Politecnico
di Torino

1859

Master of Science in Computer Engineering

Master Degree Thesis

Validation and Verification of Infrastructure as Code

Supervisors
prof. Fulvio Valenza
prof. Guido Marchetto
prof. David Palma, NTNU

Candidate
Francesco SANTORO

Academic Year 2023-2024

Abstract

In recent years, the development of technologies such as Infrastructure as Code (IaC) and Policy as Code (PaC) has transformed modern Information and Communication Technology infrastructures into more software-based systems. This evolution has enabled faster deployment, scalability, and simplified network management. Moreover, the growing number of Infrastructure as Code (IaC)-based solutions has created a diverse landscape, necessitating that each organization determine the most suitable solution for its needs while ensuring policy compliance before provisioning and deploying the infrastructure.

PaC involves codifying security and compliance policies into executable code. By integrating policies directly into the infrastructure code, organizations can ensure that security and compliance requirements are automatically enforced, thereby reducing the risk of human error and enhancing overall governance. However, various PaC solutions tailor policy compliance checking to each specific IaC and Infrastructure Provider, leading to significant redundancy and complicating code comprehension for Security and Compliance teams.

In this thesis, we define and validate an Agnostic Policy as Code (APaC) tool, where policy rules are checked regardless of the infrastructure code platforms. We demonstrate the possible use cases through a Proof of Concept (PoC) using existing IaC tools and compare the results with widespread PaC tools, highlighting the benefits of an agnostic approach. Our analysis confirms the potential of abstracting policy rules across any IaC tool or infrastructure provider, thereby aiding various stakeholders in creating simpler and less redundant policies.

Preface

This thesis concludes my 2-year master's program in Computer Engineering, Computer Networks and Cloud Computing at Politecnico di Torino. In particular, the research presented herein has been conducted at Norwegian University of Science and Technology (NTNU) during my 10-month Erasmus+ program. The thesis has been supervised by Associate Professor David Palma from NTNU and Professors Fulvio Valenza and Guido Marchetto from Politecnico di Torino.

This project presented a significant challenge due to my initial lack of experience with the primary tools involved. This difficulty provided an opportunity to become familiar with unexplored tools and to acquire knowledge on new and highly relevant topics within the current landscape of Cloud Computing.

Acknowledgements

This thesis is a result of one of my biggest academic challenge so far, and it would not have been possible without the support I received over the past five years, including the initial three years of my bachelor's degree. To all those who have supported me, I extend my deepest gratitude.

Firstly, I would like to thank my supervisor, David Palma, for his patience, guidance and precious feedback. He encouraged me to fully utilize my skills and strive for excellence.

I wish to extend my gratitude to my home university, Politecnico di Torino, for granting me the opportunity to undertake a 10-month exchange program in Trondheim at NTNU. This experience has been invaluable to me.

I would also like to acknowledge my friends in Torino, Alliste, and Trondheim, who have stood by me through both difficult and joyful times. Special thanks also go to Edisu Piemonte for providing the financial support that enabled me to pursue my studies over the past five years.

Lastly, my heartfelt gratitude goes to my family -to my father Alessandro, my mother Loredana, my sister Emanuela, and my brother Simone- for their unwavering support and love.

Contents

Appendix

List of Figures

List of Tables

List of Algorithms

List of Acronyms

APaC Agnostic Policy as Code.

API Application Programming Interface.

AWS Amazon Web Services.

BGP Border Gateway Protocol.

CD Continuous Delivery.

CI Continuous Integration.

CI/CD Continuous Integration/Continuous Delivery.

CLI Command Line Interface.

CNCF Cloud Native Computing Foundation.

DevOps Development and Operations.

DSL Domain-Specific Language.

EC2 Elastic Compute Cloud.

GCP Google Cloud Platform.

HCL HashiCorp Configuration Language.

HTML HyperText Markup Language.

HTTP Hypertext Transfer Protocol.

HTTPS Hypertext Transfer Protocol Secure.

IaaS Infrastructure as a Service.

IaC Infrastructure as Code.

IDE Integrated Development Environment.

IDS/IPS Intrusion Detection/Prevention Systems.

INI Initialization.

IP Internet Protocol.

IT Information Technology.

JSON JavaScript Object Notation.

K8s Kubernetes.

NAT Network Address Translation.

NTNU Norwegian University of Science and Technology.

OPA Open Policy Agent.

OS Operating System.

PaC Policy as Code.

PE Policy Engine.

PoC Proof of Concept.

SDG Sustainable Development Goal.

SoA State of the Art.

SSH Secure Shell.

TDD Test-driven development.

UI User interface.

VCS Version Control System.

VM Virtual Machine.

VPC Virtual Private Cloud.

VPN Virtual Private Network.

YAML YAML Ain’t Markup Language.

Chapter 1

Introduction

This introductory chapter outlines the motivation behind the research presented in this thesis. It introduces the current landscape of Development and Operations (DevOps), Infrastructure as Code (IaC), and Policy as Code (PaC), and evaluates the primary issues that form the foundation for the subsequent work. The main research questions guiding this thesis are presented, emphasizing the key topics to be analyzed. Each chapter is briefly summarized, providing an overview of their content. Lastly, the chapter addresses the ethical and sustainability considerations of this thesis, discussing the main concerns related to our work and the DevOps field in general.

1.1 Motivation

IaC is the DevOps tactic of managing and provisioning infrastructure through machine readable definition files, rather than physical hardware configuration or interactive configuration tools [1]. The idea behind the IaC approach is that both writing and executing code in order to define, deploy and update the infrastructure [2]. Furthermore, IaC has become a crucial part of cloud computing. It frees professionals from performing manual, error-prone tasks; plus, it reduces costs and improves efficiency at all stages of the DevOps lifecycle [3].

Currently, several tools (such as Terraform [4], Ansible [5], Chef [6], Puppet [7], Packer [8], Cloudify [9]) and providers (such as Amazon Web Services (AWS) [10], Google Cloud Platform (GCP) [11], Azure [12], OpenStack [13], Docker [14]) support the principles of IaC. Some of these tools and providers address different aspects of IaC technology, while others focus on similar areas, as detailed in chapter 2. This diversity offers significant benefits to the IaC community by providing a wide range of tailored choices for organizations. However, it also complicates understanding and adoption, as the increasing number of available solutions makes selecting the most suitable one for each organization a complex task.

Automation with IaC and similar methods can enhance cost efficiency, productivity, and security, especially for organizations implementing hybrid cloud models. By automating tasks previously performed manually, operations become faster as these tasks are now managed by code. However, automation alone does not inherently address crucial areas such as compliance, governance, and standards. Therefore, while automation increases repeatability and speed, it does not guarantee correctness. [15]. Furthermore, according to a Unit 42 Cloud Threat Report from 2020 [16], while IaC offers Security Teams a programmatic way to enforce security standards, much of its power remains largely unharnessed and, in many cases, it is simply not secure. The authors of the report analysed 200000 different IaC files, and these were the main results [16]:

- Services running with the highest privileges (root).
- Exposure of unneeded resources like port 22 (SSH).
- Hardcoded secrets.
- Use of HTTP and HTTPS on an external load balancer where HTTP does not redirect to HTTPS.

In addressing these issues with an automated solution, Policy as Code defines, updates, shares, and enforces policies using code. The emphasis on code is crucial, as this approach encodes policies through a programming language. These codified policies can facilitate the enforcement or testing of automation scripts to ensure adherence to the defined policies. PaC is not a novel concept, and some companies utilize systems to implement it. However, the challenge lies in the execution, as many PaC implementations are tailored to specific use cases and designed to function only with certain environments or technologies. Specifically, policies are often embedded within business logic code or enforced manually, resulting in the same policies being written in different languages, stored in multiple code repositories, and managed by various teams. This fragmentation can lead to inconsistent interpretations of the same policy, and any changes or new policy versions may take weeks or months to implement and test, complicating enforcement. [15].

Figure 1.1 shows the typical workflow between a Compliance Team using a Policy as Code approach combined with the development and deployment part of the infrastructure. The compliance checking is performed before the infrastructure is ultimately deployed. Furthermore, it is worth noticing how Compliance and Developers Teams should work independently from each other, empowering the principles of DevOps. This also highlights the need for these two actors to try to keep their workflow separated and independent from each other. This concept is one of the primary principles used as a basis throughout this entire thesis.

Within each PaC tool, every policy is verified against the infrastructure provisioned by the IaC tool. Although both organization-specific and open-source PaC tools appear robust and reliable, particularly concerning the variety of IaC tools and infrastructure providers they support, they lack any form of abstraction for either.

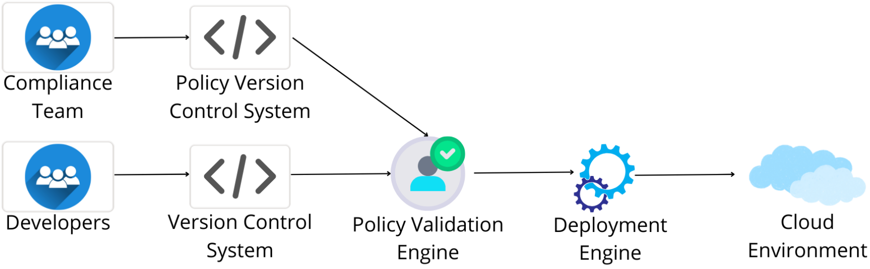

Figure 1.1: Policy as Code (PaC) workflow, adapted from [15]

The issue with this approach is that to implement the same policy, these PaC tools generate as many policy-checking files as there are IaC tools and infrastructure providers supported. For example, if a PaC tool supports 10 different IaC tools and 10 different infrastructure providers, it would result in 100 distinct files for checking the same policy. This redundancy becomes evident upon observing the similarity of these files, as they only differ in the specific functions or methods tailored to each particular use case.

In this thesis, we propose a policy checking architecture agnostic to both the Infrastructure as Code tool and the infrastructure provider. This abstraction level allows to significantly reduce the number of lines of code as well as to better understand how IaC tools, PaC tools and infrastructure providers work together. A Proof of Concept (PoC) is implemented and the results validated against other PaC tools. This PoC may be used as a base to define an actual PaC tool. For the rest of this thesis, this PoC will be referred to as Agnostic Policy as Code (APaC); the name emphasizes the main feature provided by our tool.

1.2 Research questions

Considering the importance of IaC, the needs for correctness and policy compliance, the variety of both IaC and PaC tools, within this thesis, we expect to find answers to the following research questions:

- **RQ1:** What is the current status of Infrastructure as Code, which tools are most popular, and how are they used in practice?
- **RQ2:** What is the current status of Policy as Code, which tools are most popular, and how are they used in practice?
- **RQ3:** Which tools do we need to define a domain-agnostic architecture and how would this be used in practice?

1.3 Thesis structure

The remaining chapters of this thesis are organized as follows:

- **chapter 2** presents the background needed to understand what the main technologies and the respective primary tools are. In particular, the practices analysed are DevOps, two popular IaC tools which will be later part of APaC, two infrastructure providers which will be used to deploy the network infrastructure, and the paradigm of PaC, where its main features and principles will be assessed.
- **chapter 3** presents the current State of the Art of PaC. In particular three PaC solutions are explained and their main benefits and drawbacks discussed, making clear what the primary open issues are nowadays.
- **chapter 4** summarises the methodology applied for the research and the implementation parts of this project.
- **chapter 5** explains the implementation of APaC and its domain-agnostic architecture, as well as its importance in the current Policy as Code landscape. This chapter also includes a validation and evaluation of the solution's potential, providing a comparative analysis with existing alternatives.
- **chapter 6** discusses the possible implications and avenues of the research presented and summarizes main contributions. It also discusses the limitations and the possible future research we may have using APaC as a starting point for an actual PaC tool, where the tool-independence appears as the main feature and contribution.

Furthermore, **Appendix A** includes the code details from APaC.

1.4 Ethics and Sustainability Aspects of the Thesis

The domain-agnostic Policy as Code approach can enhance collaboration between IaC stakeholders and the PaC team, aligning with DevOps principles. This method promotes better interoperability and collaboration, addressing social sustainability issues [17]. Moreover, this thesis contributes to the following Sustainable Development Goals (SDGs) [18]:

- **Goal 8: Decent work and economic growth**. Our approach abstracts infrastructure and policy definition keywords, helping stakeholders in the DevOps field to increase their understanding by separating IaC and PaC concepts.

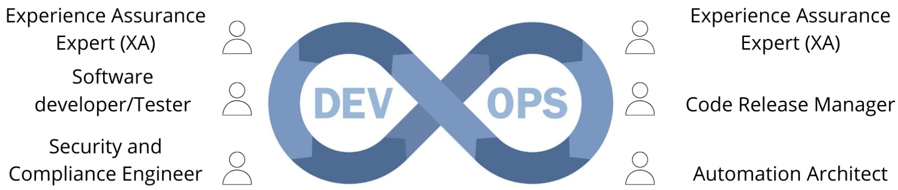

Figure 1.2: DevOps Stakeholders: different actors, dealing with different aspects of the DevOps paradigm, work together to shorten the system developments lifecycle. Adapted from [19]

Figure 1.2 illustrates the main stakeholders in a DevOps scenario, who typically do not speak the same technical language. By leveraging DevOps principles and the abstraction provided by APaC, we intend to improve collaboration and minimize interoperability issues between different actors. This contributes to a more efficient working environment.

- **Goal 9: Industry, innovation and infrastructure**. By implementing policies independently from the infrastructure code, we aim to foster trust among various human actors. Our approach helps detecting issues before deploying network infrastructure, facilitating understanding and resolution of such violations. Although automation speeds up network infrastructure creation and deployment, it is important to balance automation with human involvement. The principles of this thesis do not aim to eliminate human roles but to facilitate better understanding and integration between IaC and PaC teams within DevOps. Another challenge that APaC, and PaC tools in general, may face is the risk of malicious users exploiting common policies checked against IaC environments. If the Security and Compliance Team has not yet addressed certain security issues, these users could potentially gain access to a list of all the vulnerabilities within an organization. However, the way APaC is intended to work is not entirely new in the DevOps landscape; it is an improvement to existing solutions and, therefore, already a well-known potential security issue.

In conclusion, APaC is intended to provide an efficient way for Security and Compliance team to validate and verify policy compliance of infrastructure defined as code configuration files, enhancing overall collaboration and efficiency in DevOps practices. Regarding the ethical concerns, instead, it is important to keep in mind that removing humans from the DevOps lifecycle does not represent the goal of our tool; it represents an ethical concern which shall not be underestimated.

Chapter 2

Background

This chapter provides an overview of different terms and specifications used in this thesis. In particular, the main technologies such as Development and Operations (DevOps), Infrastructure as Code (IaC), Policy as Code (PaC) are explained in general. The main tools used throughout this project are also presented, and their main features assessed.

2.1 DevOps

A new movement denominated as Development and Operations is promoting the continuous collaboration between developers and operations staff. In this scenario, automating the provisioning of the infrastructure accelerates the deployment process in the software delivery cycle [2].

DevOps refers to a collection of terminology, procedures, techniques, and ideas aimed at improving the efficiency, reliability, security and speed of software development. The idea of automation is central to the DevOps philosophy. DevOps integrates automation throughout the entire software delivery pipeline, encompassing build, test, deployment, and monitoring processes.

One critical practice within DevOps is Continuous Integration (CI), which involves the frequent integration of code changes into a shared repository. This practice ensures that developers merge their code changes into a central repository multiple times during the development process. Each integration is followed by automated tests to ensure code quality and identify issues early. Continuous Delivery (CD) extends the principles of CI by automating the deployment of code changes to production environments once they have passed automated testing. This practice enables organizations to deploy changes to production rapidly and frequently.

Infrastructure as Code (IaC) is a pivotal component of DevOps. It involves the definition, management, and provisioning of computing infrastructure through machine-readable script files, as opposed to manual hardware configuration. IaC is a fundamental enabler within the DevOps methodology, allowing for automated and repeatable infrastructure deployment [20].

According to Hashicorp's 2021 State of the Cloud Survey Report [21], 76% of IT enterprises have embraced a multi-cloud strategy. The report also suggests that the shift to a multi-cloud environment is a dominant strategy that most enterprises are adopting. Within this framework, IaC has become a crucial part of cloud computing, since it frees professionals from performing manual, error-prone tasks; plus, it reduces costs and improves efficiency at all stages of the DevOps lifecycle [3]. On the other hand, implementing IaC requires understanding new tools and languages, which can be challenging for teams not already familiar with these technologies; moreover, if not managed properly, IaC scripts can inadvertently expose sensitive information. Bugs and security vulnerabilities in IaC scripts can lead to misconfigured infrastructure, creating potential security gaps.

2.2 Infrastructure as Code

Information Technology (IT) systems are not just business vital, but they are the business for organizations such as Amazon [22], Netflix [23], and Google [24], among others. Every day, such organizations' systems process hundreds of millions of data points [20]. The primary objectives for employing Infrastructure as Code within these organizations reflect a strategic vision to transform IT infrastructure into a facilitator and enabler of change, rather than an impediment. By leveraging IaC, these organizations dynamically adjust their infrastructure to meet evolving business needs, encouraging innovation and agility.

Moreover, IaC allows users to define, set up, and manage their infrastructure on their own, greatly reducing the need for IT staff. This self-service approach speeds up the deployment of resources and improves operational efficiency, enabling faster responses to business demands.

The main feature of IaC relies in the support of the management of the entire lifecyle of a computing environment consisting of infrastructure, software/platforms, and applications. Infrastructure includes the fundamental computing resources such as server, networks, and storage. Instead, software/platforms are used to deploy, run, and manage applications, such as programming languages, frameworks, libraries, services, and tools. Finally, application-specific capabilities are defining the desired state of the application deployment, by deploying, (re)configuring, un-deploying the application using its deployment definition [25].

According to Bali *et al.* [26] IaC can also be explained as a technique of defining and deploying infrastructure, such as networks, virtual machines, load balancers, and connecting topologies, using the DevOps methodology and versioning with a descriptive model.

Furthermore, IaC replaces the conventional processes used to manage a computing environment with a process that enables applying software engineering practices. Instead of a low-level shell scripting languages, the IaC process uses high-level domain-specific languages that can be used to design, build, and test the computing environment as if it is a software application/project. The conventional management tools such as interactive shells and UI consoles are replaced by the tools that can generate an entire environment based on a descriptive model of the environment [2].

It also allows people to apply software development tools such as Version Control System (VCS). It also opens the door to exploit development practises such as Test-driven development (TDD) and Continuous Integration/Continuous Delivery (CI/CD). In particular, the practice of CI/CD enables ongoing improvements, thus avoiding the risks and costs associated with large-scale, infrequent updates [27], as well as enabling safe collaboration on infrastructure; this capability allows teams to work together on infrastructure development, with each member having individualized copies of the code.

As outlined by Guerriero *et al.* [1] Infrastructure as Code is, therefore, the DevOps practice of describing complex and (usually) cloud-based deployments by means of machine-readable code. The main enabler for IaC has been the advent of cloud computing, which, thanks to virtualization technologies, allowed the provisioning, configuration and management of computational resources to be performed programmatically.

