.js, Python, .NET, Go, Java) with fail-fast: false and `-parallel 4`; Workload Identity Federation for GCP authentication; Pulumi ESC for secret management via OIDC; scheduled upstream monitoring (daily 3 AM UTC) with automatic PR creation; PR gating with Sentinel jobs consolidating all checks; repository dispatch for manual triggering; benchmark infrastructure exists but unconditionally skipped.	High.

A.11.5. Overall Assessment

Three practices stand out. First, drift detection testing: creating infrastructure through Pulumi, modifying it externally via `gcloud` commands, then verifying that Pulumi correctly detects and reconciles the change. Second, exponential backoff E2E validation with configurable retry schedules (5 s–160 s) that distinguish transient failures from permanent ones, preventing flaky tests for eventually consistent GCP services. Third, systematic provider upgrade testing with coverage reporting and no-replacement/no-deletion assertions across a default baseline of 6.67.0, with per-resource overrides where the baseline differs. A distinctive additional practice is the inline gRPC replay suite: RPC request/response sequences are embedded as JSON and replayed against an in-process provider server, enabling regression testing of complex Diff and Configure scenarios without live GCP credentials. Static analysis includes security scanning (gosec), isolation uses the pulumitest framework, and CI leverages Workload Identity Federation for credential-free authentication. Gaps include limited negative testing for API failure scenarios, no fault injection, several permanently skipped upgrade tests tracking known flakiness issues, and benchmark infrastructure that exists but does not execute.

A.12. Evaluation of IaC Testing Practices in terraform-aws-eks-cluster

A.12.1. Repository Overview

The `terraform-aws-eks-cluster`[12] repository is a Cloud Posse-maintained Terraform module for provisioning AWS Elastic Kubernetes Service (EKS) clusters with IAM

[12] `https://github.com/cloudposse/terraform-aws-eks-cluster`

roles, OIDC providers, EKS access entries, and addons. It integrates with companion modules (`terraform-aws-eks-node-group`, `terraform-aws-eks-fargate-profile`) to form complete Kubernetes clusters on AWS.

The repository was selected because it represents infrastructure provisioning for a complex, stateful, multi-component system (managed Kubernetes cluster), providing insight into testing strategies for orchestrating cloud-managed services with intricate access control, addon management, and operational validation beyond infrastructure existence.

A.12.2. Testing Architecture

A BATS-based test harness cloned from the external `cloudposse/test-harness` repository executes linting, validation, and documentation verification. Terratest integration tests provision complete AWS EKS clusters and validate both infrastructure outputs and Kubernetes API functionality through shared informers monitoring worker node joins. GitHub Actions workflows delegate to shared reusable templates in the `cloudposse/.github` repository: the branch workflow runs static CI checks on every push or pull request, while integration tests are exclusively triggered via ChatOps (`/terratest` comments on pull requests), running inside `cloudposse/test-harness:latest` Docker containers with a 120-minute timeout and a Terraform/OpenTofu platform matrix. Provider-pinning tests are explicitly disabled due to a known upstream bug in the `hashicorp/terraform-provider-tls` provider (issue #244).

A.12.3. Test Suite Coverage

Table A.22 summarises the test suite coverage.

Table A.22.: Test suite coverage for terraform-aws-eks-cluster.

Layer	**Scope**	**Approach**
Static analysis	Linting, syntax validation, documentation	BATS harness: lint, validate, terraform-docs, input/output descriptions, module-pinning; provider-pinning disabled (TLS provider issue #244); external `test-harness` repository
Integration testing	Full EKS cluster with VPC, subnets, node groups	Terratest: `InitAndApply` with `Upgrade: true`; fixture-based config (`fixtures.us-east-2.tfvars`); output validation (VPC/subnet CIDRs, cluster ID, node group status); us-east-2 region

Table A.22.: Test suite coverage for terraform-aws-eks-cluster. (continued)

Layer	Scope	Approach
Kubernetes API validation	Worker node join detection	k8s.io/client-go shared informers with AWS IAM authenticator tokens; atomic counters with channel-based synchronisation; 5-minute timeout for 2+ nodes
Disabled mode testing	Verify `enabled=false` creates no resources	Regex extraction of Terraform output: "Resources: 0 added, 0 changed, 0 destroyed."

A.12.4. Evaluation Against IaC Testing Dimensions

Table A.23 lists the ratings across all eight dimensions.

Table A.23.: Evaluation of terraform-aws-eks-cluster against IaC testing dimensions.

Dimension	Evidence	Assessment
Static analysis	BATS harness (lint, validate, terraform-docs, input/output descriptions, module-pinning) from external `cloudposse/test-harness` repository; provider-pinning explicitly disabled due to `hashicorp/terraform-provider-tls` issue #244; test definitions external, reducing transparency; delegated to shared workflows in `cloudposse/.github`.	Moderate.
Logic-level unit tests	Not applicable; declarative module nature limits applicability; all logic consists of resource declarations and data source lookups validated through integration tests against real AWS APIs; no mocking infrastructure.	N/A.

Table A.23.: Evaluation of terraform-aws-eks-cluster against IaC testing dimensions. (continued)

Dimension	Evidence	Assessment
Integration testing	Terratest provisions complete EKS clusters (VPC, subnets, control plane, node groups) in us-east-2; validates infrastructure outputs (VPC CIDR 172.16.0.0/16, subnet CIDRs, cluster ID, node group status ACTIVE); k8s.io/client-go shared informers monitor node joins with 5-minute timeout via atomic counters; `Upgrade: true` for provider upgrade testing; `fixtures.us-east-2.tfvars` specifies Kubernetes 1.33, addons (kube-proxy, coredns), upgrade policy, zonal shift, remote network config.	High.
End-to-end / outcome-based	Full cluster provisioning with functional validation (node joins via k8s.io/client-go shared informers and IAM authenticator); no application workload deployment, pod networking validation, addon functionality testing (beyond installation), or EKS feature exercising (OIDC, access entries).	Moderate–High.
State evolution	Three migration guides (`docs/migration-0.45.x+.md`, `docs/migration-v1-v2.md`, `docs/migration-v3-v4.md`) document breaking changes; `Upgrade: true` ensures provider upgrades are tested; no automated state migration tests, resource renaming validation, or cross-version compatibility testing.	Low–Moderate.

Table A.23.: Evaluation of terraform-aws-eks-cluster against IaC testing dimensions. (continued)

Dimension	**Evidence**	**Assessment**
Test fixtures and isolation	Random resource naming via `random.UniqueId()` prevents conflicts; temporary directories (`testStructure.CopyTerraformFolderToTemp`) for `TestExamplesCompleteDisabled`; deferred cleanup with `terraform.Destroy()` and `runtime.HandleCrash()` for crash recovery; dedicated AWS sandbox account via IAM role assumption (2-hour token TTL); version-controlled fixtures; no shared state between test runs.	High.
Negative / robustness	Single disabled mode test (`TestExamplesCompleteDisabled` with `enabled=false`); no systematic error condition testing (invalid credentials, insufficient IAM permissions, quota exhaustion, network failures, K8s API unavailability, addon installation failures); no fault injection.	Low.

Table A.23.: Evaluation of terraform-aws-eks-cluster against IaC testing dimensions. (continued)

Dimension	Evidence	Assessment
CI/CD integration	Branch workflow runs static CI checks on every push/PR; integration tests run exclusively via ChatOps (`/terratest` comments with `run_terratest` permission check) inside `cloudposse/test-harness:latest` Docker containers; dual-platform matrix (Terraform and OpenTofu) via `update-alternatives`; Terraform version auto-detection via `terraform-config-inspect` and `vert`; 120-minute timeout via `robherley/go-test-action`; AWS credentials via IAM role assumption (`arn:aws:iam::799847381734:role/cptest-test-ue2-sandbox-gha-iam-terratest`, 2-hour token); scheduled workflow runs README generation only, not integration tests; `fail-fast: false` with max-parallel 10 (serialised to 1 when `test-skip-concurrency` property is set); all implementation via shared reusable templates in `cloudposse/.github`.	High.

A.12.5. Overall Assessment

Operational readiness is validated beyond Terraform state: tests authenticate to the provisioned EKS cluster and monitor worker node joins using `k8s.io/client-go` shared informers with thread-safe atomic counters and a 5-minute timeout. This catches IAM permission problems, VPC networking misconfigurations, and addon failures invisible to Terraform. ChatOps triggering (`/terratest` comments) manages the cost of provisioning full EKS clusters, and the `runtime.HandleCrash` hook ensures cleanup even on unexpected process termination. The shared CI infrastructure supports name-based secret injection for multiple cloud providers (AWS, GitHub, Opsgenie, Datadog, Spotinst, TFE, Cloudflare), activated by repository name; for this module only AWS credentials are injected. Gaps include no automated state migration testing despite three documented migration guides, a disabled provider-pinning check due to an upstream provider bug, single-region testing (us-east-2 only), no unit tests for module logic, and minimal negative testing.

A.13. Evaluation of IaC Testing Practices in terraform-aws-rds

A.13.1. Repository Overview

The `terraform-aws-rds`[13] repository is a Terraform module developed by Cloud Posse for provisioning AWS RDS database instances. It supports multiple database engines (MySQL, PostgreSQL, SQL Server), configurable storage options, multi-AZ deployments, and integration with VPC networking, security groups, and Route53 DNS management.

The repository was selected because it represents a common IaC practice: a reusable Terraform module for a specific cloud service. Unlike providers or orchestrators, this module abstracts RDS provisioning complexity while requiring careful testing of resource dependencies, networking integration, and database-specific configurations. This provides insight into how Cloud Posse's ecosystem approaches module testing with their standardised test-harness framework.

A.13.2. Testing Architecture

Cloud Posse's test-harness framework executes BATS tests for Terraform linting, validation, documentation checks (`terraform-docs`), module pinning verification, provider pinning, plugin installation, and input/output description completeness. The single Terratest-based Go test function (`TestExamplesComplete`) provisions real AWS RDS infrastructure (VPC, subnets, RDS instance, security groups, parameter group, option group, subnet group) and validates Terraform output values. GitHub Actions workflows use shared Cloud Posse templates with ChatOps integration (`/terratest` comment triggers) for cost-conscious on-demand integration testing. A `docker/test` Makefile target supports local execution inside `cloudposse/test-harness:latest` using the same Docker image as CI.

A.13.3. Test Suite Coverage

Table A.24 summarises the test suite coverage.

Table A.24.: Test suite coverage for terraform-aws-rds.

Layer	Scope	Approach
Static analysis	Terraform syntax, linting, formatting, documentation	BATS via test-harness: installed, lint, get-modules, module-pinning, get-plugins, provider-pinning, validate, terraform-docs, input-descriptions, output-descriptions

[13] `https://github.com/cloudposse/terraform-aws-rds`

Table A.24.: Test suite coverage for terraform-aws-rds. (continued)

Layer	Scope	Approach
Integration testing	VPC, subnets, RDS instance, security groups, parameter/option/subnet groups	Terratest (`TestExamplesComplete`) with real AWS provisioning; `rand.Intn(100000)` for randomised IDs; `Upgrade: true` for provider version compatibility; 60-minute local timeout
Output validation	Terraform outputs (instance ID, option/parameter/subnet group IDs, VPC CIDR, subnet CIDRs)	`assert.Equal` / `assert.Contains`; no database connectivity or query testing
Cleanup	Resource teardown	`defer terraform.Destroy`; no crash recovery handler; `docker/test` Makefile target for local reproducibility

A.13.4. Evaluation Against IaC Testing Dimensions

Table A.25 lists the ratings across all eight dimensions.

Table A.25.: Evaluation of terraform-aws-rds against IaC testing dimensions.

Dimension	Evidence	Assessment
Static analysis	BATS via test-harness (installed, lint, get-modules, module-pinning, get-plugins, provider-pinning, validate, terraform-docs, input-descriptions, output-descriptions); shared CI workflow via `cloudposse/.github`; runs on every PR.	Moderate.
Logic-level unit tests	None identified; conditional logic (subnet group creation, replica detection, engine version computation) is not tested in isolation; all validation occurs via integration tests against real AWS APIs; no mocking infrastructure.	None.

Table A.25.: Evaluation of terraform-aws-rds against IaC testing dimensions. (continued)

Dimension	Evidence	Assessment
Integration testing	`TestExamplesComplete` provisions a complete RDS environment (VPC, subnets, MySQL 5.7 instance on `db.t3.micro`, security groups, parameter group with custom parameters `myisam_sort_buffer_size` and `sort_buffer_size`); randomised attributes via `rand.Intn(100000)`; `Upgrade: true` for provider compatibility; output validation only (instance ID, group IDs, networking CIDRs); no database connectivity or query execution. A second example configuration (MSSQL) exists but has no corresponding Terratest coverage.	Moderate.
End-to-end / outcome-based	No database functionality testing (connectivity, authentication, query execution); validates infrastructure provisioning outcomes only (resource IDs, networking configuration).	Low.
State evolution	No automated state migration or upgrade tests; `apply_immediately = true` bypasses the maintenance window in all fixture configurations; no drift detection, snapshot, or point-in-time restore testing.	None.
Test fixtures and isolation	Randomised IDs via `rand.Intn(100000)` prevent resource name conflicts; `fixtures.us-east-2.tfvars` pin engine version, instance class, and parameter group; `defer terraform.Destroy` for cleanup; `docker/test` target enables local reproduction using the same `cloudposse/test-harness:latest` image as CI; no shared state between test runs.	Moderate.
Negative / robustness	No error condition testing (invalid credentials, subnet misconfiguration, storage limits, provisioning failures); the MSSQL example is not covered by any automated test; assumes successful provisioning paths only.	None.

