POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

Policy as Code, how to automate cloud compliance verification with open-source tools

Supervisor
Prof. Riccardo Sisto

Candidate
Mattia Caracciolo

Reply Liquid corporate tutors
Dr. Ivan Aimale
Dr. Francesco Borgogni
Dr. Luigi Casciaro

Accademic Year 2022-2023

Summary

Container infrastructures, along with the use of the cloud, represent a new paradigm of application development and release that has become widespread in recent years. Although, on the one hand, such infrastructures bring benefits in scalability, management, and application compatibility, on the other hand, they are not to be considered "secure-by-default."

Security enforcement in these environments is a complex task if approached with the "old" methodologies. Technologies are therefore evolving, and a new approach was born: "Policy as Code." This approach allows to abstract security policies into code that can then be executed to automate compliance verification of cloud applications and infrastructure. Furthermore, it permits the management of policies as normal source code, enabling the implementation of all proven software development best practices such as version control, automated testing, and automated deployment.

This thesis work analyzes the state of the art of Policy as Code and investigates the different open-source solutions proposed in the market, their effectiveness, and how they can be integrated into a continuous integration continuous delivery pipeline. The work starts with an overview of cloud compliance, why it is important, and what issues arise with its manual implementation. Subsequently, it investigates how the subject can be integrated into the DevSecOps methodology by analyzing how and in which steps of the development process it can be implemented in order to automate cloud compliance.

Thereafter, there is an analysis of the implementation of a proof-of-concept that has been developed specifically for this purpose. It is used to perform a security assessment and test the effectiveness of the tools against proof-of-concept behavior and default configuration. Specifically, tfsec and Regula are the tools analyzed with regard to infrastructure as code security, Cloud Custodian for cloud security posture management, and Gatekeeper for cloud workload protection.

The thesis shows a description of their output and the results obtained by testing compliance with policies against the proof-of-concept, as well as the mitigation strategies that should be applied. The results show that tfsec and Regula can be used inside a continuous integration continuous delivery pipeline to prevent the deployment of resources that are not compliant with the defined policies and also demonstrate that the default configuration of the proof-of-concept infrastructure is quite insecure; for example, public cloud storage containers are not encrypted by default. Also, it is shown how Cloud Custodian and Gatekeeper are useful for security audits, as they allow for the verification of the actual behavior of both the infrastructure and the workload, notifying the presence of non-compliant resources. Finally, the work analyzes the performance impact that infrastructure as code tools have on pipeline execution and the resource consumption of cloud security posture management and cloud workload protection tools.

Acknowledgements

I would like to thank my supervisor Professor Riccardo Sisto, for his valuable advice and helpfulness. Thanks for providing me with key insights in the writing of this thesis and for directing me in moments of indecision.

I would also like to thank my corporate tutors, Luigi Casciaro, Francesco Borgogni, and Ivan Aimale for proposing the topic covered in this thesis and giving me valuable guidance and suggestions. My acknowledgments also go to all my colleagues for their essential support and also for the experience they have allowed me to live within Reply Liquid, which has been a reason for my personal and professional growth.

I cannot but thank my parents. Mom and Dad thank you for supporting me and helping me through the most difficult times, mainly due to the distance that separated us in the past two years. Thank you for everything, I love you all.

Finally, a huge thank you goes to all my friends for supporting and sustaining me over the years.

Contents

List of Tables

List of Figures

Listings

Chapter 1

Introduction

Container infrastructures, along with the use of the cloud, represent a new paradigm of application development and release that has become widespread in recent years. Although, on the one hand, such infrastructures bring benefits in scalability, management, and application compatibility, on the other hand, they are not to be considered "secure-by-default."

Compliance verification in these environments is a complex task if approached with the "old" methodologies. Policies controls can be enforced manually or embedded into the code. However, these approaches can lead to various disadvantages in policy development and enforcement. Manual enforcement is tedious, error-prone, and inefficient. On the other hand, policy enforcement can occur automatically. However, in this case, the controls are embedded into the code. As a result, policies turn out to be tightly bound to the code itself, resulting in the inability to reuse the policy code in other projects. Also, the rest of the code can be affected by changes and errors in the policy code.

Technologies are therefore evolving, and a new approach was born: "Policy as Code." By following this methodology, security policies can be abstracted into code that can then be executed by an external tool to automate compliance verification of cloud applications and infrastructure without the problems noted above. Another advantage given by this approach is that policies can be treated in the same way as normal source code hence enabling the implementation of all proven software development best practices such as version control, automated testing, and automated deployment. Since Policy as Code encourages automation, this work also investigates how the approach and tools can be integrated into the DevSecOps methodology.

1.1 Objective

The thesis explores the state of the art of Policy as Code as well as analyzes the different open-source solutions proposed in the market. The objective of this study is to show how cloud compliance can be automated by utilizing the open-source solutions found during the first phase. This is done by analyzing a proof-of-concept implemented by following the Policy as Code approach. The study aims to demonstrate that the default configuration of the proof-of-concept infrastructure is quite insecure as well as to analyze the performance of the tools found in the first phase.

1.2 Thesis structure

The work is structured as follows:

- **Chapter 2** contains some background about cloud computing and cloud security and explains the concept of cloud compliance;
- **Chapter 3** consists of a theoretical overview of the concepts akin to Policy as Code and how Policy as Code can be integrated inside the DevSecOps methodology;
- **Chapter 4** shows an overview of the tools used for implementing the proof-of-concept;
- **Chapter 5** explains the structure of the proof-of-concept and describes how the tools are operated within it;
- **Chapter 6** illustrates the violations detected by the tools and the actions performed over the proof-of-concept for mitigating them;
- **Chapter 7** describes the results of a performance evaluation conducted on the tools utilized inside the proof-of-concept;
- **Chapter 8** closes the thesis with the conclusions and possible future works.

Chapter 2

Cloud Compliance

This chapter contains some background and explains the basics of cloud computing and cloud security. Particular attention is given to the concepts of compliance, guidelines, standards, regulations, and shared responsibility.

2.1 Cloud computing overview

The National Institute of Standards and Technology (NIST) defines cloud computing as *"a model for enabling convenient, on demand network access to a shared pool of configurable computing resources (e.g., network, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction"* [1].

This computation paradigm has become increasingly popular during the last few years, and it appears that this trend will continue in the years to come. Indeed, according to Statista, the revenue in the Public Cloud market is projected to reach USD 525.601 billion in 2023. It is expected to expand at USD 881.80 billion by 2027, with a compound annual growth rate of 13.81% from 2023 to 2027 [2].

Cloud computing has become very popular thanks to its great flexibility, which is obtained by leveraging its essential characteristics and the different service models offered to consumers.
According to NIST [1] there are five essential characteristics:

- **On-demand self-service**. Computing capabilities offered by service providers can be provisioned automatically to consumers.
- **Broad network access**. Heterogeneous client platforms utilize standard mechanisms to access the capabilities available over the network.

- **Resource pooling**. The provider's computing resources are pooled to serve multiple consumers using a multi-tenant model. Both physical and virtual resources are dynamically assigned and reassigned by following consumer demand.

- **Rapid elasticity**. Capabilities can be elastically provisioned and released according to consumer demand.

- **Measured service**. Resource usage can be monitored, controlled, and reported.

NIST [1] also lists three essential service models:

- **Software-as-service**. The consumer can use the provider's applications that are hosted on a cloud infrastructure.

- **Platform-as-service**. The consumer can deploy onto the cloud infrastructure applications supported by the provider. Also, it has control over the deployed applications and possibly configuration settings for the application-hosting environment.

- **Infrastructure-as-service**. The consumer has at his disposal: processing, storage, networks, and other fundamental computing resources provided by the cloud provider. The consumer can use them to deploy and run arbitrary software. Also, it has control over operating systems, storage, and deployed applications; and possibly limited control of select networking components.

Note that in each service model, the consumer does not manage or control the underlying cloud infrastructure. As the cloud provider is the entity in charge of doing so. Finally, NIST [1] defines four deployment models:

- **Private cloud**. Only a single organization can use the cloud infrastructure. It may exist on or off premises and may be controlled by the organization, a third party, or some combination of them.

- **Community cloud**. Only a community of consumers from organizations that have shared concerns can use the cloud infrastructure. It may exist on or off premises and may be controlled by one or more of the organizations in the community, a third party, or some combination of them.

- **Public cloud**. Anyone can use the cloud infrastructure. It may be owned, managed, and operated by an organization. It exists on the premises of the cloud provider.
- **Hybrid cloud**. The cloud infrastructure is composed of two or more distinct cloud infrastructures. Despite each infrastructure remaining a unique entity, they are linked by technologies that enable data and application portability.

2.2 Cloud security overview

Just like traditional IT systems, Cloud Computing is not exempt from security threats, vulnerabilities, and attacks. Additionally, some of the essential characteristics, such as resource pooling and broad network access, can lead to serious threats. For these reasons, despite its popularity, in the last years, the concern/interest in cloud security has increased. A. Hendre and K. P. Joshi [3], identified various cybersecurity threats which could affect a cloud computing system:

- **Data breaches**. They are an incident in which data, which should be protected, are accessed by an unauthorized individual. Hence, the confidentiality of data, and eventually the organization, is compromised.
- **Data loss**. This threat could devastate an organization. Hardware failure or malicious attacks on the system can both be the cause of this threat.
- **Account or service traffic hijacking**. The confidentiality and integrity of the users are affected by this threat. Hackers can steal users' personal data like bank credentials.
- **Insecure interfaces and APIs**. Interfaces and APIs are used by users and providers to perform communications. Weak APIs and Interfaces expose security vulnerabilities in the availability, confidentiality, and integrity of cloud environments.
- **Denial of service**. Valid users could not access their data or applications.
- **Malicious Insiders**. They are people within the organization who can exploit their position to access and misuse the data.

- **Abuse of cloud services**. The multitenancy feature of the cloud can be misused by attackers in order to hack into other organizational data.
- **Insufficient due diligence**. Cost savings represents one of the main reasons why currently many organizations are adopting the cloud. However, they are not aware of the other threats.
- **Shared Technology vulnerabilities**. Resource sharing allows cloud providers to deliver their services in a scalable way. However, attackers access different resources by exploiting vulnerabilities present in the technology which implements multi-tenancy.

The table 2.1 lists cybersecurity threats and the standard, or regulation, which is best suited to address them. Despite some of them are well known in the literature, there are some threats such as abuse of cloud services, insufficient due diligence, and shared technology vulnerabilities that are specific to the cloud environment. They rely on some of its essential characteristics. More precisely, the insufficient due diligence threat is caused, even if indirectly, by the measured service. Instead, shared technology vulnerabilities and abuse of cloud services are a direct consequence of resource pooling characteristics.

2.3 Standards, guidelines, and regulations

During the last few years, due to the increasing relevance of cybersecurity and privacy awareness, more and more regulations have been developed and adopted around the world.

Adhering to standards, guidelines, and regulations can improve the "security posture" of the organization. The NIST defines security posture [6] as *"the security status of an enterprise's networks, information, and systems based on information security resources (e.g., people, hardware, software, policies) and capabilities in place to manage the defense of the enterprise and to react as the situation changes."* Therefore, standards, guidelines, and regulations can be used to produce policies that must be enforced in order to be compliant with them.

Cybersecurity standards, and also guidelines, are designed to protect a user or an organization's cyber environment (or at least they try to do so). More specifically, cybersecurity standards aim to minimize security risks that may occur in the system, such as trying to prevent an attack or at least

Type of threat	**Recommended Security Standard or Regulation**
Data breaches	STIG, FedRAMP, DMTF-CADF, ISO-27001, FIPS 140-2, PCI DSS, ISO 27002, HIPAA, SOX
Data loss	STIG, FedRAMP, DMTF-CADF, FIPS 140-2, PCI DSS, ISO 27002, HIPAA, SOX
Account or service hijacking	STIG, FedRAMP, DMTF-OVF, ISO-27002, FIPS 140-2, NIST 800-61, ISO 17799
Insecure interfaces/API	OASIS and OVF, DMTF-CADF, ISO 27002, FIPS 140-2
Denial of services	PCI DSS, ISO 27001, HIPAA, SOX, NIST 800-61, ISO 17799
Malicious insiders	ISO 27002, FIPS 140-2, Vaultive, FedRAMP
Abuse of cloud services	NIST 800-61, ISO 17799, NIST 800-50
Insufficient due diligence	EDRM, NIST 800-61
Shared technology vulnerabilities	ISO 27001, FIPS 140-2, PCI DSS, ISO 27002, HIPAA, SOX, MPAA

Table 2.1. Recommendation of security standard or regulation based on security threat [3].

mitigate it [7]. Instead, privacy standards have the purpose of integrating privacy requirements into information processes, systems, and services in order to protect the individuals' "Personal Identifiable Information" (PII) [8]. In general, there are two ways [9] in which standards are regulated:

- **Voluntary standards**, their usage is optional, even though in some cases a regulating agency may mandate their use;
- **Mandatory standards**, typically implement laws and regulations thus their usage is compulsory.

Standards and guidelines are composed of a collection of rules and requirements for activities or their results. Note that despite their similarities, there is a subtle difference: standards have a higher level of consensus and formality if compared to guidelines. [9]

Furthermore, cybersecurity and privacy regulations are composed of directives issued by governments and implemented by laws. In some cases, they may also be augmented by cybersecurity standards. Since they are implemented by laws their enforcement is mandatory and organizations must comply with them. Here are some examples of standards and regulations:

- NIST CSF
- NIST SP-800 series
- GDPR
- FISMA
- CIS Critical Security Controls
- PCI-DSS
- ISO/IEC 27000 family
- CSA Cloud Control Matrix

2.4 Cloud compliance

Nowadays compliance with regulations and standards has become one of the most important aspects taken into account when designing an Information Technology (IT) system. IT compliance can be defined as the accordance of corporate IT systems with internal policies, processes, and regulatory standards as well as local, national, and international laws [4]. It has to not be underestimated because being compliant with applicable regulations and laws is compulsory for all companies.

Being non-compliant could lead to huge fines, e.g. infringements of the General Data Protection Regulation (GDPR) provisions could lead to administrative fines up to 20 million EUR or up to 4% of the total turnover of the preceding financial year [5].

Cloud is not exempt from compliance. Cloud-based IT systems have also to be compliant with regulations and laws to not fall into legal issues. Furthermore, it is also highly desirable to be compliant with standards and

guidelines because they could make easier the development and maintenance of the system.

Unfortunately, compliance checking is not an easy task and it could become tedious if the checks are done manually. Hence, compliance automation although not trivial is a highly desirable feature since it is less prone to errors, and helps to speed up the checking phase. However, it is not possible to perform all the controls automatically, so an analysis among them must be performed with the aim of identifying which ones can be automated.

2.5 Shared Responsibility

Before the advent of cloud computing, organizations were the only entity responsible for the security of their IT systems. Also, they were the only subject in charge of ensuring an adequate level of security for the data, applications, network, and even physical building security. Things have changed with cloud computing. According to NIST [12], there are two main actors in a cloud-based IT system:

- **Cloud Provider**. It is the entity responsible for making a service available to interested parties;
- **Cloud Consumer**. It is a person or organization that maintains a business relationship with and uses the service from a cloud provider.

In a cloud IT system both the cloud provider and cloud consumer have control over the computing resources present in the system. Therefore, both entities share security responsibilities. But how are they shared? Unfortunately, there is no clear answer to this question. The problem is how to clearly define who is responsible for what. There are various solutions, but they depend on the provider itself and the offered service model [13].

The figure 2.1 underlines that cloud consumers' responsibilities are about the security "within" the cloud. Conversely, the cloud provider is responsible for the security "of" the cloud. As an example, an organization (cloud consumer) is responsible for protecting its data, but not for the physical security of the data centers. The latter is the responsibility of the cloud provider. Unfortunately, there are also some grey areas in which the division of responsibilities is not clear at all [15]. In this case they are shared among the cloud consumer and the cloud provider and they are in charge of establishing who is responsible for what.

Figure 2.1. Areas of responsibility between cloud provider and cloud consumer [15].

As said before, the shared responsibility model explains that security responsibilities are shared between the cloud consumer and the cloud provider. Therefore management, operation, and verification of IT controls can also be divided among parties in a similar way. This leads to a not-inconsiderable advantage for the cloud consumer because it can inherit, completely or partially, IT controls already operated by the cloud provider [16]. In this way, cloud consumers can focus only on their controls and try to implement them as best as they can.

Chapter 3

Theorical overview of Policy as Code

This chapter aims to give some background against some concepts behind Policy as Code and DevSecOps and to show how Policy as Code can be integrated into the DevSecOps methodology for implementing: Infrastructure as Code Security, Cloud Security Posture Management, and Cloud Workload Protection.

3.1 Definitions

3.1.1 Policy

The Cambridge Dictionary defines the word policy as *"a set of ideas or a plan of what to do in particular situations that have been agreed to officially by a group of people, a business organization, a government, or a political party"* [18].

Policies are needed by organizations to ensure compliance with legal requirements, work within technical constraints, and avoid mistakes. In the IT scenario, the word policy acquires a specific meaning i.e. *"a set of rules that governs the behavior of a software service"*. These rules describe an ideal behavior that should be followed by the target of the policy [19]. Examples of IT policies could be: the system's databases must be encrypted, containers' images must come from trusted repositories, and each component of the system must send its log data to a defined number of different secure logs.

Traditionally, IT policies are manually enforced and composed of rules that

are written in a document [17]. Unfortunately, this approach is tedious, error-prone, inefficient, and does not scale well. Imagine a developer who is ready to deploy his code, he must check compliance with policies before doing so. Since they are manually enforced, he should await feedback from the policy team, which is in charge of checking whether the code is compliant or not. This procedure results in a slowdown of the whole development process because the developer can not check autonomously and automatically compliance with policies. Furthermore, since compliance is checked manually, there is a likelihood that the decision made by the policy team could be wrong. That is because humans are prone to make mistakes such as misunderstanding policies or misimplementing them.

Another approach is to embed IT policy implementation directly into the software service's code. Although in this way the enforcement is automatic, there are some drawbacks to following this approach. As an example, hard-coded policies cannot be reused among different applications and they cannot be shared between different teams. Finally, since policies and code are tightly coupled, business functions could be affected by policy changes or errors, and in some cases, they can also lead to an entire crash of the application [19].

3.1.2 Policy as code

As discussed before, the traditional policy enforcement methods present various problems which are incompatible with today's business needs. As specified by Y. Matharu, T. Coulter [20]: *"Policy as code is an approach to policy management in which policies are defined, updated, shared, and enforced using code."* In this case, policies rather than being embedded into code, are treated as separate entities. In this way, their code is decoupled from the service's business logic. In addition, since policies are treated as source code, it is possible to implement proven software development best practices like version control, automated testing, and automated deployment. This approach leads to various benefits [21]:

- **Sandboxing**. Policies provide the guardrails for other automated systems. As the number of automated systems increases, so does the need to protect these systems from performing dangerous actions.
- **Codification**. By representing policy logic as code, the information and logic about a policy are directly represented in code. This representation can be further augmented with comments which can be used by developers to explain the reason for policies.

- **Version Control**. Since policies are treated in the same way as code, they can be stored in simple text files managed by a version control system (VCS). For that reason, all the advantages of a modern VCS such as history, diffs, pull requests, and more are therefore inherited.
- **Testing**. Policies' syntax and behavior can be easily validated because they are just code. This characteristic promotes automated testing.
- **Automation**. Since policies are written in machine-readable code, it is possible to use various automation tools. As an example, policies can be automatically deployed into a system by using specific tools created for this purpose.

These advantages over the traditional approach are the key to better management and enforcement of policies. Moreover, they also lead to a significant acceleration of the development phase since policy compliance checks can be done automatically by tools. Finally, Policy as Code greatly helps an organization to implement compliance automation, as this methodology makes it possible for policies to be enforced automatically.

3.1.3 DevOps

Traditional (waterfall) and modern (agile) software development methods both focus on software development teams, which are in charge of developing and testing software. Conversely, the IT operations teams are in charge of deploying, maintaining, and supporting the software developed by the development team. Therefore, by following these methods, development, and operation are treated as separate and independent concepts. As a result, organizational issues such as a blame culture between both parties, issues in communication, and also delays in producing software updates are arises due to this choice. This problem could cause a decrease in software quality and process productivity [24]. Hence, to avoid those drawbacks, the industry has started to integrate both software development and IT operations by following a new development methodology called DevOps [22].

The term "DevOps", is born from the combination of the terms development and operations. As usual for a novel concept, there isn't yet a consensus on what it includes. Consequently, the term is subject to misunderstandings.

R. Jabbari, N. Ali, K. Petersen, and B. Tanveer [23] characterized DevOps by studying its definitions reported in the literature. As a result, they defined DevOps as *"a development methodology aimed at bridging the gap between*

Development (Dev) and Operations (Ops), emphasizing communication and collaboration, continuous integration, quality assurance, and delivery with automated deployment utilizing a set of development practices." This definition underlines how important is the collaboration between the Development and Operation teams. In order to figure out how to implement this methodology into the software production process, it is important to understand the DevOps life cycle. According to H. Dhaduk [25], the DevOps life cycle is a continuous loop divided into seven phases:

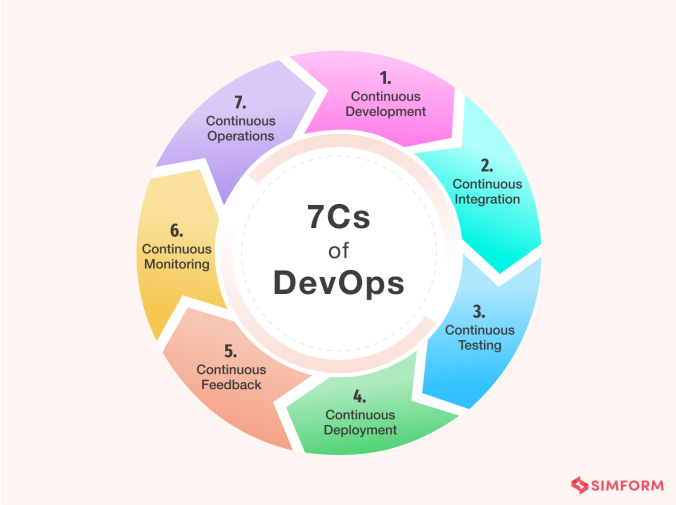

Figure 3.1. 7Cs of DevOps [25].

- **Continuous development**. This phase is primarily focused on the project's planning and coding. It is a continuous process because any requirement change or performance issue is embraced by developers and turned into code.

- **Continuous integration**. During this phase, every code update is integrated into existing code. The latter is tested at each commit with tests planned during this phase. This process makes integration a continuous approach that allows bugs to be detected or identified and source code to be modified accordingly.

- **Continuous testing**. In this phase, the software is continuously tested for bugs and errors. If a bug or an error is found, the code is sent back to the integration phase for modifications.

