José Gabriel Couceiro de Carvalho Machado

ASSURANCE AND COMPLIANCE OF SECURITY POLICIES IN CLOUD NATIVE ENVIRONMENTS

Dissertation in the context of the Master's in Informatics Engineering, advised by Professor Dr. Nuno Antunes and presented to the Department of Informatics Engineering of the Faculty of Science and Technology of the University of Coimbra.

July 2022

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D
COIMBRA
DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

José Gabriel Couceiro de Carvalho Machado

Assurance and Compliance of Security Policies in Cloud Native Environments

Dissertação no âmbito do Mestrado em Engenharia Informática, orientada pelo Professor Doutor Nuno Antunes e apresentada ao Departamento de Engenharia Informática da Faculdade de Ciências e Tecnologia da Universidade de Coimbra.

Julho, 2022

Page intentionally left blank

Acknowledgment

I would like to express my gratitude to Professor Nuno Antunes which, besides being my advisor for this thesis, introduced me to many security subjects throughout the Cybersecurity Master. It has been quite a journey until now and, when looking back, I can see how much I have done and achieved so far. Thank you also for all the support, criticism, and feedback, which were always crucial for improving all my outcomes.

Thank you, Marco Amador, it was a pleasure to be your advisee and learn with all the knowledge you have to share. It has been a very important experience that surely contributed to the increase of my interest in this theme.

I want to thank my family for always motivating me and giving me the support I needed to make my decisions, either inside or outside my academic path. You have always been by my side.

Also, I want to thank my girlfriend, Mariana. These last years have been very tough, but you’ve made them not look that bad. Your cheerfulness in every situation I faced and how you always try to understand and support me in any way you can, is a treasure that I cannot value enough. You have been crucial to my efforts and my work.

A special thanks to all my friends for always sharing their knowledge with me and always helping me grow.

Finally, I want to thank ANOVA for giving me the opportunity to work in a good enterprise environment and for their flexibility and support.

Resumo

A computação baseada na Cloud pode ser vista como a entrega de serviços hospedados na internet, nomeadamente software, hardware e armazenamento, pela Internet. As vantagens de uma implementação rápida, flexibilidade, custos iniciais baixos e escalabilidade tornaram a Cloud Computing virtualmente presente entre organizações de todo o tipo de dimensão.

Com um grande avanço na tecnologia, também são trazidos múltiplos potenciais riscos para as organizações e é aí que entra a segurança na Cloud. A segurança na Cloud é uma responsabilidade compartilhada entre o provedor dos serviços Cloud e o cliente. Refere-se às tecnologias, políticas, controlos e serviços que protegem os dados, as aplicações e a infraestrutura na Cloud contra possíveis ameaças.

Uma das melhores medidas para aprimorar a segurança na Cloud, é a definição de políticas de segurança. As políticas de segurança ajudam a minimizar o risco de fuga ou perda de dados, bem como protegem contra possíveis utilizadores internos ou externos, mal-intencionados. Estas políticas também definem diretrizes, práticas recomendadas e ajudam a garantir a conformidade.

Este trabalho apresenta uma pesquisa acerca das melhores ferramentas/serviços disponíveis para definir políticas de segurança para um ambiente Cloud-Native, uma pesquisa acerca das políticas de segurança mais adequadas a serem implementadas para este cenário, de forma a aprimorar a segurança no ambiente, assim como o processo de configuração necessário para a integração das políticas de segurança com a infraestrutura existente.

Palavras-Chave

Cloud-Native, Cloud, Segurança na Cloud, Políticas de Segurança, Conformidade, Política-como-Código, Azure Policy.

Page intentionally left blank

Abstract

Cloud computing can be seen as the delivery of hosted services, including software, hardware, and storage, over the internet. The advantages of rapid deployment, flexibility, low upfront costs, and scalability have made cloud computing virtually present among organizations of all sizes.

Of course, with this huge advance in technology, also comes a lot of potential risks for organizations and that's where cloud-security comes in. Cloud security is a shared responsibility between the cloud provider and the customer. It refers to the technologies, policies, controls, and services that protect data, applications, and cloud infrastructure from threats.

One of the best measures to improve cloud security is to define security policies across the technology stack. Security policies will minimize the risk of data leak or data loss as well as protect against potential malicious internal and external users. These policies also set guidelines, best practices, and help ensure compliance.

This work presents a research about the best tools/services to define security policies for a Cloud-Native environment as well as a research about the most significant and suitable security policies to be implemented for this scenario in order to improve security in the environment. It also presents the necessary process for the integration of security policies with the existing infrastructure. The entire process is covered, from the process of defining policies to its assignment and functionality validation.

Keywords

Cloud-Native, Cloud Security, Cloud, Security Policies, Compliance, Policy-as-Code, Azure Policy.

Page intentionally left blank

Table of Contents

Chapter 1 Introduction 19

1.1 Goals 20

1.2 Structure 21

Chapter 2 Context 23

2.1 Cloud-Native 23

2.1.1 Architecture 24
2.1.2 Technologies 25

2.2 Policy as Code 26

2.2.1 What is Policy as Code? 26
2.2.2 How it works? 26
2.2.3 Types of Policies 27
2.2.4 Benefits 28
2.2.5 Use Cases 30
2.2.6 Conclusion 31

Chapter 3 Environment Architecture 32

3.1 Technology Stack 34

3.1.1 App Definition and Development 34
3.1.2 Orchestration and Management 35
3.1.3 Runtime 35
3.1.4 Provisioning 35
3.1.5 Observability and Analysis 36

Chapter 4 State of Art 37

4.1 Open Policy Agent 37

4.1.1 How does OPA works? 37
4.1.2 Rego 38

4.2 Azure Policy 39

4.2.1 Evaluation 39
4.2.2 Response 40
4.2.3 Azure Policy for Kubernetes 40

4.3 Microsoft Defender for Cloud 42

4.4 Tools 43

4.4.1 Falco 43
4.4.2 K-Rail 43
4.4.3 Harbor 44
4.4.4 Conftest 44
4.4.5 Kyverno 44
4.4.6 Gatekeeper 44

4.5 Gatekeeper vs Kyverno for Kubernetes 45

4.6 Conclusion 47

Chapter 5 Policies 48

5.1 What is? **49**

5.2 Benefits **49**

5.3 Kubernetes Oriented Policies **50**

5.3.1 Trusted Repo 50
5.3.2 Block Pod Exec 50
5.3.3 Disallow Add Capabilities 50
5.3.4 Limit Pod Containers 50
5.3.5 Label Safety 50
5.3.6 Handle Privileged Mode 51
5.3.7 Define and Control Ingress 51
5.3.8 Define and Control Egress 51
5.3.9 Allowed Pod Priorities 52
5.3.10 Require Limits and Requests 52
5.3.11 Require Run as Non-Root 52
5.3.12 Spread Pods Across Nodes 52
5.3.13 Kubernetes cluster containers should run with a read only root file system 52

5.4 General Policies **53**

5.4.1 Restrict External IPs 53
5.4.2 Enforce SSL connections for databases 53
5.4.3 Services should use virtual network service endpoints 53
5.4.4 Databases should use customer-managed keys to encrypt data at rest 53
5.4.5 Container registry images should have vulnerability findings resolved 54

5.5 Conclusion **54**

Chapter 6 Implementation and Validation ***55***

6.1 Implementation **56**

6.2 Validation **58**

6.2.1 Kubernetes clusters should not allow privileged containers 58
6.2.2 SSL connection should be enabled for PostgreSQL database servers 59
6.2.3 Kubernetes cluster containers should run with a read only root file system 60
6.2.4 Kubernetes clusters should not allow container privilege escalation 62
6.2.5 Kubernetes cluster services should only use allowed external IPs 63

Chapter 7 Planning ***66***

7.1 First Semester **66**

7.2 Second Semester **67**

7.3 Risk Management **69**

7.4 Methodology **72**

Chapter 8 Conclusion and Future Work ***73***

8.1 Future Work **73**

References ***75***

Appendix A – Policies JSON Files ***79***

Policy 1 - Kubernetes clusters should not allow privileged containers 80
Policy 2 - SSL connection should be enabled for PostgreSQL database servers 82
Policy 3 - Kubernetes cluster containers should run with a read only root file system 83
Policy 4 - Kubernetes clusters should not allow container privilege escalation 85
Policy 5 - Kubernetes cluster services should only use allowed external IPs 87

Acronyms

CNCF – Cloud Native Computing Foundation

API – Application Programming Interface

IP Address – Internet Protocol Address

IIOT – Industrial Internet of Things

OPA – Open Policy Agent

SaaS – Software as a Service

SRE – Site Reliability Engineer

IT – Information Technology

CI – Continuous Integration

CD – Continuous Delivery

PSP – Pod Security Policy

TLS – Transport Layer Security

PII – Personal Identifiable Information

TDE – Transparent Data Encryption

SSL – Secure Socket Layer

I/O – Input/Output

VNet – Virtual Network

CLI – Command Line Interface

AKS – Azure Kubernetes Cluster

Glossary

API – Set of functions with logic to be acceded by external parties without them knowing the internal workings of the software.

Cloud – Term used for content or services stored in physical servers that are accessible to users through the Internet.

API Validation Schema – Vocabulary that allows to annotate and validate JSON documents.

DevOps – Combination of philosophies, tools and practices that improves the ability of organizations to deliver applications at a higher velocity.

Clusters – A set of nodes that run containerized applications.

Containers – Software package containing everything needed to run an application.

DaemonSets – Kubernetes feature to ensure that some or all pods are scheduled and running on every single available node.

Cronjobs – Job scheduler for Unix-like operating systems. It´s used to schedule jobs (commands) to run periodically at fixed times.

Pod – A Pod is a single instance of a running process in a cluster. It could contain multiple containers running.

Malware – Term used to define any software intentionally designed to harm a computer, server, client, or computer network.

Workloads – Applications or services running.

Ingress – Term used to define an API object that manages external access to the services in a cluster, typically HTTP.

Egress – Term used to define communications being made from Pods to anything outside of the cluster.

Configuration Drift – Term used to describe an environment in which running clusters in an infrastructure become increasingly different over time, usually due to manual changes.

Container Registry – Term used to describe a repository or collection of repositories used to store and access container images.

YAML – File extension

Load Balancer – Term used to describe a service that distributes the network traffic across multiple servers, improving application availability, responsiveness and avoiding server overload.

Figures List

Figure 1 – Monolithic vs Microservices Architecture [2] 24
Figure 2 – Most popular cloud technologies [4] 25
Figure 3 – ANOVA's system architecture 33
Figure 4 – OPA workflow diagram [5] 38
Figure 5 – Azure Policy with Gatekeeper workflow [12] 41
Figure 6 – Defender for Cloud capabilities [13] 42
Figure 7 – Policies view in Azure Portal 57
Figure 8 – Content of *privileged_test.yaml* file 58
Figure 9 – Result of *kubectl apply* command with privileged property true 59
Figure 10 – Result of *kubectl apply* command with privileged property false 59
Figure 11 – Azure Policy reporting non-compliant policy 60
Figure 12 – Azure Policy reporting compliant policy 60
Figure 13 – Content of *test.yaml* file 61
Figure 14 – Result of *kubectl apply* command with readOnlyRootFilesystem property false 61
Figure 15 – Result of *kubectl apply* command with readOnlyRootFilesystem property false 62
Figure 16 – Content *test.yaml* file with allowPrivilegeEscalation property false 62
Figure 17 – Content *test.yaml* file with allowPrivilegeEscalation property false 63
Figure 18 – Content *test.yaml* file with allowPrivilegeEscalation property false 63
Figure 19 – Content of *test.yaml* file 64
Figure 20 – *Kubectl expose* command 64
Figure 21 – Result of the *kubectl expose* command with a forbidden external IP 65
Figure 22 – Result of the *kubectl expose* command with an allowed external IP 65
Figure 23 – First Semester Work Plan 67
Figure 24 – Second Semester Work Plan 68
Figure 25 – Risk Exposure Matrix 71
Figure 26 – Policy 1 definition (1st part) 80
Figure 27 – Policy 1 definition (2nd part) 81
Figure 28 – Policy 2 definition 82
Figure 29 – Policy 3 definition (1st part) 83
Figure 30 – Policy 3 definition (2nd part) 84
Figure 31 – Policy 4 definition (1st part) 85

Figure 32 – Policy 4 definition (2nd part) 86
Figure 33 – Policy 5 definition (1st part) 87
Figure 34 – Policy 5 definition (2nd part) 88

Tables List

Table I – App Definition and Development 34
Table II – Orchestration and Management 35
Table III – Runtime 35
Table IV – Provisioning 35
Table V – Observability and Analysis 36
Table VI – Features/Capabilities comparison [22] 45
Table VII – Community/Ecosystem comparison [12] 46
Table VIII – Meta/Misc comparison [12] 46
Table IX – Risk 1 – Working full-time 69
Table X – Risk 2 - Working in Frontend field 70
Table XI – Risk 3 – Some policies may not be applied within the time 70

Page intentionally left blank

Chapter 1
Introduction

ANOVA is a leading global company in the "Industrial Internet of Things (IIOT)" for remote monitoring and asset management. ANOVA is developing a new IIOT platform that intends to be the platform that unifies several existing platforms into a single one that is agnostic to the type of device. The goal is not only to build a new IIOT platform but also to use the most modern development methodologies such as "Continuous Delivery" and Cloud-Native architecture.

This document contains the description of the work done during this internship which aims to study and implement a tool to ensure compliance with security policies capable of automating and unifying the implementation of security policies across different teams and departments.

The adoption of Cloud-Native technologies has grown in recent years. This type of architecture brings several benefits over the traditional approach, such as faster time to market, greater abstraction in relation to the hardware used, better cost efficiency, among others. But this type of architecture also brings several challenges such as "*Cloud lock-in*", security, rapid technological evolution, among others. In a Cloud-Native architecture, security must be worked from the planning, development, test, and deploy phases, being a great challenge because it crosses several areas of specialization (cybersecurity, infrastructure, development).