Table A.25.: Evaluation of terraform-aws-rds against IaC testing dimensions. (continued)

Dimension	Evidence	Assessment
CI/CD integration	Branch workflow runs static CI checks only on push/PR; integration tests run exclusively via ChatOps (`/terratest` comments) through the shared `cloudposse/.github` chatops workflow, which provides Terraform/OpenTofu platform matrix, `terraform-config-inspect` version detection, and 120-minute CI timeout; scheduled workflow (daily 03:00 UTC) runs README generation only, not integration tests; `fail-fast: false` with max-parallel 10 in the shared chatops runner.	Moderate.

A.13.5. Overall Assessment

The Cloud Posse standardised test harness provides consistent static checks, with integration tests provisioning real RDS infrastructure and validating Terraform output values. The BATS suite covers ten checks including provider-pinning and module-pinning, and local Docker execution is supported via the `docker/test` Makefile target. An important gap is that tests validate infrastructure outputs (resource IDs, subnet CIDRs) but never connect to the database or execute queries, leaving a distance between configuration correctness and operational readiness. Other gaps include no automated tests for the MSSQL example configuration, no state evolution or upgrade testing, no unit tests for module logic, and no negative testing. The CI rating is moderate because integration tests are only triggered manually via ChatOps and the scheduled workflow does not run them.

A.14. Evaluation of IaC Testing Practices in terraform-aws-vpc

A.14.1. Repository Overview

The `terraform-aws-vpc`[14] repository is a Terraform module developed by Cloud Posse for provisioning AWS VPCs with Internet Gateways. It provisions VPC resources including IPv4/IPv6 CIDR blocks, internet gateways, default security groups, network ACLs, and route tables. The module includes a submodule (`modules/vpc-endpoints`) for provisioning Interface and Gateway VPC Endpoints.

[14] `https://github.com/cloudposse/terraform-aws-vpc`

The repository was selected as it represents a mature, production-grade Terraform module with an established testing infrastructure characteristic of Cloud Posse's module ecosystem. The repository demonstrates integration testing practices using Terratest with real AWS infrastructure provisioning, combined with automated static checks through a test-harness framework.

A.14.2. Testing Architecture

Cloud Posse's test-harness framework executes eight BATS checks against the module root (installed, lint, module-pinning, provider-pinning, validate, terraform-docs, input-descriptions, output-descriptions) and three checks against the example configuration (installed, lint, validate). Terratest-based Go tests provision real AWS VPC infrastructure across two independent test functions covering the basic VPC and the VPC endpoints submodule. GitHub Actions workflows use shared Cloud Posse templates with ChatOps integration (`/terratest` comment triggers) for on-demand integration testing.

A.14.3. Test Suite Coverage

Table A.26 summarises the test suite coverage.

Table A.26.: Test suite coverage for terraform-aws-vpc.

Layer	**Scope**	**Approach**
Static analysis	Terraform syntax, linting, formatting, documentation	BATS via test-harness: 8 checks on module root (installed, lint, module-pinning, provider-pinning, validate, terraform-docs, input-descriptions, output-descriptions); 3 checks on examples/complete
Integration testing	VPC, IGW, subnets (public + private), Gateway endpoints (S3, DynamoDB), Interface endpoints (EC2, Kinesis Streams)	Terratest with real AWS provisioning; `random.UniqueId()` for randomised IDs; `Upgrade: true` for version compatibility; `CopyTerraformFolderToTemp` for isolation
Output validation	VPC CIDR, subnet CIDRs, endpoint configurations, DNS entries	`assert.Equal`; `terraform.OutputStruct` with custom `VpcEndpoint` struct for endpoint attribute validation; no application connectivity testing

Table A.26.: Test suite coverage for terraform-aws-vpc. (continued)

Layer	Scope	Approach
Cleanup	Resource teardown	Shared `cleanup()` helper: `terraform.Destroy` + `os.RemoveAll` on temp directory

A.14.4. Evaluation Against IaC Testing Dimensions

Table A.27 lists the ratings across all eight dimensions.

Table A.27.: Evaluation of terraform-aws-vpc against IaC testing dimensions.

Dimension	Evidence	Assessment
Static analysis	BATS via test-harness (installed, lint, module-pinning, provider-pinning, validate, terraform-docs, input-descriptions, output-descriptions); provider-pinning enabled (not disabled); shared CI workflow via `cloudposse/.github`; runs on every PR.	Moderate.
Logic-level unit tests	None identified; `TestCompleteDisabled` validates zero-resource state only; conditional logic untested in isolation.	Low.
Integration testing	`TestExamplesComplete` provisions a VPC with CIDR 172.16.0.0/16, public and private subnets, and security hardening (default security group deny-all, default route table no-routes, default network ACL deny-all); `TestExamplesVPCEndpoints` provisions a separate VPC (172.17.0.0/16, private subnets only) with Gateway endpoints (S3, DynamoDB) and Interface endpoints (EC2, Kinesis Streams); validates endpoint types, VPC associations, and `private_dns_enabled` flags; `Upgrade: true`; no application workload deployment.	High.

Table A.27.: Evaluation of terraform-aws-vpc against IaC testing dimensions. (continued)

Dimension	Evidence	Assessment
End-to-end / outcome-based	Validates infrastructure attributes via Terraform outputs (CIDR blocks, endpoint service names, endpoint type, VPC ID associations, DNS entry fields); `VpcEndpoint` struct deserialises the `dns_entry` output to check whether expected hostnames appear in the endpoint's DNS entry list — this checks Terraform state data, not live DNS resolution; Kinesis Streams endpoint asserts the hostname is *absent* (because `private_dns_enabled = false`); no application connectivity or workload deployment testing.	Moderate.
State evolution	`Upgrade: true` tests provider version compatibility; one migration guide (`docs/migration-v1-v2.md`) documents breaking changes; no drift detection, idempotency verification, or automated state migration testing.	Low–Moderate.
Test fixtures and isolation	`CopyTerraformFolderToTemp` creates isolated temporary directories per test; shared `cleanup()` helper calls `terraform.Destroy` and removes the temp directory; randomised IDs via `random.UniqueId()`; `t.Parallel()` with temp directory isolation prevents state conflicts; fixture files parameterise inputs including security hardening flags; AWS role assumption with 2-hour sessions via shared chatops workflow.	High.
Negative / robustness	`TestExamplesCompleteDisabled` passes `enabled="false"` (string) and asserts `assert.Contains(results, "Resources: 0 added, 0 changed, 0 destroyed.")`; the Kinesis Streams DNS entry check is a structural negative assertion on endpoint behaviour; no fault injection, invalid configuration testing, or error scenario validation.	Low.

Table A.27.: Evaluation of terraform-aws-vpc against IaC testing dimensions. (continued)

Dimension	Evidence	Assessment
CI/CD integration	Branch workflow runs static CI checks only on push/PR; integration tests run exclusively via ChatOps (`/terratest` comments) through the shared `cloudposse/.github` chatops workflow, which provides Terraform/OpenTofu platform matrix, 120-minute CI timeout, and AWS credential management; scheduled workflow (daily 03:00 UTC) runs README generation only; `fail-fast: false` with max-parallel 10.	Moderate.

A.14.5. Overall Assessment

VPC endpoint testing is the most detailed aspect of this repository's test suite: tests cover both Gateway and Interface endpoint types using a custom `VpcEndpoint` Go struct to deserialise complex Terraform outputs, validating service names, endpoint types, VPC associations, and DNS entry fields. The Kinesis Streams test asserts that the private DNS hostname is absent from the endpoint's DNS entry list when `private_dns_enabled = false`, which is a structural negative assertion on endpoint behaviour. Test isolation uses temporary directories with deferred cleanup and randomised resource naming. The complete fixture enforces security hardening defaults (deny-all security group, no-routes route table, deny-all ACL), testing the module's security baseline configuration. Gaps include no unit tests for module logic, no idempotency or drift detection testing, no negative testing for invalid inputs or error conditions, and no application connectivity validation beyond Terraform state attributes. The CI rating is moderate because integration tests are only triggered via ChatOps and the scheduled workflow does not run them.

A.15. Evaluation of IaC Testing Practices in terraform-azurerm-caf-enterprise-scale

A.15.1. Repository Overview

The `terraform-azurerm-caf-enterprise-scale`[15] module is Microsoft's official Terraform implementation of Azure landing zones, providing enterprise-scale infrastructure deployment practices for Azure environments. The module manages complex hierarchies of management groups, policy definitions, policy assignments, role assignments, and core platform resources across multiple subscriptions.

[15] `https://github.com/Azure/terraform-azurerm-caf-enterprise-scale`

The repository was selected because it represents a mature, production-grade Terraform module maintained by Microsoft Azure's Cloud Adoption Framework team. As a module designed for enterprise deployments spanning multiple Azure resource types, it provides insight into testing practices for large-scale, policy-driven infrastructure modules where configuration correctness is critical.

A.15.2. Testing Architecture

GitHub Actions runs multi-language linting via `github/super-linter` on every PR. Azure Pipelines Unit Tests (plan-based validation with OPA/Conftest policy testing against stored baseline configurations), Azure Pipelines E2E Tests (sequential `terraform apply` operations across three test modules simulating incremental deployment scenarios), and Azure Pipelines Baseline Updates (dedicated workflow for regenerating OPA regression baselines). The separation between GitHub Actions and Azure Pipelines serves a security purpose: GitHub Actions run automatically without Azure credentials, while Azure Pipelines require manual comment triggers (`/azp run unit`, `/azp run e2e`) before executing tests with access to Azure infrastructure. Test execution uses a matrix strategy with version combinations pinned in a PowerShell script, with dedicated Azure subscriptions allocated per test job via the Subscription Aliases API to ensure isolation.

A.15.3. Test Suite Coverage

Table A.28 summarises the test suite coverage.

Table A.28.: Test suite coverage for terraform-azurerm-caf-enterprise-scale.

Layer	**Scope**	**Approach**
Static analysis	Terraform, Bash, PowerShell, JSON, YAML, Markdown, GitHub Actions	`github/super-linter@v6` with `VALIDATE_TERRAFORM_TFLINT: true`; standalone TFLint step commented out in current codebase
Plan-based validation	3 test modules simulating incremental deployments	`terraform init` + `plan` via Azure Pipelines; JSON plan output for OPA policy validation
Policy validation	Management groups, policies, roles (6 resource types)	OPA/Conftest with 6 Rego policies; jq-based dynamic substitution of `root_id`, `root_name`, and locations in `baseline_values.json`

Table A.28.: Test suite coverage for terraform-azurerm-caf-enterprise-scale. (continued)

Layer	Scope	Approach
Go plan-based tests	Policy assignment enforcement mode overrides	`terratest-terraform-fluent` framework with `InitPlanShowWithPrepFunc`; plan-only, no apply; `check.InPlan` assertions on resource attributes
E2E lifecycle tests	Sequential deployment across 3 modules	`terraform apply` with shared state; retry logic (up to 5 attempts); cleanup job with conditional execution
Matrix testing	Terraform and provider version combinations	PowerShell script (`azp-strategy.ps1`) generates matrix; currently pinned to Terraform 1.11.0 and AzureRM 3.117.0

A.15.4. Evaluation Against IaC Testing Dimensions

Table A.29 lists the ratings across all eight dimensions.

Table A.29.: Evaluation of terraform-azurerm-caf-enterprise-scale against IaC testing dimensions.

Dimension	Evidence	Assessment
Static analysis	`github/super-linter@v6` with `VALIDATE_TERRAFORM_TFLINT: true` plus `VALIDATE_BASH`, `VALIDATE_POWERSHELL`, `VALIDATE_JSON`, `VALIDATE_YAML`, `VALIDATE_MARKDOWN` on every PR; a standalone TFLint step with the azurerm plugin is present in the workflow but commented out in the current codebase; `terraform fmt` check in unit pipeline.	Moderate.

Table A.29.: Evaluation of terraform-azurerm-caf-enterprise-scale against IaC testing dimensions. (continued)

Dimension	Evidence	Assessment
Logic-level unit tests	One Go test file (`archetypeconfig_test.go`) using Azure's `terratest-terraform-fluent` framework; `InitPlanShowWithPrepFunc` runs `terraform plan` without apply; a prep function injects provider configuration files into a temporary directory; `check.InPlan` asserts specific resource attribute values (e.g. `enforce=false` on policy assignments); plan-based only, no mock providers or `.tftest.hcl` files.	Low–Moderate.
Integration testing	3 test modules (test_001_baseline, test_002_add_custom_core, test_003_add_mgmt_conn); OPA/Conftest with 6 Rego policy files (management_groups, policy_definitions, policy_set_definitions, policy_assignments, role_definitions, role_assignments); `baseline_values.json` regression testing with jq-based dynamic substitution (5 replacements: root_id, root_name, primary and secondary locations, uppercase root_id); matrix testing against pinned Terraform 1.11.0 and AzureRM 3.117.0.	High.
End-to-end / outcome-based	Sequential apply across 3 modules with Azure Pipelines job dependencies (run_e2e_tests_001 → 002 → 003 → cleanup); shared state file (`terraform-$TF_VERSION-$TF_AZ_VERSION.tfstate`); retry logic with up to 5 attempts for transient failures; cleanup job runs on success unconditionally and on failure/cancellation only when `ALWAYS_DESTROY != 'false'`; no application-level validation.	Moderate–High.
State evolution	Sequential E2E workflow simulates incremental updates against shared state across three module configurations; no explicit drift detection, state migration, or provider upgrade testing.	Moderate.