- **Continuous deployment**. This is the most active phase in the DevOps lifecycle. In continuous deployment, developers deploy the final code on production servers, schedule their updates, and keep the configurations consistent throughout the entire production process. Moreover, development, testing, production, and staging environments are made consistent by using containerization tools. This practice made the continuous delivery of new features in production possible

- **Continuous feedback**. During this phase, customer behavior is evaluated regularly on each release with the aim to improve future releases and deployments. Customers' feedback can be either collected through surveys and questionnaires or through social media platforms. Once feedback is collected, it is used to improve the product.

- **Continuous monitoring**. At this stage, the application's behavior is monitored continuously. This process helps the IT team to identify app performance issues, system errors, and the root cause behind them. If any critical issue is found, the application goes through the entire DevOps cycle again to find the solution.

- **Continuous operations**. This phase automates the process of launching the app and its updates by using container management systems. This approach helps to minimize application downtime, allowing companies to reduce costs due to it. It also leads to another benefit, developers save time that can be used to accelerate the application's time-to-market.

Adopting this lifecycle during software development has some significant advantages. Adopting DevOps makes continuous delivery possible, so the company can respond quickly to market needs and fix critical bugs and vulnerabilities in a timely manner. In addition, continuous testing and continuous integration assure a better quality of the final product as code is continuously tested. However, there are some drawbacks. Adopting DevOps requires an integration of various tools and technology in the software development process. That could increase the organization's IT system complexity and increment production costs. Finally, DevOps is a novel methodology thus there is a lack of standardization, experienced developers, and engineers. These issues can lead to problems with the quality of the software produced with this methodology [26].

3.1.4 Pipeline CI/CD

A Continuous Integration/Continuous Deployment (CI/CD) pipeline consists of a series of steps that must be performed in order to deliver a new version of the software. It takes advantage of automation throughout the development, testing, production, and monitoring phases of the software development life-cycle aiming to develop higher quality code, faster, minimize human error, and maintain a consistent process for how software is released. The pipeline could include tools that perform: compiling code, unit tests, code analysis, security controls, and binaries creation. CI/CD can be considered the backbone of a DevOps methodology because it brings developers and IT operations teams together to deploy software [27].

Figure 3.2. Typical CI/CD flow [27].

3.1.5 Infrastructure as Code

Historically, system administrators had the burden of configuring the hardware and software used by applications to run. Nonetheless, manual management inevitably leads to human errors and higher costs. Just like other manual processes, this one is considerably slower with respect to the automated version, and on top of that, this process does not scale well. Nowadays with Cloud Computing the number of infrastructure components has increased considerably. Therefore, it is nearly impossible to manage them with the traditional approach. Based on these considerations, a new method was needed to better control the IT infrastructure.

Infrastructure as Code (IaC) is one of the possible solutions to the previous problems. Red Hat [28] defines Infrastructure as Code as *"the managing and provisioning of infrastructure through code instead of through manual processes."*

With IaC infrastructure specifications are written into configuration files. This approach ensures that the provisioned infrastructure is always the same

unless changes are made to the configuration files. This is not the only benefit acquired. IaC, also makes easier the editing and distribution of the configurations as well as allows the division of the infrastructure into modular components. Furthermore, since the infrastructure is treated as code, it allows the use of software development methods and best practices such as version control, automated testing, and automated deployment can be used to develop the infrastructure. This is a significant advantage given by the fact that IaC allows infrastructure configuration files to be treated as if they were source code [28].

On top of that, Iac fits well with DevOps methodology, due to the automation introduced by using this approach and the possibility of integrating it into CI/CD pipelines. Moreover, it also helps to align development and operations, because the same description of the application deployment can be used by both teams. Since IaC generates the same environment each time it is used, it allows the same deployment process to be used for development, test, stage, and production environments [28]. There are two ways to approach IaC:

- **Declarative**. This approach is based on defining the desired state of the system, including what resources are needed and any properties they should have. Once defined, an IaC tool takes over the task of configuring the system. It also helps to make it easier to take down infrastructure, because it maintains a list of the current state of system objects.

- **Imperative**. With this approach, the specific commands needed to achieve the desired configuration are defined, and to do that they have to be executed in the correct order.

Although IaC tools are often able to operate in both approaches, they tend to prefer one approach over the other. Imperative tools will require developers to figure out how changes should be applied to the state of the infrastructure. Therefore, many IaC tools use instead a declarative approach. In this way, developers should define only what changes should be made to the desired state, and the declarative IaC tool will apply those changes [28].

3.1.6 DevSecOps

IT security plays a crucial role in the full life cycle of software. In the past, its role was isolated to the final stages of development and operated by a specific team. To take full advantage of the agility and responsiveness of the DevOps

approach, security must be considered throughout the whole development process. That is because, in contrast with the past, DevOps ensures rapid and frequent development cycles and even the most efficient DevOps initiatives can be nullified by outdated security. The term "DevSecOps" was created to highlight the importance of application and infrastructure security during the whole application life-cycle [29].

DevSecOps promotes collaboration between development, security, and operation teams. In this way, application and infrastructure security can be applied from the start and also integrated into the whole software development life-cycle. Automation is a distinctive feature of DevOps, it allows to implementation of rapid and frequent development cycles. DevSecOps is no different, it also encourages automation, especially for security controls, intending to not slow down the development process. Moreover, security automation is welcome to protect the Continuous Integration/Continuous Delivery (CI/CD) process as well as the overall environment and data [29].

The pipeline is an excellent basis from which a series of automated security tests and validation can be performed, in order to integrate security objectives early on in the development of an application.

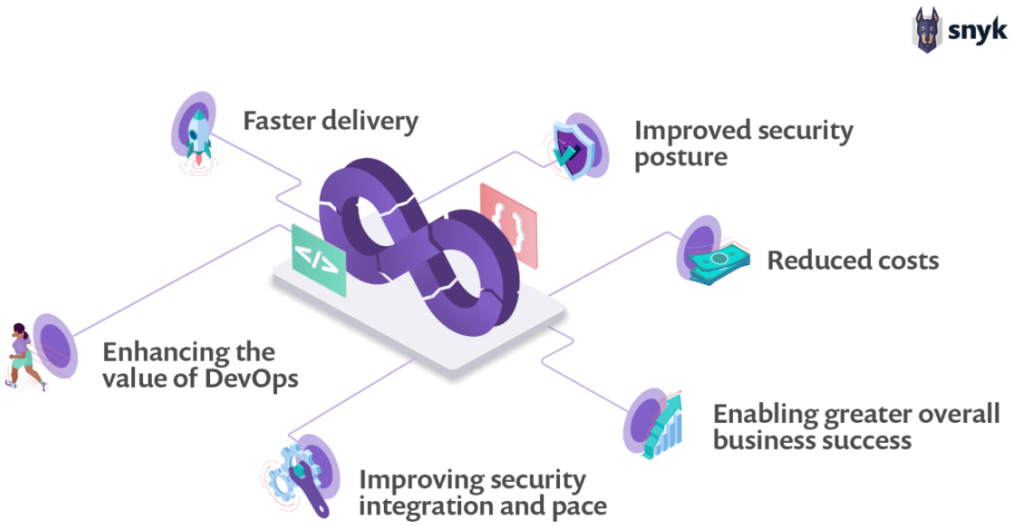

Figure 3.3. Six benefits of the DevSecOps model [30].

According to Snyk, the DevSecOps model has six main benefits [30]:

- **Faster delivery**. DevSecOps allows to identify and fix bugs before deployment by integrating security controls into the CI/CD pipeline.
- **Improved security posture**. Security is integrated into the early phases of the development process, this feature together with the shared responsibility model help to tightly integrate security, from building to deployment to the protection of production workloads.
- **Reduced costs**. Risk and operational costs are significantly reduced By identifying vulnerabilities and bugs before the deployment phase.
- **Enhancing the value of DevOps**. Integration of security practices into DevOps results in an improvement of overall security posture as well as the creation of a culture of shared responsibility.
- **Improving security integration and pace**. There is no more need to retrofit security controls in post-development, as was done in the past. This results in a reduction of cost and time of secure software delivery.
- **Enabling greater overall business success**. Increased revenue and expanded business offerings are the results of embracing new technologies and higher confidence in the security of developed software.

Static analysis, linters, and policy engines can be run any time a developer checks the code, software composition analysis be performed to verify that any open-source dependencies have compatible licenses and are free of vulnerabilities. Then, the code runs in an isolated container sandbox. It allows for automated testing of things such as network calls, input validation, and authorization. If something goes wrong, the tests fail, and the pipeline generates a report and notifies the teams concerned.

Once the first battery of tests is passed, the artifact is deployed to a wider sandbox, a limited copy of the eventual production environment. This is done with the purpose of verifying the application's behavior, by testing correct logging and access controls. Finally, the application is deployed into the production environment. To ensure that it is always running the most secure versions of software dependencies, automated patching, and configuration management is used.

Utilizing a DevSecOps CI/CD pipeline helps integrate security objectives at each phase without impacting the rapid delivery of business value to be maintained.

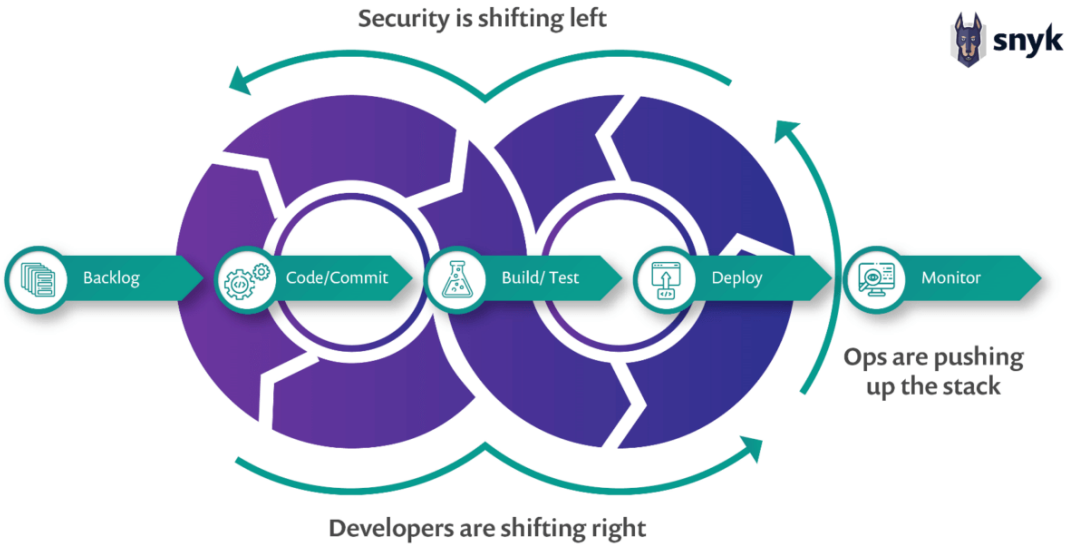

Figure 3.4. DevSecOps Adoption: Integrating Security into the CI/CD Pipeline [30].

3.2 Infrastructure as Code Security

As said before, DevOps and IaC have been adopted by more and more organizations due to the advantages offered by them. Also, DevSecOps has gained popularity since the growing concerns over information security and data privacy. Now the question is: how to ensure that security best practices and compliance requirements are built into the IaC template files? Ignoring this aspect could lead to severe consequences such as: exposure of sensitive data to unauthorized users, data leakage, unauthorized access to business-critical assets and resources, and increased attack surface [31].

Furthermore, Palo Alto Networks Unit 42 report states that: "199,000 potential vulnerabilities have been discovered in IaC templates. Also, more than 43% of cloud databases are currently unencrypted, and only 60% of cloud storage services have logging enabled which in itself is a serious concern" [32].

Snyk defines Infrastructure as Code Security (IaCSec) as *"the practice of securing cloud, infrastructure, and app configurations by scanning IaC files and the cloud deployment for compliance against a codified ruleset"* [33]. IaC-Sec aligns DevOps, Security, and Compliance for Infrastructure Management

processes. Treating infrastructure as code makes it possible to test for security and compliance of the IaC templates before they are deployed.

Applying IaCSec helps to reduce the drift of the security posture, and also can prevent security risks, non-compliance, policy violations, and misconfigurations before infrastructure deployment. Some aspects must be taken into account while implementing Infrastructure as Code security in an organization [31]:

- **IaC Templates**. Errors during the development of IaC templates could threaten the entire environment. Developers may use components with known vulnerabilities, insecure default configurations, and operating systems or container images from unknown sources.
- **Secrets**. In most cases, secrets are not managed properly. It has been observed that secrets such as authentication keys, passwords, access keys, SSH secrets, and access tokens are often hard-coded, left in plain text, or base63 encoded.
- **The Communication Channels**. Infrastructure as Code configuration management tools mostly rely on a master-node architecture. Since it contains all specifications and configuration files, if the master node and its communications with other nodes are not protected enough, the security risks are considerable.
- **User Access Management**. Credential sharing should be avoided. Also, the principle of least privilege and Role Based Access Controls (RBAC) must be implemented correctly otherwise security access management issues may arise.
- **Drifts in Configuration**. Configurations can be changed directly in the production environment, this practice is risky and leads to a configuration drift from the defined security posture.
- **Ghost Resources**. All resources must be correctly tagged. Otherwise, once created, they can not be properly monitored and tracked.
- **Risks related to Data Transmissions**. An improperly protected communication channel can significantly rise security risks. Securing data in the transmission is just as important as securing data at rest.
- **Audit Logs**. Infrastructural components which perform logging and monitoring must be included in the IaC templates. They are essential for monitoring as well as auditing the environment.

These aspects must be also considered during the design and development of security policies. The latter can be managed and enforced by following a Policy as Code approach, making it possible to take advantage of the benefits offered by this methodology. Finally, the Policy as Code tools for IaCSec can be integrated into the CI/CD pipelines used for infrastructure deployment. In this way, the IaC template files can be analyzed by them before the deployment. This allows the organization to maintain security and compliance over time by avoiding the deployment of non-compliant infrastructures in case of compliance issues reported by the tools [31].

3.3 Cloud Security Posture Management

When an organization moves to the cloud, it is often unaware of the shared responsibility model and thinks that the cloud provider is completely responsible for cloud security. However, all cloud breaches that we are aware of have been caused by cloud configuration errors that can be traced back to cloud customers and not cloud providers [34]. Therefore, they need a solution to detect and possibly avoid these issues automatically. A possible solution to the previous issues is applying Cloud Security Posture Management (CSPM) which, according to Snyk, consists of the automatic detection and mitigation of security and compliance risks across cloud infrastructure. Cloud consumers can use this solution in multiple use cases such as [34]:

- **Threat detection**. CSPM enables proactive threat detection, and centralized visibility of misconfigurations and suspicious activities so that organizations can assess and minimize risk exposure.
- **Incident response**. CSPM allows cloud consumers to detect indicators of compromise and mitigate any related threat.
- **Compliance**. CSPM can provide continuous compliance monitoring and reporting for regulations. In this way, organizations can avoid compliance violations and enforce internal security policies.
- **Securing infrastructure**. CSPM promotes the use of IaCSec to help organizations detect misconfigurations in infrastructure configuration files. CSPM also makes it possible to verify the infrastructure configuration when it is in operation.

Since CSPM promotes automation, a PaC approach can be followed to implement CSPM controls for a target cloud infrastructure, thereby automatically

inheriting all the benefits of PaC. PaC CSPM tools can be operated inside infrastructure deployment CI/CD pipelines for deploying those checks together with the infrastructure itself.

3.4 Cloud Workload Protection

The migration from legacy to cloud-native applications is not a trivial task. Cloud-native applications need additional processes and resources that support the applications and their interactions. The set of these extra components is called "workload" and allows cloud-native applications to behave properly. They should behave correctly as well to decrease any security risk introduced through their use. The process of keeping workloads that move across different cloud environments secure is known as Cloud Workload Protection (CWP) [35].

Cloud Workload Protection Platform (CWPP) solutions can discover workloads and perform a vulnerability assessment. In this way, they can identify any potentially exploitable security issues with the workload based on defined security policies and known vulnerabilities [36]. Organizations have various advantages by using them to secure their applications:

- **Agility**. CWPPs can be integrated into CI/CD pipelines to automatically secure applications developed using workloads.
- **Security**. CWPP solutions can be used by an organization to implement customized security controls for cloud workloads, protecting them accordingly from common security threats.
- **Compliance**. CWPPs, by implementing security controls to meet compliance requirements, can automatically scan for vulnerabilities and compliance violations.

A PaC approach can be followed to implement policies and controls in a way that makes it possible to automate security and compliance controls, as well as inherit all the other benefits this approach offers.

Chapter 4

Implementation of Policy as Code

The following chapter contains an overview of the several tools used for implementing a proof-of-concept of a system that implements Policy as Code to automate cloud compliance.

4.1 CI/CD pipeline: Jenkins

Jenkins is a self-contained, open-source automation server that can be used as a simple CI server or turned into a continuous delivery hub. This tool is written in Java and can be installed through native system packages, Docker, or simply run on any machine with a Java Runtime Environment (JRE) installed.

All sorts of tasks related to building, testing, and delivering or deploying software can be automated with Jenkins [37]. Finally, it is extensible via plugins and can be easily configured through its web interface. These characteristics make it an excellent candidate for implementing CI/CD pipelines. In fact, it is the favorite tool among DevOps developers holding 47.45% of the continuous integration software market [38].

Implementation and integration of continuous delivery pipelines into Jenkins are supported by a suite of plugins with the name Jenkins Pipeline. The definition of the pipeline is written via the groovy-based Pipeline domain-specific language (DSL) [39].

Jenkins Pipeline's code is used to define the build process, which typically includes stages for building, testing, and delivering an application. Pipelines

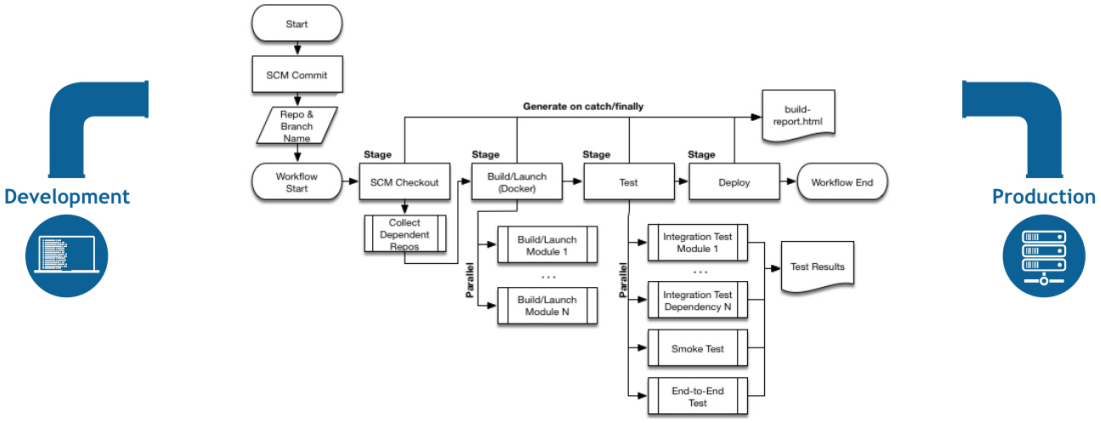

Figure 4.1. Example of one CD scenario modeled in Jenkins Pipeline [39].

can be defined via the web UI or inside a file called Jenkinsfile. Both ways use the same syntax, but the definition through the Jenkins file is the preferred one. That is because it can be put inside the repository which contains the source code and take advantage of these benefits:

- Automatic creation of a Pipeline build process for all branches and pull requests.
- Code review/iteration on the Pipeline (along with the remaining source code).
- Audit trail for the Pipeline.
- There is a single source of truth for the Pipeline, which can be viewed and edited by multiple members of the project.

Pipelines are writable with two different types of syntax: Declarative and Scripted (imperative).

- **Declarative Pipeline syntax**. All the work done throughout the entire pipeline is defined by the *pipeline* block. And the code contained inside that block defines the entire build process.
- **Scripted Pipeline syntax**. The core work throughout the entire pipeline is done by one or more *node* blocks.

Note that besides declarative and scripted pipelines are constructed differently, they have in common many of the individual syntactical components written into a Jenkinsfile. A possible example is the *stage* block, which is used for describing a stage of the pipeline. Each stage consists of a subset of steps performed by Jenkins through the entire Pipeline [39].

4.2 Infrastructure As Code: Terraform

HashiCorp Terraform is an Infrastructure as Code tool used for defining cloud and on-premise resources in human-readable configuration files. By using this tool, enterprises can take advantage of the various benefits inherited from an Infrastructure as Code approach.

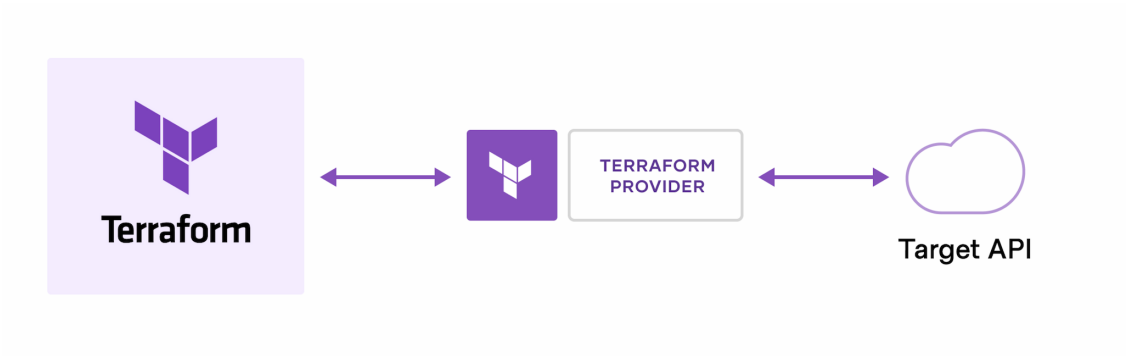

Figure 4.2. How Terraform interacts with the Target API [40].

Terraform creates and manages resources for cloud platforms and services through Providers. They can be defined as components that enable Terraform to work with virtually any platform or service with an accessible application programming interface. A lot of *providers* for various cloud providers such as Amazon Web Services [41], Microsoft Azure [42], and Google Cloud [43]. They have been written and made publicly available in the Terraform Registry [40].
As depicted by the figure 4.3 the core Terraform workflow consists of three stages:

- **Write**. Resources, which are part of the infrastructure, are described inside configuration files by using the Terraform language(HCL).

- **Plan**. The configuration files are provided to Terraform and used together with the existing infrastructure for creating a plan which describes the infrastructure that Terraform will create, update, or destroy.
- **Apply**. Once the plan is approved, Terraform performs the proposed operations in the correct order, respecting any resource dependencies.