As stated before, one of the main takeaways of IaC is that it allows users to define, set up, and manage their infrastructure independently, significantly reducing the need for IT staff. However, this reduction in human presence can also be seen as a drawback. IT staff often bring a wealth of experience and expertise in infrastructure management; thus, reducing their involvement can lead to a loss of critical oversight and guidance, potentially resulting in suboptimal configurations and missed opportunities for optimization. Additionally, automation through IaC can efficiently handle predefined tasks but may lack the contextual understanding that human judgment provides. IT staff can make nuanced decisions based on a broad understanding of the organization's needs and priorities. Moreover, heavy reliance on IaC tools and scripts can create a single point of failure. If these tools encounter bugs or compatibility issues, it can disrupt the entire infrastructure management process.

In conclusion, IaC is a powerful approach within the DevOps landscape, offering significant benefits in terms of automation, efficiency, and consistency. However, it is crucial to consider the potential challenges it presents, such as increased complexity, the need for specialized skills, and the risk of misconfigurations. By acknowledging and addressing these issues, organizations can fully leverage IaC's advantages while mitigating its drawbacks.

2.2.1 Different kinds of Infrastructure as Code tools

Subsequently to the advent of cloud computing, many different languages and corresponding platforms have been developed, each of which deals with a specific aspect of infrastructure management [1]:

- Tools able to provision and orchestrate virtual machines (e.g., Cloudify [9], Terraform [4]).
- Tools doing a similar job with respect to container technologies (e.g., Docker Swarm [28], Kubernetes [29]).
- Machine image management tools (e.g., Packer [8]).
- Configuration management tools (e.g., Ansible [5], Chef [6], Puppet [7]).

Instead, according to Sandobalín *et al.* [2], the IaC approach supports two different kinds of tools:

- **Code-centric tools** use scripts to specify the creation, updating and execution of cloud infrastructure resources. Since each cloud provider offers a different type of infrastructure, the definition of an infrastructure resource (e.g., Virtual Machine (VM)) implies writing several lines of code that greatly depend on the target cloud provider. A well-known code-centric tool is Ansible.
- **Model-driven tools**, which, abstract the complexity of using scripts through the high-level modelling of the cloud infrastructure (e.g., Argon [30]).

The same article asserts that there are two main stages defined in the IaC process: definition and provisioning. The former writes/models the infrastructure resources that will be provisioned on a cloud platform, whereas the latter employs IaC tools to execute the infrastructure and hence orchestrate cloud infrastructure provisioning.

The author also states that the DevOps community has developed several tools whose purpose is to manage the infrastructure provisioning of different cloud providers, such as Ansible and Terraform, and tools with which to install and manage software in existing servers, such as Chef and Puppet.

Alternatively, according to Kumara *et al.* [25], there are two main programming models for IaC languages: declarative and imperative (procedural). In the declarative model, the developers define the desired end state of the environment and let IaC tools determine how to achieve the defined state. In the imperative model, the developers need to specify the process that transforms the current state of the environment to the desired end state as an ordered set of steps. For instance, tools like Puppet uses a declarative style, whereas tools like Chef and Ansible use an imperative style.

As a result, it is clear that there is not a unified way or metric to define and distinguish among each IaC tool.

Moreover, this also explains why the landscape of IaC languages and tools is currently jeopardized by the technology heterogeneity and by the huge number of available solutions. On the one hand this is the result of the great interest that IaC has raised; also, all these nuances provide several alternatives to the users, according to their needs. On the other hand, it complicates the understanding and adoption of this new technology. Shedding light on the IaC current adoption, issues and challenges, is thus fundamental towards bringing IaC to maturity and ease its further development [1].

2.3 Infrastructure as Code's current landscape

Infrastructure as Code is a transformative approach in the field of IT infrastructure management that leverages the principles of software development to manage and provision computing resources. IaC enables the automation of various aspects of infrastructure management, including the provisioning of resources and configuration of systems and many more, as it clearly emerges when looking at the current and always evolving landscape provided by the Cloud Native Computing Foundation (CNCF) [31]. Among the multitude of tools highlighted, this thesis is focused on Terraform and Ansible. These have been selected due to their widespread adoption, ease of use, and the large amount of available dependencies they offer.

2.3.1 Terraform

Terraform, an IaC solution developed by HashiCorp [32], enables users to specify cloud and on-premises resources in configuration files that are simple to understand and can be used, shared, and modified. This approach allows for continuous provisioning and maintenance of infrastructure using a consistent strategy throughout its lifecycle. With Terraform, tasks such as constructing, upgrading, and maintaining infrastructure are significantly simplified. The configuration files are written in HashiCorp Configuration Language (HCL), which is a declarative language that specifies the desired end-state for the infrastructure [20].

Terraform allows to manage the whole infrastructure, from end to end. However, it does not replace the tools that can be used for managing the configuration of VMs. Moreover, Terraform is particularly advantageous when utilizing multiple cloud providers and managing cross-cloud dependencies. By reducing the complexity of administration and orchestration, operators can design and manage large-scale multicloud systems more efficiently.

Terraform's usefulness is highlighted by several key features. It goes beyond simple configuration management to include orchestration, offering complete infrastructure solutions. It supports unchangeable infrastructure, allowing for easy and consistent configuration changes. The HCL is made to be easy to understand, and switching between different providers is straightforward. Additionally, Terraform supports a wide range of cloud service providers, including AWS [10], Microsoft Azure [12], GCP [11], DigitalOcean [33], Kubernetes, Helm [34] and others.

Utilizing Application Programming Interfaces (APIs), Terraform is able to build and manage resources on cloud platforms and other services. In its current state, Terraform is compatible with the vast majority of API-supported platforms and services.

Using Terraform has several advantages over manually managing the infrastructure. In particular, Terraform can manage infrastructure across multiple cloud platform, providing a unified solution for diverse environments. Secondly, its human-readable configuration language facilitates the quick and efficient writing of infrastructure code. Additionally, Terraform's state management feature allows tracking of resource changes throughout deployments, ensuring consistency and reliability [20].

The Terraform workflow is divided into more stages:

- **Write**: It is possible to establish resources that are shared across several cloud providers and services. Here, users define their infrastructure in Terraform configuration files using HCL. These files specify the resources and components required in the infrastructure, such as servers, databases, and networking components.

- **Init**: the command `terraform init` is executed in this stage. This command initialize the working directory containing the Terraform configuration files, as well as downloading the necessary provider plugins (e.g., AWS, Azure, GCP) and preparing the environment.

- **Plan**: the command `terraform plan` is executed in this stage. Terraform gives an execution plan that outlines the infrastructure that it will construct, update, or delete depending on the current infrastructure and the current configuration settings. This plan is generated based on the existing infrastructure.

- **Apply**: the command `terraform apply` is executed in this stage; after receiving permission, Terraform will next carry out the predetermined operations in the appropriate sequence, taking into account the interdependences between the resources. This command applies the changes required to reach the desired state of the configuration, by creating, updating or deleting infrastructure resources.

- **State management**: Terraform keeps track of the infrastructure state using a state file *terraform.tfstate*. This file maps the real-world resources to the configuration and keeps track of metadata and dependencies. Moreover, the state file is critical for tracking changes and should be stored securely.

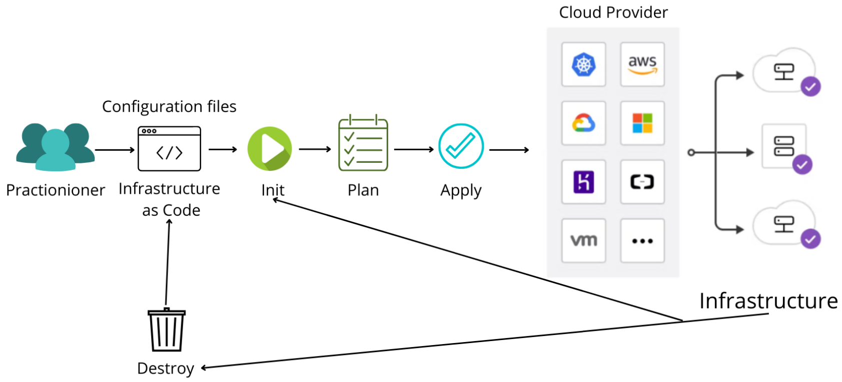

Figure 2.1: Terraform workflow, adapted from [35]

- **Destroy**: the command `terraform destroy` is executed in this stage; this will destroy Terraform-managed infrastructure or the existing environment created by Terraform.

Figure 2.1 depicts the typical workflow in Terraform. Firstly, the infrastructure is defined in Terraform configuration files using HCL; next, stages init, plan and apply are executed. Finally, the infrastructure is created or, in case the infrastructure already exists, the changes applied against the specified cloud provider. At any given time the infrastructure may be modified, by modifying the configuration files, or destroyed.

2.3.2 Ansible

Ansible is categorized as an infrastructure automation tool that enables the rapid automation of system administration tasks. It allows for the deployment of Infrastructure as Code both on-premises and on major public cloud providers. Managing containers within an organization can be a challenging task, particularly when performed manually with repetitive tasks. Often, there is a need to run a container on workstations or across server fleets. Ansible streamlines this workflow and automates tedious and complex tasks, offering new methods to distribute applications in platform-independent formats [36].

Among the primary benefits of using Ansible there are its simplicity, power, cross-platform compatibility, and compatibility with existing tools. Firstly, Ansible's code is written in YAML, a human-readable data serialization language that is widely recognized and easy to learn. This language is commonly used for configuration files and in applications where data needs to be stored or transmitted, making it accessible for users. Additionally, Ansible is a robust and well-proven solution that excels in configuration management, workflow orchestration, and application deployment. Its powerful capabilities allow it to handle complex tasks efficiently and reliably. Furthermore, Ansible's agent-less nature ensures support for all major operating systems, as well as physical, virtual, cloud, and network providers. This cross-platform compatibility means that Ansible can be used in diverse environments without the need for additional agents. Finally, Ansible's ability to integrate seamlessly with existing tools makes it easy to standardize and streamline the current environment. This compatibility ensures that users can adopt Ansible without disrupting their existing workflows and infrastructure [36].

Ansible's three prominent use cases are:

- **Provisioning** involves the setup of IT infrastructure, a critical task for system administrators aiming to manage a uniform fleet of machines. Some practitioners continue to utilize software for creating workstation images. However, a limitation of imaging technology is that it captures only a snapshot of the machine at a specific moment. Consequently, software must be reinstalled each time to accommodate modern critical activation systems or to apply the latest security patches. Ansible is highly effective in automating this process.

- **Configuration management** is the process of maintaining systems and software in a desired and consistent state. It ensures the up-to-date and consistent operation of a fleet, including the coordination of rolling updates and the scheduling of downtime. Ansible allows for the verification of the status of managed hosts and the implementation of actions on a subset of them. A wide variety of modules is available for the most common use cases.

- **Application Deployment** is the process of publishing software between testing, staging, and production environment. For example, application's Continuous Integration/Continuous Delivery workflow pipeline can be automated with Ansible.

Ansible requires only OpenSSH [37] and Python [38] to be installed. OpenSSH is used for connection and one login user, whereas the local Python interpreter in the target node will execute the Ansible commands.

Regarding its architecture, as illustrated in Figure 2.2, Ansible typically requires two or more hosts: one that executes the automation, known as the **Ansible Control Node**, and one or more hosts that receive the actions, known as **Target Node**. In this particular example, there is one Control Node which applies some rules against three distinct Target Nodes.

The Ansible Control Node applies the rules defined in the YAML playbook file

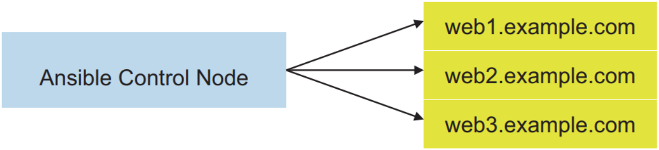

Figure 2.2: Ansible architecture [36]

Listing 1 This playbook, written in YAML, defines one task which is called "hello" and prints the message "Hello" every time it is executed. It is applied against each host defined in the Inventory. Adapted from [36]

```yaml
- name: example
  hosts: all
  tasks:
    - name: hello
      ansible.builtin.debug:
        msg: Hello
```

Listing 2 This Inventory file, written in JSON, defines three distinct Target Nodes to which the playbook is applied. Beside JSON, the Inventory file can also be written in INI [39] or YAML format. Adapted from [36]

```json
{
  "all": {
    "hosts": [
      "web1.example.com",
      "web2.example.com",
      "web3.example.com"
    ]
  }
}
```

(example at Listing 1) against each Target Node, defined in the Inventory file (example at Listing 2). The Ansible Playbook is the automation blueprint and has a step-by-step list of tasks to execute against the target hosts. Moreover, the Ansible Control Node directs the automation and effectively requires Ansible to be fully installed inside. The Ansible Target Node requires only a valid login to connect.

2.3.3 Other tools

Hereby is provided a short description of other widely used IaC tools.

- **CloudFormation** [40] is developed by Amazon Web Services (AWS) and it is a service that allows users to define and provision AWS infrastructure using JSON or YAML templates. It is tightly integrated with AWS, making it a powerful tool for users heavily invested in the AWS ecosystem. The configuration files are written in YAML and JSON.

- **Puppet** is developed by Puppet, Inc. It is a configuration management tool that automates the provisioning, configuration, and management of infrastructure. It uses a declarative language to describe system state. The configuration files are written in Puppet Domain-Specific Language (DSL), which is based on Ruby [41].

- **Chef** is developed by Progress [42]. It is a powerful configuration management and automation tool that is widely used to manage and automate the infrastructure of complex IT environments. It manages infrastructure by writing “recipes” and “cookbooks”. The latter are defined using Ruby-based DSL.

- **Pulumi** [43] is developed by Pulumi Corporation. It is an open-source tool that allows users to define and manage cloud infrastructure using real programming languages like TypeScript [44], JavaScript [45], Python, Go [46], and .NET [47].

- **Kubernetes (K8s)** was originally developed by Google, now maintained by the Cloud Native Computing Foundation (CNCF). It facilitates the deployment, scaling, and management of containerized applications in a regulated and automated manner. Essentially, Kubernetes functions as a container orchestrator. Utilizing container runtimes such as Docker, code, dependent libraries, and runtime environments can be packaged into an image, which is then executed to create containers. Additionally, Kubernetes enables resource management, the grouping of containers to form clusters, and other related functionalities. The configuration files are written in YAML and JSON.

- **SaltStack** [48] is a versatile tool with a wide range of use cases, primarily in the fields of configuration management, automation, and remote execution. SaltStack’s declarative configuration management allows administrators to define and enforce the desired state of systems, ensuring uniformity and reducing configuration drift. It can also automate the deployment of applications, libraries, and updates, making it efficient to manage software across a large infrastructure. The configuration files are written in YAML.

2.4 Infrastructure Providers

In this section the main infrastructure providers used throughout this thesis are introduced, and their main features highlighted. In particular, these tools are used during the implementation of APaC, presented later on.

2.4.1 OpenStack

OpenStack [13] is an open-source cloud computing platform providing a suite of software tools building and managing both public and private cloud. It plays a significant role as a cloud provider by offering an Infrastructure as a Service (IaaS) solution, enabling users to deploy and manage large networks of VMs and other resources.

As a cloud provider, OpenStack offers several benefits and features that makes it a popular choice for organizations looking to build and manage cloud environments. Specifically, one of the main feature of OpenStack is its flexibility and customization which allows users to deploy only the components they need. Instead its horizontally-scalable design, allows organizations to add more compute, storage, and networking resources as needed to handle increased workloads.

Even though OpenStack is a powerful cloud computing platform, it requires robust security measures to protect data and applications.

2.4.2 Docker

Docker [14] is an open-source platform designed to automate the deployment, scaling, and management of applications using containerization. Containers are lightweight, standalone, and executable software packages that include everything needed to run an application, such as code, runtime, libraries, and system tools. Docker containers are designed to run consistently across various computing environments, from local development machines to production servers in data centers or cloud environments.

Specifically, Docker encapsulates an application and its dependencies into a single container, ensuring consistent runtime environments. Containers can run on any system supporting Docker, making applications easily portable across different environments; moreover, each container runs in its own isolated environment, which enhances security and stability.

Docker itself is not a cloud provider but a platform that can be integrated with cloud services to provide containerized solutions. When combined with cloud infrastructure, Docker enhances the deployment and management of applications. In particular, in the context of cloud computing Docker facilitates hybrid and multi-cloud strategies by allowing applications to run consistently across different cloud environments. Organizations can deploy containers on various cloud platform (e.g., AWS, GCP, Azure) without modification.

Furthermore, Docker is a crucial component of the DevOps toolkit. Automation tools like Ansible, Chef and Puppet can use Docker to provision and manage containerized environments.

2.5 Policy as Code

Policy as Code (PaC) is an approach to policy management where policies are defined, updated, shared, and enforced using code. This method automates the compliance process by translating business logic from spoken language into code [15]. Additionally, it helps decoupling policy from an application's business logic. This approach offers several advantages.

Firstly, it enables the adoption of software development best practices, ensuring that policies are created and maintained with the same rigor as software code. By automating the testing of policies, PaC facilitates scalability, allowing policies to be applied consistently across large environments. PaC is also useful to enforce style guides and security rules automatically, enhancing the overall quality and security of the policies. It also provides traceability for compliance, ensuring that all changes to policies are documented and auditable [15]. Furthermore, PaC centralizes the rules, control, and management of policies, simplifying governance and oversight. By codifying policies, it allows them to be stored in Version Control Systems, which supports collaboration and historical tracking of policy changes. Finally, the PaC approach is consistent, recursive, and cost-effective. It ensures that policies are applied uniformly, can be repeatedly enforced as necessary, and reduces the costs associated with manual policy management.

PaC also requires defining (or codifying) policies using programming languages like Python, YAML, or Rego [49]. It also requires a Policy Engine (PE) to enforce the policies. This engine can be a built-in solution or use a different platform or agent for policy enforcement that is decoupled from the application or platform.

2.5.1 What is a policy?

A policy is a rule, condition, or instruction governing operations or processes. Another definition of policy is a set of rules or guidelines for an organization, people, or process to achieve compliance, standards, or consistency.

According to Matharu [15] policies can be either static or dynamic. Static policies are evaluated before execution; for example, it might test whether a device or resource name adheres to a naming convention before provisioning the device or resource. Whereas, dynamic policies are evaluated and enforced during runtime. For example, it can check whether user data is created, moved, or saved from a defined geographic zone at runtime.

2.5.2 Challenges with traditional policy enforcement

As stated by Matharu [15], conventional policy enforcement is manual or semi-automated and does not scale well. Each development or application team embeds some policy-enforcement code within its applications. This code is not easily trackable or auditable because every team implements it as it sees fit due to a lack of framework definition. Each organization follows certain practices and processes while developing and delivering software. Some must comply with industry-recognized frameworks such as SOC 2 [50], CIS [51], PCI DSS [52], or ISO 27001 [53].

Moreover, traditional policy definition and enforcement are manual processes. A compliance team drafts business requirements with specific rules that everyone is expected to follow, but this approach faces several challenges. Policy documents are continuously updated while Development Teams work against them, and they lack a framework for implementation, leading to after-the-fact, manual testing. This process is not scalable and relies heavily on human interpretation for enforcement. Significant changes may be needed to update policies, which can be both painful and wasteful, and manual changes can have unintended consequences.

The absence of a framework also complicates auditing changes, further hindering effective policy management.

2.5.3 Why use Policy as Code?

In this scenario, PaC addresses the weaknesses of the traditional enforcement method by automating the definition and enforcement of policies through a specific technology platform.

