Technische Universität Berlin

Information Systems Engineering
Fachgebiet Wirtschaftsinformatik

Fakultät IV
Einsteinufer 17
10587 Berlin
http://www.ise.tu-berlin.de

Thesis

Quantitative Cost Assessment of Infrastructure as Code Testing

Felix Miske
thesis@miske.email

Matriculation Number: 307753
02.02.2024

Supervised by
Prof. Dr. Stefan Tai
Prof. Dr. David Bermbach

Assistant Supervisor
Dr. Sebastian Werner

Abstract

The advent of DevOps has transformed software development agility and efficiency, integrating development and operations. Infrastructure as Code (IaC) is at the heart of this transformation, which automates and manages infrastructure with the precision of application code. However, despite its critical role, the testing of IaC remains in its infancy, particularly when contrasted with the established practices in traditional software testing. This gap is evident in three key areas: the lack of mature testing conventions, unique cost implications of IaC testing, and fragmentation in the IaC test tool landscape.

This thesis has three primary objectives. First, it seeks to shed light on the existing research gap in IaC testing, taking initial steps toward establishing empirically grounded best practices. Second, it provides a consolidated overview of practiced test approaches using a quantified methodology for comparison and selection. Finally, it focuses on developing a flexible test pipeline incorporating various IaC test tools to measure and compare test efficiency and costs.

The research methodology involves three critical steps: categorizing defect scenarios for IaC testing, determining which test approaches for IaC provisioning cover which defect categories, and evaluating these test approaches concerning execution time and cloud-provider-related costs. This methodology aims to assess and quantify test approaches' suitability and cost efficiency, highlighting their strengths and limitations.

The thesis confirms that static analysis methods in IaC testing yield faster results than dynamic testing, a previously unquantified but suspected advantage in the field. It reveals that the effectiveness of static unit tests is contingent on the defect category, allowing for a strategic balance between static and dynamic testing based on specific needs. Significantly, the thesis delineates the exclusive capabilities of static and dynamic methods in addressing different defect categories, facilitating a more precise and cost-efficient allocation of testing resources. This advancement in understanding empowers developers to optimize test strategies, leading toward a more effective and economical testing practice for IaC provisioning.

Further research could include validating findings by applying the methodology and Test Pipeline to a significant number of Terraform projects, extending the methodology to additional IaC tools and aspects, and collaborating with industry partners to validate and refine the theoretical model in real-world scenarios.

Zusammenfassung

Das Aufkommen von DevOps hat, durch die Integration von Entwicklungs- und Betriebsprozessen, die Agilität und Effizienz der Softwareentwicklung vorangetrieben. Infrastructure as Code (IaC) ist das Herzstück dieses Wandels, das die Anwendungsinfrastruktur mit der Präzision von Softwarecode automatisiert und verwaltet. Trotz dieser Schlüsselrolle steckt das Testen von IaC jedoch noch in den Kinderschuhen, insbesondere wenn man es mit den etablierten Praktiken der traditionellen Qualitätssicherung vergleicht. Diese Lücke zeigt sich in drei kritischen Bereichen: Die fehlende Standardisierung von Testkonventionen, die unbeachteten Kostenimplikationen von IaC-Tests und das Fehlen einheitlicher, integrativer IaC-Testwerkzeuge.

Die vorliegende Arbeit verfolgt drei Hauptziele, um der identifizierten Forschungslücke entgegenzuwirken. Zunächst zielt sie darauf ab, das Forschungsdefizit im Bereich des IaC-Testens zu verdeutlichen und einen Anstoß für die Schaffung empirisch basierter Verfahren zu geben. Das zweite Ziel ist die Erstellung eines konsolidierten Überblicks über praktizierte Testansätze, sowie die Entwicklung einer quantifizierenden Methode zum Vergleich dieser Ansätze. Als abschließendes Ziel strebt diese Arbeit die Entwicklung einer anpassungsfähigen Testpipeline an, die unterschiedliche IaC-Testwerkzeuge integriert, um gezielt Messdaten zu den Laufzeitkosten von IaC-Tests zu erfassen.

Die Forschungsmethodik umfasst drei entscheidende Schritte. Zunächst erfolgt die Kategorisierung von Fehlerszenarien für IaC Konfigurationen. Anschließend erfolgt eine Zuordnung von Testansätze für IaC Provisioning zu Fehlerkategorien. Schließlich werden diese Testansätze hinsichtlich der Ausführungszeit und der Cloud-Provider-bezogenen Kosten bewertet. Ziel dieser Methodik ist es, die Eignung und Kosteneffizienz von Testansätzen zu quantifizieren.

Die Arbeit bestätigt, dass statische Analysemethoden bei IaC-Tests signifikant schnellere Ergebnisse als dynamische Tests, ein bisher nicht quantifizierter, aber angenommener Vorteil. Sie verdeutlicht jedoch auch, dass die Anwendbarkeit und Abdeckung der Testmethoden von der jeweiligen Fehlerkategorie abhängen. Dies führt zu der Erkenntnis, dass die optimale Balance zwischen statischen und dynamischen Testverfahren maßgeblich vom spezifischen Einsatzkontext bestimmt wird. Hierzu werden in der Thesis die Stärken und Schwächen der verschiedenen Testmethoden vor dem Hintergrund der IaC Fehlerkategorien analysiert und evaluiert, um so eine präzisere und kosteneffizientere Zuweisung von Testressourcen zu ermöglichen. Dieser Fortschritt im Verständnis befähigt Entwicklern, Teststrategien zu optimieren, was zu einer effektiveren und wirtschaftlicheren Testpraxis für IaC Provisioning führt.

Contents

List of Figures

List of Tables

1 Introduction

Integrating development and operations practices, known as DevOps, has revolutionized software development, fostering agility and efficiency. IaC is at the core of this transformation, a paradigm that manages and automates infrastructure with the precision of application code. This thesis focuses on a critical aspect of IaC testing within the domain of IaC provisioning, which, along with configuration management and templating, forms the triad of primary IaC aspects [GGTP19, KGR+21, Wan22]. The research specifically examines testing in the context of Terraform, a prominent IaC provisioning tool.

DevOps, an approach centered around optimizing for change [FHK18, Mor20], seeks to enhance delivery pipelines and product stability. DevOps practices significantly accelerate the delivery of new features and updates by reducing the effort and risk associated with each change. In this context, IaC is not merely an operational convenience but a fundamental necessity. It ensures the robustness and stability of not only the infrastructure that supports or constitutes the products delivered but also the CI/CD pipelines themselves, which embody DevOps and Agile development practices. The robustness of these pipelines, defined and managed through IaC, is crucial for the effectiveness of the entire DevOps approach.

Given this context, it is crucial to emphasize the importance of testing in IaC. Testing is the cornerstone in enhancing stability and robustness - regardless of the domain. Surprisingly, despite its relevance, the field of IaC testing remains underexplored [RMW19]. While a limited number of studies exist for configuration management tools such as Ansible, Chef, or Puppet, the investigation into IaC provisioning tools, particularly Terraform, reveals an even more acute research deficit. This scarcity is noticeable in three key areas:

Maturity of Test Conventions: There is a notable lack of agreed-upon nomenclature, best practices, and approaches in IaC testing. Questions such as "What are the effective test approaches?" and "How can different test approaches be balanced?" remain largely unanswered. This lack of standardized test conventions poses significant challenges in establishing effective and efficient test procedures.

Unique Cost Implications of IaC Testing: Dynamic IaC testing differs markedly from traditional software testing regarding costs and logistics. The lack of mocking tools in IaC eliminates the possibility of isolated component tests, leading to prolonged deployment times. Furthermore, additional costs are incurred as infrastructure is often deployed to public cloud providers. These factors result in a higher cost for dynamic testing in IaC compared to traditional software testing, making static test approaches more attractive [CDPP22]. However, the questions of which static test approaches are used in practice and when and where they can replace or supplement dynamic approaches still need to be explored.

Fragmentation of the IaC Test Tool Landscape: The landscape of IaC test tools is characterized by fragmentation across different IaC aspects and specific tools like Terraform. The evolution and abandonment of numerous test tools further compound this fragmentation. While some tools may support more than one test approach, each tool stands alone, lacking integration with other tools that support additional test approaches. There is an absence of a comprehensive test suite that covers all or even the most commonly used test approaches in Terraform testing.

The focus on provisioning is particularly pertinent for larger organizations where the relevance of provisioning immutable infrastructure escalates with project size. For such organizations, the stability ensured by thorough testing becomes increasingly crucial, aligning with the ability to absorb the initial investment required to reap the mid- to long-term benefits [Wan22] of IaC testing.

Our methodology systematically assesses and quantifies test approaches for IaC provisioning to address this need. The intent is to equip practitioners with the essential knowledge to develop test strategies that are both cost-effective and technically robust. This effort addresses critical issues such as gaps in test conventions, tool fragmentation, and the unique cost dynamics of IaC testing. The subsequent section will outline the methodologies employed in this thesis to address these challenges, guiding practitioners in formulating effective test strategies for IaC environments.

1.1 Objective

This thesis sets forth three distinct yet related objectives. These objectives are designed to address the challenges identified in the maturity of test conventions, the unique cost implications of IaC testing, and the fragmentation of the IaC test tool landscape.

Highlighting a critical research gap: The first objective is to draw attention to a critical yet underexplored research field in IaC testing. This work aims to be an initial step to bridge the existing research gap. By identifying additional research opportunities, we hope to prompt further academic investigation. The ultimate goal is to minimize guesswork and establish comprehensive best practices grounded in empirical data within the IaC testing domain. Addressing this objective will contribute to mitigating the first problem identified: the lack of maturity in test conventions. By exploring this under-researched area, the thesis will provide much-needed clarity and direction in practical test approaches, enhancing the understanding and implementation of IaC test procedures.

Consolidated Overview of Test Approaches: The second objective is to provide a consolidated overview of practiced test approaches for an exemplary IaC tool, employing a quantified methodology to compare and select the most suitable test approach for each task. These approaches will be integrated into a hierarchical system, guiding and supporting IaC practitioners in developing efficient and effective test strategies. This objective lays the theoretical groundwork for addressing the first problem of the maturity of test conventions and the second problem of unique cost implications in IaC testing. By systematically measuring and quantifying different test approaches, this thesis aims to clarify the cost-benefit dynamics inherent in IaC testing, offering insights into cost-effective and technically sound test strategies.

Proof-of-Concept (PoC) Implementation: The third objective focuses on a PoC implementation that will embody the requirements for encompassing the researched test approaches. This implementation is envisaged to have the capability to collect data to measure and compare the efficiency of various IaC test approaches. The design requirements for the PoC include flexibility to integrate a range of test tools and adaptability to the evolving nature of these tools, ensuring that it remains relevant and effective as the IaC testing landscape changes. By fulfilling these requirements, the PoC aims to address the fragmentation in the IaC test tool landscape, providing a unified and comprehensive solution for practitioners. This objective is crucial in transforming theoretical insights into practical applications, offering tangible means to understand the unique cost implications of IaC testing and effectively utilize the fragmented tool landscape.

These objectives motivate the investigation into quantifying the costs of IaC test approaches, culminating in the formulation of the central research question:

> **How can we measure the suitability and efficiency of existing test approaches to help developers formulate tailored test strategies for Infrastructure as Code?**

To thoroughly address this research question, the thesis is guided by the following sub-questions:

1. **How can defect scenarios for IaC testing be categorized?** Establishing these categories will provide a structured framework for assessing the effectiveness of various test approaches.

2. **Which test approaches for IaC provisioning can cover which defect categories?** This will determine the strengths and limitations of each test approach in addressing specific defect scenarios.

3. **How do different test approaches for IaC provisioning perform concerning execution time and cloud-provider-related costs?** Analyzing these factors will yield valuable insights into the efficiency and cost-effectiveness of each test level, aiding in the development of informed test strategies.

This structured procedure, starting from the objectives and leading to formulating a primary research question and subsequent guiding questions, is designed to address and resolve the challenges identified in IaC testing methodically. The goal is to provide developers with the knowledge and tools to create optimized and effective test strategies for IaC provisioning.

1.2 Scope

The scope of this thesis is intricately designed to align with the previously identified gaps and objectives. It encapsulates a focused exploration within the broader IaC testing domain, primarily concentrating on the provisioning aspect of IaC using Terraform. This narrowed focus is adopted to provide a deep and comprehensive understanding of IaC testing in the context of provisioning, which is central to the setup and management of infrastructure resources in DevOps environments.

Research and Methodological Focus The thesis commits to researching and consolidating various test approaches within the realm of IaC provisioning, emphasizing those applicable to Terraform. A methodology is presented to assess and quantify these approaches' suitability and cost efficiency. This methodology enables the comparison of different test approaches, highlighting their strengths and limitations in practical scenarios.

Justification for Focusing on Terraform Terraform is chosen for its significance in provisioning, a pivotal aspect of IaC, directly impacting the deployment and scalability of applications in DevOps environments. Its popularity and industry adoption provide a rich context for this research. The robust community and ecosystem surrounding Terraform offer a unique opportunity to study various practices and challenges in IaC testing. Terraform's method of infrastructure provisioning is representative of modern IaC practices, making it an ideal subject for an in-depth study. This focus on Terraform allows the research to delve into a crucial segment of IaC, providing valuable insights for academic researchers and industry practitioners.

Implementation of a Test Pipeline A Test Pipeline is implemented to cover the range of test approaches, each illustrated through one or multiple test tools. This pipeline measures runtime and calculates cloud provider costs for individual test cases. Demonstrating this pipeline involves verifying the suitability of test approaches using a concrete configuration under test (CUT) and implemented test cases.

Evaluation of Test Approaches The thesis evaluates the collected measurements from the PoC, focusing on general trends and actionable advice for IaC test practitioners. This evaluation is critical in understanding the broader implications of the research and its practical application in real-world scenarios.

Delimitations While this thesis provides a comprehensive study of IaC testing, certain areas are considered out of scope:

- The focus is on the provisioning aspect of IaC, specifically using Terraform, and does not extend to other areas like infrastructure templating or configuration management.
- The thesis's primary emphasis is on quantifying test execution costs, particularly runtime and related cloud provider charges. This scope excludes a detailed analysis of ancillary cost factors such as skills required, test design complexity, implementation efforts, and maintenance.
- This research employs AWS costs as a focal point, stating that, for this study, these figures sufficiently represent general cloud provider expenses and opportunity costs in contrast to on-premise solutions. This work aims to compare various test approaches within a project, not to determine the most cost-efficient cloud provider.
- The presented methodology for quantifying and comparing test approaches is exemplified using data specific to the chosen CUT and accompanying test cases. While this data effectively demonstrates the process, it is essential to acknowledge that these findings still need to be validated on a broader empirical base. Future research is required to confirm these trends across a more comprehensive range of IaC deployments.
- End-to-end tests were excluded from this thesis due to time and resource constraints. Their technical overlap with dynamic integration tests and the substantial development effort for comprehensive scenarios necessitated this limitation to maintain a focused and achievable research scope.

This thesis delineates a clear scope for exploring IaC testing with Terraform, addressing critical challenges in the field. The following section, 'Outline,' concludes this introductory chapter by presenting the structure of the thesis, guiding the reader through the progression from fundamental concepts to conclusive insights.

1.3 Outline

This thesis is organized as follows:

- **Chapter 2** introduces core concepts of IaC and software testing. It then lays out the current academic research and industrial practice of IaC Testing, highlighting key developments and methodologies.
- **Chapter 3** consolidates and organizes the diverse IaC test approaches into an IaC Test Pyramid and outlines a methodology to assess and quantify the approaches' suitability and efficiency.
- **Chapter 4** details the design and development of a test pipeline, including pipeline stages, data collection methods, and the CUT. It also covers the design, scope, and implementation of test cases.
- **Chapter 5** analyzes and discusses the suitability and efficiency of test approaches based on data collected from the PoC implementation. It presents conclusions from this analysis, providing insights into the effectiveness of different IaC test strategies.
- **Chapter 6** offers a comprehensive summary of the research findings, reflects on the challenges encountered, and discusses future research possibilities in the field of IaC testing.

2 Fundamentals and Related Work

This chapter lays the groundwork for understanding the essential concepts and prior research relevant to studying IaC testing. It begins by examining the fundamental aspects of IaC and then delves into its evolution, defining principles, and technological landscape, all discussed within the context of Cloud Computing and DevOps. This structure provides a thorough backdrop for the thesis and establishes a framework for understanding IaC's unique characteristics compared to traditional software testing methodologies.

The subsequent section 2.2 delves into the complexities of software testing, including an examination of the Test Pyramid concept. This concept, crucial in understanding the layers and scope of testing strategies, will later be adapted for IaC testing in this thesis. The section offers a comprehensive overview of test automation, regression testing, verification and validation processes, and software development test levels. It illustrates how these established paradigms contrast with and inform the emerging practices in IaC testing, underscoring this field's unique challenges and requirements.

Finally, in section 2.3, the chapter synthesizes existing literature and industrial practices in the domain of IaC testing. This includes an analysis of current test approaches, the identification of gaps and challenges, and the exploration of emerging trends. The review of related work serves as a critical analysis, setting the stage for the thesis to contribute novel insights and methodologies in the field of IaC testing.

2.1 IaC in the Context of Cloud Computing and DevOps

"Infrastructure as Code is an approach to automate infrastructure based on practices from software development." [Mor20]

This sentence from Kief Morris, one of the key people credited with formalizing and popularizing IaC [Ste15], summarizes the core concept of IaC. As a critical innovation in software engineering, IaC's evolution and contemporary relevance are inextricably linked with the advancements in Cloud Computing and the principles of DevOps. This section aims to explore these interconnections, providing a comprehensive understanding of how IaC is situated within and shaped by these broader technological domains.

The section begins with a brief history of IaC, Cloud Computing, and DevOps, illustrating their intertwined evolution. This historical perspective sets the stage for examining Cloud Computing and DevOps, framing their relevance to IaC. Finally, the discussion of IaC concepts concludes by delving into the principles of IaC and the nature of its different aspects and tools, elucidating their practical implications in software engineering.

2.1.1 Brief History

The roots of IaC can be traced back to the practices of system administrators who employed scripts for systems management. CFEngine, introduced in 1993, is a testament to these early endeavors, which existed well before "Infrastructure as Code" became a recognized term [Bur05]. The late 2000s to early 2010s period was pivotal for IaC and DevOps. During this time, the development of IaC and DevOps occurred concurrently, with each contributing to the reinforcement of the other, and both were intrinsically tied to the emerging domain of cloud computing [Mor20]. During this period, foundational works emerged that formalized IaC principles, providing structure and clarity to practices that had been evolving informally [Ste15].

Simultaneously, between 2006 and 2012, significant developments occurred in cloud computing [SR19]. Major tech players such as AWS, IBM, Microsoft, and Google introduced their cloud offerings, defining a new era of computing. Moreover, Rackspace and NASA's joint venture brought forth OpenStack, an open-source Cloud OS, adding another significant milestone to the timeline. Kief Morris refers to this profound shift as a change of an Age in Computer Science, from the Iron Age of physical hardware to the Cloud Age of virtualized resources [Mor20].

The rise of IaC, DevOps, and cloud computing is a narrative of interdependency and mutual reinforcement. While IaC and DevOps laid the foundation for effectively harnessing cloud capabilities, the flexibility and vast potential of cloud platforms, in turn, drove the need for structured, code-based infrastructure management methodologies. Resource configurations are defined in text files and then applied via scripts or specialized tools to provision and adapt the infrastructure. It enables the treatment of infrastructure definitions as versionable artifacts, akin to software code, leading to consistent and repeatable infrastructure deployments.

Google Trends data for the search term "Infrastructure as Code" provides a tangible lens into this trajectory. Before 2010, the topic of IaC received relatively little attention. As the decade progressed, its relevance gradually increased, with a more marked and consistent increase observed from around 2015. By 2021, this interest had reached a peak and firmly established itself, indicating a sustained and robust focus on the subject (see fig. 2.1).

This transition from the Iron Age to the Cloud Age sets a compelling context for the next section, which discusses cloud computing in detail. The exploration of cloud computing's characteristics, service models, and deployment models will further clarify its role and impact in the context of IaC.

2.1.2 Cloud Computing

"Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction." [MG11].

The National Institute of Standards and Technology (NIST) definition encapsulates

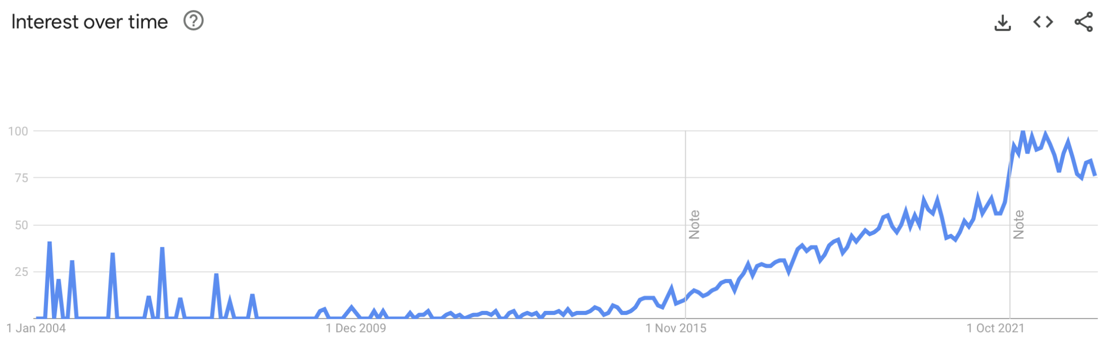

Figure 2.1: Google Trends "Infrastructure as Code": Starting in 2015, a significant surge in interest was observed, plateauing around 2021

the essence of cloud computing and highlights its transformative impact on the technology landscape. Cloud computing's ubiquitous nature makes it a pervasive force in modern computing. It allows for the provisioning of resources on an as-needed basis without extensive management or interaction with the service provider. Moreover, cloud computing commodifies computing resources, democratizing access and turning what was once a substantial capital investment into an operational expense.

The ability to quickly provision and release computing resources has revolutionized how businesses and organizations operate, making technology more accessible, flexible, and efficient. It has shifted the traditional paradigm of computing, where resources were scarce and expensive, to an age where computing power is as accessible and consumable as any other commodity.

Cloud computing is relevant and popular because it can transform traditional infrastructure into an agile environment. It enables entities to respond rapidly to changing demands, reduce operational overhead, foster innovation, and gain competitive advantage.

The five essential characteristics of cloud computing, as delineated by NIST, reflect the inherent advantages of this paradigm, offering a unique way to consume technology:

1. *On-Demand Self-Service:* Computing capabilities can be provisioned and managed without human intervention.
2. *Broad Network Access:* Resources are accessible over the network through various platforms.
3. *Resource Pooling:* Resources are dynamically pooled to serve multiple consumers.
4. *Rapid Elasticity:* Capabilities can be elastically provisioned or released to scale rapidly.
5. *Measured Service:* Resource usage is monitored, controlled, and reported, providing transparency.