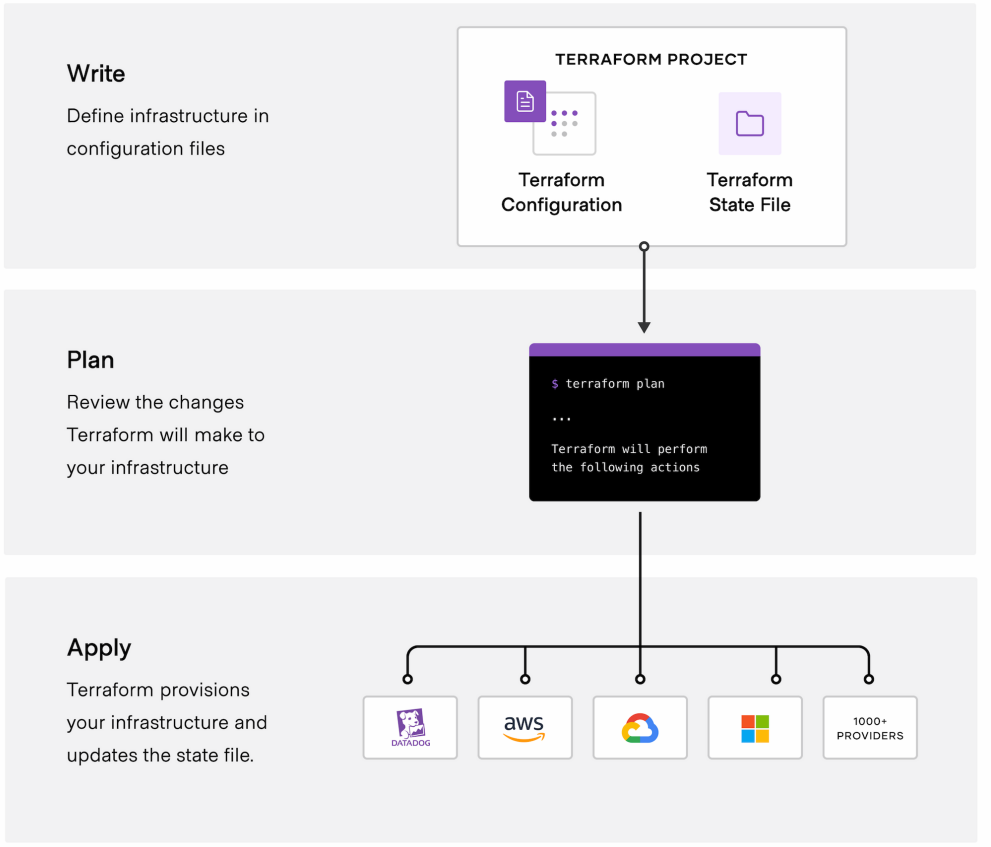

Figure 4.3. The Terraform workflow [40].

4.2.1 The Terraform language

The Terraform language is primarily used for declaring resources, which represent infrastructure objects, within configuration files. These files can be collected and organized in directories. This collection is called Terraform configuration and is used by Terraform for managing a given collection of infrastructure.

The syntax of the Terraform language consists of only a few basic elements as shown by the listing 4.1:

```
resource "aws_vpc" "main" {
   cidr_block = var.base_cidr_block
}
<BLOCK_TYPE> "<BLOCK_LABEL>" "<BLOCK_LABEL>" {
   <IDENTIFIER> = <EXPRESSION> # Argument
}
```

Listing 4.1. Basic Terraform syntax.

Blocks. They are usually used for representing the configuration of some kind of object. Blocks have a block type, can have zero or more labels, and have a body that contains any number of arguments and nested blocks.

Arguments. They are used for assigning a value to a particular name within blocks.

Expressions. They appear as values for arguments, or within other expressions. The value represented can be either literal or a combination of other values.

The Terraform language is declarative, and it requires only the description of the expected result. Then, the way the files are organized and the ordering of the blocks are not significant in general. As a result, while determining an order of operations, only implicit and explicit relationships between resources are considered by Terraform [44].

4.2.2 Terraform State

Terraform stores the state of the managed infrastructure and configuration. It is used in order to keep track of the mapping between objects in a remote system and resource instances declared in the configuration. This state can be stored either in a local file named “terraform.tfstate” or remotely.

Terraform state is essential for creating plans and making changes to the infrastructure. Therefore, Terraform performs a refresh to update the state with the actual infrastructure before performing any operation [45].

4.3 Policy as code: Open Policy Agent

The Open Policy Agent (OPA) is an open-source, general-purpose policy engine that can be used to enforce policies in microservices, Kubernetes, CI/CD pipelines, API gateways, and more [46].

OPA Policies are expressed via a declarative query language designed for defining queries over complex hierarchical data structures. This language was inspired by Datalog, a quite old query language, and it is called Rego. Since Rego queries are assertions on data stored in OPA, they can be used for enumerating instances of data that violate the expected state of the system. This feature explains why the engine uses Rego queries for making policy decisions [47].

When a software service needs to make this operation, it supplies structured data (e.g. JSON) as input and queries the engine. Then, policies and data are evaluated against the query input supplied by the service, and as a result, OPA generates a policy decision (the result of the query). Note that because those decisions are the result of a query, then they are not limited to simple "yes/no" or "allow/deny" answers [47].

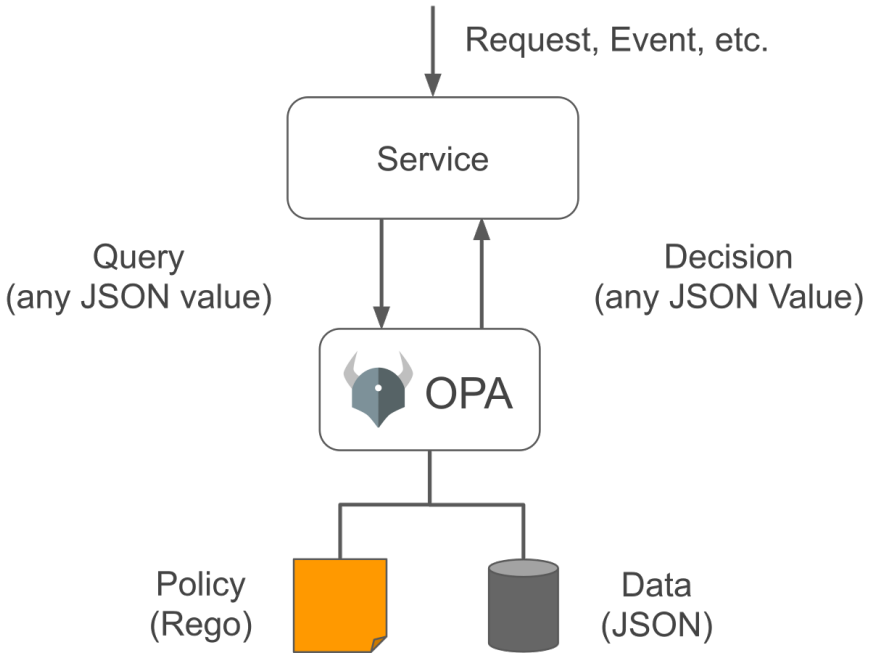

Figure 4.4. Interactions between services and OPA [46].

Since policies are written in machine-readable code, OPA is a very useful tool that can be used as a base for implementing a policy as code approach for an IT system. If OPA-like tools did not exist, an organization would have to implement policy management for its software from scratch. Finally, it is a quite flexible tool because both the engine and the policy language are domain-agnostic [46]. OPA-based tools took advantage of this feature by integrating the engine into them, although they have done so in different ways and for different purposes.

Unfortunately, there is no official policy pack, but often OPA-based tools offer their own packs which are often based on standards and best practices. In this way, they partially relieve organizations of the burden of developing policies from scratch.

4.3.1 The Rego query language

As said before, Rego [47] is a declarative query language used for writing OPA policies. Therefore, policy authors do not have to define how queries are executed, thus being able to focus only on what they should return.

OPA policies are expressed through Rego rules. These rules define the content of a document that is generated by OPA when it evaluates them. This document is the result of the query and represents the result of the policy evaluation. Rules can be defined in terms of both scalar and composite values.

Scalar values can be strings, numbers, booleans, or null. Documents can take advantage of this value type to define composite or constant values within queries. Composite values define collections. There are two main types of composite values in Rego: Objects and Sets. Objects are collections composed of unordered key-value pairs while Sets are collections that contain unordered unique values. Two scalar or composite values can be compared using operators such as "==" and "!=". The comparison is executed over the JSON representation of the value. Obviously, numeric values can be compared employing the other comparison operators such as ">", ">=", "<", and "<=".

Finally, documents can be embedded in other documents. Rego offers a way to access these nested documents through a mechanism called "reference". References are typically expressed via the "dot-access" style (e.g. `sites[0].servers[1].hostname`), but is also possible to use the "square-bracket" style (e.g. `sites[0]["servers"][1]["hostname"]`). Note that references can include variables as keys. By doing so, they can be used to select a

value from every element in a collection (e.g. `sites[i].servers[j].hostname`). The listing 4.2 some examples of rules that can be written in Rego using both scalar and composite values and some comparisons.

```
#rule defined in terms of scalar values
e := 2.71828

#rule defined in terms of composite values
rect := {"width": e, "height": 4}

#comparison between two objects
rect == {"width": 2.71828, "height": 4}

#comparison between two scalars
rect.width == e
```

Listing 4.2. Rego examples.

4.4 Infrastructure as Code Security: tfsec and Regula

Regarding Infrastructure as Code Security, two tools were chosen because they are not based on the same policy repository. Therefore, one tool may detect misconfiguration not detected by the other and vice versa.

4.4.1 tfsec

The first Infrastructure as Code Security tool analyzed is tfsec. It is an OPA-based open-source static analysis security scanner for Terraform code developed by Aqua Security [48]. This tool is built in Go and since it is OPA-based, it uses Rego as the language for defining policies. Finally, it also includes a library of predefined policies that contains rules for resources provided by major cloud providers.

The tool is operated via a command-line interface (CLI) and is designed to run locally or within a CI pipeline. Regardless of where it runs, is intended to analyze Terraform code and find misconfigurations within it before its deployment.

The input infrastructure configuration file can be checked for compliance with the policies written in its library [49] and, eventually, with those that are user-defined. In case security issues or misconfigurations are found, the tool generates an output containing various information about all the checks which have failed [48].

4.4.2 Regula

The second Infrastructure as Code Security tool analyzed is Regula. This open-source tool developed by Fugue evaluates Infrastructure as Code files for potential AWS, Azure, Google Cloud, and Kubernetes security and compliance violations before deployment.
Regula, differently from tfsec, supports the following file types:

- CloudFormation JSON/YAML templates;
- Terraform source code;
- Terraform JSON plans;
- Kubernetes YAML manifests;
- Azure Resource Manager (ARM) JSON templates.

Regula is written in Go and can work locally as well as inside CI/CD pipeline. Similarly to tfsec, Regula default checks come from its library. Although there is this similarity, Regula's library is based on the relevant parts of the CIS AWS, Azure, Google Cloud, and Kubernetes Foundations Benchmarks.

The tool reads Infrastructure as Code files passed as input via its command-line interface and it uses OPA to evaluate them against Regula's library of rules and custom ones, generating a report as output [50].

4.5 Cloud Security Posture Management: Cloud Custodian

Cloud Custodian is an open-source tool written in Python that organizations can use for managing their public cloud accounts for ensuring compliance with security policies, tag policies, garbage collection of unused resources, and cost management for their cloud environments. Policies are written by using a YAML-based DSL. In this way, they can be validated, dry-run, and reviewed. Cloud Custodian is compatible with the major cloud providers and uses a stateless rules engine for policy definition and enforcement. Furthermore, it is tightly integrated with serverless runtimes, and therefore it can be bound to serverless event streams across multiple cloud providers. Last but not least, relevant information such as metrics, structured outputs, and detailed reporting for cloud infrastructure, are also generated by the tool [52]. They can be stored locally as well as in public cloud storage containers.

4.6 Cloud Workload Protection: Gatekeeper

Gatekeeper is an open-source project that provides integration between OPA and Kubernetes. Through the use of admission controller webhooks, which are HTTP callbacks triggered anytime a resource is created, changed, or destroyed. Kubernetes enables the decoupling of policy decisions from the inner workings of the API Server. Mutating webhooks are called by the mutating admission controller and can patch objects contained inside the admission request, on the contrary validating webhooks, are called by the validating admission controller and can only reject admission requests. More precisely, Gatekeeper is a validating and mutating webhook that enforces Custom Resource Definition based policies (CRD-based policies) executed by the OPA policy engine.

Finally, Gatekeeper also provides audit functionality, which allows administrators to see what resources are currently violating any given policy, and allows them to detect and reject the deployment of non-compliant Kubernetes resources [51].

4.7 How to write custom policies

4.7.1 tfsec

As said previously, tfsec is able to apply user-defined Rego policies. To enforce them, the `--rego-policy-dir` flag must be utilized for specifying the directory containing the custom controls. By doing so, policies are loaded recursively starting from the specified directory, and the input infrastructure configuration will be checked against them as well. Note that, if the `--rego-policy-dir` flag is not specified then no local directories will be scanned.

This is a very helpful feature for all organizations that need to implement custom security policies along with those specified by the library.

```
1 package custom.aws.s3.no_insecure_buckets
2 import data.lib.result
3
4 deny[res] {
5     bucket := input.aws.s3.buckets[_]
6     bucket.name.value == "insecure-bucket"
7     msg := "Bucket name should not be 'insecure-bucket'"
8     res := result.new(msg, bucket.name)
9 }
```

Listing 4.3. tfsec user-defined policy example.

This code represents a policy whereby an s3 bucket can not be named as "insecure-bucket". As shown in the example, the package name can take any value, but it must always begin with the "custom" namespace, and the rule name must either be "deny" or start with "deny_". Also, the input variable contains cloud resources organized by providers (e.g AWS) and services (e.g. s3).

Note, the policy instead of checking the "bucket.name," property checks the "bucket.name.value". That happens because the "bucket.name" property also contains various metadata about where its value was defined.

Finally, the "result.new()" function is used for creating the object that will be returned. The function takes two parameters, the *msg* parameter is a string used to explain the detected issue, while the *source* parameter is the object or property where the problem occurred [53].

4.7.2 Regula

Analogously to tfsec, also Regula can enforce user-defined policies. However, Regula custom rule management is more structured because it defines two types of custom rules:

- Simple rules
- Advanced rules

Simple rules are used when the policy applies to a single resource type only, and it makes a simple allow/deny decision. Simple rules can be augmented by defining a custom error message within them. This is done by writing the rule in this way:

```
1 package rules.my_simple_rule
2
3 resource_type = "aws_ebs_volume"
4
5 deny[msg] {
6     not input.encrypted
7     msg = "EBS volumes should be encrypted"
8 }
```

Listing 4.4. Regula simple rule example.

Simple rules must specify "allow" or "deny" rego rules. For this example, this "deny" rule has been used for checking that the EBS volume is encrypted. Note that the rules package name can take any value, but it must always

begin with the "rules" namespace. Also, with this type of rule, the target resource type should be indicated as well. Finally, the "msg" variable contains the custom error message.

Advanced rules are more powerful, but also more complex to write. Unlike simple rules, they allow to observe different kinds of resource types and decide which specific resources are valid or invalid. Also, these rules can leverage additional functions which come from the Fugue API library [54].

```
package rules.user_attached_policy

import data.fugue

resource_type = "MULTIPLE"
ebs_volumes = fugue.resources("aws_ebs_volume")

is_encrypted(resource) {
    resource.encrypted == true
}

policy[p] {
    resource = ebs_volumes[_]
    is_encrypted(resource)
    p = fugue.allow_resource(resource)
} {
  resource = ebs_volumes[_]
  not is_encrypted(resource)
  p = fugue.deny_resource(resource)
}
```

Listing 4.5. Regula rule metadata example.

The structure depicted in the listing 4.5 is pretty similar to the simple rules one but with some differences. As an example, instead of specifying a deny or allow rego rule advanced rules must specify a rule called "policy" as the main rule. Another difference concerns the "resource_type" variable which must contain the "MULTIPLE" value. Furthermore, the resources that should be analyzed should be requested by using the "fugue.resources(resource_type)" function [54].

Finally, metadata can be added to a rule to enhance Regula's report as shown in the listing 4.6.

```
__rego__metadoc__ := {
    "id": "CUSTOM_0001",
    "title": "IAM policies must have a description of at
   least 25 characters",
```

```
4     "description": "Per company policy, it is required for
      all IAM policies to have a description of at least 25
      characters.",
5     "custom": {
6     "controls": {
7       "CORPORATE-POLICY": [
8         "CORPORATE-POLICY_1.1"
9       ]
10    },
11    "severity": "Low",
12    "rule_remediation_doc": "https://example.com"
13  }
14 }
```

Listing 4.6. Regula rule metadata example.

4.7.3 Cloud custodian

In contrast to all the other tools analyzed so far, Cloud Custodian is not based on OPA. This means that policies are not written using the Rego language. Instead, the tool uses a YAML-based DSL for defining controls. Unfortunately, the tool does not own a default controls library, therefore policies must be manually defined by the end user.

As an example, the policy shown by the listing 4.7 will automatically stop all EC2 instances that are tagged with the key "Custodian".

```
1 policies:
2   - name: my-first-policy
3     mode:
4         type: pull
5     resource: aws.ec2
6     filters:
7       - "tag:Custodian": present
8     actions:
9       - stop
```

Listing 4.7. Cloud Custodian policy example.

Cloud Custodian policies can refer to different types of resources. Depending on that is possible to use specific filters to narrow down the set of resources or specific actions to be taken on the filtered resources. The resource type is specified by the `resource` property. Conversely, filters and actions are indicated by their respective fields. Finally, it is also possible to define multiple actions and multiple filters can be specified in the same policy [57].

Policy enforcement can be triggered in different ways which are defined by using the *mode* property. The specific triggers depend on the target cloud provider, however, it is possible to group them into three main categories:

- Pull
- Periodic
- Event-Based

With the "Pull" execution mode, policy compliance checks, and the eventual actions, are triggered by using the CLI. Whereas, those marked with the "Periodic" execution modes are executed periodically with a period defined within the policy itself. Finally, policies defined using the "Event-Based" execution mode are enforced only when an event specified within the policy happens.

4.7.4 OPA Gatekeeper

Gatekeeper uses the OPA Constraint Framework [55] which leverages the concepts of Constraint and ConstraintTemplate to describe and enforce policies [56]. ConstraintTemplate resources are used for describing both the Rego that enforces the constraint and the schema of the constraint. Constraint resources are an instantiation of a ConstraintTemplate. Their purpose is to inform Gatekeeper that a ConstraintTemplate must be enforced, and how.

The listing 4.8 represents an example of the structure of a ConstraintTemplate that, analogously to other Kubernetes resources, can be defined by using YAML.

```
1  apiVersion: templates.gatekeeper.sh/v1beta1
2  kind: ConstraintTemplate
3  metadata:
4    name: k8srequiredlabels
5  spec:
6    crd:
7      spec:
8        names:
9          kind: K8sRequiredLabels
10       validation:
11         # Schema for the `parameters` field
12         openAPIV3Schema:
13           properties:
14             labels:
15               type: array
16               items:
17                 type: string
```

```
18  targets:
19    - target: admission.k8s.gatekeeper.sh
20      rego: |
21        package k8srequiredlabels
22        violation[{"msg": msg,
23            "details": {"missing_labels": missing}}] {
24          provided := {label |
25            input.review.object.metadata.labels[label]}
26          required := {label |
27            label := input.parameters.labels[_]}
28          missing := required - provided
29          count(missing) > 0
30          msg := sprintf("you must provide labels: %v",
31            [missing])
32        }
```

Listing 4.8. ConstraintTemplate example.

This example represents a policy used for enforcing the presence of a specific set of labels on an arbitrary object. The labels and the target resources will be then specified by the Constraint resource. By doing so, the ConstraintTemplate definition can be reused since it is independent of the target labels and resources.

```
1  apiVersion: constraints.gatekeeper.sh/v1beta1
2  kind: K8sRequiredLabels
3  metadata:
4    name: ns-must-have-gk
5  spec:
6    match:
7      kinds:
8        - apiGroups: [""]
9          kinds: ["Namespace"]
10   parameters:
11     labels: ["gatekeeper"]
```

Listing 4.9. Constraint example.

The listing 4.9 represents an example of the possible constraints that can be created based on the previous ConstraintTemplate. To indicate that the Cosntarint shown in the example is an instance of the ConstraintTemplate defined previously, the “spec.crd.spec.names.kind” property of the ConstraintTemplate and the “kind” property of the Constraint must contain the same value. In addition to that, the “parameters” property must respect the openAPIV3Schema specified in the ConstraintTemplate.

In this example, the property specifies which labels must be present inside the target resources. The values specified inside the “parameters” property can be accessed inside the Rego code by using the parameters field of the input object. Finally is also required to specify which is the scope of objects

that have to respect the Constraint. Those are specified inside the "match" property. The Rego code can access those objects as well by using the review field of the input object [56].

4.8 Strategy and methodology used to implement cloud compliance automation

A common characteristic of all the tools discussed so far is that each of them exposes a command line interface (CLI). Since Jenkins pipelines can execute shell scripts, this feature is crucial for integrating them into the CI/CD pipelines developed for automating cloud compliance verification.

These tools were chosen not only because they are open-source, but also because they allow policies to be written declaratively. In this way, controls can be defined with a higher level of abstraction since it is necessary to describe what they are supposed to do rather than how.

The tools mentioned before are utilized via three Jenkins CI/CD pipelines to demonstrate how Policy as Code can be employed for automating cloud compliance verification. For this purpose, the approach is applied to a sample cloud infrastructure, defined with Terraform, that will run a Kubernetes [58] cluster.
The three pipelines are executed according to the following order:

1. The first pipeline performs the deployment of the infrastructure only if the IaCSec compliance verification performed by tfsec and Regula is successful.

2. The second pipeline is responsible for deploying periodic and event-based CSPM controls using Cloud Custodian.

3. The last Jenkins pipeline deploys a sample application together with the related CWP controls and OPA Gatekeeper into the Kubernetes cluster.

Although the first two operations can be executed in any order, this order was defined since it seems to be simple to understand. However, note that regardless of the order chosen, the deployment of the example application and CWP controls must necessarily be performed after the infrastructure deployment. Otherwise, there will be no cluster on which to perform the deployment. The sample infrastructure has been developed to simulate a production-like environment, hence the tools utilized inside these pipelines can be used in this kind of environment.

Chapter 5

Policy as Code: Case of study

This chapter contains an explanation of a possible implementation of the Policy as Code methodology during the development and operation of a Kubernetes cluster inside a cloud infrastructure. To accomplish that, a proof-of-concept (PoC) was developed to explain how the methodology can be integrated into the software development life-cycle.
The PoC is composed by:

- IaC files that describe the target infrastructure and IaCSec policy files are stored within a remote version control repository;
- CSPM policy files stored inside another remote version control repository;
- an additional version control repository is used for storing CWPP policy files and other files needed for deploying the sample application and the CWPP tool;
- a CI/CD pipeline that is used for checking the infrastructure against the IaCSec tools and deploying the infrastructure;
- a CI/CD pipeline whose task is to deploy the CSPM policies;
- a CI/CD pipeline which is in charge of deploying OPA Gatekeeper, the related policies, and a sample application inside a Kubernetes cluster.

5.1 Architectural overview of the target cloud infrastructure

The target infrastructure was developed by following an IaC approach by using Terraform. It has been deployed inside an Amazon Web Services [41] (AWS) environment to emulate a possible cloud infrastructure development use case. This platform was chosen because AWS is currently the most desired cloud platform among developers [59].