Cloud security is getting more relevant day by day since Cloud-Native solutions constantly growing adoption. Using many solutions increases indirectly the attack surface since each one of those solutions can have certain vulnerabilities waiting for malicious intent to exploit them. To secure Cloud-Native technologies, organizations need to configure multiple tools, policy languages, and policy models to manage security in each one of them. This might not be the optimal solution, especially for organizations that are using many different technologies since it tends to be costly and more time-consuming.

To improve security across the whole Cloud-Native stack, the configuration of a unified platform that allows us to create, manage and audit security, would be the most desirable

solution. Using a unified platform, it would be possible to manage security policies across the entire stack using only one policy model and language without sacrificing availability or performance.

To achieve success in a Cloud-Native architecture, it is essential to understand this type of environment and have a collaborative modernization strategy between several teams, as well as specialized tools that allow you to automate and guarantee security at all stages of a Cloud-Native application.

One such tool used in Cloud-Native environments is the central management of security policies to ensure organizational standards and assess and ensure compliance in environments where multiple teams work together to bring a product to market quickly and safely. The main goal is to have a tool that allows you to ensure compliance and automate the application of security policies as well as monitor and remediate identified issues.

1.1 Goals

This has the main goal to research and implement a tool to ensure compliance with security policies capable of automating and unifying the implementation of security policies across different teams and departments in a Cloud-Native platform. With the goal of improving observability, the tool should also provide visual feedback on the current status of those policies, if they are compliant or not.

This project also has multiple sub-goals, such as:

- Understand which tools exist nowadays to handle security policies and which one is the best for the given purpose.
- Setup the whole that was chosen, in order to support the necessary Cloud-Native technologies.
- Research about the most suitable security policies to implement for each technology.
- Learn the policy language if needed and implement security policies.
- Validate the functionality of those policies.

The final obtained results will enhance the existing documentation regarding experimental work with the tool and the rest of the research topics addressed in this thesis.

1.2 Structure

This document is organized and divided into 8 main chapters. The **Introduction** (Chapter 1), the current chapter, the **Context** chapter (Chapter 2), the **Environment Architecture** (Chapter 3), the **State of Art** (Chapter 4), the **Policies** chapter (Chapter 5), the **Implementation and Validation** chapter (Chapter 6), the **Planning** chapter (Chapter 7), and finally, the **Conclusion** chapter (Chapter 8).

Chapter 1 – In this chapter is made an initial description of the theme being approached, the motivation of the research, the goals, the contributions of the author, and how the report is organized and structured.

Chapter 2 – This chapter provides a context about the subject approached throughout the report before diving into the research and implementation. It covers multiple topics, such as what is Cloud-Native, its architecture and technologies involved, what is Policy-as-Code, how it works, types of policies, its benefits, and use cases.

Chapter 3 – In this chapter is made a general description of ANOVA's environment architecture. What is its technology stack when it comes to App Definition and Development, Orchestration and Management, Runtime, Provisioning, and finally, Observability and Analysis.

Chapter 4 – State of Art chapter provides a background of what has been made so far in this field, containing a description, brief comparison, and analysis of the tools and techniques available nowadays to handle security in Cloud-Native environments and the most adequate ones to define security policies on these environments.

Chapter 5 – Policies chapter contains a research about the most important and suitable security policies to be implemented for each Cloud-Native technology. It will be an important starting point for the practical implementation since it will be gathering most of the security policies needed.

Chapter 6 – This chapter has documented the process of implementation/application of the set of policies chosen, as well as a proper validation of whether they are working as expected or not.

Chapter 7 – In the Planning chapter is where the planning of the whole project is approached. It describes the tasks and timelines for each semester, an analysis of the risks that may impact the project's success as well as the work methodology.

Chapter 8 – In the Conclusion chapter is provided a conclusion about the work performed throughout the project, as well as a summary of what still needs to be done regarding future work.

Chapter 2
Context

In order to be able to understand most of the topics covered throughout the report, some background is needed, mainly about Cloud-Native and Policy as Code. In this section will be explored these two concepts, what they are, how they work, their benefits, and so on.

2.1 Cloud-Native

Cloud-Native is a relatively new approach to managing data, building, and running software applications that exploit the flexibility, scalability, and resiliency of cloud computing. It's similar to traditional computing, except that instead of deploying applications on physical hardware locally, all software, servers, and networks are hosted in the cloud. It uses a computer-on-demand model, where IT resources are accessible through Internet connections as needed.

Cloud-Native solutions are mainly promoted by CNCF (Cloud Native Computing Foundation), a foundation that is part of the Linux Foundation and aims to promote the development of cloud solutions, whether public, private, or hybrid. In this way, CNCF supports and sustains an ecosystem of open-source projects without any linkage to any supplier, that is, solutions that can be implemented in any cloud service provider.

A typical example of cloud computing usage is email. Software as a Service (SaaS) providers such as Gmail or Microsoft Outlook allows users to store email data on external servers that can be accessed through any common browser from anywhere in the world.

Cloud-Native applications usually have some specific characteristics that distinguish them from others, such as [1]:

- **Automation**: This type of applications can be deployed and managed by machines and not by humans.
- **Flexibility**: These applications should be able to move from one container to another one without problems, it doesn't matter where it's running, it's decoupled from the infrastructure.

- **Resilience and Scalability**: They must withstand hardware failure, processing, or any other point that is not human responsibility.
- **Observability**: They should allow monitoring, logging, tracing, and metrics that help the SRE team find gaps or improvement points.
- **Distributed**: Cloud-Native is being able to benefit from the distributed and decentralized nature of the cloud. Cloud-Native applications tend to be distributed across multiple microservices that operate with each other.

2.1.1 Architecture

Microservices are the core of Cloud-Native application architecture. By using a Microservices technology that splits a big application into multiple independent smaller units, every unit will be processed as a separate service. Each one could have its own business logic, database and execute different functions.

Looking at a monolithic architecture, it takes a lot of effort to deploy new changes to the production environment when new features are implemented. Multiple teams need to coordinate their changes since any code change will affect the whole system, deploying multiple features all at once require a lot of integration and testing, applying a new technology will make the entire application be rewritten, and so on.

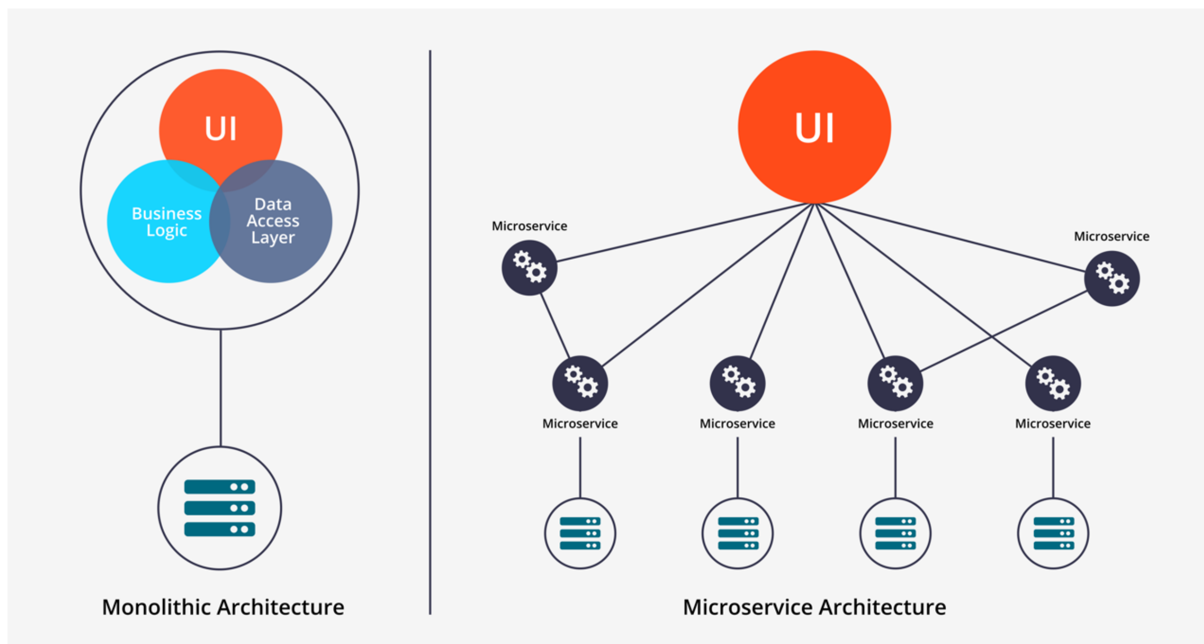

Figure 1 – Monolithic vs Microservices Architecture [2]

As we seen in Figure 1, a Microservice Architecture can split an application into several independent microservices. Each one can be deployed and updated independently providing more flexibility. Each unit can also be scaled independently. There is no need to deploy an entire application if there are only changes in one unit. Any fault appearing in the application will only affect a unit instead of affecting the entire application [3].

2.1.2 Technologies

Cloud-Native accouples several different tools and technologies, oriented to the different layers of the stack.

As presented next in the Figure 2, the Cloud-Native stack can be divided into many different layers, such as Applications & Database, CI/CD, Platforms, Container Orchestration, Container Engine, Operating System, and Virtual Infrastructure. It is up to each organization to set up its infrastructure, combining the most adequate technologies and tools for its goals.

Figure 2 – Most popular cloud technologies [4]

2.2 Policy as Code

Code used to be just something developers used to write applications. Nowadays, code has become something used for managing most aspects and stages of an application lifecycle, such as deployments, access control, security, and so on. That is all thanks to Policy as Code which brings multiple advantages to DevOps teams.

2.2.1 What is Policy as Code?

Policy as code is an extension of the infrastructure as code movement, which has been implemented in DevOps circles over the last ten years. Now is the time for policy as code to kick in, move out of its niche DevOps territory and into the core technology arena. A greater understanding of what it really is and the real challenges it solves will allow politics as code to be embraced to its full potential.

Policy as code originated from the principles of Test-Driven Development, where users first defined the business case, or 'desired state', in code. When applying these principles to the infrastructure, the desired state is known “as code” and is applied to test any changes to the infrastructure. With the rapid growth of application production, this type of pre-release testing is vital for organizations.

The software development lifecycle is under pressure to get products to market faster. This often means that compliance and security are left out and policies end up being manually enforced, which causes development delays. Embedding the policy as code in the early stages of development ensures that all changes from that point forward are validated. This means that risks that can arise later in production can be eliminated, minimizing interruptions, and giving the business greater confidence.

2.2.2 How it works?

Policy as code is a programmatic approach to applying and enforcing rules (policies) to an organization's cloud resources. It's an effective way to define, maintain, and implement policies uniformly across the software development lifecycle.

Business leaders can define these policies and teams can code these policies using some kind of programming language like YAML or Rego. These policies can be defined in a declarative format in the git repository, providing powerful features for change management - version control and fine-grained access control.

Automating security and policy compliance checks as code will allow teams to detect errors and violations early in the software lifecycle. This will create developer-centric experiences, security will be built into the system design, and organizations will have security-conscious teams.

According to *Weave*, a well-known company with the mission of empowering developers and DevOps teams to build better software faster, there are four important steps to have policies in action. Next, there will be presented each of these steps, [4]:

1. **Create Policy Playbooks -** The first step of implementing Policy-as-Code into DevOps pipelines is to create the policies. These policies can be based on organizational best practices, compliance standards, or security frameworks.

2. **Codify Policies -** The next step is to code the policies. Policies can be written in a high-level language such as Python, Yaml, or Rego. The language chosen depends on the policy enforcement engine or platform being used.

3. **Integrate Policies into CI/CD Pipelines -** By using a policy enforcement engine, it's possible to enforce the policies into CI/CD pipelines and prevent violating changes from being deployed. The engine will continuously monitor assets and configurations for any violations at every stage of the software lifecycle.

4. **Understand Cloud Security and Compliance Analytics -** In addition to preventing violating changes from being deployed, policy engines regularly scan Cloud-Native assets and generate compliance reports covering the security policies applied, standards, and best practices. Therefore, it is necessary to read and understand analytics in order to act when needed.

2.2.3 Types of Policies

Policies can be classified into three different types, such as Security and Compliance policies, Resilience policies, and Coding Standards. Next, there will be presented each one of these types, [4].

2.2.3.1 Security and Compliance Policies

These policies must ensure that security best practices are considered during the development lifecycle. Compliance policies ensure the adherence to industry standards and compliance rules.

2.2.3.2 Resilience Policies

These Policies include the best practices for deploying applications on Kubernetes, such as configuring health probes to ensure Kubernetes can do its automation correctly and many others. To ensure and improve the continuity of the business application, it is needed to make the system highly available and fault-tolerant by enforcing, for instance, specific policies in Kubernetes clusters.

2.2.3.2 Coding Standards

These policies can be company-mandated policies and checks that help organizations apply governance standards using a centralized playbook. For instance, a business has a rule that all Personally Identifiable Information (PII) must be encrypted when it's stored. A policy can be written into the system which is automatically triggered whenever a developer submits code. If the submitted code violates the policy, the code should be automatically rejected.

2.2.4 Benefits

Policy-as-Code can bring many benefits for teams, and organizations in general. Next, there will be presented some of the benefits teams and organizations can expect from a Policy-as-Code approach.

2.2.4.1 Controlling Costs

Monthly cloud bills can get sizeable due to unused resources left running or using over-sized instances for small tasks. The cost of a resource can be calculated ahead of time allowing the creation of a policy that limits the amount spent to deploy it. In addition, it is also possible to use the cloud provider's resources to implement a function with a policy that cleans up unused resources. Cloud provider native tools and practices can be used in combination with policy to control costs, [5].