Following these characteristics, cloud computing is typically categorized into three distinct service models. Each model delineates layers of service provisioning and offers a balance between reducing the complexity of managing individual components and providing access to and control over the underlying infrastructure layer:

- *Software as a Service (SaaS):* SaaS offers the highest level of abstraction, delivering applications over the Internet, thereby completely hiding the underlying infrastructure. Users can access applications through a web browser, and the cloud provider handles all data and computations.
- *Platform as a Service (PaaS):* PaaS provides an environment for developers to build, deploy, and manage applications without worrying about the underlying infrastructure. It abstracts hardware and operating system complexities, allowing developers to focus on business logic and application features. PaaS balances complexity and adaptability, offering preconfigured environments tailored to specific development needs. A particular form of PaaS, known as Function as a Service (FaaS), offers a high degree of abstraction, providing event-driven execution without the need to manage server resources.
- *Infrastructure as a Service (IaaS):* IaaS provides virtualized computing resources over the Internet, offering the lowest level of abstraction. This model provides the most flexibility and control, allowing clients to configure and manage virtual machines, containers, virtual networks, and more. Although it offers the best adaptability to specific needs, it also requires more management effort and expertise in handling the underlying infrastructure.

These service models form a continuum, with IaaS offering the most control and adaptability but at the cost of complexity, while SaaS abstracts all underlying complexities but limits specific adaptability. PaaS sits in the middle, balancing simplicity with control and adaptability. This layered architecture allows organizations to choose the model that best fits their needs, expertise, and requirements, enabling efficient utilization of cloud resources.

The adaptability of cloud computing extends to how it is deployed. The four deployment models provide flexibility in aligning cloud services with organizational goals and constraints:

- *Private Cloud:* Exclusive use by a single organization.
- *Community Cloud:* Shared by several organizations with common interests.
- *Public Cloud:* Available to the general public.
- *Hybrid Cloud:* A composition of two or more distinct cloud infrastructures, providing unique benefits.

These deployment models, combined with the essential characteristics and service models, make cloud computing a versatile and indispensable part of modern technology infrastructure.

After delineating the nature and impact of cloud computing, this section now shifts its focus to DevOps. The ensuing subsection articulates how DevOps, as a methodology, integrates with and complements cloud computing and IaC, further solidifying the nexus of these three domains.

2.1.3 DevOps

DevOps is a central paradigm in software development, facilitating the fast, flexible development and provisioning of business processes. Its inception integrates development, quality assurance, delivery, and operations, enabling automated testing, deployment, and infrastructure monitoring. Rather than relying on distributed siloed groups that perform functions separately, DevOps promotes cross-functional teams that work on continuous delivery of operational features, aspiring to deliver value more quickly and continuously. One of the cardinal virtues of DevOps is its potential to reduce problems due to miscommunication, thereby accelerating problem resolution [EGHS16].

CAMS

The CAMS model, which stands for Culture, Automation, Measurement, and Sharing, encapsulates the four fundamental dimensions[1] that enable DevOps [HF10]. These dimensions are integral to DevOps, but none can enable DevOps solely by itself.

Culture represents the first dimension of the CAMS model, highlighting the collaborative spirit between development, quality assurance, and operations [Roc13]. This aspect of DevOps presents four major challenges, which are detailed as follows [EGHS16]:

1. *Breaking Complex Architectures into Manageable Chunks:* The need to simplify complex systems into independent, deployable units is crucial. This decomposition promotes agility and flexibility in development and deployment processes.

2. *Maintaining Constant Visibility of Deployments:* Ensuring constant visibility of what is deployed, with accurate versions and dependencies, is vital for traceability and coordinated development.

3. *Introducing a Suitable Environment:* It involves creating a purpose-built development and production environment that evolves from a legacy application or product life-cycle management environment, promoting consistency and efficiency.

[1] It is worth noting that the literature is not unified in defining the fundamental dimensions of DevOps. An alternative perspective includes Collaboration, Automation, Measurement, and Monitoring as the four dimensions [LKO15].

4. *Bridging the Traditional Divide:* This challenge focuses on integrating traditionally siloed cultures of development and operations, which requires a shift in perspectives and collaboration mechanisms.

In this transformation, developers must assume a fullstack perspective, master a broad skill set, assume responsibility for testing and release environments, focus on test-driven development and Continuous Integration (CI), and ensure comprehensive automation of test cases and full code coverage [EGHS16].

Automation underscores the necessity of tools in achieving quality deliveries with short cycle time. Build tools facilitate fast iteration, managing various aspects of software development and the service life cycle, such as compiling code, managing dependencies, and deploying applications to different environments. Continuous Integration tools are instrumental in merging code from all developers and constantly testing the system, improving the software's overall quality. During deployment, treating infrastructure as code becomes paramount, enabling shared, testable, and version-controlled infrastructure, reducing problems arising from varying configurations [EGHS16].

Measurement requires tools to maintain the stability and performance of infrastructure. This includes comprehensive logging, tracing applications, and determining log levels such as fatal, error, warn, and debug. Monitoring is vital to identifying and resolving IT infrastructure problems before they affect critical business functions, continuously monitoring system health, and alerting administrators for corrective action.

Sharing encompasses the spread of tools, culture, ideas, problems, lessons learned, and data. An improved openness and transparency facilitate effective cooperation between development and operation teams, uniting the various aspects of DevOps into a cohesive whole.

In summary, DevOps transcends mere automation in software development and delivery. It integrates the dimensions of culture, automation, measurement, and sharing to form a comprehensive framework, profoundly impacting software development and organizational collaboration.

Optimizing for Change

DevOps is underpinned by a philosophy of optimizing for change. The overarching goal is not just about speed or quality in isolation; it is about enhancing both. By fully automating all phases of a Software Development Life Cycle, the effort needed for each change is dramatically reduced. This reduced effort allows even minor modifications to traverse the entire pipeline and be deployed independently swiftly. The more diminutive yet thoroughly and automatically tested a change is, the lesser the risk of it causing failures upon deployment. Furthermore, if a failure does occur, the rollback process becomes more manageable and affordable.

This dual objective of improving speed and quality is characterized by quick and frequent minor upgrades that strive for minimal cycle times. Automation is an indispensable component in this endeavor, further minimizing cycle times by reducing the effort required for each change. Adopting such a strategy can reduce cycle times to mere minutes [EGHS16].

The notion of optimizing for change is reflected in the four key metrics for DevOps success as delineated by Forsgren et al. [FHK18]:

1. *Cycle Time*: The period from the first commit to code deployment.
2. *Deployment Frequency*: The count of deployments within a specified time frame.
3. *Change Failure Rate (CFR)*: The fraction of deployments leading to failures in production.
4. *Mean Time to Recovery (MTTR)*: The average time required to restore service after a production failure.

These metrics are the yardstick for quantitatively assessing the speed and quality of changes delivered through DevOps.

In this thesis's purview, enhancing speed and stability through rigorous testing holds equal significance for IaC as for software development. Improving IaC test practices, a primary objective of this research, is pivotal in elevating the quality and speed of IaC delivery. This direct correlation reinforces the necessity and value of advanced testing methodologies in IaC, paralleling the principles established in DevOps.

The discussion on DevOps, especially its four fundamental dimensions in the CAMS model, effectively prepares the next section, which delves into the principles of IaC. Understanding DevOps helps to appreciate how IaC principles align with and enhance software development and operational processes.

2.1.4 Foundational Concepts of IaC

IaC extends beyond simple scripting of infrastructure; it is a methodology that applies the rigor of software development practices to infrastructure management [Mor20]. This approach is pivotal in enhancing system stability, scalability, resiliency, and security. By aligning closely with DevOps principles, IaC plays a crucial role in promoting automation, scalability, and reliability in the deployment and management of infrastructure.

Morris emphasizes core principles such as defining *everything* as code, continuous testing and delivery, and building systems from small, loosely coupled components [Mor20]. These principles lay the groundwork for a resilient and adaptable infrastructure. Building upon these, Wang introduces the RICE principles - Reproducibility, Idempotency, Composability, and Evolvability - as benchmarks of exemplary IaC configurations [Wan22]. These principles define the quality of IaC and highlight the integral role of testing in achieving and maintaining high standards.

Reproducibility entails using the same configuration to reproduce the same environment or infrastructure resources consistently, thus avoiding configuration drift. In the testing context, reproducibility is verified through automated tests, ensuring configurations are repeatable and reliable.

Idempotency refers to the capacity of a configuration to be executed multiple times without altering the end state or producing unintended side effects. IaC testing is crucial in identifying failures in idempotency, helping to detect defects that could lead to inconsistent infrastructure states.

Composability involves assembling any combination of infrastructure resources and updating each without affecting others. This principle dramatically enhances the testability of IaC, allowing for in-depth testing of individual parts to ensure the overall configuration functions as intended.

Composability becomes even more relevant when considering the lack of mocking tools for IaC testing.

Evolvability is the capacity to change infrastructure resources to scale with minimal effort or risk. Closely linked with regression testing, evolvability ensures that advancements in IaC do not compromise existing deployments, maintaining stability and security even as the infrastructure progresses.

In conclusion, adhering to these principles does not merely define exemplary IaC configurations; it also enhances their testability. IaC testing is not an afterthought but a fundamental aspect of developing robust, scalable, and secure infrastructure systems. Adhering to these principles ensures that IaC configurations are functionally sound, adept at accommodating changes, and scalable in the long term.

Having established the essential principles of IaC, the next step is to examine the tools and various aspects that operationalize these principles. The following section provides an in-depth look at the various tools and methodologies employed in IaC, offering a practical perspective on its implementation.

2.1.5 IaC Aspects and Tools

Building upon the foundational principles of IaC discussed earlier, exploring the diverse tools and technologies that bring these principles to life is essential. The IaC landscape comprises three main aspects: provisioning, configuration management, and image building. Each of these aspects, while distinct, complements the others, forming an integrated ecosystem for managing infrastructure as code [GGTP19, Wan22, KGR^{+}21].

Provisioning or Configuration Orchestration: This category focuses on setting up and managing the foundational infrastructure resources. Terraform and Pulumi are prime examples, enabling users to define and manage resources declaratively. Tools like Docker Swarm and Kubernetes also offer orchestration for containerized environments, highlighting the diversity within this space.

Configuration Management: This aspect involves automating the deployment and management of software on existing infrastructure. Ansible, Chef, or Puppet automate software configurations, ensuring that applications and platforms are consistently managed across various environments.

Image Building or **Infrastructure Templating:** In this aspect, tools like Packer are used to capture the current state of infrastructure resources, such as VMs or containers, and save them as standardized images. This process ensures that the environments can be precisely replicated, maintaining consistency across deployments.

The interplay among these tools illustrates a cohesive framework for IaC. For instance, Ansible may be utilized for initial configurations, Packer to capture these configurations as images, and Terraform to deploy instances using these images. This synergy is crucial in achieving efficient and reliable infrastructure management.

Each aspect represents a different facet of IaC, forming a supplementing system for efficient and effective infrastructure management. The thesis, however, narrows its focus primarily to the provisioning aspect, specifically exploring Terraform as a tool for cloud provisioning. Terraform exemplifies the IaC philosophy's ability to define, deploy, and manage cloud infrastructure coherently and predictably.

Exploring these tools, especially within the provisioning domain, sets the stage for a deeper investigation into IaC testing practices. To fully appreciate the nuances of IaC testing, it is essential to understand the context of traditional software testing. The following section delves into software testing methodologies. This examination is a backdrop against which IaC test practices can be compared and differentiated. Understanding the similarities and divergences between these testing paradigms will provide critical insights into the unique challenges and opportunities IaC testing presents.

2.2 Software Testing

Software testing plays a crucial role throughout the development process [ZMI08]. It acts as a quality assurance measure to confirm that the software aligns with its designated requirements and performs as expected. Traditionally, testing has been viewed as the last phase of a plan-driven or waterfall development process. This strategy involves distinct phases, such as requirement gathering, defining product specifications, code development, testing, and release, with formal entry and exit criteria between each phase. Due to its position at the end of the development cycle, testing often suffers from time constraints, potentially compromising software quality. In contrast, modern agile practices integrate testing throughout the development process, emphasizing the importance of test automation, verification, and validation [CG09]. This section explores the various aspects of software testing, including test automation, verification and validation, test levels, and specialized test approaches.

2.2.1 Test Automation and Regression Testing

In contrast to traditional methods, agile development practices integrate testing as an ongoing activity [CG09]. Test automation is pivotal in this context. It enables the rapid execution of tests and compares actual outcomes with expected results. This strategy uses specialized software to control various test activities, from input data generation over execution to pushing results to management software [HK07]. Automating these tasks significantly reduces the time and effort needed for testing, the number of defects found before production increases, and the need for manual regression testing is eliminated [Adz11].

Regression testing aims to ensure that new code changes do not adversely affect existing software functionalities. Automated tests are beneficial for this purpose, as they can be run frequently to verify that previously developed and tested software remains functional after modifications [KMB17].

It is helpful to consider exploratory testing briefly to appreciate the strengths of test automation and regression testing fully. For this method, test cases are not predefined. Instead, the tester dynamically designs, executes, and modifies tests based on their knowledge of the system under test (SUT) and the results of previous tests. Explorative testing is highly dependent on the skills and experience of the tester and requires a certain level of intuition and understanding of the SUT [IST23e]. This method is particularly effective for evaluating subjective, difficult-to-quantify quality metrics such as usability [Vir18].

With this context, it becomes clear that while test automation is highly efficient for repetitive tasks and regression testing ensures the stability of existing functionalities, these methods have limitations in areas where exploratory testing shines. Specifically, test automation and regression testing are less effective for evaluating intricate program logic or user experience. These are better assessed through a more nuanced and human-centric method like explorative testing.

Understanding the nuances of exploratory testing helps highlight the specific strengths of test automation and regression testing. Test automation is unparalleled in its efficiency in executing repetitive tasks, and regression testing is invaluable in maintaining the stability of existing functionalities. Although these methods may not be suited to evaluate intricate aspects of program logic or user experience, they offer their unique advantages. In particular, the speed, repeatability, and comprehensive coverage provided by test automation and regression testing make them indispensable tools in a balanced testing strategy, complementing the more human-centric insights gained from explorative testing.

2.2.2 Verification and Validation

Exploring the concepts of verification and validation helps discern the objectives that different test methods aim to achieve in software quality assurance. These principles guide the alignment of testing strategies with the specific goals of the development process, be it confirming that the software meets particular requirements or validating its overall utility and effectiveness.

Software verification involves assessing, supported by concrete evidence, whether the software has fulfilled its predefined requirements [DE05]. Verification answers the question, "Did we build the software correctly?" For example, a requirement might specify that the background color of a given button should be black. A corresponding test would confirm that the background color is indeed black. Automated tests are particularly well suited for verification tasks as they can objectively test specific, well-defined requirements.

Software validation, on the other hand, involves confirming by examination that a work product matches the needs of a stakeholder [DE05]. It answers the question, "Did we build the correct software?" A negative example could involve an exploratory test finding that the text color on a button with a black background is also black. While the requirement for a black background is met, usability is compromised when the text color matches the background. This highlights the importance of validation in assessing a software product's effectiveness and usability.

2.2.3 Test Pyramid and Test Levels

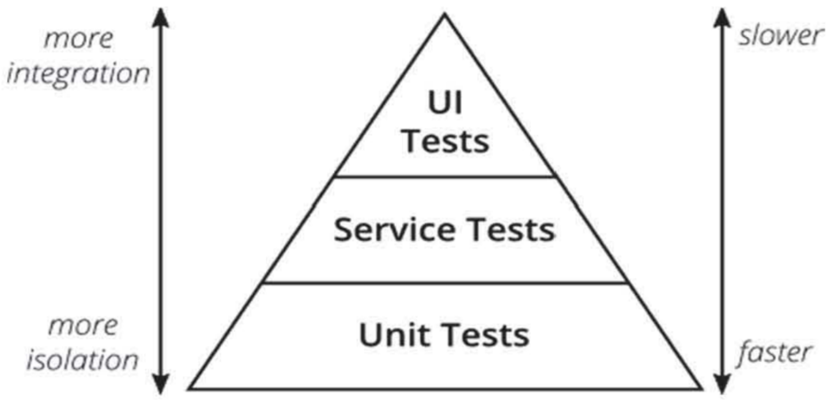

Figure 2.2: Mike Cohn's original Test Pyramid advocates for a broad base of low-level tests and successively smaller numbers of higher-level tests to optimize test efforts. ([MKB+21])

Mike Cohn introduced the Test Pyramid as a guideline for structuring testing efforts (see fig. 2.2). It advocates a broad base of fast, isolated low-level tests and a smaller number of slower, high-level tests to cover integration aspects. This structure aims to optimize testing efforts based on the cost relationships associated with different test levels [Coh09]. Since its inception in 2009, the Test Pyramid has been adapted and applied in various domains while retaining its core concept [CDM18, MKB+21].

Test levels serve as a framework for defining different layers of abstraction in the testing process. Each level is distinct in its technical complexity, has unique objectives, and therefore calls for specialized methods, tools, and skill sets [SLSS07]. These levels help manage time and resources between different phases of software development.

The utilization of this concept in section 3.4 adapts the pyramid to meet the specific needs and challenges inherent to IaC testing.

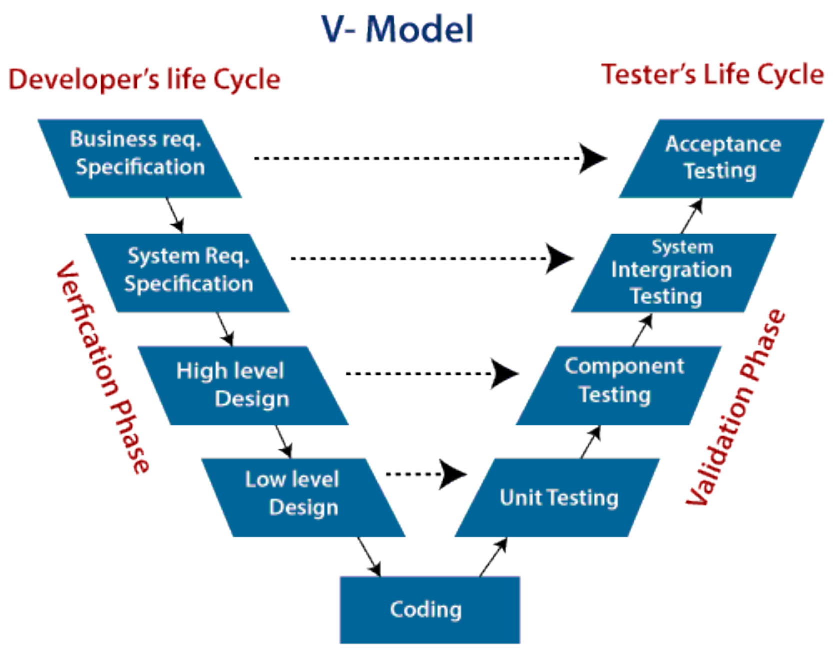

Figure 2.3: The V-Model delineates well-defined test stages corresponding to development phases, contrasting to the Test Pyramid, whose concept and test levels are comparatively less detailed. ([Jav])

Although the naming of individual levels differs, the concept of test levels or stages can also be found in the V-Model (see fig. 2.3), a process model for software development. Initially developed for contracts with the German military, the V-Model[2] is now an open-source process model used not only by German public institutions and contractors [CIO, BWHW06]. The development phases and matching test stages[3] of the V-Model, which has also been adopted by the curriculum of the International Software Testing Qualifications Board (ISTQB), making it foundational knowledge and well-known industrial practice [SLSS07].

One advantage of the V-Model is that its test stages are well-defined and documented, whereas the test levels in the original Test Pyramid are more roughly sketched [BWHW06, SLSS07, Coh09]. However, the V-Model lacks a visual representation for balancing test efforts. Therefore, the pyramid will continue to be used as a visual guide while leveraging the V-Model's well-defined test levels.

[2]The V-Model XT is a modernized and adaptable iteration of the original V-Model. For the purposes of this discussion, the differentiation between the two is not significant [CIO].

[3]The ISTQB Glossary sees the terms *test level* and *test stage* as interchangeable [IST23i]. To avoid confusion, the term *test level* will be consistently used henceforth.

In summary, understanding the Test Pyramid and the concept of test levels is essential for structuring effective test strategies. These principles are foundational in the curriculum of professional bodies like the ISTQB and find resonance in well-established models like the V-Model. Having established their significance, the forthcoming analysis will explore and delineate various test levels.

Unit Testing

Unit tests, also known as module, development, or component tests, focus on individual hardware or software components [IST23d]. The scope of what constitutes a component in this context can differ widely, ranging from individual functions and methods to classes, modules, or even complete microservices. The aim is to test the smallest unit that can be isolated and tested in a given context [IST23c]. The primary objective of unit tests is to demonstrate that the SUT meets the requirements outlined in the technical specifications [SLSS07]. This definition of unit tests aligns well with both the lowest level of Mike Cohn's Test Pyramid and the first test level in the V-Model [BWHW06, Coh09].

Unit tests are predominantly white-box tests [IST23j], which require intimate knowledge of the internal structure of the SUT [IST23h]. This specialized knowledge often leads to unit tests being written by the same developers who coded the SUT [Coh09].

In summary, unit tests are particularly well suited for verification, as they focus on confirming that the SUT meets the specified requirements. In addition, these tests are ideally suited for automation, allowing them to easily be incorporated into a regression test suite and become an integral part of continuous testing efforts.

Most of the characteristics of traditional unit testing remain relevant in the context of IaC. However, a crucial caveat is the shift towards static testing due to the increased costs and complexities associated with dynamic testing in IaC, compounded by a lack of sophisticated mocking tools.

The details of this IaC-specific approach to unit testing, particularly in the context of other IaC testing methodologies, will be discussed in section 3.3.

Integration Testing

Integration tests, also known as component integration tests or system integration tests, focus on the interactions between components or systems [IST23f]. These tests assume that the components subjected to them have already been individually tested and that potential defects have been corrected. The primary objective is to expose faults related to data exchange and function calls in the interfaces of, and interactions between, different components [SLSS07].

In the context of the V-Model, integration tests correspond closely to Mike Cohn's concept of Service Tests. While Cohn advocates testing the APIs of services, the V-Model expands this by testing various kinds of components and their interfaces, as well as the interactions between these interfaces [SLSS07, Coh09].

Integration tests are predominantly black-box tests, focusing on interfaces or APIs without knowledge of internal structures [IST23b]. This focus on external interactions differentiates them from unit tests, which require understanding the internal workings of the tested component and focus on isolated components.

In summary, integration tests ensure that different components or systems function cohesively. They serve a similar purpose to unit tests by being used to verify technical requirements. While automation is possible for these tests, it can be more intricate due to the involvement of multiple components. Despite this complexity, including automated integration tests in a regression test suite is beneficial. Doing so is valuable for the early detection of issues that may arise from diverging developments across different components, and they form an essential part of continuous testing efforts.

In the context of IaC, while the general concept of focusing on interactions between components remains unchanged, the nature of these tests evolves. Due to the static nature of IaC unit tests, integration tests in IaC become the first level of dynamic testing. This shift entails that IaC integration tests not only encompass traditional black-box test approaches but also extend to cover certain aspects of white-box testing. These aspects, which cannot be adequately addressed through static testing alone, include more in-depth verification of component interactions under dynamic conditions.

For more details on the specifics of IaC integration testing, see section 3.3.

