Mittuniversitetet
MID SWEDEN UNIVERSITY

Addressing the Testing Gap in PL-IaC: A Case Study on Pulumi .NET

Albin Rönnkvist & Piran Amedi

Final Project
Main field of study: Computer Engineering BA (C)
Credits: 15
Semester/year: Spring 2025
Supervisor: Truong Ho-Quang
Examiner: Raja-Khurram Shahzad
Course code/registration number: DT133G
Program: Software Engineering

Addressing the Testing Gap in PL-IaC: A Case Study on Pulumi .NET

Albin Rönnkvist and Piran Amedi
Department of Communication, Quality Management and Information Systems
Mid Sweden University
Östersund, Sweden
{alrn1700, roam2200}@student.miun.se

***Abstract*—Infrastructure as Code (IaC) allows developers to manage infrastructure through code, bringing software engineering practices to infrastructure. However, testing practices have not translated well to infrastructure, as reflected in low adoption rates, particularly within Programming Language-based IaC (PL-IaC). This situation undermines the reliability of infrastructure systems and highlights an urgent need to improve the testing experience. To address the concerning lack of adoption, this thesis investigates the barriers to unit testing in PL-IaC and explores how targeted tooling can encourage wider adoption.**

To achieve this, we conducted a case study on Pulumi .NET, a popular PL-IaC tool with notably low testing adoption. We began with collecting and analyzing developer-reported issues to identify common challenges. Based on these insights, we developed *Pulumock*, a tool aimed at mitigating the identified challenges. We then evaluated Pulumock through a within-subjects study using a mixed-methods approach to assess its ability to mitigate the challenges.

The analysis of issues revealed the most common challenges to be related to mocking complexity and limited access to test data. Pulumock was successfully implemented to target the challenges through focused design choices and features. The within-subjects study further indicated that Pulumock mitigated common challenges by improving perceived adoptability and maintainability compared to Pulumi's default testing framework.

The findings were specific to Pulumi .NET and can only be partially generalized due to fundamental differences in PL-IaC tools. Future research should examine how testing experiences differ across tools, how PL-IaC program design affects unit testability, and the role of unit tests within the infrastructure testing pyramid.

***Index Terms*—Infrastructure as Code (IaC), Programming Language IaC (PL-IaC), Unit Testing, Testing Adoption**

I. Introduction

Infrastructure management, once a manual and error-prone process, has been transformed by the rise of Infrastructure as Code (IaC). Now a standard practice in the cloud era and a cornerstone of DevOps, IaC allows developers to provision and manage infrastructure through code, bringing software engineering principles, practices, and tools into the process. Traditionally, IaC tools have relied on data serialization formats and domain-specific languages (DSLs), with examples including Amazon Web Services (AWS) CloudFormation[1] and Terraform[2]. However, as the field continues to evolve, there is a noticeable shift toward programming language-based IaC (PL-IaC) tools, such as Pulumi[3], AWS Cloud Development Kit (CDK)[4], and CDK for Terraform (CDKTF)[5]. These tools enable developers to define infrastructure using general-purpose programming languages (GPLs) such as TypeScript, Python, and C#, leveraging the full expressiveness of these languages and allowing infrastructure code to more closely resemble application code.

Although IaC tools align infrastructure with software engineering principles, testability remains an underexplored and challenging area. Studies have pointed to low adoption of testing practices, a scarcity of documented testing approaches, and identified testing as one of the most significant challenges faced by IaC practitioners [1]–[5]. This is particularly evident in PL-IaC, with the recent PIPr dataset [6] indicating that only 25% of PL-IaC projects include any form of testing. While a gap in testability has been identified, several studies have simultaneously emphasized the critical importance of testing in IaC, highlighting the potentially severe consequences of undetected defects, including system outages and significant financial losses [1], [4], [7], [8].

Even though PL-IaC tools offer various testing techniques, previous research by Sokolowski et al. [4] suggests that the low adoption reflects a broader perception that infrastructure testing is impractical and cumbersome. To address this, the authors developed *ProTI*[6], a tool that automates multiple aspects of testing in Pulumi. However, nearly two years after its first major release[7],

[1]https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/Welcome.html
[2]https://developer.hashicorp.com/terraform/intro
[3]https://www.pulumi.com/
[4]https://docs.aws.amazon.com/cdk/v2/guide/home.html
[5]https://developer.hashicorp.com/terraform/cdktf
[6]https://proti-iac.github.io/
[7]https://github.com/proti-iac/proti/releases/tag/v1.0.0

the project records only 4 weekly downloads[8] and 9 stars on GitHub[9]. This limited adoption, despite the tool's capabilities, raises important questions: why aren't tools such as ProTI being adopted by developers, and what barriers still discourage developers from testing their infrastructure code?

In this thesis, we build upon the work by Sokolowski et al. [4], [6], to further deepen the understanding of testability challenges in PL-IaC and identify potential solutions that may contribute to greater adoption of testing practices across the PL-IaC ecosystem. We do this through a case study of Pulumi, a well-established platform in the PL-IaC ecosystem, adopted by over 2,700 organizations[10] and garnering over 22,800 stars on GitHub[11]. Pulumi also has the lowest testing adoption among PL-IaC tools according to the PIPr dataset [6], making it a suitable case for examining the testability gap and identifying the challenges developers face. We specifically focus on Pulumi's .NET implementation, as Pulumi is the most popular PL-IaC tool among .NET developers. While Pulumi has a larger user base in other languages such as TypeScript, AWS CDK remains the dominant choice in those communities [6]. We further narrow our focus to unit testing[12] in Pulumi .NET, as it represents the foundation of the testing pyramid[13] and offers an accessible and cost-effective entry point.

To support this effort, we will examine developer-reported challenges in public forums relevant to the specific case. Insights from these sources will guide the design of *Pulumock*, a Pulumi .NET-based tool aimed at mitigating the identified challenges. Finally, we will assess how well Pulumock mitigates the identified challenges, based on perceived adoptability and maintainability relative to Pulumi's default testing framework. While this thesis focuses on a specific case, it also examines its generalizability, with the ultimate goal of finding solutions that promote broader adoption of testing practices within the PL-IaC community.

II. Purpose and Contributions

Testing infrastructure defined as code is widely acknowledged as important [1], [4], [7], [8], yet it remains difficult and is rarely adopted, particularly within the PL-IaC community [6]. The challenge begins at the bottom of the testing pyramid: unit testing, where early and frequent testing should ideally occur. Previous research by Sokolowski et al. [4] found PL-IaC unit testing to be complex, particularly in Pulumi due to the effort required in maintaining accurate mocks. Although the authors developed ProTI to address these challenges, its limited adoption suggests that significant barriers to testing still persist in practice. These difficulties discourage early-stage unit testing, resulting in limited test coverage that, in turn, undermines the reliability of infrastructure code [1], [4], [7], [8].

This thesis aims to address the testability gap in PL-IaC. It specifically investigates the challenges developers encounter when unit testing infrastructure code and examines how targeted tooling can mitigate these challenges. The ultimate goal is to make testing more accessible and encourage broader adoption of testing practices within the PL-IaC community.

To achieve this, we focus on a concrete case: Pulumi, and in particular its .NET instance. We begin by examining developer-reported issues in public forums to understand the current challenges in Pulumi .NET unit testing. Drawing on these insights, we introduce Pulumock, a .NET tool designed to mitigate the identified challenges through focused design and functionality. Finally, we conduct a within-subjects study to evaluate Pulumock's effectiveness in mitigating the identified challenges, based on perceived adoptability and maintainability relative to Pulumi's default testing framework. Since the case revolves around Pulumi .NET unit testing, the research questions are framed around this context, while also offering insights that extend to the broader PL-IaC ecosystem.

- RQ1: What are the common challenges developers face when unit testing Pulumi .NET programs?
- RQ2: In what ways can Pulumock mitigate common challenges when unit testing Pulumi .NET programs?
- RQ3: How does Pulumock compare relative to Pulumi's default testing framework in terms of mitigating common challenges when unit testing Pulumi .NET programs?

III. Background

We begin by introducing IaC (Section III-A) and outlining its evolution to show the relevance of PL-IaC (Section III-B). This is followed by an explanation of PL-IaC (Section III-C) and PL-IaC unit testing (Section III-D).

A. IaC

IaC is the practice of provisioning and managing infrastructure through code. In today's cloud-native era, where infrastructure is highly dynamic, the definition has further evolved as an automated approach to managing cloud infrastructure that embraces continuous change [9]. By codifying infrastructure, developers can apply software engineering principles, practices, and tools to

[8] https://api.npmjs.org/downloads/range/2025-01-01:2025-05-21/@proti-iac/runner

[9] https://github.com/proti-iac/proti

[10] https://www.pulumi.com/case-studies/

[11] https://github.com/pulumi/pulumi

[12] https://martinfowler.com/bliki/UnitTest.html

[13] https://martinfowler.com/articles/practical-test-pyramid.html

infrastructure management, integrating it directly into the development lifecycle. This enables infrastructure changes to be tested, reviewed, and deployed with processes similar to those of application code. As a result, IaC has become a cornerstone of modern DevOps practices, with success stories from companies such as Netflix, Spotify, and Expedia showcasing significant improvements in infrastructure agility, scalability, and reliability [10].

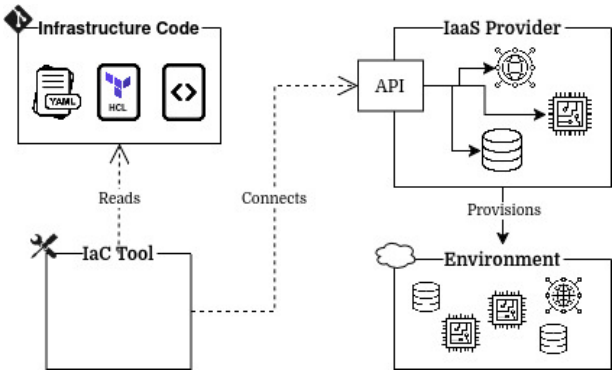


Fig. 1: Interactions between infrastructure code, IaC tool, IaaS provider and an environment.

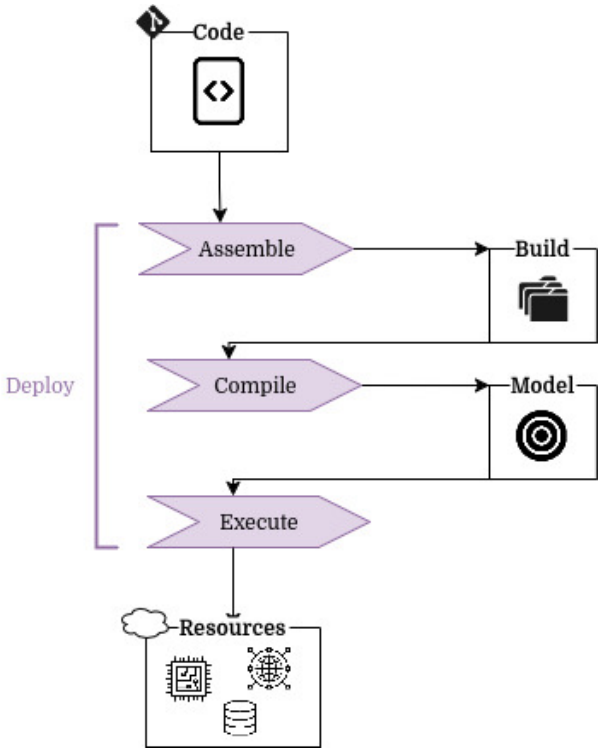


Fig. 2: IaC three-phase deployment workflow.

As illustrated in Figure 1, IaC involves defining infrastructure resources in code, which IaC tools then provision to a target environment via IaaS provider APIs. The process begins with developers writing infrastructure code to define resources such as compute, networking, and storage. These definitions are commonly written in data serialization formats such as YAML or JSON, DSLs such as HashiCorp Configuration Language (HCL), or GPLs such as TypeScript or C#, and are stored in version control systems alongside application code. IaC tools such as Ansible, Terraform, or Pulumi are then used to deploy these definitions, typically following a three-phase workflow as illustrated in Figure 2. First, in the *Assemble* phase, the tool collects the infrastructure code and its dependencies, producing a structured set of files or an artifact—referred to as a *build*. Next, in the *Compile* phase, the build is compiled and executed to generate a *desired state model*, a complete and declarative specification of the infrastructure that should exist, independent of its current state. Finally, in the *Execute* phase, the IaC tool communicates with the IaaS API to compare the desired state with the current state of the environment and apply any necessary changes to achieve the desired state. While the three-phase model may not be explicitly implemented by all IaC tools, it offers a useful conceptual framework for understanding the deployment process [9].

B. Evolution of IaC Tools

In 1993, CFEngine[14] laid the groundwork for modern IaC tools by introducing a desired state model, expressed through a declarative DSL. This methodology was later embraced by Puppet[15] and Chef[16] during the 2000s, and subsequently by Ansible[17] and Salt[18] in the early 2010s.

As IaaS platforms matured in the 2010s and introduced APIs for resource management, the focus shifted to stack-oriented IaC tools. These tools organize infrastructure into modular stacks, grouping related resources together to improve complexity management. Terraform[19] and AWS CloudFormation[20] became the leading solutions in this space, both based on a desired state model and DSLs similar to previous tools.

In the late 2010s, stack-oriented IaC tools began shifting from declarative DSLs to GPLs. Pulumi and AWS CDK led this transition, while CDKTF followed suit, remaining in beta as of early 2025. This shift has blurred the line between infrastructure and application

[14]https://cfengine.com/
[15]https://www.puppet.com/
[16]https://www.chef.io/
[17]https://www.redhat.com/en/ansible-collaborative
[18]https://saltproject.io/
[19]https://developer.hashicorp.com/terraform
[20]https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/Welcome.html

development, reflecting a move toward making infrastructure management more accessible to developers.