PaC simplifies the creation of test cases for policies and automates their checking and enforcement, in this way policies can be validated before deployment, ensuring they are correct and functional. Furthermore, environments created through automation become more secure, scalable, consistent, and preventative. PaC also facilitates easy updating, maintenance, and versioning of policies.

Common use cases for PaC include provisioning and managing cloud resources consistently and efficiently through IaC policies, applicable to both on-premises and public cloud environments. It is also used for authorization and access-control policies, as well as security policies that encompass network and endpoint protection. Operational best practices, such as configuration-management policies, are another area where PaC is beneficial [15].

2.5.4 Policy engine

Enforcing policies is as important as defining and documenting them. PEs provide the capability to systematically check if a rule is broken. A PE includes the mechanisms to automatically check logical inconsistencies, syntax errors, and missing dependencies. The PE takes decisions by evaluating inputs against policies and data. PEs should be generic enough to be applied to different scenarios, combining context-specific data with the higher-level policies, to enforce them according to each specific context [54]. PaC and PE can be used in IaC platforms to enforce infrastructure provisioning and deployment policies. IaC software might query the PE to take decisions before provisioning (e.g. depending on the type of node, storage, network dependencies, and application being targeted); thus, they also help restricting access to infrastructure and enforcing rationalization policies.

2.5.5 Why is policy decoupling important?

Software services should allow policies to be specified declaratively, updated at any time without recompiling or redeploying, and enforced automatically (which is especially valuable when decisions need to be made faster than humanly possible).

Decoupling policy helps building such software services at scale, makes them adaptable to changing business requirements, improves the ability to discover violations and conflicts, increases the consistency of policy compliance, and mitigates the risk of human error. Policies can adapt more easily to the external environment, factors that the developer could never have imagined at the time the software service was designed [55].

Figure 2.3 shows an example of decoupled PE, where the decoupling refers to the separation of policy definitions from the application's business logic. It is important to notice that when a query is submitted to the PE, it is evaluated against the pre-defined policies, previously established by the Compliance Team. The PE then provides a decision, indicating whether the query meets the requirements specified by the policies. This approach ensures that policies governing application behavior, regulatory compliance, and resource management are defined and managed independently from the code that executes the core functions of the application.

2.6 Discussion

At a first glance, the correlation among these technologies may appear subtle. However, with the proliferation of IaC solutions within the CNCF, it becomes evident that a mechanism for ensuring compliance with organizational policies and

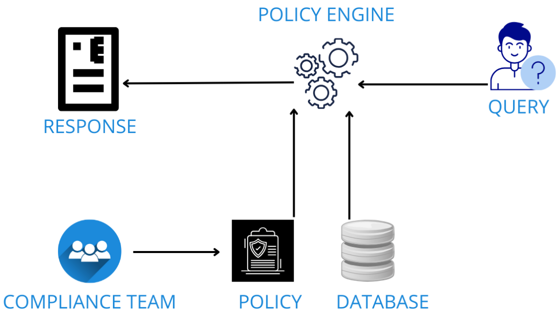

Figure 2.3: Policy as Code (PaC) policy decoupling, adapted from [15]

requirements throughout the software delivery lifecycle is imperative. PaC provides an efficient and automated way of verifying and applying such policies.

As stated before, the primary objective of this thesis is to develop a PaC tool capable of conducting compliance checks required by IaC tools in a more abstract manner by dissecting the fundamental network components underlying each infrastructure.

Consequently, a thorough comprehension of tools such as Terraform and Ansible, alongside the PaC tools introduced in chapter 3, including their main limitations, is the first step towards the creation of a robust PaC solution, that works regardless of the IaC it evaluates.

Chapter 3

State of the Art

This chapter provides a brief overview of the latest developments relevant to our research questions, required for the definition of APaC provided in chapter 5. The primary PaC solutions are presented, discussing their functionality in terms of supported IaC tools, ease of implementing new ones, and the level of abstraction they provide both for the IaC tools and the infrastructure providers. Since our main point of interest is the abstraction of principles and best practices of policies, each tool is evaluated and the best-fitting solution assessed accordingly.

3.1 The Cloud Native Landscape

Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach. These techniques allow systems to be independent, strong, easy to manage, and monitor. Along with strong automation, they let engineers make important changes often and reliably with little effort.

The Cloud Native Computing Foundation aims to promote this approach by supporting an ecosystem of open-source, vendor-neutral projects [31].

Infrastructure management involves several key areas, each supported by specific tools. For instance, Provisioning serves as the first layer in the cloud-native landscape, featuring tools designed to automatically configure, create, and manage a cloud-based network infrastructure[1]. The layer also extends to security with tools enabling policy setting and enforcement, embedded authentication and authorization, and the handling of secrets distribution.

Tools in the Automation and Configuration area are part of the Provisioning layer. They accelerate the creation and configuration of compute resources such as VMs, networks, firewall rules, and load balancers. Tools in this category either manage

---

[1] The term Infrastructure includes the elements belonging to the lower layers of the application stack as well as the upper ones, such as computer networks, firewall, load balancers, certification authorities, databases, web server.

different aspects of provisioning or control everything end-to-end. They enable engineers to build computing environments without human intervention; by codifying the environment setup, it becomes reproducible with a single click. Although these tools may take different approaches, their common goal is to reduce the workload required to provision resources through automation [31]. Examples of these tools are Ansible [5], Chef Infra [56], Cloudify [9], OpenStack [13], SaltStack [48], Terraform [4].

Another important layer of Provisioning is Security and Compliance. Such tools help harden, monitor, and enforce platform and application security. From containers to Kubernetes environments, these tools allow to set policy (for compliance), get insights into existing vulnerabilities, catch misconfigurations, and harden the containers and clusters. In particular, to run containers securely, they must be scanned for known vulnerabilities and signed to ensure they haven't been tampered with [57]. Some of these tools are rarely used directly. Trivy [58], Clair [59], and Notary [60], for example, are leveraged by registries or other scanning tools. Others represent key hardening components of a modern application platform. Examples include Falco [61] or Open Policy Agent (OPA) [62]. Other important tools in this area are Kics [63] and Checkov [64].

The CNCF also classify tools belonging to other layers, such as Runtime, Orchestration and Management, App Definition and Development, Observability and Analysis [57].

3.2 Policy as Code Solutions

Currently, multiple Policy as Code solutions are available, with some tailored specifically for certain infrastructure providers or IaC tools, such as HashiCorp Sentinel [65], Chef InSpec [66], Pulumi Crossguard PaC [67]. Conversely, other solutions provide a higher level of abstraction, like Kics, Checkov and especially OPA, which aim at supporting different IaC tools and infrastructure providers. The continuous development in this field allows more IaC tools to assess and verify their policies.

Among these tools and not, there are some available for implementing PEs. The importance of this feature is due to the fact that PaC and PE can be utilized in IaC platforms to enforce policies for infrastructure provisioning and deployment [54]. Some of these are Kyverno [68], Pulumi Crossguard, Azure PaC Microsoft [69]and Sentinel.

3.2.1 Checkov

Checkov is a static code analysis tool that scans for security vulnerabilities. It was originally developed by Bridgecrew [70], but it is currently owned by Prisma Cloud [71]. Checkov enables the identification of vulnerabilities prior to the deployment of infrastructure code. For each tool supported by Checkov, there exists a set of built-in policies against which the code is evaluated, these policies are defined as or considered best practices. Moreover, custom policies can be created using Python or YAML. The utilization of Checkov empowers the main features of PaC, since it enhances the security, reliability, and compliance of infrastructure deployments by identifying misconfigurations and vulnerabilities early in the development lifecycle, such as overly permissive security group rules, weak encryption settings, or public exposure of sensitive information. It serves as a valuable tool for organizations adopting IaC to manage their infrastructure resources. Integration with CI/CD pipelines allows, instead, for continuous and automated security checks.

One of Checkov's key features is its multi-framework support. It covers popular IaC frameworks including Terraform, CloudFormation, Kubernetes, and Serverless Frameworks [72], supporting their syntax and structure to offer specific checks tailored to their requirements [73].

Checkov 2.0 [74] introduces a new YAML format for checks, utilizing an embedded graph database. This graph database enables the creation of checks that query the connections and adjacencies between objects, rather than focusing solely on individual objects. For instance, determining whether an AWS EC2 instance is exposed to the Internet, cannot be achieved with a standard Checkov check, as it depends on various interconnected objects, for instance [74]:

- The instance might reside in a VPC with a NAT gateway forwarding a port from that gateway.
- It could be linked to an elastic load balancer.
- It might have public Internet connectivity via BGP and a routing table that exposes an IP address directly to the Internet.
- Security groups and network policies also play a role in defining the instance's public accessibility.

A graph-based analysis also offers several notable advantages such as enabling more efficient rendering of variables for Terraform and facilitating module inheritance, allowing for more complex queries about IaC templates by considering the environmental context rather than just individual resource attributes [74].

Listing 3 shows how the security policy "HTTP port (80) must not be exposed" is implemented in Checkov for a Terraform and AWS implementation. Similar to Kics, this file is tailored to a specific IaC tool and infrastructure provider (Terraform and AWS in this case). Consequently, despite the valuable features offered by Checkov, particularly the graph-based policy, the main issue is that these checks are not generic but instead specific to each platform-provider scenario supported by the PaC tool. This leads to redundancy, as well as to difficulty in understanding and adopting of such tools in a standardised way.

Listing 3 This Python code examines AWS Security Group configurations to ensure that HTTP port (80) is not open to the internet without restriction. Adapted from [75]

```
from checkov.terraform.checks.resource.aws.AbsSecurityGroupUnrestrictedIngress import
↪  AbsSecurityGroupUnrestrictedIngress

class SecurityGroupUnrestrictedIngress80(AbsSecurityGroupUnrestrictedIngress):
    def __init__(self):
        super().__init__(check_id="CKV_AWS_260", port=80)

check = SecurityGroupUnrestrictedIngress80()
```

Listing 4 Rego file. Adapted from [77]

```
not startsWith (image, "myregistry.lan/")
msg := sprintf("image '%v' comes from untrusted registry", [image])

```

3.2.2 Open Policy Agent and Rego

Open Policy Agent (OPA) is a general-purpose open source Policy Engine (PE) developed by Styra [76], designed to enforce policies across microservices, Kubernetes, CI/CD pipelines, API gateways, and more. It offers extensive tooling and over 100 integrations to support policy implementation and enforcement within the cloud-native ecosystem [15]. Policy decision-making in OPA is articulated using Rego [49], a high-level declarative language to specify PaC. The latter is tailored for defining queries over intricate hierarchical data structures. Rego enables the codification of policies as assertions on data stored in OPA, facilitating the identification of data instances that deviate from the expected system state [62].

Rego is a language specifically designed for policy writing. A major difference between Rego and more general programming languages is that the former is generally written to authorize everything unless a specific set of conditions happens. We can see an example of this in Listing 6. Another difference is that there is no explicit "if-then-else" control statements. When a code line of Rego generates a decision, the code is interpreted as "if this line is false, then stop execution". For instance, the code depicted in Listing 4 says "if the image starts with *myregistry.lan/*, then stop execution of the policy and pass this check, otherwise generate an error message" [77].

As shown in Figure 3.1, when a software service requires policy decisions, it provides structured data (e.g., JSON) as input to the OPA engine. The engine evaluates the supplied data against defined policies and data, subsequently generating a policy

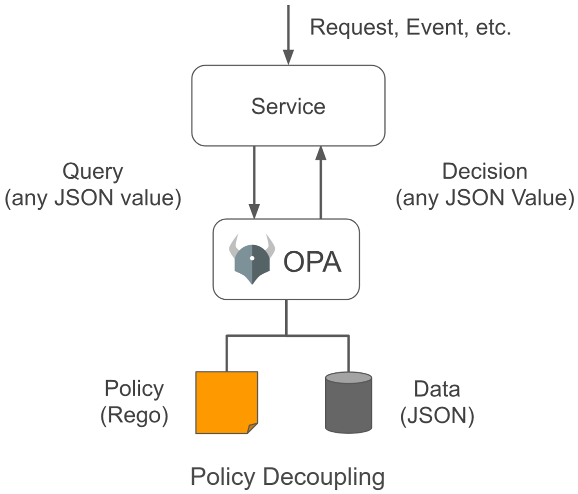

Figure 3.1: Open Policy Agent (OPA) architecture [62]

decision based on the query results. These decision are not confined to simple “yes/no” or “allow/deny” responses due to the query-based nature of Rego.

OPA serves as a foundational tool for implementing a Policy as Code approach within IT systems. Its existence obviates the need for organizations to develop custom policy management solutions from scratch. The flexibility of OPA stems from its domain-agnostic[2] Policy Engine (PE) and language, making it applicable across various contexts. Hence, it is possible to describe almost any kind of invariant in the policies. For example [62]:

- Which users can access which resources.
- Which subnets egress traffic is allowed to.
- Which clusters a workload must be deployed to.
- Which registries binaries can be downloaded from.
- Which OS capabilities a container can execute with.
- Which times of day the system can be accessed at.

Furthermore, by decoupling policy decision-making from policy enforcement, OPA allows software to query the engine with structured data inputs to obtain policy decisions. This capability underscores OPA’s versatility and utility in policy management and enforcement.

---

[2] “Domain-agnostic” describes a system, language, or framework that is not limited to any specific domain or application area. In the context of OPA this means they are versatile and can be effectively applied across various use cases without being tied to a particular domain or industry. In other words, OPA and Rego can define policies and make decisions universally, regardless of the context or sector.

Listing 5 JSON file representing a simple network infrastructure, where 5 servers are connected to some of the 4 networks, through one or more ports. Adapted from [62]

```json
{
    "servers": [
        {"id": "app", "protocols": ["https", "ssh"], "ports": ["p1", "p2", "p3"]},
        {"id": "db", "protocols": ["mysql"], "ports": ["p3"]},
        {"id": "cache", "protocols": ["memcache"], "ports": ["p3"]},
        {"id": "ci", "protocols": ["http"], "ports": ["p1", "p2"]},
        {"id": "busybox", "protocols": ["telnet"], "ports": ["p1"]}
    ],
    "networks": [
        {"id": "net1", "public": false},
        {"id": "net2", "public": false},
        {"id": "net3", "public": true},
        {"id": "net4", "public": true}
    ],
    "ports": [
        {"id": "p1", "network": "net1"},
        {"id": "p2", "network": "net3"},
        {"id": "p3", "network": "net2"}
    ]
}
```

For example, Listing 5 shows the JSON file representing a simple network infrastructure, whereas Listing 6 displays the Rego file which checks the following two policies:

1. Servers reachable from the Internet must not expose the insecure HTTP protocol.
2. Servers are not allowed to expose the “telnet” protocol.

As a result of OPA, we correctly get that there are two servers violating the above mentioned policy, as shown in the output provided in Listing 7.

OPA also has certain drawbacks such as requiring users to learn Rego: the main point to mention is that Rego is a policy evaluation language, not a generic programming language. This can be difficult for developers who are used to languages such as Golang [46], Java [79] or JavaScript [45], which support complex logic such as iterators and loops. Instead, Rego is designed to evaluate policy and is streamlined as such [77]. Moreover, the lack of libraries supporting rules for common compliance and policy standards is a consideration, which is currently the most significant limitation. Moreover, OPA requires the code being evaluated to be in JSON, which can be restrictive in some cases [62].

Despite some limitations, OPA demonstrates the most promising features as a domain-

Listing 6 Rego file checking the policies defined above. Adapted from [62]

```
package example

import rego.v1

allow if {
    count(violation) == 0
}

violation contains server.id if {
    some server in public_servers
    "http" in server.protocols
}

violation contains server.id if {
    some server in input.servers
    "telnet" in server.protocols
}

public_servers contains server if {
    some server in input.servers

    some port in server.ports
    some input_port in input.ports
    port == input_port.id

    some input_network in input.networks
    input_port.network == input_network.id
    input_network.public
}

```

agnostic PaC tool. As illustrated in Listing 6, a Rego policy rule may refer to the infrastructure using generic terms such as “server”, “port” and “network”, without the need to tailor the code to specific use cases, unlike Kics and Checkov. This high-level coding approach potentially allows for the reuse of the same code to check the same policy across almost every kind of scenario.

3.2.3 Kics

Kics, developed and maintained by Checkmarx [80], is a fully open-source PaC tool written in Golang using Open Policy Agent. It scans and finds misconfigurations and potential vulnerabilities in IaC configuration files, such as for CloudFormation,

Listing 7 Results of policy checking from the Rego file depicted in Listing 6 against the infrastructure illustrated in Listing 5. Adapted from [78]

```
{
    "public_servers": [
        {
            "id": "app",
            "ports": [
                "p1",
                "p2",
                "p3"
            ],
            "protocols": [
                "https",
                "ssh"
            ]
        },
        {
            "id": "ci",
            "ports": [
                "p1",
                "p2"
            ],
            "protocols": [
                "http"
            ]
        }
    ],
    "violation": [
        "busybox",
        "ci"
    ]
}
```

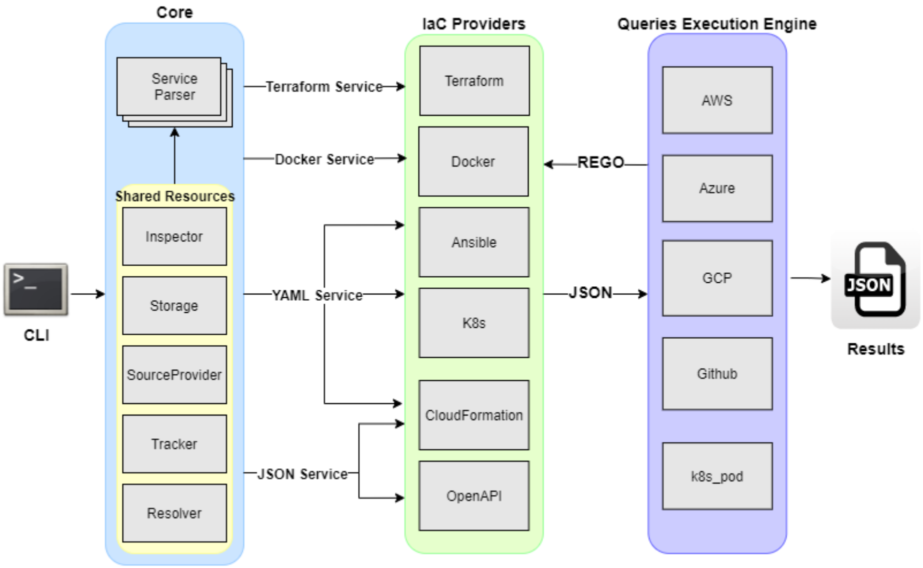

Figure 3.2: Kics architecture [81]

Ansible, Kubernetes, Terraform, Docker, Helm. To date, around 1000 ready-to-use queries have been created, covering a wide range of vulnerability checks for AWS, GCP, Azure cloud providers. Among the others, Kics comes with different queries categories such as access control, best practices, encryption, insecure configurations, networking and firewall, resource management and secret management. Moreover, Kics features a pluggable architecture with an extensible pipeline for parsing IaC languages and queries, facilitating easy integration [81].

As shown in Figure 3.2 , Kics's architecture consists of several components. In particular, the typical workflow in Kics involves several steps. First, Kics parses IaC files written in various formats such as Terraform, Dockerfile [82], or Ansible. The parser extracts relevant information, including resource definitions, configurations, and dependencies. Next, Kics uses a query engine to execute predefined queries written in Rego against the parsed IaC files. The query engine evaluates each query and generates results based on matches or violations. Moreover, Kics also includes metadata about vulnerabilities or compliance checks, such as severity levels, descriptions, and remediation steps.