Table A.29.: Evaluation of terraform-azurerm-caf-enterprise-scale against IaC testing dimensions. (continued)

Dimension	Evidence	Assessment
Test fixtures and isolation	Dedicated Azure subscriptions per matrix job via Subscription Aliases API (version `2020-09-01`); dedicated Service Principals with credential injection; isolated state files in Azure Backend Storage with job-specific paths; dynamic 8-character `TF_ROOT_ID` generation per job via `Get-RandomId`.	High.
Negative / robustness	Retry logic in `tf-apply.sh` for transient failures (up to 5 attempts); Go plan tests for enforcement mode config overrides; no systematic fault injection for API failures, throttling, quota limits, or permission errors.	Low–Moderate.
CI/CD integration	GitHub Actions (`super-linter`, automated, no Azure credentials); Azure Pipelines (manual triggers `/azp run unit|e2e|update` for security); matrix testing with PowerShell-generated version strategy; branch protection enforces code review and unit tests on PRs; dedicated baseline update pipeline (`tests-update.yml`) regenerates OPA baselines via PR automation.	High.

A.15.5. Overall Assessment

Two practices stand out. First, plan-based regression testing with OPA/Conftest: `baseline_values.json` files store expected plan outputs, and six Rego policy files are designed to identify deviations in resource counts, naming conventions, or configuration values across management groups, policies, and role assignments. jq substitutions parameterise the baseline with the current `root_id` and deployment locations, enabling the same baseline structure to validate different test runs. Second, test-level Azure subscription isolation: each CI job receives dedicated subscriptions for management and connectivity workloads via the Subscription Aliases API, reducing the risk of cross-job interference. The Go plan-based tests (`terratest-terraform-fluent`) validate specific attribute values on planned resources without requiring live Azure credentials. Gaps

include a commented-out standalone TFLint step (static analysis relies on super-linter), no native Terraform test files (`.tftest.hcl`), limited negative testing, and no drift detection or state migration testing.

A.16. Evaluation of IaC Testing Practices in terraform-google-address

A.16.1. Repository Overview

The `terraform-google-address`[16] module is a Terraform module for reserving Google Cloud Platform IP addresses (both regional and global) and managing associated DNS records, supporting forward DNS (A records) and reverse DNS (PTR records) registration in Cloud DNS.

The repository was selected as a representative example of a Terraform module within the Google Cloud ecosystem, demonstrating testing practices for infrastructure that combines GCP compute resources with DNS management. This provides insight into how Terraform modules approach testing for both resource provisioning and cross-service integration.

A.16.2. Testing Architecture

GitHub Actions runs automated linting via a Docker container executing `test_lint.sh` and `commitlint` on every PR; Cloud Build (integration testing orchestrating five test scenarios through explicit four-stage lifecycles: init, apply, verify, teardown). Tests use the Cloud Foundation Toolkit Blueprint Test framework with a dual verification approach: `DefaultVerify()` validates Terraform state consistency, followed by `gcloud` CLI commands querying actual GCP resource state via compute and DNS APIs. The Cloud Build pipeline enforces sequential execution with `waitFor` dependencies, provisions real GCP resources (compute addresses, DNS zones, DNS records), and uses a dedicated test project with isolated network infrastructure per fixture. A `sleep 240` pause follows environment preparation to allow IAM permissions to propagate before tests begin.

A.16.3. Test Suite Coverage

Table A.30 summarises the test suite coverage.

[16] `https://github.com/terraform-google-modules/terraform-google-address`

Table A.30.: Test suite coverage for terraform-google-address.

Layer	Scope	Approach
Static analysis	Terraform, Shell, Python, Go	`test_lint.sh` via `gcr.io/cloud-foundation-cicd/cft/developer-tools:1.26` Docker image; `commitlint` for conventional commit validation
Integration tests	5 test scenarios covering IP and DNS configurations	CFT Blueprint Test framework; real GCP resource provisioning; 22 Cloud Build steps; 3.5-hour timeout (12600s)
Verification approach	Compute addresses, DNS zones, A and PTR records	Dual verification: `DefaultVerify()` on Terraform state, followed by `gcloud compute addresses list` and `gcloud dns record-sets list` API queries
Test scenarios	TestIpAddressOnly, TestIpAddressWithSpecificIp, TestDnsForwardExample, TestDnsForwardExampleMultiNames, TestDnsForwardAndReverse	Sequential Cloud Build execution with `waitFor` dependencies; 4-stage lifecycle per scenario (init, apply, verify, teardown)
Isolation	Dedicated test project, isolated networks	`terraform-google-modules/project-factory` for project creation with random suffix; Docker containers; fixture-specific VPC and subnets via symlinked shared `network.tf`; `force_destroy = true` on DNS zones for clean teardown

A.16.4. Evaluation Against IaC Testing Dimensions

Table A.31 lists the ratings across all eight dimensions.

Table A.31.: Evaluation of terraform-google-address against IaC testing dimensions.

Dimension	Evidence	Assessment
Static analysis	`test_lint.sh` via `gcr.io/cloud-foundation-cicd/cft/developer-tools:1.26` Docker image (version pinned in Makefile, read dynamically by `lint.yaml`); `test_verify_boilerplate.py` for copyright header validation; `commitlint` for commit message format; GitHub Actions lint workflow on every PR.	Moderate–High.
Logic-level unit tests	None; only `test/test_verify_boilerplate.py` for copyright header validation; no mocked tests for IP allocation logic, DNS record generation, or conditional resource creation.	None.
Integration testing	5 test scenarios with real GCP resources; CFT Blueprint Test framework with `DefaultVerify()` for Terraform state validation; `gcloud` CLI verification (`compute addresses list`, `dns record-sets list`); covers compute-to-DNS integration, forward and reverse DNS mapping, and multi-name-to-IP mappings; Cloud Build orchestration with 22 steps (swap-module-refs, prepare, and 4 stages per test scenario).	High.
End-to-end / outcome-based	Infrastructure provisioning verified through `gcloud` queries; tests check that DNS records exist with the expected `rrdatas` values and that compute addresses are registered under the expected names; no DNS resolution testing from external clients; no application deployment or connectivity validation; each scenario deploys and immediately destroys.	Moderate.
State evolution	No state migration, provider upgrade, or in-place update testing; Cloud Build tests cover only fresh deployments with immediate teardown; no infrastructure update scenarios after initial deployment.	None.

Table A.31.: Evaluation of terraform-google-address against IaC testing dimensions. (continued)

Dimension	Evidence	Assessment
Test fixtures and isolation	Dedicated test project via `terraform-google-modules/project-factory` with random suffix; isolated VPC and subnet per fixture; shared fixture files (`network.tf`, `variables.tf`, `shared_outputs.tf`) distributed via symlinks across fixture directories; Docker containers for consistent execution; `force_destroy = true` on DNS managed zones for clean teardown; Cloud Build N1_HIGHCPU_8 machines.	High.
Negative / robustness	No tests for invalid inputs (malformed IPs, invalid DNS names), resource quota exhaustion, API rate limiting, service outages, or resource conflicts; `test/make.sh` includes defensive bash shebang checks but no systematic error scenarios.	Low.
CI/CD integration	GitHub Actions for automated linting on PRs; Cloud Build for integration tests with 3.5-hour timeout (12600s); sequential execution with `waitFor` dependencies preventing resource conflicts; 4-stage explicit lifecycle (init, apply, verify, teardown); Makefile targets for local testing (`docker_test_lint`, `docker_test_integration`); `commitlint` for commit message validation.	High.

A.16.5. Overall Assessment

The dual verification approach is the defining structural feature: after Terraform state validation via `DefaultVerify()`, `gcloud` API queries independently check that DNS records and compute addresses exist in actual GCP state with the expected values. Five deployment scenarios run through the CFT Blueprint Test framework with four-stage lifecycle management (init, apply, verify, teardown). Test isolation uses a dedicated GCP project and Docker containers, with shared fixture files distributed via symlinks and `force_destroy = true` on DNS zones for reliable teardown. The prepare step includes a 240-second pause to allow IAM permission propagation, indicating sensitivity

to eventual consistency in the test environment. Gaps include no logic-level unit tests, no state evolution or migration testing, and no negative testing for invalid inputs or error conditions.

A.17. Evaluation of IaC Testing Practices in terraform-google-bootstrap

A.17.1. Repository Overview

The `terraform-google-bootstrap`[17] module is a Terraform module for bootstrapping GCP organisations, provisioning seed projects, state storage buckets, service accounts, and Cloud Build CI/CD infrastructure for Terraform workflows. The module operates at the organisation bootstrap level, establishing the foundational infrastructure for subsequent IaC deployments.

The repository was selected because it demonstrates testing practices for organisation-level IaC that includes CI/CD pipeline provisioning, providing insight into how modules that create Cloud Build pipelines can be tested, including validation of the provisioned pipelines' operational behaviour.

A.17.2. Testing Architecture

Kitchen-Terraform uses InSpec controls to validate GCP resources (projects, storage buckets, service accounts, IAM bindings, API enablement), while Go tests handle scenarios requiring programmatic control over git repositories, Cloud Build execution, and Cloud Workflows orchestration. Go tests clone GitHub and GitLab repositories, push commits, create and merge pull requests programmatically using official SDKs (`go-github/v72`, `go-gitlab`), trigger Cloud Build pipelines, poll builds and workflows for completion, and verify Docker image artefacts in Artifact Registry. Cloud Build orchestrates the full test suite with a 5400-second timeout across three Kitchen-Terraform suites (simple, cloud-build_enabled, simple-folder) executed sequentially due to permission re-establishment requirements between suites, followed by eleven Go test scenarios with `waitFor` dependency management for parallel application and sequential teardown.

A.17.3. Test Suite Coverage

Table A.32 summarises the test suite coverage.

[17] `https://github.com/terraform-google-modules/terraform-google-bootstrap`

Table A.32.: Test suite coverage for terraform-google-bootstrap.

Layer	Scope	Approach
Static analysis	Terraform formatting, multi-language linting	Pre-commit `terraform_fmt`; Docker-based linting (`test_lint.sh`) via `gcr.io/cloud-foundation-cicd/cft/developer-tools:1.25`; GitHub Actions `commitlint`
Kitchen-Terraform	3 suites: simple, cloudbuild_enabled, simple-folder	InSpec controls validating GCP resources (projects, buckets, service accounts, APIs, IAM); negative assertion on SA key types
Go integration tests	11 scenarios: source repos, builder variants (CSR/GitHub/GitLab), workspace variants (CSR/GitHub/GitLab), IM workspace variants (GitHub/GitLab), repo connection variants (GitHub/-GitLab)	CFT Blueprint Test framework; `gcloud` CLI validation; real GCP resource provisioning
End-to-end tests	Git workflows, Cloud Build pipelines, Cloud Workflows, Infrastructure Manager deployments	Clone repos with PAT auth; commit/PR/merge via `go-github/v72` and `go-gitlab` SDKs; trigger Cloud Scheduler; poll workflows and builds for completion; verify Docker artefacts in Artifact Registry
Isolation	Docker containers, dedicated projects, test fixtures	Separate fixtures per scenario; Secret Manager for PATs (`IM_GITHUB_PAT`, `IM_GITLAB_PAT`); `t.TempDir()` for git operations; `t.Cleanup()` teardown guarantee

A.17.4. Evaluation Against IaC Testing Dimensions

Table A.33 lists the ratings across all eight dimensions.

Table A.33.: Evaluation of terraform-google-bootstrap against IaC testing dimensions.

Dimension	Evidence	Assessment
Static analysis	Pre-commit hooks with `terraform_fmt`; Docker-based linting via `test_lint.sh` in `gcr.io/cloud-foundation-cicd/cft/developer-tools:1.25` image; GitHub Actions lint workflow; `commitlint` for conventional commits; Makefile integration for version pinning.	Moderate.
Logic-level unit tests	None. All Go tests use `bpt.NewTFBlueprintTest(t)` to provision real GCP resources; no mocking or isolated logic testing; `go.mod` includes dependencies on blueprint-test, terratest, testify, but all are used for integration testing.	None.
Integration testing	Kitchen-Terraform: 3 InSpec suites validating 11 default APIs, bucket IAM, SA key management (with negative assertion: `should_not include 'USER_MANAGED'`), folder-level IAM; Go tests: 11 scenarios validating source repos, artifact registries, Docker tagging, trigger configuration, workspace deployments; `gcloud` CLI with JSON parsing; Cloud Build orchestration with 48 steps and 5400-second timeout; two `prepare-rerun` steps between Kitchen suites to reinstate CI integration account permissions.	High.
End-to-end / outcome-based	Git operations via blueprint-test `git` package (clone, commit, push); PR workflows using `go-github/v72` SDK (create, merge, close); Cloud Scheduler triggers Cloud Workflows via HTTP; poll workflow executions (up to 100 iterations, 20 s interval) with FAILED retry for IAM consistency; poll Cloud Build by commit SHA (up to 100 iterations, 10 s interval); verify Docker images in Artifact Registry with version tags; Infrastructure Manager deployment validation.	High.

Table A.33.: Evaluation of terraform-google-bootstrap against IaC testing dimensions. (continued)

Dimension	Evidence	Assessment
State evolution	No state migration, provider upgrade, or in-place update testing; tests provision from scratch via fixtures then destroy via `bpt.DefaultTeardown()`; Kitchen suites use converge–verify–destroy cycle but with no resource modification; no drift detection or idempotency testing.	None.
Test fixtures and isolation	Separate fixtures in `test/fixtures/` per scenario (9 fixture directories); isolated test projects via `test/setup/` infrastructure; Docker containers for all Cloud Build steps; dedicated GitHub and GitLab test repositories; `t.TempDir()` for git operations; Secret Manager injection for PATs via `availableSecrets` in Cloud Build; `t.Cleanup()` guarantee; `waitFor` dependencies separating parallel apply from sequential teardown.	High.
Negative / robustness	Retry logic in workflow polling for eventually consistent IAM (re-triggers scheduler on FAILED status); `t.Cleanup()` teardown guarantee; InSpec negative assertion on SA key types; no systematic fault injection for API failures, throttling, quota limits, or permission errors.	Low–Moderate.
CI/CD integration	GitHub Actions (automated linting, no GCP credentials); Cloud Build (48 steps, 5400-second timeout) with PATs injected from Secret Manager; three Kitchen-Terraform suites run sequentially with permission re-establishment steps between each; eleven Go scenarios run in parallel groups where possible with `waitFor` dependencies; Makefile targets for local testing (`docker_test_lint`, `docker_test_integration`).	High.