2.2.4.2 Compliance

Policy-as-Code is a way to enforce infrastructure policies that prevent inadvertent access to resources or to enforce cost policies. A Policy-as-Code approach is not a one-time activity. Newly introduced code and every CI/CD pipeline iteration can trigger policy violations. With policies defined across the software pipeline, it's possible to prevent and detect any new violations from taking place.

2.2.4.3 Efficiency and Automation

Using code to define workflows and rules is the most suitable way to make sure DevOps teams are able to enforce the same policies across their environments using automated tools that translate code-based policies into actions. These policies can be reused as many times as needed, allowing scalability. Policies are only written once, whether there is one application to deploy or many applications, [6].

2.2.4.4 Easier Testing

Policies that are defined as code are easy to test in Sandbox environments before they are deployed into production. Since policies can remain consistent between dev/test and production, teams won't have to worry about configuration drift as code flows through CI/CD pipelines, [6].

2.2.4.5 Planning and Collaboration

Policy-as-Code avoids isolated workflows where developers would write code and hand it off to IT to deploy and manage, with no visibility into which goals the developers were targeting. With policies defined as code, everyone knows what the development and deployment rules are ahead of time, making it easier for developers and IT to collaborate, [6].

2.2.4.6 Simple and Efficient Auditing

With a Policy-as-Code approach it's faster and easier to perform audits to environments. Policies can be used to define how environments should be configured and then scan them to identify deviations from policies.

It also enables developers to continuously audit as part of the software delivery and deployment process, enabling collection of data from all systems, even those that don't persist, [6].

2.2.4.7 Centralized Policy Management

The possibility of using a centralized policy management platform allows teams to define, enforce, and manage a set of policies through a single interface. It is not needed to use different programming languages anymore, APIs, and multiple tools to ensure proper governance.

Business leaders and security and compliance teams can design a set of policies according to the necessary business outcomes. These policies are written once and enforced everywhere, everything is automated.

Developers will have the autonomy to move fast, and security will have time to scale their expertise and knowledge. Business leaders will rest assured that security best practices are applied into their CI/CD systems.

2.2.4.6 Conclusion

The benefits of a Policy-as-Code approach can extend beyond DevOps and into the success of organizations. From reducing costs and increasing efficiency and automation to preventing possible malicious attacks to infrastructures, there are some of many benefits that Policy-as-Code can bring.

2.2.5 Use Cases

Policy-as-Code can have multiple use cases. Next will be presented some common use cases of a Policy-as-Code approach.

2.2.5.1 Access Control

Implementing access control policies for any service is one of the most common use cases for Policy-as-Code. To check authorization, a service makes an API call to the policy engine which then answers whether the request is authorized or not. We can take the example of OPA (Open Policy Agent) which integrates policy enforcement and allows to specify Policy-as-Code and easy APIs to load policy decisions from software.

2.2.5.2 Kubernetes Control

Kubernetes clusters can be managed with policies. These policies contain rules for the different Kubernetes resources, like pods, namespace, deployment, nodes, and so on. These policies will assure compliance for Kubernetes.

2.2.5.3 Infrastructure Provisioning

Policy-as-Code can also be used to enforce specific requirements on Cloud resources, such as mandatory tags on instances, firewall, network settings, databases, and so on. Rules can be defined regarding the access that is needed for each service or storage.

2.2.6 Conclusion

Creating checks throughout an environment is a very important part of the software delivery process. The earlier and faster the errors or noncompliance can be caught, the better the software delivery process will be.

Policy-as-Code uses codified and automatic compliance policies. In terms of repeatability, versioning, and checking, Policy-as-Code can also support directly developers and operators. The ability to automate compliance by defining the rules to be applied for each context, regarding if it is a service or an infrastructure configuration, is one of the most attractive benefits that Policy-as-Code can provide.

Chapter 3
Environment Architecture

The architecture is strongly connected with the Azure Cloud Platform, and it takes advantage of all the relevant Azure capabilities. It connects devices, and other assets (i.e., "things"), captures the data that they generate, integrates, and orchestrates the flow of that data, then manages, analyzes the data, and presents the data as usable information. This usable information enables the people who need this data to make better decisions, as well as intelligently automate operations.

The Azure Cloud can provide speed of development (with all the finished applications they offer), speed of deployment, and the ability to grow and scale solutions to millions of "things".

The Platform has adopted a microservices architecture. Microservices are small pieces of functionality that contribute to the overall platform functionality. It is a collection of services doing their job, but the individual architecture of each microservice does not have much impact on the architecture of all platforms. Each microservice has its own *NoSQL* database. For databases, it is used *Azure Cosmos DB* with *MongoDB* interface. For larger files, it is commonly used *Azure Blob Storage*.

For inter-microservice communication, it is mostly used *gRPC*, but some microservices use other means of communication, which make more sense in their contexts, like *NATS* or *Kafka*. To encode and decode device messages it is used *JSON* or *Protobufs*.

Microservices are deployed using Docker images. This way microservice dependencies are contained in the Docker image and it allows to run them in different services such as Docker Swarm, Kubernetes, and other container-based solutions. These Docker images go into a Container Registry, so they can be used later to deploy easily to Kubernetes.

Kubernetes, the preferred container orchestration system, is the base of the Cloud Native Computing Foundation (CNCF), which is used worldwide in an infinite number of projects. The Kubernetes ecosystem is huge and has services like, *Helm*, *NATS*, *Prometheus,* and *Grafana*. Therefore, it is possible to take advantage of this ecosystem to build the Platform faster, with the help of all community projects.

ANOVA's solution is composed by a group of characteristics common to all IoT solutions that are addressed when designing the solution, these characteristics are:

- There is a set of devices that generate data

- It’s needed to interact with and manage these devices
- There is a cloud-hosted backend that ingests and processes data from the devices
- The volume of data is large and is generated at high velocity
- The system needs to detect business-relevant events and react in a timely manner
- The system is inherently distributed

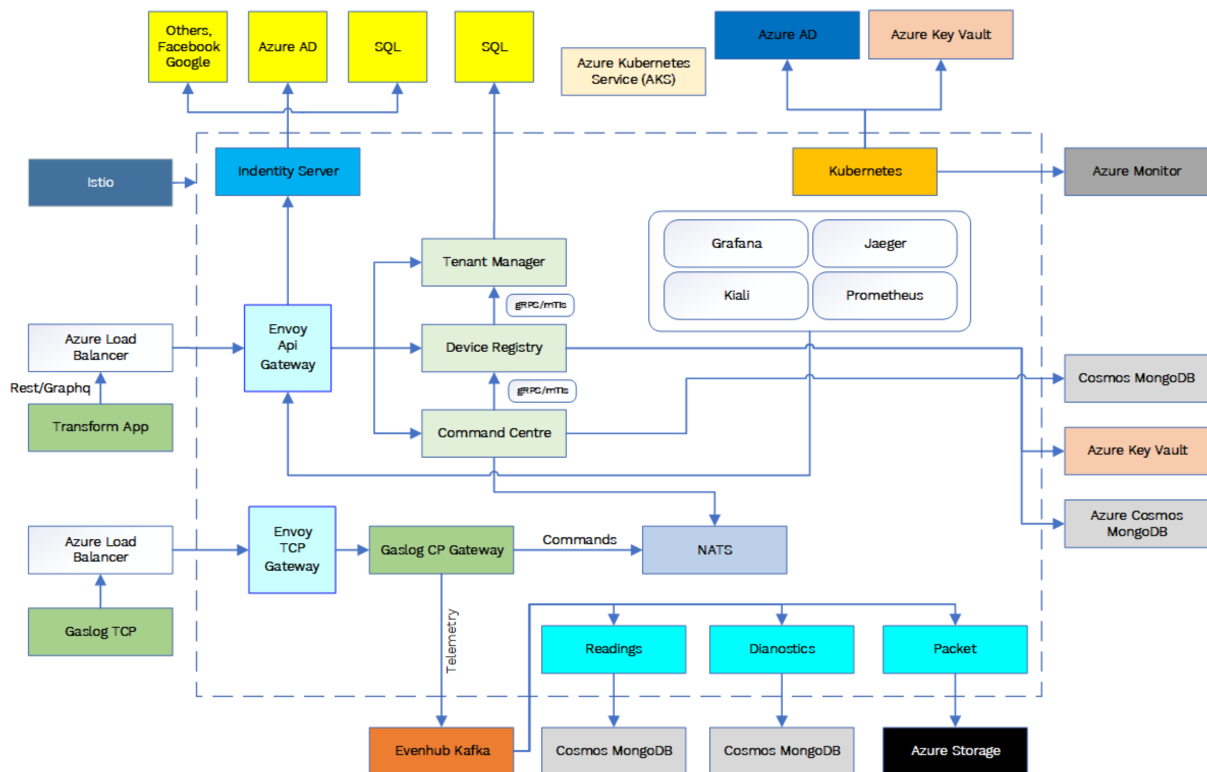

Figure 3 – ANOVA’s system architecture

As presented in Figure 3, ANOVA’s system architecture combines a wide range of technologies such as Kubernetes, Grafana, Kiali, Prometheus, Terraform, Kafka, different types of databases, multiple Azure services, and many others.

Next, there will be provided a more in-depth description of this technology stack and how it is organized.

3.1 Technology Stack

A Cloud-Native environment cannot rely on a single technology. Each environment will always be built with a combination of multiple different technologies for specific use-cases. The selection of these technologies will always depend on many factors, such as the requirements of the users, the organization's budget, how scalable the application should be in the future, and so on.

This environment's technology stack is divided up as defined at CNCF Cloud Native Interactive Landscape [7]. It has a combination of technologies from different layers such as App Definition and Development, Orchestration and Management, Runtime, Provisioning, and Observability and Analysis.

For each of these layers, there are many different tools being used. They will be presented in the following sections.

3.1.1 App Definition and Development

For App Definition and Development, as it's presented in Table IV, it is being used Azure Cosmos, MongoDB, SQL Server, Azure Redis Cache, Azure Blob Storage, and Azure Table Storage for **Databases**, NATS, Kafka, and Benthos for **Streaming and Messaging**, Helm, OpenAPI, and Docker for **App Definition and Image Build**, Azure Pipelines, Fastlane, Flagger, Flux, and Helm Operator for **CI/CD**.

Table I – App Definition and Development

Database	Streaming and Messaging	App Definition and Image Build	CI/CD
Azure Cosmos	NATS	Helm	Azure Pipelines
MongoDB	Kafka	OpenAPI	Fastlane
SQL Server	Benthos	Docker	Flagger
Azure Redis Cache			Flux
Azure Blob Storage			Helm Operator
Azure Table Storage			

3.1.2 Orchestration and Management

For Orchestration and Management, as it's presented in Table V, it is being used Kubernetes for **Scheduling and Orchestration**, CoreDNS for **Coordination and Service Delivery**, gRPC, and Initiative for **Remote Procedure Call**, Envoy for **Service Proxy**, Istio for **API Gateway** and **Service Mesh**.

Table II – Orchestration and Management

Scheduling and Orchestration	Coordination and Service Delivery	Remote Procedure Call	Service Proxy	API Gateway	Service Mesh
Kubernetes	CoreDNS	gRPC	Envoy	Istio	Istio
		Initiative			

3.1.3 Runtime

For Runtime, as it's presented in Table VI, it is being used Azure Storage for **Cloud Native Storage**, Cri-o for **Container Runtime**, and CNI for **Cloud Native Network**.

Table III – Runtime

Cloud Native Storage	Container Runtime	Cloud Native Network
Azure Storage	Cri-o	CNI

3.1.4 Provisioning

For Provisioning, as it's presented in Table VII, it is being used Terraform for **Automation and Configuration**, Azure Container Registry for **Container Registry**, Anchore and Azure Security Center for **Security and Compliance**, and Azure Key Vault for **Key Management**.

Table IV – Provisioning

Automation and Configuration	Container Registry	Security Compliance	Key Management

Terraform	Azure Container Registry	Anchore	Azure Key Vault
		Azure Security Center	

3.1.5 Observability and Analysis

For Observability and Analysis, as it's presented in Table VIII, it is being used Prometheus, Thanos, Grafana, Kiali, and Azure Monitoring for **Monitoring**, Grafana Loki, and Azure Log Analytics for **Logging**, Jaeger Tracing, and Open Tracing for **Tracing**, and Gremelin for **Chaos Engineering**.

Table V – Observability and Analysis

Monitoring	Logging	Tracing	Chaos Engineering
Prometheus	Grafana Loki	Jaeger Tracing	Gremlin
Thanos	Azure Log Analytics	Open Tracing	
Grafana			
Kiali			
Azure Monitoring			

Chapter 4
State of Art

Due to increasing software vulnerabilities, the possibility of developers committing errors that can compromise security, as well as the continuous emergence of new data privacy regulations, security is increasingly generating more concern over time, especially when we are talking about the cloud where data, resources, and private information could potentially be exposed on a massive scale. As opposed to older software architectures, modern Cloud-Native solutions come with nuances that require more and different types of security.

In the following sections, we will approach some Cloud-Native Security and Compliance solutions, such as Open Policy Agent, from Styra, Azure Policy and Microsoft Defender for Cloud, both from Microsoft, as well as some of the most popular security and compliance tools approved by Cloud-Native Computing Foundation (CNCF). These solutions can help apply and manage policies, ensure regulatory compliance, protect web traffic, assure frequent software patching, and many other vital actions. Most of the tools approached offer integrations with popular Cloud-Native infrastructures like Microsoft Azure, Amazon Web Services, Google Cloud Platform, IBM Cloud, etc. Since Kubernetes are usually one of the broadest technologies in a Cloud-Native environment, most of the tools and technologies are more focused on Kubernetes security and compliance.