More recently, a new paradigm known as Infrastructure from Code (IfC) has begun to emerge. Tools such as Ampt[21], Encore[22], and Shuttle[23] embed infrastructure logic directly into application code, allowing infrastructure to be inferred and deployed as part of the application itself. In a recent study, Aviv et al. [11] argue that IfC can reduce the complexity associated with IaC. Their empirical study demonstrated improvements in developer productivity and deployment speed, albeit on a small scale. However, larger-scale empirical studies are still lacking [11], [12] and as of early 2025 IfC is not considered a mainstream approach [9].

C. PL-IaC

PL-IaC tools allow developers to define infrastructure using familiar GPLs. By leveraging the full capabilities of these languages, they support imperative constructs such as loops and conditionals, while also providing access to native libraries and tooling. To the best of our knowledge, Pulumi, AWS CDK, and CDKTF are the only available PL-IaC tools. They share several characteristics, including a similar hierarchical project structure, reliance on a desired state model, and a comparable deployment workflow. In essence, these tools enable developers to create an IaC *program* that defines a *desired state*, which is then executed and coordinated by a *deployment engine* to bring an environment in line with the desired state.

The heart of PL-IaC tools is the IaC *project*, which contains the source code for an IaC program. In the CDK tools, each project contains at least one *app* instance that encompasses the entire infrastructure definition, while in Pulumi, this role is fulfilled directly by the program file. Within it, one or more *stacks* represent independently deployable units, each containing a set of *resources*. Provided by IaaS platforms, these resources represent the smallest units of infrastructure that can be independently defined and provisioned. They can be defined as standalone entities or organized into logical building blocks, such as *L3 Constructs*[24] in AWS CDK, *L3 Constructs*[25] and *Modules*[26] in CDKTF, or *Component Resources*[27] in Pulumi.

To enable this functionality, PL-IaC tools rely on a combination of language and provider Software Development Kits (SDKs). Language SDKs allow developers to define infrastructure using language-specific constructs, while provider SDKs provide bindings for resource types of the underlying IaaS provider. The resources are created by instantiating classes corresponding to specific resource types and supplying configuration through constructor arguments. Once provisioned, these resources become immutable and expose output values, such as platform-specific identifiers, that can be passed as inputs to other resources. This creates a directed acyclic dependency graph, enabling the tools to orchestrate resource creation, updates, and deletions in the correct order. Dependencies can also exist in other layers of the project hierarchy, such as across stacks, as seen in mechanisms like the CDK tools' Cross-Stack References [28] [29] and Stack Dependencies [30], and Pulumi's Stack References [31].

Once the desired state is defined in code, it is applied to the target environment through a deployment process. Each tool follows the three-phase deployment workflow—*Assemble*, *Compile*, and *Execute*—with slight variations in implementation.

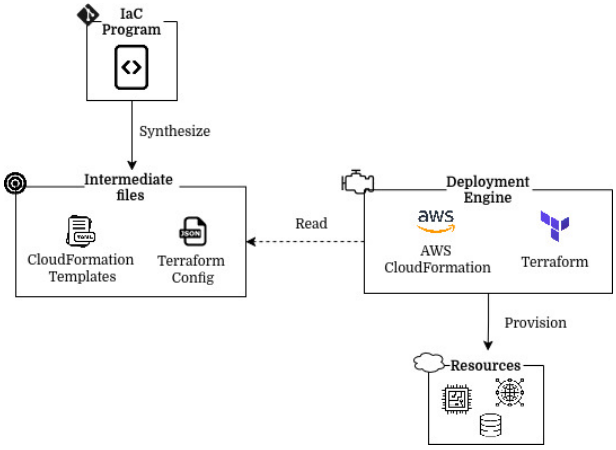


Fig. 3: CDK tools' deployment process.

As illustrated in Figure 3, AWS CDK and CDKTF adopt a *two-phase* or *synthesis-based* approach [13], [14]. During the *Compile* phase, the desired state is not constructed in memory but synthesized into intermediate files: CloudFormation templates[32] for AWS

[21] https://getampt.com/
[22] https://encore.dev/
[23] https://www.shuttle.dev/
[24] https://docs.aws.amazon.com/prescriptive-guidance/latest/aws-cdk-layers/layer-3.html
[25] https://developer.hashicorp.com/terraform/cdktf/concepts/constructs#construct-types
[26] https://developer.hashicorp.com/terraform/cdktf/concepts/modules
[27] https://www.pulumi.com/docs/iac/concepts/resources/components/
[28] https://docs.aws.amazon.com/cdk/v2/guide/resources.html#resource_stack
[29] https://developer.hashicorp.com/terraform/cdktf/concepts/stacks#cross-stack-references
[30] https://developer.hashicorp.com/terraform/cdktf/concepts/stacks#stack-dependencies
[31] https://www.pulumi.com/tutorials/building-with-pulumi/stack-references/
[32] https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/template-guide.html

CDK and JSON-based Terraform configurations[33] for CDKTF. These intermediate files are later processed in the *Execute* phase, where the actual deployment is carried out by the corresponding deployment engines: CloudFormation[34] or Terraform[35].

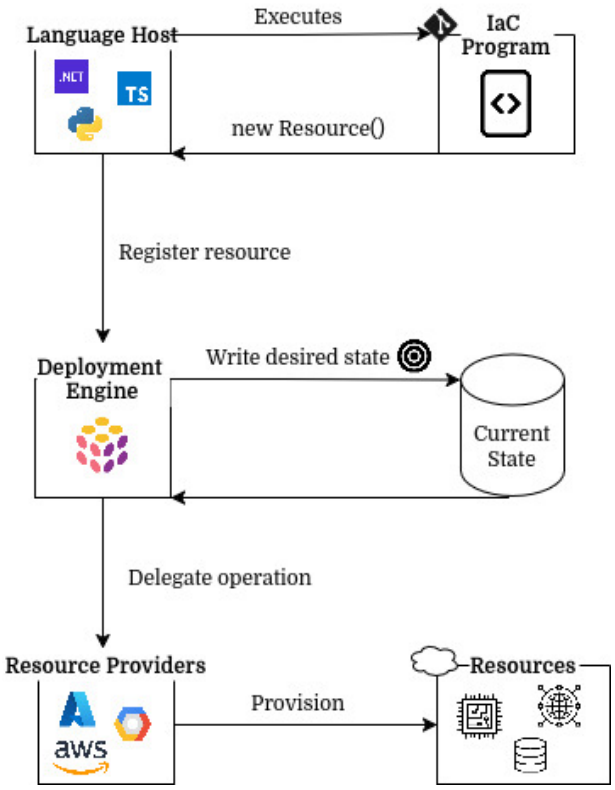


Fig. 4: Pulumi deployment process.

In contrast, Pulumi employs a *one-phase* or runtime-first approach, where the *Compile* and *Execute* phases are tightly integrated, as illustrated in Figure 4. When a deployment is invoked, the Pulumi CLI starts the language host[36] (consisting of the language runtime and SDK) which, in turn, executes the IaC program. As the program runs, each time a resource is instantiated (e.g., `new VirtualMachine(...)`), the language host registers it with the deployment engine. At this stage, the resource has not yet been provisioned, the language host merely declares it as part of the desired state and proceeds to the next resource, while the deployment engine concurrently begins processing the registration. The deployment engine compares each resource registration against the current state and delegates the necessary operations (creation, modification, or deletion) to the appropriate resource providers[37] (consisting of IaaS provider runtimes and SDKs). These providers handle all communication with the underlying IaaS APIs. Once a resource operation completes, the engine updates the stack's state file and continues processing subsequent resource requests. After the program has completed executing, the language host shuts down. The engine then examines the state file for any resources that were not re-registered during the run and instructs the resource providers to delete them, completing the deployment process. In essence, the language host registers resources during program execution, the deployment engine determines the operations required to reach the desired state by comparing with the current state, and the resource providers handle the actual interactions with the IaaS APIs. This process occurs entirely at runtime without generating intermediate files [15].

D. PL-IaC Unit Testing

Testing is perhaps one of the most valuable practices PL-IaC developers can adopt from software engineering. Since testing improves application code quality, it's reasonable to expect similar benefits for infrastructure. All PL-IaC tools support progressive testing, beginning with fast, narrowly scoped tests and progressing to broader, more comprehensive tests across integrated systems [4], [9].

An important principle of Agile development is integrating testing into the process from the start. The sooner code can be verified, the quicker issues can be addressed, and ultimately accelerate delivery. This is where unit testing comes in. A unit test is a small and fast automated test that verifies whether a specific piece of code works as expected. What constitutes a *unit* can differ depending on the programming paradigm or developer preferences. In object-oriented systems, a unit is often a class. In functional or procedural systems, it might be a single function. Sometimes developers might treat a group of closely related classes or a subset of methods as a single unit. The definition is flexible, what matters most is that it supports the ability to understand and test the system effectively. There are also different styles of unit testing. In *sociable* unit tests, real collaborators (such as dependencies or other classes) are allowed, as long as they're fast and reliable. In contrast, *solitary* tests isolate the unit under test by replacing collaborators with *test doubles*. A common type of test double is a *mock*[38], which acts as a configurable and lightweight stand-in for a collaborator. Mocks are particularly useful for testing interactions with external systems such as APIs or databases that are slow or complex to configure.

[33] https://developer.hashicorp.com/terraform/language/syntax/json
[34] https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/Welcome.html
[35] https://developer.hashicorp.com/terraform/cli/commands/apply
[36] https://pulumi-developer-docs.readthedocs.io/latest/docs/architecture/languages.html
[37] https://www.pulumi.com/docs/iac/concepts/resources/providers/
[38] https://martinfowler.com/articles/mocksArentStubs.html

Unit testing can be applied to infrastructure code, and since PL-IaC programs are built with GPLs, they naturally integrate with native testing frameworks. This allows developers to write unit tests in a way that closely resembles application testing. However, since infrastructure behaves differently from application code, these practices do not always map directly. Unlike application code, PL-IaC code doesn't run *in* the infrastructure, it runs to *generate* it. Furthermore, due to the differences with the CDK tools and Pulumi, there are further nuance to the testing process. Testing infrastructure effectively requires adapting familiar techniques to address the unique characteristics and constraints of PL-IaC.

The two-phase CDK tools translate imperative code into declarative configuration files, which are then passed to deployment engines such as CloudFormation or Terraform. In this model, there is no feedback channel from the deployment engine back to the CDK program. Consequently, unit tests are written against the synthesized output (e.g., JSON or YAML), rather than the imperative logic that generated it. This separation introduces a disconnect between the tests and the program logic, restricting verification to what is expressible in the target DSL rather than the full capabilities of the GPL [4]. The traditional test pyramid becomes less applicable in this context due to the limited expressiveness and testability of declarative configurations. As a result, test suites for declarative infrastructure often follow a diamond-shaped structure, with fewer unit tests and a heavier reliance on integration testing [9].

In contrast, the test pyramid aligns more naturally with Pulumi, owing to its one-phase architecture. Pulumi executes imperative code directly during deployment, allowing unit tests to evaluate the program's behavior at runtime. This enables more precise assertions, as the tests can interact with the actual execution flow of the code. Although the program is executed as written during testing, Pulumi does not provision real resources or communicate with external providers. Instead, it replaces these remote operations with user-defined, deterministic mocks that simulate expected behavior. This is achieved through a mocking mechanism, defined by the `IMocks` interface[39], which intercepts resource creation and function calls. Mocks are then supplied to Pulumi's built-in testing function, which executes the program in test mode and returns the resulting deployment state for validation through assertions, as illustrated in Figure 5.

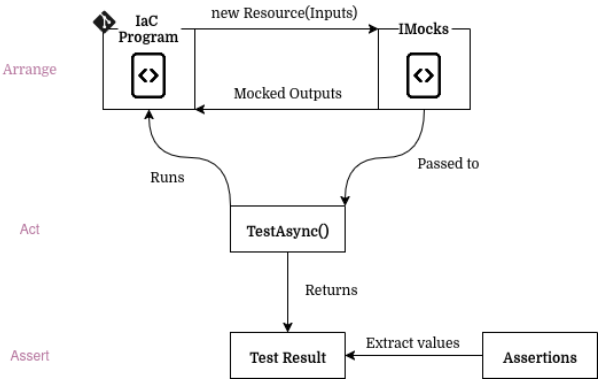


Fig. 5: Pulumi unit testing process.

IV. Related Work

We begin by providing an overview of testing in IaC, covering practices and tooling in DSL-based environments (Section IV-A). This is followed by research on PL-IaC, where testing adoption and tooling remain underexplored (Section IV-B).

A. IaC Testing

A systematic mapping study conducted by Rahman et al. [1], analyzed 31,498 publications in five databases related to IaC. The authors found that compared to other software engineering research areas, publications related to empirical studies and testing are infrequent, and further called for more empirical studies on test coverage, methodologies, and techniques within the IaC space. The issue was further highlighted by Guerriero et al. [2] who explored the practical adoption of IaC through interviews with senior cloud developers. Their qualitative analysis emphasized that among the many challenges mentioned, testability was viewed as the most critical issue.

A correlation between specific source code properties and defects in Puppet, was established by Rahman et al. [16]. The authors analyzed defect-related commits from 94 open-source repositories. Their findings revealed that certain coding practices can predict faulty scripts, with hard-coded 'magic' strings exhibiting the strongest association with defects. Based on these findings, Rahman et al. [17] conducted a replication study of the previous, with a focus on Ansible and Chef. They developed a static analysis tool, *the Security Linter for Ansible and Chef Scripts* (*SLAC*), to automatically detect security smells.