Following the analysis, Kics generates reports summarizing the findings. These reports typically include details about security vulnerabilities, compliance violations, and best practice recommendations. They can be presented in various formats such as JSON, HTML, or plaintext, making them accessible and easy to integrate with other tools and workflows. Additionally, Kics can be integrated into CI/CD pipelines

Listing 8 This Terraform code defines two AWS security group resources. Port HTTP (80) is exposed on both security groups. Adapted from [83]

```
resource "aws_security_group" "positive1" {
  name        = "http_positive_tcp_1"
  description = "Gets the HTTP port open with the tcp protocol"

  ingress {
    description = "HTTP port open"
    from_port   = 78
    to_port     = 91
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}

resource "aws_security_group" "positive2" {
  name        = "http_positive_tcp_2"
  description = "Gets the HTTP port open with the tcp protocol"

  ingress {
    description = "HTTP port open"
    from_port   = 60
    to_port     = 85
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.2/0"]
  }

  ingress {
    description = "HTTP port open"
    from_port   = 65
    to_port     = 81
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

to automate security and compliance checks. This integration allows Kics to analyze IaC files as part of the software development lifecycle, providing early feedback to developers and ensuring that infrastructure changes meet security and compliance requirements. Lastly, Kics is designed to be extensible, allowing users to define custom queries and rules to address specific security and compliance needs [63].

As previously mentioned, Kics executes predefined Rego queries (from OPA), following a straightforward anatomy. Each query is composed of a *policy* and a *result* skeleton. The *policy* builds the security patterns that are used to test the infrastructure code

Listing 9 The policy, written in Rego, is designed to check the configuration of AWS security groups to ensure they do not have the HTTP port (80) open to the internet. Adapted from [84]

```
package Cx

import data.generic.terraform as tf_lib

CxPolicy[result] {
    resource := input.document[i].resource.aws_security_group[name]

    tf_lib.portOpenToInternet(resource.ingress, 80)

    result := {
        "documentId": input.document[i].id,
        "resourceType": "aws_security_group",
        "resourceName": tf_lib.get_resource_name(resource, name),
        "searchKey": sprintf("aws_security_group[%s]", [name]),
        "issueType": "IncorrectValue",
        "keyExpectedValue": "aws_security_group.ingress shouldn't open the HTTP port
        ↪  (80)",
        "keyActualValue": "aws_security_group.ingress opens the HTTP port (80)",
    }
}

```

and which the query is looking for. The *result* defines the specific vulnerability data to be presented to the user for the given infrastructure code.

To illustrate the principle followed by Kics when using Rego, Listing 8 shows an example of Terraform code where an infrastructure is created through AWS and the HTTP port (80) is exposed on purpose. To verify this policy rule, Kics applies the same approach shown in Listing 9, which defines a rule for checking whether the HTTP port 80 is open or not. In this example, the compliance check would fail.

Each query has also a *metadata.json* companion file with all the relevant information about the vulnerability, including the severity, category and its description. For example, the JSON code showed in Listing 10 depicts the metadata corresponding to the query showed in Listing 9.

Kics queries are organised per IaC technology or tool (e.g., Terraform, K8s or Dockerfile) and grouped under cloud provider (e.g., AWS, GCP or Azure) when applicable. Per each query created, it is mandatory the creation of a metadata file and test cases with, at least, one negative and positive case and a JSON file with data about the expected results [86].

Listing 10 This JSON object represents metadata about the security policy shown in Listing 9. Adapted from [85]

```json
{
  "id": "ffac8a12-322e-42c1-b9b9-81ff85c39ef7",
  "queryName": "HTTP Port Open To Internet",
  "severity": "MEDIUM",
  "category": "Networking and Firewall",
  "descriptionText": "The HTTP port is open to the internet in a Security Group",
  "descriptionUrl": "https://registry.terraform.io/providers/hashicorp/aws/latest/do⌋
  ↪  cs/resources/security_group",
  "platform": "Terraform",
  "descriptionID": "a829609b",
  "cloudProvider": "aws",
  "cwe": "",
  "oldSeverity": "HIGH"
}
```

Kics is, therefore, a valuable PaC tool for compliance checking. The main problem with the above examples is Kics's strict dependency on the IaC tool and the infrastructure provider. While Listing 9 performs its check using a domain-agnostic language (i.e. Rego), it ultimately relies on specific functions from the Terraform library and specific elements from AWS. Listing 10 further demonstrates this strict dependency.

The primary issue with an approach dependent on a specific IaC tool and infrastructure provider, is that in order to check a specific policy (e.g., port HTTP (80) must not be exposed), we would need as many *query.rego* files as there are IaC tools and providers. These files, though similar in checking the same policy, differ only in their use of specific libraries tailored to their dependencies. This results in significant redundancy, and if a new tool or provider were to be introduced, the same policies would need to be rewritten from scratch. The limitation introduced by this architecture serves as the foundational motivation for the domain-agnostic PaC tool, APaC, proposed in chapter 5.

3.3 Summary and Open Issues

From an analysis of the State of the Art (SoA), it is evident that these tools do not utilize policy engines to fully empower the potential of PaC. Moreover, they often lack a proper abstraction level to be applicable across diverse scenarios.

In addition to Kics and Checkov, we also include a comparison of two other minor tools, Regula [87] and Trivy [58]. These are not described in greater detail due to

Parameters	**Kics** [63] [91]	**Checkov** [73] [92]	**Trivy** [93] [94]	**Regula** [95] [96]
Supported IaC solutions	Terraform, AWS CloudFormation, Ansible, Docker.	Terraform, AWS CloudFormation, Docker.	Terraform, Docker, AWS CloudFormation.	Terraform, Docker, AWS CloudFormation.
Pre-built policies	Over 2400 queries are available.	More than 1000 predefined policies.	Around 1400 built-in policies.	Almost 300 rules.
Customizability	There are fully customizable and adjustable heuristic rules, called queries.	Custom policies can be defined.	Custom policies can be defined.	Custom policies can be defined.
Policy languages	OPA (Rego).	Python and YAML.	Go and OPA (Rego).	OPA (Rego).
Tool languages	OPA, HCL, Go.	Python, HCL.	Go.	OPA, Go, HCL.
Integration with CI/CD pipelines	GitLab CI, Jenkins.	Jenkins, GitLab CI.	GitHub Actions.	GitHub Actions.
Supported Cloud Providers	AWS, Azure, GCP, Kubernetes.	AWS, Azure, GCP, OpenStack.	AWS, Azure, GCP.	AWS, Azure, GCP.
Community and Support	Over 7800 commits by 119 contributors; 15130 lines of code.	Over 16000 commits by 358 contributors; 8084 lines of code.	Over 2600 commits by 383 contributors; 2898 lines of code.	Over 300 commits by 30 contributors; 1371 lines of code.

Table 3.1: Policy as Code tools comparison

their limited adoption and the lack of comprehensive documentation.

Each of these tools is an open source, static analysis tool. To compare them the following parameters are taken into account:

- Supported IaC languages: it representes the most used amongst the IaC languages the tool supports.
- Pre-built policies: evaluate the availability of pre-built policies or rule sets covering common security best practices and compliance standards.
- Customizability: assess the ease and flexibility of creating custom policies tailored to the organization's specific requirements and compliance needs.
- Policy languages: define the language used to define policies.
- Tool languages: define the most used languages by the tool source code.
- Integration with CI/CD pipelines: Check which amongst the most used CI/CD tools (e.g., Jenkins [88], GitLab CI [89], GitHub Actions [90]), the tool seamlessly integrates with.
- Supported Cloud Providers: the main cloud providers the tool is compatible with.
- Community and Support: Consider the size and activity of the tool's community, and responsiveness of support channels. Consider also the number of collaborators and lines of code.

Table 3.1 shows that each of these PaC tools supports the main IaC solutions as well as the most commonly used infrastructure providers. It is also worth noting that all of them allow the creation of custom policies, in addition to offering a significant number of pre-built policies, to better meet the organizations' needs. Moreover, Kics, Checkov and Trivy demonstrate considerable community contributions. These features underscore the importance and benefits of having an open-source solution, as it allows for constant improvements in the reliability and accuracy of such tools.

While Kics, Trivy and Regula use OPA to define their policies, none of them fully leverage this powerful tool to abstract the platform on which the infrastructure is implemented. Instead they heavily utilize specific functions and languages. For instance, Kics, Checkov and Regula extensively use HCL in their source code, showing a tailored approach towards a specific IaC language (i.e., Terraform). Consequently, the primary objective of chapter 5 will be to provide a domain-agnostic solution to evaluate policy compliance without relying on specific platforms or tools.

Chapter 4

Methodology

This chapter provides an overview of the research methodology implemented throughout this thesis. It contains an adaptation of the design science methodology with a description of the iterative steps of the design cycle. Additionally, the outline of the methodology that guided the development of APaC is presented.

4.1 Research Design

We begin our research by thoroughly reviewing the relevant literature in Infrastructure as Code and Policy as Code. This phase defines our scope by identifying Terraform and Ansible as popular IaC tools, and Kics and Checkov as commonly used PaC tools. It also highlights the limitations of current PaC tools and shows how OPA may create a more versatile PaC solution. At this stage, we consider the context and issues of our thesis, leading to the research questions presented in chapter 1. To establish our research design and answer these questions, we set the following tasks:

1. Experiment with basic configurations using Terraform and Ansible to gain familiarity with these tools and the IaC coding approach.
2. Investigate the creation of PaC tools to ensure the compliance of network infrastructure within an IaC environment, and identify potential improvements.
3. Develop a domain-agnostic-based PaC architecture, referred to as APaC, which abstracts the IaC used and the cloud provider where the IaC configuration is applied.
4. Validate the correct behavior of the newly created prototype against a simple network infrastructure, ensuring the verification of several policies.

These tasks outline the steps to be taken in our adapted version of Wieringa's design cycle [97]. The problem-solving process usually involves multiple iterations of the design cycle steps, as shown in Figure 4.1, which includes:

- **Problem investigation**: The starting phase where we evaluate the problem within the IaC and PaC context[1] and its potential effects. We conduct a literature review to identify challenges and assess existing tools for a domain-agnostic PaC solution.
- **Treatment[2] design**. In this stage, we study the domain, requirements, and available treatments and design the artefacts thoroughly. However, in our design cycle, the main action in this step is designing the artefact[3]. In the first iteration of the design cycle, we provision and deploy a simple infrastructure from the IaC tools, such as Terraform and Ansible, to make ourselves familiar with the specific keywords of such tools. Towards the end of the design process, we add more artefacts, in particular we design an architecture and develop a Parser to convert the infrastructure-code-specific configuration files into generic ones representing the infrastructure on a higher level. Finally, we define infrastructure-independent policy files, written in Rego, to check the previously created infrastructure for policy compliance.
- **Treatment Validation** is a phase in which the investigation of the interaction between the artefact and the problem context takes place. During our design cycle, we assess how the artefact behaves in different use cases, meaning that we validate the parsing of IaC files into infrastructure-independent files, and ensuring OPA detects policy violations through Rego files.

Treatment implementation and evaluation are not included in our design cycle as we do not study how the artefacts interact in a real-world environment. As suggested by Wieringa [97], a potential way of executing these two tasks might include artefacts' interaction with the stakeholders (human evaluators) through surveys.

In conclusion, as illustrated in Figure 4.1, we repeatedly go through the three steps of the design cycle to answer our research questions. We use an agile development process, beginning with a small-scale prototype and gradually testing and adding more features. Additional details on addressing the design tasks are provided in the next section.

4.2 Domain-agnostic Policy as Code Development

The first design task, as detailed in the previous section, involves provisioning and deploying a network infrastructure using both Terraform and Ansible, along with

---

[1] The context can be, for instance, people, norms, methods. In general, any element interacting with the artefact [97].

[2] According to Wieringa [97], the *treatment* refers to the solution that can potentially solve the research problem.

[3] An artefact can be anything designed and created by humans, both as a real, physical object or an abstract concept. For instance, software, hardware, methods, techniques [98].

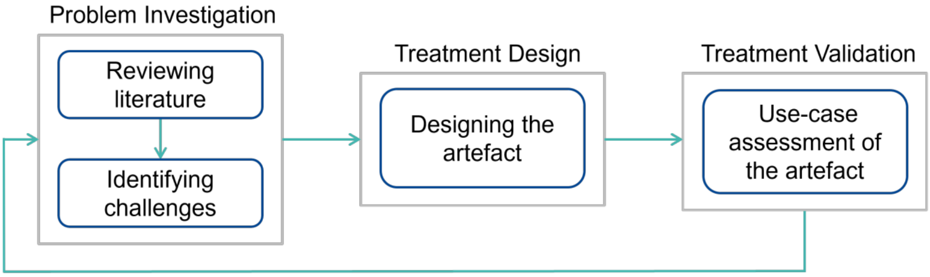

Figure 4.1: The research design cycle, adapted from [98]

defining the Parser and the policy rules. As illustrated in Figure 4.2, the coding process for the development of APaC is performed using an IDE on a remote host. However, the design implementation differs slightly for each artefact:

- **Infrastructure deployment and provisioning**: The code is first submitted to a GitHub repository for safe version control. The updated code is then retrieved from a Linux host, where all necessary dependencies are pre-installed. Here, the infrastructure is provisioned and deployed by executing the Terraform or Ansible code. The actual infrastructure can be observed and validated through the providers used in this thesis, Docker or OpenStack. If any issue arises, the code is updated again from the IDE interface. While this step may not be fully recognised as a typical research artefact, it was a necessary step for understanding the relevance of each infrastructure-code-specific keyword, which is crucial for defining the Parser in the next step, and for serving as a testbed during its validation.

- **Architecture**: the architecture, illustrated in the next chapter in Figure 5.1, is designed, discussed and tested “by hand” from the Linux host.

- **APaC, Parser definition**: the Parser is developed and validated from the IDE taking the `main.tf` file from Terraform or the `playbook.yml` from Ansible and converting it into a generic JSON file. The code is submitted to GitHub only upon reaching significant milestones.

- **Policy rules in Rego and OPA validation**: The policy rules are written in Rego and then submitted to GitHub. The newly created policy is retrieved from a Linux host, and the execution of OPA is evaluated against the predefined generic JSON file representing the infrastructure. This step is necessary because the Linux host provides the required dependencies to run the OPA engine. If any issues occur, the code is updated again from the IDE interface.

In conclusion, this approach provides the necessary tools to understand and apply the primary principles of IaC and PaC. Continuous Integration is facilitated through

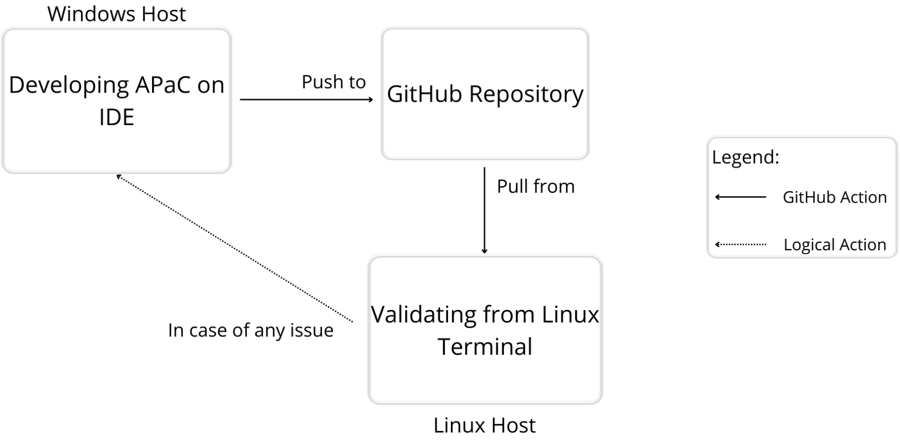

Figure 4.2: Domain-agnostic PaC development

the GitHub repository, ensuring safe and continuous code versioning and validation for the APaC development. This is needed to demonstrate the feasibility of a domain-agnostic PaC tool.

Chapter 5

Domain-agnostic Policy as Code

Each Policy as Code tool we have examined functions properly and possesses all the necessary features to ensure the automation of policy compliance and management. However, the primary issue we have encountered is that tools like Kics or Checkov tailor the policy-checking code to specific use cases. For example, Kics may have a policy-checking file specifically designed for a Terraform infrastructure implementation deployed on a particular infrastructure provider, such as OpenStack. This approach leads to significant redundancy in the code and complicates the understanding of the PaC field, since the Compliance Team needs to know every required keyword or function of the specific use case. This redundancy leads to numerous files tailored to each supported use case, differing only in the specific keywords used, while the core logic remains similar since they check the same policy.

Primary goal of this chapter is to propose an abstract method for checking policies using generic keywords regardless of the IaC tool or infrastructure provider being analysed. The infrastructure will be examined using generic terms such as “server”, “port” or “network” instead of specific ones like “aws_security_group” or functions from the IaC tool library. This approach potentially allows the same file to check the same policy across various scenarios. The benefits of this method include reduced lines of code, increased clarity and awareness of the policies the tool can check, decreased redundancy, and improved ease of writing and understanding policy files from the Compliance Team.

5.1 APaC’s architecture

Figure 5.1 depicts the proposed architecture for the development of APaC. Firstly, a taxonomy needs to be established, defining infrastructure objects (e.g., servers, networks, ports) with generic keywords. This ensures the taxonomy’s applicability across various provisioning and deployment platforms.

Secondly, a common infrastructure is defined and implemented on both Terraform and Ansible, and deployed on Docker and OpenStack. This results in four distinct implementations of the same infrastructure. It should be noted that while APaC

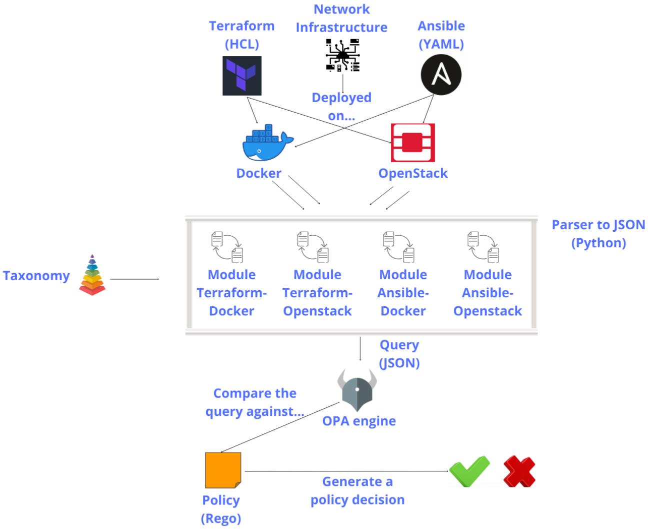

Figure 5.1: Architecture Agnostic Policy as Code (APaC)

focuses on specific IaC tools for the sake of a Proof of Concept, the proposed approach can easily be extended to support other tools.

The infrastructure code is taken as input by a Parser, which converts the IaC-tool-and-infrastructure-provider-specific code, written in YAML or HCL, into a generic format, written in JSON, based on the previously defined taxonomy. This parsing is applied to each of the four implementations. The Parser's output will be mostly the same JSON file for each of the four implementations, thereby proving the tool's agnosticism, as the output remains consistent regardless of the input infrastructure code.

Finally, the OPA engine receives the JSON file as input and compares it against the Rego file to verify policy compliance of the infrastructure code. Consequently, the OPA engine generates a policy decision indicating whether the infrastructure adheres to the defined policy rules.

It is also important to note that the Rego file is defined according to the proposed taxonomy (c.f., subsection 5.3.1), using the same generic keywords to refer to the infrastructure.

5.2 The choice of the tools

The presented architecture serves as a Proof of Concept to demonstrate the feasibility of an abstract PaC tool, referred to as APaC, incorporating all previously discussed features. Consequently, two widely-used IaC tools, Terraform and Ansible, and two infrastructure providers, Docker and OpenStack, have been selected for infrastructure provisioning and deployment. These tools were chosen for their widespread usage, well-maintained documentation[1], and ease of implementation within this architecture.

Python is employed for implementing the Parser, the core component of APaC, as it enables the conversion of infrastructure-specific code written in HCL or YAML into a generic JSON file representing the infrastructure. The choice of Python is due to its clean and readable syntax, facilitating easier code writing and comprehension, along with its extensive ecosystem of libraries and frameworks that streamline Parser development.

Lastly, the final element of the architecture is provided by OPA, chosen for its unique ability to natively perform high-level evaluations of policy compliance for infrastructure code.

Familiarity with these tools was nonexistent, therefore, significant effort was put into testing and documenting them. This was not always straightforward, which highlights the importance of having accessible documentation. It further attests the value of consistency in important matters such as PaC. A more detailed explanation of the project structure is provided in section A.1.

5.3 Implementation

This section is focused on the implementation of the architecture illustrated in Figure 5.1. The working flow is explained and a generic understanding is provided.

5.3.1 Definition of a taxonomy

Figure 5.2 illustrates the taxonomy proposed for this project. Generic keywords representing the infrastructure, regardless of where this is provisioned or deployed, are introduced and used as a base for referring to the infrastructure in an agnostic way when checking for policy compliance.

For the sake of simplicity and for an easier understanding, the taxonomy does not aim to provide a full coverage of every possible network infrastructure scenario; the aim is, instead, to prove that the creation of common keywords is possible and that this approach helps in providing a better understanding of the policy compliance field.

---

[1] This documentation has played a significant role in providing the necessary functions and methods for the implementation of APaC. Specifically, it has guided the deployment on OpenStack via Ansible [99], on Openstack via Terraform [100], on Docker via Ansible [101] and on Docker via Terraform [102].

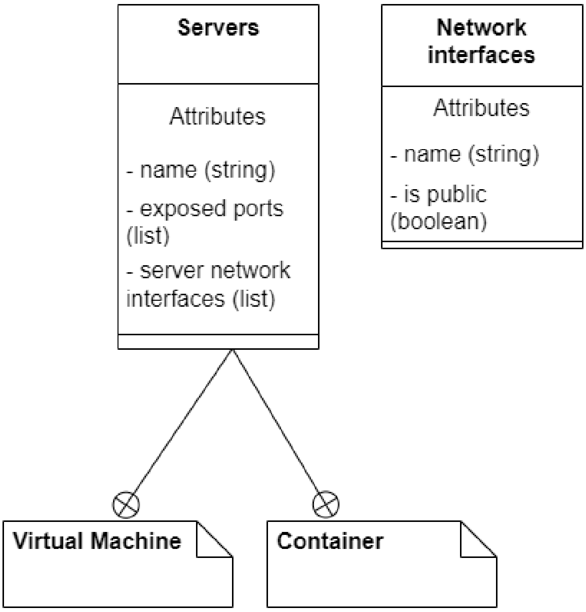

Figure 5.2: Taxonomy

Two main concepts are defined in our taxonomy: *servers* and *network interfaces*. In the specific context of APaC, a *server*[2] is a generic term used to represent a VM (for OpenStack) or a container (for Docker). In particular, the field *name* represents the name of the server itself. The *exposed ports* field, instead, represents the list of each port exposed from the server defined above. This is crucial for detecting vulnerable ports which can cause security issues. Finally, the *server network interfaces* field illustrates the list of each network interfaces belonging to the server.

The *network interfaces* keyword represents, for each network interface of the infrastructure, whether such interface is public or not (accessible from outside the network where the server belongs). A single network interface of the infrastructure represents an IP address from which the server can communicate to other servers.

5.3.2 Architecture implementation

Based on the taxonomy, a common infrastructure is provisioned and deployed using both IaC solutions, Terraform and Ansible, and infrastructure providers, OpenStack and Docker. Hence, we create the same infrastructure in four different ways. This is achieved by coding the Ansible infrastructure in YAML and the Terraform one in HCL.

---

[2] On a higher level, this may also refer to a software that provides services to another software or client, such as web servers or database servers.

The next step is to create the Parser coded in Python. This Parser defines the functions and methods needed to convert the keywords specific to the IaC solution into the generic ones defined by the Taxonomy in Figure 5.2. The common functions are only defined once; instead, there is a parsing function for each IaC-infrastructure-provider-combination. The Parser is executed via the CLI by specifying with which IaC and provider the Python code itself defines the infrastructure. The input file where the infrastructure code is located must also be specified, as well as the output file where the resulting JSON file will be saved. For instance, to convert the Ansible file *playbook.yml*, one must specify the path of this file, the IaC solution used (Ansible in this case), the infrastructure provider (either OpenStack or Docker, in case of APaC), and the output file path where the JSON file will be generated. The result of the Parser execution is, therefore, a JSON file representing the infrastructure regardless of the IaC and provider used. This adaptability allows the code to avoid hard coding, thereby easily allowing any infrastructure code file to be checked by specifying its path.

Next, the policy rules are defined in Rego to be evaluated by the OPA. This file, along with the JSON file generated by the Parser, refers to the infrastructure using the generic keywords defined by the taxonomy in Figure 5.2. Consequently, the JSON file generated by the Parser is compared against the Rego file by the OPA engine. This is done by specifying the paths of both the JSON and Rego files.

Finally, the output of OPA defines the policy decision, indicating whether the infrastructure complies with the policy rules defined by the Rego file. The output shows the policy decision and, if the infrastructure is not compliant, it also identifies which servers caused the violation.

5.4 Validation and Evaluation

This section provides the evaluation details for the implementation of APaC. The Appendix A, instead, presents the code details.

The infrastructure proposed for this project is illustrated in Figure 5.3 and represents a simple configuration where three servers belonging to the same network are created. Among these three instances, *server1* exposes port 80 and it is accessible from outside its own network; *server2* exposes port 22 and it is not accessible from outside; lastly, *server3* exposes port 443 and it is accessible from outside. As illustrated in Figure 5.1, such infrastructure is deployed in four different ways: on Docker from Terraform, on Docker from Ansible, on OpenStack from Terraform and on OpenStack from Ansible.

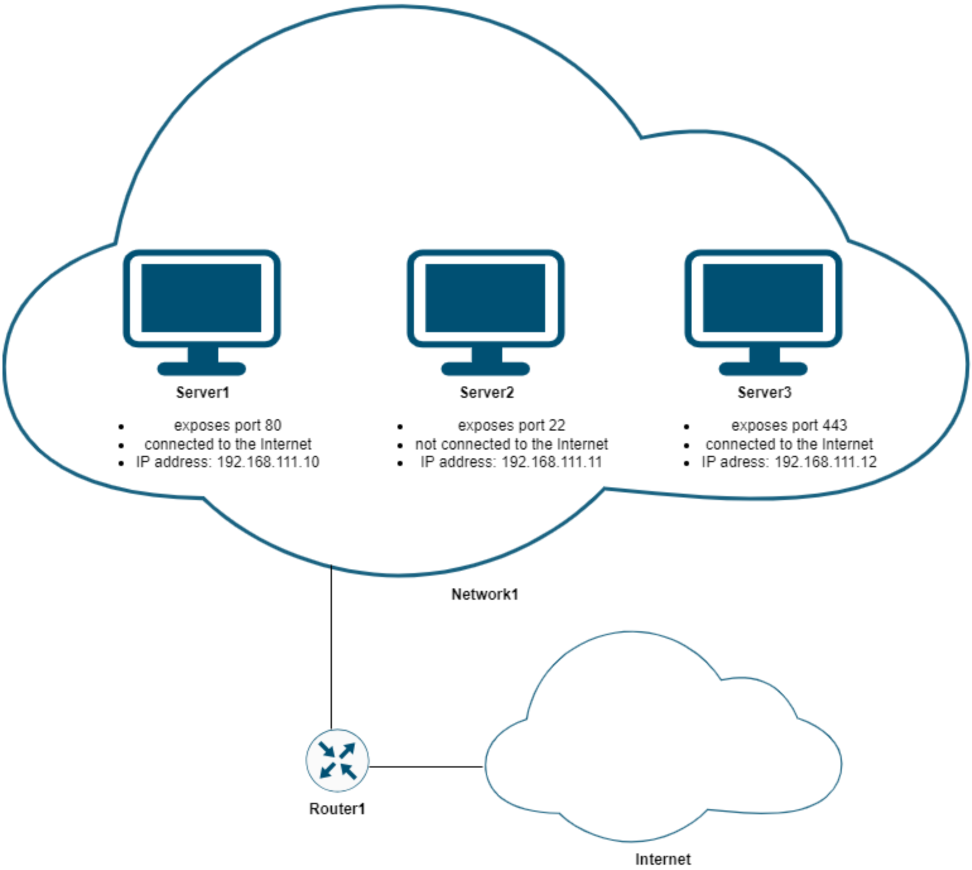

Figure 5.3: Infrastructure proposed for the PoC of APaC

5.4.1 Infrastructure provisioning and deployment

The four implementations using Terraform, Ansible, Docker, and OpenStack essentially generate the same infrastructure. However, each implementation differs due to the specific IaC tool and deployment platform employed. Notably, comparing the infrastructure code in Ansible with that in Terraform reveals distinct approaches and keywords utilized by each tool, compounded by the fact that they employ different programming languages. Additionally, differences are evident when comparing the infrastructure code deployed on Docker versus OpenStack, arising from the distinct methodologies required by each platform to deploy the same infrastructure.

All these implementations occur policy misconfigurations (exposure of port 22 and 80), therefore a unified policy checking tool is needed to highlight such misconfigurations and allow the Compliance Team to identify and correct them.

5.4.2 Parser definition

The Parser serves as the core element of the APaC architecture, enabling the conversion of specific infrastructure implementation files into a standardized JSON file. This JSON file can subsequently be compared and evaluated by the Policy Engine of OPA. The JSON file represents the whole infrastructure using generic keywords, therefore regardless of the infrastructure code or the deployment platforms used, this file will always look mostly the same. Moreover, the next elements of this architecture (OPA and Rego) will perform policy evaluations independently from the infrastructure's provisioning and deployment sources.

It is important to note that this Parser does not need to execute Ansible or Terraform code to generate the standardized JSON file. Instead, it directly transforms the infrastructure code file by converting each specific keyword into the generic ones defined in the taxonomy illustrated in Figure 5.2.

Listing 11 represents one of the standardized JSON files generated by the Parser, representing the infrastructure illustrated in Figure 5.3.

One out of the main powerful features of APaC relies on the Parser, since it easily allows developers to extend this tool to support other IaC solutions and providers. This is more clear when looking at the modular implementation of the Parser itself in the architecture shown in Figure 5.1; a new module, allowing the translation from a new IaC solution or provider, can easily be placed without any other modification needed. The already implemented policy rules would seamlessly work on these new solutions.

5.4.3 Definition of policy rules and compliance checking

The final step before executing the OPA engine is to define the policy with which the infrastructure must comply. In the context of this thesis, we define a security policy comprising two rules written in Rego:

1. Servers reachable from the Internet must not expose the insecure HTTP protocol (port 80).
2. Servers are not allowed to expose the SSH port (port 22).

It is important to notice that this policy rules are just an example. Any kind of policy rules may be defined in Rego and applied to any infrastructure, due to the domain-agnostic nature of Rego itself.

This Rego file, illustrated in Listing 12, along with the JSON file generated by the Parser, refers to the infrastructure using the generic keywords defined in Taxonomy illustrated in Figure 5.2. Thus, this same file can be applied to each of the four

Listing 11 This JSON file represents the outcome from the Parser execution. The infrastructure is depicted using the generic keywords defined in the Taxonomy in Figure 5.2.