A.17.5. Overall Assessment

The end-to-end git workflow testing is the most complex testing practice observed in the corpus: Go tests clone repositories, push commits on specific branches, create and merge pull requests using official GitHub and GitLab SDKs, trigger Cloud Scheduler jobs, poll Cloud Workflows executions for completion with retry logic for eventually consistent IAM, poll Cloud Build runs by commit SHA, and verify that the correct Docker images appear in Artifact Registry with the expected version tags. This tests the behaviour of the provisioned CI/CD infrastructure rather than only verifying the creation of Terraform resources. The three Kitchen-Terraform suites validate foundational GCP resources using InSpec, including a negative assertion on service account key types. Test isolation uses dedicated fixtures and projects, Secret Manager for PAT injection, and `t.Cleanup()` guarantees. The two `prepare-rerun` steps between Kitchen suites reflect a real permission management constraint in the test environment. Gaps include no logic-level unit tests, no state evolution or migration testing, and limited negative testing for error conditions.

A.18. Evaluation of IaC Testing Practices in terraform-google-kubernetes-engine

A.18.1. Repository Overview

The `terraform-google-kubernetes-engine`[18] module is an official Terraform module for creating and managing Google Kubernetes Engine (GKE) clusters with configurations for node pools, networking, security features, and GKE-specific capabilities. The module includes multiple submodules for private clusters, beta features, autopilot clusters, and specialised configurations such as safer clusters.

The repository was selected because it represents an enterprise-grade infrastructure module for complex, stateful Kubernetes infrastructure involving multiple interconnected resources (clusters, node pools, networking, IAM), providing insight into testing approaches for multi-resource IaC modules maintained by a major cloud provider.

A.18.2. Testing Architecture

Tests provision four dedicated GCP projects via `test/setup` infrastructure (gke-project-1, gke-project-2, gke-project-asm, gke-project-fleet) with 21 APIs enabled per project and IAM role assignments. Thirty-six Go-based integration test files each provision real GKE clusters using fixtures in `test/fixtures/`, validate configurations via golden file comparison (JSON snapshots of `gcloud container clusters describe` output with data sanitisation using `SERVICE_ACCOUNT`, `PROJECT_ID`, and `CLUSTER_NAME` tokens), and execute through four stages (init, apply, verify, teardown) orchestrated by Cloud Build with `waitFor` dependencies. The prepare step includes a `sleep 120` pause for permission propagation. `DefaultVerify` is commented out in most tests due to a known

[18] `https://github.com/terraform-google-modules/terraform-google-kubernetes-engine`

client certificate token change issue; the primary verification methods are `TGKEVerify`, `TGKEAssertGolden`, and explicit `gcloud` assertions. The `deploy_service` test provides application-level validation by deploying Nginx to a cluster and polling the LoadBalancer endpoint.

A.18.3. Test Suite Coverage

Table A.34 summarises the test suite coverage.

Table A.34.: Test suite coverage for terraform-google-kubernetes-engine.

Layer	**Scope**	**Approach**
Static analysis	Terraform formatting, linting	Docker-based linting via CFT `developer-tools:1.26`; `terraform-docs` for documentation; `commitlint`; no explicit local linting config files
Integration tests	36 scenarios: regional/zonal, private/public, autopilot, node pools, networking, Windows, security	Blueprint Test framework (Go); real GKE provisioning; transient error retries (3 attempts, 2-minute intervals); 105 Cloud Build steps; 3.5-hour timeout
Golden file validation	Cluster configuration snapshots	`JSONPath` assertions on `gcloud` output via `TGKEAssertGolden`; data sanitisation; 28 `testdata/*.json` fixtures
Application E2E	Nginx deployment, LoadBalancer validation, HTTP checks	`kubectl` service query; IP polling (20 attempts, 10s intervals); response content validation
Plan verification	Post-apply idempotency checks	`TGKEVerify` / `TGKEVerifyExemptResources` assert no-op plans; `DefaultVerify` commented out in most tests due to known client cert token issue
Isolation	4 dedicated GCP projects, Docker containers, randomised fixtures	`test/setup` provisions 4 projects with 21 APIs; VPC isolation; Cloud Build `waitFor` orchestration; E2_HIGHCPU_8 machines

A.18.4. Evaluation Against IaC Testing Dimensions

Table A.35 lists the ratings across all eight dimensions.

Table A.35.: Evaluation of terraform-google-kubernetes-engine against IaC testing dimensions.

Dimension	Evidence	Assessment
Static analysis	Docker-based linting via `make docker_test_lint` in CFT `developer-tools:1.26`; `terraform-docs` for documentation generation; `commitlint`; no explicit `.tflint.hcl` or local linting configs; relies on CFT defaults.	Low–Moderate.
Logic-level unit tests	None. All 36 tests use `tft.NewTFBlueprintTest` to provision real GKE clusters; `go.mod` includes Terratest, Blueprint Test, gcloud, but all are used for integration testing; no mocking libraries.	None.
Integration testing	36 Go tests covering diverse GKE scenarios (regional, zonal, autopilot, private, shared VPC, Windows, security features); Blueprint Test framework with four-phase execution (init, apply, verify, teardown); transient error handling via `RetryableTransientErrors` map (Error 409 operation queuing, Error 409 concurrent policy, `rateLimitExceeded`, internal Error 13, Error 400 incompatible operation); `gcloud` CLI validation; Cloud Build orchestration with 105 steps and `waitFor` dependencies.	High.

Table A.35.: Evaluation of terraform-google-kubernetes-engine against IaC testing dimensions. (continued)

Dimension	Evidence	Assessment
End-to-end / outcome-based	`deploy_service_test.go` deploys an Nginx service, uses `kubectl` to retrieve the LoadBalancer IP, polls the HTTP endpoint (20 attempts, 10s intervals), and checks that the response body contains the expected nginx welcome string; golden file validation compares `gcloud` output against `testdata/*.json` with sanitisation; `TGKEVerify` asserts no-op plans post-apply; `DefaultVerify` is commented out in most tests, citing a known client certificate token change issue.	High.
State evolution	No automated upgrade, migration, or drift testing; the repository includes upgrade documentation (`docs/upgrading_to_v*.md`); no tests for Kubernetes version updates, node pool modifications, or state reconciliation.	None.
Test fixtures and isolation	`test/setup` provisions 4 GCP projects (gke-project-1, gke-project-2, gke-project-asm, gke-project-fleet) with 21 APIs and IAM role assignments; 36 fixture directories in `test/fixtures/`; dedicated VPCs and subnets per fixture; Docker container execution; `sleep 120` in prepare step for permission propagation; Cloud Build `waitFor` dependencies manage execution order.	High.
Negative / robustness	Transient error retries via `RetryableTransientErrors` (5 error signatures); `beta_cluster_test.go` uses `assert.False` to verify absent fields (no `initialNodeCount`, no `autoscaling.enabled`) for the default pool; no systematic fault injection, invalid config testing, quota exhaustion, or security boundary tests.	Low.

Table A.35.: Evaluation of terraform-google-kubernetes-engine against IaC testing dimensions. (continued)

Dimension	Evidence	Assessment
CI/CD integration	GitHub Actions lint workflow (`lint.yaml`) triggers on every PR with CFT Docker-based linting and `commitlint`; Cloud Build (`build/int.cloudbuild.yaml`) with 105 steps, 3.5-hour timeout, E2_HIGHCPU_8 machines, and `waitFor` orchestration; module swapping; `sleep 120` environment prep; Makefile targets for local Docker testing; same two-pipeline architecture (GitHub Actions for PR linting, Cloud Build for integration) as the other Google CFT repositories in the corpus.	High.

A.18.5. Overall Assessment

The `deploy_service` test is structurally distinct from other GKE tests in the corpus: it deploys an Nginx service to a provisioned cluster, retrieves the LoadBalancer IP via `kubectl`, and polls the HTTP endpoint until a valid response is received, checking response body content. This confirms that the cluster can schedule and expose workloads, not just that Terraform reports a successful apply. Golden file validation with data sanitisation checks cluster configuration against stored JSON snapshots, providing a structured approach to detecting configuration drift between test runs. The `TGKEVerify` function asserts no-op Terraform plans after apply across 19 tests. A notable observation from code inspection is that `DefaultVerify` is commented out in most tests with a note citing a known client certificate token change issue, which means the CFT framework's built-in state consistency check is not active for the majority of the test suite. Transient error retries handle five categories of GKE API errors. Gaps include no logic-level unit tests, no state evolution or migration testing, limited negative testing, and the commented-out `DefaultVerify` in most scenarios.

A.19. Evaluation of IaC Testing Practices in terraform-google-network

A.19.1. Repository Overview

The `terraform-google-network`[19] module is an official Terraform module for creating and managing Google Cloud Platform (GCP) VPC networks and related networking

[19] `https://github.com/terraform-google-modules/terraform-google-network`

resources. The module includes 16 submodules for subnets, firewall rules, routes, VPC peering, Private Service Connect endpoints, network firewall policies (global, regional, and hierarchical), and Network Connectivity Center configurations.

The repository was selected because it represents a production-grade Terraform module whose integration testing covers complex networking infrastructure using real cloud resources, providing insight into how widely-used modules approach testing for multi-submodule networking configurations.

A.19.2. Testing Architecture

Tests provision dedicated GCP resources via `test/setup` infrastructure (one project and two additional projects across three folders). Seventeen Go-based integration test files each provision real GCP networking resources using fixtures in `test/fixtures/`, validate configurations via `gcloud` commands with JSON response parsing using `gjson`, and execute through four stages (init, apply, verify, teardown) orchestrated by Cloud Build with explicit `waitFor` dependencies across 54 steps. `DefaultVerify` is skipped in two tests: `submodule_network_peering` (noting that peering state changes asynchronously) and `network_connectivity_center` (noting a bug in the provider). The Cloud Build file uses two invocation approaches: older tests use the legacy `RUN_STAGE={stage} go test` approach, while newer tests (Private Service Connect, firewall policies, Network Connectivity Center) use the modern `cft test run {TestName} -stage {stage}` command. A Python migration script (`helpers/migrate.py`) generates `terraform state mv` commands for module refactoring but has no automated tests.

A.19.3. Test Suite Coverage

Table A.36 summarises the test suite coverage.

Table A.36.: Test suite coverage for terraform-google-network.

Layer	**Scope**	**Approach**
Static analysis	TFLint, `terraform validate`, `terraform fmt`	Docker-based execution via `test_lint.sh` in CFT `developer-tools:1.25`; GitHub Actions and separate `lint.cloudbuild.yaml` pipeline
Integration tests	17 scenarios: VPC, subnets, firewall rules, peering, PSC, network policies	Blueprint Test framework (Go); real GCP provisioning; 4-stage lifecycle (init, apply, verify, teardown); sequential orchestration; 54 Cloud Build steps

Table A.36.: Test suite coverage for terraform-google-network. (continued)

Layer	Scope	Approach
Verification	Resource configuration, relationships, metadata	`gcloud` commands with `gjson` JSON parsing; detailed assertions on firewall rules, peering state, policy rules
Test orchestration	54 Cloud Build steps across 17 scenarios	Explicit staging via `RUN_STAGE` environment variable (legacy) and `cft test run` (modern); converge → verify → destroy lifecycle; `waitFor` dependencies
Isolation	Dedicated test projects, Docker containers, automatic cleanup	`test/setup` provisions 1 main project and 1 producer project across 3 org folders; sequential execution; 4-stage lifecycle with teardown
Migration tooling	Python script for state refactoring	`helpers/migrate.py` generates `terraform state mv` commands; count-to-`for_each` migrations; `-dryrun` preview mode; no automated tests

A.19.4. Evaluation Against IaC Testing Dimensions

Table A.37 lists the ratings across all eight dimensions.

Table A.37.: Evaluation of terraform-google-network against IaC testing dimensions.

Dimension	Evidence	Assessment
Static analysis	Docker-based linting via `test_lint.sh` in CFT `developer-tools:1.25`; `terraform validate` and `terraform fmt` via shared Docker image; GitHub Actions `lint.yaml` on PRs; separate `lint.cloudbuild.yaml` pipeline; `commitlint` for conventional commits; no local TFLint configuration or security scanning tools.	Moderate–High.

Table A.37.: Evaluation of terraform-google-network against IaC testing dimensions. (continued)

Dimension	Evidence	Assessment
Logic-level unit tests	None. All 17 tests use `tft.NewTFBlueprintTest` to provision real GCP resources; `go.mod` includes Terratest, Blueprint Test, gjson, but all are used for integration testing; no unit test files or mocking libraries.	None.
Integration testing	17 Go tests covering VPC networks, subnets (basic, regional, secondary ranges), firewall rules (priorities, logging, tags, service accounts), VPC peering (custom routes, ACTIVE state assertion), Private Service Connect (Google APIs, producers, endpoints), network firewall policies (global, regional, hierarchical), Network Connectivity Center; Blueprint Test framework; `gcloud` verification with `gjson` parsing; 4-stage lifecycle per scenario; dual invocation approaches (legacy `RUN_STAGE={stage} go test`; modern `cft test run`).	High.
End-to-end / outcome-based	Tests validate infrastructure configuration through `gcloud` queries; no application-level connectivity testing through firewall rules; no service validation through PSC endpoints; `gcloud` verification checks resource existence and configuration attributes rather than functional behaviour.	Low.
State evolution	`helpers/migrate.py` generates `terraform state mv` commands for module refactoring (count-to-`for_each` migrations) with a `-dryrun` preview mode; seven upgrade guides in `docs/` (`upgrading_to_v2.0.md` through `upgrading_to_v14.0.0.md`); no automated upgrade tests, drift detection, or migration validation.	Low.