Hasan et al. [8] analyzed 50 artifacts online to identify testing practices used in Ansible and Chef. They derived six strategies to reduce IaC defects: (1) avoiding anti-patterns, (2) behavior-focused test coverage, (3) remote testing, (4) sandbox testing, (5) testing every change, and (6) automation. They further concluded that inadequate testing can introduce defects with potentially serious operational consequences. Building on these insights, Kumara et al. [18] conducted a similar study by analyzing 67 artifacts online, to identify best and bad

[39]https://www.pulumi.com/docs/iac/concepts/testing/unit/#add-mocks

practices in IaC, focusing on Ansible, Puppet, and Chef. They derived a taxonomy that covers implementation, design, and principle violations. They also emphasized testing, including static analysis, unit, integration, and functional testing, as a best practice for the reliability of the infrastructure. Expanding the scope, Drosos et al. [7], conducted an empirical study of 360 bugs from DSL-based IaC ecosystems, specifically Ansible, Puppet and Chef, which revealed common failure patterns and underscored the need for improved testing and defect prevention approaches.

While these studies explore testing adoption and practices in DSL-based IaC tools, they do not cover PL-IaC, a gap this thesis seeks to address.

B. PL-IaC Testing

Sokolowski et al. [6] were the first to address the gap in PL-IaC research by introducing PIPr, a dataset focused on PL-IaC tools and their usage among developers. The dataset contains metadata for 37,712 public IaC programs and includes source code for 15,504 of them, all collected from GitHub in August 2022. Most notably, PIPr provides insights into testing practices and adoption across the PL-IaC ecosystem. The study reveals a significantly low rate of testing adoption with only 25% of PL-IaC projects including any form of testing. AWS CDK had the highest adoption rate at around 38%, whereas Pulumi had the lowest at approximately 1%.

Building on the PIPr dataset, Sokolowski et al. [4] further investigated the low adoption of testing in PL-IaC, linking Pulumi's particularly low rates to the manual effort in maintaining accurate mocks. The authors proposed Automated Configuration Testing (ACT), a fuzzing- and property-based technique for detecting configuration bugs. They implemented this approach in ProTI, a Pulumi TypeScript tool that generates realistic mocks and validates inputs via pluggable generators and oracles.

Building on the work of Sokolowski et al. [4], [6], this thesis aims to deepen the understanding of testing challenges in PL-IaC and explore ways to increase testing adoption. To achieve this, we complement the PIPr findings by focusing on developer-reported challenges in public forums. Drawing on these insights, along with lessons from ProTI, we implement Pulumock, a tool aimed at addressing the identified challenges.

V. Research Methodology

This thesis adopts a case study methodology, guided by the definition from Runeson et al. [19]:

> *An empirical enquiry that draws on multiple sources of evidence to investigate one instance (or a small number of instances) of a contemporary software engineering phenomenon within its real-life context, especially when the boundary between phenomenon and context cannot be clearly specified.*

This approach was deemed appropriate as the study investigated a single contemporary phenomenon, namely challenges of unit testing in PL-IaC. These challenges are closely tied to the tools and workflows developers interact with, making it difficult to separate the phenomenon from its surrounding context. To examine this phenomenon within its real-world context, the study focused on a scoped case: Pulumi .NET, selected for its popularity and notably low testing adoption. Data was collected empirically from multiple sources and organized around the three research questions.

First, to address RQ1, we detail the data collection and analysis used to identify common challenges when unit testing Pulumi .NET programs (Section V-A). Then, to answer RQ2, we outline the design and implementation of Pulumock, a tool developed to mitigate the identified challenges (Section V-B). Finally, in response to RQ3, we describe the evaluation of Pulumock's effectiveness in mitigating these challenges, compared to Pulumi's default testing framework (Section V-C). All artifacts supporting this study, including source code and datasets, are publicly available on GitHub (Section V-D).

A. Dataset Construction

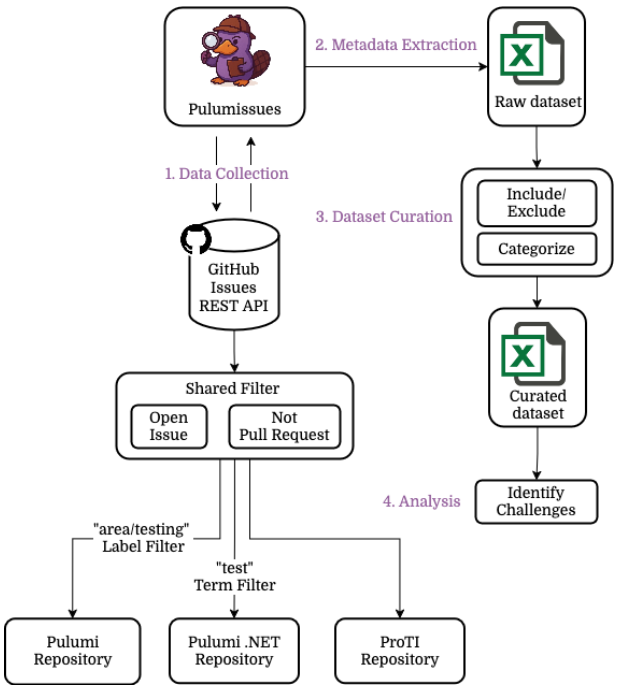


Fig. 6: The process of dataset construction to produce a curated dataset containing identified challenges.

To address RQ1, we applied a four-stage process, outlined in Figure 6: data collection (Section V-A1),

metadata extraction (Section V-A2), dataset curation (Section V-A3), and analysis to identify common challenges (Section V-A4).

1) Data Collection: To collect relevant data on unit testing challenges in Pulumi .NET, we developed a .NET-based console application named *Pulumissues*, designed to interact with the GitHub Issues REST API[40]. This repository mining approach was informed by the methodology used in constructing the PIPr dataset [6], but differed by focusing on issue reports rather than source code. GitHub was selected as the data source since it hosts Pulumi's core repositories and relevant testing tools. Among GitHub's various collaboration features, GitHub Issues was chosen as the primary data source. Issues capture actionable feedback such as bug reports, feature requests, and other potential improvements. While GitHub Pull Requests could have provided additional context, they typically represent work already in progress and are centered around proposed solutions rather than open problems. Similarly, GitHub Discussions support broader, informal conversations that are less focused on actionable development work. Compared to these alternatives, Issues offer a more direct view into the challenges developers face, making them the most relevant source for this study.

Using Pulumissues, we collected issue data through parallel API requests to three repositories: the main Pulumi repository[41], the Pulumi .NET repository[42], and the ProTI repository. These repositories were chosen based on their relevance to Pulumi .NET unit testing, either through direct contributions or by shaping the broader Pulumi testing ecosystem that indirectly impacts it. The tool was designed with a modular and extensible architecture, allowing additional repositories to be incorporated with minimal configuration should further relevant tools be identified in future work. To ensure the relevance and quality of the dataset, shared filters were applied across all repositories. Specifically, we excluded pull requests and limited the dataset to open issues. This focus on unresolved issues was intended to highlight ongoing challenges that are currently hindering testing practices. In addition to these shared filters, repository-specific filters were applied. For the Pulumi repository, issues labeled with *"area/testing"* were selected to isolate discussions directly related to testing. The Pulumi .NET repository lacked a consistent labeling scheme, so we instead retrieved issues containing the term *"test"* in either the title or body. For the ProTI repository, no additional filtering was applied due to the limited number of available issues. The API responses provided a rich set of metadata for each issue, forming the basis for the next stage of the process: extracting this information into a raw dataset suitable for further curation.

2) Metadata Extraction: After retrieving metadata from the GitHub API responses, relevant fields were extracted from each issue to construct a raw dataset for further analysis. The fields included the issue URL, title, body, number of reactions, number of upvotes, and creation date. Each of these fields contributed to different aspects of the evaluation process. The title and body provided the necessary context to understand the issue and assess its relevance. Reactions and upvotes reflected the level of community engagement and the perceived importance of the issue. The creation date identified long-standing issues that may indicate persistent and difficult challenges. The URL allowed direct access to the original issue for clarification or further exploration. The extracted data was compiled into an Excel file, with each repository placed in a separate sheet to support a structured review and manual annotation.

3) Dataset Curation: Following the construction of the raw dataset, a manual review was conducted to assess each issue for inclusion or exclusion based on predefined criteria. During this phase, preliminary categorization was also performed to justify each inclusion or exclusion decision. An *Include* column was added to the dataset, marking issues as `TRUE` if considered relevant and `FALSE` if excluded from further analysis, while a *Category* column was added to document the assigned category and rationale.

Issues were excluded according to the following criteria with a corresponding *Category* tag:

- **internal:** Issues that were internal to Pulumi and not intended for end-user interaction. They included issues related to internal testing infrastructure, coverage, and workflows.
- **automation-api:** Issues related to unit testing Pulumi *Automation API* programs, which are not directly focused on core Pulumi functionality.
- **not-unit-test:** Issues that addressed other forms of testing (e.g., integration testing) without direct relevance to unit testing, or that were unrelated to testing altogether.
- **not-dotnet:** Issues without mention of .NET, or any contextual relevance to the .NET ecosystem. However, issues that were not explicitly .NET-specific but still relevant to .NET users (e.g., cross-language unit testing) were considered relevant and included.

With irrelevant issues excluded, the remaining included issues were further investigated, during which their initial categorizations were refined and expanded. The final curation organized the included issues into the following categories:

[40]https://docs.github.com/en/rest/issues/issues?apiVersion=2022-11-28

[41]https://github.com/pulumi/pulumi

[42]https://github.com/pulumi/pulumi-dotnet

- **breaking:** Issues that identified missing functionality where workarounds were necessary to prevent unit tests from crashing under certain conditions.
- **complexity:** Issues highlighting challenges in writing or maintaining unit tests due to complex interfaces, tooling constraints, or verbose setups.
- **coverage:** Issues that limited test coverage, including the inability to mock or inspect certain data, or write tests for specific program styles.
- **bug:** Issues related to bugs within the Pulumi testing framework.
- **other:** Issues that did not fit clearly in the above categories but were still relevant to unit testing.

To maintain the integrity of the curated dataset, we retained both included and excluded issues. Those meeting the exclusion criteria were preserved to ensure traceability and support potential future reassessment. Both the raw and curated datasets are provided under the Artifacts Section V-D.

4) Analysis: After inclusion and categorization, a more detailed analysis was conducted on each issue to better understand the specific challenges in relation to Pulumi unit testing. A *Subcategories* column was added to the curated dataset to record both a finer-grained subcategory and a brief summary of each issue's core concern. Each issue was then evaluated based on its direct impact on hindering testability and the number of community reactions it received, serving as an indicator of its perceived importance. The evaluation was organized into tables detailing categories, subcategories, the number of issues, and corresponding reactions. This structure informed tool development by identifying the most pressing and widely acknowledged challenges.

B. Tool Construction

To address RQ2, we constructed Pulumock as a targeted response to the challenges identified through our analysis of the curated dataset.

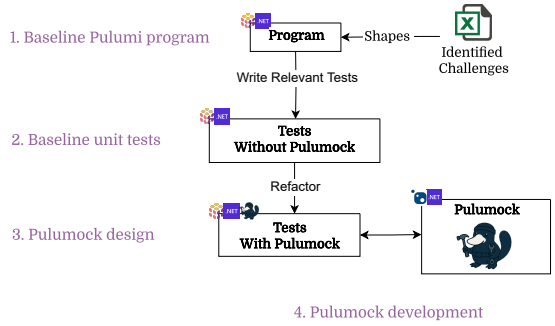


Fig. 7: The construction process of Pulumock, guided by identified challenges in the Curated dataset.

As illustrated in Figure 7, we began by developing a baseline Pulumi program (Section V-B1) and corresponding unit tests (Section V-B2) using only Pulumi's default testing framework, guided by the mined issues. Building on this foundation, we designed Pulumock (Section V-B3) and iteratively developed it (Section V-B4) by enhancing the baseline tests with new features and abstractions.

1) Baseline Pulumi Program: We began by developing a baseline Pulumi program that incorporated a wide range of behaviors and configurations observed in the issues collected during the data analysis phase. Our prioritization of issues followed the categories defined in the dataset curation process. We first addressed issues labeled **breaking**, as they represented critical blockers where unit tests could not function without additional tooling support. Next, we focused on issues categorized as **complexity**, motivated by its large amount of reactions and by prior research highlighting complexity as a major obstacle to unit testing adoption [4]. Finally, we considered issues related to **coverage**, treating them as opportunities for increased testing coverage rather than necessary preconditions for testing adoption. Issues categorized as **bug** were excluded, as they corresponded to defects within Pulumi's internal codebase that require fixes at the source rather than external tooling support. Finally, issues categorized as **other** were also excluded, as there was only a single issue, and it pertained to language-independent testing, which was beyond the scope of this work.

2) Baseline Unit Tests: With the baseline Pulumi program established, we created a suite of unit tests designed to exercise the scenarios described in the included issues. These tests were initially written using only Pulumi's default testing framework, without Pulumock. This approach allowed us to directly experience the challenges faced by developers, and provided a concrete baseline for identifying the improvements Pulumock should offer. Once the baseline test suite was complete, we duplicated the tests with the intention of adapting them to use Pulumock. This parallel structure enabled us to evaluate and iteratively refine Pulumock's design during its development.

3) Pulumock Design: Building on insights gained from the baseline test suite, we then designed Pulumock with the goal of enhancing, rather than replacing, Pulumi's existing unit testing capabilities. By extending the current mechanisms instead of introducing an entirely separate framework, Pulumock remains compatible with the existing Pulumi testing model. This approach not only supports flexibility for developers, allowing them to combine Pulumock with Pulumi's default testing framework, but also positions Pulumock as a proof of concept for potential future integration into Pulumi's

core tooling. To support easy adoption, we made an early decision to distribute Pulumock as a NuGet package, aligning with standard package management practices in the .NET ecosystem and enabling seamless integration into existing test projects. To further minimize the learning curve, a decision was made to adopt a fluent interface[43] backed by the builder pattern[44], following conventions established by popular testing libraries such as FluentAssertions[45], AutoFixture[46], and NSubstitute[47]. This design approach supports Pulumock's overarching goal: integrating naturally into existing developer workflows while extending familiar Pulumi testing mechanisms through established conventions. As illustrated in Figure 8, its architecture resembles Pulumi's default testing framework (Figure 5) but mainly differs in its usage of a fluent interface.

