UNIVERSITY OF ST. GALLEN

School of Management, Economics,
Law, Social Sciences, International Affairs and Computer Science

Master Thesis

Enhancing Early Verification and Testing for Infrastructure as Code

Christoph Bühler
University of St. Gallen
St. Gallen, Switzerland
christoph.buehler@unisg.ch
Matriculation number: 10-152-882

Supervisor: **Prof. Dr. Guido Salvaneschi**
University of St. Gallen
St. Gallen, Switzerland
guido.salvaneschi@unisg.ch

Co-Supervisor: **Dr. Pascal Weisenburger**
University of St. Gallen
St. Gallen, Switzerland
pascal.weisenburger@unisg.ch

School of Computer Science
University of St. Gallen
Switzerland
June 27, 2025

Abstract

Infrastructure as Code (IaC) has become essential in managing modern cloud environments, yet misconfigurations remain a significant risk, often leading to costly outages and security breaches. This thesis introduces a novel static analysis approach for verifying network-level connectivity in Terraform-based IaC before deployment. By transforming declarative configurations into abstract graph models, the method enables early detection of unreachable services and misconfigurations without requiring infrastructure provisioning. A proof-of-concept tool, evaluated on the TerraDS dataset, demonstrated strong scalability and effectiveness in real-world scenarios. The approach integrates seamlessly into DevOps pipelines, enhancing existing IaC testing workflows. Looking ahead, the method offers potential for broader verification across multi-cloud environments, including cost and energy efficiency assessments.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Cloud Computing and Infrastructure as Code

Cloud computing has rapidly become a fundamental component of organizational innovation across various sectors, including businesses, governments, and industry. Its adoption enables organizations to dynamically provision computing resources without substantial upfront capital investments, thus revolutionizing traditional IT infrastructure paradigms [20, 41]. The growing reliance on cloud solutions has necessitated new methods for efficiently managing and deploying complex infrastructures, leading to the adoption of Infrastructure as Code (IaC).

Infrastructure as Code represents a significant advancement in IT infrastructure management by employing code-based (or rather: machine-readable) automation for defining, provisioning, and managing cloud resources [31]. By applying traditional software engineering practices to infrastructure management, IaC fosters consistency, version control, reproducibility, and collaboration among developers and operations teams [4]. This approach streamlines infrastructure deployments, reduces manual setup procedures, and improves overall system reliability and compliance.

1.2 Challenges of Infrastructure as Code Misconfigurations

Despite its numerous advantages, IaC brings its own set of challenges, notably related to misconfigurations. IaC configurations typically encapsulate reusable code components, promoting efficient reuse, collaboration, and rapid deployment across environments. However, this reusability can inadvertently amplify the impact of misconfigurations: a single faulty configuration can propagate quickly and affect multiple deployment environments simultaneously [31, 4].

Misconfigurations in IaC scripts share the same negative consequences as direct configuration errors within live cloud environments. A seemingly minor error in an IaC script – such as incorrect firewall rules, security group settings, or network access controls – can lead to significant system outages, vulnerabilities to cyberattacks, data breaches, and costly compliance violations [34, 37]. For example, incorrect network configurations might inadvertently expose internal resources to unauthorized external access, resulting in data leaks or unauthorized system access. Similarly, wrong resource allocation or incorrect dependencies between infrastructure elements could disrupt critical services, affecting both organizational operations and customer satisfaction.

For instance, in 2017, a misconfigured Amazon S3 bucket by Booz Allen Hamilton exposed sensitive data, highlighting the risks associated with improper access controls in cloud storage

services [16]. Similarly, in 2023, Microsoft experienced a disruption in its Exchange Online services due to a configuration glitch, underscoring the potential operational impacts of misconfigurations in cloud environments [30].

Another notable example is the 2019 incident involving *AutoClerk*, a travel reservations platform, where a misconfigured database led to the exposure of sensitive information of U.S. government (military) personnel [42]. These incidents exemplify how misconfigurations in IaC can have far-reaching consequences, affecting not only organizational operations but also national security. As organizations increasingly adopt IaC for managing their infrastructures, they must implement robust validation mechanisms to detect and rectify misconfigurations before deployment.

1.3 Limitations of Current Verification Tools

While tools such as "Checkov" [13] provide basic static analysis capabilities to identify configuration errors, these tools often fall short in fully capturing the complexity of real-world scenarios. Therefore, organizations still face significant risks and often resort to resource-intensive, iterative trial-and-error deployments, substantially increasing operational overhead and vulnerability periods. In the example of Checkov, the tool can provide basic analysis, but fails to check the full abstract syntax tree of the infrastructure as code program. Thus, it cannot check certain errors but only check if some conditions are met. One of the found issues is – as example – if a security group (essentially firewall rules in Amazon Web Services) contains a description that states what the group is for. This is only a very limited basic check to ensure the most basic level of software quality.

1.4 Research Objectives and Approach

This thesis addresses this critical gap by developing a proof of concept for automated verification approaches to enhance early-stage IaC testing. By implementing a verification tool that systematically checks critical properties – specifically for the PoC: network reachability between cloud instances – prior to deployment, developers can swiftly detect and correct misconfigurations, significantly minimizing the risk of outages, security breaches, and compliance failures [31, 4]. Through this proactive approach, the research contributes towards safer, more reliable cloud operations, and lays the groundwork for further studies in robust, pre-deployment verification techniques for declarative IaC solutions. The emphasis lies on pre-deployment since the deployment and then subsequent analysis of infrastructure is time-consuming and costly. The goal is to find the errors before the deployment, so that the deployment can be done in a more efficient way.

To guide this research, we formulated two key research questions:

- **RQ 1**: How applicable is the proposed approach to real-world IaC programs?
- **RQ 2**: How does the performance of the proposed approach compare to post-deployment tools?

These questions focus on evaluating both the practical applicability of our verification approach across diverse real-world IaC configurations and its performance characteristics compared to existing post-deployment solutions. The answers to these questions will help determine the viability and potential impact of pre-deployment verification in real-world cloud infrastructure management scenarios.

1.5 Thesis Structure

The remainder of this thesis follows this structure: Chapter 2 establishes the fundamental concepts and terminology used throughout this thesis. Chapter 3 examines related work in the field of infrastructure verification and automated testing, also, it provides an overview of the current state of the art in cloud infrastructure management and verification. Chapter 4 presents the conceptual framework for the proposed verification approach. Chapter 5 details the implementation of the proof of concept, including the technical architecture and key components. Chapter 6 evaluates the implemented prototype, Chapter 7 discusses its advantages, limitations and future work, and Chapter 8 summarizes the key findings and outlines directions for future research.

Chapter 2

Context and Background

Before diving into the technical contributions of this thesis, it is essential to establish a foundational understanding of the core concepts, technologies, and distinctions. This section provides a concise but comprehensive overview of the terminology and tools relevant to this work, aiming to clarify the context in which the proposed solution operates.

The concept of IaC (Infrastructure as Code) and its significance within modern DevOps and cloud engineering practices is introduced. Then, a distinction between IaC and IfC (Infrastructure *from* Code) shows how the two similar concepts are different. Following this, "Terraform" – a widely adopted IaC tool used throughout this thesis – is examined, detailing its language, workflow, and position within the IaC ecosystem. The section then distinguishes between configuration management and infrastructure management, two paradigms that play complementary roles in infrastructure automation. Lastly, an overview of Amazon Web Services (AWS), the cloud platform used in the proof of concept, is offered, with a focus on the infrastructure components most relevant to this research.

This background lays the groundwork for understanding the design choices, challenges, and objectives addressed in subsequent sections of the thesis.

2.1 Infrastructure as Code

Infrastructure as Code (IaC) is a practice within software engineering and DevOps paradigms [4], emphasizing the automation of infrastructure provisioning and management through code rather than manual processes. This approach treats infrastructure resources – such as servers, networks, and services – as software, enabling reproducibility, consistency, version control, and rapid deployment across environments. IaC allows teams to define, deploy, and manage infrastructure components systematically, promoting greater collaboration, rapid development cycles, and consistent operational environments [31]. Not to be confused with Infrastructure *from* Code (IfC), which is briefly introduced in Section 2.2.

Several tools facilitate the adoption and practice of IaC by organizations, each with distinct features and use cases:

Terraform Is a widely adopted IaC tool developed by HashiCorp that allows users to define infrastructure components across multiple cloud and on-premise environments using a high-level declarative language (HCL – HashiCorp Configuration Language [24]). It enables infrastructure automation, state management, and lifecycle management, supporting multiple cloud providers [26].

AWS CloudFormation Offered by Amazon Web Services (AWS), provides users with an AWS-specific service to describe and provision resources through JSON or YAML templates. It integrates deeply with other AWS services, offering automated and repeatable infrastructure provisioning tailored explicitly to the AWS cloud ecosystem [3].

Chef Is a configuration management tool developed by Progress Software Corporation that enables infrastructure automation through code. Chef employs a domain-specific language (Ruby-based DSL) for defining infrastructure configurations and enforcing the desired state. It is particularly effective for managing complex server configurations, deploying applications, and ensuring system consistency across multiple environments [32].

Ansible Developed by Red Hat, is an automation platform providing comprehensive capabilities to automate infrastructure, manage configurations, and streamline deployment workflows. It excels in configuration management and is known for its ability to handle highly dynamic environments through flexible, code-driven automation [35].

2.2 Infrastructure from Code

Infrastructure from Code (IfC) is an emerging paradigm in cloud infrastructure management that automates the provisioning and configuration of infrastructure directly from application code. Unlike Infrastructure *as* Code (see Section 2.1), which requires separate configuration files to define infrastructure, IfC tools analyze the application code to infer and generate the necessary infrastructure components automatically [14]. This approach streamlines the development process by reducing the need for manual infrastructure definitions and ensuring that the infrastructure aligns seamlessly with the application's requirements.

A notable example of IfC in practice is Nitric [5], an open-source framework designed to facilitate cloud application development. Nitric enables developers to declare infrastructure requirements within their application code using multi-language SDKs. During deployment, Nitric interprets these declarations to provision the appropriate cloud resources across various providers such as AWS, Azure, and Google Cloud. This integration allows developers to focus on writing application logic while Nitric manages the underlying infrastructure, thereby enhancing productivity and reducing the complexity associated with cloud deployments.

In summary, while both IaC and IfC aim to automate infrastructure management, they differ in their approaches and use cases. IaC offers detailed control through separate configuration files, making it suitable for complex infrastructures requiring precise customization. In contrast, IfC emphasizes automation and integration by deriving infrastructure directly from application code, which can enhance development efficiency, particularly in serverless and rapidly evolving environments.

2.3 HashiCorp Terraform

Terraform [26], developed by HashiCorp, is an open-source Infrastructure as Code tool designed to enable users to define and manage infrastructure resources across various cloud providers and on-premises environments using a consistent workflow. Terraform leverages the HashiCorp Configuration Language (HCL) [24], a declarative language that allows users to specify the desired state of infrastructure clearly and concisely. HCL enhances readability, maintainability,

and scalability of infrastructure definitions, contributing significantly to the efficiency of IaC practices.

```
1  provider "aws" {
2    region = "us-east-1"
3  }
4
5  resource "aws_vpc" "main" {
6    cidr_block = "10.0.0.0/16"
7  }
8
9  resource "aws_subnet" "main" {
10   vpc_id     = aws_vpc.main.id
11   cidr_block = "10.0.1.0/24"
12 }
```

Listing 1: Terraform Configuration Example in HCL

Listing 1 illustrates a simple Terraform configuration in HCL. At the beginning, the required provider – AWS[1] in this case – is configured to use the "us-east-1" region as default value. Then, the HCL configuration defines an AWS Virtual Private Cloud ("VPC") (lines 5–7) and a subnet (lines 9–12) within that VPC. One thing to note is, that in HCL, a whole object is called a block. The structure of the language contains blocks (`resource`), which may contain labels (`aws_vpc` and `main`) and attributes (`cidr_block`).

Terraform's declarative approach specifies the desired state rather than the steps required to achieve that state. It calculates and manages resource dependencies via directed acyclic graphs, automates provisioning, and maintains state consistency through its planning and execution phases. Terraform is renowned for its robust ecosystem, offering extensive provider support, community modules, and integration capabilities, thus facilitating flexible, reliable, and automated infrastructure management.

A typical Terraform workflow, as depicted in Figure 2.1, involves multiple sequential steps initiated by the user. Initially, the user writes configuration files defining the infrastructure, then initializes Terraform (to download the required providers). Then, the user authenticates with the cloud provider's API. Terraform validates the configuration, and generates an execution plan based on the current infrastructure state. After reviewing the displayed plan, the user applies the deployment, triggering Terraform to provision resources via the cloud provider's API. The cloud provider subsequently creates the defined resources within the infrastructure, and Terraform confirms the successful completion of the deployment process.

Recent licensing changes by HashiCorp, transitioning to the Business Source License (BUSL), have sparked community-driven initiatives such as OpenTofu. OpenTofu is an open-source fork aimed at preserving an open-source infrastructure as code solution, adhering closely to Terraform's original philosophy and maintaining full compatibility with existing Terraform configurations [25, 17].

[1] Amazon Web Services

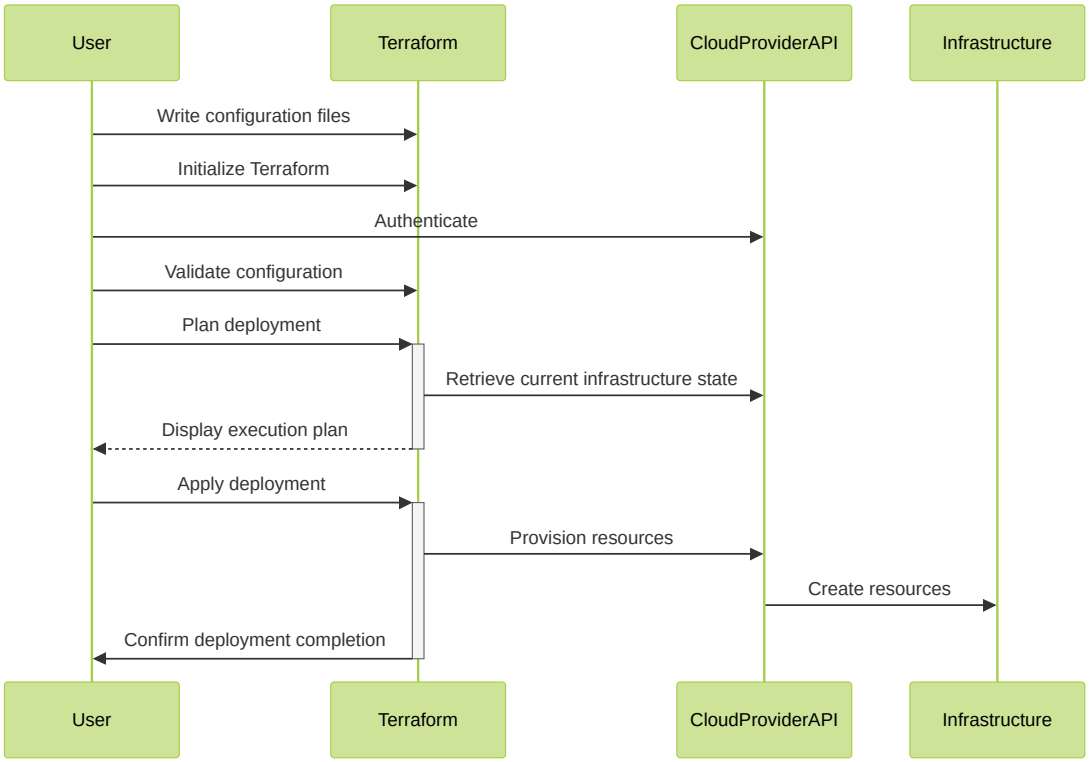

Figure 2.1: Terraform Engineering Workflow

2.4 Difference between Configuration and Infrastructure Management

Understanding the distinction between configuration management and infrastructure management is crucial in effectively leveraging IaC tools. Although both are closely related and often intertwined, they address different aspects of infrastructure automation and maintenance.