Table A.37.: Evaluation of terraform-google-network against IaC testing dimensions. (continued)

Dimension	Evidence	Assessment
Test fixtures and isolation	`test/setup` provisions a main project and a PSC producer project across three Google Cloud folders; `test/fixtures/` contains fixture configurations; Docker container execution (`gcr.io/cloud-foundation-cicd/cft/developer-tools:1.25`); Cloud Build sequential orchestration with explicit `waitFor` dependencies; 4-stage lifecycle with automatic teardown.	High.
Negative / robustness	Some tests note asynchronous state behaviour (peering state changes, provider bug in NCC) and skip `DefaultVerify` accordingly rather than asserting expected error states; `submodule_firewall` checks a `disabled` firewall rule with `assert.True(fwDeny.Get("disabled").Bool()`; no systematic failure testing, invalid input combinations, API failures, or quota exhaustion scenarios.	Low.
CI/CD integration	GitHub Actions for lint on PRs (`.github/workflows/lint.yaml`); Cloud Build for integration tests (`build/int.cloudbuild.yaml`) with 7200-second timeout and 54 sequential steps; explicit staging (init, apply, verify, teardown) with `waitFor` dependencies; Makefile targets (`docker_test_integration`, `docker_test_lint`) for local development; dual invocation approaches (legacy `RUN_STAGE` and modern `cft test run`) within the same Cloud Build file.	High.

A.19.5. Overall Assessment

Seventeen integration tests cover networking scenarios from basic VPC creation through Private Service Connect and Network Connectivity Center, with no unit tests. Verification uses `gcloud` commands with `gjson` JSON parsing to check cloud resource state

independently of Terraform outputs. CI orchestration models the four-stage lifecycle as explicit Cloud Build steps with `waitFor` dependencies across 54 sequential operations. An observable practice from code inspection is the coexistence of two invocation styles in the same Cloud Build file: older scenarios use the legacy `RUN_STAGE` environment variable approach, while more recent additions use `cft test run`. Two tests skip `DefaultVerify` with inline comments noting specific known issues (asynchronous peering state and a provider bug), rather than asserting expected error states. The `helpers/migrate.py` script alongside seven upgrade guides illustrates a gap between migration tooling and migration validation: the script generates correct `terraform state mv` commands but is not itself covered by automated tests. Other gaps include no application-level connectivity validation and limited negative testing.

A.20. Evaluation of IaC Testing Practices in Terragrunt

A.20.1. Repository Overview

Terragrunt[20] is a Go-based CLI wrapper and orchestration tool for Terraform and OpenTofu, maintained by Gruntwork. It provides DRY configuration management, dependency orchestration across multi-module stacks, remote state management, and parallel execution of IaC operations.

The repository was selected because, as an orchestration layer around Terraform/OpenTofu rather than a module or provider, it demonstrates how IaC tooling approaches testing the interaction between the orchestrator, the underlying IaC engine, cloud providers, and complex multi-module dependency graphs. Its support for both Terraform and OpenTofu backends and its large-scale testing infrastructure make it structurally distinct from the other repositories in the corpus.

A.20.2. Testing Architecture

Static analysis uses golangci-lint with 45 explicitly enabled linters (including seven test-quality-specific linters), pre-commit hooks, codespell, markdownlint, licensei, SonarCloud, and CodeRabbit; (2) unit tests across internal packages using testify assertions and mock generation via `uber/mock`; (3) a large integration test suite with 66 test files and 179 fixture directories, exercising the full CLI as a subprocess against real cloud resources, partitioned into 19 build-tag-isolated CI scenarios plus a separate OIDC scenario; (4) three fuzz targets for the filter expression parser with one seed input per target; and (5) benchmark tests at both internal and integration levels. The CI/CD pipeline comprises 22 workflows coordinated by a central `ci.yml`, with parallel quality gates, cross-platform base tests (Ubuntu, macOS), multi-platform build verification with static linking checks, and JUnit reporting via `go-junit-report` across all workflows. Additional testing infrastructure includes dedicated race condition test files, a standalone flaky test detection

[20] `https://github.com/gruntwork-io/terragrunt`

tool (`test/flake/`), and an install script verification workflow with GPG and Cosign signature validation.

A.20.3. Test Suite Coverage

Table A.38 summarises the test suite coverage.

Table A.38.: Test suite coverage for Terragrunt.

Layer	Scope	Approach
Static analysis	golangci-lint (45 linters), pre-commit hooks, codespell, markdownlint, licensei, SonarCloud, CodeRabbit	Dynamic build-tag discovery via Makefile; 7 test-quality linters enforce testing best practices; dedicated CI workflow for each tool
Unit testing	Internal packages: cache, CAS, CLI, filter, git, shell, provider cache, queue, telemetry, etc.	Go testing with testify/assert and testify/require; `t.Parallel()` throughout; platform-specific test files; mock generation via `uber/mock` (mockgen v0.6.0)
Integration testing	66 test files, 179 fixture directories, 19+1 CI scenarios: AWS, GCP, cross-cloud, OIDC, SSH, SOPS, TFLint, Engine, Mocks, Windows, provider cache, Parse, CAS, etc.	Full CLI invocation as subprocess; real cloud resource provisioning; build-tag isolation per scenario; 45-minute timeout; JUnit reporting via `go-junit-report`; helper library with platform-specific utilities
Fuzz testing	Filter expression parser: FuzzParse, FuzzLexer, FuzzParser	Three fuzz targets in `internal/filter/fuzz_test.go`; one seed input per target via `f.Add`; CI runs each target for 30 seconds via `make fuzz`; corpus caching; failure archival
Benchmark testing	CAS clone/store, git operations, provider caching, HCL formatting	Internal benchmarks in `internal/cas`, `internal/git`, `internal/discovery`; integration-level benchmarks in `test/benchmarks/`

Table A.38.: Test suite coverage for Terragrunt. (continued)

Layer	Scope	Approach
Race condition testing	CAS getter concurrency, SOPS concurrency	Dedicated `race_test.go` files; CI "Race" scenario with `-race` flag matching `.*WithRacing` expression
Test reliability	Flaky test detection and analysis	Standalone Go module in `test/flake/` with analyzer, GitHub integration, parser components
Build verification	7 platform/arch combinations; static linking; no-proxy builds	`build.yml` with static binary verification; `build-no-proxy.yml` with `GOPROXY=direct`; release signing (macOS, Windows)

A.20.4. Evaluation Against IaC Testing Dimensions

Table A.39 lists the ratings across all eight dimensions.

Table A.39.: Evaluation of Terragrunt against IaC testing dimensions.

Dimension	Evidence	Assessment
Static analysis	golangci-lint with 45 explicitly enabled linters across categories including test quality, performance, correctness, and code quality; 7 test-quality linters (paralleltest, tparallel, testifylint, testpackage, usetesting, testableexamples, thelper) enforce testing conventions; pre-commit hooks (`tofu-fmt`, `goimports`); codespell; markdownlint-cli2; licensei for dependency licence compliance; SonarCloud; CodeRabbit for AI-assisted review; dynamic build-tag discovery via Makefile `LINT_TAGS` ensures build-tag-gated code is linted.	High.

Table A.39.: Evaluation of Terragrunt against IaC testing dimensions. (continued)

Dimension	Evidence	Assessment
Logic-level unit tests	Unit tests across internal packages (awshelper, cache, cas, cli, clihelper, codegen, discovery, filter, git, queue, shell, telemetry, tf, util, vfs, worker, worktrees, etc.); testify/assert and testify/require assertions; `t.Parallel()` throughout; platform-specific test files (`run_cmd_unix_test.go`, `run_cmd_windows_test.go`); mock generation via `uber/mock` (mockgen v0.6.0).	High.
Integration testing	66 test files in `test/` with 179 fixture directories; 19 build-tag-isolated CI scenarios in `integration-test.yml` (including AWS, GCP, AWSGCP, SSH, SOPS, TFLint, Engine, Windows, Provider Cache, Deprecated, Mocks, Race, Parse, CAS, Experiment, OpenTofu variants) plus 1 OIDC scenario; full CLI subprocess invocation; real cloud resource provisioning (AWS, GCP); internal integration tests (`cas/`, `discovery/`); helper library (21 non-test Go files).	High.
End-to-end / outcome-based	Integration tests cover full `terragrunt run-all apply` workflows across multi-module stacks with dependency resolution, provider caching, plan file management, and output verification; tests validate forwarding to Terraform/OpenTofu and resulting infrastructure state; dual IaC engine testing (Terraform + OpenTofu via build tags).	Moderate–High.
State evolution	No dedicated state migration tests; no drift detection testing; version constraint validation tests check compatibility but not upgrade path testing.	Low.

Table A.39.: Evaluation of Terragrunt against IaC testing dimensions. (continued)

Dimension	Evidence	Assessment
Test fixtures and isolation	179 fixture directories with self-contained Terragrunt/Terraform configurations; `CopyEnvironment` helper creates isolated copies; `CleanupTerraformFolder` for cleanup; build-tag isolation separates CI scenarios; provider cache sharing across tests reduces redundant downloads.	High.
Negative / robustness	Race condition tests in dedicated files (`internal/cas/race_test.go`, `pkg/config/sops_race_test.go`) with CI `-race` flag scenario; fuzz tests exercise the filter expression parser (FuzzParse, FuzzLexer, FuzzParser) with 30s per target; dependency cycle detection tests; credential error explanation tests (`ExplainError`); invalid source handling; version constraint failure tests; flaky test detection tooling (`test/flake/`).	Moderate.
CI/CD integration	22-workflow pipeline via `ci.yml`; base tests on Ubuntu and macOS; 19+1 build-tag-isolated integration scenarios with per-scenario cloud credentials; separate OIDC workflow with `id-token:write` permissions; JUnit reporting via `go-junit-report` across all workflows; `mise`-based tool pinning (`mise.toml` for dev, `mise.cicd.toml` for CI); install script verification with GPG + Cosign; build signing for macOS/Windows; no-proxy build verification (`GOPROXY=direct`).	High.

A.20.5. Overall Assessment

Terragrunt has the most extensive testing infrastructure observed in the corpus across five levels: static analysis (45 linters including 7 test-quality-specific ones), unit testing, integration testing (19+1 CI scenarios partitioned by build tags), fuzz testing with automatic target discovery, and benchmarks. The 7 test-quality linters (paralleltest, tparallel, testifylint, testpackage, usetesting, testableexamples, thelper) enforce testing conventions

through static analysis. A standalone flaky test detection module in `test/flake/` queries GitHub Actions logs, parses failure logs, and generates resolution plans. Supply chain verification tests check GPG and Cosign signatures of distributed binaries. Integration tests run against both Terraform and OpenTofu via build tags. The fuzz suite covers the filter expression parser with three targets (FuzzParse, FuzzLexer, FuzzParser), each run for 30 seconds in CI with one seed input per target. The main gaps are the absence of state migration or drift detection testing and no automated upgrade path validation.

A.21. Evaluation of IaC Testing Practices in pulumi-cdk

A.21.1. Repository Overview

The `pulumi-cdk`[21] library is a TypeScript interop layer that enables AWS CDK constructs to be used within Pulumi programs. It translates CDK construct trees into Pulumi resource graphs, resolves CloudFormation intrinsic functions, and maps CloudFormation resource types to Pulumi providers (primarily AWS Cloud Control). The repository is maintained by Pulumi and published as `@pulumi/cdk` on npm.

The repository was selected because it represents a bridge between two IaC ecosystems, requiring testing of both translation correctness (CDK-to-Pulumi graph conversion, intrinsic resolution) and deployed infrastructure behaviour. This dual requirement produces a distinctive two-tier testing architecture combining mock-based TypeScript unit tests with cloud-provisioning Go integration tests.

A.21.2. Testing Architecture

Unit tests are written in TypeScript using Jest (configured via `package.json` with the ts-jest preset), while integration tests are written in Go using Pulumi's `integration.ProgramTest` framework. The unit test suite comprises 19 test files in the `tests/` directory, organised by functional area: resource registration (`basic.test.ts`), dependency graph construction (`graph.test.ts`), resource naming (`naming.test.ts`), CloudFormation-to-Pulumi type mapping (`cfn-resource-mappings.test.ts`), and a `converters/` subdirectory containing tests for the core translation engine. A shared mock infrastructure (`tests/mocks.ts`) provides Pulumi runtime mocks simulating AWS API responses without cloud credentials.

The repository maintains two separate integration test suites, each with its own Go test runner: `integration/` contains 17 scenario directories covering specific AWS service integrations, and `examples/` contains 19 higher-level example programs. Both suites are run independently in CI via the reusable `acceptance-tests.yml` workflow. One integration scenario (`TestApiGatewayDomain`) is permanently skipped with an inline `t.Skip` comment noting that it requires a valid public Route53 domain not available in the CI account.

[21] `https://github.com/pulumi/pulumi-cdk`

The CI/CD pipeline comprises 10 YAML files in `.github/workflows/` implementing ChatOps-gated acceptance testing, parallel test splitting across three runners via `go-test-split-action`, scheduled nightly runs, and security-conscious separation between internal and external contributors.

A.21.3. Test Suite Coverage

Table A.40 summarises the test suite coverage.

Table A.40.: Test suite coverage for pulumi-cdk.