System Testing

System tests, alternatively known as end-to-end tests or UI tests, focus on the system as a complete entity in an environment that closely resembles the production setting [IST23g]. These tests are conducted from a user's perspective, typically via the user interface rather than through APIs. The scope of system tests is comprehensive, covering the entire application or system, including its interactions with external interfaces and systems. The primary objective is to validate that the system as a whole meets the user-specified requirements and functions correctly within its intended environment. This characterization of system tests is consistent with the uppermost layer of Mike Cohn's Test Pyramid, referred to as UI tests, as well as the system test level in the V-Model [BWHW06, Coh09].

System tests are predominantly black-box tests, focusing on the system's overall behavior without requiring knowledge of the internal structures. This external focus differentiates them from both unit tests, which require understanding a component's internal workings, and integration tests, which focus on the interactions between components.

These tests require fully assembled components and an environment similar to the intended operational setting. While specialized tools for automated GUI testing exist, such tests are complex to assemble, prone to becoming quickly outdated, and challenging to maintain, making automated GUI testing expensive [NAF21].

In summary, system tests are essential for attesting that the entire application or system functions as intended from a user's perspective. They are particularly useful for validation, confirming that the system meets non-technical user-centric requirements. Due to the associated expenses, automated GUI tests should be minimal and very selective, focusing on requirements and test cases that cannot be adequately covered at earlier test levels.

In the context of IaC, the term "end-to-end" (E2E) testing is more prevalently used than "system testing." The fundamental focus on the system as a complete entity, without needing detailed knowledge of its internal workings and viewed from a user's perspective, remains integral in IaC. However, these tests are predominantly automated in IaC. E2E tests for infrastructure provisioning in IaC rely more on API-enabled test tools than the UI-centric methods in traditional system testing. This shift reflects the nature of IaC, where infrastructure components are managed and orchestrated through APIs rather than graphical interfaces. A more detailed discussion on the specifics of E2E testing in the IaC domain, particularly in contrast with other IaC test approaches, will be provided in section 3.3.

Acceptance Testing

Acceptance tests, also known as user acceptance tests, operational tests, field tests, or alpha and beta tests, focus on meeting the customer's needs and requirements. The primary objective is to establish that the customer accepts the product [IST23a]. Mike Cohn's Test Pyramid focuses primarily on the technical layers of testing and does not offer a corresponding level specifically for acceptance tests.

The critical differentiator between system and acceptance tests is not technical but organizational. While system tests are conducted from a user's perspective but still under internal control, acceptance tests are the customer's responsibility. Acceptance tests are often specified in the contract and serve as a criterion for product acceptance [SLSS07].

These tests are generally conducted in an environment controlled by the customer and should ideally be designed collaboratively. Although similar tests may have been part of the internal system tests, they are rerun during the acceptance testing phase to demonstrate the product's capabilities to the customer [SLSS07].

In summary, acceptance tests are essential for verifying that the system meets its contractual requirements and validating that the customer receives the expected product. Unlike the other test levels, which are more internally focused, acceptance tests provide external validation and are crucial for the customer's final acceptance of the system.

In the context of this thesis and its focus on the technical layers of IaC testing, the exploration does not include acceptance tests. Mirroring the emphasis of Mike Cohn's Test Pyramid on the technical facets of testing, acceptance testing, being primarily organizational and customer-focused, falls outside the primary scope of the technical examination of IaC test approaches.

2.3 Related Work

The field of IaC has been gaining popularity, necessitating focused research on quality assurance tailored to its specificities. While Rahman et al. [RMW19] have identified a general deficit in specialized studies within quality assurance (QA) for IaC, this lack of research is not uniformly distributed across all subdomains.

In the areas of anti-patterns and smell detection within IaC, substantial foundational work has been conducted, as evidenced by the volume of studies focusing on Configuration Management tools like Ansible, Chef, and Puppet. Akond Rahman, a pivotal researcher in this field, laid crucial groundwork in his doctoral thesis [Rah19]. Parallel to and following his thesis, Rahman and various co-authors expanded this research to encompass a broad spectrum of IaC quality assurance aspects. This includes studies on source code properties [RW19], characteristics of defective infrastructure [Rah18], and the identification of security-related issues [RPW19]. This research trajectory is complemented by the contributions of Bhuiyan et al. [BR20], Opdebeeck et al. [OZDR22, OZDR23], and Saavedra et al. [SF23], who further augment the understanding in this domain.

Initial efforts have been made to establish best practices for IaC, building on the foundational research in anti-patterns and smell detection. Rahman et al.'s work outline strategies for secret management [RBM21] and practices to enhance Puppet script quality [RS22], demonstrating the progression towards refined IaC best practices.

However, this research landscape presents a stark contrast regarding specialized areas such as individual test approaches, evaluating and quantifying test approaches, and strategies for balancing and optimizing multiple test approaches. These critical aspects of IaC QA are markedly under-researched. To effectively address these gaps, the methodology presented in chapter 3 focuses on assessing and comparing IaC test approaches within the context of a defect taxonomy. The following subsections will examine the state of academic research in defect categorization and test approaches for IaC, and review related industrial practices.

2.3.1 Defect Categorization

The concept of defect categorization is well-researched in software engineering. It has been extensively studied, and various frameworks have been proposed from both academia and industry [Wag08, Bei03, IEE10, Chi96]. These frameworks contribute to enhancing the quality of software by identifying, categorizing, and ultimately preventing defects.

Existing general categorizations for defects, while comprehensive, do not fully encompass the unique challenges presented by IaC, highlighting the need for a categorization framework specifically tailored to its nuances. Rahman et al. have made notable contributions in this direction by developing a taxonomy specifically geared towards IaC [RFPW20]. Their work focused on analyzing Puppet scripts, providing a customized categorization scheme that accommodates the distinct characteristics of IaC. Similarly, Hassan and Rahman extend the understanding of defect implications by researching the consequences of bugs in Ansible scripts [HR22].

In this thesis, the taxonomy of defects proposed by Rahman et al. serves as a fundamental framework for comparing and evaluating different test approaches for IaC Provisioning. The systematic categorization of defects provided in section 3.2 offers a structured reference for assessing test approaches. The capability of each test approach in detecting various defect categories is assessed in section 3.5, utilizing the defect taxonomy framework as the foundation. This assessment then informs and directs the subsequent selection and design of tests, as elaborated in section 4.3.

The suitability categorization in section 5.3 is the culmination of the pivotal role of defect categorization, particularly Rahman's IaC taxonomy, in shaping this thesis's methodology and analytical perspectives. This methodology emphasizes the indispensable value of such taxonomies in developing effective test strategies for IaC.

With the current state of research in IaC defect categorization established, the focus now turns to the academic perspectives on IaC test approaches.

2.3.2 Test Approaches

While software testing has seen extensive research and development, its adaptation to IaC presents unique challenges and opportunities, particularly in terms of practical application in the industry. This work predominantly centers on test approaches actively practiced in the industry instead of theoretical constructs or purely academic proposals. This focus aligns with the goal to furnish practitioners with practical, evidence-based insights, aiming to refine current test strategies for IaC provisioning.

Generalized meta-testing practices for IaC, as examined by Hassan et al. [HBR20], provide a broad spectrum of insights. While foundational, these practices, such as the Use of Automation, Sandbox Testing, and Testing Every IaC Change, lack the specificity required to translate directly into actionable test approaches for IaC. This observation underscores the importance of focusing on methods that are not only theoretically sound but also practically implementable, thus bridging the gap between academic research and industry application.

In a more focused perspective, Hummer et al. proposes a model-based test methodology for ensuring the idempotency of Chef scripts [HROE13b, HROE13a]. While this methodology exemplifies the detailed, applicable test approach sought in this research, it also illustrates a critical limitation in the current landscape of IaC testing.

Hummer et al.'s static approach offers cost benefits over resource-intensive dynamic test methods. However, despite its promise, this test approach has yet to develop practical implementation for the chosen IaC tool, Terraform, limiting its application in industrial contexts. As such, while it represents a promising direction, model-driven static testing remains predominantly a theoretical concept and is not included in this thesis's evaluation of applied IaC test methods. This exclusion underscores the focus of this work on test approaches that have already been translated into practical tools and practices within the industry.

Chiari et al. provide a broader survey of scientific methodologies to static analysis in IaC [CDPP22], differentiating between syntactic features and model-based behavioral analysis. However, their work does not link these academic findings to the industry-standard practices such as formatting and linting tools, nor does it consider other prevalent static analysis approaches like Policy as Code or static unit testing. This gap further emphasizes the need for a comprehensive overview that integrates academic and industrial perspectives, mainly focusing on feasible and practical test approaches for industry practitioners.

Caracciolo's thesis [Car23], which focuses on Policy as Code using Terraform in a CI/CD pipeline, provides a practical example of the type of test approach this thesis aims to explore. While it is specific to a single method, its close alignment with industry tooling and methodology makes it a relevant study of practical, applicable IaC test procedures.

Despite these foundational works, more research is needed that thoroughly explores practical and feasible test approaches for IaC as employed in the industry. This research gap, highlighted throughout the discussion of various works, is a precursor to the further exploration this thesis undertakes. It seeks to integrate these methods into comprehensive methodologies and evaluate them for test coverage and cost-effectiveness, thus contributing to best practices grounded in empirical data and real-world applicability.

2.3.3 Industrial Practices

The thesis expands to include practitioner sources to address the noticeable need for more research on industrial test approaches for IaC. This encompasses technical talks, blog posts, and publications by professionals and companies involved in IaC development. The insights gleaned from these sources provide valuable context to the limited academic research available, especially into test approaches that are both available and sensible for provisioning tools.

Terragrunt, the company responsible for the Terratest framework, offers a perspective on the hierarchy of test approaches for Terraform and other provisioning IaC tools in a technical talk at QCon [Bri19]. This talk identifies static analysis as the foundational layer in a Test Pyramid tailored to IaC. Additionally, the talk offers a comparative evaluation of various test approaches, focusing on metrics such as runtime, brittleness, and defect coverage.

Wang enriches the discourse on IaC testing with a detailed exploration of various test approaches, including the adaptation of traditional software test methods to IaC [Wan21, Wan22]. Notably, she diverges from conventional software test practices in her advocacy for a static unit test approach in IaC. This strategy, which Wang suggests, focuses on the verification of configuration files and dry-run outputs without necessitating the deployment of infrastructure. While this perspective is not entirely unprecedented, as evidenced by earlier suggestions like those of Nunez [Nun17], Wang's argument is particularly compelling. It has influenced the adoption of static unit testing in the discussion of IaC test approaches in section 3.3.

While these grey sources add nuance and practical insights, they also underline the importance of further academic inquiry to validate these industrial practices, especially concerning test coverage and cost-effectiveness, which are the primary foci of this thesis.

This chapter has established the core concepts for IaC and Software Testing. These foundational elements and the discussion of academic research and industrial practices in IaC form a comprehensive basis for the following methodology and analyses. The subsequent chapter will explore the theoretical framework for quantitative assessment in IaC testing.

3 Concept

The central objective of this thesis is to answer the research question: "How can we measure the suitability and efficiency of existing test approaches to help developers formulate tailored test strategies for IaC?"

The methodology employed to address this question is divided into four main steps.

1. Theoretical concepts related to IaC testing are extracted, refined, and extrapolated.

2. Quantifiable assessment criteria are determined, focusing on the efficiency dimension.

3. Theoretical considerations are applied and demonstrated in a practical Proof-of-Concept.

4. The results are evaluated using the previously defined quantitative assessment criteria.

This chapter, therefore, serves as the cornerstone for the first and second steps of the research methodology. Firstly, the terms for assessing the efficiency of IaC test methods are defined in section 3.1. This section serves as the groundwork for a quantitative cost assessment, elucidating the metrics and methods that will be utilized to evaluate the efficiency of different test approaches in subsequent chapters.

Complementing the efficiency exploration is the second methodological pillar, the suitability assessment of test approaches. The process begins with identifying relevant defect categories in section 3.2. The categories of defects are critical for assessing the *suitability* of existing test approaches, as they help to understand what types of errors or issues are commonly encountered in IaC. This forms the basis upon which tailored test strategies can be developed.

Subsequently, the research and consolidation of test approaches are undertaken in section 3.3. This involves dissecting existing methodologies to assess their capability in addressing the identified defect categories, further contributing to the suitability evaluation. Following this, the identified test approaches are organized within the framework of a Test Pyramid adapted for IaC, as detailed in section 3.4. It ranks test approaches hierarchically, organized by varying cost aspects to achieve an optimal balance between coverage (suitability) and cost-effectiveness (efficiency). This Test Pyramid will be applied and empirically validated in the subsequent chapter 4 Implementation and the outcome will be assessed in the chapter 5 Evaluation.

Finally, section 3.5 extrapolates on test coverage. This section explores which test approaches cover which defect categories, building the foundation for the practical assessment of test approach suitability.

By engaging in this dual exploration of suitability and efficiency, the current chapter lays a robust theoretical foundation for answering the research question. It paves the way for the following empirical steps: Applying these theories in a Proof-of-Concept and quantifying the observations, thus providing a comprehensive, quantifiable understanding of the costs and efficacy of IaC testing.

3.1 Quantitative Cost Assessment

In IaC testing, different cost factors influence the efficiency of TAs. These factors encompass both monetary and non-monetary aspects. However, a detailed analysis of every cost aspect is beyond the scope of this thesis. Initially, the discussion will briefly explore some significant but less quantifiable costs before focusing on the cost components relevant to the thesis.

Skills represent one such cost. Proficiency is required in handling specific testing tools and grasping the overarching concepts of quality assurance. While taking into account the commercial interests of certification bodies, the ISTQB Exams overview[1] highlights the breadth and diversity of the industry's methods in software testing can be, even before considering IaC or specific test tools.

Next, the cost associated with *test design, implementation, and maintenance* must be considered. Contrary to the common perception, test development is not a one-off effort. Each test case requires ongoing maintenance to adapt to changes in requirements or new understandings of existing requirements.

Given these complexities, this thesis deliberately narrows its focus to one of the more quantifiable aspects of IaC testing costs: *runtime costs*. We define runtime costs by two readily quantifiable components - *execution time* and *infrastructure deployment costs*.

Execution time is a particularly critical cost factor in IaC testing, more so than in traditional software testing. This significance arises due to the substantial overhead associated with the deployment and teardown phases in dynamic test approaches within IaC. As a result, the discrepancy in runtime between different test approaches, such as static and end-to-end (E2E) tests, can be markedly more significant. According to Terragrunt's estimates, static test approaches in IaC can have execution times ranging from mere seconds to a minute, while E2E tests may extend to several hours [Bri19].

This variance in execution time is particularly consequential in a DevOps environment, where minimizing cycle times and maintaining high deployment frequencies are imperative (section 2.1.3). Frequent code commits further amplify the importance of test execution time. This factor not only influences resource utilization and scalability but also determines the length of feedback cycles for developers. Consequently, it plays a pivotal role in determining how and where different TAs are integrated into the software delivery pipeline.

Infrastructure deployment costs in IaC testing can manifest as direct costs billed by cloud providers or opportunity costs for utilizing on-premise resources. While costs billed

[1]ISTQB Exam Overview: `https://i0.wp.com/www.istqb.com/wp-content/uploads/2022/12/ISTQB-Certification-Levels-2022.png`

by cloud providers are straightforward to quantify, opportunity costs associated with on-premise deployments are more challenging to standardize across different organizations. Due to this challenge, this thesis proposes using the costs billed by cloud providers as a proxy for both types of expenses.

Infrastructure deployment costs significantly differ between static and dynamic test approaches in IaC. Static tests, which do not require the actual deployment of infrastructure, avoid these costs entirely. In contrast, dynamic tests necessitate full-scale deployments, either incurring direct costs from cloud providers or opportunity costs through on-premise resource utilization. The absence of mocking tools in IaC, requiring complete system deployments for even minor dynamic tests, exacerbates this cost factor. Therefore, the distinction in infrastructure deployment costs is pivotal in evaluating the efficiency and cost-effectiveness of test approaches within the IaC paradigm.

The quantitative assessment of these costs is instrumental in understanding and comparing the efficiency of each TA. This focus on quantifiable metrics offers a robust methodology to compare and evaluate different test approaches for IaC.

The next step in the methodology is to examine the second pillar: suitability. To initiate this examination, a reference framework is established by identifying relevant defect categories for IaC. This framework will serve as the basis for assessing the suitability of test approaches.

3.2 Defect Categories

Defects, defined as flaws in a work item that prevent it from meeting its intended requirements, can lead to failures when executed [IEE10]. To assert if a test approach can detect these defects, it is crucial first to comprehend the diverse nature of defects [XZLZ13, CPZF19, HR22].

Different defects often necessitate varied detection methods. Thus, understanding the spectrum of defects is foundational to this thesis' methodology: it allows for determining which test approaches are best suited to identify specific defects [LBE17].

The classification of defects has been a topic of interest in software development for some time [Wag08]. Research from the scientific community [Bei03], guidelines from standardization bodies [IEE10], and methodologies from the industry [Chi96] have all contributed to this area.

When focusing on IaC, while many traditional defect categories (DC) are applicable, IaC introduces its nuances. Rahman et al. have distilled a taxonomy that's attuned to IaC by analyzing Puppet scripts, integrating known categories and introducing 'idempotency' as a category unique to IaC [RFPW20].

For easier reference, each defect category is assigned a unique symbol (DCx):

1. **DC1: Conditional** - Even though IaC tools, especially declarative tools, are limited in their expressive power compared to traditional programming languages, they still can handle basic logical constructs. This includes the likes of loops, conditionals, and constructed values. However, with this capability comes the

potential for defects. Erroneous logic or misinterpreted conditional values can lead to unintended configurations or behaviors in the provisioned infrastructure.

For instance, incorrect branching decisions can lead to undesired outcomes. A specific case highlighted by Rahman et al. [RFPW20] reports that in the Wikimedia Commons project cdh, a defective conditional statement caused a status request to output 0, regardless of the actual status consistently.

Moreover, a simple oversight, such as a typo causing a loop to run 1000 times instead of just 10, can have significant repercussions in IaC. Unlike a traditional software setting where this might slow down a program, in IaC, it could inadvertently provision or de-provision an extensive number of resources. Such a mistake strains the infrastructure and can inflate costs, especially when resources are provisioned on cloud platforms [Wan22]. In the case of Amazon, such a typo allegedly caused $150 Million in damage in 2017[2].

2. **DC2: Configuration Data** - Configuration data defects arise when static values specified within an IaC script are erroneous. These defects can lead to system failures even when paired with correct logic and appropriate tools. In a study by Hassan et al. [HR22], configuration defect-related bugs were identified as the most frequent bug category in Ansible scripts.

   The repercussions of defective configuration data can be severe. For instance, an outage in 2014 at Stack Exchange was caused by misconfigured iptables[3]. In another significant incident, the 2017 Google Compute Engine incident #17007 was attributed to an outdated load-balancer configuration[4].

3. **DC3: Dependency** - Infrastructure dependencies arise when one resource relies on the existence and specific attributes of another resource or artifact [Wan22]. A defect in this context emerges when the artifact that is depended upon is either absent or inaccurately specified. Such a dependency can take various forms, be it another resource, a file, a class, a package, a Puppet manifest, an Ansible role, or a Terraform module.

   Even with the support of sophisticated tools, managing intricate dependencies can become challenging, even likened to a nightmare [Har]. Hassan et al. [HR22], identified dependency defect-related bugs as the second most frequent bug category in Ansible scripts. Together with configuration defect-related bugs, these two categories have a significantly higher frequency than any other category.

4. **DC4: Documentation** - Accurate documentation is pivotal for understanding, maintaining, and evolving IaC scripts. However, defects can arise when informa-

[2]Amazon typo: `https://www.npr.org/sections/thetwo-way/2017/03/03/518322734/amazon-and-the-150-million-typo`

[3]Stack Exchange outage: `https://stackstatus.tumblr.com/post/96025967369/outage-post-mortem-august-25th-2014`

[4]GCE incident: `https://status.cloud.google.com/incident/compute/17007#5659118702428160`

tion about IaC scripts in source code comments, maintenance notes, or documentation files like READMEs is incorrect or misleading. Such inaccuracies can lead to misunderstandings, misconfigurations, or even failures when other developers or operators rely on this documentation for their tasks. Ensuring that the documentation aligns with the actual behavior and intent of the IaC scripts is crucial for the infrastructure's overall reliability and maintainability. For instance, an incompletely documented module can result in its incorrect usage by other teams, losing the value of making it available in the first place [Wan22].

5. **DC5: Idempotency** - Idempotency, as described by Wang [Wan22], is an enhanced form of repeatability. It demands consistent results upon reruns and ensures that the defined end state remains unchanged. In the context of IaC, idempotency guarantees that a script will only modify resources if they deviate from the desired end state. For instance, executing an IaC script to configure a database for the first time should establish a running instance. Subsequent executions of the same script should neither instantiate a second database nor delete and recreate the original one. The existing database instance should remain untouched if it aligns with its IaC definition.

   However, when idempotency is not maintained, it can lead to many issues. Defects related to idempotency can result in challenges like file backup errors, complications in package updates, database setup issues, logging discrepancies, and network configuration problems.

6. **DC6: Security** - Security defects violate the CIA: a system's confidentiality, integrity, or availability [SC]. In the context of IaC, specific anti-patterns can jeopardize these principles.

   For instance, not isolating secrets from code, leaving data unprotected at rest, or neglecting standard secure coding practices can introduce vulnerabilities [KGR$^+$21].

   Examples of these standard secure coding practices include ensuring that credentials used in automation adhere strictly to the principle of least privilege, using IaC tool mechanisms to prevent sensitive data like passwords from appearing in logs, and leveraging secret managers to revoke and rotate passwords post-deployment [Wan22].

7. **DC7: Service** - Service defects pertain to the improper provisioning and inadequate availability of computing services. Unlike defects that arise from incorrect configurations, service defects stem from incorrect assumptions about what configuration would be necessary.

   Drawing a parallel to the distinction between verification and validation, as discussed in section 2.2.2, while the infrastructure might be correctly set up as per the IaC script (verification), it might not meet the actual needs or demands of the system or its users (validation).

For instance, an underprovisioned service might technically be set up correctly, but it could fail to handle customer requests adequately due to its limited capacity.

8. **DC8: Syntax** - Syntax defects violate a language's prescribed writing style or structure.

   In traditional software development, a classic example is the omission of a semicolon at the end of a line in languages like Java or those based on C. Other syntax defects include typographical errors in language keywords, mismatched or missing brackets, inappropriate variable types for a given context, or, in the case of languages like YAML, flawed formatting.

   Such defects can lead to various failures: they might halt execution entirely, result in erroneous execution, or, more insidious, alter the intended meaning of a segment of code.

In this section, defects pertinent to IaC have been systematically categorized. Drawing from traditional software development and specific characteristics of IaC, Rahman et al.'s taxonomy of defect categories has been delineated [RFPW20]. Informed by academic research and industry practices, this taxonomy serves as a foundational reference for evaluating test approaches.

After outlining the various defect categories, the next step is identifying and examining existing test approaches. Once a comprehensive understanding of these methods is achieved, the evaluation will focus on how effectively each test approach can detect and address the previously discussed defect categories.

3.3 Test Approaches

Up to this point, the discussion has encompassed IaC in a broad sense, exploring its foundational concepts, benefits, and associated challenges. However, it is imperative to understand that the landscape of IaC is vast, and the test approaches and the tools utilized for such tests can differ significantly depending on the specific aspects of IaC under consideration and the IaC tool in use. This variance is attributed to the unique features, constructs, and paradigms each tool introduces, which can influence the test strategies and the tooling ecosystem surrounding it.