Configuration management focuses on maintaining and managing the state of software and configurations within existing infrastructure. It ensures consistency and integrity across system deployments, software installations, and configuration settings. Tools like Ansible [35] belong into this category by managing the internal state of servers, virtual machines, or containers. Ansible uses a procedural approach, where specific tasks and steps are explicitly defined. For example, Ansible can automate tasks such as installing software packages, applying security patches, and configuring application settings across multiple servers, ensuring uniformity and compliance with organizational standards.

On the other hand, infrastructure management involves provisioning and managing infrastructure resources themselves, typically at a higher abstraction level. Infrastructure management tools, such as Terraform [26] (see Section 2.3), deal with the creation, modification, and lifecycle management of entire infrastructures – such as cloud resources, networking, and storage – across various providers. Terraform adopts a declarative approach, as shown in Listing 1, where users specify the desired end state of infrastructure, and Terraform manages the intermediate steps to reach that state. For example, Terraform can provision a complete cloud environment, including virtual machines, networking resources, load balancers, and databases,

maintaining a consistent state and handling dependencies among these components automatically.

The choice between configuration management (e.g. Ansible) and infrastructure management (e.g. Terraform) is driven by specific use cases and requirements. Typically, Terraform is utilized to set up and manage infrastructure resources across cloud providers, whereas Ansible is used after the provisioning process to configure and maintain software and system settings on the deployed resources. In practice, these tools are often used in conjunction to achieve comprehensive infrastructure automation: Terraform provisions the infrastructure, while Ansible ensures the desired configuration within that infrastructure.

2.5 Amazon Web Services

Amazon Web Services (AWS), developed by Amazon Inc., is a comprehensive, evolving cloud computing platform that offers a broad set of infrastructure services. Examples are cloud storage, databases, analytics, networking, mobile computing, and enterprise applications [28]. Launched in 2006, AWS pioneered the cloud computing industry and has since maintained a dominant position, being widely recognized for its scalability, reliability, and global reach. AWS operates across numerous regions globally[2], allowing users to distribute applications geographically for redundancy and lower latency [22].

AWS operates on a pay-as-you-go pricing model, enabling users to scale infrastructure up or down according to real-time demand, thereby significantly reducing initial capital investment and ongoing maintenance costs. AWS offers various types of cloud computing services, primarily categorized into Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). AWS's infrastructure enables companies to rapidly deploy computing resources without significant upfront investment, fostering agility and innovation [22].

Core AWS components include Amazon EC2 for virtual servers, Amazon S3 for storage, and services like Amazon RDS for database management. AWS also provides load balancing services, such as Elastic Load Balancing, to automatically distribute incoming application traffic across multiple targets, enhancing availability and fault tolerance. Moreover, AWS incorporates advanced security features, including security groups functioning as network firewalls, allowing fine-grained access control to cloud resources [22].

Due to its broad service offerings, advanced features, and strong security infrastructure, AWS is extensively used by prominent enterprises worldwide, including Netflix, Facebook, Adobe, and Twitter, demonstrating its versatility and capability to support complex, large-scale operations effectively [22].

Since AWS is still the most used cloud provider in conjunction with Terraform [9], the proof of concept in this thesis is based on AWS. The focus lies on the network reachability between two EC2 instances, which are virtual machines in AWS.

[2] By the time of writing, AWS spans 114 availability zones with 36 geographic regions [28]

2.6 Static Analysis and Model Checking

Static analysis is a method for understanding program behavior without executing the code, relying instead on mathematical models of program semantics. As detailed by Rival and Yi [36], static analysis frameworks typically use abstraction to construct a computable representation of possible runtime behaviors, allowing developers to reason about potential errors, security vulnerabilities, or policy violations before deployment. The abstract interpretation framework, which underpins many static analyzers, provides a sound (but incomplete) way to over-approximate program behavior. This is particularly beneficial for analyzing infrastructure-as-code (IaC), where the goal is not to execute the infrastructure, but to ensure its correctness and security properties prior to provisioning.

Model checking offers another influential approach to verifying system correctness through exhaustive exploration of finite-state models. Clarke, Grumberg, and Long [12] introduced abstraction techniques to make model checking scalable to large systems, addressing the state explosion problem. Their work demonstrates how abstractions can be used to conservatively approximate the system's state space, allowing verification of safety properties even when a complete analysis is infeasible. This concept is especially relevant in cloud infrastructure scenarios, where a static tool can simulate the potential interactions between resources to detect misconfigurations or unreachable services. These insights support the use of abstract, graph-based models in the thesis's proposed approach to statically analyzing network reachability within Terraform configurations.

2.7 Challenges in Infrastructure as Code

Infrastructure as Code (IaC) has significantly transformed IT infrastructure management by automating the deployment and management of complex infrastructure. However, the paradigm shift toward IaC introduces several critical challenges. Foremost: the issue with misconfigurations that happen. IaC (e.g. Terraform), while reducing manual intervention, still relies on accurate code representation of the desired infrastructure state, as described in Section 2.3. Consequently, misconfigurations in IaC scripts can propagate rapidly and cause operational disruptions, outages, and significant security vulnerabilities [37, 1]. Without proper testing tools, it is nearly impossible to ensure that the growing infrastructural code in a project stays correct.

Another prominent issue in IaC practices is the prevalence of "code smells," poor coding practices that reduce the maintainability, readability, and correctness of IaC scripts. Empirical research highlights the negative impact of these smells, demonstrating their frequent correlation with higher defect rates and increased maintenance efforts [8]. Addressing such challenges requires rigorous code review practices and automated detection methods, yet existing solutions provide limited comprehensive coverage.

Moreover, the complexity in modern cloud-native architectures fosters these issues. IaC typically – at least in cloud environments – interacts with highly distributed, containerized, and microservice-based applications. This complicates dependency management and increases the risk of configuration drift[3]. Ensuring resilience, scalability, and elasticity becomes significantly

[3] Configuration drift happens when users change the environment outside the IaC program and thus the current state does not match the last applied state anymore

more challenging in such dynamic, interconnected environments [11].

Automation, while essential in Site Reliability Engineering (SRE) to maintain high availability and reduce downtime, introduces additional complexity in cloud infrastructure management. Automation involves complex orchestrations and dependencies, requiring careful integration with existing infrastructure, meticulous maintenance, and ongoing management to prevent failures [2]. As the infrastructure scales, ensuring seamless integration and consistency across automated workflows becomes increasingly difficult, highlighting the need for robust and intelligent verification and testing tools.

Finally, existing static analysis tools, such as "Checkov," [13] offer valuable but insufficient capabilities for comprehensive IaC validation. These tools typically handle common and straightforward misconfigurations but struggle with complex, dynamic infrastructure dependencies and interactions. Consequently, organizations often rely on resource-intensive iterative testing or manual reviews, increasing operational overhead and the potential for human error [7].

Thus, addressing these critical challenges through advanced automated verification approaches remains an essential goal for improving the robustness, reliability, and security of IaC-managed cloud environments.

Chapter 3

State of the Art

This thesis addresses the critical gap of lacking pre-deployment reachability verification for Infrastructure as Code (IaC). The following chapter surveys the current landscape of research and industrial tooling related to Infrastructure as Code (IaC). Given the critical importance of reproducibility, reliability, and correctness in the management of cloud-based infrastructures, the need for automated testing, analysis, and verification of IaC scripts has grown significantly [38, 7, 6]. The state of the art contains efforts from both academia and industry, spanning topics such as static analysis, dynamic testing, and formal verification. However, as we will demonstrate, most existing tools and approaches provide only partial support or operate post-deployment. This limitation impedes early detection of critical issues such as unreachable resources, misconfigurations, and faulty access rules. Ultimately, this leads to increased costs and operational risks. This chapter derives advancements across several subdomains and identifies where further research and development is needed. Building on these findings, Chapter 4 will present our conceptual framework for addressing these gaps through a novel pre-deployment verification approach.

3.1 Industry Adoption and Practitioner Challenges

Guerriero et al. [21] investigate the practical adoption and challenges associated with Infrastructure as Code through empirical research involving 44 semi-structured interviews with senior developers from diverse companies. Their findings shed light on the industry's current practices, the variety of available tools, and significant challenges that practitioners encounter.

The study highlights issues such as the lack of comprehensive automated testing frameworks, difficulties in managing readability, consistency, and portability of IaC scripts, and the critical need for improved tooling to support maintenance and evolution. Practitioners emphasize the necessity of enhanced testing practices and robust tooling to address these challenges effectively. Additionally, the authors discuss organizational barriers, such as skill gaps, cultural resistance to automation, and the complexity involved in transitioning from traditional manual processes to IaC-based automated workflows. These insights underscore the urgent need for further research and innovative solutions [21] to overcome these barriers and improve the quality, maintainability, and adoption of IaC, directly correlating with the current project's objectives to enhance automated verification and testing strategies.

3.2 Static Analysis of Infrastructure as Code

3.2.1 Overview of Static Analysis Approaches

Chiari, De Pascalis, and Pradella [10] present a comprehensive survey on static analysis techniques applied to Infrastructure as Code, highlighting the growing necessity for rigorous verification due to the potential defects that IaC scripts may contain, which can lead to significant security and reliability issues. The authors systematically categorize existing static analysis approaches into two main groups: syntactic-based techniques, including code smell detection and machine learning, and semantic-based methods employing formal verification and model checking [10].

Their review identifies common defect categories such as security vulnerabilities, idempotency issues, dependency errors, and improper configurations. This systematic categorization reveals that, despite advancements, current static analysis tools still face limitations in adequately addressing the complex and dynamic nature of cloud infrastructures. Additionally, the authors explore the effectiveness and limitations of current tools, emphasizing that more sophisticated semantic analysis techniques are necessary to capture context-sensitive issues accurately. Chiari, De Pascalis, and Pradella [10] advocate for an integrated approach combining syntactic and semantic methods to enhance the overall accuracy and reliability of pre-deployment verifications, aligning closely with the present project's aim to improve automated early verification methods in IaC.

3.2.2 Checkov: Rule-Based Security Analysis

One of the possibilities to statically analyze IaC is *Checkov*. Checkov is an open-source static analysis tool developed by Cloud [13] that scans Infrastructure as Code (IaC) configurations to detect misconfigurations before deployment. It supports a variety of IaC frameworks, including Terraform, CloudFormation, Kubernetes, Helm, ARM Templates, and Serverless frameworks. By integrating Checkov into the development pipeline, teams can identify and remediate potential security and compliance issues early in the software development lifecycle, thereby enhancing the overall security posture of their cloud infrastructures [13].

One of Checkov's key features is its extensive library of over 750 predefined policies that align with industry standards such as the Center for Internet Security (CIS) benchmarks. Additionally, it allows users to create custom policies using Python or YAML, providing flexibility to enforce organization-specific security requirements. Checkov's command-line interface facilitates seamless integration into continuous integration and continuous deployment (CI/CD) pipelines, enabling automated scanning of IaC files. The tool also offers various output formats, including CLI, JSON, JUnit XML, and SARIF, to accommodate different reporting needs [13].

3.2.3 TerraMetrics: Quality Assessment for Terraform

Also, a more recent example from academia is *TerraMetrics* [6]. Begoug, Chouchen, and Ouni [6] introduce *TerraMetrics*[1], an open-source static analysis tool designed to assess the quality of Terraform configurations written in the HashiCorp Configuration Language. Recognizing the growing complexity and widespread use of Terraform in modern cloud deployments, the tool targets key software quality concerns by quantifying 40 structural and semantic metrics at the block level (see Section 2.3). TerraMetrics processes Terraform files by constructing their

[1] `https://github.com/stilab-ets/terametrics`

Abstract Syntax Trees (ASTs) and extracting detailed metrics that relate to aspects such as cyclomatic complexity, nesting depth, string values, references, and control flow constructs like loops and conditions. Compared to the early versions of checkov, this approach offers a more robust and semantically aware assessment compared to traditional regular-expression-based scanning.

The tool functions as a command-line application that accepts individual files, folders, or GitHub repositories as input. It outputs metrics in JSON format, making it well-suited for integration into CI/CD pipelines. TerraMetrics has demonstrated high precision (88%) and recall (97%) in measuring code understandability [6], with its cyclomatic complexity metric correlating well with manually assessed readability. As Begoug, Chouchen, and Ouni [6] argue, the lack of tools for analyzing Terraform scripts beyond basic security scanning (like checkov – the example above) has hindered deeper quality assessments. TerraMetrics thus fills a critical gap by enabling maintainability evaluations, code review support, and empirical research into Terraform code quality.

3.3 Testing Infrastructure as Code

3.3.1 Testing Practices and Methodologies

Hasan, Bhuiyan, and Rahman [23] examine testing practices specifically designed for Infrastructure as Code scripts. Utilizing open coding methodologies on internet artifacts, they identify critical testing practices that are essential for improving the quality of IaC scripts. Among the identified practices are behavior-focused test coverage, the regular use of automated testing, sandbox and remote testing environments, and the importance of avoiding coding antipatterns in test scripts.

The authors emphasize the significant prevalence of defects in IaC scripts and the necessity for systematic testing approaches to prevent costly incidents. Hasan, Bhuiyan, and Rahman [23] provide concrete examples of major incidents caused by IaC script defects, highlighting the potential financial and operational impacts these defects can have on organizations. The paper also explores perspectives from industry on testing challenges, suggesting that many professionals struggle with identifying suitable testing methodologies and lack adequate resources to implement them effectively. These insights show a clear gap in systematic testing methodologies and underscore the need for improved guidance, tools, and frameworks within the IaC space, aligning with the present project's focus on developing more rigorous pre-deployment testing and verification methods to enhance infrastructure reliability and security.

3.3.2 Terraform Native Testing Framework

With regard to two popular IaC tools, Terraform and Pulumi, we now analyze the state of the art in testing IaC.

In version 1.6, Terraform introduced a native testing framework that enables authors to validate module configurations without impacting existing infrastructure or state. This framework allows for the creation of both integration and unit tests directly within the HashiCorp Configuration Language. By default, tests provision real infrastructure, facilitating integration testing by applying configurations and verifying the resulting infrastructure.

Alternatively, setting the `command` attribute to `plan` within a `run` block allows for unit testing by validating logical operations without actual resource creation. Tests are defined in files with the `.tftest.hcl` or `.tftest.json` extensions and can include multiple `run` blocks, a

```
provider "aws" {
  region = "eu-central-1"
}

variable "bucket_prefix" {
  type = string
}

resource "aws_s3_bucket" "bucket" {
  bucket = "${var.bucket_prefix}-bucket"
}

output "bucket_name" {
  value = aws_s3_bucket.bucket.bucket
}
```

Listing 2: Terraform Test Example: `main.tf`

`variables` block, and `provider` blocks. This structured approach ensures that module updates do not introduce breaking changes, enhancing the reliability and maintainability of Terraform configurations [27]. Both parts of the example, Listing 2 and Listing 3, are directly taken from the official Terraform documentation [27].

3.3.3 Automated Configuration Testing (ACT) for Pulumi

Sokolowski, Spielmann, and Salvaneschi [39] present a novel methodology, Automated Configuration Testing (ACT), addressing the crucial, but underexplored, area of testing Infrastructure as Code programs. Recognizing the complexities inherent to IaC, particularly with programs written in general-purpose languages such as TypeScript or Python, the authors highlight the limited adoption of systematic testing approaches in this domain. Their study reveals – based on the PIPr dataset [40] – that less than 1% of public IaC programs on GitHub utilize systematic tests, underscoring the critical need for effective testing methodologies.

ACT introduces an automated approach designed to rapidly test numerous configurations with minimal manual intervention. It achieves this by automatically mocking resource definitions within IaC programs, supported by generator and oracle plugins (partially provided by the community) for efficient test case creation and validation. Implemented in the ProTI testing tool for Pulumi TypeScript, this framework demonstrates significant advantages over existing techniques, including enhanced bug detection capability, reduced testing time, and the ability to easily integrate third-party tools through a pluggable architecture.