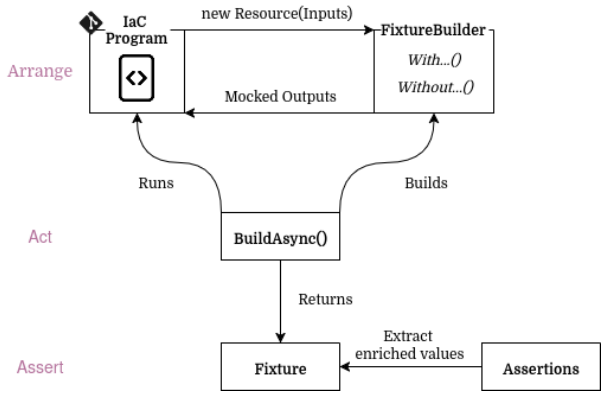


Fig. 8: Pulumock unit testing components and interactions.

4) Pulumock Development: Following the initial design, Pulumock was iteratively developed by rewriting the baseline tests to adopt the fluent interface and other abstractions. Real-time feedback during this migration process guided continuous refinements of Pulumock's features, API surface, and overall usability. As part of the development process, documentation was added to support adoption and usability. This included sections on architecture, getting started, a cheat sheet, and usage examples.

C. Tool Evaluation

To address RQ3, we conducted a within-subjects study to evaluate Pulumock's ability to mitigate common unit testing challenges in Pulumi .NET, in comparison to Pulumi's default testing framework. The evaluation focused on the most common challenges identified in RQ1, which involved complexity and time-consuming workarounds for unclear or incomplete APIs. Adoptability and maintainability were chosen as the main assessment criteria because they directly reflect these issues: adoptability captures how easily developers can navigate complexity and workarounds during initial adoption, while maintainability reflects the ability to manage growing complexity as tests scale.

We begin by explaining the criteria and recruitment of participants (Section V-C1). We then describe the study design and procedure (Section V-C2). Finally, we detail the mixed-methods approach used to assess adoptability and maintainability, incorporating both task completion times (Section V-C3) and questionnaires (Section V-C4).

1) Participant Selection & Recruitment: Participants were selected based on their background in software engineering, with at least a basic understanding of software development practices. Preference was given to those with experience in Pulumi or similar IaC tools. This criteria was defined so that participants could complete the tasks independently and provide feedback informed by relevant experience.

To recruit participants, we shared posts on public developer forums focused on Pulumi .NET, such as GitHub Discussions[48]. We also reached out to professionals in the Software Engineering field with relevant experience. In addition, we contacted students enrolled in Software Engineering programs to diversify our participant pool and include perspectives from individuals with less professional experience. This aimed to capture a broad spectrum of experience levels, enabling us to explore Pulumock was perceived by both seasoned developers and those newer to the field.

2) Study Design & Procedure: We applied a within-subjects repeated measures design, where each participant completed the same test cases with and without Pulumock, serving as their own control. For clarity, we refer to these as the *W* condition (with Pulumock) and the *W/o* condition (without Pulumock, using Pulumi's default testing framework).

The study began with participants receiving a brief presentation on Pulumi, unit testing within Pulumi, and the Pulumock tool. This overview aimed to equip them with the necessary context and understanding for the tasks ahead.

Participants were then given access to the *Puluvaluation* repository[49], which included a sample Pulumi program and a corresponding test project containing skeleton test cases. These test cases were based on issues labeled as *breaking* and *complexity* in the curated dataset, aligning with the most pressing challenges identified. To

[43]https://martinfowler.com/bliki/FluentInterface.html
[44]https://refactoring.guru/design-patterns/builder
[45]https://fluentassertions.com/
[46]https://github.com/AutoFixture/AutoFixture
[47]https://nsubstitute.github.io/

[48]https://github.com/pulumi/pulumi/discussions/19416
[49]https://github.com/Pulumock/Puluvaluation

support the tasks, participants were also provided with condition-specific documentation. For the *W* condition, the Pulumock GitHub Wiki[50] was supplied, containing usage instructions and examples. For the *W/o* condition, participants received Pulumi's official unit testing documentation[51], relevant examples, and GitHub issues.

With the necessary background and resources provided, participants were instructed to clone the repository and implement the test cases, once in the *W* condition and once in the *W/o* condition. Participants were evenly split to start with either condition to counterbalance learning effects and reduce order bias. Both conditions were carried out on the participants' own machines, along with unrestricted access to tools and internet resources, to simulate a realistic development environment. They completed the tasks independently, without interaction or influence from other participants.

3) Task Completion Times: As part of assessing adoptability, task completion times were recorded for each participant under both conditions and stored in structured CSV files for analysis.

Since each participant provided paired observations across both conditions, a *paired t-test* [20] was written in R[52] to determine whether the *W* condition led to a reduction in task completion time. While our sample size was too small to claim statistical significance, we included this analysis to demonstrate how such evaluations could be performed in future studies with larger samples. The hypotheses tested were as follows:

- $\mathbf{H_0}$: The *W* condition does not reduce task completion time.
- $\mathbf{H_1}$: The *W* condition reduces task completion time.

Prior to conducting the paired t-test, the standard assumptions [21], [22] were verified:

- **Normality of difference scores:** For a paired t-test, this refers to the normal distribution of the *differences* between paired scores, not the scores themselves [20]. A Shapiro-Wilk test was used to evaluate this, and a Q–Q plot was generated for visual inspection.
- **Continuous dependent variable:** The dependent variable must be measured on a continuous scale, typically interval or ratio.
- **Independent scores across participants:** One participant's performance should not affect another's.

4) Questionnaires: After finishing the implementation tasks, participants were asked to fill out two questionnaires. Their anonymity was preserved throughout the study and was intended to encourage unbiased responses without concerns about judgment.

The first was a background questionnaire, as shown in Appendix 1: Experience Questionnaire. This included their current role, programming experience, and familiarity with technologies relevant to the study. This information was used to characterize the participant pool and provide a context for interpreting the results.

Participants were then asked to complete an assessment questionnaire, as shown in Appendix 2: Assessment Questionnaire. This included multiple-choice questions to collect structured feedback, as well as open-ended questions to capture more detailed insights and personal reflections on both conditions. The questions focused on adoptability and maintainability to evaluate whether and how Pulumock mitigated the challenges embedded in each test case. Specifically, they aimed to assess learning curve, which complements the recorded task completion times, as well as the perception of how maintainable the tests would be at scale.

D. Artifacts

All artifacts produced in this study are publicly available to promote transparency, reproducibility, and future work.

The data collection source code and datasets are hosted in the Pulumissues repository[53].

The source code for the Pulumock tool can be found in the Pulumock repository[54]. Documentation is provided in the accompanying GitHub Wiki[55].

The evaluation materials are available in the Puluvaluation repository[56]. This includes source code for tests and statistics, task completion time results, and questionnaire responses.

VI. Results

This section presents the results corresponding to each research question. First, we show the findings from the Pulumissues dataset to answer RQ1 (Section VI-A). Next, we describe how Pulumock was constructed to address RQ2 (Section VI-B), followed by an evaluation of its impact to answer RQ3 (Section VI-C).

A. RQ1: What are the common challenges developers face when unit testing Pulumi .NET programs?

To understand the challenges developers face when unit testing Pulumi .NET programs, we present the results derived from the Pulumissues dataset. The initial raw dataset included 113 open issues from the Pulumi repository, 34 from the Pulumi .NET repository, and 2 from the ProTI repository, totaling 149 open issues. Following the inclusion and exclusion criteria, the dataset

[50] https://github.com/Pulumock/Pulumock/wiki
[51] https://www.pulumi.com/docs/guides/testing/unit/
[52] https://www.r-project.org/
[53] https://github.com/Pulumock/Pulumissues
[54] https://github.com/Pulumock/Pulumock
[55] https://github.com/Pulumock/Pulumock/wiki
[56] https://github.com/Pulumock/Puluvaluation

was substantially narrowed down. The Pulumi repository was reduced to 12 issues, the Pulumi .NET repository to 7 issues, and the ProTI repository to 1 issue. As a result, the final curated dataset consisted of 20 open issues out of the original 149. Figure 9 summarizes the number of included issues by category with corresponding number of reactions received. Some issues were classified into multiple categories due to containing several distinct concerns, which result in overlapping category totals.

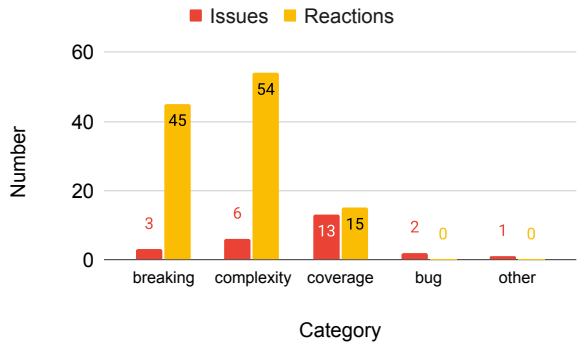


Fig. 9: Number of issues and reactions by Category.

In the following subsections, we examine each category in detail. A summary of the overall findings is provided in Section VI-A5.

1) Breaking Issues: 3 issues were labeled as *breaking*, collectively receiving 45 upvotes. They fall into two subcategories, *Stack Configuration* and *Stack References*, with some issues overlapping as shown in Figure 10.

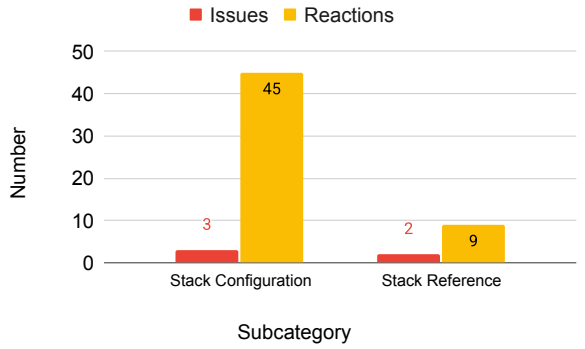


Fig. 10: Number of issues and reactions by subcategories of the Breaking category.

Stack Configuration and *Stack References* are commonly used Pulumi features that provide access to values defined in configuration files and external stacks, respectively. If a Pulumi program under test uses either of these components without proper mocking, the test will crash. This is why issues related to them are categorized as *breaking*. Although mocking these components is technically possible, it is described as difficult due to the lack of official APIs and documentation. These issues remain unresolved, with the most upvoted report filed in 2020 and two more added in 2024, highlighting persistent challenges that still require attention.

2) Complexity Issues: A total of 6 issues were labeled as *complexity*, making it the most popular category with 54 reactions. These issues were further grouped into subcategories, as shown in Figure 11.

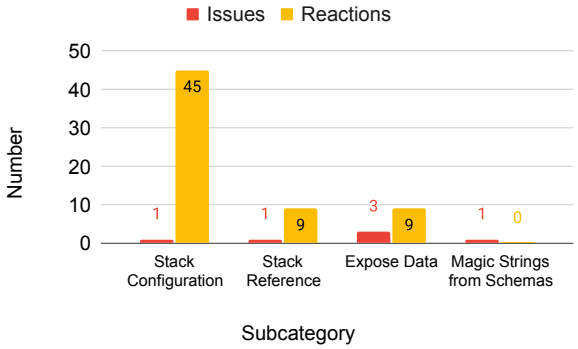


Fig. 11: Number of issues and reactions by subcategories of the Complexity category.

The *Stack Configuration* and *Stack References* subcategories revisit issues discussed earlier, but with a focus on the complexity of mocking these components and user requests for better APIs and a smoother testing experience.

These concerns connect closely with the *Expose Data* subcategory, which highlights the complexity of accessing data during testing. The most notable example involves stack outputs, which is a feature commonly used to export values from a given stack. Users report limited built-in support and the need for workarounds to extract and assert against this data.

The *Magic Strings from Schemas* subcategory further reflects the complexities of testing in Pulumi, as users report difficulties mocking resources and function calls. They describe having to manually construct serialized inputs and outputs to match internal Pulumi package schemas [57], which are often unclear or undocumented.

3) Coverage Issues: A total of 13 issues were classified under the *coverage* category, making it the most frequent category by count. However, it ranked third in community engagement, with only 15 reactions. These issues were further grouped into subcategories, as shown in Figure 12.

[57]https://www.pulumi.com/docs/iac/using-pulumi/extending-pulumi/schema/

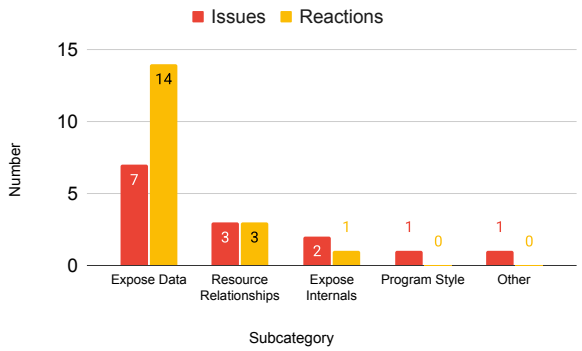


Fig. 12: Number of issues and reactions by subcategories of the Coverage category.

The *Expose Data* subcategory captures requests for broader access to data during testing, both to enable mocking and to support assertions. Specific concerns include limited visibility into stack outputs, component resource properties, and resource metadata. For example, users report difficulties asserting on resource options, which define behaviors such as protection from deletion, aliases, and custom transformations. This issue also overlaps with the *Resource Relationships* subcategory, which highlights challenges in asserting on parent-child relationships and dependency ordering, details that are also contained within resource options.

The *Expose Internals* subcategory involves issues related to asserting on Pulumi-specific exceptions and making `Pulumi.Log` safe to invoke outside the standard Pulumi runtime. The *Program Style* subcategory relates to enabling unit testing for programs that use top-level statements, as an alternative to the traditional `Stack` class. Finally, the *Other* subcategory includes the inability to mock engine-based transforms.