The infrastructure's first version was developed using the default configurations of the related Terraform resources. This was done to demonstrate that their default configuration is not always secure by default. Indeed, the IaCSec tools used to scan the infrastructure reported several failed checks during the first scan.

Therefore, another version of the infrastructure was developed taking into account the feedback provided by those tools.

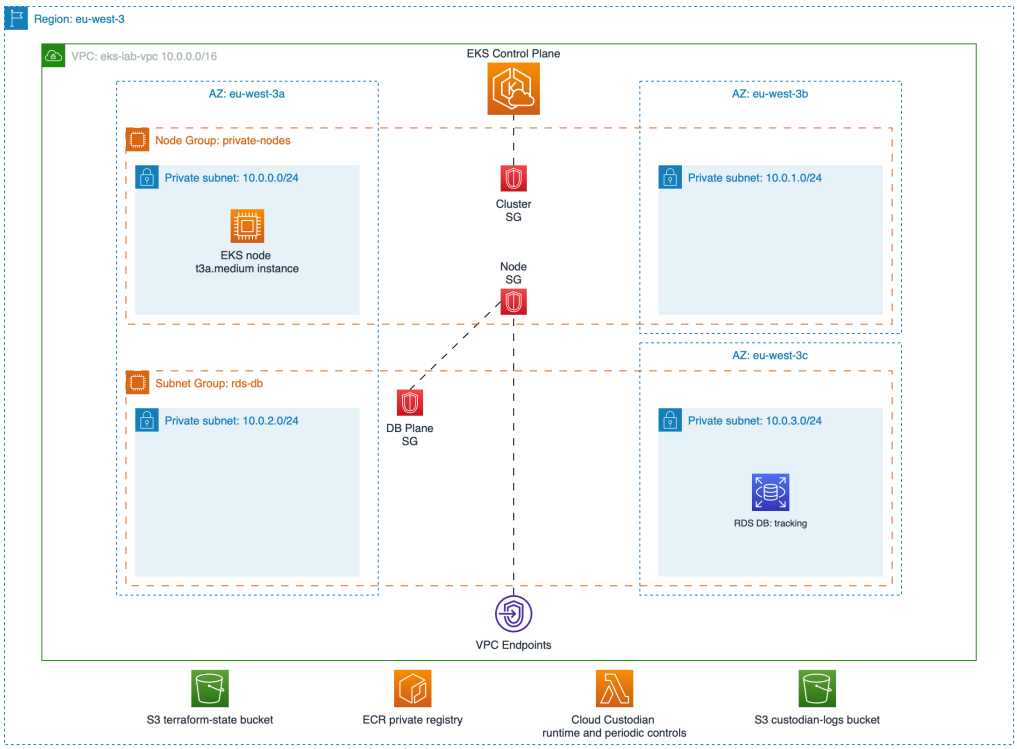

Figure 5.1. The target infrastructure

The target infrastructure aims to simulate a production-like environment. To accomplish this, several components and their characteristics were used. The infrastructure is based on the following services offered by AWS [60]:

- Amazon Virtual Private Cloud (Amazon VPC)
- Amazon Elastic Kubernetes Service (Amazon EKS)
- Amazon Simple Storage Service (Amazon S3)
- Amazon Relational Database Service (Amazon RDS)
- Amazon Elastic Container Registry (Amazon ECR)
- Amazon Elastic Compute Cloud (Amazon EC2)

A VPC is a virtual network that resembles a traditional network operated within a data center. Therefore, it is possible to assign a range of IP addresses to the entire VPC (both IPv4 and IPv6 addressing are supported) and further subdivide this range to create subnets.

Once created, AWS resources such as EC2 and RDS instances can be deployed within it. The developed infrastructure relies on this concept for the implementation of an EKS cluster. The latter is used to emulate a production environment in which a simple containerized application runs.

An EKS cluster consists of a Kubernetes cluster in which control plane or nodes are directly managed by AWS rather than by the customer.

In this case, the worker node role for the cluster is assumed by EC2 instances inside the VPC. Each instance is a virtual computing environment offered by AWS, and various types of configurations can be chosen, which differ in CPU, memory, storage, and network capacity. The images of the containers that will run inside the cluster are stored inside a private repository inside the Amazon ECR.

Finally, is important to note that the cluster should be deployed inside a VPC which spans over at least two Availability Zones. This requirement was defined to improve the overall availability of the cluster since Availability Zones consist of one or more discrete data centers hosted in separate facilities [61].

AWS allows instantiating DB instances within the VPC by using Amazon RDS. Also, in this case, the VPC must span over at least two Availability Zones. A DB instance represents an isolated database environment running in the cloud. Just like EC2 resources, each DB instance belongs to a class that determines its computational and memory capacity. In this case, a

simple instance is used for running a PostgreSQL DBMS inside the VPC of the target infrastructure. Note that the DB instance does not run inside the EKS cluster. However, it can be used by the sample application for saving and reading data.

Finally, the last main components of the target infrastructure are two S3 buckets. One was created via the AWS Management Console. This was done because it is used for storing the Terraform state file, hence the creation cannot be done via Terraform. The other one, is a preexisting bucket utilized for storing the logs produced by Cloud Custodian's controls.

5.2 AWS Terraform resources

As mentioned previously, the target infrastructure was developed by following an IaC approach by using Terraform. To speed up the development, the VPC, the VPC endpoints, and the EKS cluster were defined using the relative Terraform modules [62]. A Terraform module is a container for multiple resources that are used together and can be called in the configuration code by using the module block. This block is used by compiling its input values to configure the resources contained inside the module.

The last operation is done to include modules' contents within the configuration itself. The remaining components such as security groups and availability zones were defined in a classical way using Resources and Data Sources respectively.

Resource blocks are used for declaring the configuration of a particular infrastructure object [63], while data sources allow Terraform to query information defined by an external source [64].

5.2.1 Variables

Input Variables have been defined to allow the reuse of frequently requested values in the code. They work like function arguments and can be used across all the Terraform configuration files. Each input variable is defined via a "variable" block. Each block has a name, which is the label after the variable keyword. Furthermore, variable blocks can be augmented by adding a default value and a type that specifies what value types are accepted for the variable [65]. The listing 5.1 illustrates the variables used inside the configuration files which define the target infrastructure.

```
variable "region" {
  type    = string
  default = "eu-west-3"
}

variable "vpc_name" {
  type    = string
  default = "eks-lab-vpc"
}

variable "eks_cluster_name" {
  type    = string
  default = "eks-lab-cluster-module"
}

variable "private_subnets_num" {
  type    = number
  default = 2
}

variable "db_subnets_num" {
  type    = number
  default = 2
}
```

Listing 5.1. The Terraform variables used inside the infrastructure configuration files.

5.2.2 VPC Module

The Amazon VPC is created using the "vpc" module [66] as shown in the listing 5.2.

As stated before, each VPC must have an IP address range, hence the IPv4 Classless Inter-domain Routing (CIDR) block assigned for the VPC of the developed architecture, is defined inside the `cidr_block` variable. Furthermore, it has been further subdivided to create `subnets_num` private subnets used for deploying the worker nodes of the EKS cluster and `db_subnets_num` subnets used for deploying the DB instances. Each subnet IPv4 CIDR is generated by the `cidrsubnet` function starting from the IPv4 CIDR block assigned for the VPC. The lists for the private subnets and the DB subnets are generated by taking advantage of the list comprehension functionality offered by the Terraform language. The definition of the availability zones where the VPC spans are contained inside the `azs` variable. Finally, there is the definition of the tags needed by the EKS cluster to work properly [67].

```
module "vpc" {
  source  = "registry.terraform.io/terraform-aws-modules/vpc/aws"
  version = "3.18.1"

  cidr = var.cidr_block
  azs  = data.aws_availability_zones.avzs.names

  enable_dns_support   = true
  enable_dns_hostnames = true

  private_subnets      = [for i in range(var.private_subnets_num) : cidrsubnet(var.cidr_block, 8, i)]
  private_subnet_names = [for i in range(var.private_subnets_num) : "private-subnet-${i}"]
  private_subnet_tags = {
    "kubernetes.io/cluster/${var.eks_cluster_name}" = "owned"
  }

  database_subnets           = [for i in range(var.db_subnets_num) : cidrsubnet(var.cidr_block, 8, var.private_subnets_num + i)]
  database_subnet_names      = [for i in range(var.db_subnets_num) : "db-subnet-${i}"]
  database_subnet_group_name = "rds-db"

  vpc_tags = {
    Name = var.vpc_name
    "kubernetes.io/cluster/${var.eks_cluster_name}" = "owned"
  }

  tags = {
    LAB   = "tesi_mattia"
    infra = "terraform"
  }
}
```

Listing 5.2. The VPC module block.

5.2.3 EKS Module

The Amazon EKS cluster is created using the "eks" module [68] as shown in the listing 5.3.

Each EKS cluster is identified by its name, therefore the `cluster_name` property is used for defining it inside the module block. For the sample infrastructure, the name of the cluster is contained inside the `eks_cluster_name` variable and assigned to the `cluster_name` property. Furthermore, the VPC in which the cluster will be deployed must be specified as well, and in this

case, the ID of the VPC previously defined is extracted from its module and entered into the `vpc_id` field. The same thing happens with the list of worker nodes subnet IDs, also this value is extracted from the relative VPC module and entered into the `subnet_ids` field. The module creates also three security groups needed for controlling the traffic that is allowed to reach and leave the resources that they are associated with [69]. In this case, there are two groups of resources: the control-plane and the data-plane (worker nodes).

The security groups are created by the module for controlling the traffic between:

- control-plane and data-plane
- data-plane and data-plane
- control-plane and control-plane

Finally, the field `iam_role_name` specifies the name of the IAM role [70] created to provide permissions for the Kubernetes control plane to make calls to AWS API operations.

```
module "eks" {
  source  = "registry.terraform.io/terraform-aws-modules/eks/aws"
  version = "19.0.4"

  cluster_name = var.eks_cluster_name

  iam_role_name = "${var.eks_cluster_name}-cluster"
  iam_role_tags = {
    Name = "${var.eks_cluster_name}-cluster"
  }

  vpc_id     = module.vpc.vpc_id
  subnet_ids = module.vpc.private_subnets

  eks_managed_node_groups = {
    private-nodes = {
      create         = true
      capacity_type  = "ON_DEMAND"
      instance_types = ["t3a.medium"]
      desired_size   = 2
      max_size       = 4
      min_size       = 1
      iam_role_additional_policies = {
        "arn:aws:iam::aws:policy/AmazonSSMManagedInstanceCore" = "arn:aws:iam::aws:policy/AmazonSSMManagedInstanceCore"
      }
    }
  }

```

```
29   tags = {
30     LAB   = "tesi_mattia"
31     infra = "terraform"
32   }
33 }
```

Listing 5.3. The EKS module block.

Finally, this cluster uses managed node groups to automatically provision or register the EC2 instances utilized as worker nodes. The parameters used for configuring the node group are directly specified as values for the `private-nodes` key. Note that EC2 instances are also configured for being contacted through the Amazon Session Manager [71].

5.2.4 VPC endpoints module

Since the target infrastructure's VPC is composed only of private subnets there must be a way to connect the EC2 instances with the other services offered by AWS. This is done by using AWS PrivateLink service, which enables to privately connect VPC to external services as if they were inside the VPC itself. This service provides two different types of endpoint [72]:

- **Interface Endpoint**. It is composed of a collection of one or more elastic network interfaces (ENIs) with a private IP address that serves as an entry point for traffic destined for a supported service.
- **Gateway Endpoint**. It targets specific IP routes in an Amazon VPC route table, in the form of a prefix-list, used for traffic destined for Amazon S3.

These components are necessary to enable the creation of the EKS cluster, as it requires some AWS-managed services to be properly configured. The listing 5.4 shows how to define the two types of the endpoint with the Terraform "vpc_vpc-endpoints" module [73].

```
1 module "vpc_vpc-endpoints" {
2   source  = "registry.terraform.io/terraform-aws-modules/vpc/
     aws//modules/vpc-endpoints"
3   version = "3.18.1"
4
5   vpc_id             = module.vpc.vpc_id
6   security_group_ids = [aws_security_group.
     node_security_group_id]
7   subnet_ids         = module.vpc.private_subnets
8
```

```
  endpoints = {
    s3 = {
      service         = "s3"
      route_table_ids = module.vpc.private_route_table_ids
      service_type    = "Gateway"
      tags            = { Name = "s3" }
    },
    ec2 = {
      service             = "ec2"
      private_dns_enabled = true
      tags                = { Name = "ec2" }
    },
    ...
  }

  tags = {
    LAB   = "tesi_mattia"
    infra = "terraform"
  }
}
```

Listing 5.4. VPC endpoints module.

The `vpc_id` property contains the ID of the VPC in which the endpoints will be deployed, while the `subnet_ids` one contains the default list of subnet IDs that will be associated with the interface endpoints. The `security_group_ids` field is used to specify a list containing the default security groups that will be associated with the interface endpoints.

Finally, the complete definition of the module block contains also the following endpoints: `sts`, `ecr_api`, `ecr_dkr`, `logs`, `ssmmessages`, `ec2messages`, and `ssm`. The first three, along with those defined in the listing, are required to properly configure the EKS cluster. The `logs` endpoint provides the ability to send logs to the CloudWatch Logs service [74] in order to centralize the log management. Finally, the others must be enabled to use the AWS Systems Manager Session Manager [71] in case it is necessary to manage the EC2 instances created by the cluster.

5.2.5 RDS DB instance resource

The RDS DB instance is defined as shown by the listing 5.5 by using the "aws_db_instance" resource. It permits to define the name of the database via the `db_name` field. The resource block allows also to define the allocated storage for the instance and its class via the `allocated_storage` and `instance_class` fields respectively. As regards networking, the field

`db_subnet_group_name` contains the name of the group of subnets utilized for instantiating the DB instances, while the `vpc_security_group_ids` property contains the security groups that will be associated with the DB instances' network interfaces. Finally, there is the definition of the database's master username and password which were defined in plain text on purpose [75].

```
resource "aws_db_instance" "rds_db" {
  db_name                = "tracking"
  allocated_storage      = 20
  engine                 = "postgres"
  instance_class         = "db.t3.micro"
  username               = "mattia"
  password               = "terraform"
  db_subnet_group_name   = module.vpc.
   database_subnet_group_name
  vpc_security_group_ids = [aws_security_group.db_plane_sg.id
   ]
  tags = {
    LAB     = "tesi_mattia"
    infra   = "terraform"
    db_name = "rds_db"
  }
}
```

Listing 5.5. RDS instance resource.

5.2.6 Security groups

As mentioned before, security groups are used for controlling the traffic among the resources that they are associated with. Each security group allows to specify inbound or outbound rules based on protocols, port numbers, CIDR blocks, or other security group IDs [69]. To make worker nodes able to interact with the DB instances and the VPC interface endpoints two security groups are created. Both of them are shown by the listing 5.6 along with the corresponding security group rules.

```
resource "aws_security_group" "endpoints" {
  name        = "endpoints-ingress"
  vpc_id      = module.vpc.vpc_id
  description = "Security group for interface endpoints"

  tags = {
    Name  = "endpoints-ingress"
    LAB   = "tesi_mattia"
    infra = "terraform"
```

```
10   }
11 }
12 resource "aws_security_group_rule" "endpoint-ingress" {
13   description              = "Endpoint ingress rule"
14   type                     = "ingress"
15   security_group_id        = aws_security_group.endpoints.id
16   from_port                = 1025
17   to_port                  = 65535
18   protocol                 = -1
19   source_security_group_id = module.eks.
      node_security_group_id
20 }
21
22 resource "aws_security_group" "db_plane_sg" {
23   name        = "db-plane-sg"
24   vpc_id      = module.vpc.vpc_id
25   description = "Security group for dbs"
26
27   tags = {
28     Name  = "db-plane-sg"
29     LAB   = "tesi_mattia"
30     infra = "terraform"
31   }
32 }
33 resource "aws_security_group_rule" "node_ingress" {
34   description              = "DB ingress rule"
35   type                     = "ingress"
36   security_group_id        = aws_security_group.db_plane_sg.
      id
37   from_port                = 5432
38   to_port                  = 5432
39   protocol                 = "tcp"
40   source_security_group_id = module.eks.
      node_security_group_id
41 }
```

Listing 5.6. Security group resources.

Security groups are defined using an `aws_security_group` [76] resource. Each one has a name used as an identifier and contained inside the `name` property, and the VPC in which the security group will be utilized is specified inside the `vpc_id` field.

The rules of each group are specified inside the `aws_security_group_rule` resources [77], while the `security_group_id` allows referring the rule to its security group. The `type` property can assume either the value *ingress* or *egress* and it defines the type of the rule. Finally, the properties, `from_port`, `to_port`, `protocol`, and `source_security_group_id` are needed to control the incoming or outgoing traffic of resources that belong to the security group.

The first security group contains only one rule. The latter allows all traffic

from worker nodes not destined for a well-known port of a resource that is part of the security group (VPC interface endpoints). Similarly, the rule defined for the security group for DB instances allows only the traffic from worker nodes that is directed to port 5432 of the resources which belong to the security group.

5.3 Set up of the infrastructure in AWS

The first thing to do for deploying the target infrastructure consists in defining a backend block [78] and a provider block [79]. A backend defines where Terraform stores its state data files, while a provider is a plugin that allows Terraform to interact with cloud providers, SaaS providers, and other APIs.

By default, the state file is stored locally, but as shown in the listing 5.7 in this case it is stored inside an S3 Bucket.

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.44"
    }
  }
  backend "s3" {
    bucket  = "tesi-mattia-terraform-state-bucket"
    key     = "terraform/mattia-tesi-eks-module/tfstate.json"
    region  = "eu-west-3"
    encrypt = true
  }
}
```

Listing 5.7. Terraform backend.

The effective deployment of the target infrastructure is performed via the Terraform CLI [80]. The procedure starts with the initialization of the working directory. This directory contains the Terraform configuration files and other files used for provisioning the infrastructure such as modules and plugins. It must be initialized before performing any other operation, and this initialization is done by invoking the `terraform init` [81] command inside the working directory. The invocation of the command starts with the initialization of the chosen backend, as well as downloading and installing the defined modules and plugins.

Once the initialization phase is terminated, the `terraform plan` [82] command can be invoked for defining an execution plan. This plan is produced

by analyzing the difference between the current state of any already-existing remote objects and the desired state. The plan presents a description of the changes necessary to achieve the desired state and can be saved as an artifact. The latter can be used for provisioning the infrastructure.

The provisioning of the infrastructure is done by executing the `terraform apply` [82] command and passing the plan file, then the command executes the actions provisioned by the plan. Note that if a plan file is not passed, the command generates one and uses it for provisioning the infrastructure.

Finally, Terraform also permits the destruction of all remote objects managed by a Terraform configuration by executing the `terraform destroy` [82] command inside the working directory of the configuration.

5.4 Policy code management

Following a Policy as Code approach implies that policies can be treated in the same way as normal source code. This allows policy code to be stored and managed within a remote version control repository, in this case, GitHub. Furthermore, CI/CD pipelines can be executed over those repositories in order to enforce policies and eventually deploy the infrastructure and the sample application.

5.4.1 IaCSec policies

IaCSec policies are stored inside the same repository that contains the infrastructure configuration files for simplifying the infrastructure code scanning phase. Regula and tfsec make it possible to apply custom policies in addition to those contained in their library. Those policies must be specified inside folders dedicated to this purpose. For this reason, the tfsec additional policies are contained inside a folder called *.tfsec_rules* and analogously the Regula custom policies are stored in the *.regula_rules* folder.

5.4.2 CSPM policies

A dedicated repository is reserved for the CSPM policies code. This repository contains a folder called *pack*. This folder is the place where the policies are stored. The *pack* folder is further subdivided into other folders since CSPM policies are grouped by execution mode i.e. there is a folder for each group. Pull execution mode-based policies are contained inside the *dry-run*

folder, Event-Based policies are stored inside the *event-based* folder, and finally, Periodic policies are defined inside a folder called *periodic*.

5.4.3 CWPP policies

CWPP policies code and the files needed for deploying the sample application and OPA Gatekeeper are stored inside the same repository. That simplifies the deployment procedure of the Kubernetes objects into the cluster contained inside the target infrastructure.

CWPP policies are defined inside a sub-folder of the `policies` folder. Each sub-folder contains the definition of the Kubernetes objects that encode the policy and a *kustomization.yaml* file required by Kustomize [85] for producing two main YAML files. One contains all constraint definitions, the other contains all defined constraint templates. These two files are deployed inside the cluster for enforcing the relative policies.

5.5 Compliance Automation and how it is implemented

The implementation of the Policy as Code approach for compliance automation is done by defining three CI/CD pipelines using Jenkins.

- The first one is in charge of scanning the target infrastructure using Regula and tfsec, deploying it into the cloud environment if no errors occur.

- The second pipeline is used to deploy periodic and event-based Cloud Custodian policies into the cloud environment and to obtain a report on compliance violations against dry-run, event-based and periodic policies.

- The third one permits the deployment of Gatekeeper and the relative policies together with the sample application. Also, it allows obtaining a compliance violation report against the defined policies.

All the defined pipelines contain a section that is always executed which is used for saving the output produced by the tools.

5.5.1 Set up of the pipeline for IaC and IaCSec

The listing 5.8 contains the definition of the pipeline for IaC and IaCSec. Since it is a declarative Jenkins pipeline, it is composed of a `pipeline` block containing the various phases of the pipeline.

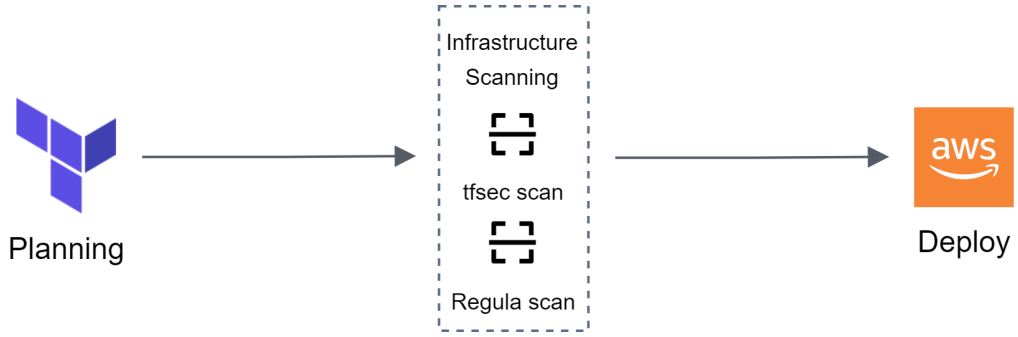

Figure 5.2. Stages of the pipeline for IaC and IaCSec.

The "Planning" stage initializes the Terraform working directory and produces the Terraform plan. This plan is further converted into JSON format by the `terraform show` command in order to be checked by Regula. Note that two Terraform operations are encapsulated inside a `withCredentials` block. This block is offered by the Credentials Binding plugin [83] and has the purpose of binding credentials [84] with variables. In this way, the credentials are not written in plain text in the pipeline, but retrieved directly from Jenkins where they are stored in an encrypted form. These variables are used by Terraform commands for authenticating with AWS. The `terraform init` command needs these credentials for initializing the S3 backend, while the `terraform plan` command requires them since it has to read the Terraform state for defining the execution plan.

Once the "Planning" phase is over, the "Infrastructure Scanning" phase begins. During this phase, the target infrastructure is scanned against tfsec and Regula. The `parallel` block allows performing the scanning with both tools concurrently. If at least one of them reports a failing check, the whole pipeline fails, and the next stages are not executed. The checks are performed by executing the `tfsec` and `regula run` commands. The output of the scansion is collected inside the *tfsec_audit.json* and *regula_audit.json* files respectively. The directory containing the custom policies is specified via the `--rego-policy-dir` parameter for tfsec and `--include` parameter for Regula. If the previous phase ends correctly, then there is the execution

of the "Deploy" phase. It has the purpose of deploying the infrastructure into the cloud environment by applying the `terraform apply` command over the plan produced in the first phase. Note that also this command is encapsulated inside a `withCredentials` block because it needs to perform operations described in the plan for deploying the infrastructure in AWS.

The last phase saves the output of the tools and the Terraform plan as build artifacts. In this way, developers can analyze them in case a misconfiguration in the infrastructure code occurs.