The relevance of this study to the present project lies in its clear identification of a significant gap in IaC testing practices. By explicitly showcasing the necessity and benefit of automated, systematic testing for IaC programs, Sokolowski, Spielmann, and Salvaneschi [39] establish a foundational context for ongoing research into enhancing IaC verification methods. This aligns directly with the objectives of the present project, which seeks to further improve early-stage verification and testing frameworks to mitigate risks associated with infrastructure misconfigurations.

```
variables {
  bucket_prefix = "test"
}

run "valid_string_concat" {
  command = plan
  assert {
    condition     = aws_s3_bucket.bucket.bucket == "test-bucket"
    error_message = "S3 bucket name did not match expected"
  }
}
```

Listing 3: Terraform Test Example: `valid_string_concat.tftest.hcl`

3.4 Formal Verification of Infrastructure as Code

Formal verification of Infrastructure as Code programs remains an emerging and underdeveloped area of research. One notable contribution to this field is the work by Sokolowski and Salvaneschi [38], who present a vision for achieving greater reliability in modern IaC programs, particularly those written in general-purpose languages such as TypeScript through Pulumi [33]. Their proposal centers on ProTI (as also mentioned in Section 3.3.3), a tool designed to support automated unit testing and verification. While much of the paper focuses on fuzzing and property-based testing, it also introduces a path toward automated verification by embedding domain-specific specifications into IaC programs and leveraging resource plugin support from cloud providers. The vision includes verifying properties such as access paths or network reachability offline, prior to deployment, using formal backends such as SMT solvers (e.g., Z3[2] or MonoSAT[3]). This demonstrates a promising direction but is still largely conceptual, with initial prototyping rather than mature, large-scale implementations.

Another concrete step towards formal verification in IaC is the master's thesis by De Pascalis [15]. The author develops and evaluates DOML-MC, a prototype model checker tailored for the DevOps Modelling Language (DOML), which is part of the EU-funded PIACERE project. This tool translates infrastructure descriptions into Satisfiability Modulo Theories (SMT) problems and uses the Z3 solver to verify properties about infrastructure topology and configuration. The approach is grounded in static analysis and emphasizes structural correctness, such as ensuring that interconnected components can communicate or validating dependency constraints. While limited in scope and maturity, the thesis shows that SMT-based verification of IaC models is technically feasible. Nonetheless, the thesis also underscores the lack of widespread tool support or integration with mainstream IaC workflows, highlighting that formal verification in IaC is still in its infancy.

[2] https://github.com/Z3Prover/z3

[3] https://github.com/sambayless/monosat

3.5 Post-Deployment Network Analysis

3.5.1 AWS Reachability Analyzer

The AWS Reachability Analyzer is a network diagnostics tool designed to verify network paths between two resources within an Amazon Virtual Private Cloud (VPC). It enables users to determine whether a network packet originating from one resource (e.g., a network interface of an EC2 instance, or an internet gateway), can reach another resource. The tool considers the applied routing, security groups, and network access control list (ACL) configurations – among other elements that could influence networking on AWS. This tool is particularly valuable for validating and troubleshooting network connectivity within and across VPCs, as it models the data plane rather than relying on live packet delivery, making it non-intrusive and safe for production environments [29].

Functionally, the Reachability Analyzer creates a virtual representation of the user's network configuration and performs reachability analysis by tracing potential paths between selected source and destination resources. The tool analyzes influencing factors such as route tables, as mentioned before, to determine if a network flow is permitted. If a valid path exists, the tool provides a detailed route highlighting each hop in the network, otherwise it explains where the traffic is blocked. Additionally, the analysis results can assist in auditing security configurations and identifying misconfigurations or overly permissive rules that could lead to security risks. The analyzer supports both *intra*-VPC and *inter*-VPC scenarios, including multi-account environments, and integrates with AWS Organizations to simplify cross-account visibility [29].

3.6 Empirical Datasets for IaC Research

3.6.1 TerraDS: A Dataset for Terraform HCL Programs

The TerraDS dataset, introduced by Bühler et al. [9], provides a comprehensive collection of Terraform HCL (HashiCorp Configuration Language) [24] programs designed to address the absence of large-scale datasets in the Infrastructure as Code research community. The dataset comprises publicly accessible Terraform repositories from GitHub, totaling 62,406 repositories after extensive filtering based on licensing and repository validity. These repositories collectively include 279,344 Terraform modules and 1,773,991 resources, making TerraDS the most extensive publicly available dataset specifically tailored for Terraform and IaC research.

TerraDS is structured to support diverse research applications, including static analysis, vulnerability detection, and best practice identification. Bühler et al. [9] demonstrated the dataset's usefulness through metadata analyses – such as distribution patterns of modules per repository and lines of code – and by applying static analysis tools like Checkov to reveal common security violations and issues in Terraform scripts.

The availability of TerraDS significantly advances research potential by providing a substantial empirical basis to study Terraform configurations. Researchers can leverage this dataset to develop improved static analysis tools, conduct in-depth vulnerability assessments, and explore best practices in infrastructure automation.

TerraDS is also the data foundation of this thesis. The available data allows for evaluation and testing of the implemented tool against real world problems and programs. Instead of searching for suitable programs, the TerraDS dataset can be used to evaluate the tool's performance and effectiveness.

3.6.2 The PIPr Dataset of Public Infrastructure as Code Programs

The PIPr dataset, introduced by Sokolowski, Spielmann, and Salvaneschi [40], addresses a critical gap in Infrastructure as Code research by providing an extensive collection of publicly available Pulumi programs. Pulumi [33] is an IaC framework that enables users to define and manage cloud infrastructure through general-purpose programming languages, offering flexibility and integration with existing software development practices.

The PIPr dataset contains Pulumi programs sourced from public repositories, systematically collected and curated to support empirical studies and tool development within the IaC research community. Specifically, the dataset includes a wide variety of infrastructure configurations, resource definitions, and metadata to facilitate analyses, including static code analysis, vulnerability detection, and best practice evaluation.

In relation to this project, PIPr [40] closely parallels the TerraDS [9] dataset; however, while TerraDS focuses on Terraform programs written in HCL, PIPr specifically targets Pulumi programs authored in general-purpose programming languages such as TypeScript, Python, and Go. Both datasets provide critical infrastructure data resources, though targeting different IaC tools, to enable rigorous testing, verification, and analysis. This similarity underscores their complementary roles in advancing research into automated verification methods for IaC solutions across diverse tooling ecosystems.

3.7 Summary and Identified Gaps

The reviewed body of work demonstrates that while there has been substantial progress in addressing the challenges of Infrastructure as Code, significant limitations persist. Static analysis tools such as Checkov and TerraMetrics help detect syntactic and semantic issues before deployment. But, they are often limited to rule-based mechanisms and do not handle complex resource interactions or dynamic infrastructure behavior. TerraMetrics improves older tools by incorporating deeper quality metrics, yet it also only focuses on code understandability and maintainability, leaving functional validation out of scope.

Testing practices in IaC remain an open challenge. Terraform's built-in test framework and academic contributions like the ACT model for Pulumi offer valuable approaches to verifying behavior pre-deployment, but they rely on mocking cloud APIs, which introduces additional complexity. Moreover, these approaches require infrastructure definitions to be tested in isolation or through sandboxed environments, limiting their coverage of real-world production scenarios. Testing is still largely ad hoc or dependent on integration environments.

Efforts in formal verification, such as the work by De Pascalis [15] and Sokolowski and Salvaneschi [38], are notable yet only not that developed as they could be. These approaches aim to apply formal methods, such as property-based testing and SMT solving, to validate IaC configurations. However, their adoption is limited due to prototype status, lack of integration with existing development workflows, and the high expertise barrier required to define formal specifications.

Finally, industrial tools – such as the AWS Reachability Analyzer – provide robust, visual diagnostics for network paths within cloud environments. However, these tools function only after infrastructure has been deployed. As such, they are not suitable for catching reachability or configuration issues before deployment.

In summary, the current ecosystem lacks a comprehensive solution that allows for expressive, pre-deployment verification of key infrastructure properties such as network reachability. The ability to statically analyze and verify the effects of a change before infrastructure is provisioned

remains out of reach. There is a clear need for a verification tool that integrates into the IaC development process, operates without full deployment, and offers guarantees about properties such as connectivity and access control. Such a tool would reduce iteration cycles, prevent costly outages, and contribute to more secure and reliable cloud infrastructure management.

Having reviewed the capabilities and shortcomings of current static and dynamic verification tools, it becomes evident that existing approaches struggle to provide scalable, provider-independent, and pre-deployment-compatible solutions for reachability analysis. The limitations discussed, such as constrained protocol coverage, reliance on deployed infrastructure, or lack of extensibility, reinforce the need for a tool that can operate early in the development lifecycle, leveraging only the declarative configuration. The next section introduces a conceptual framework and proof-of-concept implementation designed to directly address these challenges. It outlines the specific goals, assumptions, and design choices that underpin a novel static analysis approach to verifying network connectivity in Terraform-based cloud infrastructures.

Chapter 4

Analysis of Infrastructure as Code

This section presents the conceptual foundation for performing static analysis on Infrastructure as Code (IaC) programs, with a particular focus on Terraform. Given the dynamic and declarative nature of IaC, traditional static analysis methods face substantial challenges; especially when reasoning about dependencies, references, and resource behaviors that are typically resolved only at runtime. We begin by exploring the challenges of dependency management in IaC tools, where resources reference each other through dynamically resolved attributes. Following that, we define the functional and non-functional requirements that guide the design of the analysis tool, before introducing the concept and scope of the intended proof of concept. The central goal is to enable pre-deployment validation of properties such as network reachability, using only the static program definitions – without relying on provider APIs or runtime state – thus, supporting more reliable and efficient cloud infrastructure engineering.

4.1 The Challenge of Dependencies

Terraform [26] operates by first constructing a directed acyclic graph (DAG) that models the dependencies between all defined resources. This graph ensures that resources are created, modified, or destroyed in an order that respects their interdependencies. Once the DAG is constructed, Terraform proceeds through its execution plan by traversing the graph and making API calls to the respective cloud provider(s) to create the required infrastructure components. Each resource is provisioned only after its dependencies have been satisfied. As each step in the plan is executed, the results of these API interactions – such as dynamically generated values like IP addresses or identifiers – are stored in a state file. This state acts as a source of truth for subsequent operations, enabling references to the outputs or attributes of previously created resources within the configuration. This mechanism ensures consistency, reproducibility, and traceability across deployments.

The HCL code in Listing 4 results in the DAG shown in Figure 4.1. Thus, Terraform will first create the VPC, then the subnet, and finally the two EC2 instances. The arrows in the DAG indicate the dependencies between resources. The calculation is done via references inside the code blocks. It is obvious, that the subnet cannot be created before the VPC because it depends on the ID of the VPC. Also, the ID is only every available *after* the VPC has been created on the cloud API. Modelling this fact is hard when it comes to static analysis. Because the system should also work when no prior deployment has been done, there is no way to foretell the resulting ID of the VPC. Note that the same concept also applies for Pulumi.

```
provider "aws" {
  region = "us-east-1"
}

resource "aws_vpc" "main" {
  cidr_block = "10.0.0.0/16"
}

resource "aws_subnet" "sn_main" {
  vpc_id     = aws_vpc.main.id
  cidr_block = "10.0.1.0/24"
}

resource "aws_instance" "example_a" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
  subnet_id     = aws_subnet.sn:main.id
}

resource "aws_instance" "example_b" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
  subnet_id     = aws_subnet.sn_main.id
}
```

Listing 4: Terraform example - VPC, Subnet and two EC2 instances

As a result – shown in Figure 4.2 – the VPC is created first, then the subnet, and finally the two EC2 instances. Terraform will optimize the creation of the two instances since there have no further dependencies. This results in a parallel execution, if possible.

As the example above shows, there is a challenge in modeling the infrastructure locally. In the example of an `ID` field, the value can be artificially created on the fly while evaluation happens, but other values like IP addresses and such would require a more sophisticated mechanism, such as ProTI [39].

4.2 Goals and Non-Goals

This section outlines the functional and non-functional requirements that define the scope of this thesis and the accompanying proof of concept. The functional requirements describe the core behaviors the tool must support, such as analyzing Infrastructure as Code definitions and detecting potential network connectivity issues before deployment. The non-functional requirements capture essential quality attributes – such as performance, correctness, and generalizability – that ensure the tool can operate efficiently and reliably in a practical DevOps environment.

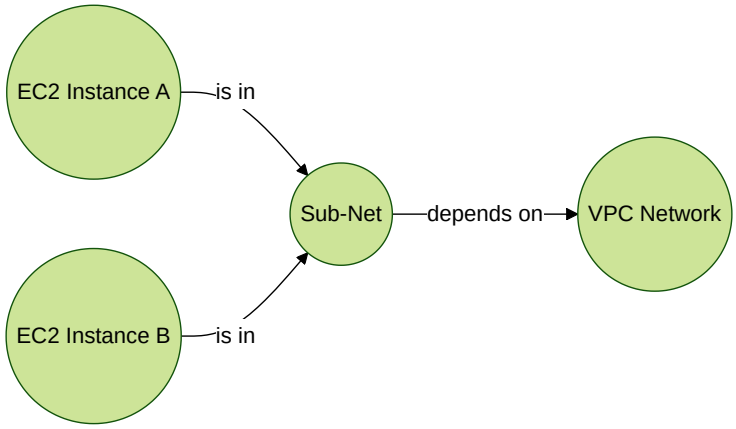

Figure 4.1: Terraform DAG Example

FRQ-1 The analysis must be able to run *pre-deployment*. This means that the analysis should be able to run before the IaC program is deployed to the cloud provider. This is important because it allows engineers to catch potential issues early in the development process, before they become more difficult and costly to fix. Further, this saves time and resources by preventing the need for rework or rollbacks after deployment.

FRQ-2 In the case of Terraform, the analyzer is able to perform the analysis *without executing Terraform* itself. Thus, it allows parsing of the HCL or JSON representation of the IaC program. This is important because it allows the analysis to be performed without requiring access to the cloud provider or the need to deploy any resources. This is particularly useful for teams that are working in a local development environment or that are using a CI/CD pipeline.

FRQ-3 The analysis *can be integrated into CI/CD* pipelines. This allows engineers to use the tool locally and in their automation settings. As such, teams can profit from automated tests and checks before they even deploy the infrastructural programs (in conjunction with FRQ 1).

FRQ-4 The analysis *supports the required resources* from the cloud provider to be able to fulfill its task. In the example of the Proof of Concept (see Section 4.3), the analysis must be able to parse and analyze all network related resources. This includes, but is not limited to, virtual machines, gateways, load balancers, and networks.

FRQ-5 The analyzer tool must allow (in the case of the PoC) *the specification of ports, protocols and the WAN*[1]. This allows specific checks for network reachability and also fosters reusability as a CLI tool.

NFRQ-1 The tool must provide *rapid feedback* to support seamless integration into CI/CD pipelines. High performance is essential to ensure that infrastructure changes can be tested and validated quickly without delaying the development or deployment process. Fast execution enables frequent testing cycles and promotes early detection of errors, aligning with agile and DevOps best practices.

[1] WAN: Wide Area Network; Internet

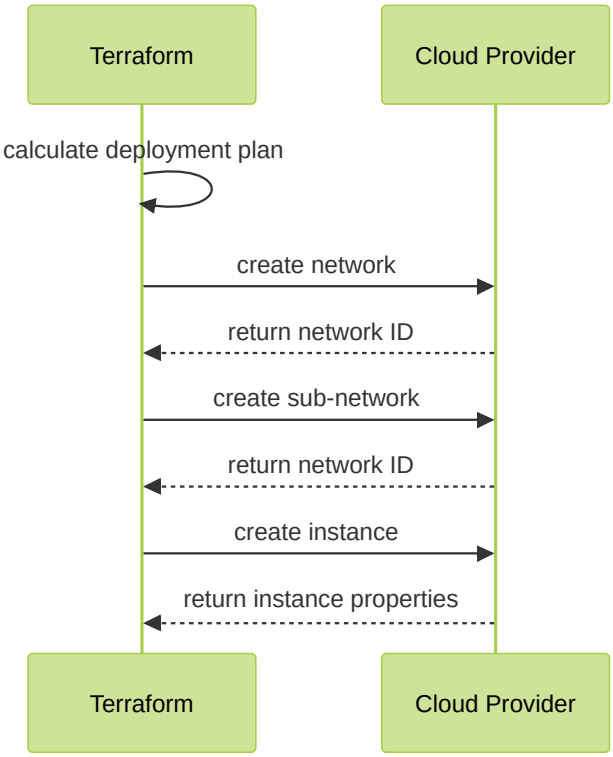

Figure 4.2: Terraform DAG Deployment

NFRQ-2 The analyzer must be designed in a *modular and extensible manner* so that its core concepts—such as parsing, modeling, and property evaluation – can be reused or adapted to check additional infrastructure properties beyond network reachability. This generalizability ensures long-term applicability and supports future use cases like verifying access control policies, resource dependencies, or cost-related constraints.