Thus, while the test concepts presented apply to IaC tools focusing on configuration orchestration or provisioning in general [Bri19], the associated tools might be specific to Terraform. Given the prominence and widespread adoption of Terraform[5], and its declarative focus on the provisioning aspect of IaC, it presents a suitable candidate for an in-depth exploration.

To our knowledge, no prior scientific study delves into the spectrum of IaC test approaches in general or Terraform test approaches in particular. Consequently, reference sources for this research will include non-peer-reviewed materials such as practitioner

[5]Terraform: `https://www.terraform.io/`

books, independent as well as tool-specific blogs, and a recorded tech talk in video format. Additionally, scientific sources relevant to specific approaches or aspects will be integrated when they provide specific insights.

The exploration of these sources revealed several noteworthy observations. First, there is a noticeable inconsistency in terminologies. For example, while some sources [Wan21, Wan22] use the term *unit testing* in the context of static analysis techniques, others [Bri19, Mod21b] employ it for dynamic testing of Terraform resources or modules.

Second, the perception of the role of *static analysis* varies across the board. Some sources [Bri19] view it as a foundational element in an IaC Test Pyramid. In contrast, others [Mod21a] integrate static analysis tools within CI/CD processes without clearly defining them as distinct test approaches.

Lastly, it is observed that *static analysis* techniques are diverse. They span from basic formatting and linting to programmatic verification of code files or dry-run outputs to Policy as Code (PaC). Despite this variety, no clear demarcation separates these diverse methods into distinct test approaches or levels.

Building on these initial insights, this work aims to reduce the confusion in the IaC test domain. To accomplish this objective, a specific naming schema is proposed to rectify the inconsistencies in terminology. Furthermore, *static analysis* techniques are systematically categorized into discrete test approaches, focusing on differentiating their distinctive characteristics and resource requirements. Additionally, test tools relevant to each approach are highlighted, with the understanding that the aim is to provide an illustrative selection rather than an exhaustive compilation[6].

For easier reference, each test approach is assigned a unique symbol (TAx):

1. **Static Analysis** - These methods inspect code files or execute IaC code to generate a plan of anticipated infrastructure changes without actual deployment (dry-run) and then run tests against the static output [Bri19, Wan21, Wan22].
    - **TA1: Formatting** - Ensures consistent code alignment and formatting. Especially vital when indentation determines scope (e.g., in YAML) [Bri19, Wan21, Wan22]. Notable tool: terraform fmt[7].
    - **TA2: Linting** - Verifies syntax and spots basic issues, like incorrect attribute names [Bri19, Wan21, Wan22]. Tool example: terraform validate[8].
    - **TA3: PaC** - Policies in IT operations define rule sets for resource utilization [Sea22]. The PaC approach codifies these rule sets, similar to how IaC treats infrastructure. This codification facilitates managing policies akin to software, thus enabling practical advantages like version control for codified policies. Importantly, codification enables automated compliance checks [Car23]. The origins of PaC are deeply rooted in DevSecOps, which emphasizes the infusion of security best practices into the broader DevOps methodology.

[6]For a more expansive overview of test tools, refer to `https://github.com/joatmon08/tdd-infrastructure`

[7]terraform fmt: `https://developer.hashicorp.com/terraform/cli/commands/fmt`

[8]terraform validate: `https://developer.hashicorp.com/terraform/cli/commands/validate`

At their core, linting tools can be perceived as basic forms of PaC with rudimentary hard-coded policies. However, dedicated PaC tools not only come with a more extensive set of built-in policies, often centered around security best practices, but also provide the pivotal feature of allowing custom policy additions. This capability for customization is instrumental for organizations aiming to ensure adherence to internal guidelines, industry best practices, or legal regulations [Car23].

Examples of PaC tools include HashiCorp Sentinel[9], tfsec[10], and Regula[11].

- **TA4: Unit Testing** - Traditional software unit testing emphasizes assessing the inner structure or workings of the SUT (see section 2.2.3). Declarative IaC tools aim to describe the desired infrastructure state rather than imperatively define the deployment steps, so the emphasis is not on the inner workings. Instead, the focus shifts towards ensuring the correctness of configuration values, which can be verified without necessitating a live infrastructure deployment [Wan22].

  Using programming languages to parse plans and make assertions provides developers with the flexibility to evaluate various facets of their IaC configurations. This encompasses verifying the number of resources or attributes generated by constructs like `for_each` or `count`, checking values produced by `for` expressions, and verifying the outputs of built-in functions. Moreover, unit testing ensures that dependencies between modules are resolvable, verifies the existence and readability of interpolated values, and checks that variables containing sensitive information, such as passwords, are appropriately flagged as sensitive [Wan21].

  Developers verify and assert particular facets of specific IaC configurations through unit tests, unlike PaC, which is used to verify compliance with generalized policies.

  A variety of test tools can effectively support static unit tests. Due to the nature of static unit tests, which primarily involve parsing text in configuration files or dry-run outputs, any general-purpose programming language (GPL) is adequate. For instance, Python, when used with `pytest`[12], offers a flexible and powerful environment for implementing unit tests.

  In addition, Terraform has recently introduced a feature for build-in test capabilities: the `terraform test` command[13]. This new functionality allows for defining static and dynamic tests using HCL, the same DSL used for defining Terraform configurations.

[9]HashiCorp Sentinel: `https://docs.hashicorp.com/sentinel`
[10]tfsec: `https://github.com/aquasecurity/tfsec`
[11]Regula: `https://regula.dev`
[12]pytest: `https://docs.pytest.org/en/7.4.x/`
[13]Terraform 1.6.0, featuring support for `terraform test`, was released on October 4, 2023. See the release notes at `https://github.com/hashicorp/terraform/blob/v1.6/CHANGELOG.md`.

2. **Dynamic Testing** - These test approaches deploy infrastructure to ensure reliability and functionality. Terratest[14] is a known tool in this domain. Within dynamic testing, there are:
   - **TA5: Integration Testing** - Similar to software development practices (see section 2.2.3, Integration Testing in the realm of IaC focuses on assessing how various "units" collaborate. This entails verifying whether multiple resources or modules interoperate as anticipated [Bri19].

     Integration Testing is the initial dynamic test approach, verifying that the IaC configuration effectively deploys the expected resources. While (static) unit testing verifies the correctness of interpolated and generated values, integration testing, being the inaugural dynamic test approach, ensures that the deployed resources interact as expected [Wan22].
   - **TA6: E2E Testing** - End-to-end testing evaluates the system's ability to function successfully from a user's perspective. It assesses the complete infrastructure, including networks, compute clusters, load balancers, and other integral components, to ensure they operate in concert as intended [Bri19, Wan22].

     However, it is essential to note that E2E tests are the most resource-intensive and time-consuming test approaches in IaC. These tests can take anywhere from 60 to 240 minutes to complete, rendering them often impractical for frequent use. Moreover, they are more susceptible to failure due to the sheer number of resources involved. For instance, if a single resource has a failure rate of 0.1%, this compounds with the scale of the infrastructure being tested. In comparison, unit tests, which operate on a smaller scale with fewer resources, have a lower risk of failure [Bri19].

     Additionally, the extensive time required for E2E testing and the higher likelihood of failure make retries for these tests a prolonged endeavor. Thus, while E2E testing is indispensable for evaluating critical business functionality, its limitations in terms of time, resources, and brittleness necessitate careful consideration when implementing this test approach in IaC workflows [Bri19].
3. **Other Techniques:**
   - **CRUD Testing** - Most IaC tools, including Terraform[15], incorporate built-in mechanisms to ensure resources create, (read,) update, and delete (CRUD)[16] as intended [Wan22]. To sidestep any confusion with "Customer Acceptance Tests" elaborated in section 2.2.3, this thesis abstains from employing the label "acceptance testing" for the CRUD testing process.

     Given their automatic nature and the absence of user input requirements, CRUD testing will not be discussed further in the subsequent thesis. It is

[14] Terratest: `https://terratest.gruntwork.io`
[15] Terraform Acceptance Tests: `https://developer.hashicorp.com/terraform/plugin/sdkv2/testing/acceptance-tests?product_intent=terraform`
[16] CRUD: `https://codebots.com/crud/how-to-test-CRUD`

mentioned here to clarify why the upcoming test approaches do not cover CRUD aspects.

- **Contract Testing** - As elucidated by Wang [Wan21, Wan22], contract tests are centered on modules. These tests harness Terraform's built-in mechanisms to authenticate variable values, employ static analysis methods to oversee dependencies and verify inputs and outputs. Furthermore, they resort to dynamic testing, invoking API calls to corroborate responses.

  While we recognize the merit of applying techniques emphasizing module testing rather than a broad IaC testing scope, we contend that it does not qualify as a distinct test approach in its own right. This assertion stems from contract testing's lack of a unifying mechanism. Instead, it repurposes a mix of techniques inherent to established test approaches.

- **Model-Driven Static Testing** Model-driven static testing in IaC involves simulating IaC script behavior to detect defects without actual deployment [CDPP22]. An example is Hummer et al.'s testing for idempotency in Chef scripts [HROE13b]. This method offers cost benefits over resource-intensive dynamic testing. However, this technique has yet to yield practical tools for Terraform, limiting its application in industrial contexts. Thus, while promising, model-driven static testing remains a theoretical concept, not included in this thesis's evaluation of applied IaC testing methods.

In conclusion, exploring various test approaches in IaC has revealed a spectrum of methodologies, each with distinct characteristics and applicability. Recognizing the diversity and complexity of these methods, it becomes essential to structure them in a manner that optimizes their usage.

The following section introduces the IaC Test Pyramid, an adapted model that categorizes these test approaches hierarchically. This model balances coverage and cost-effectiveness by organizing the methods according to their cost implications and test scope. It serves as a strategic guide to employing these test approaches efficiently, ensuring a balance between thoroughness in testing and pragmatic resource allocation.

3.4 IaC Test Pyramid

Drawing from the Test Pyramid concept (see section 2.2.3), this section offers an adaptation tailored to IaC testing. IaC test approaches are categorized into several test levels, emphasizing their differences in technical complexity, objectives, methods, tools, and requisite skills [SLSS07].

An adapted Test Pyramid for IaC is proposed (see fig. 3.1). This structure arranges test levels and corresponding test approaches in line with cost relationships [Coh09], striving to promote balanced test strategies.

The model emphasizes a substantial base of low-level tests consistent with the original Test Pyramid. Progressing up the pyramid, the quantity of tests diminishes for each successive level. While runtime costs across static test levels show negligible variance, the primary cost differentiation emerges from the effort required for test development and maintenance. In contrast, pronounced disparities in runtime costs are observed when comparing static to dynamic test levels and between different dynamic test tiers.

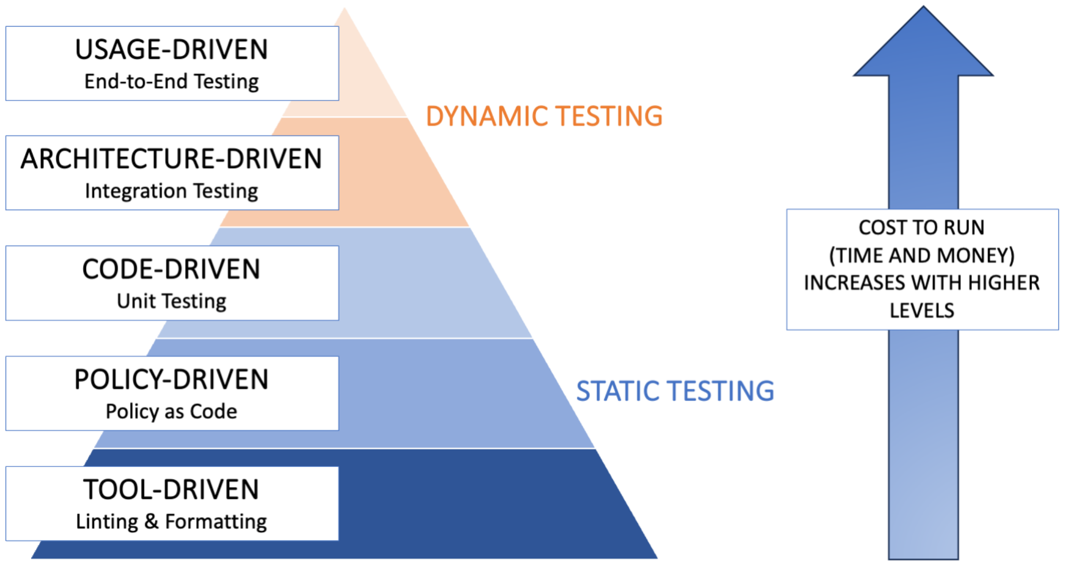

Figure 3.1: IaC Test Pyramid: Adapted for IaC, our model emphasizes a strategic shift. A spectrum of static test approaches forms the base. Traditionally dynamic unit testing is now a static layer. The peak consists of dynamic tests, used selectively due to their higher costs.

Outlined below are the proposed test levels (TLx):

- **TL1: Tool-Driven Static Testing** - Test approaches governed by specific tool categories, such as Formatting (TA1) and Linting (TA2), require minimal developer input due to their limited scope for configuration and lack of support for custom test cases. Given their favorable cost-to-benefit ratio, they should invariably be incorporated into every IaC test pipeline.
- **TL2: Policy-Driven Static Testing** - Policies (TA3) are generalized and transcend individual projects. Consequently, the effort to codify and sustain them can be distributed across multiple projects, diminishing a particular project's portion of the overall PaC expenditure.
- **TL3: Code-Driven Static Testing** - These unit tests are intrinsically driven by the specific IaC configuration - the *code* - unique to each project. The inner workings of the code - compare the discussion of white-box tests in section 2.2.3 - determine each test's necessity and design.

  While generic tests, such as "always verifying that the correct number of subnets will be created," might apply broadly, the project's code determines the particular implementation for each test. The tests' project-specific nature necessitates customized test development and maintenance, which results in higher effort for both compared to Policy-Driven Static Testing (TL2).

To reduce resource usage, universally applicable tests should be formalized into standardized policies (TL2), avoiding duplicated costs across projects. In contrast, project-unique tests must be managed as unit tests (TA4) despite the elevated development and maintenance requirements.

- **TL4: Architecture-Driven Dynamic Testing** - The architectural nuances of an IaC deployment, in terms of how live resources interact and how the underlying configuration is modularized, inform integration tests (TA5). Being dynamic, integration tests mandate a live infrastructure deployment to confirm coordination among distinct resources. With the prerequisite of deploying actual infrastructure, integration tests command a significantly extended runtime compared to any static testing method. Furthermore, they either lead to cloud provider charges or necessitate resource allocations in a private cloud.

- **TL5: Usage-Driven Dynamic Testing** - This test level concentrates on E2E tests (TA6) that evaluate the entire IaC deployment as a cohesive system within a live environment. These tests are distinctively guided by business-critical functionalities instead of technical requirements or modular interactions. They require complex infrastructure setups, typically in the production environment or a specialized test environment closely resembling the production setup.

  The tests in this category are designed to simulate real-world scenarios, thereby validating the behavior and performance of the deployed infrastructure from an end-user perspective. Due to the comprehensive nature of these tests, they come with higher costs in terms of time, computational resources, and possibly cloud provisioning expenses. Consequently, as the apex of the IaC Test Pyramid, these tests should be employed judiciously and selectively.

  Given their cost implications, end-to-end tests are most effectively reserved for validating business-critical use cases that less expensive testing strategies cannot adequately cover. The aim is to maximize the return on investment by focusing on tests that are indispensable for ensuring the operational integrity of the IaC deployment.

In conclusion, the IaC Test Pyramid provides a structured framework for categorizing test approaches based on their complexity, costs, and objectives. With this comprehensive understanding of test approaches suitable for IaC provisioning in place, the focus shifts to assessing test coverage by utilizing the previously identified defect categories as a reference framework.

3.5 Test Coverage

Evaluating the suitability of the identified test approaches is crucial for effective IaC testing and an essential element of this research. The initial focus is assessing coverage as a binary metric to determine if a test approach can detect defects in a given category. This foundational assessment, articulated through the Test Coverage Matrix (table 3.1), categorizes TAs based on their ability or inability to cover specific DCs. This binary method lays the groundwork for subsequent, more comprehensive analyses, including test case implementation and runtime cost evaluation later in this thesis.

	TA1 Format	TA2 Lint	TA3 PaC	TA4 Unit	TA5 Integration	TA6 E2E
DC1 Conditional				○	○	
DC2 Configuration Data				○	○	
DC3 Dependency				○	○	○
DC4 Documentation				○		
DC5 Idempotency					○	○
DC6 Security			○	○	○	○
DC7 Service						○
DC8 Syntax	○	○				

Table 3.1: Test Coverage Matrix: Binary assessment of TAs' ability to detect defects across various DCs, providing a foundation for further suitability analysis.

The test coverage data used to populate the matrix is derived from existing literature and our informed understanding of how each TA functions. While data sourced from literature lends credibility, it is essential to acknowledge certain limitations.

Firstly, this work includes our interpretive judgments and data derived from the existing but limited literature, which could introduce biases or errors into the matrix. Such biases could arise from only partially understanding specific TAs or DCs or preconceived notions regarding their effectiveness. Secondly, given the nature of this work as a master thesis, it is not an ongoing study and will not be updated with future empirical data.

In the absence of empirical research specifically focusing on the suitability of TAs for IaC, these biases or errors could manifest in the following ways:

1. **Overestimation or Underestimation**: The matrix might depict certain TAs as more or less effective than they are, affecting the integrity of potential test strategies.

2. **Misrepresentation of Suitability**: The perceived suitability of a TA for a specific DC may not necessarily align with real-world applications, leading to skewed test strategy recommendations.

3. **Omission of Relevant TAs**: Our initial assumptions could result in the exclusion of TAs that are potentially highly suitable for certain DCs.

Given these potential limitations, this work should be considered a preliminary step. The following detailed discussion on the coverage of individual DCs by TAs aims to offer an in-depth account of our rationale, with the intention that it will stimulate further research in this area rather than serving as a definitive guide:

- **DC1**: One of TA4's designated functions is to verify the logic employed in IaC scripts [Wan22, Chapter 6.2.3]. TA5, while not directly examining code logic, can indirectly confirm its correctness by assessing the deployed infrastructure.

  TA6 is not intended to verify configuration but end-user functionality [Wan22, Chapter 6.5]. In contrast, TA1 and TA2 are not designed to execute test cases, so they cannot verify specific code conditions. TA3, utilizing generalized policies, is also unsuitable for specific code verification.

- **DC2**: Echoing the findings for DC1, TA4 is designated to verify IaC configurations [Wan22, Chapter 6.2.3], and TA5 continues to provide indirect verification by evaluating the resulting infrastructure.

  TA1, TA2, TA3, and TA6 remain unsuitable for addressing specific configuration deficiencies.

- **DC3**: TA6 is the only test approach deploying the complete, interdependent infrastructure, potentially even in a production environment [Bri19, Wan22]. Some DC3 defects can only be discovered with TA6, such as those related to external providers that cannot be integrated into a test environment.

  TA5, being the first dynamic TA, not only verifies module interaction but also validates the intended interaction of resources. TA4 is designed to verify static dependencies, including the availability and versions of referenced modules [Wan22].

  TA1, TA2, and TA3 are not intended to identify defects falling under DC3.

- **DC4**: Test automation offers only limited avenues for effectively addressing DC4-related defects. As a modest example, TA4 can be aligned with the agile practice of using code as documentation to communicate essential or standard configurations to other teams [Wan22, Chapter 6.2].

- **DC5**: Idempotency is fundamentally tied to the dynamic deployment of infrastructure. Although TA4 can verify the underlying logic as per DC1, only the dynamic TAs - specifically TA5 and TA6 - can detect idempotency issues.

- **DC6**: TA3 is designed to verify that infrastructure metadata complies with general security requirements [Wan22, Car23]. TA4 allows test cases to be specifically tailored to address security concerns unique to a particular project. In addition, there might be security defects only detectable through dynamic testing using TA5 and TA6.

- **DC7**: Defects in this category do not arise from incorrect configurations but from incorrect assumptions about the required resources. Therefore, end-to-end test cases validating customer needs, primarily achievable through TA6, are essential for detecting such flawed assumptions. The other discussed TAs are not suited to end-to-end validation.

- **DC8**: TA1 and TA2's fundamental purpose is the detection of syntax defects. To achieve comprehensive coverage of syntax defects, the combined capabilities of both TA1 and TA2 are necessary.

  Other TAs are not intended to identify syntax defects

The preceding sections have systematically developed the foundational concepts that inform the proposal for an IaC Test Pyramid. These foundational elements encompass a detailed taxonomy of DCs, a clarified nomenclature and categorization for TAs, and a methodology for assessing the cost implications of each TA. Moreover, the Test Coverage Matrix explicitly maps TAs to DCs, thereby enabling an assessment of the suitability of different TAs in detecting specific defects.

These conceptual building blocks underpin the proposed IaC Test Pyramid, a model that guides test planning and resource allocation. The pyramid aims to provide a strategically balanced model for achieving adequate test coverage by aligning test levels with their corresponding cost implications. The pyramid integrates these disparate elements into a cohesive concept as a culminating point of the theoretical framework.

The logical next step in this research is to subject the theoretical framework and the associated hypotheses to empirical scrutiny. The subsequent chapter will feature a PoC implementation to validate the proposed IaC Test Pyramid's practical applicability and efficacy. Applying the framework to a real-world IaC environment is intended to garner empirical insights that either corroborate or challenge the theoretical construct.

4 Implementation

The previous chapters have systematically developed a theoretical framework for testing Infrastructure as Code. This framework, culminating in conceptualizing an IaC Test Pyramid (see fig. 3.1) and the Test Coverage Matrix (see table 3.1), has outlined a methodology for evaluating the suitability and efficiency of various Test Approaches against the framework of Defect Categories. The forthcoming chapter transitions from theoretical exploration to empirical application. It introduces a Proof-of-Concept (see fig. 4.1) to validate the practical effectiveness of the proposed IaC Test Pyramid.

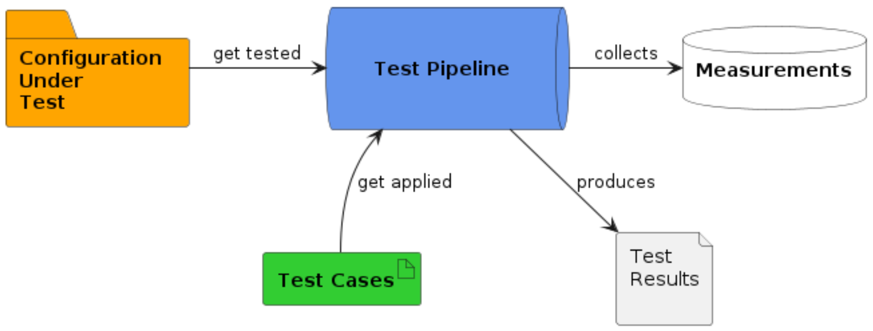

Figure 4.1: The PoC structure features the Test Pipeline (blue), applicable to any Terraform project, alongside an exemplary project's CUT (orange) and Test Cases (green). The pipeline executes the Test Cases against the CUT, producing Test Results (grey) and aggregating Measurements (white) across all runs, providing data to assess the efficacy of test approaches.