4.1 Open Policy Agent

Open Policy Agent provides a universal policy engine across an entire Cloud-Native stack. This decoupled nature makes it easier to apply policy controls across containers, Kubernetes, APIs, service mesh, or at the application level. It relies on a unique high-level declarative language. Using this language, it is possible to specify policies across all create, update, and delete operations. This is helpful to ensure provenance is trusted and lock down access to only those with correct authentication credentials.

4.1.1 How does OPA works?

Open Policy Agent Workflow can be described the following way [5]:

1. API queries OPA for a decision. The query will contain some attributes like HTTP method used in the request, the path, the user, and so on.
2. OPA validates those attributes against data already provided to it.
3. OPA sends a decision to the requesting API with either allow or deny.

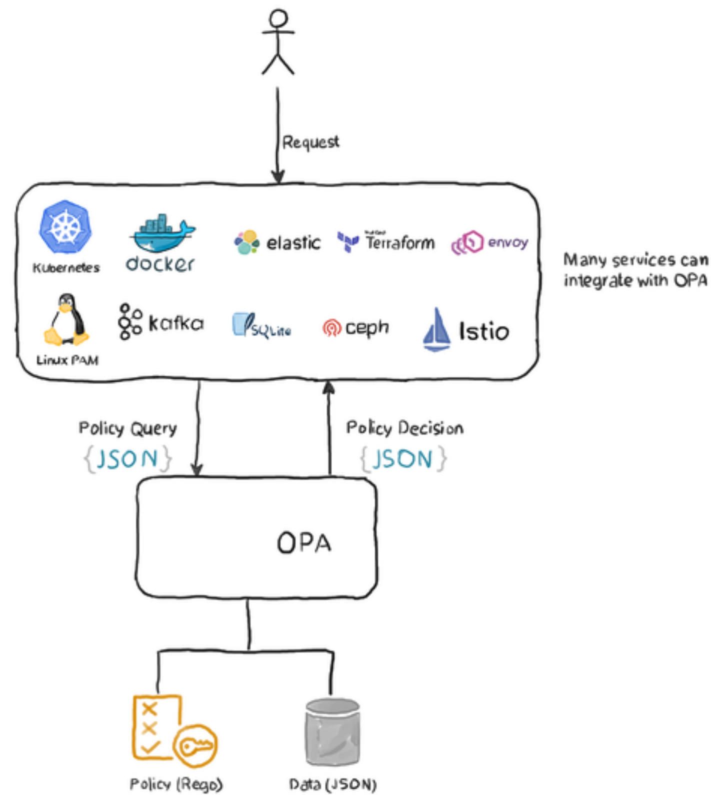

Figure 4 – OPA workflow diagram [5]

As shown in Figure 4, users make requests that will be handled by APIs being used. Then, the API will ask OPA for a decision about the request. Finally, OPA will validate the decision and send the answer back to the API so it can serve the user.

4.1.2 Rego

Regarding programming languages, there are a lot of general-purpose programming languages like Go, Rust, and Python designed for software developers. But general-purpose programming languages aren't well-suited to specific problems and so a new class of languages is required. Each language is designed to address requirements that arose from the

specifics of the domain. Similarly, Policy requirements are not well-served by general-purpose programming languages, so it needs its own language. Rego is a high-level policy language for defining rules that are evaluated by the Open Policy Agent engine. It allows specifying policy as code and simple APIs to offload policy decision-making from software. Rego focuses on providing powerful support for referencing nested documents and ensuring that queries are correct and ambiguous. It is declarative so policy authors can focus on what queries should return rather than how queries should be executed.

The design of Rego was mainly influenced by the following requirements [8]:

1. Humans need to be able to write the policies they want in a way that is also understandable to machines.

2. The policy language must be able to deal natively with the inherited hierarchical information that defines the cloud-native environment.

3. Policy is something that many stakeholders throughout the organization need to understand, e.g. developers, operations, security, and compliance.

4.2 Azure Policy

Azure Policy is a service, provided by Microsoft, that helps to enforce organizational standards and to assess compliance at a large scale. It has multiple use cases, such as implementing governance for resource consistency, regulatory compliance, security, cost, and management.

It uses JSON Format to define business rules that will be compared to Azure resources and evaluate whether it is compliant or not. Policies can be assigned to resources using .NET, JavaScript, Python, REST, Terraform, and a few other technologies.

4.2.1 Evaluation

Regarding evaluation, resources can be evaluated at different times, such as during the resource lifecycle, the policy assignment lifecycle, and for regular ongoing compliance evaluation. Next, will be presented some of the times or events that can make a resource to be evaluated [9]:

- A resource is created or updated in a scope with a policy assignment.
- A policy or initiative is newly assigned to a scope.
- A policy or initiative already assigned to a scope is updated.
- During the standard compliance evaluation cycle, which occurs.

4.2.2 Response

Business responses to non-compliant resources can vary widely between organizations. These responses are known as *effects,* in Azure, and are set in the policy rule definition. Organizations can define different *effects*, such as [9]:

- Deny the resource change.
- Log the change to the resource.
- Alter the resource before the change.
- Alter the resource after the change.
- Deploy related compliant resources.

4.2.3 Azure Policy for Kubernetes

In order to improve the security of Azure Kubernetes Service (AKS) clusters, it is recommended to apply and enforce security policies by using, for instance, Azure Policy. Azure Policy helps to enforce organizational standards and to assess compliance. Azure provides a specific Azure Policy add-on for AKS that can apply individual policy definitions or groups of policy definitions called initiatives to your clusters [10].

Azure provides multiple policies already set, but in addition, it is also possible to define custom policies to fit different needs. Once these policy definitions have been created, they need then to be assigned to Kubernetes clusters.

Azure Policy makes it possible to manage and report on the compliance state of Kubernetes clusters from one place, the Azure Portal. The Azure Policy add-on enacts the following functions:

- Checks with Azure Policy service for policy assignments to the cluster.
- Deploys policy definitions into the cluster as constraint templates and constraint custom resources.

- Reports auditing and compliance details back to Azure Policy service.

As mentioned before, Azure provides a built-in set of policies [11], covering multiple different application categories, that can be adjusted to suit different specifications. These policies can enforce as well, for example:

- Security Practices
- Cost Management
- Organization-specific rules (name, locations, and so on)

Azure Policy extends Gatekeeper, the admission controller webhook for Open Policy Agent (OPA), to apply enforcements and safeguards on Kubernetes clusters in a centralized and consistent manner. Policies are defined in Azure, in JSON format.

Next, in Figure 5, is presented the workflow between Azure Policy and Gatekeeper admission controller.

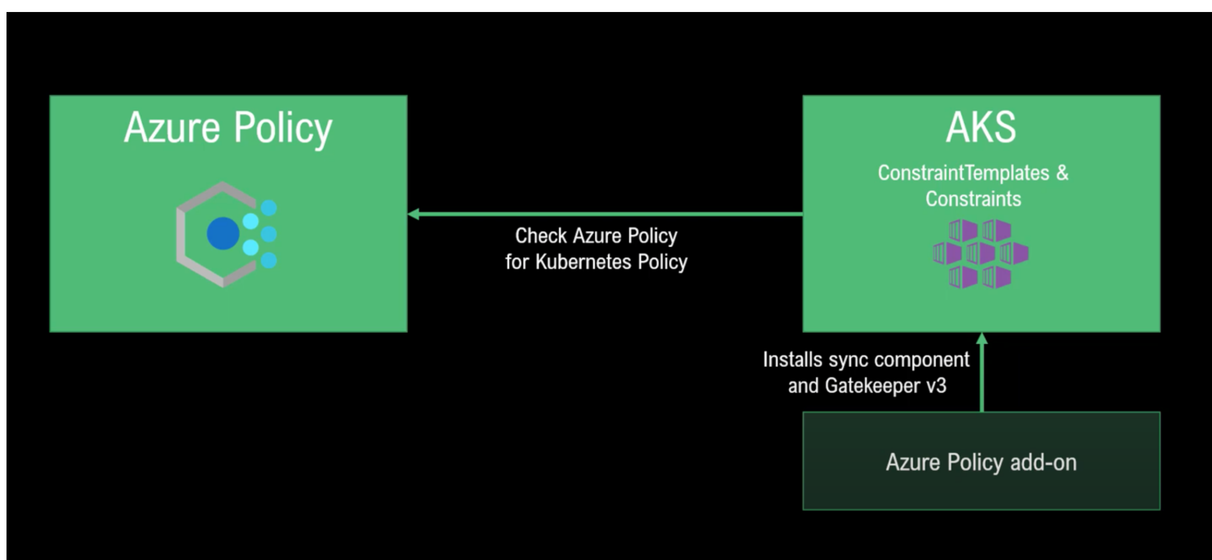

Figure 5 – Azure Policy with Gatekeeper workflow [12]

As seen in Figure 5, on one hand, there's Azure Policy, and on the other hand, there's an Azure Kubernetes Cluster (AKS). The Azure Policy add-on is added to the cluster which will allow the application of policies on the cluster. This add-on will install a sync component and Gatekeeper v3 automatically. Then, it will check Azure Policy for specific Kubernetes policies

and create *ContraintTemplates* and *Constraints* which are artifacts that Gatekeeper can work with. With those artifacts, Gatekeeper is able to decide whether a request should be allowed or denied and then, send the result of the validation back to Azure Policy.

4.3 Microsoft Defender for Cloud

While Cloud brings multiple benefits for organizations compared to the other solutions, there are also security challenges inherited that require a pragmatic approach to reduce vulnerabilities and the attack surface.

Cloud-Native environments usually relies on the integration of several internal and external services to host applications. Attackers can study the usage patterns of these services, identify security gaps, and then attempt to perform breaches.

Defender for Cloud can offer a solution for the mentioned challenges, in a single platform, to manage threats and the security posture of workloads in Azure. As presented in Figure 6, it fits three vital needs for managing the security of resources, such as, **Continuously Assess**, **Secure**, and **Defend**.

Figure 6 – Defender for Cloud capabilities [13]

For **Continuous Assessment**, it provides a solution named *Secure Score* which shows a score representing the current security situation. For **Secure**, it provides the possibility of implementing *Security Recommendations* (policies), to improve security posture. Finally, for **Defend**, it provides *Security Alerts* which are triggered when Defender for Cloud detects threats to resources and workloads [13].

Microsoft Defender for Cloud can work together with Azure Policy and provide an even stronger layer of security. Azure Policy can enforce - by defining a policy - for instance, a

vulnerability scan to Kubernetes images before they are deployed, and depending on the result, allow or deny the deployment.

4.4 Tools

Although Cloud-native environments are seen as secure environments, they are also prone to many types of threats. Therefore, developers are continuously trying to incorporate more automated threat detection and management tools to help get a better visibility into vulnerabilities. There are many tools designed for security and compliance in Cloud-Native environments under the CNCF umbrella. However, we will approach some of the well-known and most used tools nowadays.

4.4.1 Falco

Falco is a threat detection package that can be used to specify rules for containers. It can scan for known common vulnerabilities and exposures and trigger alerts to help respond to threats quickly. Falco ships with default rules to check for unusual behaviors such as privilege escalation, namespace changes, risky read/write abilities, unexpected network connections and other potential exploits. Falco also provides integration with tools such as OPA, Prometheus, Helm, Kubernetes, Elasticsearch, and others. Falco was the first runtime security project to join CNCF as an incubating project and since has seen adoption by many companies including GitLab, Shopify, Skyscanner and many others [5].

4.4.2 K-Rail

K-rail is a tool built mainly to help manage security in Kubernetes while maintaining high developer productivity. As a workload policy enforcement tool designed for Kubernetes, K-rail allows us to [15]:

- Measure violations before and after enforcing them.
- Use flexible and easy-to-use policy exemptions.
- Use many impactful policies out of the box.
- Get real-time, interactive feedback to users when they apply resources, even high-level resources such as Deployments, DaemonSets, and CronJobs.

4.4.3 Harbor

Harbor is an open-source tool that aims to secure artifacts with policies and role-based access control, ensures images are scanned and free from vulnerabilities, and signs images as trusted. Harbor, also a CNCF Graduated project, delivers compliance, performance, and interoperability to consistently and securely manage artifacts across Cloud-Native platforms like Kubernetes, Docker, and many others [16].

4.4.4 Conftest

Conftest is a tool created to help write tests against structured configuration files. It was built purely focusing on building the best developer experience for testing configuration files.

Conftest relies on the Rego language created by Open Policy Agent. It gives the possibility to write tests for Kubernetes configurations, Terraform code, Tekton pipeline definitions, and many other configurations and structured data.

With Conftest it's possible to test three types of rules: *deny, violation,* and *warn.* These rules can be applied to JSON files, YAML files, or Helm charts, with the help of a external plugin [17].

4.4.5 Kyverno

Kyverno the Kubernetes Native Policy Management tool. With Kyverno, policies are managed as Kubernetes resources, and no new language is required to write policies. This allows using familiar tools such as *kubectl*, *git*, and *kustomize* to manage policies. Kyverno policies can perform the following operations on generated Kubernetes resources:

- **Validation** - ability to verify resource configurations for policy compliance [18].
- **Mutation** - ability to modify a resource during admission control [19].
- **Generation** - ability to create additional resources based on resource creation or source updates [20].

The Kyverno Command Line Interface can also be used to test policies and validate resources as part of a CI/CD pipeline.

4.4.6 Gatekeeper

Gatekeeper was created to enable users to customize admission control via configuration, not code and to bring awareness of the cluster's state, not just the single object

under evaluation at admission time. Gatekeeper is a customizable admission webhook for Kubernetes that enforces policies executed by the Open Policy Agent (OPA), a policy engine for Cloud Native environments hosted by CNCF.

It allows us to enforce policies like [21]:

- All images must be from approved repositories.
- All ingress hostnames must be globally unique.
- All pods must have resource limits.
- All namespaces must have a label that lists a point-of-contact.

Gatekeeper acts as a bridge between the API server and OPA. The API server will enforce all policies executed by OPA, during the validation process. It can also be integrated with Azure Policy, through an add-on, where it will work as a validator of the policies defined and sent by Azure.