4) Bug & Other Issues: The 3 remaining issues were labeled as *bug* or *other* and received no reactions. The *bug* category covers problems such as unit tests hanging when accessing an undefined service metadata name, and incorrect inclusion of an implicit parent in resource URNs, which leads to mismatches and failed assertions. The *other* category includes a suggestion for recording and replaying Pulumi's RPC calls to support language-independent testing.

5) Summary: To answer RQ1, our results show that the most common challenges in unit testing Pulumi .NET programs involve the complexity and lack of coverage of mocking and accessing test data. These challenges often arise from unclear or incomplete testing APIs. While some issues can be addressed with cumbersome workarounds, others currently have no viable solution. In some cases, these limitations lead to breaking issues that prevent tests from running entirely.

Listing 1: Complete Pulumock example

```
1  var fixtureBuilder = new FixtureBuilder()
2      .WithMockResource(new MockResourceBuilder
           <ResourceGroup>()
3          .WithOutput(x => x.Location, "eu")
4          .Build());
5
6  var fixture = await fixtureBuilder
7      .BuildAsync(async () => await Create());
8
9  var resourceGroup = fixture
10     .Resources.Require<ResourceGroup>();
11
12 var location = await resourceGroup
13     .Location.GetValueAsync();
14
15 location.ShouldBe("eu");
```

B. RQ2: In what ways can Pulumock mitigate common challenges in unit testing Pulumi .NET programs?

To address the challenges identified in RQ1 (Section VI-A5), we successfully developed Pulumock, a testing tool for Pulumi .NET programs. It is publicly available as a NuGet package and can be integrated into existing .NET testing projects.

To provide a clearer understanding of how Pulumock operates and targets these challenges, we begin this section with an overview of its architecture (Section VI-B1). We then examine the design more closely through the lens of the common Arrange-Act-Assert (AAA) pattern[58] demonstrating how Pulumock tackles each phase: the *Arrange* phase through fluent mocking (Section VI-B2), the *Act* phase by executing the Pulumi program in test mode (Section VI-B3), and the *Assert* phase with enriched test data and custom extension methods (Section VI-B4). Together, the architectural overview and the AAA-based breakdown provide the necessary context to understand how Pulumock's design directly responds to the identified challenges discussed earlier. A focused summary of these design decisions and their alignment with the identified issues is presented in Section VI-B5.

1) Overview of the Pulumock Architecture: Pulumock's architecture centers around two core components: *FixtureBuilder* and *Fixture*, as demonstrated in Listing 1. The *FixtureBuilder*, shown in lines 1–4, serves as the main entry point for setting up tests. It uses the builder pattern to chain together mocks for different components. Once setup is complete, calling `BuildAsync()` on lines 6–7 runs the Pulumi program in test mode, and returns a *Fixture* object. This object contains enriched data that can be used to extract values and perform assertions, as demonstrated in lines 9–15.

[58] https://learn.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices#arrange-your-tests

Listing 2: Mocking example in Pulumi's default testing framework

```
1  public class Mocks : IMocks
2  {
3    public Task<(string? id, object state)>
         NewResourceAsync(MockResourceArgs args
         )
4    {
5      var outputs = ImmutableDictionary.
           CreateBuilder<string, object>();
6
7      outputs.AddRange(args.Inputs);
8
9      if (string.Equals("azure-native:resources
           :ResourceGroup", args.Type,
           StringComparison.Ordinal))
10     {
11       outputs.Add("azureApiVersion", "
             2021-04-01");
12     }
13
14     args.Id ??= $"{args.Name}_id";
15     return Task.FromResult<(string? id,
           object state)>((args.Id, (object)
           outputs));
16   }
17 }
```

Listing 3: Mocking example in Pulumock

```
1  new FixtureBuilder()
2    .WithMockResource(new MockResourceBuilder<
         ResourceGroup>()
3      .WithOutput(x => x.AzureApiVersion, "
           2021-04-01")
4      .Build())
```

This high-level overview is explored in more detail in the following sections.

2) Arrange (Mocking): Listing 2 depicts the mocking approach used by Pulumi's default testing framework, as described in the official Pulumi unit testing documentation[59]. This method involves implementing the `IMocks` interface and defining resource mocks using raw Pulumi schema strings[60]. Pulumock abstracts away direct interaction with `IMocks` by exposing a fluent API through the `FixtureBuilder` class, as demonstrated in Listing 3. It follows the builder pattern and supports method chaining using `With...()` and `Without...()` methods to add, override, or remove mocks. These methods accept dedicated builder classes for each mock type such as stack configuration, stack references, resources, and function calls. Support for *stack configuration* and *stack references* was introduced here to mitigate all issues in the *breaking* category and two issues in the *complexity* category, where the absence of APIs and reliance on complex workarounds were common concerns. Furthermore, the builders allow defining mocks with .NET types and lambda expressions, as shown in Listing 3 on lines 2 and 3. This design choice responded to the issue in the *Magic Strings from Schemas* subcategory, reducing the complexity of manually crafting serialized inputs and outputs by replacing it with strong typing.

Internally, Pulumock connects the specified .NET types and lambda expressions to Pulumi's string-based schemas through a custom `IMocks` implementation. It relies on reflection[61] to read metadata from .NET types at runtime. Specifically, it accesses the `Type` property in the `ResourceTypeAttribute`, which contains the Pulumi schema identifier. This identifier is used to determine which resource should be mocked. Pulumock then uses lambda expressions such as `x => x.Location`, to specify which output values to mock on the identified resource. It parses the expression to determine the target output property and then reads the `Name` from the `OutputAttribute` to resolve the corresponding schema output name. In cases where reflection cannot be applied, such as when a provider does not follow expected conventions, Pulumock provides a fallback mechanism using raw strings while preserving the same fluent interface.

3) Act (Test Execution): Once mocks are configured during the *Arrange* phase, calling `Build()` or `BuildAsync()` on the `FixtureBuilder` runs the Pulumi program in test mode. This produces a `Fixture` object that contains enriched data for inspection and assertions, as demonstrated in Listing 1, lines 6–7.

Under the hood, Pulumock uses Pulumi's `Deployment.TestAsync()` method, inheriting the same capabilities as Pulumi's default testing framework. It supports the top-level statement style, as requested in a reported issue under the *Program Style* subcategory. Additionally, it supports both full stack testing and more targeted tests of individual units, such as `ComponentResource` types. However, support for additional styles, such as the `Stack` class and service provider-based tests, remains to be implemented.

4) Assert (Accessing Data): Following the *Act* phase, the *Assert* phase focuses on validating the behavior of the system under test. The returned `Fixture` instance consolidates all relevant test data into a single interface, including:

- **Resources**: Standard resource metadata, limited to output values.
- **Enriched Resources**: Extended resource metadata such as raw input values.

[59] https://www.pulumi.com/docs/iac/concepts/testing/unit/#add-mocks

[60] https://www.pulumi.com/docs/iac/using-pulumi/extending-pulumi/schema/

[61] https://learn.microsoft.com/en-us/dotnet/fundamentals/reflection/reflection

- **Enriched Calls**: Captured details of function calls, including inputs, outputs, and invocation count.
- **Registered Outputs**: Output values explicitly declared by the component under test.

The typical workflow in this phase involves accessing a resource, function call, or other component from the `Fixture`, then extracting its input or output values for assertions. Pulumock provides extension methods that enable querying the `Fixture` and its components using .NET types and lambda expressions, as illustrated in Listing 1, lines 9–15.

Resource instances such as `CustomResource`, `ComponentResource`, and `StackReference` can be accessed through `Fixture.Resources`, which exposes the standard output data provided by Pulumi. For deeper inspection, `Fixture.EnrichedResources` offers additional metadata, including the original input values passed to each resource. This was implemented to expose richer data, a common theme in the *Expose Data* subcategories, but it was not linked to any specific issues. Additionally, support for asserting on parent-child relationships between resources was implemented using reflection. This addressed two out of three concerns raised under the *Resource Relationships* subcategory. However, support for more advanced scenarios such as dependency ordering remains incomplete.

Function calls made during the test are accessible through `Fixture.EnrichedCalls`. Each call is captured as an individual `EnrichedCall` instance, recording its type, input arguments, and mocked outputs. Multiple invocations of the same function result in multiple `EnrichedCall` entries, allowing fine-grained assertions on behavior such as call count and argument variation. Similar to `Fixture.EnrichedResources`, this was implemented to expose richer data, but it was not linked to any specific issues

Lastly, `Fixture.RegisteredOutputs` stores the output values returned from the component under test. This includes stack outputs, which addresses all *Expose Data*-related issues under the *Complexity* category and three under *Coverage*. These outputs are stored as key-value pairs where values may be resource instances or data wrapped in `Output<T>`. Pulumock provides extension methods to access and make assertions against these outputs.

5) Summary: To answer RQ2, Pulumock targeted several developer-reported challenges spanning multiple issue categories, as illustrated in Figure 13. The design specifically addressed all reported issues in the two prioritized categories, *breaking* and *complexity*, and 6 out of 13 of issues in the less prioritized *coverage* category.

The *coverage* category is presented in more detail in Figure 14, as not all related issues were fully addressed.

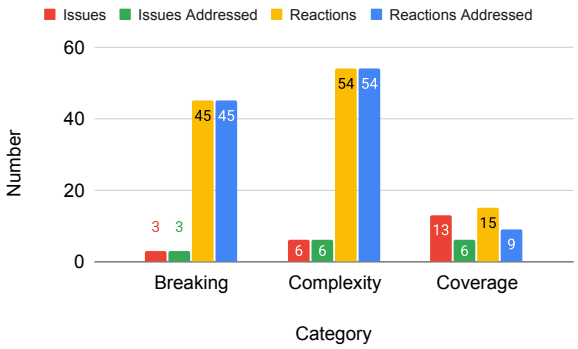


Fig. 13: Number of reported and addressed issues with corresponding reactions by category.

Note that the figure shows one resolved issue under the *Expose Internals* subcategory. This refers to the issue of asserting on Pulumi-specific exceptions, which was found to already be supported by Pulumi's default testing framework and is therefore included as resolved.

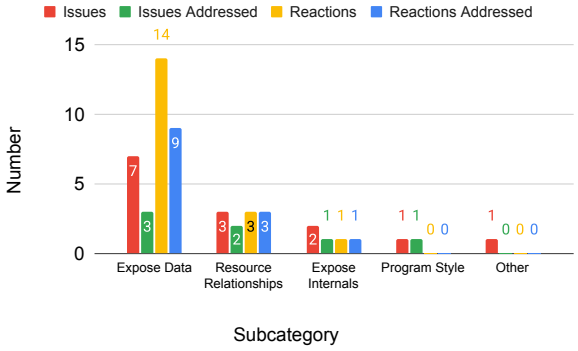


Fig. 14: Number of reported and addressed issues with corresponding reactions by subcategories of the Coverage category.

No changes were made in response to issues under the *Bug* and *Other* categories, consistent with the rationale discussed in the methodology section V-B1.

C. RQ3: How does Pulumock compare relative to Pulumi's default testing framework in terms of mitigating common challenges when unit testing Pulumi .NET programs?

To compare Pulumock with the Pulumi's default testing framework, we present results from a within-subjects study in which four participants completed identical tasks in both conditions. We begin by introducing the participants (Section VI-C1). We then present differences in task completion times for each participant (Section VI-C2). Next, we present questionnaire responses capturing participant experiences, including individual

preferences and motivations (Section VI-C3). Finally, we summarize the findings in relation to RQ3 (Section VI-C4).

1) Participants: The participants (P) are presented in Appendix 3: Participants (P) with experience across relevant areas with their current role and experience in relevant areas. P1 was a professional DevOps engineer with extensive programming background, while the others were less experienced students. None had prior experience with Pulumi, though all had at least some familiarity with other IaC tools.

2) Task Completion Times: Detailed task completion times for each participant are shown in Appendix 4: Participant (P) task completion times with (W/) and without (W/o) Pulumock. Two participants began in the *W* condition, and two in the *W/o* condition.

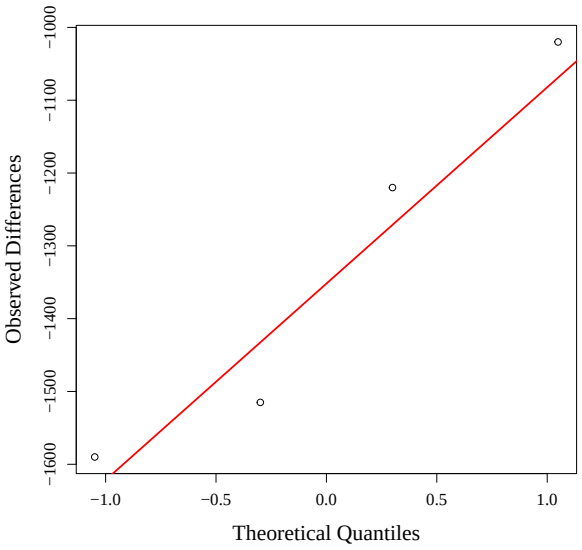


Fig. 15: Q–Q Plot for assessing normality of task completion time differences

The Shapiro-Wilk test indicated that the difference task completion times were approximately normally distributed ($W = 0.926$, $p = 0.570$), as the p-value exceeds the $\alpha = 0.05$ threshold. The accompanying Q–Q plot illustrated in Figure 15 supports this, with points closely following the reference line. Task completion time was measured on a ratio scale (seconds), meeting the requirement for continuous dependent variable. While repeated measurements were taken from the same individuals, each participant's task execution were independent of those from other participants, satisfying the assumption of independence across participants. Thus, all assumptions were met, and a paired t-test was deemed appropriate.