Central to this empirical investigation are three primary components[1],[2]:

1. **Test Pipeline**: The Test Pipeline, implemented as a Jenkins declarative pipeline, is the operational core of the PoC. Each TA of the proposed IaC Test Pyramid (fig. 3.1) is represented by one or more dedicated stages, thereby creating a comprehensive and automated test environment. The pipeline not only facilitates the execution of TCs but also standardizes the test process, ensuring consistency and repeatability.

[1]CUT, test implementations, Test Pipeline definition, and supporting scripts for measurements and cost calculations are available at `https://github.com/fex01/thesis-tf`.

[2]Tooling and setup instructions for the Test Pipeline, development environment, and data collection process are available at `https://github.com/fex01/thesis-ws`.

More importantly, it serves as an instrument for data collection, capturing vital metrics such as runtime and cost estimations. This information is pivotal for evaluating the efficiency of each TA, allowing for a quantitative assessment of time and cost implications, which are central to the thesis's objectives.

2. **Configuration Under Test**: Adapted from Caracciolo's work on Policy as Code [Car23], the CUT is a Terraform configuration deployed on Amazon Web Services. This configuration has been carefully chosen to mirror a typical, real-world IaC environment, setting the stage for an in-depth exploration and assessment of various test approaches.

   The CUT serves as a foundational backdrop and a contextually rich testbed, allowing for the practical application and evaluation of TAs. It encompasses a diverse array of standard infrastructure components, such as Virtual Private Clouds, Elastic Kubernetes Service clusters, and Relational Database Services. These components provide a comprehensive environment to assess the suitability of different TAs in identifying and addressing various DCs.

   In this capacity, the CUT plays a supportive role in the thesis, enabling the focused examination of the application and efficacy of TAs, which are realized through the intricacies of the Test Pipeline and the Test Cases at the heart of the research.

3. **Test Cases**: The TCs, developed to align with and validate the Test Coverage Matrix, serve as crucial experimental variables of the PoC. Each test is designed to empirically validate the suitability of various Test Approaches for specific Defect Categories as outlined in the matrix.

   This strategy ensures that TCs not only test the intersection of TAs and DCs but also provide a direct mechanism for confirming the accuracy of the Test Coverage Matrix in predicting TA effectiveness. By implementing similar TCs across different TAs, the thesis enables an anecdotal examination and a direct comparative analysis of their effectiveness in similar scenarios, offering insightful data regarding their performance.

   While the selection of TCs does not exhaustively cover every potential test scenario, it ensures comprehensive coverage of the diverse DCs within IaC, reflecting the real-world application challenges. Notably, E2E Testing has been deliberately excluded to maintain a manageable scope for the thesis, given the constraints of resources and the limited added value it would bring in this context.

This initial overview of the Proof-of-Concept's three key components establishes the context for understanding their interrelationships. This chapter progresses to an in-depth examination of each component's implementation, which is crucial for comprehending how they collectively actualize the theoretical framework.

4.1 Test Pipeline

Establishing the Test Pipeline for this thesis is predicated on a set of deliberately chosen requirements. These requirements are integral to demonstrating the feasibility of the theoretical concepts and are crucial for quantifying IaC testing costs. The requirements, formulated to guide the design and implementation of an effective and comprehensive IaC testing framework, are as follows:

1. *Test Approach Coverage Requirement*: The system must be capable of covering each test approach outlined in the IaC Test Pyramid (fig. 3.1), ensuring comprehensive testing across various levels.
2. *Test Tool Flexibility Requirement*: The system must offer flexibility in integrating multiple IaC test tools, acknowledging the limited test approach coverage of individual tools and the need for interoperability among diverse tools.
3. *Test Tool Adaptability Requirement*: The system must be adaptable for integrating and exchanging IaC test tools, addressing the evolving landscape of these tools within heterogeneous test contexts.
4. *Automation Requirement*: The system must execute all steps automatically upon triggering, enhancing efficiency and consistency in test execution.
5. *Runtime Measurement Requirement*: The system must accurately measure test runtimes, a critical component in quantifying the overall costs of IaC testing.
6. *Cost Calculation Requirement*: The system must be capable of calculating and analyzing the costs associated with IaC test execution, encompassing both cloud provider charges and resource utilization.
7. *Data Collection Requirement*: The system must be capable of collecting measurement data over multiple test cycles for comprehensive analysis and trend observation.

Implemented as a declarative Jenkins Pipeline[3], the Test Pipeline serves as the operational backbone of the PoC. Utilizing Jenkins, an open-source automation server[4], for its flexibility and its capacity to run the pipeline locally, as delineated in the provided setup instructions[5], the pipeline is crafted to align with these meticulously defined requirements.

In addressing these requirements, the pipeline's coverage of various IaC test approaches is detailed in section 4.1.1. The implementation addresses the integration of various IaC test tools through Containerization. This design choice not only meets the *Test Tool Flexibility* and *Adaptability Requirements* but also simplifies the setup process and

[3] `https://github.com/fex01/thesis-tf/blob/main/Jenkinsfile`
[4] `https://www.jenkins.io/`
[5] `https://github.com/fex01/thesis-ws/blob/main/jenkins/README.md`

enhances the adaptability of the test toolchain. The *Automation Requirement* is realized through Jenkins' autonomous nature, facilitating the automatic execution of all pipeline steps upon triggering.

The inclusion of custom support and data collection scripts within the pipeline is pivotal for capturing essential metrics such as test execution times and cost estimations, directly addressing the *Runtime Measurement*, *Cost Calculation*, and *Data Collection Requirements*. These aspects are critical to the quantitative evaluation of different IaC test approaches and are further discussed in section 4.1.2.

Crucially, the Test Pipeline is designed to be independent of any specific project or test case, rendering it a versatile instrument suitable for many Terraform projects. This characteristic is vital as it demonstrates that the pipeline is not merely an adjunct to a particular Configuration Under Test and accompanying Test Cases. Instead, it serves as a paradigmatic example of how to collect data to quantify IaC testing costs.

The following sections will detail the specific stages of the pipeline and their alignment with the established requirements, illustrating the pipeline's comprehensive applicability in diverse test environments.

4.1.1 Pipeline Stages

The diagram fig. 4.2 illustrates the dedicated Test Pipeline, designed to comprehensively fulfill the *Test Approach Coverage Requirement*. The stages of this pipeline are aligned with the levels of the proposed Test Pyramid (see fig. 3.1) and are color-coded accordingly. Stages with a white background are utility stages that support the test process and include *Initialize*, *Dry Run*, *Cost Breakdown*, *Cost Calculation*, and *Cleanup*:

- **Preparation**: At the initiation of each test cycle, Jenkins performs a fresh checkout from the project's Git repository. This action ensures the workspace is populated exclusively with code committed to the repository, eliminating discrepancies arising from uncommitted changes in a developer's local environment. Such a practice is essential to ascertain that the test results are influenced solely by the repository's official codebase, thus preserving the consistency and accuracy of the test outcomes. The clean state established by this process is fundamental in preventing the carryover of any residual state or data from previous pipeline activities.

- **Initialize**: The first stage of the pipeline is *Initialize*, wherein the *terraform init*[6] command is used to prepare the working directory for Terraform operations. This step configures the backend, installs modules, and downloads provider plugins that the code depends on.

- **Tool-Driven Static Testing (TL1)**: Following initialization, the pipeline engages the dark blue stages of the Tool-Driven Static Testing. The *terraform fmt* command ensures consistent formatting across the codebase, thus aiding in readability and reducing the cognitive load on developers.

[6]terraform init: `https://developer.hashicorp.com/terraform/cli/commands/init`

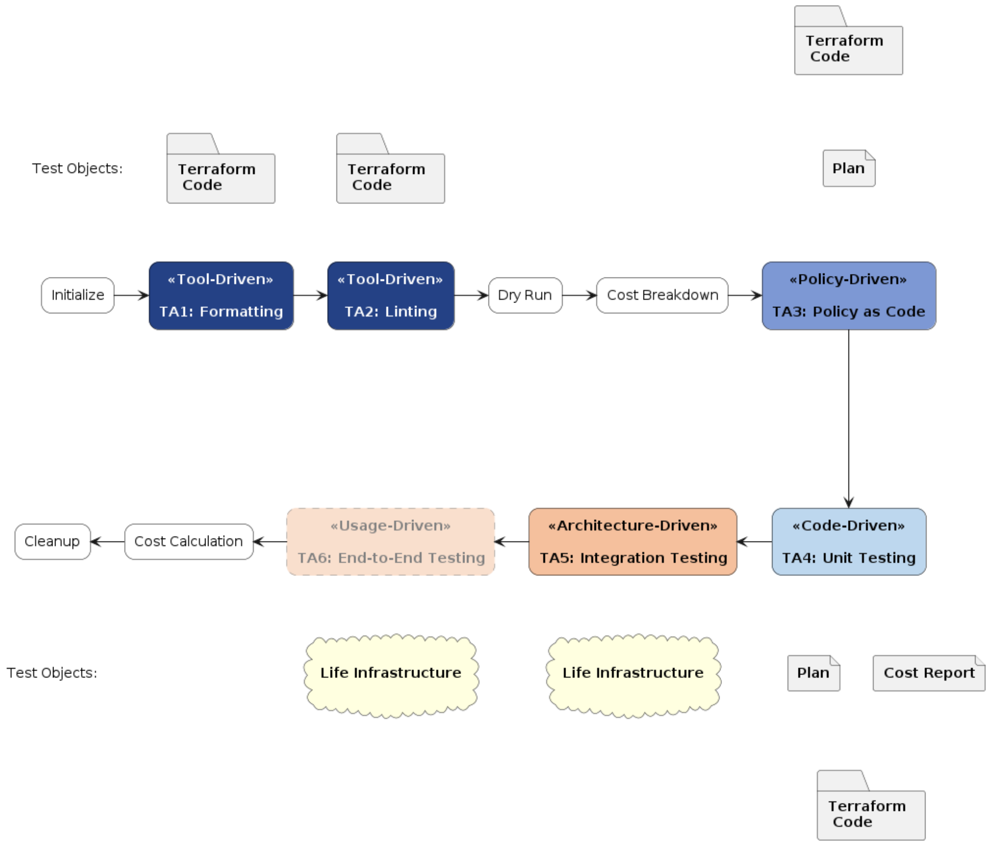
Terraform Code
Test Objects:
Terraform Code
Terraform Code
Plan
Initialize
«Tool-Driven» TA1: Formatting
«Tool-Driven» TA2: Linting
Dry Run
Cost Breakdown
«Policy-Driven» TA3: Policy as Code
Cleanup
Cost Calculation
«Usage-Driven» TA6: End-to-End Testing
«Architecture-Driven» TA5: Integration Testing
«Code-Driven» TA4: Unit Testing
Test Objects:
Life Infrastructure
Life Infrastructure
Plan
Cost Report
Terraform Code

Figure 4.2: Test Pipeline: Incorporates various exchangeable test tools, demonstrating flexibility. The stages are color-coded for visual alignment with the IaC Test Pyramid (fig. 3.1), while static test objects are grey, dynamic infrastructure is yellow, and supporting utility stages are white.

On the same test level, *terraform validate* acts as a linting tool within the pipeline. It statically analyzes the IaC configuration for syntactical correctness and structural compliance without accessing remote services, providing immediate feedback on potential configuration issues.

For detailed information on how the PoC measures runtimes during each Test Stage and for each test approach, please refer to section 4.1.2.

- **Dry Run**: The subsequent *Dry Run* stage, another utility stage, employs *terraform plan*[7] to create a deployment plan. This plan articulates the actions Terraform will execute upon applying the code without making any actual changes to the infrastructure, allowing for a review of potential changes.

[7] terraform plan: `https://developer.hashicorp.com/terraform/cli/commands/plan`

- **Cost Breakdown**: *Cost Breakdown* is the next utility stage, leveraging *Infracost*[8] to query AWS for resource pricing, using the deployment plan as input. This generates a cost report, providing the prices for the configured resources and aiding in financial governance.

- **Policy-Driven Static Testing (TL2)**: At the blue Policy-Driven Static Test level, *tfsec* is employed. This tool scans the Terraform code for potential security issues, ensuring the configuration adheres to security best practices and predefined policies.

- **Code-Driven Static Testing (TL3)**: The light blue Code-Driven Static Test stage continues with static testing involving configuration files, the deployment plan, and the cost report. The tools utilized in this stage include *pytest* and the native *terraform test* command.

  The *pytest* framework leverages the full capabilities of Python, a General Programming Language (GPL), offering the flexibility to construct complex test cases that extend beyond the scope of Terraform's domain-specific functionalities. This allows for implementing a wide range of test scenarios with the depth and breadth that Python's extensive programming features afford.

  Conversely, the *terraform test* command is designed to execute automated tests that are defined within the declarative domain of HashiCorp Configuration Language (HCL)[9], the same Domain Specific Language (DSL) used for Terraform configurations. This ensures a seamless integration of test cases with the infrastructure code, built upon the same conceptual framework. However, this congruence with Terraform's declarative nature inherently limits the complexity of the test cases that can be devised, confining them within the boundaries of Terraform's concepts and capabilities.

  Implementation detail: For Test Stages that involve multiple test tools, specific stages are created for each, such as *Code-Driven 1: pytest* and *Code-Driven 2: terraform test* in the Code-Driven Static Test stage. This is necessary because each stage uses a distinct Docker container tailored to the specific test tool. These sub-stages are not visualized to maintain clarity and focus of fig. 4.2.

- **Architecture-Driven Testing (TL4)**: For Architecture-Driven Testing, salmon-colored, *terraform test* is first employed to extend testing to live infrastructure. As previously discussed, while it operates within HCL's declarative domain, it also leverages Terraform's provider ecosystem to examine the state of deployed resources. The same constraints on the complexity of test cases within the static analysis context remain relevant here, as the tests are still bound by the constructs available within Terraform's configuration language.

[8]Infracost: `https://github.com/infracost/infracost`
[9]HCL: `https://github.com/hashicorp/hcl`

To address test requirements that exceed the scope of *terraform test*, *Terratest* is utilized. Based on Go, this tool can implement test cases that necessitate a broader programming capability. Go's use in the broader Terraform ecosystem is specifically prevalent among developers who create and test custom Terraform providers, as documented by HashiCorp[10]. This distinction is important; while Go is integral for those extending Terraform's functionality, infrastructure practitioners using standard Terraform workflows primarily work within HCL and may not interact with Go. *Terratest* is thus reserved for more complex test scenarios, particularly useful in dynamic integration testing where the robustness of Go can be fully leveraged.

- **Usage-Driven Testing (TL5)**: The Usage-Driven Test stage, depicted with a lighter, pastel shade, focuses on E2E testing, which validates the system's behavior from the user's perspective. This includes testing the infrastructure's actual deployment and operation and simulating real-world usage scenarios to ensure the system meets the required specifications.

  However, it is marked with a dashed border to indicate that test cases for this stage have not been implemented.

- **Cost Calculation**: Post-testing, the *Cost Calculation* stage is introduced, where custom Python scripts calculate the costs per test case. These scripts take the cost report and the measurements CSV file as inputs and produce an augmented CSV file that includes the calculated costs of each test case. For details, see section 4.1.2.

- **Cleanup**: In the final stage, *Cleanup*, the pipeline ensures that the test environment is reset to its original state. This involves archiving measurements, deleting the workspace, and using *cloud-nuke*[11]-a tool that is emphasized to be highly destructive and therefore should only be used in test environments-to remove resources. However, as *cloud-nuke* fails to destroy DB subnet groups[12], the *aws-cli*[13] tool is employed to delete these remaining resources explicitly.

In summary, the Test Pipeline is a meticulous construct that prioritizes test efficacy and cost efficiency, with each stage carefully chosen to contribute to the overarching goal of collecting runtime measurements and calculated cost estimations without deploying any resources permanently.

4.1.2 Data Collection

Collecting accurate and comprehensive data is foundational to the quantitative assessment of IaC testing. This subsection highlights the methodologies implemented to capture, process, and consolidate data efficiently. The focus is on custom scripts that cater

[10]Testing Terraform plugins: `https://developer.hashicorp.com/terraform/plugin/sdkv2/testing`
[11]cloud-nuke: `https://github.com/gruntwork-io/cloud-nuke`
[12]cloud-nuke issue #623: `https://github.com/gruntwork-io/cloud-nuke/issues/623`
[13]aws-cli v2: `https://github.com/aws/aws-cli/tree/v2`

to specific aspects of the data collection process, ensuring precision and relevance in the context of IaC testing.

Each script is tailored to address distinct data collection needs: runtime measurement, cost calculation, and post-build data aggregation. While the *run_ test.sh* script captures detailed runtime metrics, *calculate_ costs.py* deals with the complexities of cloud provider cost estimation. Finally, *collect_ data.sh* takes center stage in the post-build phase, merging data from multiple builds and enriching it with essential metadata.

Together, these scripts form a cohesive system that captures critical test data and adds layers of context and traceability. It is vital for a thorough and nuanced analysis of IaC test approaches and their implications.

Runtime Measurement

Accurately measuring test runtimes in the IaC testing pipeline is central to evaluating test approaches. This section discusses the detailed requirements for runtime measurement and the rationale behind the chosen methodology. It focuses on custom scripting due to its flexibility and compatibility with varying test environments.

The primary goal is to allow for direct comparison across different TAs for the same test case, leading to the following requirements for runtime measurement:

1. Measure the runtimes of each individual test.
2. Independence from specific test tools to maintain flexibility and interchangeability.
3. Utilization of seconds as the unit of measurement, ensuring a balance between accuracy, granularity, and practical relevance.
4. Attribution of runtimes to distinct dimensions: build, defect category, test case, test approach, and test tool.
5. Automation of the measurement and data collection process.

When exploring methods to achieve these requirements, two initial strategies were considered but ultimately deemed unsuitable: Jenkins metadata, specifically stage runtimes, includes additional overhead, such as the building and execution of Docker images, as well as not differentiating between individual tests. On the other hand, most test tool-specific reports offer individual test runtimes but vary significantly in format, which would have impeded the interchangeability of test tools - an essential aspect of the intended test framework.

The decision to opt for custom scripting resulted in its own challenge. The necessity of executing scripts within dockerized test tool environments without altering official Docker images dictated a need for POSIX-compliant scripting. This requirement emerged from the interaction between the chosen test environment and the commitment to maintaining tool interchangeability.

To address the stated requirements, the *run_test.sh*[14] script was implemented (see listing 4.1).

```sh
#!/bin/sh
# argument parsing and validation...
# Parse DC, TC, TA, and TEST_TOOL from TEST_COMMAND if not provided
# Change directory if EXECUTION_CONTEXT is not empty ...

# Get start time
start_time=$(date +%s)

# Execute the test command
eval "$TEST_COMMAND"
exit_code=$?

# Get end time
end_time=$(date +%s)

# Go back to the original directory if EXECUTION_CONTEXT is not empty...

# Calculate runtime
runtime=$(expr $end_time - $start_time)

# Determine if runtime costs are applicable; otherwise, set to 'NA' ...

# Prepare the CSV entry
csv_entry="$BUILD_NUMBER,$DEFECT_CATEGORY,$TEST_CASE,$TEST_APPROACH,
    $TEST_TOOL,$runtime,$runtime_costs"

# Append the CSV entry to the OUTPUT_FILE
echo "$csv_entry" >> $CSV_FILE

# Check if the test command was successful
if [ $exit_code -ne 0 ]; then
    echo "Error: Test \"$TEST_COMMAND\" failed."
    exit $exit_code
fi
```

Listing 4.1: run_test.sh snippet: measure and log runtimes independent of Jenkins and test tools

The functionality of the *run_test.sh* script aligns seamlessly with the requirements. It is designed to handle various parameters - from the build number to the test tool - and captures the start and end times of test execution. The calculated runtime is then systematically recorded in a CSV file, ensuring consistency and ease of data analysis. Adhering to POSIX standards, the script efficiently navigates the operational constraint and demonstrates compatibility across diverse test scenarios and environments.

For ease of use, the *run_grouped_tests.sh*[15] script was created: This auxiliary script facilitates the sequential execution of groups of tests, thus avoiding side effects which

[14]run_test.sh: https://github.com/fex01/thesis-tf/blob/main/scripts/run_test.sh

[15]run_grouped_tests.sh: https://github.com/fex01/thesis-tf/blob/main/scripts/run_grouped_tests.sh

could potentially be caused by parallel execution.

The runtime measurement scripts are integral to implementing the test pipeline at every test stage, ensuring comprehensive and accurate data capture for all test executions. This data is systematically aggregated into a build-specific CSV file, further adapted during the Cost Calculation stage to include additional insights. It is then archived in the Cleanup stage, rendering it available for detailed post-build analysis.

```
stage("Code-Driven 2: terraform test") {
    agent{
        docker{
            args '--entrypoint=""'
            image "hashicorp/terraform:${params.terraform_version}"
            reuseNode true
        }
    }
    environment {
        TEST_FOLDER = 'tests'
        TEST_APPROACH = '4'
        TEST_COMMAND = "terraform test -no-color -filter="
    }
    steps {
        withCredentials([usernamePassword(
            credentialsId: "aws-terraform-credentials",
                usernameVariable: "AWS_ACCESS_KEY_ID",
                passwordVariable: "AWS_SECRET_ACCESS_KEY"
        )]) {
            sh """scripts/run_grouped_tests.sh \\
                --build-number ${BUILD_NUMBER} \\
                --test-folder ${TEST_FOLDER} \\
                --test-approach ${TEST_APPROACH} \\
                --test-command '${TEST_COMMAND}' \\
                --csv-file ${CSV_FILE}"""
        }
    }
}
```

Listing 4.2: Exemplary Test Stage: use custom measurement scripts to collect data

To exemplify the application of the test script, consider the first Code Driven test stages (TL3), as illustrated in listing 4.2. The script is executed here with parameters tailored to this stage, including the build number and relevant test command. This process ensures that each test execution within this stage is methodically measured and recorded, contributing to the thoroughness of the overall test assessment.

Cost Calculation

Accurately calculating cloud provider costs in real-time for IaC testing presents a significant challenge. Tools like Infracost can estimate hourly or monthly charges by matching prices with individual Terraform deployments. However, the utility of these tools is constrained by several factors.

A key issue is the availability of detailed usage reports from AWS. These reports are not provided by default and require active configuration, which involves additional costs for S3 storage[16]. More critically, AWS inherently provides usage data with a delay, making these reports unsuitable for real-time analysis.

Compounding this issue is the varied billing modality across cloud resources. Some resources are billed in full-hour intervals, meaning the commencement of an hour incurs charges for the entire hour. In contrast, other resources are billed more granularly. The lack of clear and comprehensive information regarding the billing intervals for each resource type further hinders the ability to accurately calculate costs for shorter test runs.

Nevertheless, the aim is to approximate these costs as closely as possible. To facilitate a direct comparison across different TAs for the same test case, the cost calculation must meet specific requirements:

- Calculating cloud provider costs for each individual dynamic test.
- Delivering cost calculation results immediately for each build rather than at the end of a billing cycle.
- Basing cost calculations on the actual test runtimes rather than rounding to the nearest hour.
- Expressing cloud provider costs in USD, rounded to five decimals, to balance the need for granularity against the risk of comparing methodological errors rather than actual cost differences.
- Automating the calculation and collection of cost estimates.