Layer	**Scope**	**Approach**
Static analysis	ESLint (TypeScript parser), Prettier, TypeScript compiler	ESLint extends `eslint:recommended` and `@typescript-eslint/recommended` with Prettier integration; linting and formatting integrated into `yarn build`; dedicated CI composite action (`.github/actions/lint`)
Unit testing	19 test files: resource registration, graph construction, intrinsic resolution, type mapping, naming, converters, synthesiser, options, interop	Jest with ts-jest; Pulumi runtime mocks (`pulumi.runtime.setMocks`); filesystem mocking (mock-fs); parameterised tests (`test.each`) for intrinsics and graph edges; JSON test data fixtures; coverage reporting (JSON, LCOV, Clover, Cobertura, text); JUnit reporter
Integration testing	17 scenario directories in `integration/`: API Gateway, CloudFront, EC2, KMS, Kinesis, Route53, Secrets Manager, SSM, custom resources, nested stacks, removal policy, errors, etc. (1 test permanently skipped); 19 examples in `examples/` with separate Go test runner	Go `integration.ProgramTest` framework; real AWS resource provisioning; `ExtraRuntimeValidation` callbacks for output assertions; `AssertHTTPResultWithRetry` for endpoint validation; `EditDirs` for multi-step lifecycle tests; AWS SDK direct assertions (S3 bucket existence); `RetryFailedSteps: true` for transient provisioning failures

Table A.40.: Test suite coverage for pulumi-cdk. (continued)

Layer	Scope	Approach
Error testing	Invalid CDK configurations, unsupported CloudFormation types, missing mappings, unresolvable references	Unit: parameterised error cases in converter tests; Integration: `ExpectFailure` with stderr capture (`errors-test`, `unsupported-error`)

A.21.4. Evaluation Against IaC Testing Dimensions

Table A.41 lists the ratings across all eight dimensions.

Table A.41.: Evaluation of pulumi-cdk against IaC testing dimensions.

Dimension	Evidence	Assessment
Static analysis	ESLint with TypeScript parser extending `eslint:recommended` and `@typescript-eslint/recommended`; Prettier for formatting (120-char width, trailing commas, single quotes); both integrated into `yarn build` pipeline; dedicated CI lint composite action; no security scanning tools.	Moderate.
Logic-level unit tests	19 Jest test files across resource registration, graph construction, intrinsic resolution, type mapping, naming, converters, synthesiser, and options; parameterised testing (`test.each`) covering CloudFormation intrinsic function variants and dependency edge types; Pulumi runtime mocks simulating account IDs, regions, partitions, AZs, SSM, Secrets Manager, and ECR responses; filesystem mocking via mock-fs for CDK assembly manifests; coverage collection in 5 formats; JUnit reporting.	High.

Table A.41.: Evaluation of pulumi-cdk against IaC testing dimensions. (continued)

Dimension	Evidence	Assessment
Integration testing	17 Go-orchestrated scenarios in `integration/` deploying real AWS resources (EC2, KMS, S3, Route53, API Gateway, CloudFront, Kinesis, Secrets Manager, SSM, custom resources, nested stacks); `TestApiGatewayDomain` permanently skipped due to Route53 domain unavailability in CI; output validation via `ExtraRuntimeValidation`; HTTP endpoint verification with retry; multi-step lifecycle tests via `EditDirs`; parallel splitting across 3 CI runners via `go-test-split-action`; `gotestsum` with GitHub Actions formatting; `RetryFailedSteps: true` for transient failures in AWS Cloud Control provisioning.	High.
End-to-end / outcome-based	HTTP endpoint validation for custom resource and nested stack tests (`AssertHTTPResultWithRetry`, 60-second timeout); AWS SDK assertions checking S3 bucket existence and deletion for removal policy lifecycle; error message verification via stderr capture; stack output assertions (bucket names, stream names, repository names).	Moderate–High.
State evolution	Multi-step tests via `EditDirs` for replace-on-changes, SSM parameter updates, and removal policy lifecycle; `TestRemovalPolicy` validates retain-then-destroy sequence with intermediate AWS SDK verification; no dedicated drift detection or migration path testing.	Moderate.

Table A.41.: Evaluation of pulumi-cdk against IaC testing dimensions. (continued)

Dimension	Evidence	Assessment
Test fixtures and isolation	Pulumi runtime mocks requiring no cloud credentials for unit tests; mock-fs for filesystem isolation; prefix-based resource naming (from `GITHUB_SHA` or random) preventing integration test collisions; each integration scenario is a self-contained Pulumi project with own `package.json` and `Pulumi.yaml`; JSON test data fixtures (`tests/test-data/`) for deterministic graph and converter testing.	High.
Negative / robustness	Unit tests include parameterised error cases for missing CloudFormation mappings, absent mapping sections, incorrect parameter counts, missing keys at multiple nesting levels, unresolvable cross-stack references, and unmapped resources; integration tests include dedicated error scenarios (`errors-test`, `unsupported-error`) with `Expect-Failure` and stderr verification of specific error messages.	Moderate–High.
CI/CD integration	10 YAML workflow files: `main.yml` (push + nightly schedule, runs both `examples` and `integration` test suites), `run-acceptance-tests.yml` (PR + ChatOps dispatch with sentinel aggregation), `command-dispatch.yml` (slash command for external PRs), `pull-request.yml` (fork PR notification), `acceptance-tests.yml` (reusable workflow with 3-way parallel matrix), `release.yml` (tag-triggered), plus `cdk-validate-agentic-workflows.yml`, `copilot-setup-steps.yml`, and two PR review lock files; ChatOps gating restricts cloud tests to trusted contributors; sentinel job aggregates matrix results for branch protection; Renovate for automated dependency updates; mise for tool version pinning.	High.

A.21.5. Overall Assessment

The dual-language testing approach corresponds to the library's bridging role: TypeScript/Jest unit tests cover CDK-to-Pulumi translation correctness (including CloudFormation intrinsic function variants via parameterised `test.each`), while Go integration tests deploy translated constructs to real AWS infrastructure. The parameterised intrinsic function testing covers compositions and error cases in a structured way not observed in other repositories in the corpus. Multi-step lifecycle tests validate removal policies (retain-then-destroy with intermediate AWS SDK verification), resource replacement, and SSM parameter updates. CI uses ChatOps gating for external contributors with sentinel status aggregation across dynamic parallel matrices, plus nightly scheduled runs for cloud tests. A notable observation from code inspection is that one integration test (`TestApiGatewayDomain`) is permanently skipped due to a Route53 domain requirement not available in the CI account. The repository also maintains a separate `examples/` directory with 19 example programs and its own Go test runner, which runs as a distinct CI job alongside the `integration/` suite. Gaps include no security scanning tools and no drift detection testing.

A.22. Evaluation of IaC Testing Practices in ansible.netcommon

A.22.1. Repository Overview

The `ansible.netcommon`[22] repository is an Ansible collection providing foundational network automation code: connection plugins (`network_cli`, `httpapi`, `netconf`, `libssh`, `grpc`, `persistent`), a default cliconf plugin, modules for CLI command execution, NETCONF/RESTCONF/gRPC operations, and filter plugins for parsing and transforming network output. Maintained by the Ansible Network Community under Red Hat stewardship and published on Ansible Galaxy, it serves as a dependency for vendor-specific collections (e.g., `cisco.nxos`, `cisco.iosxr`).

The repository was selected because it represents a network automation library not covered by the other repositories in the corpus. Unlike infrastructure-provisioning modules that create cloud resources, `ansible.netcommon` automates configuration and state retrieval on existing network devices through multiple transport protocols, producing distinctive testing requirements centred on protocol-level correctness and multi-vendor device compatibility.

A.22.2. Testing Architecture

Unit tests are written in Python using pytest with `MagicMock` and a custom `DictDataLoader` for filesystem simulation. The unit test suite comprises 19 test files covering connection plugins, filter plugins, cliconf, action plugins, module utilities, and CLI parsing.

[22] `https://github.com/ansible-collections/ansible.netcommon`

A shared mock infrastructure in `tests/unit/mock/` provides helpers for path, process environment, vault, and YAML operations. Integration tests are executed against real virtual network devices provisioned through Cisco Modeling Labs (CML), deploying NX-OS 9000v, IOS-XR 9000v, Catalyst 8000v/IOS-XE, and Ubuntu nodes connected via an unmanaged switch with an external connector. Dynamic inventory is generated from CML DHCP-assigned IPs with computed port-forwarding offsets (SSH: +2000, HTTPS: +4000, HTTP: +8000, NETCONF: +3000). The 8 active integration test targets cover CLI operations (`cli_tests`, `cli_parse` with 5 parser engines), HTTP API interactions (`httpapi_tests`), NETCONF operations (`netconf_config`, `netconf_get`, `netconf_rpc`), and RESTCONF operations (`restconf_config`, `restconf_get`). Two gRPC targets (`grpc_config`, `grpc_get`) exist under `tests/integration/hold/` and are not currently run. The CI/CD pipeline comprises 6 GitHub Actions workflow files implementing two-tier delegated workflows, label-gated CML integration, SonarCloud quality monitoring, multi-version sanity testing, and sentinel job aggregation.

A.22.3. Test Suite Coverage

Table A.42 summarises the test suite coverage.

Table A.42.: Test suite coverage for ansible.netcommon.

Layer	**Scope**	**Approach**
Static analysis	Python code quality, YAML, formatting; Ansible sanity; SonarCloud	flake8 (120-char line length, per-file ignores), black (100-char target), isort, prettier, ansible-lint (production profile), pre-commit-hooks; SonarCloud scanning `plugins/` against `tests/unit/`; `ansible-test sanity` across Ansible 2.14–2.21 via delegated workflows

Table A.42.: Test suite coverage for ansible.netcommon. (continued)

Layer	Scope	Approach
Unit testing	19 pytest files: connection plugins, filter plugins, cliconf, action plugins, module utilities, CLI parsing	pytest with `MagicMock` and custom `DictDataLoader`; parameterised connection plugin tests covering combinatorial SSH option spaces (`look_for_keys` × `password` × `private_key_file` × `ssh_type`); mock-imported `ncclient` for NETCONF; large fixture-based filter validation (`test_pop_ace.py`: 1,325 lines); parallel execution (`-n 2`); coverage via pytest-cov to Codecov and SonarCloud
Integration testing	8 active targets: CLI (NX-OS), HTTPAPI (NX-OS NXAPI), NETCONF (IOS-XR), RESTCONF (IOS-XE), multi-parser CLI parsing (Ubuntu, NX-OS)	CML-provisioned virtual network devices; dynamic inventory from DHCP IPs with computed port-forwarding; multi-protocol validation; 5 parser engines (native, pyATS, NTC Templates, TextFSM, XML); device preparation targets resetting configuration state
Error testing	Invalid OS detection, unsupported cliconf methods, format errors, duplicate resources, unsupported sources	Unit: error paths for invalid network OS, unsupported methods, `check_rc` variations; Integration: `ignore_errors: true` with subsequent error message assertions for RESTCONF duplicates, NETCONF unsupported sources, invalid JSON formats

A.22.4. Evaluation Against IaC Testing Dimensions

Table A.43 lists the ratings across all eight dimensions.

Table A.43.: Evaluation of ansible.netcommon against IaC testing dimensions.

Dimension	Evidence	Assessment
Static analysis	flake8 (120-char line length, per-file ignores), black (100-char target), isort, prettier, ansible-lint, pre-commit-hooks configured in `.pre-commit-config.yaml`; SonarCloud scanning `plugins/` against `tests/unit/` with PR-specific parameters; multi-version Ansible sanity testing (2.14–2.21) via delegated workflows; yamllint configuration present (`.yamllint`); no dedicated security scanning tools.	High.
Logic-level unit tests	19 pytest files across connection plugins, filter plugins, cliconf, action plugins, module utilities, and CLI parsing; parameterised tests covering combinatorial spaces (SSH key type × password × private key × SSH implementation); mock-imported `ncclient` for NETCONF session testing; large fixture-based filter validation (1,325-line ACL transformation tests); custom `DictDataLoader` for filesystem simulation; parallel execution via pytest-xdist (`-n 2`); coverage reporting to Codecov and SonarCloud. Gaps: httpapi, grpc, and persistent connection plugins lack unit tests; most modules are untested at unit level.	Moderate–High.
Integration testing	CML-provisioned virtual devices (NX-OS 9000v, IOS-XR 9000v, Catalyst 8000v/IOS-XE, Ubuntu); multi-protocol coverage mapping connection plugins to vendor platforms (`network_cli` → NX-OS, `httpapi` → NX-OS NXAPI, `netconf` → IOS-XR, `restconf` → IOS-XE); dynamic inventory from DHCP IPs with computed port-forwarding offsets; 5 parser engines validated; label-gated security boundary for CML credentials.	High.

Table A.43.: Evaluation of ansible.netcommon against IaC testing dimensions. (continued)

Dimension	Evidence	Assessment
End-to-end / outcome-based	Device response validation with field-level assertions (IP addresses, interface names, enabled status); diff content verification for NETCONF configurations; structured JSON and XML response checking; RESTCONF resource state validation after CRUD operations. No operational behaviour validation (e.g., traffic passing through configured interfaces).	Moderate.
State evolution	RESTCONF CRUD lifecycle with idempotent deletion assertions (DELETE → `changed: true`, DELETE again → `changed: false`); NETCONF confirmed commit with `rollback-on-error` (`confirm: 10`); no multi-step evolution tests with change-count assertions; no migration testing for collection version upgrades.	Low–Moderate.
Test fixtures and isolation	`MagicMock` and `DictDataLoader` for unit tests; mock-imported `ncclient` avoiding dependency requirements; CML lab lifecycle (`lab-create` → `integration-tests` → `lab-destroy` with `always()` condition); device preparation targets resetting configuration state. Fixed resource naming (Loopback42518 in RESTCONF tests, Loopback998 in CLI tests, Loopback999 in NETCONF tests) rather than randomised identifiers.	Moderate.