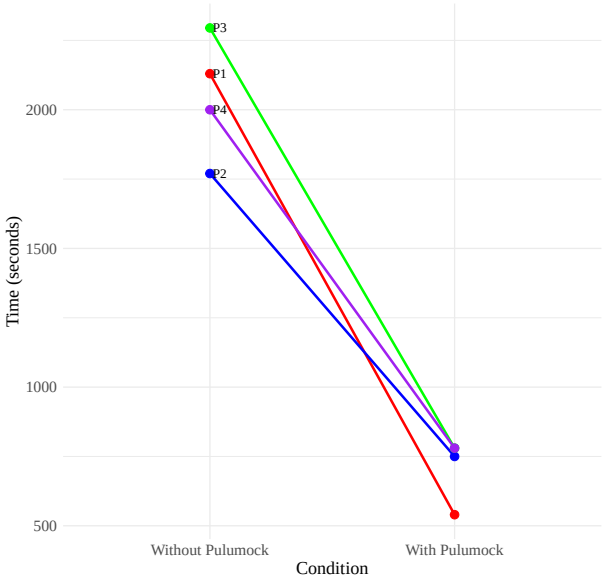


Fig. 16: Paired comparison of task completion times in the *W/o* and *W* conditions (without and with Pulumock).

The paired t-test result showed that task completion times were lower in the *W* condition (M = 712.5 s, SD = 115.87) compared to the *W/o* condition (M = 2048.75 s, SD = 221.6), as illustrated in Figure 16. The observed difference yielded a low p-value (p = 0.0021), below the commonly used significance threshold (α = 0.05), which led to the rejection of the null hypothesis. However, as noted in Section V-C3, the small sample size limits the reliability of this result, and we do not claim broader statistical significance.

In total, tasks took 47 minutes and 30 seconds with Pulumock, compared to 2 hours, 16 minutes, and 35 seconds using Pulumi's default testing framework. On average, Pulumock reduced development time by approximately 65% per participant.

3) Questionnaires: In addition to reducing development time, participant feedback showed that the *W* condition improved the overall testing experience. As illustrated in Figure 17, all participants responded positively to Pulumock, expressing a clear preference for it across all questions.

Participants found the *W* condition easy to learn, where P1, P3, and P4 reported that they required little to no guidance. In contrast, P1 and P3 expressed the need for supervisor support to be able to complete the tasks in the *W/o* condition. Both P2 and P3 appreciated the documentation in the *W* condition, describing it as helpful for getting started.

In terms of readability, P3 and P4 noted that the *W* condition made the code less verbose and easier to

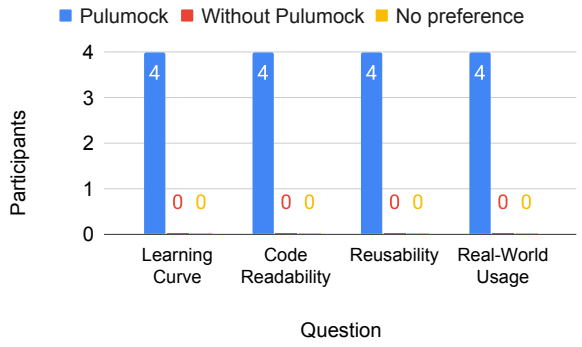


Fig. 17: Distribution of participant tool preferences by multiple-choice questions from the assessment questionnaire.

understand by reducing boilerplate and technical jargon. P1 and P2 found it easier to get an overview of the tests as related code was defined closer together, with P1 attributing this to the use of the builder pattern. However, P2 also expressed confusion regarding the usage of nested builders in this condition. P2, P3, and P4 summarized the *W* condition as simpler overall, with fewer steps required to complete the tasks.

When reflecting on reusability, P1 again pointed to the builder pattern used in the *W* condition as an enabling factor and P4 highlighted the benefit of writing fewer lines of code. However, P2 and P3 felt that it was difficult to fully assess reusability based on the limited test cases, but they still saw more potential in the *W* condition to offer better reuse.

Regarding real-world applicability, P2, P3, and P4 believed that the *W* condition would save time and effort in practice. While P2 acknowledged these benefits, they also expressed concerns about Pulumock's level of maturity:

> *...not having a community around it can suck if you run into a problem, and LLMs won't be able to help you.*

Taking a different view, P1 preferred testing through deployment rather than writing unit tests for Pulumi:

> *Wouldn't write a Pulumi normal test ever, would test by deploying to an environment instead.*

In response to the open-ended questions about challenges, overall experience, and suggestions for Pulumock, participants shared several insights. P1 and P4 found the *W/o* condition difficult and nearly unworkable, with the *W* condition making the tasks much more manageable. P2 and P3 faced general C# syntax challenges due to limited experience and P3 further expressed confusion with nested builders in the *W* condition. However, P3 found a stark contrast between the two approaches, strongly favoring the *W* condition.

Participants were also asked whether they had any suggestions for improving Pulumock. Most of the participants did not express a clear need for changes. The only concrete suggestion was related to confusion about the nested builder methods, a recurring theme across the questionnaire. However, P3 acknowledged:

> *...but that might be due to inexperience*

4) Summary: To answer RQ3, the results show that participants perceived Pulumock as more effective than Pulumi's default testing framework in mitigating common challenges embedded within the tasks. Faster task completion times together with positive questionnaire responses indicate that Pulumock contributed to a more adoptable and maintainable development experience.

VII. Discussion

This section discusses the findings of this study in relation to the thesis objectives: to understand the challenges developers face when unit testing PL-IaC, and to explore how improved tooling can mitigate these challenges and increase testing adoption. We begin discussing the challenges developers encounter in Pulumi unit testing (Section VII-A). We then evaluate the role of tooling in mitigating these challenges (Section VII-B). The discussion then turns to the broader implications of the findings, particularly how they may generalize to the PL-IaC ecosystem (Section VII-C). Following this, the study's limitations and potential threats to validity are considered (Section VII-D). Finally, societal and ethical considerations are explored (Section VII-E).

A. Understanding the Challenges of Unit Testing in PL-IaC

1) Identifying Persistent and Newly Observed Barriers: The curated dataset and the within-subjects study revealed a clear pattern: developers face major barriers when testing Pulumi .NET programs, particularly related to the complexity and lack of coverage when mocking and accessing test data for assertions. These findings partially align with prior work by Sokolowski et al. [4], who argue that while mocking resource definitions in Pulumi is relatively straightforward, the complexity lies in the manual effort required to maintain these mocks accurately over time. In contrast, our analysis reveals that the challenge lies not just in maintaining mocks, but in creating them in the first place. In practice, the available interfaces are often difficult to use and lack essential feature support. Furthermore, our findings extend prior work by showing that developers not only face difficulties with mocking but also in accessing necessary data for making assertions over desired scenarios.

2) A Self-Reinforcing Cycle of Testing Barriers and Limited Adoption: The curated dataset extends PIPr [6] by capturing real-world testing challenges reported by developers in issue trackers. While our inclusion criteria ensured relevance to Pulumi .NET unit testing, the final dataset was limited to 20 issues. The limited number of relevant issues could be interpreted as a sign of general developer satisfaction with Pulumi's testing support. However, a more likely explanation is that developers are not engaging with unit testing in the first place. This interpretation is supported by findings in the PIPr dataset, which showed remarkably low adoption of unit testing in Pulumi projects.

Findings from the within-subjects study further support this view of low adoption and offer additional insight into its underlying causes. All participants encountered significant difficulty completing the tasks independently in the *W/o* condition, with some describing the process as nearly unworkable. These findings closely align with those of Guerriero et al. [2], who identified "impossible testing" as the most frequently mentioned challenge pointed out by interviewees, highlighting the lack of established testing frameworks and practices. The long-standing perception that infrastructure testing is difficult likely contributes to the low adoption rates seen in the PIPr dataset and may also explain why the most experienced participant, P1, viewed unit tests as unsuitable for Pulumi.

These observations reveal a deeper problem. The existence of a small number of long-standing, unresolved issues in the Pulumi community suggests that testing was not a core concern in the tool's original design and continues to receive limited attention in its ongoing development. Consequently, developers encounter significant friction when attempting to write tests and are often discouraged by the excessive time and effort required to implement them. Over time, this has led to a perception across the community that unit tests are not worth adopting. We argue this creates a situation where Pulumi doesn't improve testing due to low demand, and demand remains low because testing support is lacking. This self-reinforcing cycle continues to suppress the widespread adoption of testing practices in Pulumi projects.

3) Suggestions for Future Work: Future work could benefit from conducting interviews or surveys with experienced PL-IaC developers to offer deeper insights into testing challenges that may not be visible through public issue trackers. Examining related issues across other PL-IaC tools could further reveal both common and tool-specific challenges, offering opportunities for tools to learn from and build upon each other's approaches.

B. The Role of Tooling in Mitigating Unit Testing Challenges in PL-IaC

1) The Importance of Tool Design to Address Complexity: In designing Pulumock, we saw that many of the reported challenges stemmed from two factors: missing features in the testing API and a complex design that diverges from established .NET development conventions. Although missing features can be resolved by adding functionality, the complexity of the design is a broader issue that warrants further discussion.

Pulumi's current approach to testing can feel unfamiliar to .NET developers, particularly the reliance on hard-coded "magic strings," as reflected in the *Magic Strings from Schemas* issue subcategory. This concern aligns with the findings of Rahman et al. [16], who identified a strong correlation between hard-coded strings and defects in infrastructure code. While their work focuses on the IaC program under test, we argue that similar risks apply to test code, which must reliably evolve alongside the system it verifies. Another contributor to design complexity is Pulumi's reliance on `IMocks`, which adds verbosity and requires understanding of internal mechanics, as seen in Listing 2. Additionally, `IMocks` must be implemented and managed in a separate class, reducing cohesion between the test logic and its supporting mocks.

In contrast, Pulumock embraces established .NET conventions through strong typing and a familiar fluent interface. By removing the need to implement `IMocks`, Pulumock reduces verbosity and allows test fixtures to be constructed or extended directly within test cases. This abstracts away complexity, improves cohesion and ultimately makes tests easier to write and understand. These design choices were well received in the within-subjects study, where developers reported that the *W* condition was easier to adopt and maintain. This was further supported by faster task completion times in the *W* condition. Notably, in the *W/o* condition, task completion times varied greatly, with the professional DevOps engineer ranking only third in speed. In contrast, student times were nearly identical in the *W* condition, while the DevOps engineer was the fastest and showed the greatest improvement. Although based on a small sample, this suggests that Pulumock's design aligns better with practices of experienced .NET developers.

2) Comparing Pulumock and ProTI: Scope and Design Differences: While Pulumock was designed in response to reported issues, it also draws inspiration from Sokolowski et al.'s ProTI [4], but ultimately adopts a different approach. ProTI focuses on stack testing by invoking the test runner once per `Pulumi.yaml`, treating the entire Pulumi program as a single unit. In contrast, Pulumock supports both stack and component-level testing, with an emphasis on unit testing *component resources* to improve modularity. These components

serve not only as reusable modules, but also as logical boundaries for testable units. We argue that unit testing in PL-IaC is better suited for smaller components, while broader stack-level testing is more appropriate at higher levels of the testing pyramid, such as integration testing. In addition to this shift in scope, Pulumock addresses functional limitations observed in ProTI. It enables mocking of *stack configuration* and *stack references*, and supports more complex scenarios such as asserting on resource relationships. These gaps, some of which are breaking, may help explain ProTI's limited adoption in practice. At the same time, Pulumock could benefit from incorporating some of ProTI's ideas to better support large-scale testing. For instance, providing reusable test oracles and integrating automated mocking into the `FixtureBuilder` could reduce manual effort.

3) Downsides of Third-Party Tools: While tools such as Pulumock and ProTI have the potential to enhance test adoption in Pulumi, a more sustainable approach would involve improving Pulumi's native testing capabilities. Relying on third-party libraries introduces dependencies on active maintenance, community support, and trust in both the tool's functionality and the stability of its licensing model. As expressed by P2 in the within-subjects study, developers may be reluctant to adopt such tools. This is especially relevant to the .NET ecosystem given recent trends where previously free tools, such as Moq and FluentAssertions, have made sudden moves to paid licenses for commercial use[62].

4) Suggestions for Future Work: As noted in prior work [1], [2], [7], research on IaC testing remains limited, highlighting the ongoing need for improved testing approaches, particularly for PL-IaC. Future work should focus on gathering more data on PL-IaC program design and its influence on unit testing. Important questions include how these programs can be structured to encourage unit testing, where unit tests are most appropriately applied, and how they align with the infrastructure testing pyramid. Addressing these issues is crucial for increasing adoption, as many developers still perceive unit testing as unsuitable for infrastructure code.

C. How the Findings Apply to General PL-IaC

Although the case in this thesis centers on Pulumi .NET, it is valuable to explore whether the findings reflect broader patterns contributing to low test adoption in PL-IaC. In particular, it is useful to understand why Pulumi shows only 1% unit testing adoption across all supported languages, compared to 15% for CDKTF and 38% for AWS CDK.

1) Mocking Complexity in Pulumi: One possible explanation for this difference is that the two-phase CDK tools avoid the need for mocking, as suggested by Sokolowski et al. [4]. Our findings in the curated dataset further support this perspective, as mocking-related issues received the highest number of reactions, indicating that mocking is a major barrier for developers.

2) Test Scaffolding Convenience in CDK Tools: Another possible explanation is that the CDK CLI scaffolds unit tests by default, which Pulumi does not, something also noted by Sokolowski et al. [4]. Interestingly, we also identified contrasting factors within AWS CDK that may partially help explain why 49% of AWS CDK TypeScript programs in the PIPr dataset included tests, compared to just 2% for .NET. Specifically, we found that the AWS CDK includes a basic test setup for TypeScript, while no such scaffolding was available in .NET. We also saw that the official example repository[63] contained some examples of testing for TypeScript, but no examples for .NET. Additionally, the provided .NET testing guide[64] failed to compile due to syntax errors, access modifiers, and missing custom `using` statements. Tests also crashed unless source files were manually copied to the output directory and due to use of outdated Node.js[65] versions. Furthermore, the example's implementation logic did not align with what the tests were asserting. Some resources expected by the tests were not actually created by the example code, leading to test failures where success was expected.