```
{
    "servers": [
        {
            "name": "server1",
            "exposed_ports": [
                80
            ],
            "server_network_interfaces": [
                "port_server_1"
            ]
        },
        {
            "name": "server2",
            "exposed_ports": [
                22
            ],
            "server_network_interfaces": [
                "port_server_2"
            ]
        },
        {
            "name": "server3",
            "exposed_ports": [
                443
            ],
            "server_network_interfaces": [
                "port_server_3"
            ]
        }
    ],
    "network_interfaces": [
        {
            "name": "port_server_1",
            "is_public": true
        },
        {
            "name": "port_server_2",
            "is_public": false
        },
        {
            "name": "port_server_3",
            "is_public": true
        }
    ]
}
```

Listing 12 Rego file describing the policy rules previously defined. This file verifies any violations by initially checking for servers exposing port 22, followed by checking for any servers exposing port 80 while permitting communication from external networks. Each time a violation is detected, a counter is incremented. If the counter registers at least one violation, it outputs a negative response, highlighting the specific rules being violated.

```
package example

import rego.v1

default allow := false

allow if {
    count(violation) == 0
}

violation contains server.name if {
    some server
    public_servers[server]
    server.exposed_ports[_] == 80
}

violation contains server.name if {
        server := input.servers[_]
        server.exposed_ports[_] == 22
}

public_servers contains server if {
    some i, j
    server := input.servers[_]
    input.servers[i].network_interfaces[_] == input.network_interfaces[j].name
    input.network_interfaces[j].is_public
}

# METADATA
# title: Exposure of vulnerable ports 22 and 80
# description: Port 22 must not be exposed. Port 80 must not be exposed if the server
↪  is accessible from outside its own network
output := decision if {
    count(violation) > 0

    annotation := rego.metadata.rule()
    decision := {
        "title": annotation.title,
        "message": annotation.description,
        "violations": violation
    }
}
```

Listing 13 JSON file representing the policy decision from OPA