4.5 Gatekeeper vs Kyverno for Kubernetes

Pod Security Policy in Kubernetes is a set of mechanisms for ensuring validating controls over Pods and their attributes. Although it can add a layer of security, it only operates on Pods and can only block their creation without being able to perform any remediation.

In contrast, with policy engines such as OPA Gatekeeper and Kyverno, the capabilities are far broader (i.e., applicable to more than just Pods) and deep (i.e., more than just simple validation) [14].

In Table I, we can see a comparison between Gatekeeper and Kyverno in terms of features and capabilities which represents technical attributes.

In Table II, it is shown a comparison between both projects in terms of their community and ecosystem which represents the adoption and organizational attributes of each one.

Lastly, in Table III, we can see a comparison in terms of meta and miscellaneous which represents cognitive and miscellaneous attributes.

Table VI – Features/Capabilities comparison [22]

Features/Capabilities	**Gatekeeper**	**Kyverno**
Validation	✓	✓

Mutation	✓	✓
Generation	✗	✓
Policy as native resources	✓	✓
Open API validation schema	✗	✓
High availability	✓	✓
API object lookup	✓	✓
CLI with test ability	✓	✓
Policy audit ability	✓	✓
Self-service reports	✗	✓

In terms of **Features and Capabilities**, it's possible to see that Gatekeeper has some downsides prior to Kyverno since it doesn't allow to do Generation, doesn't provide an API validation schema and self-service reports as well.

Table VII – Community/Ecosystem comparison [12]

Community/Ecosystem	**Gatekeeper**	**Kyverno**
CNCF status	Graduated (OPA)	Sandbox
Partner ecosystem adoption	5/10	3/10
GitHub status (starts, forks, releases, commits)	2405, 493, 51, 875	2015, 272, 116, 4034
Community traction	5/10	3/10

In terms of **Community and Ecosystem**, we can see that in general, Gatekeeper has more adoption than Kyverno so far since it is a more mature project, and it is already graduated by CNCF (numbers registered in March 2022).

Table VIII – Meta/Misc comparison [12]

Meta/Misc	**Gatekeeper**	**Kyverno**
Programming required	✓	✗
Use outsider Kubernetes	✓	✗

Birth	July 2017	May 2019
Origin Company	Styra (OPA)	Nirmata
Documentation maturity	5/10	7/10

Looking at a **Meta and Miscellaneous** comparison, it's possible to highlight that is no need to learn a new programing language to use Kyverno, even though Gatekeeper is an older project, Kyverno has more mature documentation and it doesn't need to learn a specific programming language to write policies.

Based upon the information presented, both OPA Gatekeeper and Kyverno have their strengths and weaknesses. Although they are both very capable tools, the chosen between both will depend on several aspects and goals of the organization/users. OPA Gatekeeper could be the most suitable solution for an organization/user but can be a worse solution for an organization/user that has another plan or a different work model.

Users should evaluate which tool suits best their plan and work model but assure that they will use a policy engine to secure their clusters.

4.6 Conclusion

After researching and evaluating the most well-known projects and tools available to help ensure security and compliance for Cloud-Native services and technologies, it's clear that there are multiple different and powerful tools that can be used for this scope. Each project and tool have its own specifications that make them more suitable or less suitable for an organization depending on its goals, budget, IT capacity, and so on.

Although there are many good tools that could be used for this scope, since ANOVA's infrastructure is all built around Azure services, the wiser choice to implement policies across the whole Cloud-Native stack would be Azure Policy.

Azure Policy is obviously a service that is integrated with Azure and can be integrated with Kubernetes as well. It is a general policy engine that is integrated with Gatekeeper specifically for Kubernetes.

Therefore, it would be possible to define policies to control costs, resource creation (types, tiers, and so on), and security policies for multiple different technologies being used.

Chapter 5
Policies

Cloud-Native environments have the potential to be more secure than other types of environments. However, protecting systems, applications, and data in the cloud brings a completely new set of challenges to overcome. Security teams need to adapt, plan, and learn how to utilize the tools, controls, and design models needed to properly secure the cloud environments.

In this new territory, the process of setting policies is not only applicable, but it's essential to provide another layer of security and control. The more security layers an environment has, the more difficult would be for a malicious party to get in or simply for developers to commit errors that might compromise the system.

To well define policies, it's important to differentiate different types of policies. Policies can be categorized as company-wise policies and project-wise policies, policies that are defined according to organizational requirements and policies that are defined for each project specifications, respectively.

In this chapter, will be approached some of the most important topics about policies, as well as focus on gathering some of the prime security and compliance policies and best practices for securing a Cloud-Native environment and provide a brief description of how it works and the importance of each one.

The majority of policies are mainly focused on Kubernetes since Kubernetes are one of the biggest and most significant parts of the environment. Some of the policies are recommended by *Styra*, the creators of Open Policy Agent, *Nirmata*, the Kyverno creators [23], and by *Microsoft,* which provides multiple built-in policies as well. These policies should be implemented to ensure that the apps being built has an increased security layer that could prevent bad consequences like customer data being exposed to the entire internet, infecting clusters with malware, allowing privilege escalation and getting full access to a server, and so on.

5.1 What is?

The National Institute of Science and Technology (NIST) defines an information security policy as an “aggregate of directives, regulations, rules, and practices that prescribes how an organization manages, protects, and distributes information”, [24].

Since organizations have different business requirements and compliance obligations, it is not possible to define a single policy that works for everyone. Instead, each security team should evaluate and define the policy choices that fit better their needs.

5.2 Benefits

Policies are a must-have for organizations in many aspects. They can provide controls and procedures that help ensure security and compliance for different resources. More specifically, they are essential for the following reasons [25]:

- **Ensure confidentiality, integrity, and availability of data:** provides a standard approach for identifying and mitigating risk, as well as an appropriate response.
- **Minimize risk:** policies can help to evaluate and mitigate vulnerabilities to block security threats.
- **Communicate security measures:** enables an organization to easily communicate its security measures to employees, internal stakeholders, external auditors, contractors and other third parties.
- **Regulatory compliance:** it’s important for an organization to pass compliance audits for security and standards and regulations.
- **Controlling costs:** it´s important to keep costs controlled so the organization’s software bills don’t get sizeable.

By having a well-defined policy strategy, organizations can benefit from many different aspects, from security to costs.

5.3 Kubernetes Oriented Policies

In this section will be gathered the most important security and compliance policies, focused on Kubernetes. These policies aim to improve security and reduce the attack surface of a Kubernetes system.

5.3.1 Trusted Repo

Pulling images from unknown providers from the internet can bring risks, such as malware. By ensuring that images can only be pulled from trusted repos, it's easier to closely control the image inventory, mitigate the risks, and increase the overall security of your cluster.

This policy aims to only allow container images that are pulled from trusted repositories and, optionally, pull only those that match a list of approved repo image paths.

5.3.2 Block Pod Exec

The 'exec' command could be used to gain shell access or run several commands in a Pod's container.

This policy aims to block Pod 'exec' commands to Pods that are within a specific namespace, that have their name starting by a specific pattern, or if they contain a specific label.

5.3.3 Disallow Add Capabilities

Pod capabilities allow privileged actions without having full root access. To prevent this, the possibility to add capabilities beyond the default set, must not be allowed.

This policy ensures that users cannot add any additional capabilities to a Pod.

5.3.4 Limit Pod Containers

Pods can have many different containers which are usually coupled. It may be desirable to limit the number of containers that can be in a single pod to control best practice application so policy can be applied consistently.

This policy aims to check all Pods to ensure they have no more than four containers.

5.3.5 Label Safety

The downside of manual label entry is that it increases the probability of committing errors, especially because labels are both extremely flexible and extremely powerful in Kubernetes. They identify the groupings of Kubernetes objects and policies, including where workloads can run, either in Frontend, Backend, or Data-tier, and which resources can send

traffic. Getting labeling wrong can lead to untold deployment and supportability issues in production.

This policy requires all Kubernetes resources to include a label that follows a specific pattern and format. Through its application, it's easier to ensure that the labels are configured correctly and consistently which will reduce the probability of committing errors.

5.3.6 Handle Privileged Mode

One of the key measures to avoid access to the host's recourses and kernel capabilities, which includes the ability to disable host-level protections, is obviously not running containers in privileged mode. If a privileged container gets compromised, a whole system could get compromised next.

This policy ensures that containers cannot run in privileged mode, by default. If there are any specific circumstances where containers need to run in privileged mode, they can be specified as exceptions.

5.3.7 Define and Control Ingress

In Kubernetes, it's easy to activate a service that talks to the public internet which can cause to activate unnecessary services and quickly become very expensive and exceed the budget. Moreover, it's easy to break an application when two services try to share the same Ingress.

This policy aims to prevent Ingress objects in different namespaces from sharing the same hostname by allowing to expose specific services (allow Ingress) or doesn't expose any to the public. Therefore, new workloads won't steal internet traffic from existing workloads, preventing service outage, data exposure, etc.

5.3.8 Define and Control Egress

Like what happens with Ingress, it's easy to allow Egress to every IP in the world by default. Furthermore, it's also possible, at an intra-cluster level, to unintentionally send data to services that shouldn't have it. These situations carry risks of data exfiltration or theft if services get compromised.

This policy allows controlling how Egress traffic can flow. It lets specifying when both intra and extra cluster communications can occur and to which services.

5.3.9 Allowed Pod Priorities

This policy is used to provide a guarantee on the scheduling of a Pod relative to others. If not all users are trusted in a cluster, a malicious user could create Pods with the highest priority, causing other Pods to be overridden or not get scheduled.

This policy will check the defined priority of a Pod and block it if needed.

5.3.10 Require Limits and Requests

As application workloads usually share cluster resources, it is important to limit resources requested and consumed by each Pod. By requiring resource requests limits per Pod, it will reduce the change of memory and CPU reach their limit.

This policy validates that all containers have something specified for memory and CPU requests and memory limits.

5.3.11 Require Run as Non-Root

Running containers as root is not safe for security reasons. By running them as root, anyone with access to the containers will be able to read all kinds of information, including for instance database connection credentials and, especially in cloud environments, multiple cloud technologies credentials. Perhaps, it might also be possible to escape the container and start, for example, new services that could ramp up huge costs. This policy will ensure that containers must not run as non-root and only specific users will be able to access them as root.

5.3.12 Spread Pods Across Nodes

Deployments to a Kubernetes cluster with many different availability zones sometimes need to distribute those replicas to align with those zones to ensure site-level failures don’t impact availability.

This policy will check Deployments distributed configuration is set and mutates them to spread Pods across zones.

5.3.13 Kubernetes cluster containers should run with a read only root file system

Running containers with a read-only root file system is a great measure to protect from bad intentioned changes at run-time. Using an immutable root filesystem and a verified boot mechanism prevents against malicious parties from “owning” the machine through permanent local changes. An immutable root filesystem can also prevent malicious binaries from writing to the host system.

5.4 General Policies

In this section will be approached some general security policies and best practices that should be followed to increase security strength. These policies can fit a wide array of usage on ecosystem resources and subjects such as API management, App configurations, Servers and Virtual Machines configurations, and specific technologies and services policies, such as Kubernetes, Key Vault, Cosmos DB, Event Hub, and many other.

5.4.1 Restrict External IPs

External IPs can be used to perform, for instance, Man in the Middle Attacks. To prevent this, blocking external IPs or defining a set of allowed IPs is a good practice to follow so it is possible to restrict access.

5.4.2 Enforce SSL connections for databases

Azure Database for PostgreSQL and MySQL supports connecting Azure Database for PostgreSQL and MySQL servers to client applications using Secure Sockets Layer (SSL). Enforcing SSL connections between database servers and client applications helps improving security against "man in the middle" attacks by encrypting the data stream between the server and applications. This configuration enforces that SSL is always enabled for accessing database servers.

5.4.3 Services should use virtual network service endpoints

SQL Server, Key Vault, Cosmos SD, Event Hub, and other services should be enforced to use virtual network service endpoints.

Virtual Network (Vnet) service endpoint provides secure and direct connectivity to Azure services over an optimized route over the Azure backbone network. Endpoints allow to secure critical Azure service resources to only virtual networks. Service Endpoints enable private IP addresses in the Vnet to reach the endpoint of an Azure service without needing a public IP address on the Vnet [26].

This policy audits whether those services are configured to use a virtual network service endpoint or not.

5.4.4 Databases should use customer-managed keys to encrypt data at rest

Databases should be enforced to use customer-managed keys to encrypt data at rest. A malicious party who steals physical media like drives or backup tapes can restore or attach the database and browse its data.

One solution is to encrypt sensitive data in a database and use a certificate to protect the keys that encrypt the data. This solution prevents anyone without the keys from using the data. TDE (Transparent Data Encryption) does real-time I/O encryption and decryption of data and log files [27].

This policy audits whether a database is configured or not to use customer-managed keys to encrypt data at rest.

5.4.5 Container registry images should have vulnerability findings resolved

Container image vulnerability assessment from Azure Defender for Cloud, scans registries for security vulnerabilities and exposes detailed findings for each image. Resolving the vulnerabilities can greatly improve containers security posture and protect them from attacks.

This policy audits whether container registries have their vulnerability findings resolved or not.

5.5 Conclusion

As seen throughout this report, there is a vast amount of already built-in policies that were developed to improve security along with many already identified issues. The ones mentioned so far are the ones found most important and more quickly needed for ANOVA's context. There are many more policies that can be later set in order to increase security.

By implementing a suitable set of policies for a Cloud-Native environment it would be possible to prevent developers from accidentally bringing services down, exposing data to a non-authorized environment, and avoiding the manual remediation needed from teams, as well as cover multiple possible entry points for malicious parties to exploit and possibly harming systems.