NFRQ-3 The analyzer must *yield results that are consistent* with those produced by established tools like the AWS Reachability Analyzer. Ensuring correctness is crucial for building trust in the tool's output; inaccurate assessments could lead to false confidence in flawed infrastructure configurations. Therefore, correctness serves as a foundational quality that underpins the reliability and adoption of the tool.

Summary In summary, the functional requirements of the project emphasize the need for an analysis tool that operates pre-deployment, works independently of cloud provider execution (e.g., without invoking Terraform), integrates seamlessly with CI/CD pipelines, and supports the parsing and understanding of relevant cloud resources. Additionally, the analyzer must accommodate protocol-, and port-specific inputs to enable detailed property checks, such as network reachability. Complementing these, the non-functional requirements underscore the importance of high performance for rapid feedback, modularity for generalizing the approach to other properties, and correctness to ensure the tool produces trustworthy results comparable to established verification services like AWS Reachability Analyzer. Collectively, these requirements define the scope and expectations of the proof of concept and its potential evolution into a general-purpose pre-deployment verification tool.

4.3 Proof of Concept: Network Reachability

The proposed Proof of Concept (PoC) introduces an offline network reachability analysis tool that emulates the core functionality of the AWS Reachability Analyzer [29]. In contrast to the AWS-native tool, which analyzes network connectivity only after the infrastructure is deployed, this PoC enables pre-deployment verification (FRQ 1). By statically analyzing infrastructure definitions – such as those written in Terraform (see Section 2.3) – the tool constructs an internal model of the virtual network topology. It then simulates communication paths between defined resources (e.g., virtual machines, gateways, load balancers, and networks) and identifies potential misconfigurations or blocked routes based on route tables, security groups, and other access control mechanisms. This proactive, static approach significantly reduces the risk of deployment-time outages by surfacing network-level errors early in the development cycle, without requiring actual infrastructure to be provisioned.

A key advantage of this approach lies in its seamless integration into Continuous Integration/Continuous Deployment (CI/CD) pipelines. By incorporating network reachability checks directly into the IaC validation workflow, the tool enables developers and DevOps engineers to detect issues immediately as part of their routine version control and testing processes [23, 39]. This capability addresses one of the major shortcomings in existing static analysis tools such as Checkov, which focus predominantly on syntactic and compliance checks but cannot reason about dynamic properties such as path reachability [13]. Similarly, while the ACT testing framework for Pulumi introduces testable infrastructure semantics through mocking [39], the proposed PoC offers a language-agnostic and cloud-agnostic alternative for early-stage verification. By bridging the gap between static analysis and post-deployment diagnostics, this PoC lays the groundwork for a scalable, pre-deployment verification solution that could save both time and cost in modern DevOps pipelines.

While network reachability is the central focus of this PoC, it is only one of many properties that can benefit from static, pre-deployment analysis. The underlying architectural approach is generalizable and can be extended to verify additional properties such as resource dependency correctness, IAM permission scopes, encryption settings, or compliance with architectural security patterns. These properties often span multiple configuration files and resource types, and detecting inconsistencies or violations before deployment would provide significant value. As infrastructure environments continue to grow in scale and complexity, the ability to validate a broad range of infrastructural and security properties offline becomes increasingly crucial for ensuring reliable, secure, and cost-effective cloud operations.

Chapter 5

System Design

This section presents the implementation of the proof of concept (PoC) developed for the automated validation of Infrastructure as Code (IaC) configurations. It provides a comprehensive overview of the design rationale, the selection of technologies, and the architectural principles that guided the development process. Furthermore, this section details the methods employed for parsing Terraform configurations, constructing and analyzing network topologies, and exposing the system's functionality through a command-line interface and supporting tools. By systematically outlining the boundaries, workflow, and auxiliary utilities of the PoC, this section establishes a clear understanding of the technical foundation and operational capabilities of the proposed solution.

5.1 The Go Programming Language

Go, often referred to as Golang, is a statically typed, compiled programming language developed by Google in 2009. Designed with simplicity, efficiency, and reliability in mind, Go offers a clean syntax that is easy to learn and use, making it an excellent choice for both beginners and experienced developers. Its straightforward design facilitates rapid development and maintenance of scalable software systems.

One of Go's significant advantages lies in its native support for concurrency, allowing developers to write efficient and scalable code for multicore and distributed systems. Additionally, Go's fast compilation times and cross-platform support enhance developer productivity and software portability [19].

Importantly, the HashiCorp Configuration Language (HCL, see Listing 1), which is the foundation of Terraform configurations, is implemented in Go[1][24]. This implementation choice simplifies the process of parsing and analyzing Terraform files within Go applications. Developers can leverage Go's robust standard library and the HCL Go API to decode HCL configurations directly into Go structures, facilitating seamless integration and manipulation of infrastructure code.

In summary, Go's simplicity, performance, and native compatibility with HCL make it an ideal language for developing tools that analyze and manage Infrastructure as Code programs. Its features align well with the requirements of static analysis tools aiming to provide pre-deployment verification of infrastructure configurations.

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

5.2 Overview of the PoC

The PoC demonstrates the feasibility of automatically validating infrastructure configurations as code. The primary goal of this PoC is to provide a working implementation that:

- Section 5.5: Parses Terraform configuration files to extract and structure key resources.
- Section 5.6: Constructs a network topology from the parsed data, modeling components such as virtual networks, subnets, instances, security groups, and routing elements.
- Section 5.7: Evaluates network reachability between interfaces using security group rules, route tables, and other connectivity elements.
- Section 5.8: Offers a command-line interface to trigger and display the results of the reachability analysis.
- Section 5.9: Utilizes complementary tools to enhance and support the analysis process.

By integrating these components, the PoC serves as a concrete example for validating properties of Infrastructure as Code setups. Detailed architecture, including individual modules, is discussed in subsequent sections.

5.3 Architecture of the PoC

The architecture of the proof of concept is designed with modularity and clear separation of concerns in mind. The overall structure follows Go best practices by leveraging two main directories:

- **internal**/: Contains packages that are private to the project and are not intended for external consumption. This folder includes core packages, such as:
  - `internal/reachability`: Implements the network reachability logic, including topology creation (see, e.g., `topology.go`) and interface rules (e.g., `interface.go`).
  - `internal/terraform`: Provides the resource parsing and configuration mapping from Terraform files (e.g., `subnet.go` and `vpc.go`).
  - `internal/cli`: Implements the command-line interface using Cobra[2][18], handling user interactions and command dispatch (see, for example, `reachability.go`).
- **pkg**/: Contains packages that are designed to be reusable and offer more generic functionality for external use. These include:
  - `pkg/parser`: Provides a generic interface for parsing Terraform sources and converting them to internal representations while enforcing the structure required by the PoC.
  - `pkg/reachability`: Exposes key constants and functions related to the network reachability analysis.

This clear separation ensures that internal implementation details remain encapsulated within the project, while common functionality that might be reused in other projects is available in the `pkg` directory. The design emphasizes modularity, testability, and adherence to Go conventions, thus supporting maintainability and potential future extensions.

[2] `https://github.com/spf13/cobra`

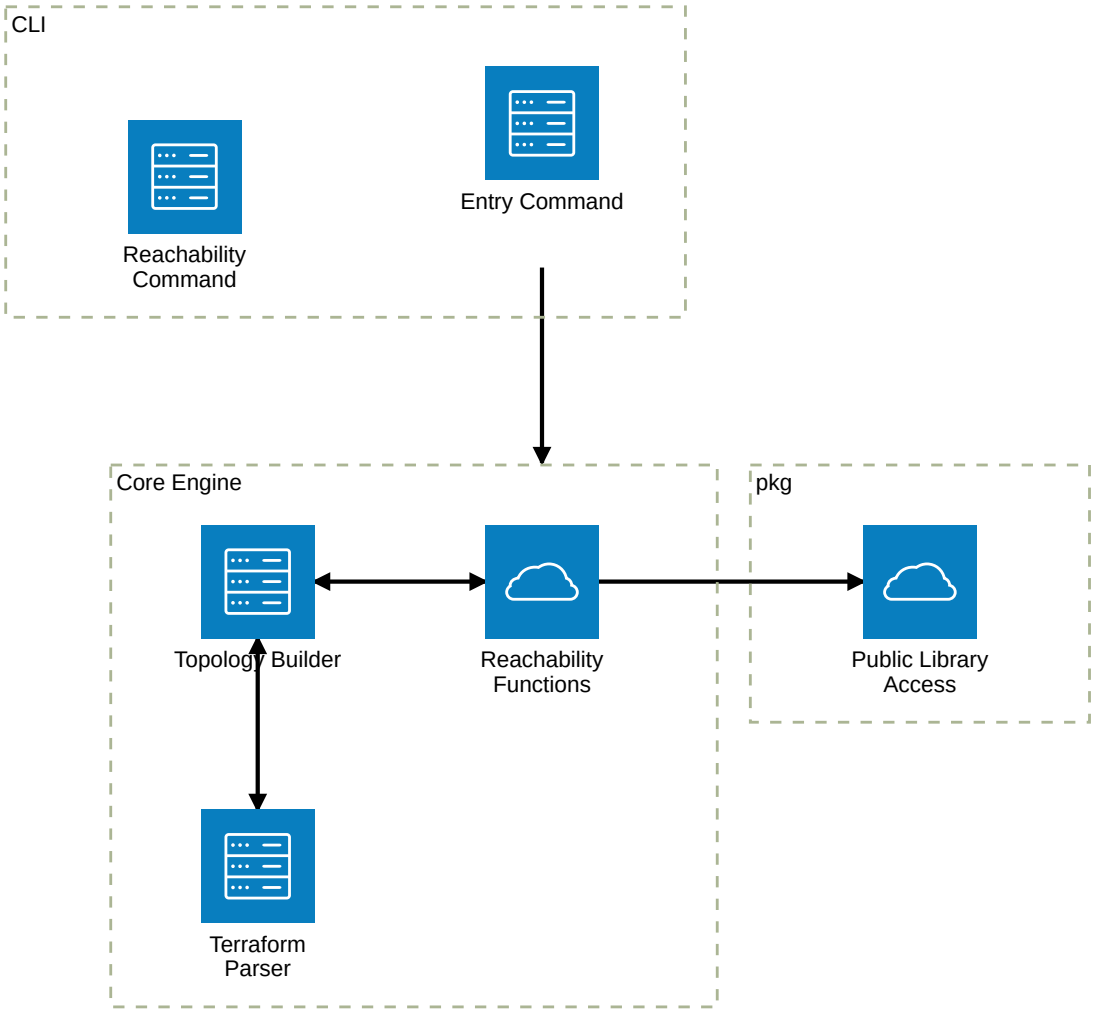

Figure 5.1: Architecture of the PoC

Figure 5.1 provides a visual representation of the architectural layout described above. The system is divided into three major logical components: *CLI*, *Core Engine*, and *Public Library Access*. The CLI layer includes the `Entry Command` and the specific `Reachability Command`, both responsible for initiating execution based on user input. These interact with the central `Reachability Functions` located in the Core Engine. The Core Engine also encapsulates the `Topology Builder`, which constructs a model of the infrastructure's network topology based on parsed Terraform configurations provided by the `Terraform Parser`. This process forms the foundation for simulating and evaluating network reachability. The architecture further exposes selected reachability functions and Terraform parsing utilities through the `Public Library Access`. This clean separation of concerns enables both CLI interaction and library-based integration while supporting extensibility and maintainability across use cases.

5.4 Boundaries of the PoC

The PoC presented in this thesis explicitly focuses on network reachability within Terraform-managed AWS infrastructure. This limitation is deliberate and motivated by several practical considerations. First, network reachability was chosen as the verification property due to the existence of the AWS Reachability Analyzer [29]. By comparing results from our tool to those of an established industry-standard analyzer, we can effectively verify correctness and reliability, which is essential for validating the effectiveness of our approach.

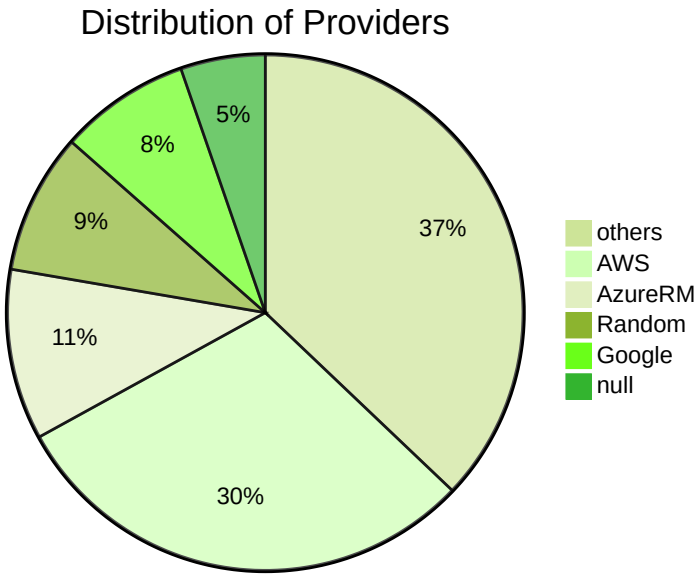

Figure 5.2: Distribution of Providers in Terraform

Terraform was selected as the target Infrastructure as Code tool primarily because of its widespread adoption and significant presence in industry and open-source communities. As evidenced by the recently published TerraDS dataset, Terraform currently has extensive coverage in publicly available repositories, significantly surpassing alternative IaC tools such as Pulumi [9, 40]. Additionally, the TerraDS dataset provides researchers with a robust foundation of Terraform modules and resources, enabling comprehensive analytics and systematic studies. The dataset clearly indicates that AWS is the most frequently utilized cloud provider (see Figure 5.2), making it an ideal initial target for developing and demonstrating the effectiveness of our analysis tool. Thus, the choice of AWS and Terraform facilitates practical validation, leveraging readily available resources and existing benchmarks in the domain of static analysis and IaC verification.

5.5 Parse Terraform Configuration Files

The first iteration of the implementation focused on parsing Terraform configuration files by manually resolving attribute values in the HashiCorp Configuration Language. Specifically, it attempted to interpret expressions directly within the configuration during the process of constructing the network topology. For instance, if a resource attribute referred to another resource using a reference expression (e.g., `aws_vpc.main.id`), the implementation would try

to trace and substitute the referenced value dynamically during parsing. While this approach demonstrated the feasibility of on-the-fly resolution for simple attributes, it quickly revealed limitations in handling complex expressions, nested references, and dynamically computed values. This led to challenges when more complex patterns – like the one in Listing 5 – were encountered. Referencing values became a big issue.

```
resource "aws_vpc" "main" {
  cidr_block = "10.0.0.0/16"
}

resource "aws_route" "demo" {
  route_table_id         = aws_route_table.main.id
  destination_cidr_block = aws_vpc.main.cidr_block
}
```

Listing 5: Terraform example - Resource references

The subsequent iteration of the implementation aligned more closely with Terraform's internal mechanics by adopting a directed acyclic graph (DAG) approach to resolve dependencies between resources. In this design, all resources from the configuration files are first added as graph nodes. Then, during a secondary pass, each resource's expressions are analyzed to determine inter-resource dependencies, and edges are added accordingly to reflect these relationships. The resulting DAG accurately represents the evaluation order required to resolve all references in compliance with Terraform's dependency model. This approach enables a consistent and reproducible resolution of references without relying on actual deployment or provider APIs.