```
pipeline {
 agent any
 stages {
  stage("Planning") {
   steps {
    withCredentials([usernamePassword(credentialsId: "aws-terraform-credentials", usernameVariable: "AWS_ACCESS_KEY_ID", passwordVariable: "AWS_SECRET_ACCESS_KEY")]) {
      sh "terraform init -no-color"
      sh "terraform plan -out plan.tfplan -refresh=false -no-color"
    }
      sh "terraform show -json plan.tfplan > plan.json"
   }
  }
  stage("Infrastructure Scanning") {
   parallel {
    stage("tfsec") {
     steps {
      sh "tfsec . --no-colour --no-code --include-passed --format json --rego-policy-dir .tfsec_rules > tfsec_audit.json"
     }
    }
    stage("Regula") {
     steps {
      sh "regula run plan.json --input-type tf-plan --include .regula_rules --format json > regula_audit.json"
     }
    }
   }
  }
  stage("Deploy") {
   steps {
    withCredentials(...) {
     sh "terraform apply tfplan -no-color"
    }
   }
  }
 }
```

```
35  post {
36   always {
37     archiveArtifacts "*audit.json"
38     archiveArtifacts "plan.tfplan"
39   }
40  }
41 }
```

Listing 5.8. IaCSec pipeline.

5.5.2 Set up of the pipeline for CSPM

The listing 5.9 shows how the CSPM pipeline is implemented. This pipeline is designed for deploying CSPM policies and obtaining a compliance violation report. To accomplish this, the Docker image provided by Cloud Custodian is chosen as the execution environment to show this feature of Jenkins. The `agent` block is used for this purpose, it allows specifying the default execution environment for the entire pipeline or a specific one for a single stage. In this case, only the image and some additional arguments are specified. Those arguments are the entry point of the container and an environment variable containing the default AWS region used for deploying policies and getting reports.

This Pipeline specifies two parameters: `deploy_policies` and `get_report`. The first is utilized for specifying whether execute the stage in charge of deploying policies, while the second specifies whether to perform the stage defined for obtaining reports.

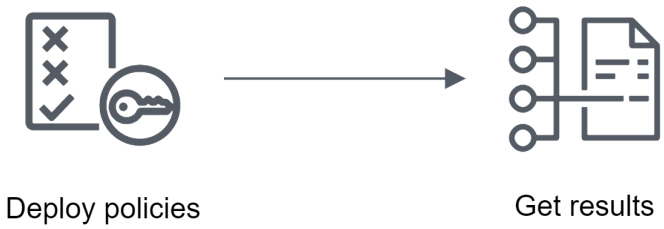

Figure 5.3. Stages of the pipeline for CSPM.

The first phase involves the deployment of Periodic and Event-Based policies within the cloud environment. This is made possible by executing the `custodian run` command over the folders which contain the policies. The command also defines where to write the output of the policies evaluation.

In this case, the destination is an S3 bucket specified via the `--output-dir` parameter. Note that Cloud Custodian, in order to function, needs to authenticate with AWS and, analogously to the previous pipeline, this is accomplished using the `withCredentials` block.

The last stage is designed for producing the policy compliance violation report. The latter is obtained by executing the `custodian report` command. To obtain a comprehensive report regarding each defined policy, the first operation that is performed is a compliance check against dry-run policies, i.e. those defined with a Pull execution mode. Once the check is done, information about eventual compliance violations are gathered from the S3 bucket to compose and store the final report.

```
1  pipeline {
2   agent{
3    docker{
4     args '--entrypoint="" -e AWS_DEFAULT_REGION=eu-west-3'
5     image "cloudcustodian/c7n:0.9.23.0"
6    }
7   }
8   parameters {
9    booleanParam defaultValue: true, name: "deploy_policies"
10   booleanParam defaultValue: true, name: "get_report"
11  }
12  stages {
13   stage("Deploy policies") {
14    when {
15     expression {
16      return params.deploy_policies
17     }
18    }
19    steps {
20       withCredentials(...) {
21       sh "custodian run --output-dir s3:<URI>/periodic pack/
     periodic/"
22       sh "custodian run --output-dir s3:<URI>/event-based
     pack/event-based/ "
23       }
24     }
25   }
26   stage("Get results") {
27    when {
28     expression {
29      return params.get_report
30     }
31    }
```

```
32    steps {
33     withCredentials(...) {
34      sh "custodian run --output-dir s3:<URI>/dry-run pack/dry
    -run"
35      sh '''
36      echo "[" > custodian_audit.json
37      for dir in $(ls pack); do
38       for file in $(ls pack/$dir); do
39        custodian report -s s3://custodian-log-bucket/
    custodian-policies/$dir pack/$dir/$file --format=json
40         echo ,
41       done
42      done >> custodian_audit.json
43      truncate -s -2 custodian_audit.json
44      echo "]"  >> custodian_audit.json
45      '''
46      archiveArtifacts "custodian_audit.json"
47     }
48    }
49   }
50 }
```

Listing 5.9. CSPM pipeline.

5.5.3 Set up of the pipeline for CWPP and the sample application

The listing 5.10 contains the implementation of the pipeline in charge of deploying the CWPP tools and policies together with a sample application.

Also, this Pipeline is parametric and two parameters are specified within it: `deploy_gatekeeper` and `DB_URI`. The first specifies whether execute the stage in charge of deploying OPA gatekeeper, while the second one is employed for specifying the uri of the RDS DB instance utilized by the sample application.

Figure 5.4. Stages of the pipeline for CWPP.

The first stage is in charge of configuring the *kubeconfig* file required for connecting to the EKS cluster. This stage leverages the `aws eks update -kubeconfig` command. The name of the cluster that will be configured is specified by the parameter `--name`. This command is encapsulated inside a `withCredentials` block since it requires the credentials for authenticating to AWS to operate.

The second stage has the duty to deploy OPA gatekeeper into the cluster. This is done through the `kubectl apply` command [86] by passing to it the configuration file containing the definition needed for deploying the tool. Also, this command and its other occurrences need to be contained inside a `withCredentials` block. This is because AWS authentication is required for operating with EKSs clusters

The third stage deploys the CWPP policies. It firstly creates two configuration files: `templates-manifest.yml` and `constraints-manifest.yml`. Those files contain the definitions for constraint templates and constraints respectively, and they are generated via the `kustomize build` command. Once generated, they are deployed inside the cluster by using the `kubectl apply` [86] command.

The fourth stage defines how to deploy the application. The procedure is analogous to the one performed during the second step. The only difference is the `sed` command executed before the `kubectl apply` for inserting the value contained inside the `DB_URI` parameter into the *app.yaml* file.

The last phase utilizes the `kubectl log` command for capturing the log entries emitted by OPA Gatekeeper in the form of JSON objects. Those entries are filtered by the `jq` command to keep only those entries that refer to a policy violation detected during a periodic audit performed by OPA Gatekeeper.

```
1  pipeline {
2   agent any
3   parameters {
4    booleanParam defaultValue: true, name: "deploy_gatekeeper"
5    string name: 'DB_URI'
6   }
7   stages {
8    stage("Configure EKS cluster") {
9     steps {
10     withCredentials(...){
11      sh "aws eks --region eu-west-3 update-kubeconfig --name
     eks-lab-cluster"
12     }
13    }
14   }
```

```
15   stage("Deploy gatekeeper") {
16    when {
17     expression {
18        return params.deploy_gatekeeper
19     }
20    }
21    steps {
22     withCredentials(...){
23      sh "kubectl apply -f target/gatekeeper.yaml
24     }
25    }
26   }
27   stage("Deploy rules") {
28    steps {
29     sh "customize build overlays/templates/ > templates-
     manifest.yml"
30     sh "customize build overlays/constraints/ > constraints-
     manifest.yml"
31     withCredentials(...){
32      sh "kubectl apply -f templates-manifest.yml"
33      sh "kubectl apply -f constraints-manifest.yml"
34     }
35    }
36   }
37   stage("Deploy app") {
38    steps {
39      sh "sed  \"s/<DB_URI>/${params.DB_URI}/\" target/app.
     yaml > target/pod.yaml"
40     withCredentials(...){
41      sh "kubectl apply -f target/app.yaml
42     }
43    }
44   }
45  }
46  post {
47   always {
48    sh "kubectl logs -n gatekeeper-system deployments/
     gatekeeper-audit | sed '1d' | jq -s '. | map(. | select(.
     event_type=="violation_audited") | del(.ts) | del(.
     audit_id)) | unique | .[]' > gatekeeper_audit.json"
49    archiveArtifacts "gatekeeper_audit.json"
50   }
51  }
52 }
```

Listing 5.10. CWPP pipeline.

5.6 Custom policies

The PoC contains some custom policies that are developed and deployed if needed, to show how an organization can define and employ them.

5.6.1 IaCSec policy

Unfortunately, the default policy package of tfsec and Regula does not recognize a serious vulnerability present within the Terraform infrastructure code.

The listing 5.5 shows how the RDS DB instance is defined. This configuration requires the specification of a master password directly inside the Terraform code, the problem is that this password could be hard-coded.

To show this problem, the first version of the architecture defines the RDS instance password with a value hard-coded in the file. At the moment of the first scan with the IaCSec tools, neither tfsec nor Regula recognized it. However, this happens because recognizing whether or not a value is hard-coded within an argument is not a trivial task. The only way for doing so is to scan the Terraform plan code because it contains all the information about the source of each value that will be assigned to an argument.

Unfortunately, by default Regula and tfsec scan the directory containing Terraform configurations but, as said before, in this way it is not possible to recognize the provenance of a value inserted into an argument. This happens because both tools take as input only the JSON representation of the final infrastructure.

The listing 5.11 shows the policy code developed for mitigating this problem.

```
1  package rules.user_attached_policy
2
3  import data.fugue
4
5  __rego__metadoc__ := {
6    "custom": {"severity": "High"},
7    "description": "RDS DB instance password must not be
       neither hard-coded nor specified inside a variable marked
       as not sensitive or with a default value",
8    "id": "CUS_R00001",
9    "title": "RDS DB instance password must not be hard-coded"
10 }
11
12 resource_type := "MULTIPLE"
```

```
configuration_resources := [res | input._plan.configuration.
    root_module.resources[i].type == "aws_db_instance"; res =
    input._plan.configuration.root_module.resources[i]]
variables := input._plan.configuration.root_module.variables

uncompliant_rds[p] {
 variables[var]["default"]
 sprintf("var.%v", [var]) == configuration_resources[i].
    expressions.password.references[_]
 p = {"id": configuration_resources[i].address, "msg":
    sprintf("the variable \"%v\" used for specifying the DB
    instance password has a default value", [var])}
}{
 variables[var]
 not variables[var].sensitive
 sprintf("var.%v", [var]) == configuration_resources[i].
    expressions.password.references[_]
 p = {"id": configuration_resources[i].address, "msg":
    sprintf("the variable \"%v\" used for specifying the DB
    instance password is not sensitive", [var])}
}{
 not configuration_resources[i].expressions.password.
    references
 p = {"id": configuration_resources[i].address, "msg": "the
    DB instance password is hard coded"}
}

policy[p] {
 single_resource := fugue.resources("aws_db_instance")[_]
 single_resource.id == uncompliant_rds[i].id
 p = fugue.deny_resource_with_message(single_resource,
    sprintf("%v", [uncompliant_rds[i].msg]))
}{
 single_resource := fugue.resources("aws_db_instance")[_]
 startswith(single_resource.id, sprintf("%v[", [
    uncompliant_rds[i].id]))
 p = fugue.deny_resource_with_message(single_resource,
    sprintf("%v", [uncompliant_rds[i].msg]))
}
```

Listing 5.11. Hard-coded password detection policy.

This policy was developed for Regula since tfsec is not able to scan the Terraform plan. The policy helps to mitigate the problem discussed previously since it does not allow the deployment of an infrastructure that presents an `aws_db_instance` resource block which contains a password hard-coded or defined inside a variable that is not sensitive or with a default value.

The policy lays on four rules. The `configuration_resources` rule queries a list of all the resources of type `aws_db_instance` inside the Terraform plan configuration field. This field contains a list of all resources that should be deployed, and for each resource property, the origin of the values assumed by their properties is defined.

The `variables` rule defines a list of all the defined variables, while the `uncompliant_rds` rule verifies the effective compliance of the policy. This rule has three definitions:

- the first definition queries all the DB instance resources which own a password whose value comes from a variable with a default value;
- the second definition queries all the DB instance resources which own a password whose value is not marked as sensitive;
- the third definition checks that the password of a DB instance is not hard-coded.

The rule produces a collection of documents, and each document contains the ID of the non-compliant resources and a message that will be printed out by the tool. The `policy` rule is used by Regula to obtain evaluation results. In this case, it relies on the `id` field document produced by the `uncompliant_rds` rule to identify non-compliant resources among those returned by the `fugue.resources` function. These resources are provided to the `fugue.deny_resource_with_message` function along with the corresponding messages to indicate that they are not compliant.

5.6.2 CSPM policies

The CSPM policies developed are taken directly from the CIS Amazon Web Services Benchmarks [87].

The listings 5.12, 5.13, and 5.14 show an example of how policies written inside a regulation can be translated into code and enforced automatically. Furthermore, they are also useful for showing an example of the various execution modes provided by Cloud Custodian.

The figure 5.5 depicts how periodic and event-based policies, deployed by the pipeline for CSPM, operate in AWS. In order to execute these kinds of controls, Cloud Custodian leverage on AWS Lambda, a computing service that permits running code without provisioning or managing servers [88].

Lambda functions can be triggered by EventBridge rules managed by Amazon EventBridge [89] a serverless service that uses events to link the components of an application together. An EventBridge rule simply listens for incoming events and sends them to targets for processing.

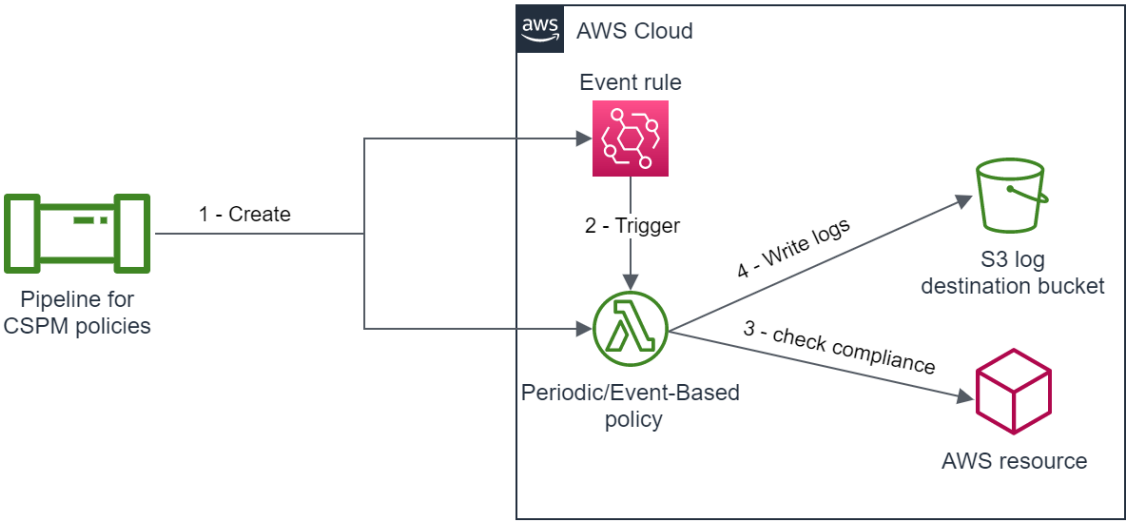

Figure 5.5. CSPM Periodic/Event-Based policy operation.

In the proof-of-concept that was developed, the flow is:

1. the execution of the pipeline for CSPM creates an EventBridge rule and the corresponding lambda that will be triggered by the rule;

2. when the event specified inside the rule occurs, it will trigger the execution of the lambda;

3. the execution of the lambda checks that all the resources whose type corresponds to the one specified in the policy code are compliant with the policy itself;

4. the result of the evaluation is written into the *custodian-logs* S3 bucket.

The listing 5.12 contains a definition of a policy that will be executed in pull mode. This policy is used to spot Amazon RDS instances that are unencrypted. This is accomplished by filtering the resources of type `aws.rds` that have the `StorageEncrypted` property set to false.

```
policies:
- name: cis-rds-encryption-not-enabled-at-rest
  resource: aws.rds
  comment: |
    CIS AWS Foundations v1.5.0 (2.3.1). Databases are likely
   to hold sensitive and critical data, it is highly
   recommended to implement encryption to protect your data
   from unauthorized access or disclosure. With RDS
   encryption enabled, the data stored on the instance's
   underlying storage, the automated backups, read replicas,
   and snapshots, are all encrypted.
  filters:
    - type: value
      key: StorageEncrypted
      op: ne
      value: true
```

Listing 5.12. Dry-run policy that verifies whether all Amazon RDS instances are encrypted.

The listing 5.13 contains the definition of a policy that will be executed in periodic mode. This policy has the same purpose as the one mentioned in the listing 5.12. The only thing that changes is the execution mode. In this case, the policy is executed once a day.

```
policies:
- name: cis-rds-encryption-not-enabled-at-rest
  resource: aws.rds
  comment: |
    CIS AWS Foundations v1.5.0 (2.3.1). Databases are likely
   to hold sensitive and critical data, it is highly
   recommended to implement encryption in order to protect
   your data from unauthorized access or disclosure. With RDS
    encryption enabled, the data stored on the instance's
   underlying storage, the automated backups, read replicas,
   and snapshots, are all encrypted.
   mode:
    schedule: "rate(24 hours)"
    type: periodic
    tags:
     custoidan-test: "test-2"
    role: arn:aws:iam::<ACCOUNT_ID>:role/
   CustodianLambdaTestRole1
   filters:
    - type: value
      key: StorageEncrypted
      op: ne
      value: true
```

Listing 5.13. Periodic policy that verifies whether all Amazon RDS instances are encrypted.

To perform this periodic evaluation, Cloud Custodian deploys an AWS Lambda function triggered once a day by a periodic EventBridge rule. This allows the execution of the evaluation using the IAM role specified by the `role` property and redirects the output to the destination.

```
policies:
 - name: terminate-unencrypted-ebs
   comment: |
     CIS Amazon Web Services Foundations v1.5.0. (2.2.1)
   Elastic Compute Cloud (EC2) supports encryption at rest
   when using the Elastic Block Store (EBS) service. While
   disabled by default, forcing encryption at EBS volume
   creation is supported.
   resource: aws.ec2
   mode:
     type: ec2-instance-state
     events:
       - running
     tags:
       custoidan-test: "test-2"
     role: arn:aws:iam::<ACCOUNT_ID>:role/
   CustodianLambdaTestRole2
   filters:
     - type: ebs
       key: Encrypted
       value: false
   actions:
     - type: stop
```

Listing 5.14. Run-time policy that stops all EC2 instances with an unencrypted EBS that switches from pending to running state.

Finally, the listing 5.14 shows an example of a policy that will be evaluated each time an event happens and performs a remediation. This policy has the purpose to stop all EC2 instances connected to an unencrypted Amazon Elastic Block Store (EBS) volume [90] that switches from pending to running state. For doing so, Cloud Custodian deploys an AWS Lambda function triggered by an Amazon EventBridge event generated each time an EC2 instance that violates the policy, switches from pending to running state. This behavior is described by the `mode` property of the policy. The `actions` property describes the actions that should be performed over the filtered resources. The evaluation and the auto-remediation actions are executed using the IAM role specified by the `role` property.

5.6.3 CWPP policy

The listings 5.15 and 5.16 show the policy code developed to avoid a particular set of environment variables utilized by a container being defined with hard-coded values. The listings contain the definition of the policy constraint template and constraint, respectively.

```
apiVersion: templates.gatekeeper.sh/v1beta1
kind: ConstraintTemplate
metadata:
 creationTimestamp: null
 name: no-hardcoded-env-variables
spec:
 crd:
  spec:
   names:
    kind: no-hardcoded-env-variables
   validation:
    openAPIV3Schema:
     properties:
      container_name:
       type: string
       description: The name of the container to which apply the policy
      env_vars:
       type: array
       minItems: 1
       items:
        type: string
       description: The environment variables that should not be hard-coded
 targets:
 - target: admission.k8s.gatekeeper.sh
   rego: |
    package k8srequiredlabels
    violation[{"msg": msg}] {
     container = input.review.object.spec.containers[_]
     container.name == input.parameters.container_name
     container_envs := {
      name |
      container.env[i].value
      name = container.env[i].name
     }
     input_envs = {env | env = input.parameters.env_vars[_]}
     res_envs = input_envs & container_envs
     count(res_envs) > 0
     msg = sprintf("The following environment variables for the container %v are hardcoded: %v",[container.name, res_envs])
    }
```

Listing 5.15. Constraint template of the hard-coded environment variables detection policy.

The set of variables and the name of the target container are defined in the `parameters` field of the constraint. The contents of this property are accessible to the rule specified in the constraint template via the `input.parameters` reference. This allows the constraint template to be reused by specifying other container names or other sets of variables.

The `violation` rule creates a set containing the hard-coded environment variables and performs the intersection between it and the set defined within parameters. This operation is performed for each container with a name equal to the value specified in the parameters. If the resulting set is not empty, the policy is violated and a message is printed with the relevant information.