Chapter 6
Implementation and Validation

Before starting the implementation, it was necessary to define the set of policies that were really important to have as soon as possible and suitable for the context. Thus, the ANOVA's Infrastructure team elements were questioned about what problems did they notice so far that would need to be covered with policies. In addition, it was a presentation about the existing policies that could be applied in ANOVA's environment to the team. Finally, the Infrastructure team's needs, combined with the policies gathered before that should be enforced, resulted in the set of policies approached in the next topics.

In that sense, this chapter will cover two very significant parts of the whole project, the implementation process, and the validation process. In order to be able to implement, apply, and test the functionality of the policies, it was necessary to think about the best approach to do the job, without harming ANOVA's systems and services availability initially.

Therefore, implementation and testing policies in production were obviously put apart. There were two main possibilities left, the first one was creating a specific cluster in Azure just to work and test policies but this one, other than the possibility of increasing costs, would not be the ideal solution since many policies would need to have different services running inside the cluster. The second possibility was working in a staging cluster, which contains many services running inside. Since there was different region staging clusters, the best way would be to work in a staging cluster in a specific region and, if something happened that could affect services availability, it would only affect temporarily a single region in staging which would not be a serious problem.

It was defined that the work would then be done in a specific region staging cluster context initially and if everything worked as expected with the desired output, the policies would be applied to the other staging and production clusters. Furthermore, since Azure Polices can be defined with the "Audit" *effect*, all policies were firstly applied with that *effect* so it would be possible to have feedback if services were compliant or not with policies, without blocking any services and harming availability as well.

As for the policies implemented, the names could be slightly different from the ones mentioned before, mainly in Chapter 4, but they are quite similar and easy to notice.

Additionally, every time that is possible to use Azure built-in policies, they should be used other than writing custom ones since Microsoft maintains them, and it's easier to keep them updated over time.

6.1 Implementation

After collecting the most important and needed policies to be implemented for the context, the following step is to put them to work. For this process, Azure Portal gives the possibility of assigning policies to scopes or even specific resources inside those scopes. In addition, it also allows to specify exclusions for some resources, for instance, as well as define specific parameters for each policy, depending on its goal, and define a remediation task. Remediation tasks will assure that those policies will take effect on existent resources, instead of only in newly created ones.

With this being said, next will be enumerated the policies defined during this process in Azure Portal:

- SQL servers should use customer-managed keys to encrypt data at rest
- Azure Cosmos DB accounts should use customer-managed keys to encrypt data at rest
- Kubernetes cluster pod hostPath volumes should only use allowed host paths
- Kubernetes cluster pods should only use allowed volume types
- Kubernetes clusters should not allow container privilege escalation
- Kubernetes cluster containers should run with a read only root file system
- Kubernetes cluster containers should only use allowed pull policy
- Kubernetes cluster should not allow privileged containers
- Kubernetes cluster containers should only use allowed images
- Kubernetes cluster containers should only use allowed capabilities
- Azure Policy Add-on for Kubernetes service (AKS) should be installed and enabled on your clusters
- Kubernetes cluster services should only use allowed external IPs
- SQL Server should use a virtual network service endpoint
- Enforce SSL connection should be enabled for PostgreSQL database servers

Although this set of policies was defined in Azure Portal, if there is a need for that, it is also possible to define more policies in the future. In this case, a policy is considered **Compliant** when all the resources on its scope are compliant with it, and **Non-compliant** when there is at least one resource non-compliant with it on its scope.

At the time, most of the policies are non-compliant, since there are resources that need to face some configuration changes before in order to be compliant without compromising their availability. Those policies are defined with the *effect* property set as "Audit" so the policy doesn't block anything but still audits the resources.

In the following figure, Figure 7, it is presented the Azure Portal view for Policies, where it's possible to view, assign, and modify policies, as well as see relevant information about them.

Name	Scope	Compliance state
GM: SQL servers should use customer-managed keys to encrypt data at rest	- staging	Non-compliant
GM: Azure Cosmos DB accounts should use customer-managed keys to encrypt data at rest	- staging	Non-compliant
GM: Kubernetes clusters should not allow container privilege escalation	- staging	Non-compliant
GM: Kubernetes cluster containers should only use allowed pull policy	- staging	Non-compliant
GM: Kubernetes cluster pods should only use allowed volume types	- staging	Non-compliant
GM: Kubernetes cluster should not allow privileged containers	- staging	Non-compliant
GM: Kubernetes cluster containers should only use allowed capabilities	- staging	Non-compliant
GM: Kubernetes cluster pod hostPath volumes should only use allowed host paths	- staging	Non-compliant
GM: Kubernetes cluster containers should only use allowed images	- staging	Non-compliant
GM: Kubernetes cluster containers should run with a read only root file system	- staging	Non-compliant
GM: Azure Policy Add-on for Kubernetes service (AKS) should be installed and enabled on your clusters	- staging	Non-compliant
GM: Kubernetes cluster services should only use allowed external IPs	- staging	Compliant
GM: SQL Server should use a virtual network service endpoint	- staging	Compliant
GM: Enforce SSL connection should be enabled for PostgreSQL database servers	- staging	Compliant

Figure 7 – Policies view in Azure Portal

As for the policy "Azure Policy Add-on for Kubernetes service (AKS) should be installed and enabled on your clusters", it is basically a policy to allow keeping track of clusters that are configured to use Azure Policy. Since in this context policies are only applied in staging scope, the policy audit is considered non-compliant.

Some of the policies are not yet assigned with the final parameters defined, such as the *hostPaths* needed, the customer-managed keys configured to encrypt data at rest, and some others, which will be addressed with the team in a near future. Thus, for the Validation section (6.2), there will be considered only the policies that are finalized and ready to be applied in production as well.

Since the JSON files of the built-in policies provided by Microsoft are quite long, those files will be attached in the Appendix section.

6.2 Validation

This section covers the validation of some policies. In order to validate if a policy is working as expected or not, it is usually performed a test in favor and against that policy.

To help perform these tests, it is used a tool called *kubectl*, which is a Kubernetes command-line tool. It basically allows running commands against Kubernetes clusters. It's possible to use *kubectl* to deploy applications, inspect and manage cluster resources, and view logs.

Let's say there's a policy that should deny the creation of pods with the tag "TestTag". In this case, a good validation test would be trying to create a pod with the tag "TestTag", and one without the tag "TestTag". After each attempt of creation, *kubectl* should return feedback on whether the creation of the pod was successful or not. Therefore, it's possible to validate if a policy is compliant or not.

6.2.1 Kubernetes clusters should not allow privileged containers

To validate this policy, it was created a *privileged_test.yaml* file with the security context of `privileged: true`, as presented in Figure 8. This security context escalates the pod's privileges. The policy disallows the creation of privileged pods, so the request should be denied resulting in the deployment being rejected.

Figure 8 – Content of *privileged_test.yaml* file

After the creation of *priviledged_test.yaml* file with the property privileged: true, it's used the *kubectl apply* command followed by the file name to create the pod. As expected, the scheduling of the pod failed with the output presented in Figure 9.

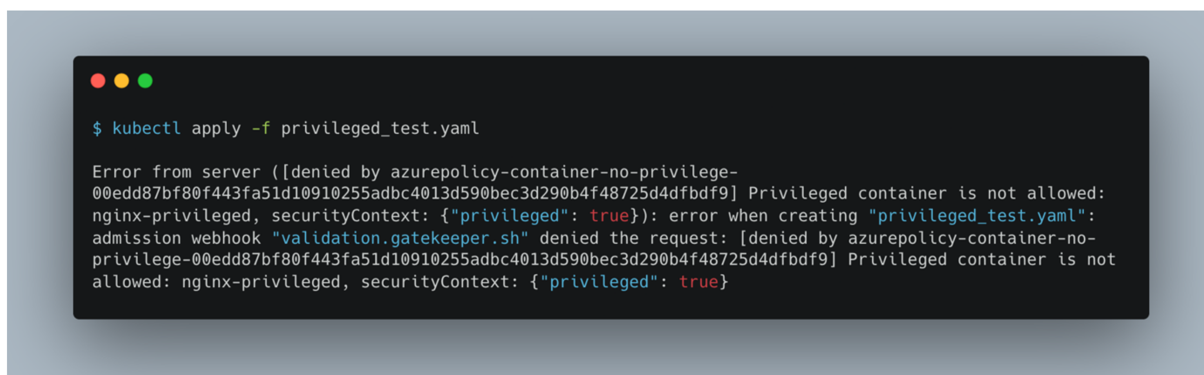

Figure 9 – Result of *kubectl apply* command with privileged property true

As for the YAML file created before, but with the property privileged: false, the pod is scheduled correctly, as presented in Figure 10.

Figure 10 – Result of *kubectl apply* command with privileged property false

6.2.2 SSL connection should be enabled for PostgreSQL database servers

For this validation, Azure Policy was essential. Azure Policy identified that one of the PostgreSQL database servers was not compliant with the policy, as presented in Figure 11. For

this specific situation, Azure Portal allows enforcing SSL connection in server settings without further configurations.

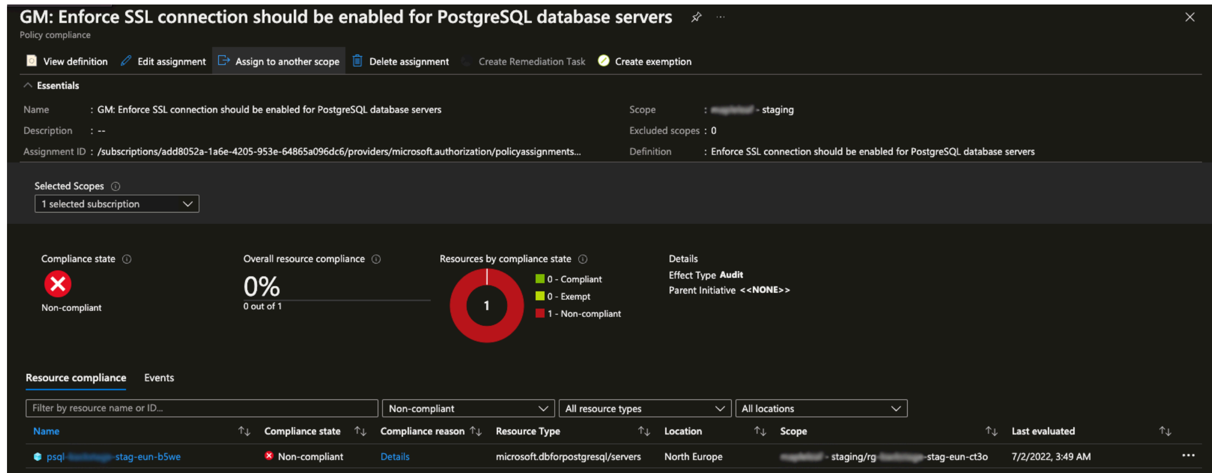

Figure 11 – Azure Policy reporting non-compliant policy

For this validation, Azure Policy was essential. Azure Policy identified that one of the PostgreSQL database servers was not compliant with the policy, as presented before in Figure 11. For this specific situation, Azure Portal allows enforcing SSL connection in server settings without further configurations.

After enforcing the SSL connection for the database server, Azure Policy rechecked the policies and detected it as compliant, as presented next, in Figure 12, meaning that the SSL connection for that database was enabled.

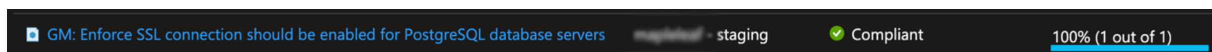

Figure 12 – Azure Policy reporting compliant policy

6.2.3 Kubernetes cluster containers should run with a read only root file system

For this validation, it was again created a *test.yaml* file, with the security context property readOnlyRootFilesystem: false, which is the default value for the property, as presented next in Figure 13.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: test-pod
spec:
  containers:
    - name: test-pod
      image: mcr.microsoft.com/oss/nginx/nginx:1.15.5-alpine
      securityContext:
        readOnlyRootFilesystem: false
```

Figure 13 – Content of *test.yaml* file

As expected, the pod scheduling was denied, because the policy was enforcing pods to run with a read-only root filesystem. Next, in Figure 14, it's possible to see the error returned by Azure Policy after trying to create the pod.