Once the DAG is constructed, the implementation performs a directed traversal of the graph to evaluate each resource node in topological order. As each vertex is visited, its attribute expressions are resolved, and their resulting values are stored in a shared evaluation context. This context allows subsequent resources to access the resolved values of previously evaluated dependencies. For example, a subnet resource can now be resolved with full knowledge of its parent VPC's CIDR block or identifier. By systematically walking the DAG and maintaining a progressively enriched evaluation context, the parser can substitute references with effective values throughout the entire configuration. As a result, all resources are ultimately annotated with concrete, statically derived values, enabling accurate static analysis of infrastructural properties like network reachability.

The complete Terraform parsing process is illustrated in Figure 5.3. It begins with decoding the raw HCL body, extracting all defined resources, variables, and expressions. These are initially added to the internal model as vertices in a DAG. Once all components are present, a second pass adds the edges that represent inter-resource dependencies by analyzing expressions and references. The core logic then proceeds to walk the DAG in topological order. During this traversal, each node is processed depending on its type – whether it is a variable, expression, or resource definition. Expressions are evaluated in the context of already resolved dependencies, and their resulting values are stored. This ensures that all references within the configuration,

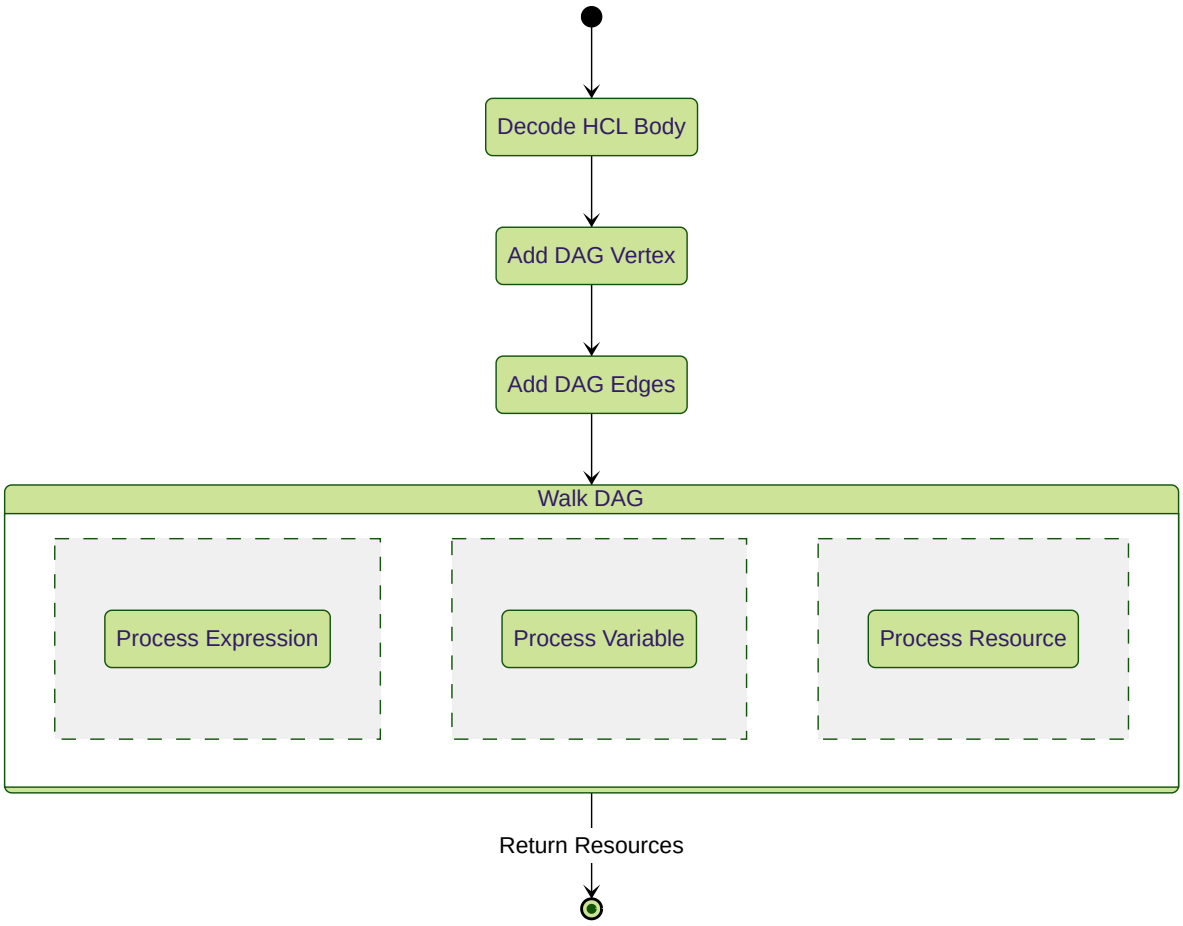

Figure 5.3: Parse Terraform Configuration

such as those involving CIDR blocks or resource identifiers, are statically resolved and materialized. Finally, the evaluated and fully populated resource representations are returned, enabling accurate and consistent downstream static analysis.

5.6 Create a Network Topology from Resources

In the context of AWS, constructing a network topology involves defining and interconnecting various resources that facilitate communication within and outside the cloud environment. Key components include:

- **Virtual Private Cloud (VPC)**: Serves as the fundamental network boundary, allowing the creation of isolated networks within AWS.[3]
- **Subnets**: Subdivisions within a VPC that segment the network into smaller, manageable sections.[4]
- **EC2 Instances**: Virtual servers that reside within subnets and perform compute tasks.[5]

[3] `https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/vpc`
[4] `https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/subnet`
[5] `https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/instance`

- **VPC Peering Connections**: Enable communication between VPCs, facilitating resource sharing across different networks.[6]
- **Route Tables and Routes**: Define the traffic flow within the network by specifying how packets are directed.[7]
- **Internet Gateway**: Allows resources within the VPC to access the internet.[8]
- **Security Groups**: Act as virtual firewalls, controlling inbound and outbound traffic to AWS resources.[9]

These components collectively define the structure and security of the network, ensuring controlled and efficient communication pathways.

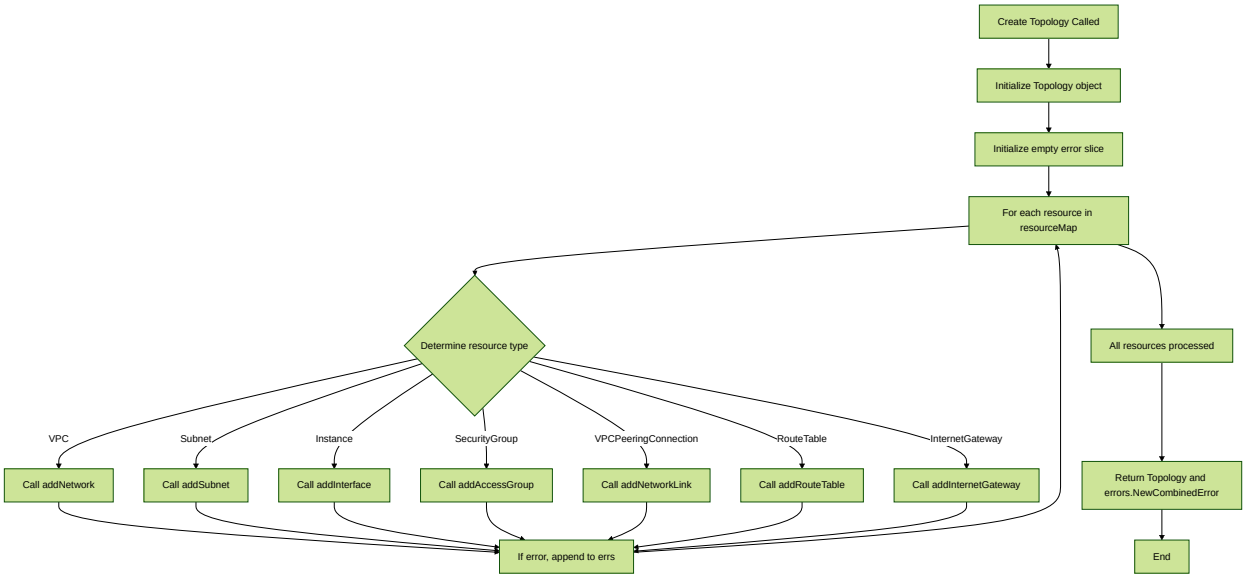

Figure 5.4: Creating the network topology

The process of building the network topology from parsed Terraform resources is outlined in Figure 5.4. The function begins by initializing the topology builder, which involves setting up the core data structure for tracking the network graph and allocating an empty slice to collect any errors encountered during processing. It then iterates over all entries in the internal `resourceMap`, where each resource is dynamically classified based on its type.

Depending on the resource type, the system dispatches to the corresponding handler function: for example, `aws_vpc` resources invoke `addNetwork`, `aws_subnet` calls `addSubnet`, `aws_instance` routes to `addInterface`, and so on. Specialized functions exist for handling

[6] `https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/vpc_peering_connection`

[7] `https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/route_table` and `https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/route`

[8] `https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/internet_gateway`

[9] `https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/security_group`

access control (`addAccessGroup` for `aws_security_group`), connectivity (`addNetworkLink` for `aws_vpc_peering_connection`), routing (`addRouteTable` for `aws_route_table`), and external access (`addInternetGateway` for `aws_internet_gateway`). Each handler enriches the topology with objects specific to the role of the resource in the network structure. If any errors occur during processing, they are appended to the error list. During this step, a business-logic mapping is applied to ensure that the resource is interpreted according to its semantics. For example, a `aws_subnet` resource is added as a subnet object, while an `aws_vpc` resource is added as a network object. The topology struct in Listing 6 serves as the core data structure for this process, encapsulating all networks, network links, and route tables. It also maintains a reference to the resource map, which is essential for resolving dependencies and references between resources.

```
type Topology struct {
        Networks     map[string]*Network
        NetworkLinks map[string]*NetworkLink
        routeTables  map[string]*RouteTable

        resourceMap *terraform.ResourceMap
}
```

Listing 6: The Topology Struct

Each of the resource-specific functions – such as `addSubnet`, `addInterface`, or `addRouteTable` – is designed to be order-independent by internally verifying the presence of any required parent resources within the topology object. If a referenced parent (e.g., a VPC for a subnet, or a subnet for an instance) has not yet been added, the function will proactively create and insert the missing parent into the topology before proceeding. This strategy ensures that the overall construction process is robust against input order and does not rely on resources being parsed or processed in a specific sequence. As a result, the topology graph remains structurally consistent and complete regardless of how the resources were originally ordered in the Terraform configuration files.

Once all resources have been processed, the function aggregates any collected errors and returns the final topology object along with an optional combined error. This modular dispatch pattern ensures that each resource is interpreted according to its semantics, leading to a well-structured and semantically rich topology model ready for subsequent analysis.

5.7 Network Reachability Analysis

The `Topology` object is the central in-memory model of the network structure, constructed from parsed Terraform resources. As depicted in Figure 5.5, it organizes and interrelates key domain entities, including `Network` (representing a VPC), `Subnet`, `Interface` (modeling subnets and EC2 instances), `RouteTable`, `InternetGateway`, `NetworkLink` (for VPC peering), and `AccessRuleGroup` (for security groups). Each of these components encapsulates a specific aspect of AWS networking: for example, a `Network` contains multiple `Subnet` objects,

each of which may host several `Interface` instances. `AccessRuleGroup` aggregates ingress and egress `AccessRule`s, capturing the semantics of AWS security groups and related rules. The `Topology` object maintains explicit relationships between these entities: subnets are associated with networks, interfaces are attached to subnets, and route tables and gateways are linked to their respective networks. Routing logic is modeled via `RouteTable` and `Route` objects, which determine how traffic is forwarded within and between networks, including support for VPC peering[10] (`NetworkLink`) and internet gateways. Access control is enforced through the association of interfaces with one or more `AccessRuleGroup`s, reflecting the effective security group rules for each network interface. Importantly, the topology is incrementally constructed as Terraform resources are parsed and is further updated during reachability analysis. This results in a live, queryable structure that accurately reflects all relevant network paths, routing policies, and access restrictions present in the infrastructure configuration. The object-oriented design of the topology enables efficient computation of connectivity and reachability, and provides a flexible foundation for future extensions, such as modeling additional cloud providers or supporting advanced analyses of network isolation and policy compliance.

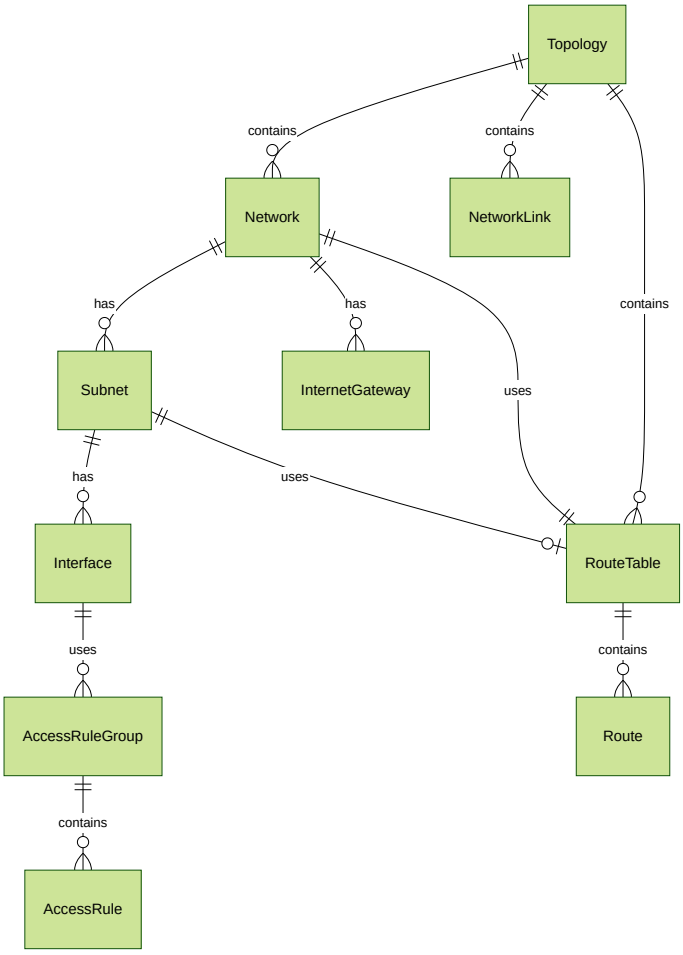

Figure 5.5: The Topology Construct

The `IsReachable` method in Listing 7 is a core function of the `Topology` object, responsible for determining whether a network source can reach a given destination on a specified port

[10] VPC Peering: Mechanism for intercommunication between different and/or distant VPC networks.

```go
func (t *Topology) IsReachable(
  source, destination string,
  port uint16,
  protocol Protocol,
) ReachabilityResult {
  // ...
}
```

Listing 7: The Main Reachability Check Function

and protocol. The method begins by resolving the source and destination identifiers to their corresponding network interfaces. It then verifies the existence of these interfaces and their parent networks, returning early if any are missing. The method retrieves the relevant egress rules from the source and ingress rules from the destination, which represent the effective security group policies applied to each interface. If both interfaces reside within the same network and neither has any access rules, the method immediately concludes that reachability is allowed.

If access rules are present, the method checks whether the source's egress rules permit traffic to the destination's IP, and whether the destination's ingress rules allow traffic from the source's IP, port, and protocol. If either check fails, the method returns a result indicating the specific rule that blocks connectivity. For interfaces in different networks, the method examines the existence of network links (such as VPC peering connections) and the presence of appropriate routes in the source's route table. If a valid route and link are found, reachability is granted; otherwise, the method checks for public internet access via an internet gateway. If no valid path is found, the method returns an unreachable result. This structured sequence of checks enables precise modeling of AWS network reachability semantics within the in-memory topology.