Table A.43.: Evaluation of ansible.netcommon against IaC testing dimensions. (continued)

Dimension	Evidence	Assessment
Negative / robustness	Unit: invalid network OS detection, unsupported cliconf method rejection, `check_rc` error handling variations, invalid version comparisons. Integration: RESTCONF duplicate resource error assertions ("object already exists"), NETCONF unsupported startup source errors, invalid JSON string format detection, unsupported filter type errors. Limited systematic coverage of edge cases (malformed XML, connection timeouts, authentication failures).	Moderate.
CI/CD integration	6 GitHub Actions workflows; two-tier delegation to organisation-wide (`ansible/ansible-content-actions`) and team-specific (`ansible-network/github_actions`) reusable workflows; "safe to test" label gate for CML integration; SonarCloud via `sonar` job in `tests.yml`; `all_green` sentinel job aggregating delegated workflow results; daily scheduled test runs; release pipeline publishing to Galaxy and Automation Hub; daily token refresh workflow.	High.

A.22.5. Overall Assessment

Among the repositories in the corpus, `ansible.netcommon` is the one that provisions and tests against physical-class virtual network devices. Where most repositories use containers or mock services, this collection provisions Cisco NX-OS 9000v, IOS-XR 9000v, Catalyst 8000v/IOS-XE, and Ubuntu virtual devices through Cisco Modeling Labs. This enables protocol-level validation (CLI, HTTPAPI, NETCONF, RESTCONF) that would not be possible through mocking. The multi-protocol integration test suite maps connection plugins to appropriate vendor platforms. Static analysis combines pre-commit tools (flake8, black, isort, prettier, ansible-lint) with SonarCloud monitoring and Ansible sanity testing across core versions 2.14–2.21. CI uses a two-tier workflow delegation approach: organisation-wide templates from `ansible/ansible-content-actions` layered with network-team-specific actions, plus a "safe to test" label gate protecting CML credentials. Gaps include limited unit test coverage for several connection plugins (httpapi, grpc, persistent), deferred gRPC integration test targets (placed in `hold/` rather

than active targets), fixed resource naming (not randomised), and limited state evolution testing beyond RESTCONF CRUD and NETCONF confirmed commit.

A.23. Evaluation of IaC Testing Practices in yor

A.23.1. Repository Overview

The `yor`[23] repository is a Go-based CLI tool developed by Bridgecrew that automatically tags IaC resources with metadata such as git commit hashes, last modified timestamps, author information, and traceability identifiers. It supports Terraform (HCL), CloudFormation (YAML/JSON), and Serverless Framework files, parsing resources, resolving git blame information, generating or updating tags, and writing modified files back. The tool supports custom tags via a Go plugin system and externally-defined tag groups via YAML configuration, and is distributed as multi-platform binaries via GoReleaser, Homebrew tap, and Docker image.

The repository was selected because it represents an IaC tooling project—a static analysis and code modification tool that operates on IaC files rather than provisioning infrastructure. Testing such a tool involves validating multi-format parsing, tag generation logic, git blame resolution, file writing correctness, and plugin extensibility.

A.23.2. Testing Architecture

Unit tests (23 test files, approximately 5,512 LOC co-located with source in `src/`) validate individual components: three IaC format parsers (Terraform, CloudFormation, Serverless), four tag group types (git, code2cloud, simple, external), JSON and YAML writers, the runner orchestration layer, report generation, CODEOWNERS parsing, CLI options, git service, and utility functions. Table-driven tests cover module source parsing (10 provider extraction variants across registry, git, and GitHub formats in `TestExtractProviderFromModuleSrc`; 6 subdirectory extraction variants in `TestExtractSubdirFromRemoteModuleSrc`). The runner tests exercise the Go plugin extensibility mechanism by compiling `.so` shared libraries from test plugin source at build time and loading them at runtime. Integration tests (481 lines in `tests/integration/`) exercise the complete tagging workflow: programmatic Git repository creation with multi-commit sequences via `go-git/v5`, and external repository cloning at pinned commits (`terragoat` at `063dc2db`, `terraform-aws-bridgecrew-read-only` at `a8686215`, and `JamesWoolfenden/terraform-aws-activemq` at `05ab598c`), with selective tag/skip-tag verification, idempotent re-tagging, uncommitted change detection, minor change selective update, and local module tagging. Integration tests run with the `-race` flag in CI. The CI/CD pipeline comprises five GitHub Actions workflow files implementing environment-gated security scanning, coverage badge generation, GoReleaser multi-platform release, Docker Hub publishing, and downstream project notification.

[23] `https://github.com/bridgecrewio/yor`

A.23.3. Test Suite Coverage

Table A.44 summarises the test suite coverage.

Table A.44.: Test suite coverage for yor.

Layer	Scope	Approach
Static analysis	Go code quality, security scanning, credential detection, IaC file scanning	golangci-lint (13 unique linters, with `gocritic` listed twice in config); 7 pre-commit hook repositories (credential detection, Checkov, shell lint, gofmt, goimports, go-test); gosec + TruffleHog + Checkov secrets in reusable security workflow; CodeQL weekly + PR analysis
Unit testing	23 test files: 3 format parsers, 4 tag group types, JSON/YAML writers, runner, reports, CODEOWNERS, CLI, git service, utilities	Go testing with testify/assert; table-driven parameterised tests (10 provider extraction + 6 subdirectory extraction variants); Go plugin `.so` compilation and dynamic loading; `testing/fstest.MapFS` for filesystem simulation; mock git blame generation; coverage profiling
Integration testing	Complete tagging workflow: multi-commit Git repos, external repo tagging, selective tags, idempotent re-tagging, uncommitted changes	Programmatic Git repository creation via `go-git/v5`; three external repos cloned at pinned commits (`terragoat`, `terraform-aws-bridgecrew-read-only`, `terraform-aws-activemq`) with JSON report validation; `-race` flag for concurrency detection; deferred cleanup of temp directories and cloned repos

Table A.44.: Test suite coverage for yor. (continued)

Layer	Scope	Approach
Error testing	Malformed files, tag collisions, skip-by-comment, non-serverless rejection, directory skip warnings	Malformed HCL error assertion without crash; collision detection returning error; warning message capture via stdout interception; no systematic edge case coverage (corrupt Git, permissions, concurrency)

A.23.4. Evaluation Against IaC Testing Dimensions

Table A.45 lists the ratings across all eight dimensions.

Table A.45.: Evaluation of yor against IaC testing dimensions.

Dimension	Evidence	Assessment
Static analysis	golangci-lint with 13 unique linters (errcheck, gosimple, govet, ineffassign, staticcheck, typecheck, unused, gofmt, goimports, gocritic, unconvert, unparam, revive; note: `gocritic` appears twice in the config file); 7 pre-commit hook repositories (credential detection, Checkov, shell lint, gofmt, goimports, go-test); gosec security scanner; TruffleHog secret detection; Checkov secret scanning with Prisma Cloud API; CodeQL weekly + PR analysis. Among the repositories in the corpus, yor has the largest number of distinct security scanning tools in its pipeline.	High.

Table A.45.: Evaluation of yor against IaC testing dimensions. (continued)

Dimension	Evidence	Assessment
Logic-level unit tests	23 test files (∼5,512 LOC) across Terraform/CloudFormation/Serverless parsers, 4 tag group types, JSON/YAML writers, runner, reports, CODEOWNERS, CLI, git service, utilities; table-driven parameterised tests for module source parsing (10 variants); Go plugin compilation and loading; `testing/fstest.MapFS` for filesystem simulation. Gaps: limited concurrent path coverage.	High.
Integration testing	Complete end-to-end tagging workflow against programmatically created Git repositories via `go-git/v5`; three external repository clones at pinned commits with author/-timestamp verification; selective tag/skip-tag testing; local module tagging; race detection via `-race` flag. No cloud infrastructure provisioning.	High.
End-to-end / outcome-based	JSON report validation with lower-bound tag count assertions on `terragoat` (AWS $\geq$ 312, GCP $\geq$ 32, Azure $\geq$ 160); file modification verification via diff comparison; CLI output parsing for `list-tags` command. No downstream tool validation (tagged files not checked against `terraform validate` or `cfn-lint`).	Moderate.
State evolution	Two-step idempotent re-tagging (11 tags present after second run in `TestTagUncommittedResults`); minor change detection with selective metadata update; multi-commit tag evolution across 3 sequential commits with hash stability and trace preservation assertions. No version migration testing.	Moderate.

Table A.45.: Evaluation of yor against IaC testing dimensions. (continued)

Dimension	Evidence	Assessment
Test fixtures and isolation	Format-specific fixture directories (Terraform, CloudFormation, Serverless) with paired input/expected files; `os.MkdirTemp` with deferred removal; deferred file restoration for modified sources; `testing/fstest.MapFS` for in-memory filesystem; three external repos cloned to temp paths. Network dependency on external repositories for integration tests.	Moderate–High.
Negative / robustness	Malformed Terraform file handling (error without crash); tag collision detection; skip-by-comment exclusion (Terraform + CloudFormation); non-serverless file rejection; directory skip warning verification; resource skip by ID. No tests for corrupt Git repos, permission errors, concurrent file access, or invalid plugin configurations.	Low–Moderate.
CI/CD integration	5 GitHub Actions workflows; security scanning gated behind `scan-security` environment approval using `pull_request_target`; reusable `security-shared.yml` workflow; coverage badge auto-generation committed to README; GoReleaser multi-platform release (linux/darwin/windows × 386/amd64/arm/arm64, minus darwin/386); Docker Hub publish with README sync; downstream repository dispatch (`yor-action`, `yor-choco`, Jenkins); Dependabot for Actions; CodeQL scheduled + PR; self-hosted runners for integration tests.	High.

A.23.5. Overall Assessment

Golden file comparison is a central practice in yor's testing: unit tests insert tags into Terraform HCL, CloudFormation YAML/JSON, and Serverless YAML files and compare the output against stored reference files, with the aim of checking that tagging preserves the original formatting. Multi-commit integration tests create temporary git repositories with

programmatic commits to check that `yor_trace` identifiers persist across re-tagging while git metadata tags update correctly. Plugin architecture testing compiles and loads Go plugins at runtime, exercising the extensibility mechanism. The security scanning pipeline combines gosec, TruffleHog, Checkov secrets, and CodeQL with environment approval gates for credential-sensitive PR scans. A notable observation from code inspection is that integration tests clone three external repositories at pinned commits (not only `terragoat` as the CI workflow illustrates, but also `terraform-aws-bridgecrew-read-only` and `terraform-aws-activemq`), introducing a network dependency on external repositories at test runtime. Gaps include limited negative testing (no corrupted repository or concurrent access scenarios), no migration testing for tool version upgrades, and a duplicate `gocritic` entry in the linter configuration yielding 13 unique linters despite 14 listed entries.

A.24. Evaluation of IaC Testing Practices in crossplane

A.24.1. Repository Overview

The `crossplane`[24] repository is a CNCF graduated project implementing a Kubernetes-native control plane framework for composing cloud infrastructure and services. Written in Go, Crossplane extends Kubernetes with Custom Resource Definitions (CRDs) that enable declarative infrastructure composition via XRDs, Compositions, and Functions, managing the lifecycle of composed resources through reconciliation loops and gRPC-based pipeline execution. The project is distributed via Helm chart, multi-architecture OCI images, and a CLI tool (`crank`).

The repository was selected because it represents a Kubernetes-native IaC control plane—a paradigm distinct from declarative configuration tools (Terraform, Pulumi) or configuration management (Ansible). Testing a Kubernetes controller involves validating reconciliation loops, CRD schema generation, webhook admission control, RBAC rule generation, package dependency resolution, and composition pipeline execution, producing distinctive testing requirements centred on controller-runtime conventions and Kubernetes API semantics.

A.24.2. Testing Architecture

Unit tests (138 test files in the main codebase, with a further 38 test files in the temporarily vendored `tmp/crossplane-runtime` local module, totalling 176) validate 20 reconciler controllers across five domains (apiextensions, pkg, rbac, protection, ops), composition function pipeline execution with protobuf serialisation, CRD schema generation, package management, DAG operations, CLI commands, circuit breaker logic, SSA managed fields, and initialiser components. All tests follow a consistent table-driven approach with explicit reason strings, using the `crossplane-runtime/v2/pkg/test` MockClient (imported in 69 files within `internal/`) and `google/go-cmp` for comparisons; third-party assertion libraries (`testify`, `gomega`, `ginkgo`) are banned via `depguard` linter rules. Fuzz tests cover

[24] `https://github.com/crossplane/crossplane`

9 functions across 8 files in the main codebase (`internal/xcrd/fuzz_test.go` contributes two targets: `FuzzForCompositeResourceXcrd` and `FuzzForCompositeResourceClaim`), plus one additional target in `tmp/crossplane-runtime`; all are registered with Google's OSS-Fuzz for continuous fuzzing (300-second CI runs with automatic target discovery and crash artefact archiving). E2E tests (51 test functions across 11 files, approximately 6,235 LOC of test code, helpers, and configuration, plus 283 YAML manifests across 160 directories) deploy Crossplane into ephemeral Kind clusters via Helm, testing compositions (revision selection, environment configs, function pipelines, circuit breakers), package management (installation, private registries, signature verification, dependency upgrades), protection (usage-based deletion blocking with webhook rejection), lifecycle upgrades (prior-version install, claim creation, upgrade, survival verification), and operations (CronOperations, WatchOperations). The CI/CD pipeline comprises 9 GitHub Actions workflows using Nix flakes as the primary build system with Cachix binary caching for hermetic reproducibility.

A.24.3. Test Suite Coverage

Table A.46 summarises the test suite coverage.

Table A.46.: Test suite coverage for crossplane.