```
$ kubectl apply -f test.yaml

Error from server ([azurepolicy-psp-readonly-root-filesystem-98afa644ab42977b6ef3] Readonly root
filesystem is required for container. pod:'test-pod', container:'test-pod'): error when creating
"test.yaml": admission webhook "validation.gatekeeper.sh" denied the request: [azurepolicy-psp-
readonly-root-filesystem-98afa644ab42977b6ef3] Readonly root filesystem is required for container.
pod:'test-pod', container:'test-pod'
```

Figure 14 – Result of *kubectl apply* command with readOnlyRootFilesystem property false

On the other hand, when setting the readOnlyRootFilesystem property to true, the pod is scheduled without any problem, as shown in Figure 15.

Figure 15 – Result of *kubectl apply* command with readOnlyRootFilesystem property false

6.2.4 Kubernetes clusters should not allow container privilege escalation

To validate this policy, it is used the same approach as before. It is again created a *test.yaml* file, in this case, with the security context property allowPrivilegeEscalation: false as presented next in Figure 16. This will ensure that container's child processes can't gain more privileges than its parent.

Figure 16 – Content *test.yaml* file with allowPrivilegeEscalation property false

In this case, the pod was successfully created since the configuration is compliant with the policy. The *test.yaml* file has its security context property allowPrivilegeEscalation: false,

meaning that the pod does not allow privilege escalation. Next, in Figure 17 is presented the output of the pod scheduling command.

Figure 17 – Content *test.yaml* file with allowPrivilegeEscalation property false

As for the default configuration where allowPrivilegeEscalation: true, the pod scheduling was denied by Azure Policy since there is a policy enforcing clusters to not allow privilege escalation. As presented next in Figure 18, Azure Policy blocked the creation of the pod.

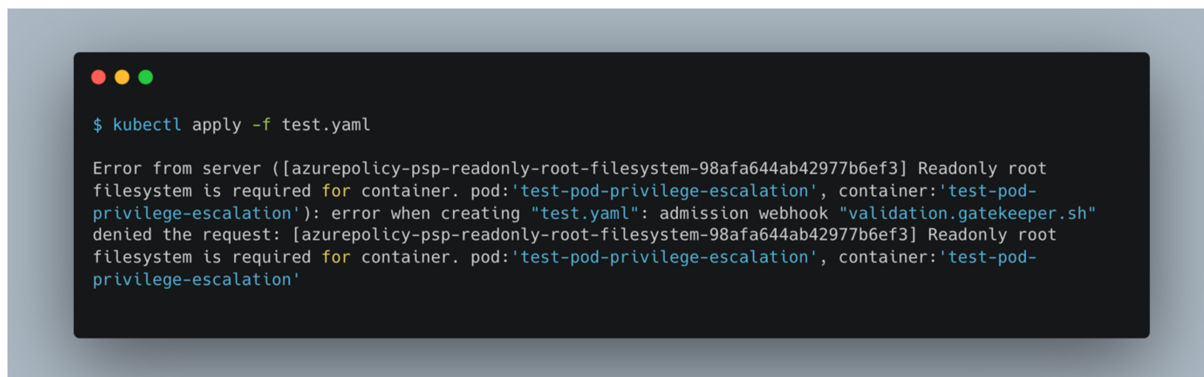

Figure 18 – Content *test.yaml* file with allowPrivilegeEscalation property false

6.2.5 Kubernetes cluster services should only use allowed external IPs

To validate this policy, the process of validation will follow the *kubernetes.io* tutorial to expose external IP addresses to access an application in a Cluster [28].

Firstly, for this example, it is created a simple service for an application that is running in 5 pods, as presented in Figure 19, *test.yaml* file.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app.kubernetes.io/name: load-balancer-example
  name: hello-world
spec:
  replicas: 5
  selector:
    matchLabels:
      app.kubernetes.io/name: load-balancer-example
  template:
    metadata:
      labels:
        app.kubernetes.io/name: load-balancer-example
    spec:
      containers:
      - image: gcr.io/google-samples/node-hello:1.0
        name: hello-world
        ports:
        - containerPort: 8080
```

Figure 19 – Content of *test.yaml* file

Then, like on the validations done before, it is used the *kubectl apply* command to schedule the service, named "hello-world".

The next step was to create a service Object that exposes the deployment to a specific IP. It was defined a random external IP, different from the ones specified to be allowed on the Policy. In Figure 20, is described the command used to expose the deployment. For the case, it was used the 1.1.1.1 IP Address.

```
$ kubectl expose deployment hello-world --type=LoadBalancer --name=my-service --external-ip 1.1.1.1
```

Figure 20 – *Kubectl expose* command

As expected, since there is a Policy assigned refusing external IPs other than the ones specified to be allowed, Azure Policy denied the *kubectl expose* command because it was created with a forbidden external IP, as presented next in Figure 23.

```
$ kubectl expose deployment hello-world --type=LoadBalancer --name=my-service --external-ip 1.1.1.1

Error from server ([azurepolicy-k8sazureexternalips-a961715c36718324330c] service has forbidden
external IPs: {"1.1.1.1"}): admission webhook "validation.gatekeeper.sh" denied the request:
[azurepolicy-k8sazureexternalips-a961715c36718324330c] service has forbidden external IPs: {"1.1.1.1"}
```

Figure 21 – Result of the *kubectl expose* command with a forbidden external IP

Next, in Figure 22, is shown the result of the *kubectl expose* command when using an allowed external IP. As expected, the service was exposed correctly.

```
$ kubectl expose deployment hello-world --type=LoadBalancer --name=my-service --external-ip [illegible].205

service/my-service exposed
```

Figure 22 – Result of the *kubectl expose* command with an allowed external IP

Chapter 7
Planning

In this chapter will be approached the planning of the project. This project will be mainly divided into two semesters, the first semester, and the second semester.

In section 7.1 will be addressed the work plan for the **First Semester**, in section 7.2 will be addressed the work plan for the **Second Semester**. Section 7.3 will cover the **Risk Management**, and finally, section 7.4 will approach the **Methodology**.

7.1 First Semester

The work performed during the first semester focused initially on studying the State of The Art of policy-based tools to assure compliance in Cloud-Native environments. Then was made a research about the Cloud-Native technologies being used in the organization followed by a presentation of the most suitable solution to assure compliance.

A more detailed description of each task to be performed during the first semester is presented next:

- **Task 1 – State of The Art –** This task was based on studying the tools available to implement security policies in a Cloud-Native environment.

- **Task 2 – Study Cloud-Native Technologies –** This task was mainly based on studying the Cloud-Native tools and technologies that are being used in our organization to be able to find the most suitable solution.

- **Task 3 – Policy Compliance Solution –** For this task, were made various discussions with the organization's cyber-security team about which solution/solutions could fit the best.

- **Task 4 – Write Intermediate Report –** The Intermediate Report was written since the State of The Art was mainly done. The State of The Art was the starting point for the Intermediate Report writing.

2021															2022				
October					November					December					January				
W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11	W12	W13	W14	W15	W16	W17	W18	W19	W20

Task 1 - State of The Art

Task 2 - Study Cloud-Native Technologies

Task 3 - Policy Compliance Solution

Task 4 - Write Intermediate Report

Figure 23 – First Semester Work Plan

7.2 Second Semester

The work performed during the second semester was mainly divided into 6 tasks. The **Planning with Team** task (Task 1), the **Work Environment Preparation** task (Task 2), the **Policies Definition** task (Task 3), the **Apply Policies** task (Task 4), the **Validation** task (Task 5), and finally, the **Write Final Report** task (Task 6).

A more detailed description of each task to be performed during the second semester is presented next:

- **Task 1 – Planning with Team –** This task aims to plan with the team how the work should flow, define the most suitable tool/service to use to implement the policies, and what would be the best approach to follow so the system Availability was not compromised, the most adequate role to have for this job, and so on.

- **Task 2 – Work Environment Preparation –** This task aims to prepare the work environment to implement and test policies. In this case, preparing the staging cluster to support the Azure Policy add-on which is the intermediary that syncs Policies defined in Azure with Gatekeeper, the policy validator. Other than that,

it's needed to prepare the local machine with specific tools to interact with ANOVAS's Azure subscriptions by the terminal.

- **Task 3 – Policies Definition –** This task will be a starting point for the implementation of policies. It will begin with the definition of which policies are actually needed as well as which policies could also be implemented in order to improve security.

- **Task 4 – Apply Policies –** In this task will be collected and implemented/applied all the policies defined in the step before.

- **Task 5 – Validation –** In this task will be validated the functionality of the policies, see if they are working as expected as well as fix some issues that could have appeared.

- **Task 6 – Write Final Report –** This task will start as soon as the planning with team is completed. It will accompany all the practical work done throughout the semester.

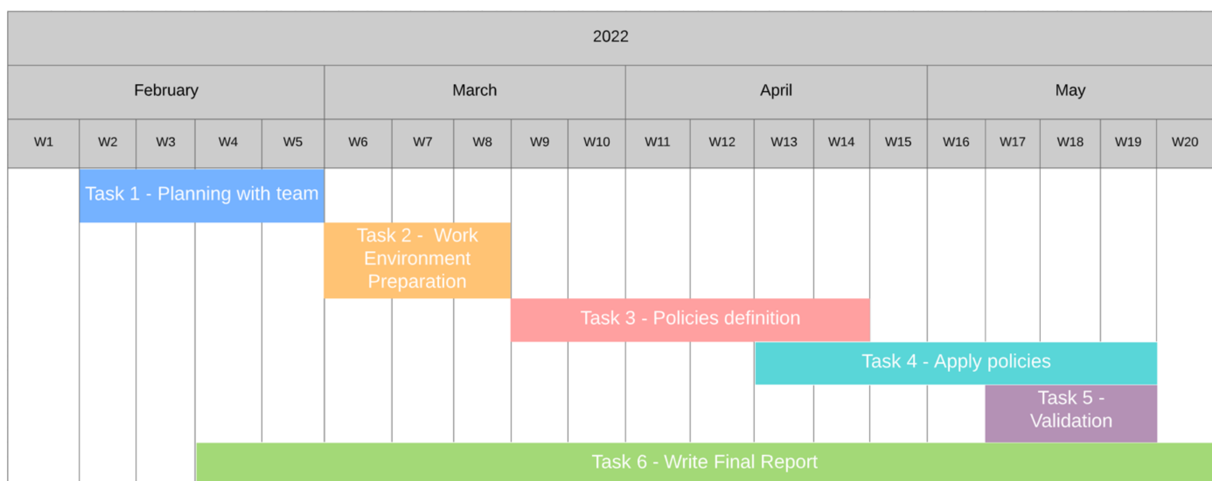

Figure 24 – Second Semester Work Plan

7.3 Risk Management

During the development of a project, there is always a chance that certain types of events will have a negative impact on the project, so at an early stage, it is important to list and analyze the potential risks that could impact the success of the project. In this context, the risk analysis step arises, which aims, for each identified risk, to calculate its probability of occurrence and the level of impact on the project and also to try to outline a strategy to prevent the risk from happening.

In this section will be identified and described the risks that might affect the success of the project. These risks will be classified according to a scale presented next, corresponding to the Probability of happening and respective Impact on the project, followed by their possible mitigation, and finally, a Status, observed or not observed, that represents if the risk really occurred during the project development or not, respectively.

Probability level can vary on a scale from "**0 – Very Low Probability"** to **"5 – Very High Probability"**.

On the other side, impact level can vary on a scale from "**0 – No Impact"** to **"5 – Very High Impact"**.

Table IX – Risk 1 – Working full-time

Risk	Working full-time
Description	Working in a full-time job at the same time as doing the internship (although there is one week per sprint where I can focus specifically on internship work) and a discipline for the master's degree might affect the quality of the work since the time needed to be distributed between these three commitments.
Probability	4
Impact	4

Mitigation	Try to plan the tasks as best as possible as well as the time needed to be dedicated to each task may be the best way to take.
Status	Observed

Table X – Risk 2 – Working in Frontend field

Risk	Working in Frontend field
Description	Since I work as a Frontend developer and this subject encompasses several terms and techniques that have nothing to do with Frontend development, this will probably increase the difficulty of the project.
Probability	5
Impact	4
Mitigation	Try to do a preliminary study that covers the largest number of terms, techniques, and technologies to be used during the implementation phase, as well as share some knowledge with experienced people in the organization.
Status	Observed

Table XI – Risk 3 – Some policies may not be applied within the time

Risk	Some policies may not be applied within the time
Description	Since all policies will be enforced over already existing resources that are constantly being used, it may not be possible to apply some of them before a proper preparation of the resources, which could lead to a delay.

Probability	3
Impact	3
Mitigation	Try to identify earlier which policies can need some preparation before so that is possible to apply them within time.
Status	Observed

Next, in Figure 25, there is a presentation of the risks identified for this project, in a Risk Exposure Matrix where the Xaxis scale goes from an **Insignificant** consequence to a **Severe** impact, and the Yaxis scale goes from a **Rare** likelihood to a **Very Likely** to happen probability.

In this matrix, Risk 1 is represented by R-1, Risk 2 is represented by R-2, and finally, Risk 3 is represented by R-3.

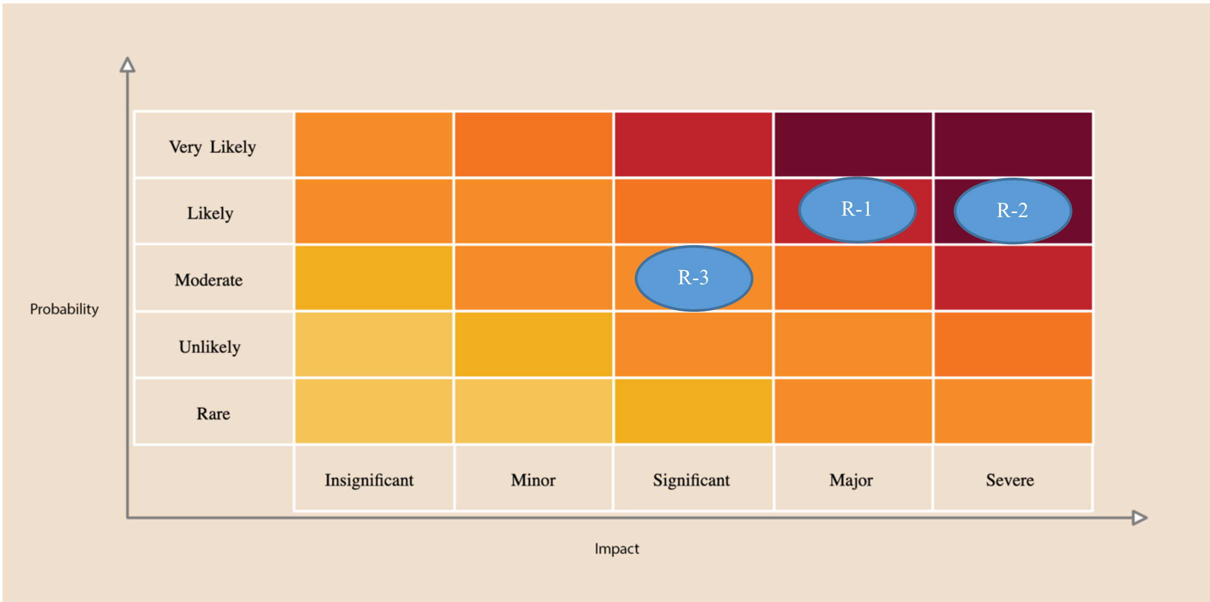

Figure 25 – Risk Exposure Matrix

As presented in Figure 25, it is possible to realize that Risk 1 (R-1) occupies a position **Likely** to happen with a possible **Major** impact. For Risk 2 (R-2), it occupies a position **Likely** to happen with a possible **Severe** impact. As for Risk 3 (R-3), it occupies a position with a **Moderate** likelihood, with a **Significant** impact.