The `IsReachable` (Listing 7) method returns a `ReachabilityResult` enum, which – currently – can have the following values:

- `Reachable`: The source can successfully reach the destination on the specified port and protocol.
- `Unreachable`: No valid path exists between the source and destination; connectivity is not possible.
- `SourceInterfaceNotAvailable`: The specified source interface does not exist in the topology.
- `DestinationInterfaceNotAvailable`: The specified destination interface does not exist in the topology.
- `SourceNetworkNotAvailable`: The network containing the source interface is missing or undefined.
- `DestinationNetworkNotAvailable`: The network containing the destination interface is missing or undefined.

- `BlockedByEgressRules`: The source's egress rules explicitly block traffic to the destination (if the rule list is not empty).
- `BlockedByIngressRules`: The destination's ingress rules explicitly block traffic from the source (if the rule list is not empty).
- `NoRouteAvailable`: There is no valid route in the route table to forward traffic from the source to the destination.
- `NoPublicAccess`: The network lacks a route to an internet gateway, preventing public access.

This enumeration is used to provide a clear and structured way to communicate the result of the reachability check, allowing for easy interpretation and handling of different scenarios.

5.8 Exposing the CLI

The proof of concept provides a command-line interface (CLI) that enables users to perform network reachability analysis on infrastructures defined using Terraform. The main command, `reachability`, allows querying whether two network interfaces – such as EC2 instances or the public internet (WAN) – are able to communicate over a specified port and protocol. The CLI requires the user to provide the path to a directory containing Terraform configuration files. Additionally, users can optionally specify a source address, a destination address, a port number, and a protocol (from the supported set: `any`, `tcp`, or `udp`). If any of these optional arguments are omitted, the tool interactively prompts the user for input, thereby ensuring usability even when not all parameters are explicitly provided.

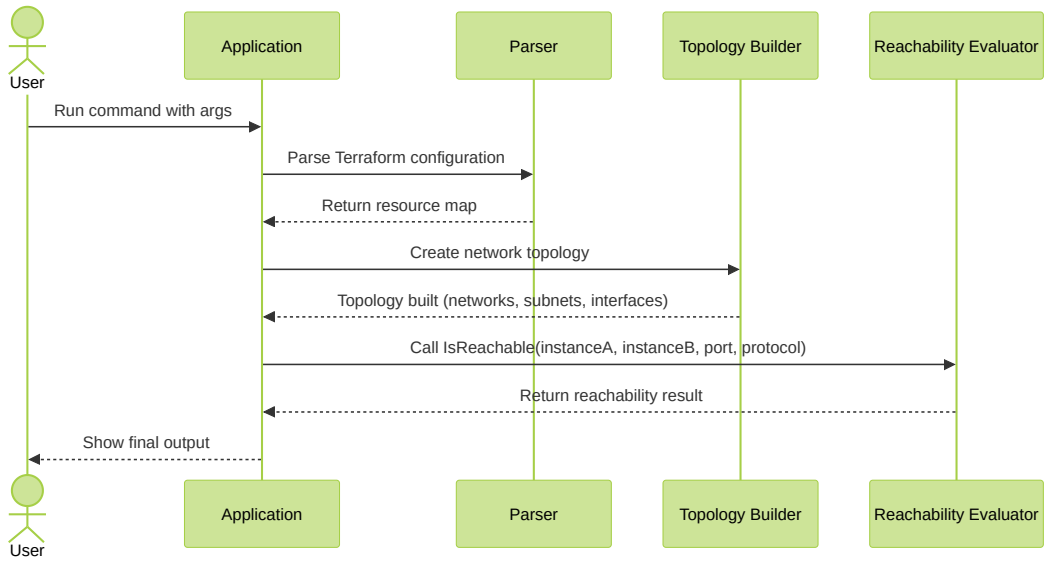

Figure 5.6: Process Overview for CLI Usage

Internally, the CLI initiates a parsing phase in which the provided Terraform configurations are converted into an internal resource representation (Section 5.5). Subsequently, a topology builder constructs the in-memory model of the network (Section 5.6), containing all networks,

subnets, interfaces, and their connections. With this model, the reachability analysis module evaluates whether the specified source and destination can communicate, considering route tables, security group rules, and network structures (Section 5.7). Finally, the result is presented to the user – either confirming successful connectivity or providing the reason for failure (e.g., blocked by a security group or missing route). This mechanism, illustrated in Figure 5.6, supports both automated CI/CD workflows and manual, ad-hoc usage by engineers or security analysts.

5.9 Complementary Tools

This subsection introduces a set of complementary tools developed solely for the evaluation of the network reachability library presented in the PoC. These tools are not part of the core implementation, but are designed to facilitate systematic benchmarking and validation of the library's analysis capabilities.

5.9.1 WAN Access Checker

The `wan_access` checker is an evaluation tool designed to systematically assess the network reachability of cloud resources provided by TerraDS [9]. It operates by reading a CSV file (`all_instances.csv`) that enumerates instances across various repositories and modules. For each instance, the checker generates a comprehensive set of reachability queries, including tests for inbound and outbound connectivity to and from the public internet (WAN) on common ports (such as 22, 80, and 443), as well as cross-instance reachability within the same module. These queries are executed in parallel, leveraging all available CPU cores to maximize throughput and efficiency during large-scale evaluations.

For each reachability query, the checker ensures the relevant Terraform repository is available (extracting it from an archive if necessary), parses the infrastructure using the project's parser, and invokes the reachability analysis engine. The results, including the outcome (e.g., reachable, blocked by rules, or no route available), any errors encountered, and the duration of the check, are recorded in a results CSV file. This systematic approach enables automated, reproducible benchmarking of the IaC verification tool's ability to detect and explain network exposure and isolation, providing valuable insights into the security posture of diverse infrastructure-as-code projects.

5.9.2 Plan Creator

The `plan_creator` tool is a utility designed to generate synthetic Terraform infrastructure plans for evaluation and benchmarking purposes. It programmatically constructs AWS network topologies with a configurable number of VPCs (up to four[11]), each containing subnets, security groups, and EC2 instances. The tool uses Go templates to produce Terraform code that defines these resources, including VPC peering connections and the necessary routing to enable inter-VPC communication. Each generated VPC is assigned a subnetted portion of a larger supernet, and security groups are configured with a variety of ingress and egress rules to simulate realistic access control scenarios.

[11] The number of VPCs is limited – including the default VPC – to five VPCs by AWS. Going over that limit is possible but requires inquires to AWS to be made.

By allowing users to specify the number of networks as a command-line argument, `plan_creator` enables systematic scaling of infrastructure complexity for testing the IaC verification tool. The generated plans include not only intra-VPC resources but also cross-VPC peering and routing, providing a rich environment for evaluating reachability, isolation, and policy enforcement. This approach ensures that the verification tool can be rigorously assessed against a range of topologies, supporting both functional validation and performance benchmarking in a controlled, reproducible manner.

5.9.3 Instance Analyzer

The `instance_analyzer` tool is a lightweight command-line utility designed to evaluate network reachability between two specific instances within a Terraform-managed infrastructure. By accepting the path to a Terraform configuration directory, the identifiers of the source and destination instances, and a port number as arguments, the tool parses the infrastructure, constructs the in-memory network topology, and determines whether the specified source can reach the destination on the given port. The analysis leverages the project's reachability engine, which takes into account security groups, routing, and other network policies defined in the Terraform files. This tool is basically the same as the CLI of the library. But it was used in a very specific way for the automated evaluation of the library and is not part of the final implementation.

This tool is particularly useful for targeted, instance-level connectivity checks during evaluation or debugging of the IaC verification system. It provides immediate feedback on whether communication is possible between two endpoints, reporting the result via structured log output. By automating the process of parsing, topology construction, and reachability analysis, `instance_analyzer` streamlines the validation of network policies and supports fine-grained testing scenarios within larger evaluation workflows.

5.9.4 AWS Reachability Analyzer

The `aws_reachability_analyzer` tool serves as an interface to AWS's native Network Reachability Analyzer, automating the process of empirically verifying network connectivity between two EC2 instances in a specified AWS region. By accepting the names of the source and destination instances, a port number, and a protocol (TCP or UDP) as command-line arguments, the tool uses the AWS SDK to resolve instance IDs, create a Network Insights Path, and initiate a Network Insights Analysis. It then polls the analysis status until completion, measuring the time taken for the analysis and reporting this duration as its output. The tool also ensures cleanup by deleting the created network insights resources after the analysis concludes.

This tool is particularly valuable for benchmarking and validating the results of the IaC verification tool against the actual behavior of AWS's own reachability analysis service. By providing a ground truth for network connectivity in real AWS environments, it enables systematic comparison between the static analysis performed by the verification tool and the dynamic, cloud-native analysis provided by AWS. This approach supports both correctness validation and performance evaluation, ensuring that the verification tool's predictions align with real-world network behavior as observed in the cloud provider's infrastructure.

Chapter 6

Evaluation

This thesis addresses the critical gap of lacking pre-deployment reachability verification for Infrastructure as Code (IaC). This section presents a structured evaluation of the proposed Infrastructure as Code (IaC) verification tool, focusing on its applicability and scalability. We conduct a series of empirical experiments on both real-world and synthetic Terraform configurations to assess how effectively the tool handles diverse infrastructure setups and how its performance compares to existing post-deployment solutions. The evaluation is organized around key research questions, with each subsection addressing a distinct dimension of the tool's capabilities, highlighting both its strengths and potential limitations.

6.1 Research Questions

RQ-1 *Generalized Applicability: Can the IaC verifier effectively analyze diverse and complex real-world Infrastructure as Code programs?*

This research question investigates the general applicability of the proposed verifier across a broad spectrum of real-world IaC configurations. It aims to determine whether the tool can handle the structural complexity, varying resource types, and real-use scenarios commonly found in practical Terraform deployments. Demonstrating successful application to such configurations would validate the verifier's robustness and indicate its readiness for use in real-world DevOps and infrastructure engineering contexts.

RQ-2 *Scaling: How does the performance of the IaC verification tool scale with the size of the infrastructure? How does it compare to the performance of the AWS Network Reachability Analyzer?*

This research question explores the scalability of the verifier when analyzing network reachability across infrastructures of varying sizes. Specifically, it compares the verifier's performance against AWS Reachability Analyzer, which performs post-deployment checks for similar network queries. Since static analyzers typically do not support reachability queries, they are excluded from this comparison. The goal is to assess whether the verifier's pre-deployment, local analysis approach enables faster and more scalable evaluations—especially in scenarios where repeated or automated checks are required.

6.2 RQ 1 - Applicability

Research Question: *To what extent can the verifier be applied successfully to existing, real-world Infrastructure as Code (IaC) programs written in Terraform?*

This research question focuses on the operational feasibility of using the verifier in practice. Specifically, it assesses whether the tool can process actual Terraform modules from open-source repositories and yield meaningful analysis results. Rather than exploring the theoretical capabilities of the tool, this question addresses its practical integration: Can the verifier parse and analyze real infrastructure programs without failure, and can it respond to meaningful reachability queries within those configurations?

6.2.1 Procedure

To investigate this, we executed the verifier on 1,313 Terraform modules containing a total of 2,502 EC2 instances, sourced from TerraDS [9]; the largest curated dataset of Terraform programs to date. As shown in Figure 5.2, AWS is the most frequently used cloud provider in the dataset, making it an ideal target for our initial evaluation. The modules chosen include AWS resource types supported by the verifier and define valid, real-world virtual network topologies. Each module was evaluated as a self-contained infrastructure unit.

The evaluation consists of two main tasks: First, *external reachability* analysis was performed to determine whether EC2 instances could be reached from or connect to the public internet (WAN) over ports 80 (HTTP) and 443 (HTTPS). Second, *internal reachability* was tested between EC2 instances within the same module. For this, the dataset is filtered to select modules with at least two EC2 instances, yielding 796 modules and 7,390 unique reachability queries (sourced in the Cartesian product of the queries time the instances). The queries included typical internal communication over port 22 (SSH), port 80 (HTTP), port 443 (HTTPS), and an "any"-port reachability check.

6.2.2 Metrics

We measured two primary outcomes:

- **Percentage of successful reachability queries completed by the verifier:** This metric indicates the breadth of the tool's applicability across diverse real-world configurations.
- **Average execution time per query:** This captures the time needed to parse, model, and analyze each module.

These metrics serve as proxies for usability in practical settings, where reliability and performance are essential for adoption in CI/CD pipelines or infrastructure validation workflows.

6.2.3 Results

External Reachability Analysis Table 6.1 presents the results of the *external reachability* evaluation using the PoC. The table is organized by reachability direction (ingress and egress) and by port (80 and 443). Column 2 reports the ability to reach EC2 instances from the internet (ingress), while Column 3 shows the ability of instances to reach the internet (egress). Column 4 provides the average execution time per query, including parsing, topology building, and the actual analysis step. The rows are grouped by the outcome of the reachability query.

The *Not Reachable* results are further categorized into two error classes: *No Public Access* and *Blocked by Rule*. The former indicates that the infrastructure configuration lacked access to or from the internet (e.g., missing Internet Gateway or Route configuration), while the latter reflects missing security group rules that explicitly allowed the corresponding traffic. The final category, *Failed to Analyze*, aggregated all modules that could not be processed by the PoC, typically due to syntax errors or unsupported constructs.

Analysis of Failures A deeper inspection of the 836 failures in the ingress analysis on port 80 reveals that 448 cases lacked the `subnet_id` field, and 548 contained unparsable CIDR values. The absence of `subnet_id` prevented correct mapping of EC2 instances to subnets, thereby rendering their location within the VPC unresolvable. Similarly, malformed or empty CIDR values violated expected formats for defining network ranges and thus invalidated the resource. These findings underscored the prevalence of incorrect or incomplete real-world IaC configurations. Importantly, the PoC did not attempt to infer missing values, as such inference would undermine the integrity and trustworthiness of the analysis.

Success Rates and Performance Despite these challenges, the PoC successfully completed approximately two-thirds of all external reachability queries. This result was notable given that the evaluation did not exclude erroneous or deprecated modules. Additionally, the results highlighted a clear asymmetry between port access: 44% of instances were reachable on port 80 (HTTP), but only 4% on port 443 (HTTPS). This discrepancy was partly explained by repositories with duplicated modules that defaulted to HTTP.

Performance-wise, the PoC's average runtime of just 51ms per query suggested it was well-suited for integration into the development workflow, particularly in CI/CD pipelines where rapid feedback was critical.

Internal Reachability Analysis Table 6.2 summarized the results of the *internal reachability* analysis between EC2 instances defined within the same Terraform module. Three port categories were evaluated: port 22 (SSH), port 80 (HTTP), and "any port". Each pair of instances was analyzed for connectivity, and results were grouped into categories: *Reachable*, *Blocked (Ingress)*, *Blocked (Egress)*, *No Route*, *No Peering*, and *Failed to Analyze*. *No Route* captured cases with misconfigured/absent route tables, while *No Peering* denoted instances located in separate VPCs without a peering connection. VPC peering is a specialty of AWS that allowed communication between VPCs. But to do so, a VPC Peering and corresponding route objects had to be configured. Figure 6.1 showed an example with two VPCs that were connected via a peering connection.

Module-Level Analysis Overall, the PoC answered approximately one-third of internal reachability queries. The higher failure rate relative to external analysis was explained by query volume: each module may yield numerous internal pairwise queries, and a single parsing failure led to a complete loss of results for all pairs in that module.

In addition to query-level statistics, we evaluated the number of modules that could be successfully analyzed. The PoC completed external reachability analysis on 805 out of 1,313 modules (61%) and internal analysis on 571 out of 796 modules (72%). In most failing cases, analysis was impeded by structural issues in the IaC code, such as syntax violations or deprecated language constructs, which prevented the construction of a valid model. These results reaffirmed both the practicality and resilience of the PoC when applied to a diverse and imperfect landscape of real-world Terraform programs.

6.2.4 Conclusion

The PoC successfully analyzed the majority of real-world IaC modules and executed most of the reachability queries. Its low runtime overhead, combined with informative diagnostic messages for unreachable paths, made it well-suited for use as a pre-deployment debugging tool in development and CI/CD workflows.