```json
{
  "result": [
    {
      "expressions": [
        {
          "value": {
            "message": "Port 22 must not be exposed. Port 80 must not be exposed if
            ↪  the server is accessible from outside its own network",
            "title": "Exposure of vulnerable ports 22 and 80",
            "violations": [
              "server1",
              "server2"
            ]
          },
          "text": "data.example.output",
          "location": {
            "row": 1,
            "col": 1
          }
        }
      ]
    }
  ]
}
```

previously provided implementations. It enforces the policy defined above using Rego syntax rules.

The OPA engine processes the JSON file representing the infrastructure and evaluates it against the Rego file. Finally, a policy decision is generated, indicating whether the infrastructure complies with the defined policy rules. This decision is encapsulated in a JSON file, which specifies, among other details, whether the infrastructure complies with the policy and, if it does not, identifies the *servers* responsible for the violation. These *servers* are accurately identified as *server1* and *server2*.

5.5 Summary

This section analyses the results obtained from APaC and compares it against the approach adopted by the modern PaC solutions, such as Kics or Checkov, to prove the importance and the potential of such architecture (Figure 5.1).

5.5.1 Results

In the infrastructure depicted in Figure 5.3, it is evident that *server1* and *server2* do not adhere to one or both security policy rules previously defined, whereas *server3* fulfills both rules. As a matter of fact Listing 13, representing from the execution of APaC, shows that the infrastructure violates the policy rules outlined in Listing 12. Moreover, it correctly identifies the *servers* that caused such violation, namely *server1* and *server2*.

We have demonstrated the feasibility of creating a Policy as Code tool that operates independently of the platform used for network infrastructure creation and management. Specifically, APaC addresses both infrastructure and policy management in a platform-agnostic manner. This abstraction of network elements is achieved through the Parser, which removes the specificity of the infrastructure code, and through OPA's effective utilization of Rego and its high-level principles.

We have reached a valuable level of abstraction, which potentially reduce the lines of code and complexity in a more advanced PaC tool. The Parser itself may be written in any language, as long as the output is a JSON file. Speaking of which, the JSON file remains the only non-abstract component of APaC, as OPA requires it as input.

This domain independence is further exemplified in the project structure used for APaC (c.f., as detailed in section A.1) where the OPA and Parser logic is independent from the infrastructure code. As a matter of fact, any infrastructure code that may be provided, among the solutions defined in the Parser (i.e., Terraform, Ansible, OpenStack and Docker), would be checked for policy rule compliance without needing to modify the Parser or the policy compliance code. Thus, only a single Rego file would be used to enforce the same policy across various scenarios.

5.5.2 APaC compared to Kics

In chapter 3 we analysed the features and the main issues of some of the most used PaC tools, such as Kics and Checkov. By having another look at one of the Rego file implemented by Kics and illustrated in Listing 9, we may notice that this file performs a policy checking rule, i.e. HTTP port 80 must not be exposed, specifically designed for an AWS deployment on a Terraform file. Kics implements the same rule for each IaC solution (Terraform, Ansible, CloudFormation, Pulumi and the other solutions supported) and for each infrastructure provider. Consequently, using Kics's approach results in four different Rego files for checking the same policy rule: one for each combination of Terraform, Ansible and Docker, OpenStack.

On the other hand, the Rego file implemented in APaC works for any infrastructure code among the ones implemented in the architecture depicted in Figure 5.1.

Comparing the two approaches shows that to implement the same policy, Kics requires as many Rego files as the number of combinations between the IaC solutions and infrastructure providers supported. For instance, if Kics wants to verify compliance across two different IaC tools and two different infrastructure providers, it must provide four different Rego files to check the same policy. Generally, the number of Rego files needed to implement the same policy rule across each supported platform is given by the product of the number of IaC solutions supported and the number of infrastructure providers supported. In contrast, the number of Rego files needed to implement the same policy rule across the infrastructure provided in APaC will always be one. In case a new solution needs to be supported by APaC, only a module in the Parser that supports such an IaC solution or provider is required. Neither the Rego file nor the structure of the JSON file generated by the Parser would need to be changed, as they are independent of the platforms where the infrastructure code is defined.

APaC offers a more efficient and less redundant way of checking the same policy across different platforms, due to the domain-agnostic features of the architecture proposed in Figure 5.1.

5.5.3 Final remarks

As previously mentioned, the policy rules implemented in APaC serve as an example to demonstrate the potential of this architecture and to validate its functionality. Any type of policy can be implemented by leveraging the domain-agnostic feature natively supported by Rego. These new policies would integrate seamlessly with the rest of the architecture, due to the independence of OPA and Rego from the infrastructure code solutions.

Additionally, we have demonstrated the necessity of a Parser for each IaC tool and cloud provider to generate a JSON file that represents the infrastructure in an abstract manner. Consequently, new tools can be seamlessly and easily supported by defining a new module in the Parser definition.

To achieve this, we combined several tools and implemented the proposed architecture. The evaluation has proven the effectiveness and power of APaC. The comparison with existing PaC tools has provided us with a deeper understanding of the potential improvements for these tools. APaC effectively addresses the main issues of tools such as Kics or Checkov and provides a solid foundation for developing a new PaC tool, incorporating all the features discussed in this chapter. This may be achieved by expanding the number of supported infrastructure solutions (not only limited to Ansible, Terraform, Docker and OpenStack), enhancing the modularity of the Parser, and implementing more policy rules to ensure comprehensive compliance and security reliability.

Chapter 6

Discussion and Conclusion

Throughout this thesis, we captured the definitions of IaC as well as of PaC and how they can be integrated in the DevOps methodology to enhance efficiency, consistency, and governance. In particular, we assessed and discussed that they allow a significant degree of automation both in the provisioning and deployment of network infrastructures, and in the enforcement of policy compliance. Starting from this knowledge we have analysed benefits and drawbacks of the current PaC solutions and how they could be improved. Finally, in chapter 5, we have proposed a PoC showing the potential of an abstract Policy as Code solution, referred to as APaC, which is fundamental for reducing code redundancy and enhancing the efficiency of such a tool.

6.1 Discussion

In this section, we reflect on the possible implications and constraints of the research presented in this thesis. Furthermore, we examine potential avenues for improving our findings.

6.1.1 The scope of the thesis

The primary objective of this thesis is to demonstrate the potential of a domain-agnostic approach to represent policy enforcement through a PaC solution, and apply policy compliance and checking to heterogeneous IaC-based environments. To achieve this goal, our scope converged towards modern PaC solutions, such as Kics and Checkov. Nevertheless, we soon realised how ‘rigid’ these solutions are, since they require specific policy implementations for each combination of IaC tool and infrastructure provider, in spite of their modular architectures. This leads to high redundancy in the code, since the same policy may have to be implemented several times with few differences in the lines of code. The same difficulty occurs when a policy needs to be updated and re-evaluated, which is an important practice.

By narrowing the scope of the current State of the Art of PaC, we found out that the tool with the most promising features in the terms of abstraction of policies definition and enforcement is Open Policy Agent, which should only require a Parser for converting tailored-IaC configuration files into a generic JSON representation of the same infrastructure. With this in mind, we implemented APaC where we defined a prototype taxonomy to represent an example infrastructure. The latter was provisioned and deployed following a DevOps mindset, using two IaC tools (Terraform and Ansible) and two infrastructure providers (Docker and OpenStack), resulting in four different setups. We defined and implemented a Parser to convert these distinct infrastructure code files into a high-level JSON file, representing the very same infrastructure. The Parser converts the infrastructure code directly, without requiring execution. This is beneficial because it allows for policy compliance checks during the planning or design phase, eliminating the need to deploy the infrastructure beforehand. Finally, the resulting JSON file has been evaluated against a policy file, written in Rego, and embedding the same abstraction level.

We demonstrated that it is possible to define a PaC solution which is not tailored to any specific infrastructure code. The main benefit out of this, is that this same policy file may be reused countless times against any infrastructure code solution, as long as the proper Parser module is defined, according to the infrastructure defined in Figure 5.1. Hence, this allows defining and updating a policy one time only, reusing it multiple times, enhancing the understanding and effective application of policy rules by the Compliance Team.

With current PaC tools, if a new IaC solution appears in the market, or if a new solution is adopted by a company, the Compliance Team and policy enforcement mechanisms will likely have to re-define, or at least re-implement, every single policy from scratch. This situation becomes even problematic if multiple IaC solutions are used simultaneously (e.g., between different Development Teams). Using a domain-agnostic solution, such as APaC, the Compliance Team would just require the Parser module from converting the infrastructure code of each specific solution into the standardised file, representing the infrastructure with abstract well-understood keywords. Existing policy rules and their implementations would be independent of the IaC platforms and remain valid. It is worth noticing that the implementation of APaC, proposed in chapter 5, only represents a Proof of Concept implementing a working solution for four tools (i.e., Terraform, Ansible, OpenStack and Docker). Nevertheless, the tool may easily be extended, requiring only the implementation of an adequate Parser module. Similarly, the used taxonomy and defined policies serve as a PoC only, but can easily be extended using the domain-agnostic features provided by Rego.

6.1.2 Limitations and future research

A dependency of our APaC is the JSON file required by OPA to evaluate the infrastructure against policy rules. This file must be in JSON format due to OPA's inherent architecture. However, JSON is a well-known and flexible format, allowing users to define their own structure, taxonomy, and policy rules implementation. Our primary dependencies are OPA and the Rego policy language. Nonetheless, the APaC implementation separates the Parser from OPA, making it possible to implement other PEs easily.

The evaluation of a server's accessibility outside its network (e.g., directly in the code, or even through forwarding or routing) is based on certain assumptions that suffice for this PoC (c.f., section A.4). A thorough assessment in a realistic environment would require a more detailed definition, and the selection of infrastructure providers may significantly impact this effort due the different approaches to infrastructure deployment and available APIs. The challenge in representing these differences into abstract principles would require a comprehensive taxonomy for systematically classifying Infrastructure as Code concepts and their equivalent from different providers, which is out of the scope of this thesis. In addition, this effort would potentially require a full ontology design to represent not only the concepts and categories, but also the relation between them.

From an implementation perspective, future steps may involve expanding the taxonomy to include all the main concepts and elements relevant to an infrastructure provider, such as network devices (routers, switches, firewalls, access points) or network security components (VPN, IDS/IPS, authentication servers). With these concepts, it would be possible to consider connections and dependencies among different infrastructure elements. An intriguing and promising approach to address this is seen Checkov's graph-based policy definition (c.f., section 3.2), which would have to be extended to Rego. Additionally, supporting the parsing of more IaC tools and infrastructure provider solutions would be essential to consider this tool in an enterprise environment. Similarly, the implementation of the most common and important policy rules in Rego, would be a desirable feature to promote a quick adoption by interested parties. Such policies would be shared and scrutinised by all users, exploiting the domain-agnostic nature of our solution.

6.2 Summary of Findings

In this section, we highlight the key contributions achieved while creating APaC. Moreover, we provide an overview of the main findings obtained by answering the research questions.

RQ1: What is the current status of Infrastructure as Code, which tools are most popular, and how are they used in practice?

To address this research question, we conducted a literature review to identify the key principles of software development where IaC is utilized. We then performed an in-depth analysis of two primary IaC tools, Ansible and Terraform, to understand their main features and functionalities. Specifically, tools like Terraform and Ansible focus on the automation and configuration aspects of the provisioning layer. Teams use these tools to write configuration files that describe the desired state of the infrastructure, which are then version-controlled, reviewed, and tested before deployment.

RQ2: What is the current status of Policy as Code, which tools are most popular, and how are they used in practice?

Upon reviewing the relevant literature, we found that PaC is an evolving practice that integrates policy enforcement and compliance into the development and operations workflow. This approach aligns with the principles of IaC and DevOps, promoting automation, consistency, and transparency. We analyzed in detail two key open-source tools, namely Kics and Checkov, to identify the main features and limitations of current implementations. However, the most notable tool was OPA, which is also used by Kics and allows users to define policies using Rego, a domain-agnostic language for policy definition.

RQ3: What tools do we need to define a domain-agnostic architecture and how would this be used in practice?

Based on an analysis of relevant research and the primary limitations of current PaC solutions, we proposed the implementation of APaC, i.e., a domain-agnostic PaC solution, demonstrating an efficient and modular approach to policy compliance. The tools chosen for assessing APaC were Terraform, Ansible, Docker, and OpenStack. A parser was developed to translate IaC-specific configuration files into generic JSON files, using Python as the programming language. For policy checking, we utilized OPA and its language Rego, due to its high level of abstraction.

The proposed architecture is modular and involves defining the policy rules against which the infrastructure code will be checked. The execution of APaC begins by parsing an IaC-specific file, translating it into a platform-independent format, based on a given taxonomy. This is then checked against the predefined policy rules for compliance. The final output indicates whether the provided infrastructure code adheres to the specified policies or not.

Bibliography

[1] M. Guerriero, M. Garriga, *et al.*, "Adoption, support, and challenges of infrastructure-as-code: Insights from industry", in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, ISSN: 2576-3148, Sep. 2019, pp. 580–589. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8919181 (last visited: Mar. 8, 2024).

[2] J. Sandobalín, E. Insfran, and S. Abrahão, "On the effectiveness of tools to support infrastructure as code: Model-driven versus code-centric", *IEEE Access*, vol. 8, pp. 17 734–17 761, 2020, Conference Name: IEEE Access. [Online]. Available: https://ieeexplore.ieee.org/document/8959180 (last visited: Mar. 8, 2024).

[3] A. Dalvi, "Cloud infrastructure self service delivery system using infrastructure as code", in *2022 International Conference on Computing, Communication, and Intelligent Systems (ICCCIS)*, Nov. 2022, pp. 1–6. [Online]. Available: https://ieeexplore.ieee.org/document/10037603 (last visited: Mar. 8, 2024).

[4] "Terraform by HashiCorp", Terraform by HashiCorp. (2024), [Online]. Available: https://www.terraform.io/ (last visited: May 6, 2024).

[5] "Homepage | ansible collaborative". (2024), [Online]. Available: https://www.ansible.com/ (last visited: May 6, 2024).

[6] "Chef software DevOps automation solutions | chef", Chef Software. (2024), [Online]. Available: https://www.chef.io/ (last visited: May 21, 2024).

[7] "Puppet infrastructure & IT automation at scale | puppet by perforce". (2024), [Online]. Available: https://www.puppet.com/ (last visited: May 21, 2024).

[8] "Packer by HashiCorp", Packer by HashiCorp. (2024), [Online]. Available: https://www.packer.io/ (last visited: May 21, 2024).

[9] "Cloudify documentation center | cloudify documentation center". (2024), [Online]. Available: https://docs.cloudify.co/ (last visited: May 21, 2024).

[10] "Cloud computing services - amazon web services (AWS)", Amazon Web Services, Inc. (2024), [Online]. Available: https://aws.amazon.com/ (last visited: May 22, 2024).

[11] "Cloud computing services", Google Cloud. (2024), [Online]. Available: https://cloud.google.com/ (last visited: May 22, 2024).

[12] "Cloud computing services | microsoft azure". (2024), [Online]. Available: https://azure.microsoft.com/en-us (last visited: May 22, 2024).

[13] "Open source cloud computing infrastructure", OpenStack. (2024), [Online]. Available: https://www.openstack.org/ (last visited: May 28, 2024).

[14] “Docker: Accelerated container application development”. (May 10, 2022), [Online]. Available: https://www.docker.com/ (last visited: May 28, 2024).

[15] Y. Matharu. “Introduction to policy as code with automation”. Publisher: Red Hat, Inc. Section: Enable Sysadmin. (Dec. 16, 2022), [Online]. Available: https://www.redhat.com/sysadmin/policy-as-code-automation (last visited: Mar. 16, 2024).

[16] “Unit 42 cloud threat report”. (2024), [Online]. Available: https://start.paloaltonetworks.com/unit-42-cloud-threat-report (last visited: Apr. 24, 2024).

[17] “Social sustainability | UN global compact”. (2024), [Online]. Available: https://unglobalcompact.org/what-is-gc/our-work/social (last visited: Jun. 27, 2024).

[18] “THE 17 GOALS | sustainable development”. (2024), [Online]. Available: https://sdgs.un.org/goals (last visited: Jun. 27, 2024).

[19] “Social sustainability | UN global compact”. (2024), [Online]. Available: https://unglobalcompact.org/what-is-gc/our-work/social (last visited: Jun. 27, 2024).

[20] S. Pandya and R. Guha Thakurta, *Introduction to Infrastructure as Code: A Brief Guide to the Future of DevOps*. Berkeley, CA: Apress, 2022. [Online]. Available: https://link.springer.com/10.1007/978-1-4842-8689-0 (last visited: Mar. 8, 2024).

[21] @HashiCorp. “HashiCorp state of cloud”, HashiCorp. (2024), [Online]. Available: https://www.hashicorp.com/state-of-the-cloud (last visited: Jun. 5, 2024).

[22] “Amazon.com. spend less. smile more.” (2024), [Online]. Available: https://www.amazon.com/ (last visited: May 21, 2024).

[23] “Netflix Norway – watch shows online, watch movies online”. (2024), [Online]. Available: https://www.netflix.com/no/ (last visited: May 21, 2024).

[24] “Google”. (2024), [Online]. Available: https://www.google.no/ (last visited: May 21, 2024).

[25] I. Kumara, M. Garriga, *et al.*, “The do’s and don’ts of infrastructure code: A systematic gray literature review”, *Information and Software Technology*, vol. 137, p. 106 593, Sep. 1, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584921000720 (last visited: Mar. 8, 2024).

[26] M. K. Bali and R. Walia, “Enhancing efficiency through infrastructure automation: An in-depth analysis of infrastructure as code (IaC) tools”, in *2023 International Conference on Computing, Communication, and Intelligent Systems (ICCCIS)*, Nov. 2023, pp. 857–863. [Online]. Available: https://ieeexplore.ieee.org/document/10425162 (last visited: Mar. 8, 2024).

[27] K. Morris, *Infrastructure as Code: Managing Servers in the Cloud*, 1st. O’Reilly Media, Inc., May 2016, 362 pp.

[28] “Swarm mode overview”, Docker Documentation. (2024), [Online]. Available: https://docs.docker.com/engine/swarm/ (last visited: May 21, 2024).

[29] “Production-grade container orchestration”. (2024), [Online]. Available: https://kubernetes.io/ (last visited: May 21, 2024).

[30] “Design for azure cloud”, Argon Systems. (2024), [Online]. Available: https://argonsys.com/ (last visited: May 21, 2024).

[31] “CNCF landscape”. (2024), [Online]. Available: https://landscape.cncf.io/ (last visited: May 22, 2024).

[32] HashiCorp. “HashiCorp | the infrastructure cloud company”, HashiCorp | The Infrastructure Cloud Company. (2024), [Online]. Available: https://www.hashicorp.com/ (last visited: May 22, 2024).

[33] “DigitalOcean | cloud infrastructure for developers”. (2024), [Online]. Available: https://www.digitalocean.com/ (last visited: May 22, 2024).

[34] “Helm”. (2024), [Online]. Available: https://helm.sh/ (last visited: May 22, 2024).

[35] “Tutorials | terraform | HashiCorp developer”, Tutorials | Terraform | HashiCorp Developer. (2024), [Online]. Available: https://developer.hashicorp.com/terraform/tutorials/aws-get-started/infrastructure-as-code (last visited: May 22, 2024).

[36] L. Berton, *Ansible for Kubernetes by Example: Automate Your Kubernetes Cluster with Ansible.* Berkeley, CA: Apress, 2023. [Online]. Available: https://link.springer.com/10.1007/978-1-4842-9285-3 (last visited: Mar. 14, 2024).

[37] “OpenSSH”. (2024), [Online]. Available: https://www.openssh.com/ (last visited: May 22, 2024).

[38] “Welcome to python.org”, Python.org. (May 8, 2024), [Online]. Available: https://www.python.org/ (last visited: May 21, 2024).

[39] “INI cheat sheet & quick reference”, QuickRef.ME. (2024), [Online]. Available: https://quickref.me/ini.html (last visited: May 29, 2024).

[40] “Infrastructure as code provisioning tool - AWS CloudFormation - AWS”, Amazon Web Services, Inc. (2024), [Online]. Available: https://aws.amazon.com/cloudformation/ (last visited: May 23, 2024).

[41] “Ruby programming language”. (2024), [Online]. Available: https://www.ruby-lang.org/en/ (last visited: May 23, 2024).

[42] “Develop, deploy & manage high-impact business apps | progress software”, Progress.com. (2024), [Online]. Available: https://www.progress.com/ (last visited: May 23, 2024).

[43] “Pulumi - infrastructure as code in any programming language”, pulumi. (2024), [Online]. Available: https://www.pulumi.com/ (last visited: May 23, 2024).

[44] “JavaScript with syntax for types.” (2024), [Online]. Available: https://www.typescriptlang.org/ (last visited: May 23, 2024).

[45] “Learn JavaScript online - courses for beginners - javascript.com”. (2024), [Online]. Available: https://www.javascript.com/ (last visited: May 23, 2024).

[46] “The go programming language”. (2024), [Online]. Available: https://go.dev/ (last visited: May 23, 2024).

[47] “.NET | Costruire. Test. Distribuisci.”, Microsoft. (2024), [Online]. Available: https://dotnet.microsoft.com/it-it/ (last visited: May 23, 2024).

[48] “Saltproject.io”. (2024), [Online]. Available: https://saltproject.io/ (last visited: May 23, 2024).

[49] "Policy language", Open Policy Agent. (2024), [Online]. Available: https://www.openpolicyagent.org/docs/latest/policy-language/ (last visited: May 21, 2024).

[50] "Home page | SOC2". (2024), [Online]. Available: https://soc2.co.uk/?language=en (last visited: May 21, 2024).

[51] "CIS", CIS. (2024), [Online]. Available: https://www.cisecurity.org (last visited: May 21, 2024).

[52] "PCI data security standard (PCI DSS)", PCI Security Standards Council. (2024), [Online]. Available: https://www.pcisecuritystandards.org/standards/pci-dss/ (last visited: May 21, 2024).

[53] 14:00-17:00. "ISO/IEC 27001:2022", ISO. (2024), [Online]. Available: https://www.iso.org/standard/27001 (last visited: May 21, 2024).

[54] J. Henriques, F. Caldeira, *et al.*, "An automated closed-loop framework to enforce security policies from anomaly detection", *Computers & Security*, vol. 123, p. 102 949, Dec. 1, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404822003418 (last visited: Jun. 13, 2024).

[55] "Philosophy", Open Policy Agent. (2024), [Online]. Available: https://www.openpolicyagent.org/docs/latest/philosophy/ (last visited: May 21, 2024).

[56] "Configuration management system software - chef infra | chef", Chef Software. (2024), [Online]. Available: https://www.chef.io/products/chef-infra (last visited: Jun. 6, 2024).

[57] "CNCF landscape". (2024), [Online]. Available: https://landscape.cncf.io/guide#introduction (last visited: Jun. 13, 2024).

[58] "Trivy home", Trivy. (2024), [Online]. Available: https://trivy.dev/ (last visited: May 22, 2024).

[59] *Quay/clair*, original-date: 2015-11-13T18:46:16Z, Jun. 13, 2024. [Online]. Available: https://github.com/quay/clair (last visited: Jun. 13, 2024).

[60] *Notaryproject/notation*, original-date: 2020-04-20T16:56:00Z, Jun. 12, 2024. [Online]. Available: https://github.com/notaryproject/notation (last visited: Jun. 13, 2024).

[61] *Falcosecurity/falco*, original-date: 2016-01-19T21:58:12Z, Jun. 13, 2024. [Online]. Available: https://github.com/falcosecurity/falco (last visited: Jun. 13, 2024).

[62] "Introduction", Open Policy Agent. (2024), [Online]. Available: https://www.openpolicyagent.org/docs/latest/ (last visited: Mar. 18, 2024).

[63] "KICS". (2024), [Online]. Available: https://docs.kics.io/latest/ (last visited: May 6, 2024).

[64] Checkov contributors, *Checkov*, Accessed 2024-05-22, 2024. [Online]. Available: https://www.checkov.io/.

[65] "Policy as code | sentinel | HashiCorp developer", Policy as Code | Sentinel | HashiCorp Developer. (2024), [Online]. Available: https://developer.hashicorp.com/sentinel/docs/concepts/policy-as-code (last visited: Mar. 18, 2024).

[66] "An overview of chef InSpec". (2024), [Online]. Available: https://docs.chef.io/inspec/ (last visited: May 23, 2024).

[67] "Policy as code", pulumi. (2024), [Online]. Available: https://www.pulumi.com/docs/using-pulumi/crossguard/ (last visited: May 23, 2024).

[68] "Kyverno". (2024), [Online]. Available: https://kyverno.io/ (last visited: Jun. 13, 2024).

[69] "Enterprise policy as code (EPAC)". (2024), [Online]. Available: https://azure.github.io/enterprise-azure-policy-as-code/ (last visited: Jun. 13, 2024).

[70] "Bridgecrew", GitHub. (2024), [Online]. Available: https://github.com/bridgecrewio (last visited: May 24, 2024).

[71] "Prisma cloud | comprehensive cloud security", Palo Alto Networks. (2024), [Online]. Available: https://www.paloaltonetworks.com/prisma/cloud (last visited: May 24, 2024).

[72] "Serverless framework: Build apps on AWS lambda". (2024), [Online]. Available: https://serverless.com/framework (last visited: May 24, 2024).

[73] "What is checkov? - checkov". (2024), [Online]. Available: https://www.checkov.io/1.Welcome/What%20is%20Checkov.html (last visited: May 24, 2024).

[74] "Announcing checkov 2.0: Deepening open source IaC security", Palo Alto Networks Blog. (Apr. 8, 2021), [Online]. Available: https://www.paloaltonetworks.com/blog/prisma-cloud/checkov-2-deepening-open-source-iac-security/ (last visited: May 24, 2024).

[75] "Checkov/checkov/terraform/checks/resource/aws/SecurityGroupUnrestrictedIngress80.py at main · bridgecrewio/checkov". (2024), [Online]. Available: https://github.com/bridgecrewio/checkov/blob/main/checkov/terraform/checks/resource/aws/SecurityGroupUnrestrictedIngress80.py (last visited: Jun. 6, 2024).

[76] "Styra", Styra. (2024), [Online]. Available: https://www.styra.com/ (last visited: May 27, 2024).

[77] Scott Surovich and Marc Boorshtein, *Kubernetes and Docker - An Enterprise Guide : Effectively Containerize Applications, Integrate Enterprise Systems, and Scale Applications in Your Enterprise*. Birmingham, UK: Packt Publishing, 2020. [Online]. Available: https://search.ebscohost.com/login.aspx?direct=true&db=nlebk&AN=2650829&site=ehost-live&scope=site (last visited: Jun. 13, 2024).

[78] "The rego playground". (2024), [Online]. Available: https://play.openpolicyagent.org/p/OoVJOzJx7q (last visited: Jun. 6, 2024).

[79] "What is java and why do i need it?" (2024), [Online]. Available: https://www.java.com/en/download/help/whatis_java.html (last visited: Jun. 13, 2024).

[80] "Application security testing tool | software security testing solutions | checkmarx", Checkmarx.com. (2024), [Online]. Available: https://checkmarx.com/ (last visited: May 23, 2024).

[81] "Architecture - Kics". (2024), [Online]. Available: https://docs.kics.io/latest/architecture/ (last visited: May 23, 2024).

[82] “Dockerfile reference”, Docker Documentation. (2024), [Online]. Available: https://docs.docker.com/reference/dockerfile/ (last visited: May 23, 2024).

[83] “Kics/assets/queries/terraform/aws/http_port_open/test/positive.tf at master · checkmarx/kics”. (2024), [Online]. Available: https://github.com/Checkmarx/kics/blob/master/assets/queries/terraform/aws/http_port_open/test/positive.tf (last visited: Jun. 6, 2024).

[84] “Kics/assets/queries/terraform/aws/http_port_open/query.rego at master · checkmarx/kics”. (2024), [Online]. Available: https://github.com/Checkmarx/kics/blob/master/assets/queries/terraform/aws/http_port_open/query.rego (last visited: Jun. 6, 2024).

[85] “Kics/assets/queries/terraform/aws/http_port_open/metadata.json at master · checkmarx/kics”. (2024), [Online]. Available: https://github.com/Checkmarx/kics/blob/master/assets/queries/terraform/aws/http_port_open/metadata.json (last visited: Jun. 6, 2024).

[86] “Kics/docs/queries.md at master · checkmarx/kics”. (2024), [Online]. Available: https://github.com/Checkmarx/kics/blob/master/docs/queries.md (last visited: May 23, 2024).

[87] “Identity verification solutions & forensic devices by regula”, Regula. (2024), [Online]. Available: https://regulaforensics.com/ (last visited: May 24, 2024).

[88] “Jenkins”, Jenkins. (2024), [Online]. Available: https://www.jenkins.io/ (last visited: May 24, 2024).

[89] “Get started with GitLab CI/CD | GitLab”. (2024), [Online]. Available: https://docs.gitlab.com/ee/ci/ (last visited: Jun. 7, 2024).

[90] “GitHub actions documentation”, GitHub Docs. (2024), [Online]. Available: https://docs.github.com/en/actions (last visited: Jun. 7, 2024).

[91] *Checkmarx/kics*, original-date: 2020-07-08T21:46:15Z, May 24, 2024. [Online]. Available: https://github.com/Checkmarx/kics (last visited: May 25, 2024).

[92] “Bridgecrewio/checkov: Prevent cloud misconfigurations and find vulnerabilities during build-time in infrastructure as code, container images and open source packages with checkov by bridgecrew.” (2024), [Online]. Available: https://github.com/bridgecrewio/checkov/tree/main (last visited: May 25, 2024).

[93] “Overview - trivy”. (2024), [Online]. Available: https://aquasecurity.github.io/trivy/v0.52/docs/ (last visited: Jun. 7, 2024).

[94] “Trivy/rpc at main · aquasecurity/trivy”. (2024), [Online]. Available: https://github.com/aquasecurity/trivy/tree/main (last visited: May 25, 2024).

[95] “Fugue/regula: Regula checks infrastructure as code templates (terraform, CloudFormation, k8s manifests) for AWS, azure, google cloud, and kubernetes security and compliance using open policy agent/rego”. (2024), [Online]. Available: https://github.com/fugue/regula/tree/master (last visited: May 25, 2024).

[96] “Regula”. (2024), [Online]. Available: https://regula.dev/ (last visited: Jun. 7, 2024).

[97] R. J. Wieringa, *Design Science Methodology for Information Systems and Software Engineering*. Berlin, Heidelberg: Springer, 2014. [Online]. Available: https://link.springer.com/10.1007/978-3-662-43839-8 (last visited: Jun. 28, 2024).

[98] “Design science seminar”. (2024), [Online]. Available: https://falkr.github.io/design science/preparation.html (last visited: May 27, 2024).

[99] “Openstack.cloud — ansible community documentation”. (2024), [Online]. Available: https://docs.ansible.com/ansible/latest/collections/openstack/cloud/index.html (last visited: Jun. 9, 2024).

[100] “Docs overview | terraform-provider-openstack/openstack | terraform | terraform registry”. (2024), [Online]. Available: https://registry.terraform.io/providers/terraform-provider-openstack/openstack/latest/docs (last visited: Jun. 9, 2024).

[101] “Ansible galaxy - community.docker”. (2024), [Online]. Available: https://galaxy.ansible.com/ui/repo/published/community/docker/docs/ (last visited: Jun. 9, 2024).

[102] “Docs overview | kreuzwerker/docker | terraform | terraform registry”. (2024), [Online]. Available: https://registry.terraform.io/providers/kreuzwerker/docker/latest/docs (last visited: Jun. 9, 2024).

[103] A. Education, *Python-hcl2: A parser for HCL2*, version 4.3.4. [Online]. Available: https://github.com/amplify-education/python-hcl2 (last visited: Jun. 17, 2024).

[104] “Convert YAML file to dictionary in python [practical examples] | GoLinuxCloud”. Section: Python. (Mar. 5, 2022), [Online]. Available: https://www.golinuxcloud.com/python-convert-yaml-file-to-dictionary/ (last visited: Jun. 17, 2024).

[105] “Services top-level elements”, Docker Documentation. (2024), [Online]. Available: https://docs.docker.com/compose/compose-file/05-services/ (last visited: Jun. 17, 2024).

Appendix A

In-depth Domain-agnostic Policy as Code

This Appendix provides a more comprehensive view and implementation details of the domain-agnostic PaC, also known as APaC, presented in chapter 5.

A.1 Project structure

The structure of this project[1] consists of two main folders:

- *infrastructure-provisioning-and-deployment-examples*: this folder contains the four implementations of the infrastructure proposed in Figure 5.3.
    - *ansible-docker*: this folder contains the file *playbook.yml*, representing the infrastructure code for Ansible deployed on Docker, written in YAML.
    - *ansible-openstack*: this folder contains the file *playbook.yml*, representing the infrastructure code for Ansible deployed on OpenStack., written in YAML.
    - *terraform-docker*: This folder contains the file *main.tf*, representing the infrastructure code for Terraform deployed on Docker, written in HCL.
    - *terraform-openstack*: This folder contains the file *main.tf*, representing the infrastructure code for Terraform deployed on OpenStack, written in HCL.
- *proof-of-concept*: this folder contains three different files, which are fundamental in the execution of the APaC architecture defined in Figure 5.1.
    - *vulnerable-ports-exposure.rego*: this policy file, written in Rego, implementing the security rules defined in subsection 5.4.3. Such file is shown in Listing 12.

---

[1] The GitHub repository for this project is available at https://github.com/frasan15/Agnostic-Policy-as-Code-APaC

  - *parser.py*: the Parser, written in Python, implements the conversion from the infrastructure specific code to a generic JSON file. Such file is illustrated in details in section A.3.
  - *network_infrastructure.json*: after the execution of the Parser, this file will be created, containing the JSON file representing the infrastructure itself.

A.2 Infrastructure code details

This section provides the code details for the four infrastructure implementations provided in this project.

A.2.1 Terraform's infrastructure code

Listing 14 illustrates the HCL code for deploying the infrastructure on OpenStack from Terraform.

Listing 15 illustrates the HCL code for deploying the infrastructure on Docker from Terraform.