This suggests that including test scaffolding out of the box or working examples could improve testing adoption. Note however, that this could be skewed since TypeScript projects might have scaffolded tests included that might not have been further adjusted by developers to make any real assertions, a validity threat also considered by the creators if PIPr.

3) Similar Challenges Across PL-IaC Tools: While the CDK tools have greater test adoption than Pulumi, they still only peak at 38%, indicating these tools also need improvements. We argue that the CDK tools suffer from similar issues as Pulumi in other areas, although this needs more exploration. One notable example is the use of magic strings in AWS CDK's official testing guide[66], as shown in Listing 4. Line 1 uses a string identifier for the Lambda Function resource and lines 2-4 uses a dictionary of properties with string keys. This design corresponds to the *Magic Strings from Schemas* issue subcategory in this study, a contributing factor to increased complexity.

Listing 4: AWS CDK magic strings in assertion

[62] https://dariusz-wozniak.github.io/fossed/

[63] https://github.com/aws-samples/aws-cdk-examples

[64] https://docs.aws.amazon.com/cdk/v2/guide/testing.html

[65] https://nodejs.org/en

[66] https://docs.aws.amazon.com/cdk/v2/guide/testing.html

```
template.HasResourceProperties("AWS::Lambda
    ::Function", new Dictionary<string,
    string> {
    { "Handler", "handler"},
    { "Runtime", "nodejs22.x" }
});
```

4) Suggestions for Future Work: All of these factors contribute to low adoption of testing in PL-IaC and we see that some issues span across multiple PL-IaC tools. For future work it would be interesting to compare the testing experience across PL-IaC tools and gather similarities and contrasts to understand what the PL-IaC tools can borrow from each other to improve the ecosystem as a whole.

D. Threats to Validity

This section presents threats to the validity of our study using the categorization proposed by Feldt et al. [23], which distinguishes between conclusion, internal, construct, and external validity. For each identified threat, we briefly describe the corresponding mitigation strategy.

1) Internal Validity: The within-subjects design introduces a risk of learning effects, where exposure to one condition (i.e., *W/o*) may improve performance in the other (i.e., *W*). To mitigate this, we evenly alternated the starting order of conditions between the participants. Furthermore, the similarity between the documentation examples and the evaluation tasks may have made it easier for the participants to complete the tasks in the *W* condition. However, participants still had to understand and apply the examples correctly to complete the tasks, which involved the integration of multiple components.

Task completion times may have been influenced by the level of assistance provided. To control for this, the type of help was kept consistent across conditions, limited to general C# syntax guidance, documentation pointers, and minor hints regarding tooling usage. It is worth noting that support was needed more frequently in the *W/o* condition, while participants in the *W* condition generally completed the task with little or no help, as reflected in the assessment questionnaire. Nonetheless, task completion times were shorter in the *W* condition, suggesting that faster completion was not simply due to receiving more help.

2) Construct Validity: While shorter task completion times do not fully capture the complexity of the conditions or directly indicate mitigated challenges or increased test adoption, we interpret them alongside responses to the learning curve question in the assessment questionnaire. Together, reduced time and a perceived lower learning curve suggest reduced perceived complexity, which may help lower the barrier to test adoption.

While the assessment questionnaire included questions on readability, reusability, and real-world usage preference, the examples provided were limited and may have been difficult for participants to fully assess. However, we argue that even small-scale examples reveal patterns that offer insight into how these factors might manifest at scale. Unlike the question regarding learning curve, these questions addressed maintainability, focusing on managing tests as the PL-IaC program grows. We believe these questions are highly relevant, as a short learning curve alone is not enough to ensure continued use. If tests become too difficult to maintain over time, developers are likely to abandon them, ultimately leading to lower adoption rates or reduced test coverage.

3) External Validity: The tooling and test cases were shaped by specific issues and the evaluation did not include real-world production systems. As a result, our findings may limit the ability to determine whether Pulumock improves testability in a general sense, rather than just in selected cases. Future evaluations should include a broader set of test cases and involve more participants to strengthen the results.

This case study focused specifically on Pulumi .NET, so it is unclear whether the findings generalize to other Pulumi GPL instances, which are maintained independently and may differ in design or developer adoption. Furthermore, while Pulumi shares high-level concepts with the two-phase CDK tools, their deployment and testing models differ considerably (see Sections III-C and III-D). Therefore, the challenges and solutions discussed here may not be directly transferable to those tools.

E. Societal and Ethical Considerations

Improving testing practices for IaC carries meaningful benefits for both society and the IT industry. By enabling earlier detection of issues, testing can prevent misconfigurations from reaching production environments. This not only enhances the reliability of infrastructure code but also reduces the risk of security vulnerabilities and outages that could affect businesses and end-users.

There are also environmental and economic considerations. While integration tests are critical for validating real-world behavior, they are resource-intensive, often requiring the provisioning of ephemeral environments. Catching errors only at this stage can result in wasted developer time, increased cloud costs, and unnecessary energy consumption. Enhancing unit testing capabilities allows developers to identify basic issues earlier in the development cycle, reducing reliance on costly integration tests.

From an ethical standpoint, one key concern is the risk of false confidence. If Pulumock is misconfigured or fails to accurately simulate infrastructure behavior,

developers may be misled by passing unit tests while real deployments continue to fail. Furthermore, the long-term effectiveness of Pulumock depends on sustained maintenance and community involvement. This reliance raises questions about the tool's reliability over time and the trust developers can place in it. For instance, Pulumock currently lacks comprehensive test coverage for its internal components, which may impact its robustness and credibility.

VIII. Conclusions

This thesis explored the low adoption of unit testing in PL-IaC through a case study on Pulumi .NET, focusing on the challenges developers face and how targeted tooling can help overcome them.

By analyzing developer-reported issues in the Pulumissues dataset, we found that the most common challenges relate to the complexity and limited coverage of mocking and access to test data for assertions. These issues were often caused by unclear or incomplete testing APIs. We argue that such limitations discourage developers from writing unit tests, either due to the complexity involved or the inability to cover desired scenarios.

In response, we developed Pulumock, a lightweight tool designed to improve the unit testing experience and specifically target reported challenges. A within-subjects study, using test cases derived from common challenges, showed that Pulumock improved perceived adoptability and maintainability compared to Pulumi's default testing framework. Albeit on a small scale, these findings suggest that tooling design choices and feature support can help mitigate common challenges and lower the barrier for testing adoption in PL-IaC.

These findings are specific to the case of Pulumi .NET, and due to notable differences between other Pulumi GPL instances and two-phase CDK tools, the results can only be partially generalized.

Future research should conduct interviews or surveys with PL-IaC developers, and compare testing experiences across tools to identify shared and tool-specific challenges. It should also explore how PL-IaC program design affects unit testing, particularly how meaningful boundaries can be established within complex dependency graphs to support testing of smaller, isolated components. Moreover, the role of unit tests in the infrastructure testing pyramid is still not well understood, and further research is needed to develop clear and practical testing guidelines.

References

[1] A. Rahman, R. Mahdavi-Hezaveh, and L. Williams, "A systematic mapping study of infrastructure as code research," *Information and Software Technology*, vol. 108, pp. 65–77, 2019. [Online]. Available: https://doi.org/10.1016/j.infsof.2018.12.004

[2] M. Guerriero, M. Garriga, D. A. Tamburri, and F. Palomba, "Adoption, support, and challenges of infrastructure-as-code: Insights from industry," in *Proc. 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, Cleveland, OH, USA, 2019, pp. 580–589. [Online]. Available: https://doi.org/10.1109/ICSME.2019.00092

[3] N. Saavedra, J. F. Ferreira, and A. Mendes, "Infrafix: Technology-agnostic repair of infrastructure as code," *arXiv preprint arXiv:2503.17220*, 2025. [Online]. Available: https://arxiv.org/abs/2503.17220

[4] D. Sokolowski, D. Spielmann, and G. Salvaneschi, "Automated infrastructure as code program testing," *IEEE Transactions on Software Engineering*, vol. 50, no. 6, pp. 1585–1599, 2024. [Online]. Available: https://doi.org/10.1109/TSE.2024.3393070

[5] N. Suwanachote, S. Pornmaneerattanatri, Y. Kashiwa, and K. Ichikawa, "A pilot study of testing infrastructure as code for cloud systems," in *Proc. 30th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, Seoul, Korea, 2023, pp. 584–588. [Online]. Available: https://doi.org/10.1109/APSEC60848.2023.00075

[6] D. Sokolowski, D. Spielmann, and G. Salvaneschi, "The pipr dataset of public infrastructure as code programs," in *Proc. 21st International Conference on Mining Software Repositories (MSR '24)*. Association for Computing Machinery, New York, NY, USA, 2024, pp. 498–503. [Online]. Available: https://doi.org/10.1145/3643991.3644888

[7] G.-P. Drosos, T. Sotiropoulos, G. Alexopoulos, D. Mitropoulos, and Z. Su, "When your infrastructure is a buggy program: Understanding faults in infrastructure as code ecosystems," in *Proc. ACM Programming Languages, 8, OOPSLA2, Article 359 (October 2024)*. Association for Computing Machinery, New York, NY, USA, 2024, pp. 2490 – 2520. [Online]. Available: https://doi-org.proxybib.miun.se/10.1145/3689799

[8] M. M. Hasan, F. A. Bhuiyan, and A. Rahman, "Testing practices for infrastructure as code," in *Proceedings of the 1st ACM SIGSOFT International Workshop on Languages and Tools for Next-Generation Testing (LANGETI 2020)*. Association for Computing Machinery, New York, NY, USA, 2020, pp. 7–12. [Online]. Available: https://doi.org/10.1145/3416504.3424334

[9] K. Morris, *Infrastructure as Code, 3rd Edition*, 3rd ed. O'Reilly Media, Inc., 2025.

[10] V. Sikha, D. Siramgari, and S. Somepalli, "Infrastructure as code: Historical insights and future directions," *International Journal of Science and Research (IJSR)*, vol. 12, pp. 2549–2558, 2023. [Online]. Available: https://doi.org/10.21275/SR24820064820

[11] I. Aviv, R. Gafni, S. Sherman, B. Aviv, A. Sterkin, and E. Bega, "Infrastructure from code: The next generation of cloud lifecycle automation," *IEEE Software*, vol. 40, no. 1, pp. 42–49, 2023.

[12] G. Simhandl and U. Zdun, "Cloud programming languages and infrastructure from code: An empirical study," in *Proceedings of the 17th ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 143–156. [Online]. Available: https://doi.org/10.1145/3687997.3695643

[13] AWS. (2025) What is the aws cdk? Accessed: 2025-04-09. [Online]. Available: https://docs.aws.amazon.com/cdk/v2/guide/home.html

[14] HashiCorp. (2025) Architecture. Accessed: 2025-04-09. [Online]. Available: https://developer.hashicorp.com/terraform/cdktf/concepts/cdktf-architecture

[15] Pulumi. (2025) How pulumi works. Accessed: 2025-04-09. [Online]. Available: https://www.pulumi.com/docs/iac/concepts/how-pulumi-works/

[16] A. Rahman and L. Williams, "Source code properties of defective infrastructure as code scripts," *Information and Software Technology*, vol. 112, pp. 148–163, 2019. [Online]. Available: https://doi.org/10.1016/j.infsof.2019.04.013

[17] A. Rahman, M. R. Rahman, C. Parnin, and L. Williams, "Security smells in ansible and chef scripts: A replication study," *ACM Transactions on Software Engineering and Methodology*

(TOSEM), Volume 30, Issue 1, vol. 30, pp. 148–163, 2021. [Online]. Available: https://doi.org/10.1145/3408897

[18] I. Kumara, M. Garriga, A. U. Romeu, D. D. Nucci, F. Palomba, D. A. Tamburri, and W.-J. van den Heuvel, "The do's and don'ts of infrastructure code: A systematic gray literature review," *Information and Software Technology*, vol. 137, p. 106593, 2021. [Online]. Available: https://doi.org/10.1016/j.infsof.2021.106593

[19] P. Runeson, M. Host, A. Rainer, and B. Regnell, *Case study research in software engineering: Guidelines and examples*. John Wiley & Sons, 2012.

[20] T. Rietveld and R. van Hout, "The paired t test and beyond: Recommendations for testing the central tendencies of two paired samples in research on speech, language and hearing pathology," *Journal of Communication Disorders*, vol. 69, pp. 44–57, 2017. [Online]. Available: https://doi.org/10.1016/j.jcomdis.2017.07.002

[21] L. L. Havlicek and N. L. Peterson, "Robustness of the t test: A guide for researchers on effect of violations of assumptions," *Psychological Reports*, vol. 34, no. 3_suppl, pp. 1095–1114, 1974. [Online]. Available: https://doi.org/10.2466/pr0.1974.34.3c.1095

[22] D. Linnell, "Dependent t-test." [Online]. Available: https://danalinnell.github.io/statistics-with-jamovi/dependent-t-test.html#step-2-check-assumptions-5

[23] R. Feldt and A. Magazinius, "Validity threats in empirical software engineering research—an initial survey," in *Proceedings of the International Conference on Software Engineering and Knowledge Engineering (SEKE)*, 2010, pp. 374–379.

Appendix 1: Experience Questionnaire

This appendix describes the pre-study questionnaire used to collect background information about the participants.