Category	**Internet to EC2** *Port 80* *Port 443*	**EC2 to Internet** *Port 80* *Port 443*	**Avg. Time**
Reachable	1,105 (44%) 89 (4%)	1,304 (52%) 1,303 (52%)	36.7ms 34.7ms
No Public Access	20 (1%) 26 (1%)	100 (4%) 104 (4%)	21.5ms 23.5ms
Blocked by Rule	541 (22%) 1,547 (62%)	250 (10%) 249 (10%)	39.9ms 39.5ms
Failed to Analyze	836 (33%) 840 (34%)	848 (34%) 846 (34%)	86.8ms 84.9ms
Total	2,502 (100%)	2,502 (100%)	51.5ms

Table 6.1: Evaluation of external reachability of the IaC verification tool on the TerraDS [9] dataset.

Category	**EC2 to EC2**		
	Port 22	Port 80	any port
Reachable	1,790 (24%)	1,640 (22%)	1,867 (25%)
Blocked (ingress)	432 (6%)	615 (8%)	350 (5%)
Blocked (egress)	59 (1%)	53 (1%)	58 (1%)
No Route	0 (0%)	2 (0%)	2 (0%)
No Peering	29 (0%)	19 (0%)	41 (1%)
Failed to Analyze	5,080 (69%)	5,061 (68%)	5,072 (69%)
Total	7,390 (100%)	7,390 (100%)	7,390 (100%)

Table 6.2: Evaluation of internal reachability of the IaC verification tool on the TerraDS [9] dataset.

6.3 RQ 2 - Scaling

Research Question: *How does the performance of the IaC verification tool scale with the size of the infrastructure? How does it compare to the performance of the AWS Network Reachability Analyzer?*

This research question explores the scalability of the proposed PoC in contrast to the current state-of-the-practice, namely post-deployment tools such as AWS Reachability Analyzer (AWS RA). Specifically, it examines how both tools behaved as the size and complexity of the infrastructure grew. While AWS RA performed analysis after infrastructure was provisioned in the cloud, the PoC performed local, static analysis pre-deployment. We hypothesized that the PoC offered significant performance advantages by avoiding the overhead of actual deployment and leveraging a local model of the infrastructure.

6.3.1 Procedure

To address this question, we constructed – with the help of the "plan creator" (Section 5.9.2) – four synthetic Terraform IaC programs in the form of Figure 6.1, each representing a distinct network configuration with an increasing number of Virtual Private Clouds (VPCs), ranging from one to four. Each VPC contained two subnets, and each subnet hosted two EC2 instances, creating a symmetrical and scalable infrastructure setup.

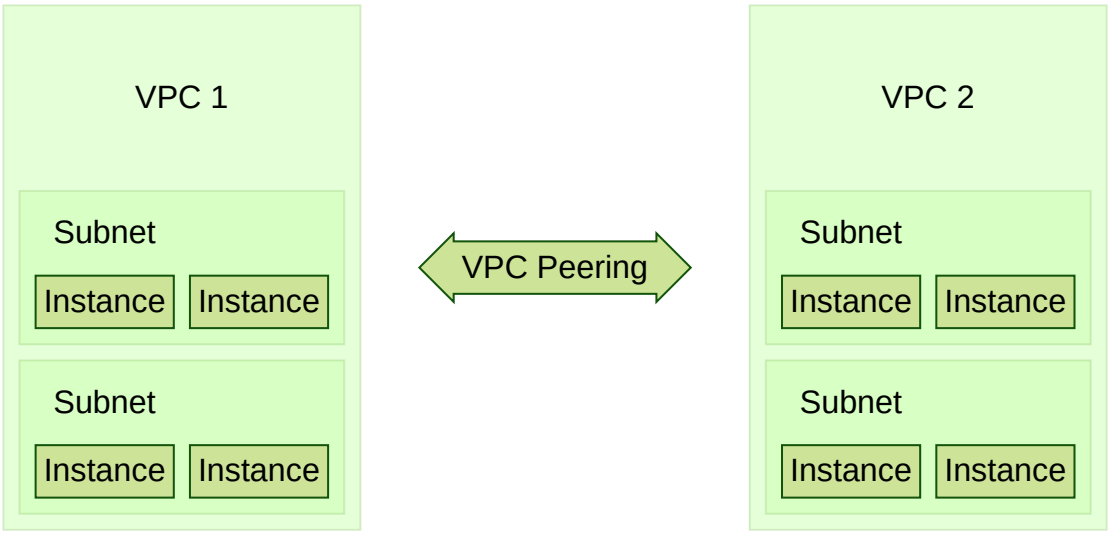

Figure 6.1: Artificial VPCs used for the evaluation of the IaC verification tool.

The intra-VPC configuration constructed according to Figure 6.1. The artificial IaC programs were created with the "plan creator" (Section 5.9.2), whereas the measurement of the AWS RA execution was done with the "AWS Reachability Analyzer" helper tool (Section 5.9.4). As the number of VPCs increased, the size of the Terraform configuration ranged from 172 lines of code (LOC) for one VPC to 784 LOC for four VPCs. This range reflected real-world usage as observed in the TerraDS dataset [9], which showed that 20,966 modules defined exactly one VPC, 1,174 defined two, 101 defined three, and only 32 defined four. Modules defining more than four VPCs were rare and not representative of common practice.

For each IaC configuration, we ensured full connectivity across instances by defining appropriate VPC peerings and route tables. All instances were assigned security groups containing

five rules (three inbound, two outbound), ensuring that security constraints did not hinder the scalability measurement. We compared the PoC against AWS RA, which was limited to single-region deployments [29]. Reachability queries spanning multiple regions were unsupported by AWS RA but posed no issue for the PoC, which operated on a region-agnostic, static topology.

We executed the PoC locally on each configuration and compared its performance against AWS RA by deploying the infrastructure and issuing equivalent reachability queries via the AWS web console.

6.3.2 Metrics

We measured the execution time for each AWS RA query, which included both the time to deploy the infrastructure and the time to perform the reachability analysis post-deployment. Each AWS RA query was run five times, and the average duration was computed to reduce variability caused by deployment overhead and AWS-side delays.

For AWS RA, the timing was broken down into five phases:

- **Init**: Initializes the working directory and downloads Terraform providers.
- **Validate**: Performs static syntax checks on the configuration.
- **Apply**: Deploys the infrastructure to AWS.
- **AWS RA**: Executes the reachability query using AWS Reachability Analyzer.
- **Destroy**: Tears down all deployed resources.

For the PoC, we measured the end-to-end duration of each query from parsing the Terraform configuration to building the internal model and executing the reachability analysis. Since the PoC was deterministic and did not depend on cloud-side operations, we recorded the execution time once per configuration.

6.3.3 Results

Figure 6.2 provided a breakdown of total execution time across five operational phases – *Init*, *Validate*, *Apply*, *AWS RA*, and *Destroy* – when varying the number of VPCs from one to four. The overall trend demonstrated a substantial increase in time, from 127 seconds for a single VPC to over 204 seconds for four VPCs.

The *Destroy* phase consistently dominated the total runtime, growing from 75.1 seconds (1 VPC) to nearly 120 seconds (4 VPCs). The *Apply* phase also exhibited significant scaling behavior, increasing from 22.8 seconds (1 VPC) to 54.5 seconds (4 VPCs). In contrast, the *Init* and *Validate* phases remained relatively stable across all configurations, ranging from 7.7–8.4 seconds and 4.3–4.4 seconds respectively.

Interestingly, the *AWS Reachability Analyzer* phase showed minimal variance, hovering around 17 seconds regardless of the infrastructure size. This constancy reflected its bounded runtime but highlighted that the bulk of performance costs stemmed from provisioning and decommissioning infrastructure, not the analysis itself.

These results emphasized the high time overhead incurred by post-deployment validation methods like AWS RA, especially in scenarios with increasingly complex infrastructure. They underscored the advantage of the PoC's pre-deployment static analysis, which avoided such latency entirely and maintained a constant analysis duration of approximately 75 milliseconds.

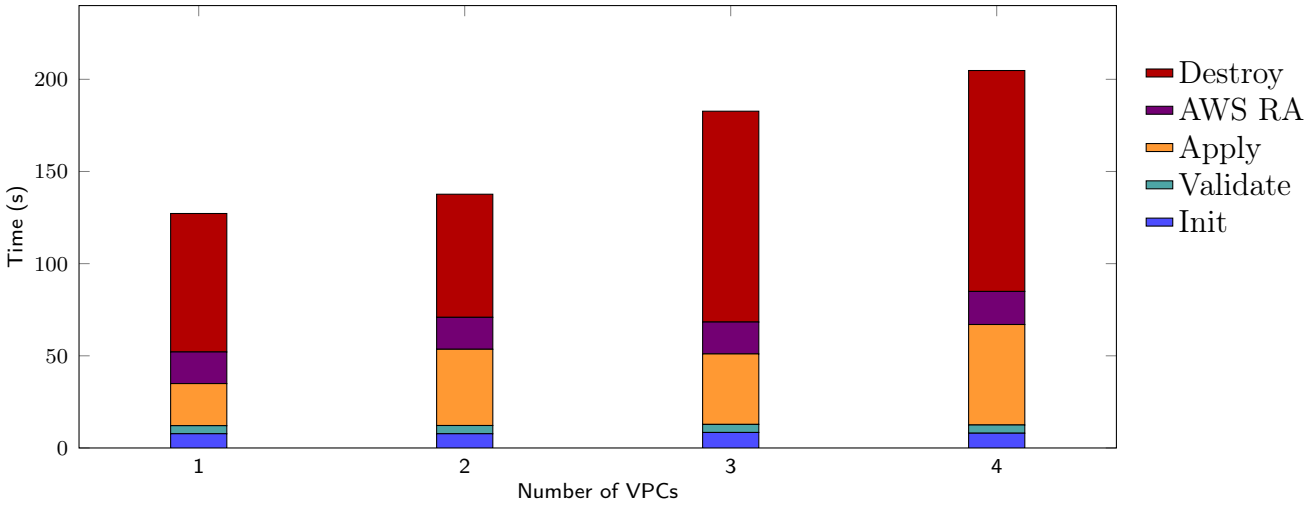

Figure 6.2: Performance metrics across deployment phases per number of VPCs. Averaged over 5 runs.

6.3.4 Conclusion

The PoC demonstrated clear scalability advantages. Its runtime remained virtually constant across different configurations, highlighting its suitability for iterative development cycles where infrastructure was frequently modified and revalidated. Unlike AWS RA, which incurred considerable latency due to deployment and teardown phases, the PoC enabled instant feedback without provisioning any resources.

This showed that the PoC could dramatically reduce analysis time in real-world DevOps workflows, enabling rapid iteration without sacrificing accuracy or completeness.

6.4 Summary

In summary, the evaluation demonstrated that the proof of concept (PoC) verifier was broadly applicable to real-world Infrastructure as Code programs, successfully analyzing the majority of Terraform modules and reachability queries sourced from the TerraDS dataset (Table 6.1, Table 6.2). The tool exhibited low runtime overhead, with average query times suitable for integration into CI/CD pipelines and development workflows. Comparative analysis with AWS Reachability Analyzer highlighted the PoC's significant scalability advantage, as it avoided the time-consuming deployment and teardown phases required by post-deployment tools (Figure 6.2). While some limitations remained – primarily due to incomplete or erroneous IaC configurations – the results confirmed the practical value and efficiency of pre-deployment reachability analysis. Overall, the PoC provided a robust foundation for further research and development in automated IaC verification.

Chapter 7

Discussion and Outlook

The evaluation results demonstrate that the proof of concept successfully addresses both research questions: it proves applicable to real-world IaC programs, successfully analyzing the majority of Terraform modules from the TerraDS [9] dataset, and shows significant performance advantages over post-deployment tools like AWS Reachability Analyzer. Building on these findings, this discussion section examines the broader implications of the proposed approach. First, we analyze the key advantages of the reachability analyzer, including its reusability through abstract modeling, time and cost efficiency, and seamless integration into DevOps workflows. Next, we address the limitations of the current implementation, particularly regarding dynamic configuration values and provider-specific behaviors. Finally, we explore potential future work directions, such as extending the verification approach to other infrastructure properties like cost and energy efficiency, and integrating dynamic adaptation mechanisms for continuous optimization.

7.1 Advantages of the Reachability Analyzer Approach

7.1.1 Reusability Through Abstract Modeling

A key contribution of this thesis was the implementation of provider-specific resource types for abstract infrastructure modeling. This process included analyzing cloud provider APIs, defining abstract representations of resources such as VPCs, subnets, and instances, extending the Terraform parser, and translating parsed configurations into their abstract equivalents. For property-specific analyses – such as the network reachability use case presented in this work – these abstractions were further mapped to domain-specific models (topology). Once implemented, these resource abstractions were reusable across different IaC programs, configurations, and verification tools, making the overall approach portable and extensible.

This modular design laid the foundation for a broader ecosystem of reusable infrastructure models. Future work could extend this effort into a community-driven repository of shared resource definitions. Similar initiatives, such as the AWS Resource Providers project[1], demonstrated the feasibility and benefits of such collaborative development, where resource types were defined and maintained across tool boundaries.

[1] `https://github.com/org-formation/aws-resource-providers`

7.1.2 Time and Cost Efficiency

By eliminating the need for infrastructure deployment, the reachability analyzer achieved a substantial performance improvement over post-deployment tools such as AWS Reachability Analyzer. Unlike static analyzers, it supported expressive queries involving the network topology. Developers received actionable feedback within milliseconds, enabling fast iterations during the development phase. This rapid feedback loop reduced turnaround time and avoided the operational and financial costs associated with deploying test infrastructure.

7.1.3 Seamless Integration into DevOps Workflows

The reachability analyzer was designed for easy integration into existing DevOps pipelines and developer tools. Because it operated locally and did not depend on actual cloud deployments, it could be used in environments without cloud access, including local development setups and CI/CD pipelines. This independence made it highly versatile: it could function as a standalone verification step or be embedded into Integrated Development Environments (IDEs), such as Visual Studio Code, to provide real-time validation as developers wrote their IaC programs. This supported a shift-left approach to infrastructure validation, catching errors early and improving overall development efficiency.

7.2 Limitations

The reachability analyzer, by design, performed offline verification and did not have access to the *output configuration*. That is, values assigned dynamically by the cloud provider during infrastructure deployment. Consequently, it could not reason about configuration behaviors that depended on runtime-assigned values, such as IP addresses or instance IDs that were referenced in other resources. For example, if a security group rule was configured to allow traffic only from an IP that was dynamically assigned during deployment, the analyzer could not determine this link ahead of time.

Another inherent limitation of the reachability analyzer was its reliance on the correctness of cloud provider specifications. Since the analysis operated solely on the declared configuration, it assumed that resources behaved exactly as documented by the provider. Unlike post-deployment tools that could query live infrastructure for actual behavior or enforcement (e.g., verifying whether security group rules were applied as expected), the reachability analyzer could not identify discrepancies caused by undocumented behavior or provider-side bugs. As such, it could not validate the runtime enforcement of policies. This responsibility ultimately lay with the cloud provider and fell outside the scope of static verification tools.

7.3 Future Work: Extending Early Verification of IaC Programs

While this work focused on the early detection and verification of network reachability issues in Infrastructure as Code (IaC) programs, the underlying approach was broadly applicable to other infrastructure properties that could be analyzed pre-deployment. One promising direction was the evaluation of cost efficiency: assessing whether the declared configuration led to unnecessarily expensive infrastructure deployments. Integrating cost modeling into the early

verification process could allow teams to optimize infrastructure spending before provisioning any resources.

Another possible research avenue was the analysis of energy efficiency. For instance, developers could define infrastructure-specific energy budgets or carbon impact goals, and the verifier could provide actionable feedback when a configuration was likely to exceed those thresholds. This aligned with the increasing emphasis on sustainable computing and green software engineering practices.

Moreover, combining static analysis with dynamic adaptation mechanisms opened the door for continuous optimization. A future extension of this work could involve runtime monitoring and auto-configuration of IaC-defined infrastructure, where resource usage patterns informed automated adjustments to the infrastructure (e.g., scaling policies, instance types, or region selection).