```
terraform {
  required_version = ">= 0.14.0"
  required_providers {
    openstack = {
      source  = "terraform-provider-openstack/openstack"
      version = "~> 1.53.0"
    }
  }
}

resource "openstack_networking_network_v2" "network_1" {
  name = "network1"
  admin_state_up = "true"
}

resource "openstack_networking_subnet_v2" "subnet_1" {
  name = "subnet1"
  network_id = openstack_networking_network_v2.network_1.id
  cidr = "192.168.111.0/24"
  ip_version = 4
}

resource "openstack_networking_secgroup_v2" "secgroup_1" {
  name        = "secgroup_1"
  description = "Expose port 80" # remember to change this if you modify the rules
}
```

```

resource "openstack_networking_secgroup_rule_v2" "secgroup_rule_1" {
  direction         = "ingress"
  ethertype         = "IPv4"
  protocol          = "tcp"
  port_range_min    = 80
  port_range_max    = 80
  remote_ip_prefix  = "0.0.0.0/0"
  security_group_id = openstack_networking_secgroup_v2.secgroup_1.id
}

resource "openstack_networking_secgroup_v2" "secgroup_2" {
  name = "secgroup_2"
  description = "Expose port 22"
}

resource "openstack_networking_secgroup_rule_v2" "secgroup_rule_2" {
  direction         = "ingress"
  ethertype         = "IPv4"
  protocol          = "tcp"
  port_range_min    = 22
  port_range_max    = 22
  remote_ip_prefix  = "0.0.0.0/0"
  security_group_id = openstack_networking_secgroup_v2.secgroup_2.id
}

resource "openstack_networking_secgroup_v2" "secgroup_3" {
  name = "secgroup_3"
  description = "Expose port 443"
}

resource "openstack_networking_secgroup_rule_v2" "secgroup_rule_3" {
  direction         = "ingress"
  ethertype         = "IPv4"
  protocol          = "tcp"
  port_range_min    = 443
  port_range_max    = 443
  remote_ip_prefix  = "0.0.0.0/0"
  security_group_id = openstack_networking_secgroup_v2.secgroup_3.id
}

resource "openstack_networking_port_v2" "port_server_1" {
  name = "port_server_1"
  network_id = openstack_networking_network_v2.network_1.id
  admin_state_up = "true"
  security_group_ids = [openstack_networking_secgroup_v2.secgroup_1.id]

  fixed_ip {
    subnet_id = openstack_networking_subnet_v2.subnet_1.id
    ip_address = "192.168.111.10"
  }
}
```

```

resource "openstack_networking_port_v2" "port_server_2" {
  name = "port_server_2"
  network_id = openstack_networking_network_v2.network_1.id
  admin_state_up = "true"
  security_group_ids = [openstack_networking_secgroup_v2.secgroup_2.id]

  fixed_ip {
    subnet_id = openstack_networking_subnet_v2.subnet_1.id
    ip_address = "192.168.111.11"
  }
}

resource "openstack_networking_port_v2" "port_server_3" {
  name = "port_server_3"
  network_id = openstack_networking_network_v2.network_1.id
  admin_state_up = "true"
  security_group_ids = [openstack_networking_secgroup_v2.secgroup_3.id]

  fixed_ip {
    subnet_id = openstack_networking_subnet_v2.subnet_1.id
    ip_address = "192.168.111.12"
  }
}

resource "openstack_compute_instance_v2" "server_1" {
  depends_on = [ openstack_networking_secgroup_rule_v2.secgroup_rule_1 ]
  name            = "server1"
  flavor_name     = "gx1.2c4r"
  image_id        = "db1bc18e-81e3-477e-9067-eecaa459ec33"
  network {
    port = openstack_networking_port_v2.port_server_1.id
  }
  security_groups = [openstack_networking_secgroup_v2.secgroup_1.name]
  key_pair = "MySecondKey"

}

resource "openstack_compute_instance_v2" "server_2" {
  depends_on = [ openstack_networking_secgroup_rule_v2.secgroup_rule_2 ]
  name            = "server2"
  flavor_name     = "gx1.2c4r"
  image_id        = "db1bc18e-81e3-477e-9067-eecaa459ec33"
  network {
    port = openstack_networking_port_v2.port_server_2.id
  }
  security_groups = [openstack_networking_secgroup_v2.secgroup_2.name]
  key_pair = "MySecondKey"

}

resource "openstack_compute_instance_v2" "server_3" {
```

```
  depends_on = [ openstack_networking_secgroup_rule_v2.secgroup_rule_3 ]
  name            = "server3"
  flavor_name     = "gx1.2c4r"
  image_id        = "db1bc18e-81e3-477e-9067-eecaa459ec33"
  network {
    port = openstack_networking_port_v2.port_server_3.id
  }
  security_groups = [openstack_networking_secgroup_v2.secgroup_3.name]
  key_pair = "MySecondKey"

}

resource "openstack_networking_router_v2" "router_1" {
  name = "router1"
  admin_state_up = "true"
  external_network_id = "730cb16e-a460-4a87-8c73-50a2cb2293f9" # ntnu-internal
}

resource "openstack_networking_router_interface_v2" "router_interface_1" {
  router_id = openstack_networking_router_v2.router_1.id
  subnet_id = openstack_networking_subnet_v2.subnet_1.id
}

resource "openstack_networking_floatingip_v2" "myip"{
  depends_on = [ openstack_compute_instance_v2.server_1, openstack_networking_router_interface_v2.rou
  pool = "ntnu-internal"
  port_id = openstack_networking_port_v2.port_server_1.id
}

resource "openstack_networking_floatingip_v2" "myip1"{
  depends_on = [ openstack_compute_instance_v2.server_3, openstack_networking_router_interface_v2.rou
  pool = "ntnu-internal"
  port_id = openstack_networking_port_v2.port_server_3.id
}
```

Listing 14: This HCL file represents the Terraform configuration for deploying the infrastructure on OpenStack

```
terraform {
  required_providers {
    docker = {
      source  = "kreuzwerker/docker"
      version = "~> 3.0.1"
    }
  }
}

resource "docker_image" "nginx" {
  name         = "nginx"
  keep_locally = false
}
```

```

resource "docker_network" "network1" {
  name   = "network1"
  driver = "bridge"
  ipam_config {
    subnet = "192.168.111.0/24"
  }
}

resource "docker_container" "server1" {
  image = docker_image.nginx.image_id
  name  = "server1"

  networks_advanced {
    name = docker_network.network1.name
    ipv4_address = "192.168.111.10"
  }
    ports {
      internal = 80
      external = 8000
      ip = "0.0.0.0/0"
      protocol = "tcp"
    }
}

resource "docker_container" "server2" {
  image = docker_image.nginx.image_id
  name  = "server2"

  networks_advanced {
    name = docker_network.network1.name
    ipv4_address = "192.168.111.11"
  }
  ports {
    internal = 22
    external = 8001
    ip = "255.255.255.255/0"
    protocol = "tcp"
   }
}

resource "docker_container" "server3" {
  image = docker_image.nginx.image_id
  name  = "server3"

  networks_advanced {
    name = docker_network.network1.name
    ipv4_address = "192.168.111.12"
  }
  ports {
    internal = 443
    external = 8002
```

```
    ip = "0.0.0.0/0"
    protocol = "tcp"
  }
}
```

Listing 15: This HCL file represents the Terraform configuration for deploying the infrastructure on Docker

A.2.2 Ansible's infrastructure code

Listing 16 illustrates the YAML code for deploying the infrastructure on OpenStack from Ansible.

Listing 17 illustrates the YAML code for deploying the infrastructure on Docker from Ansible.