Implementing the stated requirements for cost calculation in the IaC test framework lead to a series of considerations.

To approximate cloud provider costs for each dynamic test case, hourly resource costs in the configuration are determined. This is achieved by querying current AWS pricing via Infracost during the Cost Breakdown stage. The focus is primarily on runtime costs in alignment with the stated requirements. However, it is worth noting that Infracost can also include traffic-based costs, though this aspect is not within the current scope of analysis[17].

Understanding and applying the AWS billing modality is a significant aspect of the cost calculation. The investigation, particularly for the CUT, has led to the classification of AWS resources into two distinct categories based on their billing intervals:

- Resources such as "EKS cluster usage", "RDS Multi-AZ instance hours", and "Elastic Compute Cloud nodes" are billed by fractions of an hour, with precision to three

[16] `https://docs.aws.amazon.com/cur/latest/userguide/what-is-cur.html`

[17] Infracost - Usage-based resources: `https://www.infracost.io/docs/features/usage_based_resources/`

decimal places, as per the AWS Billing Overview. These resources are categorized under *fineGranularResourceSet* as delineated in formula 4.1:

$$\text{hourlyCostsFineGranular} = \sum \text{hourlyCost}(\text{fineGranularResourceSet}) \tag{4.1}$$

- Conversely, VPC Endpoints are billed based on total hours, according to the AWS Billing Overview. Additionally, AWS KMS keys are explicitly stated by AWS to be billed in hourly intervals[18]. These resources fall under the *hourlyBilledResourceSet*, as referenced in formula 4.2:

$$\text{hourlyCostsHourlyBilled} = \sum \text{hourlyCost}(\text{hourlyBilledResourceSet}) \tag{4.2}$$

This classification enables a cost calculation strategy tailored to each resource set's specific billing granularity. A conservative approach is adopted for resources with unclear billing intervals, treating them as if billed in full-hour intervals. While this strategy may result in slightly higher estimated costs, avoiding underestimation is preferable.

Deployment and destruction times are factored in, recognizing that not all resources begin accruing costs at the onset of the deployment phase, nor do they cease at the conclusion of the destruction phase. This calculation method may estimate slightly above the actual billed amount, serving as another proactive step to avoid underestimating costs.

For scenarios with multiple test cases in a single deploy-/destroy-cycle, like TC4, TC7, and TC9, the runtimes are equitably distributed using the *splitBy* variable. The conversion of runtime from seconds to hours is performed to match the unit of time used for AWS resource pricing, as specified in formula 4.3:

$$\text{runtime}_\text{h} = \frac{\left(\text{runtime}_\text{s}(\text{testCase}) + \frac{\text{runtime}_\text{s}(\text{deploy})+\text{runtime}_\text{s}(\text{destroy})}{\text{splitBy}}\right)}{3600} \tag{4.3}$$

These considerations culminate in the comprehensive formula 4.4, which calculates the total cloud provider costs for an individual dynamic test case. This formula integrates the hourly costs for both fine granular and hourly billed resources, factoring in the proportionate runtime for each test case within a deploy/destroy cycle:

$$\begin{aligned}\text{totalCost}(\text{testCase}) = {} & (\text{hourlyCostsFineGranular} \times \text{runtime}_\text{h}) \\ & + \left(\text{hourlyCostsHourlyBilled} \times \frac{\lceil \text{runtime}_\text{h} \rceil}{\text{splitBy}}\right)\end{aligned} \tag{4.4}$$

This theoretical framework is translated into practical application by the script *calculate_costs.py*[19], as indicated in listing 4.3. The code snippet exemplifies the classification of resources, calculation of costs, and the rounding of the final result to five decimal places for precision.

[18] AWS Key Management Service Pricing, visited 18.11.2023: `https://aws.amazon.com/kms/pricing/`

[19] calculate_costs.py: `https://github.com/fex01/thesis-tf/blob/main/scripts/calculate_costs.py`

```python
# resource classification
fine_granular_resource_types = ["aws_db_instance", "aws_eks_cluster", "
    aws_eks_node_group"]
hourly_interval_resource_types = ["aws_vpc_endpoint", "aws_kms_key"]
traffic_based_resource_types = ["aws_cloudwatch_log_group"]

# cost report parsing...

# Calculate costs
total_hourly_costs_fine_granular = hourly_costs_fine_granular * (runtime
    / 3600.0)
total_hourly_costs_interval = (hourly_costs_interval * ((runtime // 3600)
     + 1)) / split_by
total_costs = total_hourly_costs_fine_granular +
    total_hourly_costs_interval

# round result to 5 decimals
round(total_costs, 5)
```

Listing 4.3: Calculate the total cloud provider costs for an individual dynamic test case, incorporating the hourly costs for both fine granular and hourly billed resources, factoring in the proportionate runtime for each test case within a deploy/destroy cycle (calculate_costs.py).

The process of extending the measurements CSV file with calculated costs is facilitated by the script 'extend_measurements_with_costs.py'[20]. This script iterates over every entry in the measurements CSV file, methodically adding the cost calculation results specifically for dynamic test entries. This operation occurs towards the end of each build during the Cost Calculation stage, ensuring that each dynamic test entry is accurately updated with the relevant cost information.

Post-Build: Collect and Merge

The post-build phase is handled by the *collect_ data.sh*[21] script, located in the workspace repository to signify its use outside the pipeline context. This script consolidates test data from various builds and enriches it with specific build metadata, namely the Git commit ID, build start time, and build duration.

Including the Git commit ID is particularly significant as it links each dataset to the corresponding Terraform configuration or Test Pipeline changes, enhancing traceability and context. The build start time and duration offer additional insights into the build process, augmenting the data's analytical value.

Overall, the *collect_ data.sh* script functions as a critical component in post-build analysis, enabling a cohesive and context-rich compilation of test data across multiple builds.

[20] extend_measurements_with_costs.py: `https://github.com/fex01/thesis-tf/blob/main/scripts/extend_measurements_with_costs.py`

[21] collect_data.sh: `https://github.com/fex01/thesis-ws/blob/main/measurements/collect_data.sh`

4.2 Configuration Under Test

This thesis's CUT[22] is a Terraform configuration deployed on AWS. Adopted from Caracciolo's *Policy as Code* research [Car23], this configuration is designed to reflect a realistic IaC environment, providing a pertinent foundation for assessing various Test Approaches.

While the CUT was initially developed for a different study, its selection for this research was strategic. Its integration of multiple AWS resources and manageable size and complexity make it apt for demonstrating various test approaches.

In its existing form, the CUT offers a multifaceted environment reflective of real-world scenarios and is well suited for a detailed exploration of test approaches within the IaC context. The focus on interconnectivity and individual component functionality enhances the CUT's relevance for comprehensively evaluating the effectiveness of different TAs in a practical IaC setting. The configuration comprises the following services:

- *Virtual Private Cloud (VPC)*: The CUT begins with establishing an AWS VPC. This VPC forms the primary network layer, offering an isolated cloud environment that spans multiple availability zones. It is structured with subnets that categorize resources based on security and operational requirements, laying the groundwork for a secure and efficient network infrastructure.

- *Elastic Kubernetes Service (EKS) Cluster*: The AWS EKS Cluster is central to the CUT's application management. EKS simplifies the orchestration of containerized applications by managing the Kubernetes control plane. The service focuses on deploying and scaling applications while ensuring high availability across the infrastructure.

- *Elastic Compute Cloud (EC2) Instances*: The AWS EC2 instances in the CUT are worker nodes within the EKS cluster. These instances provide the necessary computational power and resources, enhancing the infrastructure's capacity to manage and run containerized applications efficiently.

- *Relational Database Service (RDS)*: Completing the configuration is the AWS RDS. RDS offers managed database services, integral for applications requiring reliable and scalable data storage solutions. Its integration within the VPC ensures secure and fast database connectivity.

The culmination of these components within the CUT creates a cohesive and functional infrastructure. This setup mirrors a realistic cloud environment and provides a well-suited background for evaluating various Test Approaches in IaC. The interplay of these services within the AWS ecosystem underscores the CUT's practicality for testing and demonstrates its capacity to simulate real-world challenges and solutions in infrastructure management.

[22]Configuration Under Test: `https://github.com/fex01/thesis-tf`

4.3 Test Cases

Implementing test cases is a pivotal aspect of this research, embodying the practical application of theoretical concepts delineated in the Test Coverage Matrix (see table 3.1). These TCs are not merely designed for functionality verification but are integral in evaluating the suitability of various Test Approaches for specific Defect Categories.

4.3.1 Requirements and Scope

The design and implementation of the Test Cases are guided by specific requirements, coupled with an understanding of the limitations in scope:

1. Implement at least one test for each Defect Category and Test Approach pair, thereby validating the proposed Test Coverage Matrix.
2. Explore the interchangeability of Test Approaches by attempting to implement the same test for different TAs.
3. Investigate the non-interchangeability of Test Approaches, particularly in scenarios where the DC is covered differently by distinct TAs.
4. The scope does *not* extend to exhaustive testing of the CUT but instead focuses on designing and executing selected representative tests.
5. E2E Testing is deliberately excluded from this exploration. This decision stems from a need to maintain the focus and manageability of the project, considering the limited incremental value E2E tests would offer in this specific context.

4.3.2 Overview

Table 4.1 provides an overview of the implemented test cases, correlating each with the relevant DC and TA. This visual representation facilitates a clear understanding of each test case's specific DC/TA intersection. It serves as a reference for the ensuing discussion on the design and implementation of the Test Cases.

4.3.3 Test Design and Implementation

With the context set by the requirements and the overview table, the focus shifts to the test design and implementation details for each DC. The process is characterized by careful selection and execution of tests, ensuring that each test case aligns with the predefined objectives and scope.

DC1: Conditional Logic The detection of conditional logic defects is addressed through Static Unit Testing (TA4) and Dynamic Integration Testing (TA5), as argued in section 3.5. The method involves designing tests implemented for both TAs rather than

	TA1 Format	TA2 Lint	TA3 PaC	TA4 Unit	TA5 Integration	TA6 E2E
DC1 Conditional				TC1, TC2	TC3, TC4	
DC2 Configuration Data				TC5	TC7	
DC3 Dependency				TC8	TC9	NI
DC4 Documentation				TC10		
DC5 Idempotency					TC11	NI
DC6 Security			TC12	TC13	TC14	NI
DC7 Service						NI
DC8 Syntax	NA	NA				
NI: Not Implemented (E2E Tests), NA: Not Applicable (TA does not support custom tests)						

Table 4.1: Mapping of Test Cases to Defect Categories and Test Approaches: Demonstrates the application of various Test Approaches to specific Defect Categories, enabling an evaluation of their suitability and serving as a basis for the following detailed analysis.

tailoring tests exclusively for one TA. This enables the validation of the claim that both TAs can effectively cover the same defect aspects.

The initial aspect under examination is variable validation. Utilizing Terraform's capability for custom condition validations[23], a rule is applied to the `db_pwd` variable[24] to enforce a minimum character length of eight. The logic of this validation is verified through scenarios where:

- `db_pwd.length < 8` results in a dry run or deployment failure,
- `db_pwd.length = 8` leads to a successful dry run or deployment,
- `db_pwd.length > 8` also ensures successful execution.

The test is implemented as TC1[25] for Static Unit Testing and as TC3[26] for Dynamic Integration Testing. However, it is important to note that TC3 does not execute the first scenario due to the limitations of *terraform test* in handling dynamic test cases expected to fail.

The second aspect involves a for loop, specifically testing the logic of creating sev-

[23]Terraform Custom Conditions: `https://developer.hashicorp.com/terraform/language/expressions/custom-conditions`

[24]Variable validation - var db_pwd, l. 37ff of `https://github.com/fex01/thesis-tf/blob/main/variables.tf`

[25]TC1 (tool: terraform test): `https://github.com/fex01/thesis-tf/blob/main/tests/tc1_dc1_ta4.tftest.hcl`

[26]TC3 (tool: terraform test): `https://github.com/fex01/thesis-tf/blob/main/tests/tc3_dc1_ta_5_no-predeployment.tftest.hcl`

eral private subnets based on `var.private_subnets_num`[27]. The test asserts the number of items in `vpc.private_subnets[]` equals `private_subnets_num`. This test is implemented as TC2[28] for Static Unit Testing and TC4[29] for Dynamic Integration Testing.

These implementations facilitate a direct comparison of TA4 and TA5 in terms of their effectiveness in detecting defects in conditional logic. The results and implications are further explored in chapter 5.

Implementation detail: TC4, TC7 (DC2), and TC9 (DC3) differ from standard dynamic tests by not including deploy and destroy phases. These tests allow execution against existing infrastructure and separate phase measurements for detailed analysis. For clarity in this thesis, tests including deploy and destroy phases are termed *standalone tests*. However, TC4, TC7, and TC9 are evaluated based on their *net TC runtimes*, focusing solely on the test execution period, excluding the setup and teardown times.

DC2: Configuration Data For detecting configuration data defects, TA4 and TA5 are both appropriate. Similar to the strategy for DC1, a test is designed that can be implemented for both TAs to assess their efficacy in covering the same defect aspects.

The focus is on verifying the Blast Radius, explicitly ensuring the use of the AWS Test Environment account. In the context of TA4, the test is implemented as TC5[30], where the verification involves asserting that the AWS credentials currently used match the ID of the AWS Test Environment account. This test aims to confirm the appropriate environmental setup before deployment.

For TA5, the same aspect is tested as TC7[31]. In this instance, it is asserted that the *ownerID* of the deployed VPC resource aligns with the ID of the AWS Test Environment account, confirming that the deployment was facilitated with the correct credentials. Although this strategy differs in its implementation from TC5, it addresses the same configuration aspect.

Both TC5 and TC7, despite their different methodologies, aim to verify the same aspect of configuration data. This implementation provides a basis for comparing the two TAs in terms of their ability to detect configuration data defects.

DC3: Dependency Static Unit Testing, Dynamic Integration Testing, and E2E Testing are suitable for addressing dependency defects. Each focuses on distinct aspects of dependencies. The following are illustrative examples, not an exhaustive list of possible dependency defect scenarios.

[27] for loop - Module *vpc*, attribute *private_subnets*, l. 10 of `https://github.com/fex01/thesis-tf/blob/main/vpc_module.tf`

[28] TC2 (tool: terraform test): `https://github.com/fex01/thesis-tf/blob/main/tests/tc2_dc1_ta4.tftest.hcl`

[29] TC4 (tool: terraform test): `https://github.com/fex01/thesis-tf/blob/main/tests/tc4_dc1_ta5.tftest.hcl`

[30] TC5 (tool: terraform test): `https://github.com/fex01/thesis-tf/blob/main/tests/tc5_dc2_ta4.tftest.hcl`

[31] TC7 (tool: terraform test): `https://github.com/fex01/thesis-tf/blob/main/tests/tc7_dc2_ta5.tftest.hcl`

Static Unit Testing often focuses on local dependencies. An example is verifying the availability and version accuracy of specific modules. For instance, a test could ascertain that the VPC module is locally available in version 5.1.2. This is achieved in TC8[32], which checks the module directory's presence and parses the module.json file to confirm the version. These operations use *pytest*, as they fall outside Terraform's native functionality.

Dynamic Integration Testing, on the other hand, typically verifies live dependencies. This method ensures that essential resources, such as the VPC defined in the CUT, are correctly deployed. This is implemented as TC9[33], serving as real-time verification of resource deployment and overall infrastructure integrity.

Regarding E2E Testing, it is often leveraged to confirm the availability of external resources, which might only be accessible in production environments. While this aspect is critical for comprehensive tests, it should be noted that in this thesis, no E2E tests were implemented. This decision is based on the scope and objectives of the research, as detailed previously (see section 1.2.

Implementing these tests indicates that, for DC3, different test approaches complement each other rather than act as substitutes. This distinction highlights the importance of combining test approaches to cover various aspects of dependencies in IaC.

DC4: Documentation Despite the uncommon nature of automated documentation tests in software or IaC testing contexts, designing tests for documentation defects remains feasible. Such tests can confirm the presence and content of documentation files, an aspect of ensuring comprehensive project documentation.

Since documentation is file-based and not part of the live infrastructure, TA3 (PaC) and TA4 are suitable test approaches. However, the chosen PaC tool, *tfsec*, does not support documentation verification policies, leading to a focus on TA4. For TA4, a test is developed to verify the existence of a specific section within the project's documentation. This test, designated TC10[34], involves checking that the *Readme.md* file in the project's root directory includes the string `"## Acknowledgment"`. Once again, *pytest* is used, as this test is out of scope for *terraform test*.

This implementation of a test for documentation defects highlights its peculiar nature. While feasible, such a method for thorough documentation verification may not be advisable due to its specificity and limited scope within the broader context of IaC testing.

DC5: Idempotency Terraform's method for idempotency testing, unlike static methods theorized for Chef [HROE13b, HROE13a], remains inherently dynamic. This aligns with Dynamic Integration Testing.

[32]TC8 (tool: pytest): `https://github.com/fex01/thesis-tf/blob/main/pytest/test_tc8_dc3_ta4.py`

[33]TC9 (tool: terraform test): `https://github.com/fex01/thesis-tf/blob/main/tests/tc9_dc3_ta5.tftest.hcl`

[34]TC10 (tool: pytest): `https://github.com/fex01/thesis-tf/blob/main/pytest/test_tc10_dc4_ta4.py`

The specific design of the idempotency test involves deploying the infrastructure twice in succession, with the expectation that no resources change during the second deployment. This test design verifies that Terraform configurations do not induce unintended changes upon reapplication, an essential aspect of idempotent behavior.

Following this design principle, the test is implemented as TC11[35]. The test leverages the built-in function `ApplyAndIdempotent()` from the tool *Terratest*.

In the context of an E2E test, the verification process might include ensuring that the reapplied infrastructure configuration does not delete or corrupt stateful components like databases. Such verifying that idempotency does not touch intentional stateful changes.

DC6: Security In the context of DC6, which addresses security defects, the applicable test approaches are PaC, Static Unit Testing, Dynamic Integration Testing, and E2E Testing. And while both static methods, TA3 and TA4, cover the same security aspect, TA5 and TA6 each address different security aspects.

The initial test aims to verify that passwords are marked as sensitive. TC12 is implemented for PaC, verifying that elements of type `variable` with names containing `password` or `pwd` have an attribute `sensitive` set to `true`.

As a static unit test, the same test is addressed as TC13. The TC verifies that all password attributes are marked as `(sensitive value)` by parsing the Terraform Dry Run output. Notably, the Dry Run output is parsed as a text file since the JSON output does not distinguish sensitive variables and exposes sensitive values in plain text.

The second test focuses on the exposure of the RDS endpoint. Implemented as TC14 in Dynamic Integration Testing, the test involves retrieving the RDS DNS name, resolving it to an IP address, and then asserting that this IP falls within private IP address ranges as defined by RFC 1918 for IPv4 and RFC 4193 for IPv6.

A security test in the context of E2E Testing could ensure that external customers do not gain access to restricted internal portals.

DC7: Service Service defects predominantly pertain to E2E Testing, focusing on verifying that consumer-facing services fulfill consumer needs. Although the implementation of E2E tests has been outside the scope of this thesis, their significance in a comprehensive test strategy is notable. An illustrative example of such a test might involve verifying the usability of a service running on the deployed infrastructure. This type of test would be crucial in assessing the end-to-end functionality and consumer experience, ensuring that the infrastructure operates as intended and effectively supports the services it hosts.

DC8: Syntax Syntax defects are addressed by test approaches Linting and Formatting. These approaches rely on built-in logic to detect and rectify syntax issues rather than supporting the implementation of custom tests. Due to this inherent characteristic, no specific tests were designed or implemented for this defect category. The focus of

[35]TC11 (tool: Terratest): `https://github.com/fex01/thesis-tf/blob/main/terratest/tc11_dc5_ta5_test.go`

these tools on automated syntax verification and correction underscores their role in maintaining code quality and standards without the need for custom test development.

This concludes the discussion of the implemented PoC for IaC testing, a construct founded upon theoretical understanding and practical application. In the forthcoming chapter *Evaluation*, the focus will shift to evaluating the measurements derived from this PoC, analyzing the data to discern the effectiveness and real-world implications of the discussed test approaches. This evaluation seeks to connect empirical findings with theoretical concepts, thereby enhancing the understanding of the quantitative aspects of IaC testing costs.

5 Evaluation

This chapter builds on the previously detailed theoretical underpinnings and practical implementation, engaging in a comprehensive analysis of test approaches within the IaC Provisioning domain. It critically evaluates empirical outcomes to gauge the effectiveness, efficiency, and adaptability of these approaches in real-world scenarios. A vital aspect of this analysis is the examination of static versus dynamic test approaches in IaC, aiming to elucidate their comparative advantages and limitations. This scrutiny informs the development of test strategies that balance test coverage and cost efficiency. Through this evaluation, the chapter aims to validate the proposed methodology's potential to enhance the reliability and cost-effectiveness of IaC test strategies.

5.1 Test Environment and Data Transparency

The evaluation was conducted on an Apple Mac Studio, equipped with an Apple M1 Max chip, 64 GB of memory, and running macOS 14.

The Test Pipeline[1] utilizes a dockerized Jenkins 2.427-jdk21 installation[2], hosted via Docker Desktop 4.25. For IaC operations, Terraform 1.6.2 is employed. The cost analysis is performed using Infracost 0.10.30. The test tool suite comprises Terraform 1.6.2, tfsec 1.28, PyTest 0.3.4-rc.2, and Terratest 0.29.0.

To ensure transparency and reproducibility of the results, the raw measurement data from all successful builds[3], as well as the scripts[4] used for filtering and aggregating this data, are made available in the thesis's repository.

5.2 Overview

This section presents an analytical overview of the test cases involved in the evaluation process. It aims to give the reader a detailed understanding of the runtime and cost statistics measured during the test phases. It sets the stage for a critical assessment of static and dynamic test approaches in the subsequent section.