```
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: no-hardcoded-env-variables
metadata:
 name: demo-app-test
spec:
 match:
  kinds:
   - kinds: ["Pod"]
  parameters:
   container_name: demo-app
   env_vars:
    - DB_USR
    - DB_PWD
```

Listing 5.16. Constraint of the hard-coded environment variables detection policy.

In this case, the constraint is enforced on resources of kind "Pod" and specifies "`demo-app`" as the name of the target container. This is the name of the container that contains the target application. The constraint also specifies that the "`DB_USR`" and "`DB_PWD`" environment variables must not be hard-coded since they are employed by the target application for authenticating it to the DB.

Chapter 6

Wrap-Up model definition and Mitigation strategies

The following chapter describes the structure of the results of the evaluation carried out on the proof-of-concept. Also, it explains the detected violations and the actions taken to bring the system into compliance with the defined policies.

6.1 How information are gathered

The input information needed for performing the assessment are gathered directly by the tools.

Regula and tfsec analyze a JSON representation of the target infrastructure by scanning the infrastructure code's directory. Also, Regula can scan a JSON representation of the Terraform Plan which contains various useful metadata, however is not generated by Regula and must be passed directly as input to the tool.

Cloud Custodian obtains the information in a different way. It authenticates with the cloud provider to query a structured representation of the cloud infrastructure resources described in the policies. Once obtained, it verifies them against the corresponding policy.

Finally, OPA Gatekeeper performs policy evaluation both periodically and at the time of deployment of a resource subject to a policy. It scans only the Kubernetes objects which kind corresponds to the one defined inside the policies. These objects can be already deployed inside the cluster or intercepted from a request directed to the Kubernetes API Server.

In case of a violation, if it is spotted by a periodic evaluation the result is emitted as JSON object by the `gatekeeper-audit` Pod, while if it is detected during the deployment of a resource, the result is directly shown as a result of the `kubectl apply` command.

The output of the tools is collected by Jenkins during the execution of the developed CI/CD pipelines. Regula and tfsec produce directly the output once the scan is terminated. Conversely, the output of Cloud Custodian and OPA gatekeeper is not obtained in this way. Cloud Custodian output is collected by executing the `custodian report` command, while the OPA Gatekeeper one is obtained via the logs of the `gatekeeper-audit` Pod.

6.2 How the wrap-up is structured

Information about the output of the tools is collected by Jenkins in the form of artifacts. By following this approach, each time a pipeline is executed the tool output remains tied to the single execution.

As shown in figure 6.1, Jenkins keeps track of each execution of each pipeline, hence is possible to access artifacts produced during different execution of the same pipeline by hovering over the download circle.

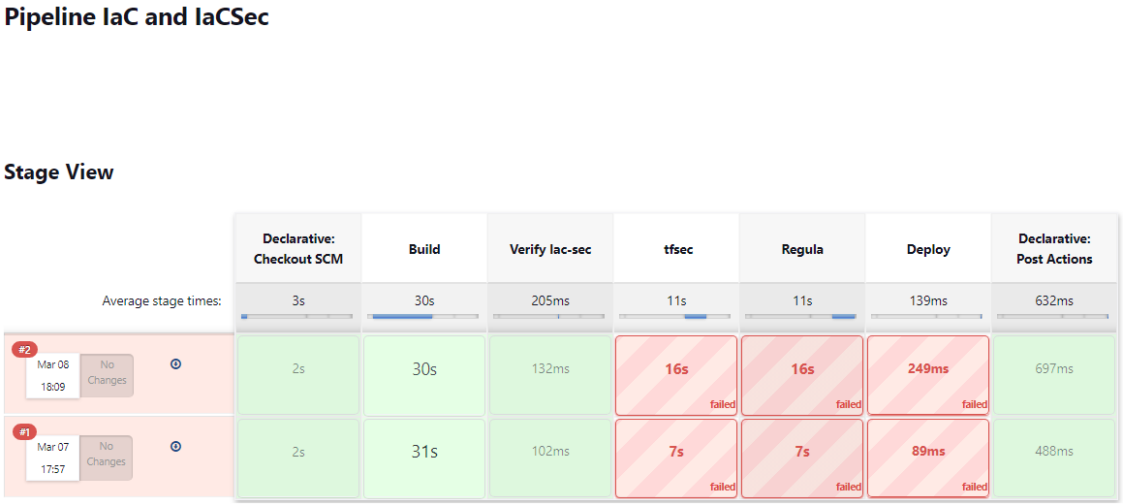

Figure 6.1. Sample pipeline execution results.

The artifact saved by Jenkins are:

- `tfsec_audit.json` and `regula_audit.json` inside the pipeline for IaC and IaCSec;
- `custodian_audit.json` inside the pipeline for CSPM;
- `gatekeeper_audit.json` inside the pipeline for CWPP.

Each file is saved in a JSON format with its own structure and contains policy violations detected by the relative tool. Also, Regula and tfsec audits give information about the severity of the violations, while the Cloud Custodian and OPA Gatekeeper do not. Finally, because OPA gatekeeper performs periodic policy scans on the cluster, the `gatekeeper-audit` Pod may emit the same violation multiple times. This problem is avoided by aggregating entries that refer to the same violation.

6.3 Wrap-up files structure

6.3.1 Content of the file tfsec_audit.json

This file contains a single JSON object with only one property called `results` and contains a list of other JSON objects. Each object in the list represents the result of an evaluation (either passed or not) and some other information as shown by the listing 6.1.

```
{
 "rule_id": "AVD-AWS-0038",
 "long_id": "aws-eks-enable-control-plane-logging",
 "rule_description": "EKS Clusters should have cluster
   control plane logging turned on",
 "rule_provider": "aws",
 "rule_service": "eks",
 "impact": "Logging provides valuable information about
   access and usage",
 "resolution": "Enable logging for the EKS control plane",
 "links": [
  "https://aquasecurity.github.io/tfsec/v1.28.1/checks/aws/
   eks/enable-control-plane-logging/", "https://registry.
   terraform.io/providers/hashicorp/aws/latest/docs/resources
   /eks_cluster#enabled_cluster_log_types"
 ],
 "description": "Control plane scheduler logging is not
   enabled.",
```

```
13  "severity": "MEDIUM",
14  "warning": false,
15  "status": 0,
16  "resource": "module.eks",
17  "location": {
18   "filename": "registry.terraform.io/terraform-aws-modules/
     eks/aws/home/mattia/tesi-terraform-iacsec-final/.terraform
     /modules/eks/main.tf",
19   "start_line": 17,
20   "end_line": 82
21  }
22 }
```

Listing 6.1. An entry of the file: `tfsec_audit.json`.

The additional information are represented by the values assumed by the object properties. Although most of them are self-explanatory, the meaning of the property `status` is not very clear. It represents the result of the evaluation and assumes the value `0` in case of violation and `1` otherwise.

6.3.2 Content of the file regula_audit.json

The structure of the file is quite similar to the `tfsec_audit.json` one. Also in this case the file contains a single JSON object but this one has two properties: `rule_results` and `summary`.

The `rule_result` property is analogous to the `results` property discussed earlier. Although both contain an array of JSON objects and each represents the result of an evaluation, the objects contained in one list have a different structure than those contained in the other.

```
1  {
2    "controls": [
3      "CIS-AWS_v1.2.0_4.3",
4      "CIS-AWS_v1.3.0_5.3",
5      "CIS-AWS_v1.4.0_5.3"
6    ],
7    "families": [
8      "CIS-AWS_v1.2.0",
9      "CIS-AWS_v1.3.0",
10     "CIS-AWS_v1.4.0"
11   ],
12   "filepath": "plan.json",
13   "input_type": "tf_plan",
14   "provider": "aws",
15   "resource_id": "module.vpc.aws_vpc.this[0]",
16   "resource_type": "aws_vpc",
```

```
17   "resource_tags": {
18     "LAB": "tesi_mattia",
19     "Name": "eks-lab-vpc-module",
20     "infra": "terraform",
21     "kubernetes.io/cluster/eks-lab-cluster-module": "owned"
22   },
23   "rule_description": "VPC default security group should
      restrict all traffic. Configuring all VPC default security
       groups to restrict all traffic encourages least privilege
       security group development and mindful placement of AWS
      resources into security groups which in turn reduces the
      exposure of those resources.",
24   "rule_id": "FG_R00089",
25   "rule_message": "",
26   "rule_name": "tf_aws_vpc_default_security_group",
27   "rule_raw_result": false,
28   "rule_remediation_doc": "https://docs.fugue.co/FG_R00089.
      html",
29   "rule_result": "FAIL",
30   "rule_severity": "Medium",
31   "rule_summary": "VPC default security group should restrict
       all traffic"
32 }
```

Listing 6.2. An entry of the file: `regula_audit.json`.

The listing 6.2 contains a sample object contained inside the list defined by the `rule_result` property. Differently from the previous case, the object properties are quite self-explanatory.

```
1 "summary": {
2   "filepaths": [
3     "plan.json"
4   ],
5   "rule_results": {
6     "FAIL": 8,
7     "PASS": 175,
8     "WAIVED": 0
9   },
10  "severities": {
11    "Critical": 0,
12    "High": 2,
13    "Informational": 0,
14    "Low": 0,
15    "Medium": 6,
16    "Unknown": 0
17  }
18 }
```

Listing 6.3. The summary property the file: `regula_audit.json`.

Finally, the listing 6.3 contains an example of the `summary` property, which contains an object that summarizes the information collected by the scan.

6.3.3 Content of the file custodian_audit.json

The file contains an array object whose elements are other array objects. Each second level array represents the result of a policy evaluation and lists the resource instances that violate the policy itself.

```
{
  "DBInstanceIdentifier": "database-1-instance-1",
  "DBInstanceClass": "db.t2.small",
  "Engine": "PostgreSQL",
  "DBInstanceStatus": "configuring-enhanced-monitoring",
  "MasterUsername": "admin",
  "Endpoint": {
   "Address": "database-1-instance-1.cznghtt5w6mg.eu-west-3.
   rds.amazonaws.com",
   "Port": 5432,
   "HostedZoneId": "ZABCDEFGHIJKLM783"
  },
  "AllocatedStorage": 1,
  "InstanceCreateTime": "2023-03-24T09:38:27.114000+00:00",
  "PreferredBackupWindow": "12:37-13:07",
  "BackupRetentionPeriod": 35,
  "DBSecurityGroups": [],
  ...
  "AvailabilityZone": "eu-west-3b",
  "DBSubnetGroup": {
   "DBSubnetGroupName": "default-vpc-123",
   "DBSubnetGroupDescription": "Created from the RDS
   Management Console",
  "c7n:MatchedFilters": [
   "StorageEncrypted"
  ],
  "CustodianDate": "2023-03-24T10:39:59.443198",
  "policy": "cis-rds-encryption-not-enabled-at-rest",
  "region": "eu-west-3"
 }
```

Listing 6.4. An entry of the file: `custodian_audit.json`.

The listing 6.4 shows an example of the result of an evaluation. Note also that each object that represents an uncompliant resource, contains also three additional properties. These are added directly by Cloud Custodian and are utilized for specifying the policy violated, the date-time on which the compliance check has been performed, and the matched filters of a policy.

6.3.4 Content of the file gatekeeper_audit.json

This file collects the history of the results of each periodic evaluation performed by OPA Gatekeeper so far. The file consists of an array in which each element represents the result of a periodic policy evaluation. The listing 6.5 shows a sample object coming from this array.

```
{
 "level": "info",
 "logger": "controller",
 "msg": "The following environment variables for the
   container demo-app are hardcoded: {\"DB_PWD\", \"DB_USR
   \"}",
 "process": "audit",
 "details": {},
 "event_type": "violation_audited",
 "constraint_group": "constraints.gatekeeper.sh",
 "constraint_api_version": "v1beta1",
 "constraint_kind": "no-hardcoded-env-variables",
 "constraint_name": "demo-app-test",
 "constraint_namespace": "",
 "constraint_action": "deny",
 "constraint_annotations": {},
 "resource_group": "",
 "resource_api_version": "v1",
 "resource_kind": "Pod",
 "resource_namespace": "default",
 "resource_name": "demo-app",
 "resource_labels": {
   "purpose": "demonstrate-envars"
 }
```

Listing 6.5. An entry of the file: `gatekeeper_audit.json`.

Similarly to the previous cases, the fields are quite self-explanatory. However, is important to note that fields `constraint_kind`, `constraint_name`, and `resource_name` are the most useful properties contained inside this object. The first two represent the kind and the name of the violated constraint, while the last represents the name of the resource that violates the policy.

6.4 Mitigation strategies description

As explained earlier, the target infrastructure, the CSPM policies, the CWPP policies, and the sample application are deployed using different CI/CD pipelines. Here are explained the policy violations detected and the relative mitigation actions.

6.4.1 tfsec violations

The violations detected by tfsec during the first execution of the pipeline are listed in the table 6.1.

Description	Severity	Resource
Security group rule allows egress to multiple public internet addresses.	CRITICAL	`module.eks`
Subnet associates public IP address.	HIGH	`module.vpc`
Instance does not have storage encryption enabled.	HIGH	`aws_db_instance.rds_db[0]`
VPC Flow Logging is not enabled for VPC	MEDIUM	`module.vpc`
Control plane controller manager logging is not enabled.	MEDIUM	`module.eks`
Control plane scheduler logging is not enabled.	MEDIUM	`module.eks`
Instance does not have Deletion Protection enabled.	MEDIUM	`aws_db_instance.rds_db[0]`
Instance does not have IAM Authentication enabled.	MEDIUM	`aws_db_instance.rds_db[0]`
Instance has very low backup retention period.	MEDIUM	`aws_db_instance.rds_db[0]`
Instance does not have performance insights enabled	LOW	`aws_db_instance.rds_db[0]`

Table 6.1. Violations detected by tfsec.

Each row in the table is derived from a JSON object present in the actual evaluation. In this case, only the values derived from the properties `description`, `severity`, and `resource` are reported.

6.4.2 regula violations

The violations detected by Regula during the first execution of the pipeline are listed in the table 6.2.

Rule summary	**Severity**	**Resource**
RDS instances should be encrypted.	HIGH	`aws_db_instance.rds_db[0]`
RDS DB instance password must not be hard-coded.	HIGH	`aws_db_instance.rds_db[0]`
VPC Flow Logging is not enabled for VPC.	MEDIUM	`module.vpc`
CloudWatch log groups should be encrypted with customer managed KMS keys.	MEDIUM	`module.eks`
VPC default security group should restrict all traffic	MEDIUM	`module.vpc`
RDS instances should have backup retention periods configured.	MEDIUM	`aws_db_instance.rds_db[0]`
Require Multi Availability Zones turned on for RDS Instances.	MEDIUM	`aws_db_instance.rds_db[0]`
RDS instance “Deletion Protection” should be enabled.	MEDIUM	`aws_db_instance.rds_db[0]`

Table 6.2. Violations detected by Regula.

Each row in the table is derived from a JSON object present in the actual evaluation. In this case, only the values derived from the properties `rule_summary`, `rule_severity`, and `resource_id` are reported.

As expected, the default configuration presents some violations concerning the custom rule and the rules defined within the default packages of the tools. The figure 6.2 summarizes the number of violations found for each level of severity. Note that violations with a similar description have been counted only once.

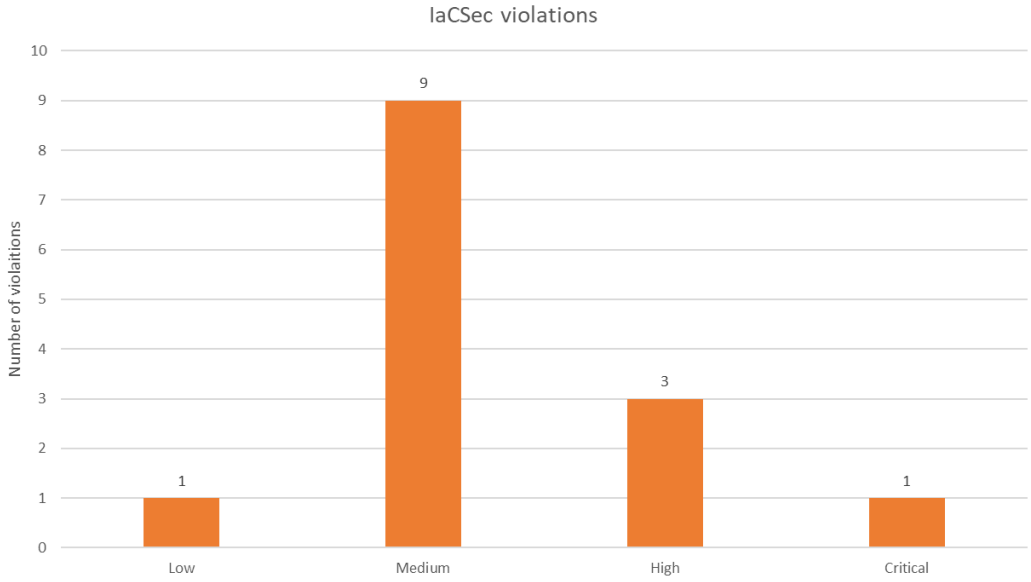

Figure 6.2. Number of violations for each level of severity.

6.4.3 IaCSec mitigations

The discovered misconfigurations can be easily fixed by following the instructions provided by the pages indicated by the links in each violation message. Also, it is interesting to note that the two scans do not provide the same results. Indeed, some violations are reported by both tools, some are not. To mitigate them, some actions over the architecture code must be performed.

The listing 6.6 shows the changes performed to the `eks` module for solving the relative misconfigurations.

```
#tfsec:ignore:aws-ec2-no-public-egress-sgr
module "eks" {
  ...
  node_security_group_additional_rules = {
    egress_all = {
      description = "Allow egress only inside the vpc and to other AWS IPs"
      protocol    = "-1"
      from_port   = 0
      to_port     = 65535
      type        = "egress"
      cidr_blocks = [ module.vpc.vpc_cidr_block,
                      "16.12.18.0/23",
                      "16.12.20.0/24",
                      "3.5.224.0/22",
                      "52.95.154.0/23",
                      "52.95.156.0/24"
                    ]
    }
  create_cloudwatch_log_group = false
  cluster_enabled_log_types   = ["api", "authenticator", "audit", "scheduler", "controllerManager"]
  ...
}
```

Listing 6.6. Adjustments made to the `eks` module.

The modified block presents an additional `node_security_group_rule` which avoids the egress traffic from the worker nodes to multiple public internet addresses, except those required for cluster operation. Additionally, it also presents a list of the log types that must be recorded. Finally, the `create_cloudwatch_log_group` field is set to false for delegating the creation of the CloudWatch log group to AWS.

Also, the `vpc` module definition must be changed according to the result of the evaluation. These changes concern the definition of the default security group and the VPC flow log configuration as shown by the listing 6.7

```
#tfsec:ignore:aws-ec2-require-vpc-flow-logs-for-all-vpcs
module "vpc" {
  ...
  manage_default_security_group  = true
  default_security_group_egress  = []
  default_security_group_ingress = []
  default_security_group_name    = "${var.vpc_name}-default-sg"
  map_public_ip_on_launch = false
  enable_flow_log                      = true
```

```
10   create_flow_log_cloudwatch_log_group = true
11   create_flow_log_cloudwatch_iam_role  = true
12   flow_log_cloudwatch_log_group_name_prefix = "/aws/${var.
      vpc_name}-logs/"
13   flow_log_cloudwatch_log_group_name_suffix = "test"
14   flow_log_cloudwatch_log_group_kms_key_id  = aws_kms_key.
      vpc_key.arn
15   ...
16 }
17
18 resource "aws_kms_key" "vpc_key" {
19   enable_key_rotation = true
20   policy = data.aws_iam_policy_document.cloudwatch.json
21 }
```

Listing 6.7. Adjustments made to the `vpc` module.

Unfortunately, some controls continue to be violated even after mitigation strategies are implemented. To remedy this, it is necessary to use the features of tfsec and Regula that allow violations to be ignored.

As shown by the listings 6.6 and 6.7, tfsec [91] allows a violation to be ignored by adding a special comment at the location where it occurs. In this case two violations are ignored. One is related to the activation of VPC flow logs, and the other is related to egress to multiple public IPs by a security group.

Instead, Regula [92] uses another mechanism to ignore violations which requires the implementation of some special custom rules that must be called `waivers`.A waiver rule is used to define a special object whose fields specify the controls to be ignored, specific resources to which no controls should be applied, or specific ones for a specific resource.

The waiver rules produced in this case are illustrated by the listing 6.8. By doing so is possible to ignore the controls relative to the activation of VPC flow logs and to the encryption of hte relative CloudWatch log groups. This is done by indicating the Terraform `resource_id` of the resource for which the control should be ignored and the `rule_id` of the control itself.

```
1 package fugue.regula.config
2
3 waivers[waiver] {
4  waiver := {
5   "resource_id": "module.vpc.aws_vpc.this[0]",
6   "rule_id": "FG_R00054",
7  }
8 }
9
```

```
waivers[waiver] {
 waiver := {
  "resource_id": "module.vpc.aws_cloudwatch_log_group.
   flow_log[0]",
  "rule_id": "FG_R00068",
 }
}
```

Listing 6.8. Regula waived rules.

The changes performed to the `aws_db_instance` resource are illustrated by the listing 6.9. It shows the resources and the variables that should be added to apply correctly the changes performed to the `aws_db_instance` resource definition.

```
resource "aws_db_instance" "rds_db" {
  ...
  password                            = var.db_pwd
  backup_retention_period             = 5
  iam_database_authentication_enabled = true
  storage_encrypted                   = true
  deletion_protection                 = true
  performance_insights_enabled        = true
  performance_insights_kms_key_id     = aws_kms_key.
   rds_performance_insights.arn
  multi_az                            = true
  ...
}

resource "aws_kms_key" "rds_performance_insights" {
  enable_key_rotation = true
  policy = data.aws_iam_policy_document.insight.json
}

variable "db_pwd" {
  type = string
  sensitive = true
}
```

Listing 6.9. Adjustments performed to the `aws_db_instance` resource block.

The changes performed are quite self-explanatory, notice that in this version the password of the DB is no more hard-coded but is specified inside the `db_pwd` Terraform variable. As shown by the listing this variable has the field sensitive set to true and is not defined with a default value. The sensitive flag should be set to true because it prevents Terraform from showing the variable value in the output produced during the `plan` or `apply` operations.

Since the variable does not own a default value, the actual value must be passed via the `terraform plan` command using the `-var` parameter. The listing 6.10 lists the changes performed to the "Build" stage of the pipeline for IaC and IaCSec.

```
...
stage("Build") {
 steps {
  echo "Building"
  withCredentials([...,
   usernamePassword(credentialsId: "terraform-db-credentials
   ", usernameVariable: "DB_USR", passwordVariable: "DB_PWD")
    ]) {
    sh "terraform init -no-color"
    sh "terraform plan -out plan.tfplan -refresh=false -no-
   color -var=db_pwd=\$DB_PWD"
   }
  sh "terraform show -json plan.tfplan > plan.json"
 }
}
...
```

Listing 6.10. Adjustments performed to the pipeline for IaC and IaCSec.

In this new version of the pipeline, the password of the database is stored inside as credentials inside Jenkins. The value is retrieved using the function `withCredentials` and put inside the `DB_PWD` environment variable. The latter is assigned to the `db_pwd` variable using the `-var` parameter of the `terraform plan` command.

6.4.4 Cloud Custodian violations

In this case, there are no violations. This is because the custom policies produced for Cloud Custodian are already enforced during the infrastructure deployment phase by the IaCSec tools. However, this does not make the CSPM controls useless since is still possible to elude the IaCSec controls by changing the infrastructure configuration after the deployment.

6.4.5 OPA Gatekeeper violations

The listing 6.11 presents the first definition of the sample application Pod. Note that it contains some hard-coded sensitive environment variables and therefore it is not compliant with the custom policy developed in the PoC.