1) What is your current role? (e.g., Student, Backend Developer, DevOps Engineer, etc.)
   - *[Free text response]*
2) How many years of programming experience do you have?
   - 0–2 years
   - 2–5 years
   - 5–8 years
   - 8+ years
3) How much experience do you have working with .NET and C#?
   -
   - No experience: I have never worked with .NET or C#.
   - Beginner: I have basic knowledge and can write simple programs with limited guidance
   - Intermediate: I can build and maintain applications using .NET and C#, and solve common problems independently.
   - Advanced: I have extensive experience with .NET and C#, including building and maintaining professional, production-level applications.
4) How much experience do you have with unit testing?
   - No experience: I have not written or worked with unit tests before and I am unfamiliar with the process or tools involved.
   - Beginner: I understand the concept of unit testing and have written a few basic test cases.
   - Intermediate: I regularly write unit tests and use testing frameworks as part of my development workflow.
   - Advanced: I have extensive experience designing and maintaining comprehensive test suites.
5) How much experience do you have with Pulumi?
   - No experience: I don't know what Pulumi is or have never used it.
   - Some experience: I've heard of it and may have used it before.
   - Beginner: I've explored Pulumi through tutorials or small personal projects for basic infrastructure provisioning.
   - Intermediate: I've used Pulumi in real projects to manage infrastructure, and I'm comfortable with its core features.
   - Advanced: I have extensive experience writing and managing large-scale Pulumi programs for production environments.
6) How much experience do you have with other IaC tools? (e.g., AWS CDK, CDKTF, Terraform, etc.).
   - No experience: I don't know what IaC tools are or have never used any of them.
   - Some experience: I've heard of it and may have used it before.
   - Beginner: I've explored IaC tools through tutorials or small personal projects for basic infrastructure provisioning.
   - Intermediate: I've used IaC tools in real projects to manage infrastructure, and I'm comfortable with their core features.
   - Advanced: I have extensive experience writing and managing large-scale IaC programs for production environments.

Appendix 2: Assessment Questionnaire

This appendix presents the post-study questionnaire used to collect participant feedback following the evaluation tasks. It includes both multiple-choice and open-ended questions.

1) Which approach had a smoother learning curve in regards to unit testing?
   - Pulumock
   - Without Pulumock
   - No Preference
2) Motivate your answer
   - *[Free text response]*

3) Which approach offered better readability in your code?
   - Pulumock
   - Without Pulumock
   - No Preference
4) Motivate your answer
   - *[Free text response]*
5) Which approach offered better reusability in your code?
   - Pulumock
   - Without Pulumock
   - No Preference
6) Motivate your answer
   - *[Free text response]*
7) Which approach would you personally prefer to use in a real-world project?
   - Pulumock
   - Without Pulumock
   - No Preference
8) Motivate your answer
   - *[Free text response]*
9) What challenges (if any) did you face when using either approach?
   - *[Free text response]*
10) What was your overall experience using Pulumock?
   - *[Free text response]*
11) Do you have any suggestions for improving Pulumock?
   - *[Free text response]*

Appendix 3: Participants (P) with experience across relevant areas

P	Role	Area	Experience
1	DevOps Engineer	Programming	8+ years
		.NET & C#	Intermediate
		Unit Testing	Advanced
		Pulumi	No experience
		IaC Tools	Advanced
2	Student	Programming	2–5 years
		.NET & C#	Beginner
		Unit Testing	Beginner
		Pulumi	No experience
		IaC Tools	Beginner
3	Student	Programming	0-2 years
		.NET & C#	No experience
		Unit Testing	Beginner
		Pulumi	No experience
		IaC Tools	Some experience
4	Student	Programming	2-5 years
		.NET & C#	No experience
		Unit Testing	Beginner
		Pulumi	No experience
		IaC Tools	Some experience

Appendix 4: Participant (P) task completion times with (W/) and without (W/o) Pulumock

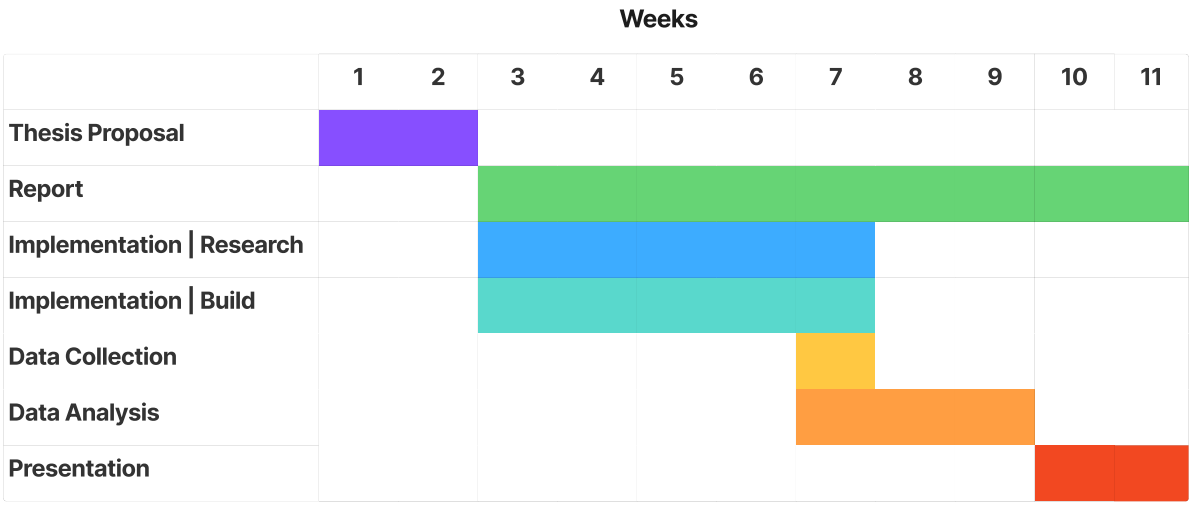
P	Starting tool	Task	Time (min)
1	W/o	W – Task 1	05:30
		W – Task 2	03:30
		W – Total	**09:00**
		W/o – Task 1	15:30
		W/o – Task 2	20:00
		W/o – Total	**35:30**
2	W	W – Task 1	05:00
		W – Task 2	07:30
		W – Total	**12:30**
		W/o – Task 1	18:30
		W/o – Task 2	11:00
		W/o – Total	**29:30**
3	W/o	W – Task 1	08:00
		W – Task 2	05:00
		W – Total	**13:00**
		W/o – Task 1	22:45
		W/o – Task 2	15:30
		W/o – Total	**38:15**
4	W	W – Task 1	06:00
		W – Task 2	07:00
		W – Total	**13:00**
		W/o – Task 1	15:00
		W/o – Task 2	18:20
		W/o – Total	**33:20**

Appendix 5: Contributions

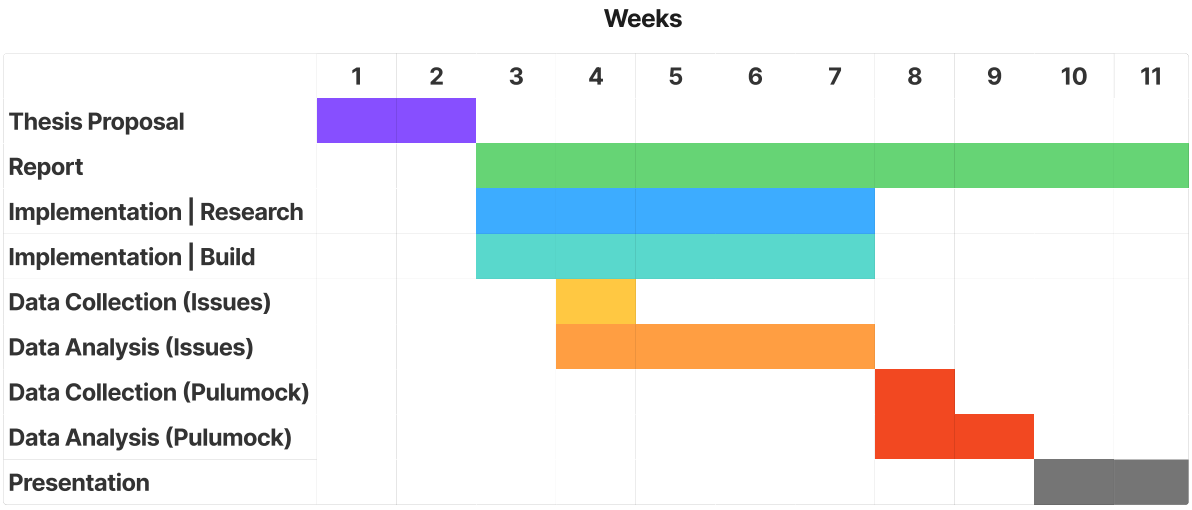
Area	Albin (%)	Piran (%)
Report - Abstract	50%	50%
Report - Introduction	50%	50%
Report - Purpose and Contributions	50%	50%
Report - Background	100%	0%
Report - Related Work	40%	60%
Report - Methodology	50%	50%
Report - Results	60%	40%
Report - Discussion	60%	40%
Report - Conclusions	60%	40%
Tool - Pulumissues	80%	20%
Tool - Pulumock	90%	10%
Tool - Puluvaluation	70%	30%

In terms of tool development, Albin has more experience with Pulumi, C#, and .NET. Due to time constraints, he was responsible for the implementation to accelerate development, as extensive pair programming or parallel work would have required ongoing support and been less efficient. In terms of the Background section, Albin's stronger background in IaC made it easier for him to explain the topic clearly. Overall, we have collaborated closely, contributed fairly, and continuously built on each other's ideas throughout the project.

Appendix 6: Time Plan



Initial Time Plan



Actual Time Plan

Addressing the Testing Gap in PL-IaC: A Case Study on Pulumi .NET

Authors: Albin Rönnkvist & Piran Amedi | {alrn1700, roam2200}@student.miun.se
Department of Communication, Quality Management and Information Systems
Mid Sweden University, Östersund, Sweden

Infrastructure as Code (IaC) brings software engineering practices to infrastructure. However, testing remains challenging and underutilized, especially in Programming Language-based IaC (PL-IaC). Low adoption rates threaten system reliability and highlight the need for better testing support. This thesis investigates barriers to unit testing in PL-IaC and explores how tailored tooling can improve adoption.

01 Introduction

Context

PL-IaC has aligned infrastructure management with Software Engineering practices by enabling the use of programming languages. However, testing practices remain limited and challenging, with only **25% adoption across the ecosystem**. This gap threatens reliability, as undetected defects can lead to security vulnerabilities and outages, impacting the stability of our increasingly digitalized society.

Contributions

This thesis investigates the barriers to unit testing in PL-IaC through a case study of Pulumi .NET, a widely used tool with the **lowest testing adoption rate at only 1%**. It examines developer-reported challenges, presents a tool designed to address them, and evaluates its effectiveness compared to existing solutions.

Research Questions

The research questions are framed around the case, while also offering insights that extend to the broader PL-IaC ecosystem:

- **RQ1**: What are the common challenges developers face when unit testing Pulumi .NET programs?
- **RQ2**: In what ways can Pulumock mitigate common challenges when unit testing Pulumi .NET programs?
- **RQ3**: How does Pulumock compare relative to Pulumi's default testing framework in terms of mitigating common challenges when unit testing Pulumi .NET programs?

02 Methodology

Dataset Construction

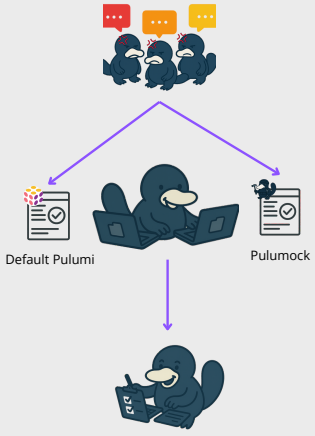
In response to RQ1, we developed Pulumissues to identify common unit testing challenges reported in public forums.

Tool Construction

As for RQ2, we built Pulumock, a tool designed to mitigate the previously identified challenges.

Tool Evaluation

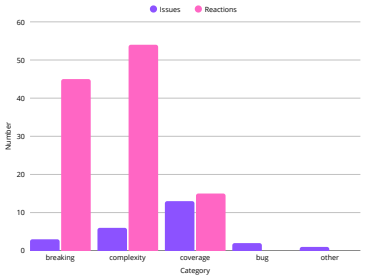
To answer RQ3, we conducted the Puluvaluation within-subjects study, comparing Pulumock to Pulumi's default testing framework. Participants completed tasks and questionnaires to evaluate whether and how Pulumock mitigated the identified challenges and impacted the overall testing experience.



03 Results

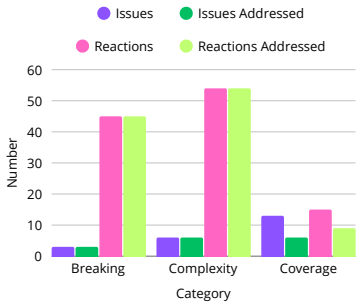
RQ1

Based on 20 relevant issues, our analysis showed that the most common challenges involve the complexity and lack of coverage when mocking and accessing test data. These challenges often arise from incomplete or unclear testing APIs.



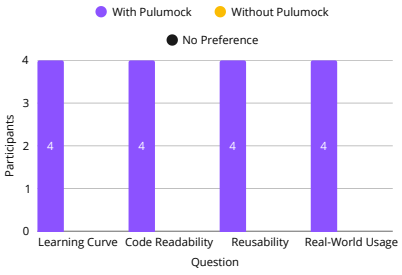
RQ2

Pulumock targeted several challenges spanning multiple issue categories. All reported issues in the two prioritized categories, *breaking* and *complexity*, and 6 out of 13 of issues in the less prioritized *coverage* category.



RQ3

All participants preferred Pulumock over Pulumi's default testing framework when implementing test cases involving common challenges. Tasks were completed 65% faster on average, and positive questionnaire feedback indicated a more efficient, adoptable, and maintainable testing experience with Pulumock.



04 Conclusion

Reported issues highlighted challenges with mocking and accessing test data due to inadequate APIs. To address this, we developed Pulumock, which participants consistently preferred over Pulumi's default testing framework. This suggests that targeted tooling can reduce barriers and support greater testing adoption in the PL-IaC community.

Future work should examine more PL-IaC tools, the impact of program design on testability, and the role of unit tests in infrastructure testing.