An important area for future development is extending the system to support additional cloud providers such as Microsoft Azure and Google Cloud Platform. This would require implementing additional abstraction layers analogous to those used for AWS. Specifically, Terraform-specific resources (e.g., `google_compute_network`, `azurerm_virtual_network`) would be mapped to internal, provider-agnostic representations such as `network`, similar to the translation from `aws_vpc` to `network`. By enriching the abstract object model and translation logic, the verifier could perform consistent reachability and policy checks across heterogeneous cloud platforms, thereby generalizing its applicability and increasing its practical relevance in multi-cloud environments.

Altogether, these extensions represent a promising step toward more intelligent infrastructure design workflows. By embedding static verification into the early stages of development, infrastructure engineers could receive actionable, context-specific feedback that goes beyond correctness to include cost, sustainability, and scalability metrics. This would shift infrastructure development from a reactive to a proactive practice, minimizing misconfigurations and resource waste before deployment.

Chapter 8

Conclusion

This thesis presents a new approach to verifying cloud infrastructure defined as code, focusing on pre-deployment analysis to detect misconfigurations before resources are provisioned. By statically analyzing the intended target state and simulating the behavior of cloud resources, we show that meaningful properties such as network reachability can be validated locally with high accuracy and minimal runtime.

We implemented a prototype for Terraform configurations targeting AWS, and evaluated it across two dimensions: applicability to real-world IaC programs and the scalability compared to the AWS Reachability Analyzer. The evaluation demonstrates that the tool can process a broad range of real-world configurations, offers orders-of-magnitude performance improvements, and yields accurate results comparable to cloud-native tools.

The motivation for this work arises from the increasing complexity and criticality of cloud infrastructure, as discussed in Chapter 1. The thesis first establishes the foundational concepts and terminology in Chapter 2, and surveys the state of the art in IaC verification and related tools in Chapter 3. Building on these foundations, Chapter 4 introduces the conceptual framework for pre-deployment verification, emphasizing the need for static, property-specific analysis that can be integrated early in the development lifecycle.

The core contribution is detailed in Chapter 5, where the design and realization of the proof of concept (PoC) are described. The implementation leverages Go for its performance and compatibility with Terraform's HCL, and is architected around a modular design that separates parsing, topology construction, and reachability analysis. The PoC demonstrates how Terraform configurations can be parsed, abstracted, and analyzed to construct a network topology model, enabling efficient reachability analysis without requiring cloud deployment.

A comprehensive evaluation, presented in Chapter 6, validates the approach on both real-world and synthetic datasets. The results confirm that the verifier is broadly applicable, while still being performant, with low runtime overhead suitable for CI/CD integration. The discussion in Chapter 7 highlights the advantages of abstract modeling, time and cost efficiency, and seamless DevOps integration, while also acknowledging limitations such as the inability to reason about runtime-assigned values and reliance on provider specifications. Future work is outlined, including extensions to cost and energy efficiency analysis, and the potential for runtime adaptation and continuous optimization.

Beyond network reachability, our approach is extensible to other properties, including compliance, cost, and energy efficiency. We envision a future where IaC verification becomes a standard part of the development cycle, supported by a modular and community-driven framework that integrates seamlessly into existing CI/CD workflows. The proposed tool contributes toward this vision by making verification both accessible and practical, ultimately improving the reliability and quality of cloud infrastructure engineering.

Bibliography

[1] Shanal Aggarwal. *Understanding Cloud Outages: Causes, Consequences, and Mitigation Strategies*. TechAhead. Dec. 20, 2024. URL: https://www.techaheadcorp.com/blog/understanding-cloud-outages-causes-consequences-and-mitigation-strategies/ (visited on 03/16/2025).

[2] Chisom Elizabeth Alozie et al. "The Role of Automation in Site Reliability Engineering: Enhancing Efficiency and Reducing Downtime in Cloud Operations". In: *International Journal of Engineering and Modern Technology* 11.1 (2025). ISSN: 2504-8848.

[3] Amazon Web Services AWS. *AWS CloudFormation*. 2024. URL: https://aws.amazon.com/cloudformation/.

[4] Matej Artac et al. "DevOps: Introducing Infrastructure-as-Code". In: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C). May 2017, pp. 497–498. DOI: 10.1109/ICSE-C.2017.162.

[5] Nitric Authors. *Nitric*. Version v1.24.2. Nitric Inc., Apr. 8, 2025. URL: https://nitric.io (visited on 04/08/2025).

[6] Mahi Begoug, Moataz Chouchen, and Ali Ouni. "TerraMetrics: An Open Source Tool for Infrastructure-as-Code (IaC) Quality Metrics in Terraform". In: *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*. ICPC '24. New York, NY, USA: Association for Computing Machinery, June 13, 2024, pp. 450–454. ISBN: 979-8-4007-0586-1. DOI: 10.1145/3643916.3644439.

[7] Mahi Begoug et al. "What Do Infrastructure-as-Code Practitioners Discuss: An Empirical Study on Stack Overflow". In: *2023 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 2023 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). New Orleans, LA, USA: IEEE, Oct. 26, 2023, pp. 1–12. ISBN: 978-1-6654-5223-6. DOI: 10.1109/ESEM56168.2023.10304847.

[8] Narjes Bessghaier et al. "On the Prevalence, Co-occurrence, and Impact of Infrastructure-as-Code Smells". In: *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). Rovaniemi, Finland: IEEE, Mar. 12, 2024, pp. 23–34. ISBN: 979-8-3503-3066-3. DOI: 10.1109/SANER60148.2024.00009.

[9] Christoph Bühler et al. "TerraDS: A Dataset for Terraform HCL Programs". In: *Proceedings of the IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*. MSR '25. Ottawa, Canada: IEEE, Apr. 2025.

[10] Michele Chiari, 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)*. 2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C). Mar. 2022, pp. 218–225. DOI: 10.1109/ICSA-C54293.2022.00049.

[11] Srinivas Chippagiri and Preethi Ravula. "Cloud-Native Development: Review of Best Practices and Frameworks for Scalable and Resilient Web Applications". In: *International Journal of New Media Studies* 8.2 (2021), p. 9. ISSN: 2394-4331.

[12] Edmund M. Clarke, Orna Grumberg, and David E. Long. "Model Checking and Abstraction". In: *ACM Trans. Program. Lang. Syst.* 16.5 (Sept. 1, 1994), pp. 1512–1542. ISSN: 0164-0925. DOI: 10.1145/186025.186051.

[13] Prisma Cloud. *Checkov*. URL: https://www.checkov.io/ (visited on 03/16/2025).

[14] Jye Cusch. *What Is Infrastructure from Code? | Get Infrastructure from Code*. Nitric. Aug. 8, 2024. URL: https://nitric.io/blog/what-is-infrastructure-from-code (visited on 04/08/2025).

[15] Michele De Pascalis. "Formal Verification of Infrastructure as Code". MA thesis. Politecnico Milano, Apr. 28, 2022. 92 pp. URL: https://www.politesi.polimi.it/handle/10589/185835 (visited on 04/10/2025).

[16] Kate Fazzini. "A Technical Slip-up Exposes Cloud Collaboration Risks". In: *Wall Street Journal. Pro Cyber* (June 13, 2017). ISSN: 0099-9660. URL: https://www.wsj.com/articles/a-technical-slip-up-exposes-cloud-collaboration-risks-1497353313 (visited on 04/11/2025).

[17] Linux Foundation. *OpenTofu*. URL: https://opentofu.org/ (visited on 02/13/2025).

[18] Steve Francia. *Spf13/Cobra*. Apr. 20, 2025. URL: https://github.com/spf13/cobra (visited on 04/20/2025).

[19] *Go Programming Language (Introduction)*. GeeksforGeeks. Apr. 15, 2025. URL: https://www.geeksforgeeks.org/go-programming-language-introduction/ (visited on 04/20/2025).

[20] Lewis Golightly et al. "Adoption of Cloud Computing as Innovation in the Organization". In: *International Journal of Engineering Business Management* 14 (Nov. 1, 2022), p. 18479790221093992. ISSN: 1847-9790. DOI: 10.1177/18479790221093992.

[21] Michele Guerriero et al. "Adoption, Support, and Challenges of Infrastructure-as-Code: Insights from Industry". In: *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME). Sept. 2019, pp. 580–589. DOI: 10.1109/ICSME.2019.00092.

[22] Bulbul Gupta, Pooja Mittal, and Tabish Mufti. "A Review on Amazon Web Service (Aws), Microsoft Azure & Google Cloud Platform (Gcp) Services". In: *Proceedings of the 2nd International Conference on ICT for Digital, Smart, and Sustainable Development, ICIDSSD 2020, 27-28 February 2020, Jamia Hamdard, New Delhi, India*. 2021, p. 9.

[23] Mohammed Mehedi Hasan, 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*. LANGETI 2020. New York, NY, USA: Association for Computing Machinery, Nov. 8, 2020, pp. 7–12. ISBN: 978-1-4503-8123-9. DOI: 10.1145/3416504.3424334.

[24] HashiCorp. *HashiCorp Configuration Language*. Version 2.21.0. GitHub, 2024. URL: https://github.com/hashicorp/hcl.

[25] HashiCorp. *HashiCorp Licensing FAQ*. HashiCorp. Apr. 15, 2024. URL: https://www.hashicorp.com/license-faq (visited on 02/13/2025).

[26] HashiCorp. *Terraform*. Version 1.9.4. Aug. 7, 2024. URL: https://www.terraform.io/.

[27] HashiCorp. *Tests - HashiCorp Developer*. Tests - Configuration Language | Terraform | HashiCorp Developer. URL: https://developer.hashicorp.com/terraform/language/tests (visited on 04/09/2025).

[28] Amazon Web Services Inc. *Cloud Computing Services - Amazon Web Services (AWS)*. Amazon Web Services. 2025. URL: https://aws.amazon.com/ (visited on 03/25/2025).

[29] Amazon Web Services Inc. *What Is Reachability Analyzer?* URL: https://docs.aws.amazon.com/vpc/latest/reachability/what-is-reachability-analyzer.html (visited on 04/10/2025).

[30] By Kurt Mackie and 01/26/2023. *Configuration Glitch Caused Microsoft's Jan. 25 Exchange Online Disruption – Redmondmag.Com*. Redmondmag. URL: https://redmondmag.com/articles/2023/01/26/configuration-glitch-caused-microsoft-jan-25-exchange-online-disruption.aspx (visited on 04/11/2025).

[31] Kief Morris. *Infrastructure as Code: Dynamic Systems for the Cloud Age*. Second edition. Beijing [China] ; Boston [MA]: O'Reilly, 2021. 399 pp. ISBN: 978-1-0981-1467-1.

[32] Progress Software Corporation. *Chef*. Version 18.5.0. July 9, 2024. URL: https://www.chef.io/.

[33] *Pulumi - Infrastructure as Code, Secrets Management, and AI*. Version 3.157.0. Pulumi, Mar. 18, 2025. URL: https://www.pulumi.com/ (visited on 03/23/2025).

[34] Ehsan Rasoulpour Shabestari and Alireza Shameli-Sendi. "An Intelligent VM Placement Method for Minimizing Energy Cost and Carbon Emission in Distributed Cloud Data Centers". In: *Journal of Grid Computing* 23.1 (Mar. 10, 2025), p. 12. ISSN: 1572-9184. DOI: 10.1007/s10723-025-09798-2.

[35] Red Hat. *Ansible*. Version 10.3.0. Aug. 13, 2024. URL: https://www.ansible.com/.

[36] Xavier Rival and Kwangkeun Yi. *Introduction to Static Analysis: An Abstract Interpretation Perspective*. Mit Press, 2020.

[37] SentinelOne. *Common Cloud Misconfigurations and How to Prevent Them*. SentinelOne. Oct. 4, 2024. URL: https://www.sentinelone.com/cybersecurity-101/cloud-security/cloud-misconfigurations/ (visited on 03/16/2025).

[38] Daniel Sokolowski and Guido Salvaneschi. "Towards Reliable Infrastructure as Code". In: *2023 IEEE 20th International Conference on Software Architecture Companion (ICSA-C)*. 2023 IEEE 20th International Conference on Software Architecture Companion (ICSA-C). Mar. 2023, pp. 318–321. DOI: 10.1109/ICSA-C57050.2023.00072.

[39] Daniel Sokolowski, David Spielmann, and Guido Salvaneschi. "Automated Infrastructure as Code Program Testing". In: *IEEE Transactions on Software Engineering* 50.6 (June 2024), pp. 1585–1599. ISSN: 1939-3520. DOI: 10.1109/TSE.2024.3393070.

[40] Daniel Sokolowski, David Spielmann, and Guido Salvaneschi. "The PIPr Dataset of Public Infrastructure as Code Programs". In: *Proceedings of the 21st International Conference on Mining Software Repositories*. MSR '24. New York, NY, USA: Association for Computing Machinery, July 2, 2024, pp. 498–503. ISBN: 979-8-4007-0587-8. DOI: 10.1145/3643991.3644888.

[41] Y. Sun. "Cloud and Edge Computing as Effective Trends in Business Model Innovation: A Bibliometric Review". In: *Journal of Software: Evolution and Process* 37.1 (2025), e2754. ISSN: 2047-7481. DOI: 10.1002/smr.2754.

[42] Vpn Monitor Research Team. *Report: Travel Reservations Platform Leaks US Government Personnel Data*. URL: https://www.vpnmentor.com/blog/us-travel-military-leak/ (visited on 04/11/2025).

Declaration of Authorship

I (Christoph Bühler) hereby declare,

- that I have written this thesis independently,
- that I have written the thesis using only the aids specified in the index;
- that all parts of the thesis produced with the help of aids have been declared;
- that I have handled both input and output responsibly when using AI. I confirm that I have therefore only read in public data or data released with consent and that I have checked, declared and comprehensibly referenced all results and/or other forms of AI assistance in the required form and that I am aware that I am responsible if incorrect content, violations of data protection law, copyright law or scientific misconduct (e.g. plagiarism) have also occurred unintentionally;
- that I have mentioned all sources used and cited them correctly according to established academic citation rules;
- that I have acquired all immaterial rights to any materials I may have used, such as images or graphics, or that these materials were created by me;
- that the topic, the thesis or parts of it have not already been the object of any work or examination of another course, unless this has been expressly agreed with the faculty member in advance and is stated as such in the thesis;
- that I am aware of the legal provisions regarding the publication and dissemination of parts or the entire thesis and that I comply with them accordingly;
- that I am aware that my thesis can be electronically checked for plagiarism and for third-party authorship of human or technical origin and that I hereby grant the University of St. Gallen the copyright according to the Examination Regulations as far as it is necessary for the administrative actions;
- that I am aware that the University will prosecute a violation of this Declaration of Authorship and that disciplinary as well as criminal consequences may result, which may lead to expulsion from the University or to the withdrawal of my title.

By submitting this thesis, I confirm through my conclusive action that I am submitting the Declaration of Authorship, that I have read and understood it, and that it is true.

Connection to other work

This thesis has been conducted in the context of a broader line of research that is currently under review. The work presented here forms a contribution to a joint submission titled "Efficient Pre-Deployment Validation of Infrastructure as Code," co-authored by David Spielmann (University of St. Gallen), Christoph Bühler (University of St. Gallen, author of this thesis), Matteo Biagiola (Università della Svizzera Italiana & University of St. Gallen), Daniel Sokolowski (Independent Researcher), Roland Meier (armasuisse), and Guido Salvaneschi (University of St. Gallen).

The concepts, implementation, and evaluation results developed throughout this thesis directly influenced the aforementioned paper, in which I am listed as a co-author. While the thesis was carried out independently, it represents a substantial contribution to the collective research efforts described in the collaborative publication.

Directory of writing aids

Aid	**Usage / Application**	**Affected Areas**
LanguageTool	Spellcheck	Whole document
Google Scholar	Literature Research	References
SciSpace	Literature Research	References
ChatGPT	Spellcheck, Outline, and Correction	Whole document
Cursor	Spellcheck, Grammar	Whole document

Table 8.1: Writing Aids (Art. 57 AB)