```
apiVersion: v1
kind: Pod
metadata:
 name: demo-app
 labels:
  purpose: demonstrate-envars
spec:
 containers:
  - name: demo-app
     image: 122333444455555.dkr.ecr.eu-west-3.amazonaws.com/
    demo-pac:tag
     env:
      - name: DB_URI
         value: <DB_URI>
      - name: DB_USR
         value: user
      - name: DB_PWD
         value: password
 restartPolicy: Never
```

Listing 6.11. First definition of the sample application Pod.

The values of the `DB_USR` and `DB_PWD` environment variables are sensible since they are employed by the sample application for authenticating with the RDS DB instance. Therefore, they should not be hard-coded.

```
Error from server (Forbidden): error when creating "target/
   app.yaml": admission webhook "validation.gatekeeper.sh"
   denied the request: [demo-app] The following environment
   variables for the container demo-app are hardcoded: {"
   DB_PWD", "DB_USR"}
```

Listing 6.12. Output of the `kubectl apply` command while passing to it a Pod definition with sensitive environment variables hard-coded in it.

6.4.6 CWPP mitigations

Since the custom CWPP policy prevents the DB credentials from being hard-coded into the Pod definition, the first action that has to be performed is to modify the file which contains it.

The listing 6.13 contains the changes performed for making the Pod compliant with the custom policy.

```
...
  env:
  - name: DB_URI
     value: <DB_URI>
```

```
5   - name: DB_USR
6     valueFrom:
7       secretKeyRef:
8         name: db-user-pass
9         key: user
10  - name: DB_PWD
11    valueFrom:
12      secretKeyRef:
13        name: db-user-pass
14        key: password
15 ...
```

Listing 6.13. Adjustments performed to the YAML file which contain the Pod definition.

This new definition gathers the values for the `DB_USR` and `DB_PWD` environment variables directly from a Kubernetes secret [93] called `db-user-pass`. A Kubernetes Secret is an object that contains a sensitive value associated with a key. This secret is created by a new step inserted inside the `Deploy app` stage of the Jenkins pipeline in charge of deploying the sample application. The new step is shown by the listing 6.14