5.2.1 Runtime

Figure 5.1 and table 5.1 are pivotal in illustrating the runtime distribution of the test cases. Figure 5.1 offers a comprehensive view of the average runtime for each test case,

[1] Test Pipeline definition: `https://github.com/fex01/thesis-tf/blob/main/Jenkinsfile`
[2] Jenkins setup: `https://github.com/fex01/thesis-ws/tree/main/jenkins`
[3] Measurement data: `https://github.com/fex01/thesis-ws/tree/main/measurements`
[4] Data aggregation scripts: `https://github.com/fex01/thesis-ws/tree/main/evaluation`

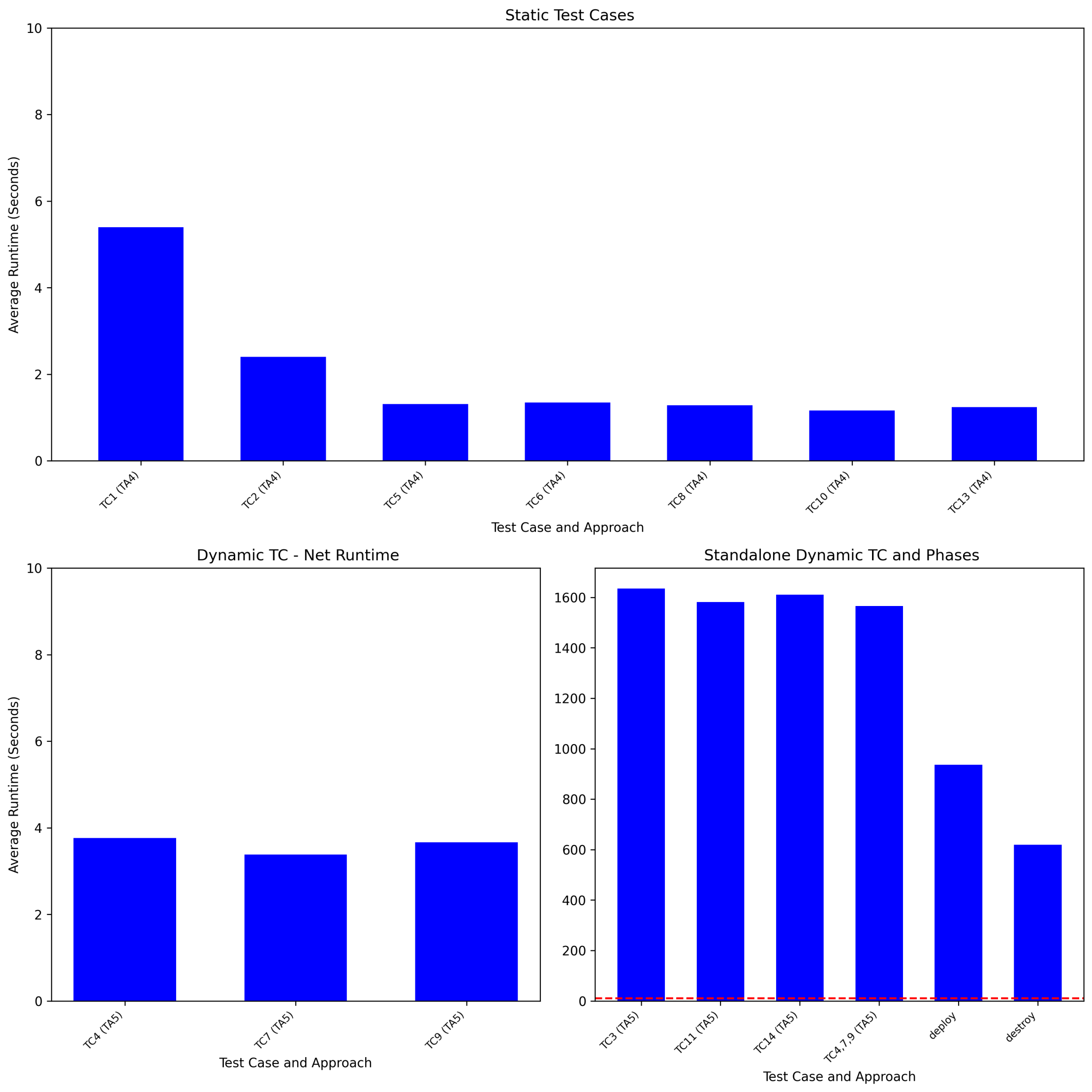

Figure 5.1: Consolidated overview over the mean runtime of each test case, including net and combined runtime for TCs 4, 7, and 9, as well as deploy and destroy phase.
The third plot's red dashed line at 10 seconds corresponds to the full scale of the first two plots, highlighting the scale disparity.

	Runtime (Seconds)							
	Mean	Median	Q1	Q3	IQR	Min	Max	Std Dev
TC1 (TA4)	5.40	5.00	5.00	6.00	1.00	5.00	6.00	0.49
TC2 (TA4)	2.40	2.00	2.00	3.00	1.00	2.00	3.00	0.49
TC5 (TA4)	1.31	1.00	1.00	2.00	1.00	1.00	2.00	0.46
TC6 (TA4)	1.35	1.00	1.00	2.00	1.00	1.00	2.00	0.48
TC8 (TA4)	1.28	1.00	1.00	2.00	1.00	1.00	2.00	0.45
TC10 (TA4)	1.16	1.00	1.00	1.00	0.00	1.00	2.00	0.37
TC13 (TA4)	1.24	1.00	1.00	1.00	0.00	1.00	2.00	0.43
TC4 (TA5)	3.76	4.00	4.00	4.00	0.00	3.00	4.00	0.44
TC7 (TA5)	3.38	3.00	3.00	4.00	1.00	3.00	4.00	0.50
TC9 (TA5)	3.67	4.00	3.00	4.00	1.00	3.00	4.00	0.48
TC3 (TA5)	1634.45	1645.00	1575.50	1716.00	140.50	1479.00	1743.00	89.64
TC11 (TA5)	1581.91	1576.00	1517.00	1668.50	151.50	1400.00	1710.00	104.08
TC14 (TA5)	1610.36	1548.00	1512.00	1594.00	82.00	1458.00	2285.00	231.08
TC4,7,9 (TA5)	1565.18	1580.00	1525.00	1617.00	92.00	1443.00	1652.00	69.02
deploy	935.82	921.00	898.50	969.00	70.50	881.00	1023.00	46.91
destroy	618.82	619.00	575.50	659.50	84.00	523.00	720.00	62.68

Table 5.1: Test Case Runtime Distribution: Detailed overview of the key runtime statistics of all implemented test cases.

including a detailed breakdown of phases such as deployment and destruction. The difference in scale accentuates the significant differences in runtimes across test cases. Table 5.1 complements this figure by providing a statistical breakdown, including mean, median, and standard deviation metrics.

5.2.2 Costs

	Cloud Provider Costs (USD)							
	Mean	Median	Q1	Q3	IQR	Min	Max	Std Dev
TC3	0.18017	0.18074	0.17700	0.18457	0.00757	0.17180	0.18602	0.00483
TC4	0.05882	0.05904	0.05810	0.05977	0.00167	0.05662	0.06039	0.00124
TC7	0.05879	0.05904	0.05808	0.05972	0.00164	0.05657	0.06039	0.00125
TC9	0.05879	0.05910	0.05804	0.05974	0.00170	0.05657	0.06034	0.00124
TC11	0.17734	0.17702	0.17384	0.18201	0.00816	0.16754	0.18424	0.00561
TC14	0.17887	0.17551	0.17358	0.17799	0.00442	0.17067	0.21522	0.01245
TC4,7,9	0.17644	0.17724	0.17428	0.17924	0.00496	0.16986	0.18112	0.00372

Table 5.2: Test Case Cloud Provider Costs Distribution

The provided box plot (fig. 5.2) and table 5.2 detail the cloud provider costs for IaC test cases, showing non-linear cost-runtime correlations due to varying resource billing intervals (see section 4.1.2).

These discrepancies are particularly noteworthy in Test Cases TC4, TC7, and TC9,

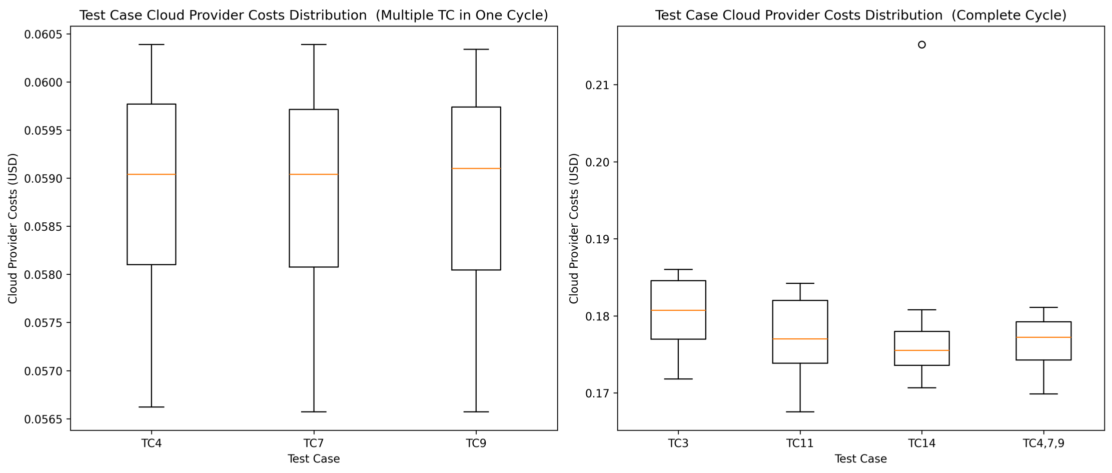

Figure 5.2: Test Case Cost Distribution: Calculated cloud provider costs per dynamic test case, split into test cases with a shared deployment cycle and standalone test cases with their own deployment cycle.
For easier comparison, test cases TC4, TC7, and TC9 are presented individually in the left plot and cumulatively as TC4,7,9 in the right plot.

which share a deploy/destroy cycle and are presented individually and cumulatively in TC4,7,9. This strategy allows for an equitable cost distribution, with setup and teardown overhead factored into individual cost assessments. Implications of this cost allocation are discussed in section 5.3.5, emphasizing the importance of understanding cost dynamics for strategic planning in IaC.

The outlier in TC14's costs is attributed to additional initial setup overhead, which is evident in its initial run.

The subsequent section will build upon this foundation, analyzing the implications of the observed runtime and cost statistics in the broader context of IaC testing.

5.3 Critical Assessment of Static and Dynamic Test Approaches

A thorough evaluation of static and dynamic test approaches is imperative for the domain of IaC. This analysis aims to uncover the strengths and limitations inherent to each method. Key considerations include execution time and cost implications. By examining empirical data alongside theoretical insights, the intent is to inform strategies for their efficient application in practice. Such a balanced evaluation is crucial for formulating an optimized test strategy that maximizes reliability and cost-efficiency in IaC deployments. The interplay of these test approaches, with their distinct characteristics, serves as the foundation for this critical assessment.

	TA1 Format	TA2 Lint	TA3 PaC	TA4 Unit	TA5 Integration	TA6 E2E
DC1 Conditional				●	◔	
DC2 Configuration Data				●	◔	
DC3 Dependency				◑	◑	◑
DC4 Documentation				◔		
DC5 Idempotency					●	◔
DC6 Security			◑	◔	◑	◑
DC7 Service						●
DC8 Syntax	◑	◑				

Table 5.3: Test Approach Suitability Overview: We categorize the ability of test approaches to cover specific defect categories, informed by implementation insights and test approach efficiency:
● TA is highly effective for the DC, offering comprehensive coverage
◑ TA addresses some but not all aspects of the DC
◔ denotes minimal coverage, reflecting a less effective TA

In addition to the efficiency evaluation, the test coverage matrix (section 3.5) and insights from the implementation of test cases (section 4.3) are incorporated to categorize the suitability of the identified test approaches. This categorization is systematically presented in table 5.3, which outlines the suitability of each test approach against different defect categories. This holistic view considers each evaluated test approach's coverage, contextual appropriateness, and efficiency.

Following this categorization, the ensuing subsections will present a detailed discussion of the rationale behind these classifications. The aim is to provide a comprehensive understanding of how and why certain test approaches may be more suitable for specific IaC scenarios, thereby guiding practitioners in optimizing their test strategies.

5.3.1 DC1, DC2: High Suitability for Static Unit Testing

Test	Validate Variables		Validate For Loop		Blast Radius		Combined TC
	TC1	TC3	TC2	TC4	TC5	TC7	TC4,7,9
Avg Runtime	5.4 sec	1634.45 sec	2.4 sec	3.76 sec	1.31 sec	3.38 sec	1565.18 sec
Avg Costs	NA	0.18 USD	NA	0.06 USD	NA	0.06 USD	0.18 USD

Table 5.4: DC1, DC2: Substitutable Test Cases

A notable observation arises from comparing Static Unit and Dynamic Integration Tests across specific DCs. For DC1 and DC2, as detailed in section 4.3.3, both test approaches target and successfully cover the same defect aspects. This crucial alignment in test coverage between static and dynamic methods allows for substituting dynamic tests with their static counterparts (see table 5.4).

The *Variable Validation* test analysis provides strong evidence to support the argument. The static test (TC1) and its dynamic equivalent (TC3) significantly differ in runtime, with the latter taking over 300 times longer than the former. Additionally, dynamic tests incur cloud provider costs during live deployment. Therefore, static tests are a more efficient alternative.

The *Loop Validation* test underscores a critical point. The net runtime of TC4 is akin to TC2. However, TC2 must be compared with the combined TC4,7,9's entire runtime for a fair assessment. This method highlights the substantial overhead from infrastructure setup and teardown.

Similar patterns emerge in the *Blast Radius* test. Comparisons involving TC5, TC7, and the combined TC4,7,9 reveal comparable trends. This dynamic is further disseminated in section 5.3.5.

Considering these findings, the recommendation tilts decidedly towards static unit tests for IaC testing, especially concerning DCs where static and dynamic tests cover identical defect aspects. The efficiency and cost-effectiveness of static tests and their ability to deliver equivalent test coverage as dynamic tests make them a viable and preferred choice in such scenarios. This conclusion is not merely a suggestion but a strong recommendation, particularly in contexts where efficiency, resource optimization, and cost containment are paramount.

5.3.2 DC3, DC6: Complementarity Suitability

Test	Local Dep.	Resource Dep.		Sensitive Passwords		RDS Endpoint
	TC8	TC9	TC4,7,9	PaC(TC12)	TC13	TC14
Avg Runtime	1.28 sec	3.67 sec	1565.18 sec	avg 1.37 sec	1.24 sec	1610.36 sec
Avg Cost	NA	0.06 USD	0.18 USD	NA	NA	0.18 USD

Table 5.5: DC3, DC6: Complementary Test Cases

In DC3 and DC6, static and dynamic tests reveal their distinct yet complementary roles (see table 5.5). For instance, while Static Unit Tests provide valuable insights, they do not render dynamic Integration or E2E tests redundant. Conversely, the dynamic test approaches cannot substitute static tests. This mutual supplementation underscores the necessity of employing different approaches to achieve a well-rounded test strategy.

In the context of DC6 Security, PaC demonstrates complementarity with dynamic approaches. While static unit tests and PaC tools may share similar test scenarios and runtime costs, PaC offers a distinct advantage with its built-in catalog of security policies. This facility enables built-in and customizable policy verification, which can be achieved with minimal effort compared to the specialized knowledge and development time required to create equivalent static unit tests.

Moreover, the efficiency and comprehensive coverage of PaC tools maintain the significance of dynamic security tests. Dynamic test approaches addressing different security aspects are deemed to have complementary suitability. They tackle security vulnerabilities and issues beyond the reach of static PaC testing, thus playing an integral role in a holistic security test strategy.

Despite these synergies, it is essential to recognize that each dynamic test invariably requires substantially more execution time, consistently being two orders of magnitude slower than their static counterparts. However, in the context of dependency and security defects, static approaches, while significantly more performant, cannot replace dynamic tests. They serve as a complement, enhancing the overall test strategy.

The strategic deployment of these test approaches thus becomes pivotal. The immediacy of test results may, at times, prioritize static approaches. Conversely, a comprehensive defect coverage necessitates the inclusion of dynamic tests. A balanced test strategy might entail the execution of static tests at every commit, coupled with requiring dynamic tests prior to branch merging. This balanced strategy effectively harnesses static and dynamic test strengths, ensuring a thorough and efficient test process.

5.3.3 DC4, DC8: Exclusive to Static Testing

Test	Readme TC10	Formatting TA1	Linting TA2
Avg Runtime	1.16 sec	0.04 sec	2.13 sec

Table 5.6: DC4, DC8: Exclusive to Static Testing

DC4 and DC8, which encompass documentation and syntax defects, respectively, are unique in their reliance solely on static test approaches (see table 5.6). This exclusivity stems from the nature of the defects, as they are intrinsically linked to IaC configuration files.

Identifying documentation defects, although feasible through automated tests as exemplified by TC10, is not a common practice. Engaging in extensive automated documentation testing may lead to a phenomenon known as *test creep*. This term describes an excessive expansion of the test scope, inadvertently increasing the maintenance requirements for tests without proportionate gains in system stability or reliability. Therefore, despite being the sole applicable test approach, static unit testing is deemed only limitedly suitable for DC4.

The test approaches for syntax defects, namely Formatting and Linting (TA1 and TA2), are primarily tool-driven. This characteristic implies that these tools are not designed for custom test implementation, eliminating the need for test development efforts. Furthermore, these approaches offer the advantage of rapid feedback. For instance, TA1's execution time is so swift that it often rounds to zero seconds. TA2, although slightly more time-consuming, still boasts a short average runtime of approximately two seconds.

This duration is comparable to that of a single static unit test, underscoring the efficiency of these static test approaches in addressing DC8. Both approaches cover different aspects of syntax defects, so they are of complementary suitability.

5.3.4 DC5, DC7: Exclusive to Dynamic Testing

Test	Idempotency TC11	Service Not Implemented
Avg Runtime	1581.91 sec	NI
Avg Cost	0.18 USD	NI

Table 5.7: DC5, DC7: Exclusive to Dynamic Testing

In contrast to DC4 and DC8, which are exclusively addressed by static test approaches, DC5 and DC7 represent areas where dynamic test approaches are essential[5].

DC5 and DC7, encompassing idempotency and service defects, respectively, are distinct in their requirement for dynamic testing. This necessity arises from the characteristics of these defects, which are inherently linked to the behavior and state of the live infrastructure rather than static aspects of the IaC configuration files.

Idempotency, evaluated through test case TC11, necessitates dynamic testing. Integration tests are ideally suited for this, as they focus on the interactions between system components, enabling precise detection of idempotency issues.

On the other hand, End-to-End (E2E) tests are less suitable for idempotency checks due to their design perspective. E2E tests, based on user perspective, primarily assess the overall system behavior rather than the functionality of individual components. While they might reveal failures caused by idempotency defects, E2E tests do not effectively isolate them as directly attributable to idempotency issues.

Service defects, represented by DC7, present a different challenge. The implementation of E2E tests, which would be pivotal in evaluating these defects, was not feasible within the scope of this thesis, as discussed in section 4.3. The absence of empirical data for this area underscores a gap in the current research. It highlights the need for further investigation into dynamic test methodologies for service defects in IaC.

5.3.5 Dynamic Tests: Net Runtime VS Overall Runtime

The evaluation of dynamic testing in IaC requires a nuanced understanding of the interplay between net runtime and overall runtime. This distinction becomes evident when considering the individual test cases and the associated phases of deployment and teardown. The orders of magnitude differences between static and dynamic test runtimes primarily arise from the overhead associated with infrastructure management.

[5] As discussed in section 3.3, there are theoretical methods such as model-driven static testing that aim to extend static testing to encompass dynamic aspects. However, these techniques are not yet of practical relevance

Test Category	Net Runtime			IaC Phases		Combined TC
	TC4	TC7	TC9	Deploy	Destroy	TC4,7,9
Avg Runtime	3.76 sec	3.38 sec	3.67 sec	935.82 sec	618.82 sec	1565.18 sec
Avg Cost	0.06 USD	0.06 USD	0.06 USD	NA	NA	0.18

Table 5.8: Dynamic Tests: Combining Multiple Tests in One Cycle

A systematic examination is conducted in the context of test cases TC4, TC7, and TC9. The infrastructure is predeployed, facilitating the isolation and measurement of the test execution time exclusively. This analysis reveals an intriguing observation: the net runtime of dynamic tests, averaging between 3 to 4 seconds, only marginally exceeds that of static tests. This finding underscores the efficiency of dynamic test executions in their isolated state.

The stark contrast in runtime between static and dynamic tests primarily emanates from the lifecycle management of the tested infrastructure. The time required for deployment and teardown of the infrastructure is not just a contributing factor but a dominant one in the overall runtime. It extends the total duration dramatically, averaging over 1550 seconds. This indicates that the overall runtime for dynamic tests is more than 380 times longer than their net runtime, a staggering testament to the immense overhead brought on by infrastructure lifecycle management.

From a strategic standpoint, this insight informs a strategic recommendation for dynamic test procedures. Aggregating multiple test cases within a singular deployment and destruction cycle emerges as a more efficient practice. This strategy effectively distributes the significant overhead across several test cases, thereby improving the cumulative efficiency of the test process.

Additionally, dynamic tests in environments where the infrastructure is permanently deployed, such as permanent test environments, staging environments, or testing in production, show improved runtime efficiency. An example is blue/green deployments, where two production environments are maintained in active and standby states. However, it is imperative to acknowledge the trade-off involved in these scenarios. The advantage of faster dynamic test execution in permanently deployed environments comes at the cost of continuous resource allocation. This cost manifests as direct financial expenses with cloud providers or opportunity costs due to resources being persistently engaged. Thus, while aligning the runtime of dynamic tests closer to static tests, maintaining a permanent infrastructure entails careful consideration of these additional costs.

The findings from this chapter's evaluation crystallize the nuanced balance between static and dynamic test approaches in IaC. They underscore the efficiency of static tests regarding execution speed and cost while affirming the necessity of dynamic tests for comprehensive defect coverage. This analysis, grounded in empirical data, lends weight to the strategic recommendations for IaC testing that will be concisely summarized in the concluding chapter.

Transitioning to the *Conclusion*, the focus shifts to integrating these insights, addressing the research question, and outlining future research directions, thus bridging the gap between specific evaluation outcomes and overarching conclusions in the field of IaC testing.

6 Conclusion

This work's contributions to IaC Testing are centered around developing a quantitative framework for cost assessment. This conclusion chapter aims to consolidate critical insights, address the thesis's limitations, and outline potential avenues for further research.

6.1 Summary

This thesis establishes a systematic methodology for quantifying testing costs for IaC provisioning, aiming to guide the development of optimized test strategies. Two critical metrics for comparing IaC test approaches, suitability and efficiency, were identified, and a process for their quantification was developed.

Efficiency is explored through various cost factors (see section 3.1), emphasizing runtime costs. This includes measuring test execution time and calculating associated cloud provider costs, which also represent opportunity costs in self-hosted infrastructures.

Analyzing suitability, the second pillar of our methodology, led to the identification of six relevant test approaches for IaC provisioning (see section 3.3). The proposed IaC Test Pyramid (section 3.4) and Rahman et al.'s Defect Taxonomy (see section 3.2) structured the test approach evaluation, culminating in a comprehensive test coverage matrix (section 3.5).

In the practical application of our methodology, we developed a comprehensive Test Pipeline (section 4.1). This Pipeline, incorporating six test tools and three utility tools, facilitates IaC testing across the broad spectrum of identified approaches. The CUT is a Terraform project, detailed in section 4.2. Custom test cases, aligned with the test coverage matrix, enabled direct comparison of different test approaches by implementing identical test scenarios (section 4.3).

The Test Pipeline's application provided reliable runtime measurements for chapter 5, quantifying, for example, that static unit tests are two orders of magnitude faster than dynamic approaches (section 5.3.1).

This research's methodology and Proof-of-Concept implementation offer a scalable framework for organizations to assess and compare test strategies across various IaC scenarios. These broadly applicable insights and methods inform test practices in diverse contexts.

In conclusion, this thesis provides a detailed examination of the cost implications of test approaches for IaC provisioning. The following section will reflect on the initial research question and the broader implications of the elaborated findings.

6.2 Synthesis

This thesis began with the intention to understand how the costs of different IaC test approaches could be quantitatively assessed and compared. From these initial objectives followed the specific research question that guided this thesis: *"How can we measure the suitability and efficiency of existing test approaches to help developers formulate tailored test strategies for Infrastructure as Code?"*. The methodology and PoC developed in this thesis have addressed the 'how' part of the question, enabling empirical measurements of suitability and efficiency in IaC testing.