Layer	**Scope**	**Approach**
Static analysis	Go code quality, Helm chart, shell scripts, Nix files, protobuf schemas, security, vulnerability scanning	golangci-lint (`default: all` minus 24 documented exclusions); Helm lint; shellcheck + shfmt; statix + deadnix + nixfmt; CodeQL; Trivy (filesystem + scheduled daily image scanning); buf protobuf lint; generated code verification via Nix sandbox
Unit testing	138 test files in main code (+ 38 in `tmp/`): 20 reconcilers, composition functions, CRD generation, package management, DAG, CLI, circuit breaker, SSA, initialiser	Go testing with `google/go-cmp`; `depguard`-banned assertion libraries; table-driven with reason strings; MockClient from crossplane-runtime; protobuf pipeline testing; Codecov coverage

Table A.46.: Test suite coverage for crossplane. (continued)

Layer	Scope	Approach
Fuzz testing	9 functions in main code: DAG, packages, CRDs, claims, compositions, RBAC, package manager, XCrd (two targets in one file)	Go native `testing.F`; OSS-Fuzz continuous integration; 300s per run; automatic target discovery; crash artefact archiving
Integration / E2E	51 test functions: compositions, packages, protection, lifecycle upgrades, operations, XRD/MRD validation	Kind cluster deployment; Helm install/upgrade; 283 YAML manifests; matrix execution (11 combinations × 3 retries); BuildPulse flake detection; 45-minute timeout

A.24.4. Evaluation Against IaC Testing Dimensions

Table A.47 lists the ratings across all eight dimensions.

Table A.47.: Evaluation of crossplane against IaC testing dimensions.

Dimension	Evidence	Assessment
Static analysis	golangci-lint `default: all` minus 24 documented top-level exclusions (each with an inline rationale comment); `depguard` banning test libraries; Helm lint; shellcheck + shfmt; statix + deadnix + nixfmt; CodeQL; Trivy filesystem scan in CI + scheduled daily image scanning of 3 most recent releases; buf protobuf lint; generated code Nix verification. Seven analysis categories.	High.

Table A.47.: Evaluation of crossplane against IaC testing dimensions. (continued)

Dimension	Evidence	Assessment
Logic-level unit tests	138 test files in main crossplane code (176 including the temporarily vendored `tmp/crossplane-runtime`), spanning 20 reconciler controller test suites with MockClient; protobuf composition function testing; CRD schema; DAG; packages; CLI; circuit breaker; SSA. Consistent table-driven approach with reason strings; third-party assertion libraries banned via `depguard`; `google/go-cmp` for all comparisons.	High.
Integration testing	51 E2E functions (50 test functions + `TestMain`) deploying into Kind clusters; Helm install/upgrade; manifest-driven scenarios with CRDs, XRDs, Compositions, Functions, Providers, Claims; condition polling; field assertions; webhook rejection; real Kubernetes API via ephemeral Kind clusters.	High.
End-to-end / outcome-based	Complete user workflows: XR creation produces composed resources; package install produces healthy providers; composition updates propagate; upgrades preserve claims. Lifecycle E2E validates the full upgrade path. No cloud provider validation (nop provider used throughout).	High.
State evolution	Lifecycle upgrade E2E (prior version → current, configured via `CROSSPLANE_PRIOR_VERSION`); composition revision propagation; package dependency upgrades (version, digest, transitive, downgrade); DAG upgrading; runtime migration. No CRD schema versioning evolution testing.	Moderate–High.
Test fixtures and isolation	283 YAML manifests in 160 scenario directories; ephemeral Kind clusters destroyed after each run; Nix sandbox hermetic isolation for unit tests; testdata co-located with source. E2E requires Docker and network access.	High.

Table A.47.: Evaluation of crossplane against IaC testing dimensions. (continued)

Dimension	Evidence	Assessment
Negative / robustness	9 fuzz targets (OSS-Fuzz continuous integration); reconciler error paths (NotFound, GetError, UpdateError); composition function errors; validation rejection (XRD, MRD, composition); webhook rejection; dependency error scenarios; lack-of-rights handling.	Moderate–High.
CI/CD integration	9 workflows; Nix hermetic builds with Cachix binary caching; matrix E2E (11 combinations × 3 retries); Codecov; BuildPulse flake detection; Trivy scheduled scanning; CodeQL; buf; multi-registry push (DockerHub, ghcr.io, xpkg.upbound.io); S3 artefact promotion; Renovate; backport automation.	High.

A.24.5. Overall Assessment

Two practices observed in crossplane are not present in other repositories in the corpus. First, the Nix-based hermetic CI approach: all unit tests run inside the Nix sandbox with no network or filesystem access beyond what Nix explicitly permits, and Cachix binary caching at package granularity means unchanged build outputs are retrieved from cache rather than recomputed. Second, the `depguard` configuration bans third-party assertion libraries (`testify`, `ginkgo`, `gomega`) outright, requiring all tests to use Go's standard testing package with `google/go-cmp` for comparisons and explicit reason strings in table-driven tests. Nine fuzz targets in the main codebase (including two in `internal/xcrd/fuzz_test.go`) are registered with Google's OSS-Fuzz for continuous fuzzing beyond CI runs, with 300-second runs per target and automatic crash artefact archiving. E2E tests deploy into Kind clusters with lifecycle upgrade validation (install prior version, create claims, upgrade, verify claims survive), and BuildPulse tracks flake rates across 11 matrix combinations run with 3 retries each. An observable detail from code inspection is that `tmp/crossplane-runtime` is a temporarily vendored local copy of the crossplane-runtime module (referenced via a `replace` directive in `go.mod`), adding 38 test files to the unit test scope. Gaps include no real cloud provider testing in E2E (nop provider only), limited CRD schema versioning testing, and Docker and network requirements for E2E execution.

A.25. Evaluation of IaC Testing Practices in cloudformation-cli

A.25.1. Repository Overview

The `cloudformation-cli`[25] repository is the official AWS CloudFormation CLI tooling (also known as `cfn` or the CloudFormation Resource Provider Development Kit). Written in Python, it provides a framework for developing, testing, and registering CloudFormation resource providers, modules, and hooks. The tool generates project scaffolding, validates resource schemas against JSON Schema specifications, runs contract tests against provider handlers (CREATE, READ, UPDATE, DELETE, LIST), packages providers for submission to the CloudFormation registry, and generates documentation from schemas. It is published as a PyPI package and integrates with language-specific plugins for Java, Python, Go, and TypeScript.

The repository was selected because it represents a testing framework for IaC providers—a tool whose primary purpose is to validate that CloudFormation resource providers conform to the CloudFormation contract. This creates a distinctive meta-testing requirement: the test framework itself must be extensively tested, and the contract test suite shipped as part of the tool constitutes both product code and reusable test infrastructure for downstream consumers.

A.25.2. Testing Architecture

Unit tests reside in `tests/` organised by functional area, all using pytest with extensive `unittest.mock` usage (569 mock/patch occurrences in `test_project.py` alone, 1,609 across all test files). The 37 test files total approximately 14,313 LOC; the four primary test files account for 6,934 LOC: `test_project.py` (3,924 LOC) covering project lifecycle, schema validation, documentation generation, submit workflow, packaging, and canary file generation; `test_resource_client.py` (1,935 LOC) covering all lifecycle operations with 16 parametrised test functions; `test_resource_generator.py` (235 LOC) testing hypothesis strategy generation from JSON Schema; and `test_test.py` (840 LOC) providing meta-testing of the `cfn test` command. The contract test suite in `src/rpdk/core/-contract/suite/` is shipped as part of the CLI tool for resource provider developers, creating a dual role as both product code and test infrastructure. The framework uses `contract_` prefixed functions decorated with pytest markers and skip decorators, organised into handler files validating CREATE (5 functions), DELETE (5 functions), UPDATE (3 functions), and invalid UPDATE (1 function) contracts, plus hook contract tests. The hypothesis-based property testing module (`resource_generator.py`, 272 LOC) converts JSON Schema definitions into hypothesis strategies. The `resource_client.py` (870 LOC) implements property pruning for model comparison. Static analysis comprises 7 pre-commit hook categories (isort, black, flake8 with 6 plugins, bandit, pygrep-hooks, pylint, pytest with coverage), a 98% coverage threshold enforced via `.coveragerc` (with the

[25] `https://github.com/aws-cloudformation/cloudformation-cli`

shipped contract suite excluded from measurement via `omit = */contract/suite/*`), and CI enforcement across a 12-combination test matrix. A daily cron workflow auto-syncs CloudFormation schemas from the upstream `aws-cloudformation-resource-schema` repository with automated PR creation.

A.25.3. Test Suite Coverage

Table A.48 summarises the test suite coverage.

Table A.48.: Test suite coverage for cloudformation-cli.

Layer	Scope	Approach
Static analysis	Python code quality, security scanning, coverage enforcement, complexity limits	Pre-commit hooks with 7 tool categories: isort, black, flake8 (6 plugins: bugbear, builtins, commas, comprehensions, debugger, pep3101), bandit (skips B101 for contract tests), pygrep-hooks, pylint, pytest with coverage; 98% coverage threshold; `flake8 max-complexity=10`
Unit testing	Project lifecycle, resource client, resource generator, meta-testing of `cfn test` command; 37 test files (∼14,313 LOC total)	pytest with `unittest.mock`; 569 mock/patch occurrences in `test_project.py`; 16 parametrised functions in `test_resource_client.py`; hypothesis strategy generation tests; golden file comparison for documentation; parameterised across artifact types (RESOURCE, MODULE, HOOK)
Contract testing	Resource provider lifecycle compliance (CREATE, READ, UPDATE, DELETE, LIST); hook invocation compliance	Shipped contract suite with `contract_` functions decorated with pytest markers and skip decorators; hypothesis-based input generation from JSON Schema; property pruning for model comparison (read-only, write-only, create-only with wildcard paths); collection comparison (ordered/unordered); Docker handler execution

Table A.48.: Test suite coverage for cloudformation-cli. (continued)

Layer	Scope	Approach
CI/CD	Multi-platform validation, coverage enforcement, schema synchronisation	12-combination matrix (4 Python versions × 3 OS); pre-commit all files; pytest with coverage; twine PyPI validation; daily cron schema auto-sync with automated PRs; pip and pre-commit caching

A.25.4. Evaluation Against IaC Testing Dimensions

Table A.49 lists the ratings across all eight dimensions.

Table A.49.: Evaluation of cloudformation-cli against IaC testing dimensions.

Dimension	Evidence	Assessment
Static analysis	Pre-commit hooks with 7 tool categories (isort, black, flake8 with 6 plugins, bandit security scanning, pygrep-hooks, pylint, pytest with coverage); 98% coverage threshold enforced via `.coveragerc`; bandit configured to skip B101 for contract tests; AWS credentials set to empty strings to prevent accidental API calls; `flake8 max-complexity=10`.	High.
Logic-level unit tests	37 test files totalling ∼14,313 LOC across project lifecycle, resource client, resource generator, and meta-testing; 569 mock/patch occurrences in `test_project.py`; 16 parametrised test functions in `test_resource_client.py`; hypothesis-based strategy generation from JSON Schema; golden file comparison for documentation output; parameterised across artifact types (RESOURCE, MODULE, HOOK); 98% coverage enforcement.	High.

Table A.49.: Evaluation of cloudformation-cli against IaC testing dimensions. (continued)

Dimension	Evidence	Assessment
Integration testing	Contract test framework validates the complete resource provider lifecycle (CREATE → READ → UPDATE → DELETE → LIST); `cfn test` command orchestrates pytest with markers, overrides (Jinja2 templating), and input files; callback delay mechanism with IN_PROGRESS polling for asynchronous operations. The framework tests the test execution machinery rather than provisioning cloud resources.	Moderate.
End-to-end / outcome-based	Submit workflow validation (dry-run and live); packaging with zip creation and S3 upload mocking; documentation generation with golden file comparison; canary file generation with template variable substitution and JSON patch operations; `cfn test` command orchestration validation. No cloud infrastructure deployment.	Moderate.
State evolution	Contract test lifecycle sequences (create → read → update → delete → list) validating state transitions; `test_input_equals_output` for model consistency after property pruning; IN_PROGRESS callback polling with credential refresh. No migration testing for framework version upgrades or schema evolution.	Low–Moderate.
Test fixtures and isolation	Extensive mocking of boto3, CloudFormation API, STS, S3, Docker, and filesystem; 104 JSON schema fixture files; `tmpdir` fixture for filesystem isolation; AWS credentials set to empty strings; contract suite excluded from coverage (`omit = */contract/suite/*`); parameterised schemas for client testing.	High.

Table A.49.: Evaluation of cloudformation-cli against IaC testing dimensions. (continued)

Dimension	Evidence	Assessment
Negative / robustness	Systematic contract negative tests (NotFound, AlreadyExists, UnsupportedTarget for all lifecycle operations); invalid schemas, missing settings, malformed inputs; path traversal blocking; credential exposure prevention; invalid payloads, boto3 exceptions, waiter failures, assertion mismatches. Limited edge case coverage for complex schema compositions and concurrent execution.	Moderate–High.
CI/CD integration	12-combination test matrix (4 Python versions × 3 OS); 98% coverage enforcement; pre-commit hooks with 7 tool categories; twine PyPI distribution validation; daily cron schema auto-sync from upstream `aws-cloudformation-resource-schema` with automated PR creation; pip and pre-commit caching.	High.

A.25.5. Overall Assessment

This repository occupies a structurally distinct position in the corpus: its contract test suite is both internal test code and a shipped product used by downstream CloudFormation provider developers to validate lifecycle compliance (CREATE → READ → UPDATE → DELETE → LIST). This dual role means the test framework itself must be extensively tested. An observable practice from code inspection is that hypothesis-based property testing converts JSON Schema definitions into hypothesis strategies for automatic input generation; this approach is not present in the other repositories in the corpus. A 98% coverage threshold is enforced in CI via `.coveragerc` with context-aware exclusion of the shipped contract suite. The decorator-based assertion composition framework (`@response_contains_primary_identifier`, `@response_does_not_contain_write_only_properties`, `@failed_event`) enables composable pre/post-condition checks with property pruning for model comparison. Gaps include no cloud infrastructure provisioning tests, limited testing of complex schema compositions, and no migration testing for framework version upgrades.