```
1 withCredentials([..., usernamePassword(credentialsId: "
    terraform-db-credentials", usernameVariable: "DB_USR",
    passwordVariable: "DB_PWD")]){
2     sh "kubectl delete secret db-user-pass"
3     sh "kubectl create secret generic db-user-pass --from-
    literal=user=\$DB_USR --from-literal=password=\$DB_PWD"
4 }
```

Listing 6.14. Adjustments performed to the CWPP pipeline.

The credentials are retrieved from the credentials stored inside Jenkins using the block `withCredentials` and put inside the `DB_USR` and `DB_PWD` environment variables. These variables are passed as values for the `--from-literal` parameter of the command `kubectl create secret`.

The parameter allows the definition of a key-value binding. In this case, the keys are called “user” and “password” while the values are the content of the `DB_USR` and `DB_PWD` environment variables. Note that before the secret creation, eventual previous definitions of it are deleted by the `kubectl delete secret` command.

Chapter 7

Testing and results

The objective of this chapter is to analyze and test the PaC tools utilized inside the PoC. This analysis is conducted for testing the performance impact and the effectiveness of the various PaC tools over the target infrastructure. More specifically the analysis starts with an evaluation of the effectiveness of the IaCSec tools. Then, there is an evaluation of their impact on the execution of the CI/CD pipeline used for deploying the infrastructure. As regards CSPM and CWPP tools, the objective is to analyze the resource consumption of the respective controls.

7.1 Test environment

All tests are conducted on a Lenovo Ideapad 530S-15IBK using a Linux virtual machine (VM). The VM is configured with 8 GB of ram, 2 CPU cores, and 50 GB of disk space. The OS used is Ubuntu 22.04.1 (Kernel version 5.19.0-35-generic). Tests on the IaCSec tools are performed using a CI/CD pipeline executed by Jenkins 2.375.2. The IaCSec tools posed under evaluation are Regula 2.10.0 and tfsec 1.28.1. The CSPM performance and resource consumption evaluation is performed directly on AWS by deploying an AWS Lambda function using Cloud Custodian 0.9.19. The performance impact of the CWPP is evaluated by monitoring an EKS cluster (Kubernetes 1.25) with a node instance of type "t3a.medium" (2 CPU cores and 4GB of ram). The cluster is employed for executing OPA Gatekeeper 3.11.0 and the sample application. The tools used to perform the monitoring are `fluentd-kubernetes-daemonset` 1.7.3 and `cloudwatch-agent` 1.247358.0b252413.

7.2 IaCSec tools effectiveness evaluation

This evaluation is performed on the first version of the target infrastructure. It aims to measure the effectiveness resulting from the combination of the tools used within the pipeline for IaC and IaCSec. Data are gathered from the reports produced by the corresponding CI/CD pipeline and inserted into an Excel file for being analyzed. Each control result is manually checked to determine whether it is a: true positive, true negative, false positive, or false negative. Also, if there are controls from different tools having the same purpose, their results are associated and classified only once.

A control result is a true positive when it identifies a misconfiguration that effectively exists, otherwise, the result is a false positive. Similarly, a result is a true negative when it indicates the absence of a misconfiguration that is not present, or else the control result is a false negative. Moreover, only the controls coming from the default policy pack of the tools are considered in this evaluation. Hence, the misconfiguration relative to the hard-coded password of the DB instance is classified as a false negative even if there is a custom policy that detects it.
The table 7.1 shows the result of the classification.

Class	**Number of results**
True positives (TP)	10
False negatives (FN)	3
False positives (FP)	3
True negatives (TN)	208

Table 7.1. Result of the classification.

The cardinality of each class is employed to calculate two metrics used for measuring the actual effectiveness derived from the combination of the tools. These parameters are called: “True Positive Rate” (TPR) 7.1 and “False Positive Rate” (FPR) 7.2. These metrics represent the probability that a control result is a true positive or a false positive, respectively. They are defined as follows:

$$TPR = \frac{TP}{TP + FN} \quad (7.1) \qquad FPR = \frac{FP}{FP + TN} \quad (7.2)$$

The table 6.2 shows the actual value of these metrics.

Metric	Value
TP rate	0.769
FP rate	0.014

Table 7.2. TPR and FPR values.

To better understand the results of this evaluation, a graphical representation of the result is constructed. This representation is based on the fact that the couple (FPR, TPR) can be interpreted as a point of a two-dimensional space. Is interesting to note that since both components represent a probability their value is limited between 0 and 1. Therefore, the axis values can be also limited to this range.

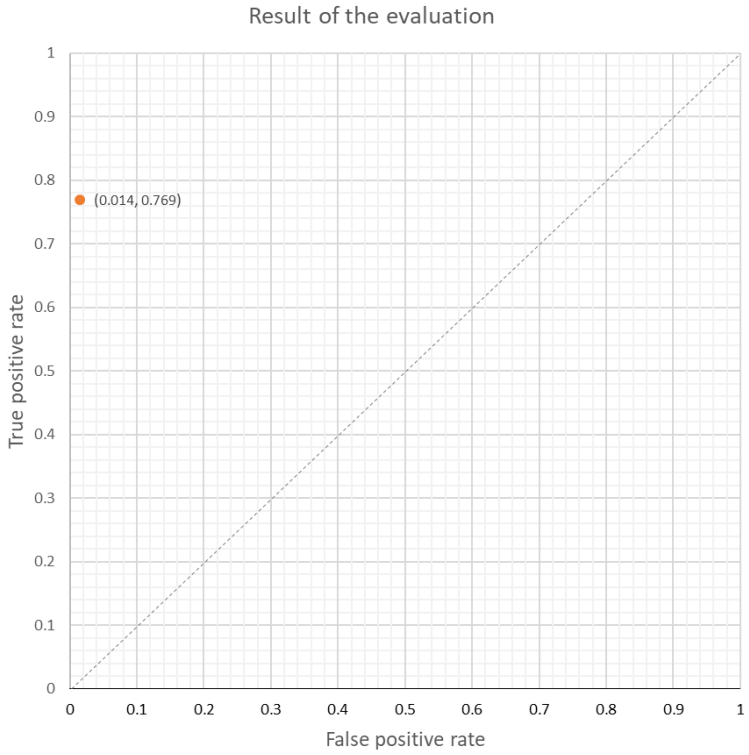

Figure 7.1. Graph representing the result of the evaluation.

A tool aims to maximize the TPR and minimize the FPR; this means that its couple (FPR, TPR) must be as close as possible to the point (0,1). The graph has a diagonal line from the bottom left to the top right corners. If a tool detects misconfigurations randomly, then the point that represents its (FPR, TPR) couple will belong to this line. If a point is above the line, it means that the corresponding tool detects misconfigurations better than randomly. Conversely, if it is below the line, the tool detects misconfigurations worse than randomly.

The graph depicted by the figure 7.1 shows a pretty good result since the point $(0.014, 0.769)$ is above the line and very close to the point with coordinates (0,1).

7.3 IaCSec tools overhead quantification

The quantification is performed by analyzing the data about the overhead caused by the IaCSec tools inside the pipeline. The analysis considers only the execution time of the stage in charge of verifying the infrastructure code. Note that during the multiple executions of the pipeline, the step relative to the infrastructure deployment is not executed since it takes at least ten minutes to terminate and the data about its execution time does not give any additional information.

The overhead is caused by the additional execution time needed for executing the "Verify Iac-sec" stage. These times are gathered by analyzing the results of 126 successful executions of the pipeline for IaC and IaCSec tools. As shown by the figure 7.2 Jenkins output the execution time of each sub-stage of the "Verify Iac-sec" stage, even if they are executed in parallel.

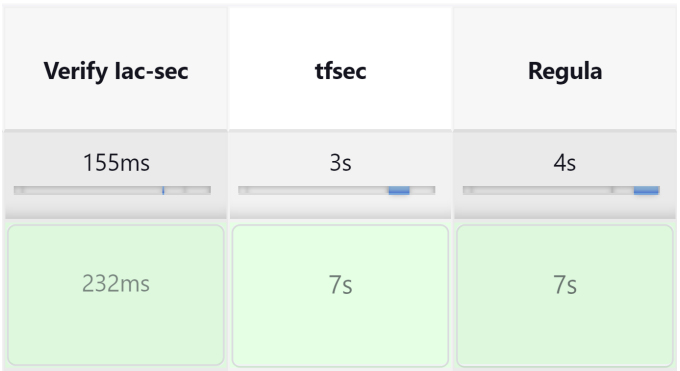

Figure 7.2. Sample of the stage execution times shown by Jenkins.

Hence, for obtaining the effective execution of the "Verify Iac-sec" stage, the execution time recorded by Jenkins for this stage is summed with the maximum between the execution time of the stages "Regula" and "tfsec". The analysis carried out that the average additional execution time needed for performing the stage in charge of performing the IaCSec check is about 4.322s.

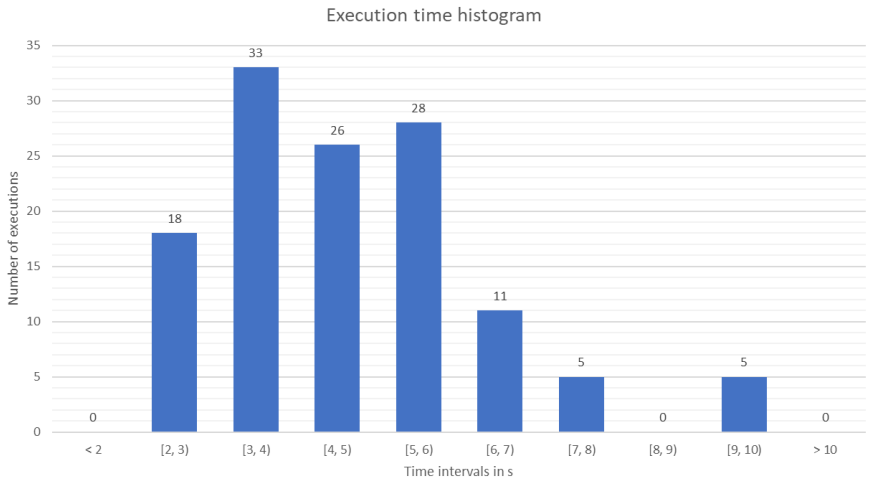

Figure 7.3. Pipeline execution time histogram.

Finally, the histogram represented by the figure 7.3 summarizes the result of the evaluation. Each bar represents the number of executions whose duration falls into the corresponding time interval. The diagram shows that most executions take less than six seconds, a negligible overhead time for a pipeline that is not executed very frequently and whose duration is of several minutes.

7.4 CSPM resource consumption

This analysis aims to quantify the average duration and memory consumption of a sample CSPM control deployed by Cloud Custodian. The control under examination is the periodic custom policy developed for the proof-of-concept. The data about the relevant metric are collected using the Amazon CloudWatch service [94]. This service allows getting and querying information from the logs produced by the execution of the lambda function in charge of enforcing the control. In this analysis, the data collected from the logs are for the duration and memory consumption of 101 lambda executions.

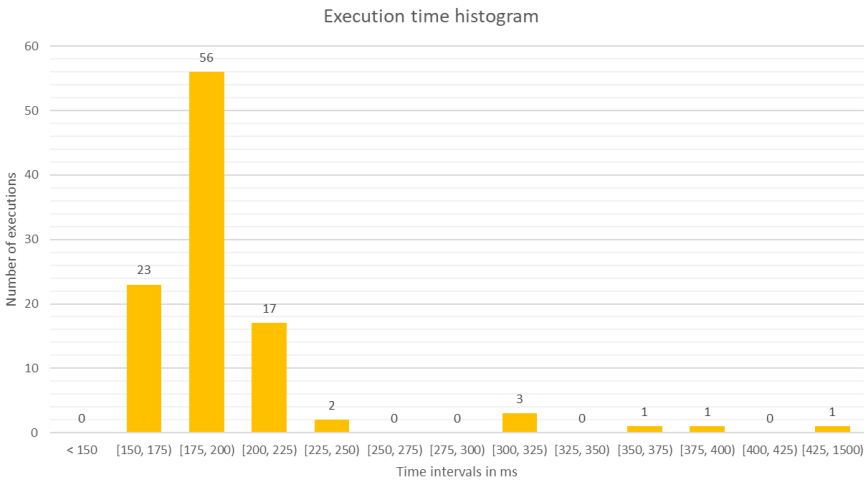

Figure 7.4. Lambda execution time histogram.

The histogram shown by the figure 7.4 is the result of the analysis over the duration of the lambda function invocations and represents the distribution of the execution times of each invocation. Note that more than half of the executions fall in the interval between 175 ms and 200 ms although the average duration is 205.73 ms.

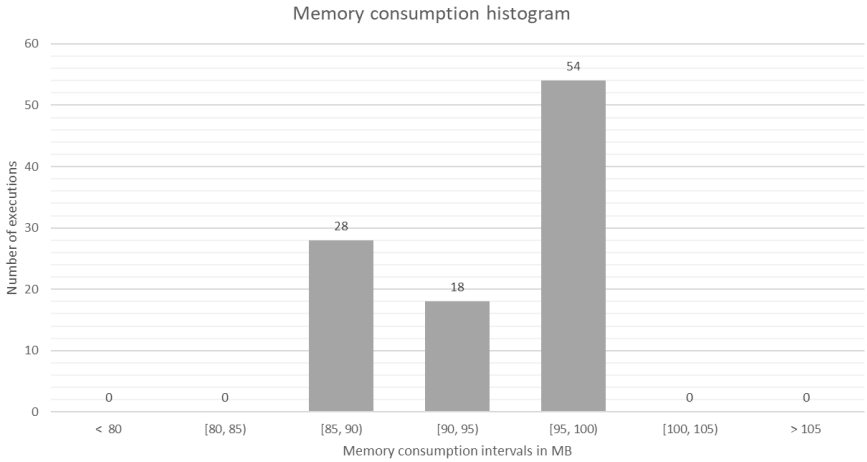

Figure 7.5. Lambda memory consumption histogram.

Finally, the diagram illustrated by the figure 7.5 derives from the evaluation of the memory consumption of each invocation. In this case, the average memory consumption is 95.052 MB and more than half of the executions fall into the interval between 95 MB and 100 MB.

7.5 CWPP resource consumption

The purpose of this analysis is to verify the effective resource consumption of OPA Gatekeeper and identify its impact on the overall performance. Also, in this case, the data for this investigation are gathered using the Amazon CloudvWatch service. Unfortunately, is not possible to monitor the cluster directly with CloudWatch. Hence, the monitoring is performed via the deployment of some Kubernetes resources whose job is to collect and send metrics to CloudWatch. These metrics are collected using Fluentd [95] and sent to CloudWatch by the CloudWatch agent [96].

7.5.1 How the monitoring was performed

The monitoring covers the cluster's network utilization and the CPU and memory utilization by the cluster, namespaces, and services. The analysis lasted about two hours and thirty minutes during which two versions of the sample application and OPA Gatekeeper were run within the cluster. The two versions of the sample applications were deployed using two different configuration files. One complaint with the custom CWPP policy developed for the proof-of-concept and one not.

To create a base reference, the cluster has been run for forty minutes without OPA Gatekeeper. In this way is possible to compare the resource consumption before and after the deployment of the tool.

Finally, during the last thirty minutes, some tests were performed to verify the behavior of OPA Gatekeeper and the cluster when multiple deployment requests are sent for one or more non-compliant resources. More specifically, three groups of requests were sent at three different times. Each group consisted of ten requests sent in parallel. What differentiates the various groups is the number of non-compliant resources contained within the requests. In the first group, each request contains only one non-compliant resource. The second contains a series of requests with exponentially increasing numbers of non-compliant resources. The first request contains one resource, the second two, and so on until the last one contains 1024 resources. The last group is composed of requests with 1024 non-compliant resources each.

7.5.2 Results

The graph shown by figure 7.6 illustrates the trend over time of CPU and memory utilization by the cluster. From this point of view, the CPU utilization seems to be unaffected by OPA Gatekeeper, unlike the memory consumption which seems to have increased slightly after the deployment of the tool.

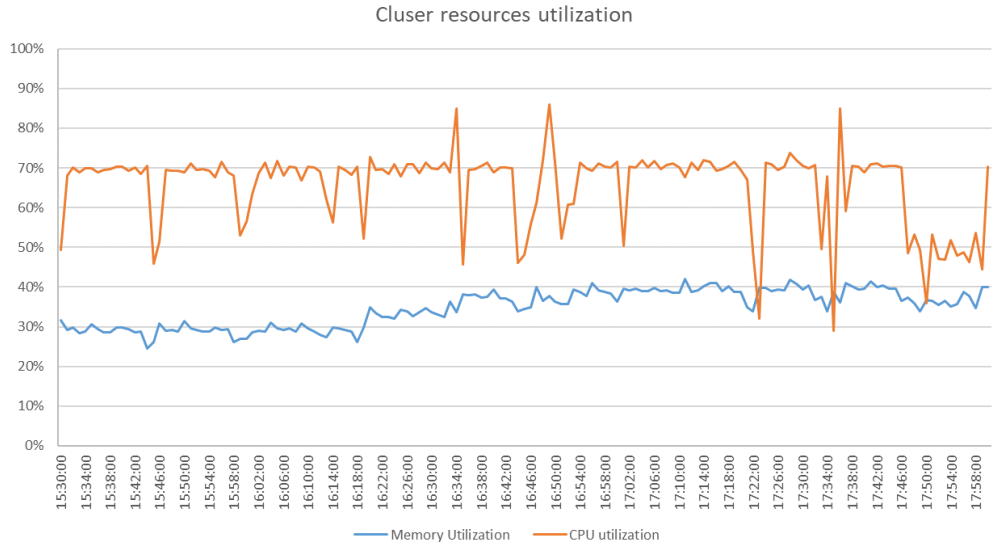

Figure 7.6. Cluster resource utilization graph.

Analyses performed on the data resulting from monitoring show that resource consumption of OPA Gatekeeper appears to be practically negligible. Indeed, the "Namespace memory utilization" graph depicted in the figure 7.7 highlights that the memory consumption of resources belonging to the namespace `gatekeeper-system` is always below 1%. Moreover, the graph referring to the CPU consumption shows a quite similar result except for some peaks.

These phenomena seem to be related to the ones shown by the "Cluster network utilization" graph, whose peaks are generated by the submission of the groups of requests defined previously. Note that memory consumption also has peaks at request forwarding, but they appear to be much lower and smoother than those shown by the "Namespace CPU utilization" graph.

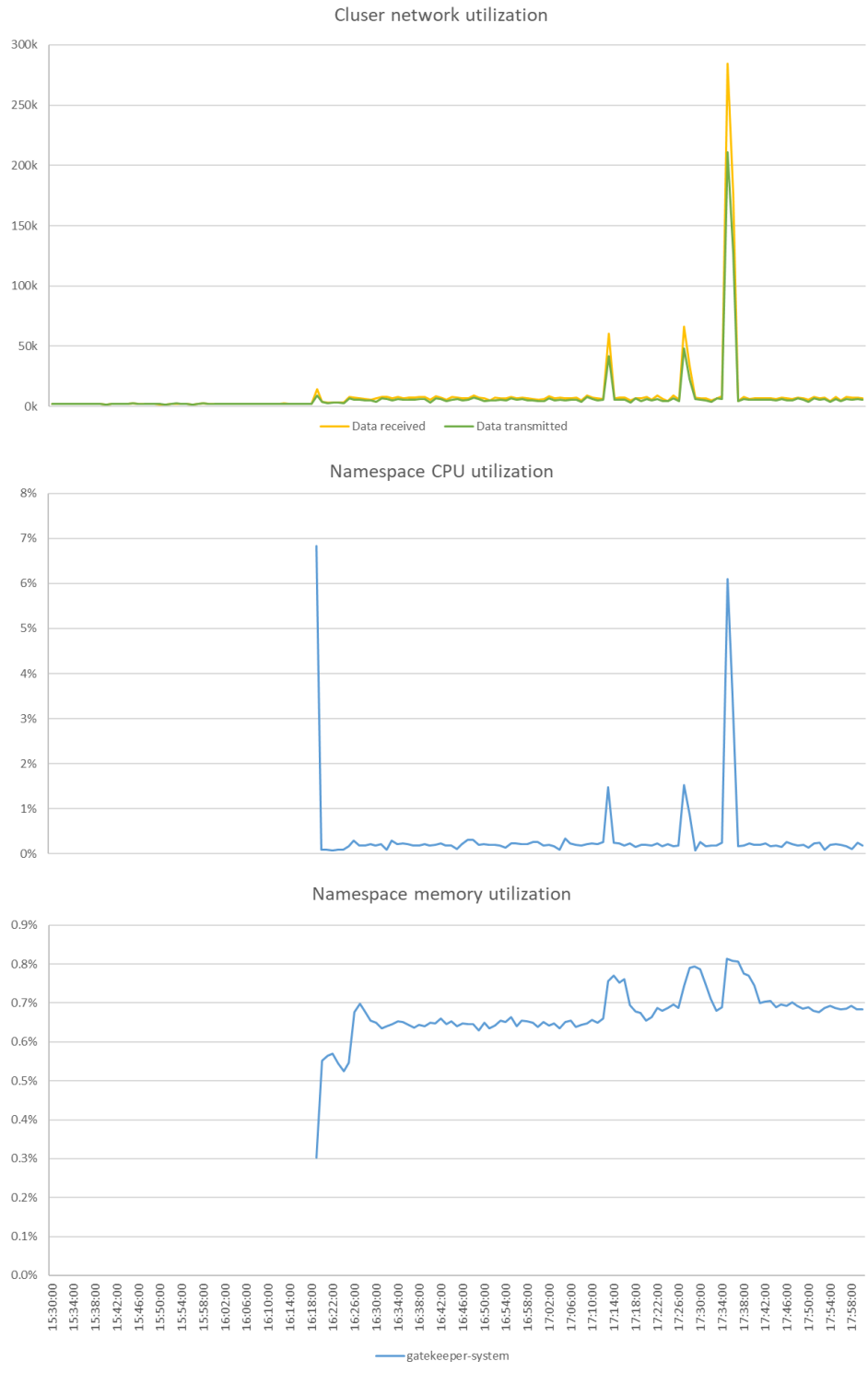

Figure 7.7. Cluster network utilization and resource consumption of the `gatekeeper-system` namespace.

The graphs depicted by the figure 7.8 show the resource consumption of the `gatekeeper-webhook-service`. As expected they follow the trend described by the "Cluster network utilization" graph, since this service is in charge of intercepting and analyzing the request directed to the Kubernetes API Server.

Finally, it is interesting to note that the line plotted on this graph is practically overlapping with the line shown by the graph on CPU consumption. From this, it can be deduced that the `gatekeeper-webhook-service` turns out to be the resource with the greatest impact on OPA Gatekeeper's total resource consumption.

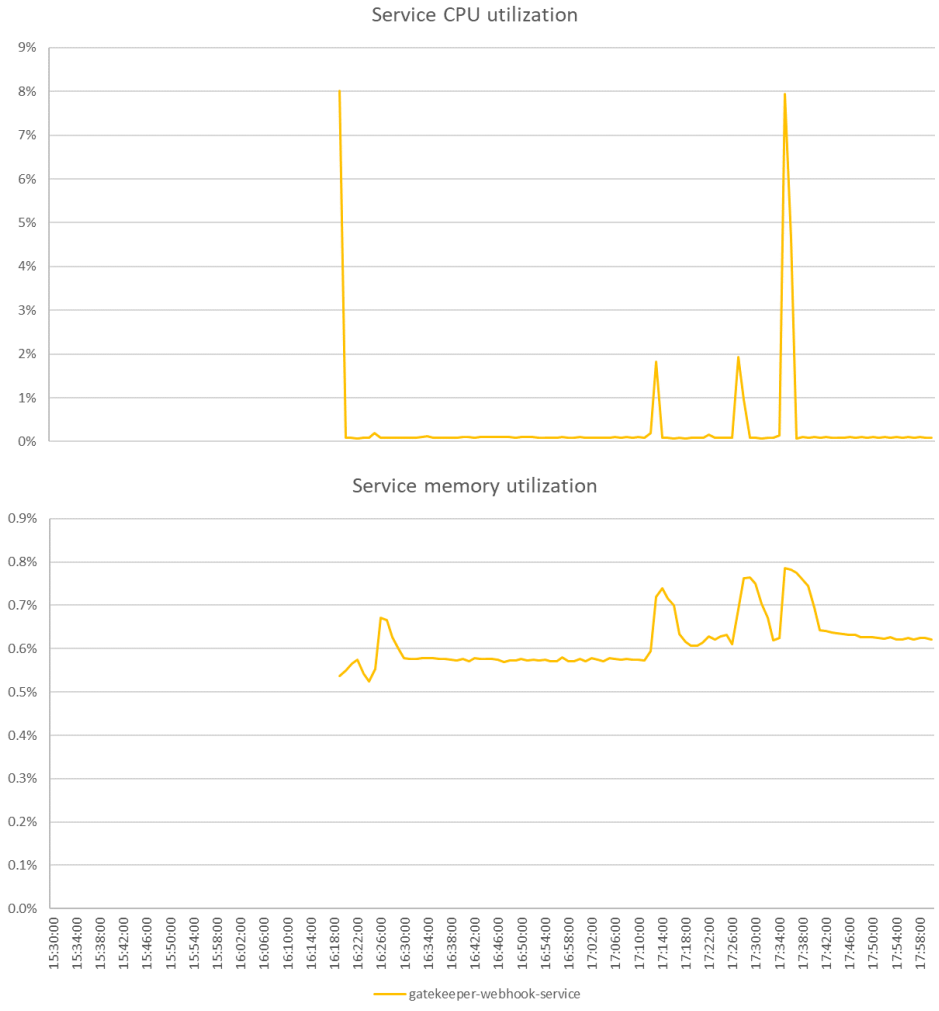

Figure 7.8. Resource utilization of the `gatekeeper-webhook-service`.

Chapter 8

Conclusions and future works

This thesis has shown how compliance automation can be implemented in a cloud environment using Policy as Code approach.

It began by defining basic concepts such as cloud computing, cloud compliance, and policy. This work has explored the state of the art of Policy as Code and how it is useful for implementing Infrastructure as Code Security, Cloud Workload Protection and Cloud Security Posture Management. Moreover, some other concepts akin to DevOps such as infrastructure as code and CI/CD pipelines were discussed. The work investigated the open-source solution available for deploying a proof-of-concept utilized for demonstrating the potentialities of the selected tools.

The set of tools employed during the development of the proof-of-concept demonstrated that Policy as Code approach works and can be integrated inside a DevSecOps methodology.

The CI/CD pipelines developed during this study permitted the identification of various policy violations in the default configuration of the proof-of-concept's cloud infrastructure and sample application. These policy violations are already present and are not inserted on purpose but detected by the tools. Therefore, the execution of the pipeline was helpful in identifying these misconfigurations.

The pipeline in charge of deploying the proof-of-concept's cloud infrastructure discovered various misconfigurations in the infrastructure code. These misconfigurations were identified by the Infrastructure as Code Security tools used directly within the pipeline.

The one produced to deploy the sample proof-of-concept application and

Cloud Workload Protection checks within the Kubernetes cluster, identified a policy violation in the resource definition used to deploy the application.

The pipeline responsible for the implementation of Cloud Security Posture Management controls made it possible to monitor the compliance status of the cloud infrastructure.

Finally, some performance evaluations were carried out. The one for Infrastructure as Code Security tools proved their quite good effectiveness as well as their negligible impact on the pipeline execution time. The evaluation against a sample Cloud Security Posture Management control showed a low memory consumption and execution time. This is a good result given that periodic and event-based controls are performed within Amazon Lambda functions that are billed based on memory consumption and execution time. The last evaluation was performed on the Kubernetes cluster and demonstrated the low impact on the cluster resource consumption of the Cloud Workload Protection tool.

8.1 Future works

Some aspects dealt with in this thesis can be further analyzed and used as hints for future works.

First, the Policy as Code tools presented in this study were tested only on an AWS environment. Since they are also compatible with Microsoft Azure [42] and Google Cloud [43], it would be interesting to understand how the tools behave when they operate in these cloud environments.

Also, the proof-of-concept can be further refined to emulate a real production environment. In this way is possible to analyze more accurately the impact of Policy as Code in the cloud compliance automation. Moreover, this study only covered the discovery of policy violations, but not how they can be notified once detected.

Furthermore, both Cloud Security Posture Management policies and the Infrastructure as Code Security ones are related to the cloud infrastructure but enforced at different moments. It might make sense to find a way to verify that each Infrastructure as Code Security policy has a corresponding Cloud Security Posture Management policy.

Finally, this thesis has only addressed open-source solutions; it might be interesting to explore proprietary ones and compare them with the solutions discussed in this work.

Bibliography

[1] Mell, P. and Grance, T. (2011), "The NIST Definition of Cloud Computing, Special Publication (NIST SP)," National Institute of Standards and Technology, Gaithersburg, MD, 2011, DOI: 10.6028/NIST.SP.800-145

[2] "Public Cloud - Worldwide", Statista. [Online]. Available: `https://www.statista.com/outlook/tmo/public-cloud/worldwide`

[3] A. Hendre and K. P. Joshi, "2015 IEEE 8th International Conference on Cloud Computing," New York, NY, USA, 2015, pp. 1081-1084, DOI: 10.1109/CLOUD.2015.157

[4] Sangkyun Kim, "IT compliance of industrial information systems: Technology management and industrial engineering perspective," Journal of Systems and Software, Volume 80, Issue 10, 2007, Pages 1590-1593, ISSN 0164-1212, DOI: 10.1016/j.jss.2007.01.016

[5] EU, "EU General Data Protection Regulation (GDPR)," 2017. [Online]. Available: `https://gdpr.eu/article-83-conditions-for-imposing-administrative-fines/`

[6] L. Johnson (NIST), Kelley Dempsey (NIST), Ron Ross (NIST), Sarbari Gupta (Electrosoft Services), Dennis Bailey (Electrosoft Services) "Guide for Security-Focused Configuration Management of Information Systems," Special Publication (NIST SP), National Institute of Standards and Technology, Gaithersburg, MD, 2011 DOI: 10.6028/NIST.SP.800-128

[7] Jangirala Srinivas, Ashok Kumar Das, Neeraj Kumar, "Government regulations in cyber security: Framework, standards and recommendations," Future Generation Computer Systems, Volume 92, 2019, Pages 178-188, ISSN 0167-739X, DOI: 10.1016/j.future.2018.09.063

[8] European Union Agency for Cybersecurity, Drogkaris, P., Bourka, A., "Guidance and gaps analysis for European standardisation : privacy

standards in the information security context," Drogkaris, P. (editor), Bourka, A. (editor), European Network and Information Security Agency, 2019, DOI: 10.2824/698562

[9] Scarfone, K. , Benigni, D. and Grance, T. (2009), "Cyber Security Standards" Wiley Handbook of Science and Technology for Homeland Security, John Wiley & Sons, Inc., Hoboken, NJ, [online]. Available: https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=152153

[10] Grance, T. and Jansen, W. (2011), "Guidelines on Security and Privacy in Public Cloud Computing," Special Publication (NIST SP), National Institute of Standards and Technology, Gaithersburg, MD, 2011 DOI: 10.6028/NIST.SP.800-144

[11] Robert Amor, Johannes Dimyadi, "The promise of automated compliance checking," Developments in the Built Environment, Volume 5, 2021, 100039, ISSN 2666-1659, DOI: 10.1016/j.dibe.2020.100039

[12] Liu, F. , Tong, J. , Mao, J. , Bohn, R. , Messina, J. , Badger, M. and Leaf, D. (2011), "NIST Cloud Computing Reference Architecture," Special Publication (NIST SP), National Institute of Standards and Technology, Gaithersburg, MD, 2011, DOI: 10.6028/NIST.SP.500-292

[13] "Shared Responsibility Model Explained," Cloud Security Alliance, 2020. [Online]. Available: https://cloudsecurityalliance.org/blog/2020/08/26/shared-responsibility-model-explained/

[14] H. Tianfield, "Security issues in cloud computing," 2012 IEEE International Conference on Systems, Man, and Cybernetics (SMC), Seoul, Korea (South), 2012, pp. 1082-1089, DOI: 10.1109/ICSMC.2012.6377874

[15] "Shared Responsibility for Cloud Security: What You Need to Know," Center for Internet Security. [Online]. Available: https://www.cisecurity.org/insights/blog/shared-responsibility-cloud-security-what-you-need-to-know

[16] "Shared Responsibility Model," Amazon Web Services. [Online]. Available: https://aws.amazon.com/compliance/shared-responsibility-model

[17] "Philosophy," Open Policy Agent, [Online]. Available: https://www.openpolicyagent.org/docs/latest/philosophy/

[18] "policy," Cambridge Dictionary, [Online]. Available: https://dictionary.cambridge.org/dictionary/english/policy

[19] W. Seaton, "What is Policy as Code," Styra [Online]. Available:

https://www.styra.com/blog/
what-is-policy-as-code-definition-and-benefits/
[20] Y. Matharu, T. Coulter, "Introduction to policy as code with automation," Red Hat [Online]. Available:
https://www.redhat.com/sysadmin/policy-as-code-automation
[21] "Policy as Code," HashiCorp [Online]. Available: https:
//docs.hashicorp.com/sentinel/concepts/policy-as-code
[22] S. Jones, J. Noppen, and F. Lettice, "Management challenges for DevOps adoption within UK SMEs," In Proceedings of the 2nd International Workshop on Quality-Aware DevOps (QUDOS 2016), Association for Computing Machinery, New York, NY, USA, 7–11, 2016, DOI: 10.1145/2945408.2945410
[23] R. Jabbari, N. Ali, K. Petersen, and B. Tanveer, "What is DevOps? A Systematic Mapping Study on Definitions and Practices," In Proceedings of the Scientific Workshop Proceedings of XP2016 (XP '16 Workshops). Association for Computing Machinery, New York, NY, USA, Article 12, 1–11, 2016, DOI: 10.1145/2962695.2962707
[24] C. A. Cois, J. Yankel and A. Connell, "Modern DevOps: Optimizing software development through effective system interactions," 2014 IEEE International Professional Communication Conference (IPCC), Pittsburgh, PA, USA, 2014, pp. 1-7, DOI: 10.1109/IPCC.2014.7020388
[25] H. Dhaduk, "DevOps Lifecycle: 7 Phases Explained in Detail with Examples," Simform, [Online]. Available:
https://www.simform.com/blog/devops-lifecycle/
[26] T. Tran, "Disadvantages of DevOps? Why Is It Challenging?," Orient Software Development Corp, [Online]. Available:
https://www.orientsoftware.com/blog/
advantages-and-disadvantages-of-devops/
[27] "What is a CI/CD pipeline?," Red Hat [Online]. Available:
https://www.redhat.com/en/topics/devops/
what-cicd-pipeline?cicd=32h281b
[28] "What is Infrastructure as Code (IaC)?," Red Hat [Online]. Available:
https://www.redhat.com/en/topics/automation/
what-is-infrastructure-as-code-iac
[29] "What is DevSecOps?," Red Hat [Online]. Available:
https://www.redhat.com/en/topics/devops/what-is-devsecops
[30] "DevSecOps Overview," Snyk [Online]. Available:
https://snyk.io/series/devsecops/

[31] P. Bhandari "Infrastructure as Code Security and Best Practices | A Quick Guide," XENONSTACK [Online]. Available: https://www.xenonstack.com/insights/infrastructure-as-code-security

[32] "Unit 42 Cloud Threat Report, Vol. 2," Palo Alto Networks [Online]. Available: https://www.paloaltonetworks.com/resources/research/cloud-threat-report-spring-2020

[33] "Infrastructure as Code in a DevSecOps World" Snyk [Online]. Available: https://snyk.io/learn/infrastructure-as-code-iac/

[34] "Cloud security posture management explained," Snyk [Online]. Available: https://snyk.io/series/cloud-security/posture-management-cspm/

[35] "What is Cloud Workload Protection?," VMware [Online]. Available: https://www.vmware.com/topics/glossary/content/workload-protection.html

[36] "What is a Cloud Workload Protection Platform (CWPP)?," Check Point Software Technologies [Online]. Available: https://www.checkpoint.com/cyber-hub/cloud-security/what-is-a-cloud-workload-protection-platform-cwpp/

[37] "Jenkins User Documentation," Jenkins [Online]. Available: https://www.jenkins.io/doc/

[38] "Continuous Integration Software Market Share," Datanyze, Last visit: 19/02/2023, [Online]. Available: https://www.datanyze.com/market-share/ci--319

[39] "Pipeline," Jenkins [Online]. Available: https://www.jenkins.io/doc/book/pipeline/

[40] "What is Terraform?," HashiCorp [Online]. Available: https://developer.hashicorp.com/terraform/intro

[41] "Amazon Web Services," Amazon Web Services [Online]. Available: https://aws.amazon.com/

[42] "Azure," Microsoft [Online]. Available: https://azure.microsoft.com/en-us/

[43] "Google Cloud," Google [Online]. Available: https://cloud.google.com/

[44] "Terraform Language Documentation," HashiCorp [Online]. Available: https://developer.hashicorp.com/terraform/language

[45] "State," HashiCorp [Online]. Available: https://developer.hashicorp.com/terraform/language/state

[46] "Introduction," Open Policy Agent [Online]. Available: https://www.openpolicyagent.org/docs/latest/

[47] "Policy Language," Open Policy Agent [Online]. Available: https://www.openpolicyagent.org/docs/latest/policy-language/
[48] "Home," Aqua Security [Online]. Available: https://aquasecurity.github.io/tfsec/v1.28.1/
[49] "Checks," Aqua Security [Online]. Available: https://aquasecurity.github.io/tfsec/v1.28.1/checks/aws/api-gateway/enable-access-logging/
[50] "Welcome to the Regula Docs!," Fugue [Online]. Available: https://regula.dev/index.html
[51] "Introduction," Gatekeeper [Online]. Available: https://open-policy-agent.github.io/gatekeeper/website/docs/
[52] "Cloud Custodian Documentation," Cloud Custodian [Online]. Available: https://cloudcustodian.io/docs/index.html
[53] "Writing Custom Rego Policies," Aqua Security [Online]. Available: https://aquasecurity.github.io/tfsec/v1.28.1/guides/rego/rego/
[54] "Writing Rules," Fugue [Online]. Available: https://regula.dev/development/writing-rules.html
[55] "OPA Constraint Framework," Open Policy Agent [Online] https://github.com/open-policy-agent/frameworks/tree/master/constraint
[56] "How to use Gatekeeper," Gatekeeper [Online]. Available: https://open-policy-agent.github.io/gatekeeper/website/docs/howto
[57] "Getting Started," Cloud Custodian [Online]. Available: https://cloudcustodian.io/docs/aws/gettingstarted.html#write-your-first-policy
[58] "Overview," Kubernetes [Online]. Available: https://kubernetes.io/docs/concepts/overview/
[59] "Most wanted cloud platform among developers worldwide as of 2022," Statista. [Online]. Available: https://www.statista.com/statistics/793884/worldwide-developer-survey-most-wanted-platform/
[60] "AWS Documentation," Amazon Web Services [Online]. Available: https://docs.aws.amazon.com/
[61] "AWS Regions and Availability Zones," Amazon Web Services [Online]. Available: https://docs.aws.amazon.com/whitepapers/latest/get-started-documentdb/aws-regions-and-availability-zones.html

[62] "Module Blocks," HashiCorp [Online]. Available: https://developer.hashicorp.com/terraform/language/modules/syntax
[63] HashiCorp [Online]. Available: https://developer.hashicorp.com/terraform/language/resources
[64] HashiCorp [Online]. Available: https://developer.hashicorp.com/terraform/language/data-sources
[65] HashiCorp [Online]. Available: https://developer.hashicorp.com/terraform/language/values/variables
[66] "AWS VPC Terraform module," HashiCorp [Online]. Available: https://registry.terraform.io/modules/terraform-aws-modules/vpc/aws/latest
[67] "Amazon EKS VPC and subnet requirements and considerations," Amazon Web Services [Online]. Available: https://docs.aws.amazon.com/eks/latest/userguide/network_reqs.html
[68] "AWS EKS Terraform module," HashiCorp [Online]. Available: https://registry.terraform.io/modules/terraform-aws-modules/eks/aws/latest
[69] "Control traffic to resources using security groups," Amazon Web Services [Online]. Available: https://docs.aws.amazon.com/vpc/latest/userguide/VPC_SecurityGroups.html
[70] "IAM roles," Amazon Web Services [Online]. Available: https://docs.aws.amazon.com/IAM/latest/UserGuide/id_roles.html
[71] "AWS Systems Manager Session Manager," Amazon Web Services [Online]. Available: https://docs.aws.amazon.com/systems-manager/latest/userguide/session-manager.html
[72] "What are VPC endpoints?," Amazon Web Services [Online]. Available: https://docs.aws.amazon.com/whitepapers/latest/aws-privatelink/what-are-vpc-endpoints.html
[73] "AWS VPC Endpoints Terraform sub-module," HashiCorp [Online]. Available: https://registry.terraform.io/modules/terraform-aws-modules/vpc/aws/latest/submodules/vpc-endpoints
[74] "What is Amazon CloudWatch Logs?" Amazon Web Services [Online]. Available: https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/WhatIsCloudWatchLogs.html
[75] "Resource: aws_db_instance," HashiCorp [Online]. Available: https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/db_instance
[76] "Resource: aws_security_group" HashiCorp [Online]. Available:

https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/security_group
[77] "Resource: aws_security_group_rule" HashiCorp [Online]. Available: https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/security_group_rule
[78] "Backend Configuration" HashiCorp [Online]. Available: https://developer.hashicorp.com/terraform/language/settings/backends/configuration
[79] "Provider Configuration," HashiCorp [Online]. Available: https://developer.hashicorp.com/terraform/language/providers/configuration
[80] "Basic CLI Features," HashiCorp [Online]. Available: https://developer.hashicorp.com/terraform/cli/commands
[81] "Command: init," HashiCorp [Online]. Available: https://developer.hashicorp.com/terraform/cli/commands/init
[82] "Provisioning Infrastructure with Terraform," HashiCorp [Online]. Available: https://developer.hashicorp.com/terraform/cli/run
[83] "Credentials Binding," Jenkins [Online]. Available: https://plugins.jenkins.io/credentials-binding/
[84] "Using credentials," Jenkins [Online]. Available: https://www.jenkins.io/doc/book/using/using-credentials/
[85] "kustomize," GitHub [Online]. Available: https://github.com/kubernetes-sigs/kustomize
[86] "Apply," Kubernetes [Online]. Available: https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#apply
[87] "Amazon Web Services," Center for Internet Security [Online]. Available: https://www.cisecurity.org/benchmark/amazon_web_services
[88] "What is AWS Lambda?," Amazon Web Services [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/welcome.html
[89] "What Is Amazon EventBridge?," Amazon Web Services [Online]. Available: https://docs.aws.amazon.com/eventbridge/latest/userguide/eb-what-is.html
[90] "Amazon Elastic Block Store (Amazon EBS)" Amazon Web Services [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AmazonEBS.html
[91] "Ignoring Checks," Aqua Security [Online]. Available: https://aquasecurity.github.io/tfsec/v1.28.1/guides/configuration/ignores/

[92] "Configuring Regula," Fugue [Online]. Available: `https://regula.dev/configuration.html`

[93] "Secrets," Kubernetes [Online]. Available: `https://kubernetes.io/docs/concepts/configuration/secret/`

[94] "What is Amazon CloudWatch?" Amazon Web Services [Online]. Available: `https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/WhatIsCloudWatch.html`

[95] "What is Fluentd?," FLuentd [Online]. Available: `https://www.fluentd.org/architecture`

[96] "Quick Start setup for Container Insights on Amazon EKS and Kubernetes," Amazon Web Services [Online]. Available: `https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/WhatIsCloudWatch.html`