From the application of this methodology, derive the following key findings that form the synthesis of this research:

- The evaluation confirms a significant speed advantage of static analysis methods over dynamic test approaches. Specifically, in the context of our data, this advantage amounts to at least two orders of magnitude (section 5.2.1). This finding provides empirical backing to what was previously a well-recognized yet unquantified understanding in the field.

- The effectiveness of static unit tests is contingent on the specific defect category. Static tests can replace dynamic approaches for defect categories Conditional Logic (DC1) and Configuration Data (DC2) (section 5.3.1) while serving as supplements for categories Dependency (DC3) and Security (DC6) (section 5.3.2). This distinction facilitates a more strategic allocation of testing resources.

- With existing test tools, the utility of static methods is exclusive to defect categories Documentation (DC4) and Syntax (DC8) (section 5.3.3), whereas dynamic approaches are uniquely suited for categories Idempotency (DC5) and Service (DC7) (section 5.3.4). This delineation offers a clearer understanding of each test approach's limitations and specific applications.

In addition to these anticipated findings, an unexpected yet informative observation emerged during our analysis. By isolating the net runtime of dynamic tests from the deployment and teardown overhead, it was observed that their adjusted net runtimes closely align with those of static tests (section 5.3.5). This discovery suggests significant optimization potential, from running multiple tests within a single deploy/destroy cycle to utilizing permanent test deployments.

These conclusions, while perhaps intuitively understood by IaC testing practitioners, are now grounded in empirical evidence. The thesis's methodology has quantified this understanding and provided a framework for systematically comparing different test approaches. This contribution is significant because it enables developers to balance test coverage with runtime costs more effectively.

While this thesis fulfills its objective by offering a quantitatively grounded methodology for evaluating IaC test approaches and informing test strategies, it is essential to acknowledge the limitations discussed in section 6.3. These limitations, including the explorative nature of this research and the absence of broader empirical studies, highlight the need for caution in generalizing the study's findings. They also underscore opportunities for future research to refine and build upon the applied methodology.

This thesis represents an initial step towards addressing the identified research gap in IaC testing, laying a foundation for empirical data-driven advancements and prompting further academic exploration in establishing best practices.

6.3 Threats to Validity

In addressing the threats to the validity of this thesis, it is pertinent first to consider the research methodology. The strategy was primarily explorative, influenced by constraints in time and resources, particularly regarding the thorough documentation of the examination process and the comprehensive evaluation of literature. While this method was expected to capture the majority of relevant sources, a systematic literature review could have identified additional significant literature. Such a systematic process would have also provided a more structured framework for source identification, thereby enhancing the rigor and reproducibility of the analysis. Despite these limitations, we have endeavored to document our methodology and source selection process in detail, laying a groundwork for future research to build upon and refine our work.

Moving to the specific content of our examination, the lack of empirical studies explicitly focusing on the suitability of IaC test approaches posed a challenge. As elucidated in section 3.5, this gap in the literature could lead to a biased understanding of the test coverage matrix. Our PoC implementation illustrates the potential of specific TAs to cover identified defect categories, yet it cannot conclusively negate the possibility of other, more effective test designs. This underscores the need for caution in generalizing the elaborated conclusions and highlights an area ripe for future empirical investigation.

Another aspect of the presented thesis relying on existing literature is the defect taxonomy proposed by Rahman et al. [RFPW20], which was based on the analysis of Puppet scripts. Applying this taxonomy to Terraform did not reveal significant inconsistencies; however, a dedicated analysis focused on Terraform could unearth additional or more fine-grained defect categories. The absence of such targeted research in this thesis signals an opportunity for further exploration in this domain.

The hypothesis regarding the Test Level hierarchy also presents a notable assumption in this work. We posited that generalized and centralized Policies for PaC are likely to incur less per-project development and maintenance effort than project-specific unit tests. While reasonable for large-scale organizations with numerous projects, this assumption may not hold in smaller organizations where the effort is not shared among a significant number of projects. Future research could thus aim to empirically determine the organizational scale at which PaC becomes beneficial, potentially validating or challenging this postulate.

Finally, the conclusions presented in section 5.3 are inherently tied to the specific tools, CUT, and test cases used in this thesis. It is posited that the trends and conclusions observed likely indicate broader patterns, but their universal applicability is yet to be confirmed. This aspect of the research highlights the importance of conducting extensive empirical studies to validate and possibly broaden the scope of the presented conclusions.

6.4 Outlook

The research conducted in this thesis opens several avenues for future exploration in the realm of IaC testing. One significant direction involves extending test objectives beyond defect coverage. This expansion could include integrating Clean Code principles and ISO Software Quality metrics. By incorporating aspects such as maintainability and readability derived from Clean Code recommendations and aligning with ISO software quality metrics, the scope of IaC testing can be substantially broadened. This strategy enhances the code's quality and aligns with industry standards, providing a more comprehensive evaluation framework.

Further, engaging with industry practitioners through surveys could yield valuable insights into existing IaC test strategies. While this thesis has put forth recommendations for test strategies based on theoretical analysis, understanding the practical applications and strategies currently used in the industry is crucial. A survey or literature review could be instrumental in identifying prevalent testing patterns and strategies, thereby bridging the gap between theoretical models and industry practices.

Collaboration with industry partners is another crucial step for future research. Such partnerships allow for validating and refining theoretical models in real-world scenarios. This cooperation enhances the practical relevance of the research and ensures that the findings are grounded in real-world applications and challenges.

In parallel, research into specific test approaches warrants dedicated attention. Investigating and developing new methodologies for testing in the context of IaC is fertile ground for further studies. Such exploration might include an in-depth analysis of existing approaches, similar to Caracciolos PaC thesis [Car23], or the creation of novel test techniques tailored to the unique challenges and opportunities presented by IaC environments.

Another promising direction for investigation involves data mining of public Terraform projects. Analyzing these projects could reveal standard testing practices and prevalent defects in IaC implementations. Potential insights from such analysis include understanding how IaC is utilized in real-world scenarios, the common challenges faced, and the effectiveness of current test practices.

Lastly, expanding the PoC implementation developed in this thesis is crucial. This expansion could include adapting the Test Pipeline to various cloud providers and configuration repositories. While tools like Infracost facilitate the integration with different cloud environments, adjusting the Test Pipeline to diverse IaC projects may require modifications in test case implementation and execution. This effort would enhance the utility of the PoC and provide a platform for testing and comparing user-specific test runtime costs in various environments. It is important to note that the current PoC is a starting point, necessitating further refinement for broader application and validation in diverse scenarios.

The thesis has established a foundation for quantitative cost assessment in IaC testing, offering methodologies applicable across various IaC environments. While specific to the work's context, the principles and frameworks presented have broader applicability, marking a significant contribution to the field and setting a direction for future research.

List of Acronyms

AWS	Amazon Web Services
CAMS	Culture, Automation, Measurement, and Sharing
CD	Continuous Deployment or Continuous Delivery
CFR	Change Failure Rate
CI	Continuous Integration
CRUD	Create, Read, Update, and Delete
CUT	Configuration Under Test
DC	Defect Category
DSL	Domain Specific Language
E2E	End-to-End
EC2	Elastic Compute Cloud
EKS	Elastic Kubernetes Service
FaaS	Function as a Service
GPL	General Programming Language
HCL	HashiCorp Configuration Language
IaaS	Infrastructure as a Service
IaC	Infrastructure as Code
ISTQB	International Software Testing Qualifications Board
MTTR	Mean Time to Recovery
NIST	National Institute of Standards and Technology
PaaS	Platform as a Service
PaC	Policy as Code
PoC	Proof of Concept
QA	Quality Assurance
RDS	Relational Database Service
SaaS	Software as a Service
SUT	System Under Test
TA	Test Approach
TC	Test Case
TL	Test Level
USD	US Dollar
VM	Virtual Machine
VPC	Virtual Private Cloud

Bibliography

[Adz11] Adzic, Gojko: *Specification by Example: How Successful Teams Deliver the Right Software.* Simon and Schuster, June 2011.

[Bei03] Beizer, Boris: *Software Testing Techniques.* Dreamtech, 2003.

[BR20] Bhuiyan, Farzana Ahamed and Akond Rahman: *Characterizing Co-Located Insecure Coding Patterns in Infrastructure as Code Scripts.* In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering Workshops*, pages 27–32, Virtual Event Australia, September 2020. ACM.

[Bri19] Brikman, Yevgeniy: *How to Test Infrastructure Code*, 2019. Accessed: 2023-09-18, URL: https://terratest.gruntwork.io/docs/getting-started/introduction/#watch-how-to-test-infrastructure-code.

[Bur05] Burgess, Mark: *A TINY OVERVIEW OF CFENGINE: CONVERGENT MAINTENANCE AGENT.* Proceedings of the 1st International Workshop on Multi-Agent and Robotic Systems, MARS/ICINCO, Vol. 2005, 2005.

[BWHW06] Biffl, Stefan, Dietmar Winkler, Reinhard Höhn and Herbert Wetzel: *Software Process Improvement in Europe: Potential of the New V-modell XT and Research Issues.* Software Process: Improvement and Practice, 11(3):229–238, 2006.

[Car23] Caracciolo, Mattia: *Policy as Code, how to automate cloud compliance verification with open-source tools.* laurea, Politecnico di Torino, April 2023.

[CDM18] Contan, Andrei, Catalin Dehelean and Liviu Miclea: *Test Automation Pyramid from Theory to Practice.* In *2018 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR)*, pages 1–5, May 2018.

[CDPP22] Chiari, Michele, Michele De Pascalis and Matteo Pradella: *Static Analysis of Infrastructure as Code: A Survey.* In *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*, pages 218–225, March 2022.

[CG09] Crispin, Lisa and Janet Gregory: *Agile Testing: A Practical Guide for Testers and Agile Teams.* Pearson Education, 2009.

[Chi96] Chillarege, R.: *Orthogonal Defect Classification.* In *Handbook of Software Reliability Engineering*, volume 1, page 886. IEEE Computer Society Press, 1996.

[CIO] CIO BRD: *V-Modell XT Kurz und Knackig – ein Überblick.* Accessed: 2023-09-01, URL: https://www.cio.bund.de/Webs/CIO/DE/digitaler-wandel/Achitekturen_und_Standards/V_modell_xt/V_modell_xt_ueberblick/v_modell_xt_ueberblick_artikel.html.

[Coh09] Cohn, Mike: *The Forgotten Layer of the Test Automation Pyramid*, December 2009. Accessed: 2023-07-17, URL: https://www.mountaingoatsoftware.com/blog/the-forgotten-layer-of-the-test-automation-pyramid.

[CPZF19] Catolino, Gemma, Fabio Palomba, Andy Zaidman and Filomena Ferrucci: *Not All Bugs Are the Same: Understanding, Characterizing, and Classifying Bug Types.* Journal of Systems and Software, 152:165–181, June 2019.

[DE05] DIN and ENISO: *9000 (2005): Quality Management Systems-Fundamentals and Vocabulary.* ISO, 9000, 2005.

[EGHS16] Ebert, Christof, Gorka Gallardo, Josune Hernantes and Nicolas Serrano: *DevOps.* IEEE Software, 33(3):94–100, May 2016.

[FHK18] Forsgren, Nicole, Jez Humble and Gene Kim: *Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations.* IT Revolution, March 2018.

[GGTP19] Guerriero, Michele, Martin Garriga, Damian A. Tamburri and Fabio Palomba: *Adoption, Support, and Challenges of Infrastructure-as-Code: Insights from Industry.* In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 580–589, September 2019.

[Har] Harrison, Russell: *How to Avoid Puppet Dependency Nightmares With Defines.* Accessed: 2023-09-16, URL: https://content.cloud.redhat.com/blog/how-to-avoid-puppet-dependency-nightmares-with-defines.

[HBR20] Hasan, Mohammed Mehedi, Farzana Ahamed Bhuiyan and Akond Rahman: *Testing Practices for Infrastructure as Code.* In *Proceedings of the 1st ACM SIGSOFT International Workshop on Languages and Tools for Next-Generation Testing*, pages 7–12, Virtual USA, November 2020. ACM.

[HF10] Humble, Jez and David Farley: *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation.* Pearson Education, July 2010.

[HK07] HUIZINGA, DOROTA and ADAM KOLAWA: *Automated Defect Prevention: Best Practices in Software Management.* John Wiley & Sons, August 2007.

[HR22] HASSAN, MOHAMMAD MEHEDI and AKOND RAHMAN: *As Code Testing: Characterizing Test Quality in Open Source Ansible Development.* In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 208–219, April 2022.

[HROE13a] HUMMER, WALDEMAR, FLORIAN ROSENBERG, FÁBIO OLIVEIRA and TAMAR EILAM: *Automated Testing of Chef Automation Scripts.* In *Proceedings Demo & Poster Track of ACM/IFIP/USENIX International Middleware Conference*, MiddlewareDPT '13, pages 1–2, New York, NY, USA, December 2013. Association for Computing Machinery.

[HROE13b] HUMMER, WALDEMAR, FLORIAN ROSENBERG, FÁBIO OLIVEIRA and TAMAR EILAM: *Testing Idempotence for Infrastructure as Code.* In EYERS, DAVID and KARSTEN SCHWAN (editors): *Middleware 2013*, Lecture Notes in Computer Science, pages 368–388, Berlin, Heidelberg, 2013. Springer.

[IEE10] IEEE: *IEEE Standard Classification for Software Anomalies.* IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993), pages 1–23, January 2010.

[IST23a] ISTQB GLOSSARY: *Acceptance Testing*, October 2023. Accessed: 2024-02-01, URL: `https://glossary.istqb.org/en_US/term/acceptance-testing-3-2`.

[IST23b] ISTQB GLOSSARY: *Black-Box Testing*, May 2023. Accessed: 2024-02-01, URL: `https://glossary.istqb.org/en_US/term/black-box-testing-3-2`.

[IST23c] ISTQB GLOSSARY: *Component*, October 2023. Accessed: 2024-02-01, URL: `https://glossary.istqb.org/en_US/term/component-2`.

[IST23d] ISTQB GLOSSARY: *Component Testing*, October 2023. Accessed: 2024-02-01, URL: `https://glossary.istqb.org/en_US/term/component-testing-4-3`.

[IST23e] ISTQB GLOSSARY: *Exploratory Testing*, October 2023. Accessed: 2024-02-01, URL: `https://glossary.istqb.org/en_US/term/exploratory-testing-4-3`.

[IST23f] ISTQB GLOSSARY: *Integration Testing*, October 2023. Accessed: 2024-02-01, URL: `https://glossary.istqb.org/en_US/term/integration-testing-3-2`.

[IST23g] ISTQB GLOSSARY: *System Testing*, October 2023. Accessed: 2024-02-01, `https://glossary.istqb.org/en_US/term/system-testing-3-2`.

[IST23h] ISTQB Glossary: *System Under Test*, May 2023. Accessed: 2024-02-01, URL: https://glossary.istqb.org/en_US/term/system-under-test-2-1.

[IST23i] ISTQB Glossary: *Test Level*, October 2023. Accessed: 2024-02-01, URL: https://glossary.istqb.org/en_US/term/test-level-2.

[IST23j] ISTQB Glossary: *White-Box Testing*, May 2023. Accessed: 2024-02-01, URL: https://glossary.istqb.org/en_US/term/white-box-testing-1.

[Jav] JavaTpoint: *V-Model (Software Engineering) - Javatpoint*. Accessed: 2023-09-04, URL: https://www.javatpoint.com/software-engineering-v-model.

[KGR+21] Kumara, Indika, Martín Garriga, Angel Urbano Romeu, Dario Di Nucci, Fabio Palomba, Damian Andrew Tamburri and Willem-Jan van den Heuvel: *The Do's and Don'ts of Infrastructure Code: A Systematic Gray Literature Review*. Information and Software Technology, 137:106593, September 2021.

[KMB17] Kandil, Passant, Sherin Moussa and Nagwa Badr: *Cluster-Based Test Cases Prioritization and Selection Technique for Agile Regression Testing*. Journal of Software: Evolution and Process, 29(6):e1794, 2017.

[LBE17] Linares-Vásquez, Mario, Gabriele Bavota and Camilo Escobar-Velásquez: *An Empirical Study on Android-Related Vulnerabilities*. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 2–13, May 2017.

[LKO15] Lwakatare, Lucy Ellen, Pasi Kuvaja and Markku Oivo: *Dimensions of DevOps*. In Lassenius, Casper, Torgeir Dingsøyr and Maria Paasivaara (editors): *Agile Processes in Software Engineering and Extreme Programming*, Lecture Notes in Business Information Processing, pages 212–217, Cham, 2015. Springer International Publishing.

[MG11] Mell, Peter and Timothy Grance: *The NIST Definition of Cloud Computing*, September 2011.

[MKB+21] Mukhin, Vadym, Yaroslav Kornaga, Yurii Bazaka, Ievgen Krylov, Andrii Barabash, Alla Yakovleva and Oleg Mukhin: *The Testing Mechanism for Software and Services Based on Mike Cohn's Testing Pyramid Modification*. In *2021 11th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, volume 1, pages 589–595, September 2021.

[Mod21a] Modi, Ritesh: *CI/CD with Terraform*. In Modi, Ritesh (editor): *Deep-Dive Terraform on Azure: Automated Delivery and Deployment of Azure Solutions*, pages 163–190. Apress, Berkeley, CA, 2021.

[Mod21b] Modi, Ritesh: *Terraform Unit Testing.* In Modi, Ritesh (editor): *Deep-Dive Terraform on Azure: Automated Delivery and Deployment of Azure Solutions*, pages 191–220. Apress, Berkeley, CA, 2021.

[Mor20] Morris, Kief: *Infrastructure as Code.* "O'Reilly Media, Inc.", December 2020.

[NAF21] Nass, Michel, Emil Alégroth and Robert Feldt: *Why Many Challenges with GUI Test Automation (Will) Remain.* Information and Software Technology, 138:106625, October 2021.

[Nun17] Nunez, Carlos: *Top 3 Terraform Testing Strategies for Ultra-Reliable Infrastructure-as-Code*, July 2017. Accessed: 2023-08-16, URL: `https://www.contino.io/insights/top-3-terraform-testing-strategies-for-ultra-reliable-infrastructure-as-code`.

[OZDR22] Opdebeeck, Ruben, Ahmed Zerouali and Coen De Roover: *Smelly Variables in Ansible Infrastructure Code: Detection, Prevalence, and Lifetime.* In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 61–72, Pittsburgh Pennsylvania, May 2022. ACM.

[OZDR23] Opdebeeck, Ruben, Ahmed Zerouali and Coen De Roover: *Control and Data Flow in Security Smell Detection for Infrastructure as Code: Is It Worth the Effort?* In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pages 534–545, May 2023.

[Rah18] Rahman, Akond: *Characteristics of Defective Infrastructure as Code Scripts in DevOps.* In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ICSE '18, pages 476–479, New York, NY, USA, May 2018. Association for Computing Machinery.

[Rah19] Rahman, Akond: *Anti-Patterns in Infrastructure as Code.* PhD thesis, North Carolina State University, Raleigh, North Carolina, 2019.

[RBM21] Rahman, Akond, Farhat Lamia Barsha and Patrick Morrison: *Shhh!: 12 Practices for Secret Management in Infrastructure as Code.* In *2021 IEEE Secure Development Conference (SecDev)*, pages 56–62, October 2021.

[RFPW20] Rahman, Akond, Effat Farhana, Chris Parnin and Laurie Williams: *Gang of Eight: A Defect Taxonomy for Infrastructure as Code Scripts.* In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, pages 752–764, New York, NY, USA, October 2020. Association for Computing Machinery.

[RMW19] Rahman, Akond, Rezvan Mahdavi-Hezaveh and Laurie Williams: *A Systematic Mapping Study of Infrastructure as Code Research.* Information and Software Technology, 108:65–77, April 2019.

[Roc13] Roche, James: *Adopting DevOps Practices in Quality Assurance.* Commun. ACM, 56(11):38–43, November 2013.

[RPW19] Rahman, Akond, Chris Parnin and Laurie Williams: *The Seven Sins: Security Smells in Infrastructure as Code Scripts.* In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 164–175, May 2019.

[RS22] Rahman, Akond and Tushar Sharma: *Lessons from Research to Practice on Writing Better Quality Puppet Scripts.* In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 63–67, March 2022.

[RW19] Rahman, Akond and Laurie Williams: *Source Code Properties of Defective Infrastructure as Code Scripts.* Information and Software Technology, 112:148–163, August 2019.

[SC] Samonas, Spyridon and David Coss: *THE CIA STRIKES BACK: REDEFINING CONFIDENTIALITY, INTEGRITY AND AVAILABILITY IN SECURITY.*

[Sea22] Seaton, Will: *What Is Policy as Code? Definition and Benefits*, November 2022. Accessed: 2023-09-21, URL: `https://www.styra.com/blog/what-is-policy-as-code-definition-and-benefits/`.

[SF23] Saavedra, Nuno and João F. Ferreira: *GLITCH: Automated Polyglot Security Smell Detection in Infrastructure as Code.* In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, pages 1–12, New York, NY, USA, January 2023. Association for Computing Machinery.

[SLSS07] Spillner, Andreas, Tilo Linz, Hans Schaefer and Andreas Spillner: *Software Testing Foundations: A Study Guide for the Certified Tester Exam ; Foundation Level - ISTQB Compliant.* Rockynook, Santa Barbara, Calif, 2. ed edition, 2007.

[SR19] Surbiryala, Jayachander and Chunming Rong: *Cloud Computing: History and Overview.* In *2019 IEEE Cloud Summit*, pages 1–7, Washington, DC, USA, August 2019. IEEE.

[Ste15] Stella, Josh: *An Introduction to Immutable Infrastructure*, June 2015. Accessed: 2023-08-20, URL: `https://www.oreilly.com/radar/an-introduction-to-immutable-infrastructure/`.

[Vir18] Virtanen, Tuukka: *Literature Review of Test Automation Models in Agile Testing.* Kirjallisuuskatsaus testiautomaatiomalleista ketterässä ohjelmistotestauksessa, 2018.

[Wag08] WAGNER, STEFAN: *Defect Classification and Defect Types Revisited.* In *Proceedings of the 2008 Workshop on Defects in Large Software Systems*, pages 39–40, Seattle Washington, July 2008. ACM.

[Wan21] WANG, ROSEMARY: *Testing HashiCorp Terraform*, March 2021. Accessed: 2023-08-16, URL: `https://www.hashicorp.com/blog/testing-hashicorp-terraform`.

[Wan22] WANG, ROSEMARY: *Infrastructure as Code, Patterns and Practices: With Examples in Python and Terraform.* Simon and Schuster, September 2022.

[XZLZ13] XIA, XIN, XIAOZHEN ZHOU, DAVID LO and XIAOQIONG ZHAO: *An Empirical Study of Bugs in Software Build Systems.* In *2013 13th International Conference on Quality Software*, pages 200–203, July 2013.

[ZMI08] ZAMLI, KAMAL and NOR ASHIDI MAT ISA: *JTst – An Automated Unit Testing Tool for Java Program.* American Journal of Applied Sciences, 5, February 2008.