```
- name: Provision an infrastructure on OpenStack
  hosts: localhost
  tags: ['deploy']
  tasks:
  - name: Create a network
    openstack.cloud.network:
      state: present
      name: network1
      external: false

  - name: Create a subnet
    openstack.cloud.subnet:
      state: present
      network_name: network1
      name: subnet1
      cidr: 192.168.111.0/24
    register: subnet_info

  - name: Create (or update) a security group with security group rules
    openstack.cloud.security_group:
      state: present
      name: secgroup_1
      security_group_rules:
        - ether_type: IPv4
          direction: ingress
          description: Expose port 80
          protocol: tcp
          port_range_min: 80
          port_range_max: 80
          remote_ip_prefix: 0.0.0.0/0

  - name: Create (or update) a security group with security group rules
    openstack.cloud.security_group:
```

```
      state: present
      name: secgroup_2
      security_group_rules:
        - ether_type: IPv4
          direction: ingress
          description: Expose port 22
          protocol: tcp
          port_range_min: 22
          port_range_max: 22
          remote_ip_prefix: 0.0.0.0/0
    register: opa

  - name: Create (or update) a security group with security group rules
    openstack.cloud.security_group:
      state: present
      name: secgroup_3
      security_group_rules:
        - ether_type: IPv4
          direction: ingress
          description: Expose port 443 (HTTPS)
          protocol: tcp
          port_range_min: 443
          port_range_max: 443
          remote_ip_prefix: 0.0.0.0/0

  - name: Create a network inteface for server1
    openstack.cloud.port:
      state: present
      name: port_server_1
      network: network1
      fixed_ips:
        - ip_address: 192.168.111.10
          subnet_id: "{{ subnet_info.id }}"

  - name: Create a network inteface for server2
    openstack.cloud.port:
      state: present
      name: port_server_2
      network: network1
      fixed_ips:
        - ip_address: 192.168.111.11
          subnet_id: "{{ subnet_info.id }}"

  - name: Create a network inteface for server3
    openstack.cloud.port:
      state: present
      name: port_server_3
      network: network1
      fixed_ips:
        - ip_address: 192.168.111.12
          subnet_id: "{{ subnet_info.id }}"

```

```
- name: Deploy server1
  openstack.cloud.server:
    state: present
    name: server1
    auto_ip: false
    image: db1bc18e-81e3-477e-9067-eecaa459ec33
    key_name: MySecondKey
    timeout: 200
    flavor: gx1.2c4r
    nics:
      - port-name: port_server_1
    security_groups:
      - secgroup_1
  register: instance

- name: Deploy server2
  openstack.cloud.server:
    state: present
    name: server2
    auto_ip: false
    image: db1bc18e-81e3-477e-9067-eecaa459ec33
    key_name: MySecondKey
    timeout: 200
    flavor: gx1.2c4r
    nics:
      - port-name: port_server_2
    security_groups:
      - secgroup_2

- name: Deploy server3
  openstack.cloud.server:
    state: present
    name: server3
    auto_ip: false
    image: db1bc18e-81e3-477e-9067-eecaa459ec33
    key_name: MySecondKey
    timeout: 200
    flavor: gx1.2c4r
    nics:
      - port-name: port_server_3
    security_groups:
      - secgroup_3

- name: Create a router
  openstack.cloud.router:
    state: present
    name: router1
    network: 730cb16e-a460-4a87-8c73-50a2cb2293f9
    interfaces:
      - net: network1
        subnet: subnet1
        portip: 192.168.111.15
```

```yaml

  - name: Assign a floating ip to server1
    openstack.cloud.floating_ip:
      state: present
      reuse: true
      server: server1
      network: 730cb16e-a460-4a87-8c73-50a2cb2293f9
      fixed_address: 192.168.111.10
      wait: true
      timeout: 180

  - name: Assign a floating ip to server3
    openstack.cloud.floating_ip:
      state: present
      reuse: true
      server: server3
      network: 730cb16e-a460-4a87-8c73-50a2cb2293f9
      fixed_address: 192.168.111.12
      wait: true
      timeout: 180
```

Listing 16: This YAML file represents the Ansible configuration for deploying the infrastructure on OpenStack

A.3 APaC, Parser

This section depicts the code for the Parser implementation. The Parser includes four functions, representing the conversions needed for the four different infrastructure implementations provided above, for converting the specific infrastructure code into a higher-level JSON file.

When the parser is invoked, we need to specify which infrastructure code we want to run the Parser against; we do this by specifying the IaC tool, the infrastructure provider, the input file path where the infrastructure code is located and the ouput file path where we want the resulting JSON file to be generated; using, respectively the CLI arguments *tool*, *provider*, *i* and *o*. For instance, if we want to run the Parser to convert the Ansible implementation deployed on OpenStack, we need to run the command `python parser.py -tool ansible -provider openstack -i` *`input file path`* `-o` *`output file path`*.

Next, the Parser converts the HCL, or YAML, code into a Python dictionary, using specific libraries [103] [104]. Finally, a new file called *network_infrastructure.json* is generated where the infrastructure is represented using the generic keywords provided by the Taxonomy in Figure 5.2. An example of this file is provided in Listing 11.

Listing 18 shows the Python code for implementing such Parser.

Listing 17 This YAML file represents the Ansible configuration for deploying the infrastructure on Docker

```yaml
- name: "Provision an infrastructure on Docker"
  hosts: localhost
  tags: ['deploy']
  become: true
  tasks:
    - name: Pull nginx Docker image
      community.docker.docker_image:
        name: nginx
        source: pull

    - name: Create network
      community.docker.docker_network:
        name: network1
        ipam_config:
          - subnet: 192.168.111.0/24

    - name: Run server1 container
      community.docker.docker_container:
        name: server1
        image: nginx
        networks:
          - name: network1
            ipv4_address: "192.168.111.10"
        ports:
          - "0.0.0.0:8000:80"

    - name: Run server2 container
      community.docker.docker_container:
        name: server2
        image: nginx
        networks:
          - name: network1
            ipv4_address: "192.168.111.11"
        ports:
          - "255.255.255.255:8001:22"

    - name: Run server3 container
      community.docker.docker_container:
        name: server3
        image: nginx
        networks:
          - name: network1
            ipv4_address: "192.168.111.12"
        ports:
          - "0.0.0.0:8002:443"
```

```
# Parser to convert yaml or hcl infrastructure code into an abstract json file
import hcl2
import yaml
import json
import re
import argparse

# The following lines are needed to handle the CLI parameters, which will be used (at
↪  the end of this file)
# to detect which version of the parser needs to be executed

# Initialize the parser
parser = argparse.ArgumentParser(description="Proof of Concept's Parser")

# Add arguments
parser.add_argument('--tool', type=str, help='Infrastructure as Code tool')
parser.add_argument('--provider', type=str, help='Infrastructure Provider')

# The following lines of code are needed to specify the right path where each
↪  infrastructure code file is located
current_dir = os.path.dirname(os.path.abspath(__file__))
parent_dir = os.path.dirname(current_dir)

# Paths for the infrastructure code for each of the four configurations
ansible_openstack_example = os.path.join(parent_dir, "infrastructure-provisioning-an⌋
↪  d-deployment-examples/ansible-openstack/playbook.yml")
ansible_docker_example = os.path.join(parent_dir, "infrastructure-provisioning-and-d⌋
↪  eployment-examples/ansible-docker/playbook.yml")
terraform_openstack_example = os.path.join(parent_dir, "infrastructure-provisioning-⌋
↪  and-deployment-examples/terraform-openstack/main.tf")
terraform_docker_example = os.path.join(parent_dir,
↪  "infrastructure-provisioning-and-deployment-examples/terraform-docker/main.tf")

# The following will be the json object representing the infrastructure using
↪  high-level keywords
final_results = {}
final_results["servers"] = []
final_results["network_interfaces"] = []

# The following function is needed to remove the regular expression ${} from each
↪  value in the dictionary,
# when dealing with Ansible playbooks
def process_value_ansible(value):
        if isinstance(value, list):
                return [process_value_ansible(v) for v in value]
        elif isinstance(value, dict):
                return {k: process_value_ansible(v) for k, v in value.items()}
        elif isinstance(value, str):
                return re.sub(r'\{{ | }}', '', value)
        else:
                return value
```

```

# The following function is needed to remove the regular expression ${} from each
↪  value in the dictionary,
# when dealing with Terraform files
def process_value_terraform(value):
        if isinstance(value, list):
                return [process_value_terraform(v) for v in value]
        elif isinstance(value, dict):
                return {k: process_value_terraform(v) for k, v in value.items()}
        elif isinstance(value, str):
                return re.sub(r'\${|}', '', value)
        else:
                return value

try:
        def ansible_openstack():
                # opening a file
                with open(ansible_openstack_example, 'r') as stream:
                        # Converts yaml document to python object
                        first = yaml.safe_load(stream)
                first = first[0]['tasks']
                second = []
                for item in first:
                        second.append(process_value_ansible(item))

                # Convert array of objects into an object of objects
                ansible_dictionary = {}
                ansible_dictionary['network'] = []
                ansible_dictionary['subnet'] = []
                ansible_dictionary['security_group'] = []
                ansible_dictionary['port'] = []
                ansible_dictionary['server'] = []
                ansible_dictionary['router'] = []
                ansible_dictionary['floating_ip'] = []

                for obj in second:
                # Get the second key of the object dynamically
                        pre_key = list(obj.keys())[1]
                        key = pre_key.split('.', 2)[2]

                        ansible_dictionary[key].append(obj[pre_key])

                for server in ansible_dictionary['server']:
                        server_name = server['name']
                        server_security_groups = server['security_groups']

                        # Initialize a list to store the exposed ports and the
                        ↪  network interfaces of the current server
                        exposed_ports = []
                        network_interfaces = []

                        # Iterate over each security group of the server
```

```
            for security_group_name in server_security_groups: # each item already represents the security group name

                # Find the corresponding security group in the list of security groups
                for sg in ansible_dictionary['security_group']:
                    if sg['name'] == security_group_name:
                        # get the security group rules of the current security group
                        security_group_rules = sg['security_group_rules']

                        # Iterate over each security group rule and port range min and max
                        for rule in security_group_rules:
                            port_range_min = rule['port_range_min']
                            port_range_max = rule['port_range_max']

                            # Add each port in the range to the exposed ports list; only if the port range is not None
                            if port_range_max is not None and port_range_min is not None:
                                exposed_ports.extend(range(port_range_min, port_range_max + 1))

            # Remove duplicates and sort the exposed ports list
            exposed_ports = sorted(list(set(exposed_ports)))

            # Iterate through each network interface of the current server, and for each of them fetches the name
            # and the info whether it has a floating ip attached to it -> you do this by scanning the floating ip
            # array, looking for a match between the server_name associated to the current floating ip and the current
            # server being analysed -> if there is a match, then the nic attached to such a server has also a floating ip
            for nic in server['nics']:
                nic_name = nic['port-name']
                network_interfaces.append(nic_name)

                is_nic_public = False

                for floating_ip in ansible_dictionary['floating_ip']:
                    if floating_ip['server'] == server_name:
```

```
                                        is_nic_public = True

                        nic_object = {
                                'name': nic_name,
                                'is_public': is_nic_public,
                        }

                        final_results['network_interfaces'].append(nic_objec⌋
                        ↪  t)

                # Create the result object for the current server, storing
                ↪  name, exposed ports and list of subnets ids
                server_object = {
                        'name': server_name,
                        'exposed_ports': exposed_ports,
                        'server_network_interfaces': network_interfaces
                }

                final_results["servers"].append(server_object)


def ansible_docker():
        with open(ansible_docker_example, 'r') as stream:
                first = yaml.safe_load(stream)
        first = first[0]['tasks']

        # Convert array of objects into an object of objects
        ansible_dictionary = {}
        ansible_dictionary['docker_image'] = []
        ansible_dictionary['docker_network'] = []
        ansible_dictionary['docker_container'] = []

        for obj in first:
        # Get the second key of the object dynamically
                pre_key = list(obj.keys())[1]
                key = pre_key.split('.', 2)[2]
                ansible_dictionary[key].append(obj[pre_key])

        for server in ansible_dictionary['docker_container']:
                server_name = server['name']
                ports_mapping = server['ports']

                # Initialize a list to store the exposed ports and the
                ↪  network interfaces of the current server
                exposed_ports = []
                network_interfaces = []

                # Iterate over each security group of the server
                # Each item already represents the security group name
                for port in ports_mapping:
                        # We use host_port:container_port as key for the
                        ↪  network interface
```

```
                                network_interfaces.append(port.split(':', 1)[1])

                                port_host_interface = port.split(':', 2)[0]
                                if port_host_interface == "0.0.0.0":
                                        is_nic_public = True
                                else:
                                        is_nic_public= False

                                nic_object = {
                                        'name': port.split(':', 1)[1],
                                        'is_public': is_nic_public
                                }
                                final_results['network_interfaces'].append(nic_objec⌋
                                ↪  t)

                                port = port.split(':', 2)[2]
                                # Here we only need the container exposed port
                                exposed_ports.append(int(port))

                        # Create the result object for the current server, storing
                        ↪  name, exposed ports and list of subnets ids
                        server_object = {
                                'name': server_name,
                                'exposed_ports': exposed_ports,
                                'server_network_interfaces': network_interfaces
                        }

                        final_results["servers"].append(server_object)


        def terraform_openstack():
                # It reads the terraform file and it parses it into a json file
                with open(terraform_openstack_example, 'r') as file:
                        first = hcl2.load(file)
                        first = {key: process_value_terraform(value) for key, value
                        ↪  in first.items()}

                first = first['resource']
                network = "openstack_networking_network_v2"
                subnet = "openstack_networking_subnet_v2"
                security_group = "openstack_networking_secgroup_v2"
                # network interfaces
                port = "openstack_networking_port_v2"
                server = "openstack_compute_instance_v2"
                router = "openstack_networking_router_v2"
                router_interface = "openstack_networking_router_interface_v2"
                floating_ip = "openstack_networking_floatingip_v2"

                terraform_dictionary = {}
                terraform_dictionary[network] = []
                terraform_dictionary[subnet] = []
```

```python
        # here there is both resources from openstack_networking_secgroup_v2 and openstack_networking_secgroup_rule_v2
        terraform_dictionary[security_group] = []
        terraform_dictionary[port] = []
        terraform_dictionary[server] = []
        terraform_dictionary[router] = []
        terraform_dictionary[router_interface] = []
        terraform_dictionary[floating_ip] = []

        # Temporary storage for secgroup rules -> this step is needed to store the resources from openstack_networking_secgroup_rule_v2 into openstack_networking_secgroup_v2
        secgroup_rules = {}

        for item in first:
            key = list(item.keys())[0]
            if key == "openstack_networking_secgroup_rule_v2":
                nested_key = list(item[key].keys())[0]
                secgroup_id = (item[key][nested_key]["security_group_id"]).split('.', 2)[1]

                if secgroup_id not in secgroup_rules:
                    secgroup_rules[secgroup_id] = []
                secgroup_rules[secgroup_id].append(item[key][nested_key])
            else:
                nested_dict = item[key]
                terraform_dictionary[key].append(list(nested_dict.values())[0])

        # Append secgroup rules to corresponding secgroup objects
        for secgroup_name, rules in secgroup_rules.items():
            for secgroup in terraform_dictionary[security_group]:
                if secgroup['name'] == secgroup_name:
                    if 'rules' not in secgroup:
                        secgroup['rules'] = []
                    secgroup['rules'].extend(rules)

        for server in terraform_dictionary[server]:
            server_name = server['name']
            server_security_groups = server['security_groups']

            # Every server_security_groups is stored as "openstack_networking_secgroup_v2.secgroup_2.name" so we need to extract the name
            for item in server_security_groups:
                item = (item).split('.', 2)[1]

            # Initialize a list to store the exposed ports and the network interfaces of the current server
            exposed_ports = []
```

```
            network_interfaces = []

            # Iterate over each security group of the server
            # Each item already represents the security group name
            for security_group_name in server_security_groups:
                # Find the corresponding security group in the list of security groups
                for sg in terraform_dictionary[security_group]:
                    if sg['name'] == (security_group_name).split('.', 2)[1]:
                        # Get the security group rules of the current security group
                        security_group_rules = sg['rules']
                        # Iterate over each security group rule and port range min and max
                        for rule in security_group_rules:
                            port_range_min = rule['port_range_min']
                            port_range_max = rule['port_range_max']

                            # Add each port in the range to the exposed ports list; only if the port range is not None
                            if port_range_max is not None and port_range_min is not None:
                                exposed_ports.extend(range(port_range_min, port_range_max + 1))

            # Remove duplicates and sort the exposed ports list
            exposed_ports = sorted(list(set(exposed_ports)))

            # Iterate through each network interface of the current server, and for each of them fetches the name
            # and the info whether it has a floating ip attached to it -> you do this by scanning the floating ip
            # array, looking for a match between the server_name associated to the current floating ip and the current
            # server being analysed -> if there is a match, then the nic attached to such a server has also a floating ip
            for nic in server['network']:
                nic_name = nic['port']
                nic_name = (nic_name).split('.', 2)[1]
                network_interfaces.append(nic_name)

                is_nic_public = False
                for item in terraform_dictionary[floating_ip]:
```

```
                if (item['port_id']).split('.', 2)[1] ==
                ↪  nic_name:
                    is_nic_public = True

            nic_object = {
                'name': nic_name,
                'is_public': is_nic_public
            }

            final_results['network_interfaces'].append(nic_objec⌟
            ↪  t)

        # Create the result object for the current server, storing
        ↪  name, exposed ports and list of subnets ids
        server_object = {
            'name': server_name,
            'exposed_ports': exposed_ports,
            'server_network_interfaces': network_interfaces
        }

        final_results["servers"].append(server_object)


def terraform_docker():
    # It reads the terraform file and it parses it onto a json file
    with open(terraform_docker_example, 'r') as file:
        first = hcl2.load(file)

    first = {key: process_value_terraform(value) for key, value in
    ↪  first.items()}
    first = first['resource']

    terraform_dictionary = {}
    terraform_dictionary['docker_image'] = []
    terraform_dictionary['docker_network'] = []
    terraform_dictionary['docker_container'] = []

    for item in first:
        key = list(item.keys())[0]
        nested_dict = item[key]
        terraform_dictionary[key].append(list(nested_dict.values())[⌟
        ↪  0])

    for server in terraform_dictionary['docker_container']:
        server_name = server['name']

        # Initialize a list to store the exposed ports and the
        ↪  network interfaces of the current server
        exposed_ports = []
        network_interfaces = []

        for port in server['ports']:
```

```
                exposed_ports.append(port['internal'])

            exposed_ports = sorted(list(set(exposed_ports)))

            for port in server['ports']:
                network_interface_id = str(port['internal']) + ':' + str(port['external'])
                network_interfaces.append(network_interface_id)

                if port['ip'] == "0.0.0.0/0":
                    is_nic_public = True
                else:
                    is_nic_public = False

                nic_object = {
                'name': network_interface_id,
                'is_public': is_nic_public
                }

                final_results['network_interfaces'].append(nic_object)

            # Create the result object for the current server, storing name, exposed ports and list of subnets ids
            server_object = {
                'name': server_name,
                'exposed_ports': exposed_ports,
                'server_network_interfaces': network_interfaces
            }

            final_results["servers"].append(server_object)

    # Parse the arguments. The arguments can be retrieve by args.tool or args.provider
    args = parser.parse_args()
    # run the proper parser according to the IaC tool and the infrastructure provider
    if args.tool == "ansible" and args.provider == "openstack":
        ansible_openstack()
    elif args.tool == "ansible" and args.provider == "docker":
        ansible_docker()
    elif args.tool == "terraform" and args.provider == "openstack":
        terraform_openstack()
    elif args.tool == "terraform" and args.provider == "docker":
        terraform_docker()
    else:
        raise Exception("Infrastructure as Code tool or Infrastructure Provider not supported")

    print("FINAL JSON: ", json.dumps(final_results, indent=4))
    # The following are the operations needed to write the json file on the current folder
```

```
        # Define the path for the JSON file
        json_file_path = os.path.join(current_dir, "network_infrastructure.json")
        # Write data to the JSON file
        with open(json_file_path, 'w') as json_file:
                json.dump(final_results, json_file, indent=4)
        print("JSON file has been generated and saved at:", json_file_path)

except Exception as e:
    print(e)
```

Listing 18: This Python file is the Parser for translating any infrastructure code file, among the ones mentioned in this thesis, into a generic JSON file

A.4 How to detect whether a server is accessible from outside

For OpenStack deployments, the server's internet connectivity is determined by the presence of a floating IP. If a floating IP is assigned, it is assumed that the server is accessible from outside the internal network.

For Docker deployments, specifying a floating IP is not possible. Instead, internet accessibility is assessed by examining the IP range to which the container exposes its ports. If the range is 0.0.0.0/0, the server is assumed to be accessible from outside; otherwise, it is not. The initial plan was to verify port exposure using the "ports" field in Docker Compose [105], which only specifies the container port to be exposed without associating it with a specific host port. However, due to API limitations, the current APaC implementation uses the "expose" field in Docker Compose, which maps the exposed container port to a host port. Consequently, it would be more accurate to state that every port is exposed in the present configuration, as exposing container port 80 to any host port implies that this port is accessible from the outside.

A.5 Open Policy Agent's running commands

The Rego file used to check the policy rules defined in subsection 5.4.3, is illustrated in Listing 12. This file is checked against the *network_infrastructure.json* file by the command `/usr/local/bin/opa eval -i network_infrastructure.json -d vulnerable-ports-exposure.rego "data.example.output"`. An example of the policy decision result from the OPA engine is shown in Listing 13.