7.4 Methodology

During the planning stage of a project, it is fundamental to decide how to organize a team in the different tasks to be executed. In this sense, it is necessary to define a working methodology. The methodology will shape the team's work method in order to organize it in the most efficient way in an effort to reduce possible risks that may occur. There are two main groups when it comes to existing work methodologies, traditional and agile. The first one stands out for delivering the product at the end of its production. In this methodology, the phases of software development are performed in a specific order, the next phase is started only when the one that is being executed, is completed. On the other hand, agile methodologies are distinguished by having small deliveries throughout the development of the product and greater contact with the customer. In this type of approach, the analysis, development, testing, integration, and validation phases are executed in small tasks, with the goal of developing and delivering small parts of the final product and presenting them to the customer to obtain their feedback more frequently.

For the development of this project, it was used an agile methodology based on Scrum. This working method consists of production cycles known as sprints. Sprints are defined as small iterations of work, where the tasks to be performed are defined, as well as the development methods. At the end of each sprint, usually occurs a meeting to review the work done so far and plan the next iteration. During the course of the internship, this was the working method used with a slightly different change regarding the meetings. In this methodology, these meetings are usually with all the team elements, however, due to the difficulty of bringing all the members together at the same time, meetings were held throughout the year individual depending on the working section and team availability. Despite this slight adaptation, a group chat conversation with the team members was used to discuss different questions that may have appeared at any moment.

Chapter 8
Conclusion and Future Work

Although Cloud-Native technologies can bring many advantages for organizations, in terms of scalability, reliability, provisioning, reduced costs, and so on, it's very important to keep in mind that they also bring a wider attack surface to be explored by malicious parties or even for developers to commit mistakes that could compromise a whole system.

Looking at the Kubernetes side, typically, malicious parties look at ways of taking control of the host worker node by hacking into a Kubernetes application. Then they can use that opportunity to shut down clusters or exploit them for malicious activities. One way of preventing such control is by enforcing proper security policies, without impacting development speed and adding admin overhead.

If an organization is thinking about defining security policies specifically for Kubernetes, Kyverno could be the best option to take since it is a powerful and easy-to-use policy engine designed specifically for Kubernetes that doesn't require learning new languages and adopting different tools to manage policies. On the other hand, if it is needed to write more complex policies, Kyverno will probably struggle and OPA Gatekeeper could be a better option [29].

If the goal of an organization is to define security policies across multiple Cloud-Native technologies, like Kubernetes, Terraform, Docker, SQL Databases, Kafka, and so on, maybe using Open Policy Agent as a unified platform could be the best solution. If the organization's infrastructure is built over Azure services, the wiser choice to take might be Azure Policy by Microsoft. Microsoft provides countless policies, from cost management to security policies, which are maintained by them, that can be used in a pretty straightforward way inside Azure Portal.

8.1 Future Work

Regarding future work, firstly, it's important to summarize what was achieved throughout the project.

For this project context, it was discovered the best approach and the best tool/service to use in order to implement and monitor security policies across multiple technologies in

ANOVA’s Cloud-Native environment, the Azure Policy. The most suitable and significant policies were also defined and assigned to the scope used during the process (staging cluster). Some policies were defined, set with the right parameters, and all the resources are already in compliance with them, which means that those policies, hypothetically, are ready to be assigned to production scope as well.

As for the future work itself, the goal is to continue the process until all the policies are compliant and ready to be assigned to production scope. For this to be achieved, some changes and adaptations will need to be addressed by ANOVA’s Infrastructure team to the existing resources in order to support the policies without compromising services availability. In addition, currently, some of the non-compliant policies are set with the “Audit” *effect*, meaning that those policies are only evaluating resources, instead of blocking their creation or update if they are non-compliant.

References

[1] "What is Cloud Native? | LinkedIn." https://www.linkedin.com/pulse/o-que-%C3%A9-nativo-de-nuvem-cloud-native-renato-souza/?originalSubdomain=pt (accessed Jan. 08, 2022).

[2] "Going Cloud Native: 6 essential things you need to know." www.weave.works/use-cases/going-cloud-native-6-essential-things-you-need-to-know (accessed Jan. 08, 2022).

[3] "Microservices vs Monolith: which architecture is the best choice for your business?" https://www.n-ix.com/microservices-vs-monolith-which-architecture-best-choice-your-business/ (accessed Jan. 08, 2022).

[4] "Codified Security and Compliance with Policy as Code." https://www.weave.worksurl! (accessed Apr. 04, 2022).

[5] "Benefits of Policy as Code," *pulumi*. https://www.pulumi.com/blog/benefits-of-policy-as-code/ (accessed Apr. 13, 2022).

[6] "Why DevOps are choosing Policy-as-Code," *Apolicy*, Mar. 08, 2021. https://apolicy.io/why-devops-are-choosing-policy-as-code/ (accessed Apr. 13, 2022).

[7] "CNCF Cloud Native Interactive Landscape," *CNCF Cloud Native Interactive Landscape*. https://landscape.cncf.io/?category=database (accessed Apr. 27, 2022).

[8] "The Origin of Open Policy Agent and Rego." https://blog.styra.com/blog/origin-of-open-policy-agent-rego (accessed Jan. 04, 2022).

[9] George Wallace, "Overview of Azure Policy - Azure Policy." https://docs.microsoft.com/en-us/azure/governance/policy/overview (accessed Jan. 16, 2022).

[10] zr-msft, "Use Azure Policy to secure your cluster - Azure Kubernetes Service." https://docs.microsoft.com/en-us/azure/aks/use-azure-policy (accessed Jun. 08, 2022).

[11] timwarner-msft, "List of built-in policy definitions - Azure Policy." https://docs.microsoft.com/en-us/azure/governance/policy/samples/built-in-policies (accessed Jun. 08, 2022).

[12] Geert Baeke, *Admission Control on AKS with Azure Policy*, (Mar. 26, 2021). Accessed: Jul. 02, 2022. [Online Video]. Available: https://www.youtube.com/watch?v=OJGmwCMsUNE

[13] bmansheim, "What is Microsoft Defender for Cloud?" https://docs.microsoft.com/en-us/azure/defender-for-cloud/defender-for-cloud-introduction (accessed May 26, 2022).

[14] “Open Source Container Security Tools: Falco,” *Sysdig*. https://sysdig.com/opensource/falco/ (accessed Jan. 04, 2022).

[15] D. Decker, “Securing Kubernetes with K-rail,” *Cruise*, Apr. 03, 2020. https://medium.com/cruise/securing-kubernetes-with-k-rail-5f77a1a11174 (accessed Jan. 04, 2022).

[16] “Harbor.” https://goharbor.io/ (accessed Jan. 08, 2022).

[17] “Conftest joins the Open Policy Agent project,” *Cloud Native Computing Foundation*, Jul. 23, 2020. https://www.cncf.io/blog/2020/07/23/conftest-joins-the-open-policy-agent-project/ (accessed Jan. 20, 2022).

[18] “Validate Resources,” *Kyverno*. https://kyverno.io/docs/writing-policies/validate/ (accessed Jan. 04, 2022).

[19] “Mutate Resources,” *Kyverno*. https://kyverno.io/docs/writing-policies/mutate/ (accessed Jan. 04, 2022).

[20] “Generate Resources,” *Kyverno*. https://kyverno.io/docs/writing-policies/generate/ (accessed Jan. 04, 2022).

[21] “OPA Gatekeeper: Policy and Governance for Kubernetes,” *Kubernetes*, Aug. 06, 2019. https://kubernetes.io/blog/2019/08/06/opa-gatekeeper-policy-and-governance-for-kubernetes/ (accessed Jan. 04, 2022).

[22] “Kubernetes Policy Comparison: OPA/Gatekeeper vs Kyverno.” https://neonmirrors.net/post/2021-02/kubernetes-policy-comparison-opa-gatekeeper-vs-kyverno/ (accessed Jan. 04, 2022).

[23] “Kyverno Policies,” *Kyverno*. https://kyverno.io/policies/ (accessed Jan. 24, 2022).

[24] C. C. Editor, “information security policy - Glossary | CSRC.” https://csrc.nist.gov/glossary/term/information_security_policy (accessed Jun. 08, 2022).

[25] “IT Security Policy: Must-Have Elements and Tips,” *https://blog.netwrix.com/*. https://blog.netwrix.com/2021/02/25/security-policy/ (accessed Jun. 08, 2022).

[26] sumeetmittal, “Azure virtual network service endpoints.” https://docs.microsoft.com/en-us/azure/virtual-network/virtual-network-service-endpoints-overview (accessed Jun. 28, 2022).

[27] shohamMSFT, “Transparent data encryption (TDE) - SQL Server.” https://docs.microsoft.com/en-us/sql/relational-databases/security/encryption/transparent-data-encryption (accessed Jun. 28, 2022).

[28] “Exposing an External IP Address to Access an Application in a Cluster,” *Kubernetes*. https://kubernetes.io/docs/tutorials/stateless-application/expose-external-ip-address/ (accessed Jul. 02, 2022).

[29] DevOps Toolkit, *Kubernetes Policy Management Tools Compared - OPA with Gatekeeper vs. Kyverno*, (Jul. 01, 2021). Accessed: Jan. 16, 2022. [Online Video]. Available: https://www.youtube.com/watch?v=9gSrRNmmKBc

[30] timwarner-msft, "Details of the policy definition structure - Azure Policy." https://docs.microsoft.com/en-us/azure/governance/policy/concepts/definition-structure (accessed Jul. 04, 2022).

Page intentionally left blank

Appendix A – Policies JSON Files

This appendix presents the JSON files of each policy approached during the Implementation and Validation process, detailed in Chapter 6. These JSON files correspond to the definition of each built-in policy, provided by Microsoft, and differ from policy to policy. As noticeable in the file's content, these policies can receive multiple different arguments for their configuration.

Policies can contain elements for [30]:

- **Display name** – policy name
- **Description** – policy description
- **Mode** – which resource types should be evaluated
- **Metadata** – version, category, preview, deprecated, and portalReview
- **Parameters** – the parameters that each policy could receive
- **Policy Rule** – logical evaluation and its effect (Audit, Deny, Disabled, etc)

Policy 1 - Kubernetes clusters should not allow privileged containers

```
{
  "properties": {
    "displayName": "Kubernetes cluster should not allow privileged containers",
    "policyType": "BuiltIn",
    "mode": "Microsoft.Kubernetes.Data",
    "description": "Do not allow privileged containers creation in a Kubernetes cluster. This recommendation is
part of CIS 5.2.1 which is intended to improve the security of your Kubernetes environments. This policy is
generally available for Kubernetes Service (AKS), and preview for AKS Engine and Azure Arc enabled Kubernetes.
For more information, see https://aka.ms/kubepolicydoc.",
    "metadata": {
      "version": "8.0.0",
      "category": "Kubernetes"
    },
    "parameters": {
      "effect": {
        "type": "String",
        "metadata": {
          "displayName": "Effect",
          "description": "'Audit' allows a non-compliant resource to be created, but flags it as non-compliant.
'Deny' blocks the resource creation. 'Disable' turns off the policy."
        },
        "allowedValues": [
          "audit",
          "Audit",
          "deny",
          "Deny",
          "disabled",
          "Disabled"
        ],
        "defaultValue": "Deny"
      },
      "excludedNamespaces": {
        "type": "Array",
        "metadata": {
          "displayName": "Namespace exclusions",
          "description": "List of Kubernetes namespaces to exclude from policy evaluation. System namespaces
\"kube-system\", \"gatekeeper-system\" and \"azure-arc\" are always excluded by design."
        },
        "defaultValue": [
          "kube-system",
          "gatekeeper-system",
          "azure-arc"
        ]
      },
      "namespaces": {
        "type": "Array",
        "metadata": {
          "displayName": "Namespace inclusions",
          "description": "List of Kubernetes namespaces to only include in policy evaluation. An empty list
means the policy is applied to all resources in all namespaces."
        },
        "defaultValue": []
      },
      "labelSelector": {
        "type": "Object",
        "metadata": {
          "displayName": "Kubernetes label selector",
          "description": "Label query to select Kubernetes resources for policy evaluation. An empty label
selector matches all Kubernetes resources."
        },
        "defaultValue": {},
        "schema": {
          "description": "A label selector is a label query over a set of resources. The result of matchLabels
and matchExpressions are ANDed. An empty label selector matches all resources.",
          "type": "object",
          "properties": {
            "matchLabels": {
              "description": "matchLabels is a map of {key,value} pairs.",
              "type": "object",
              "additionalProperties": {
                "type": "string"
              },
              "minProperties": 1
            },
```

Figure 26 – Policy 1 definition (1st part)

```
              "matchExpressions": {
                "description": "matchExpressions is a list of values, a key, and an operator.",
                "type": "array",
                "items": {
                  "type": "object",
                  "properties": {
                    "key": {
                      "description": "key is the label key that the selector applies to.",
                      "type": "string"
                    },
                    "operator": {
                      "description": "operator represents a key's relationship to a set of values.",
                      "type": "string",
                      "enum": [
                        "In",
                        "NotIn",
                        "Exists",
                        "DoesNotExist"
                      ]
                    },
                    "values": {
                      "description": "values is an array of string values. If the operator is In or NotIn, the
values array must be non-empty. If the operator is Exists or DoesNotExist, the values array must be empty.",
                      "type": "array",
                      "items": {
                        "type": "string"
                      }
                    }
                  },
                  "required": [
                    "key",
                    "operator"
                  ],
                  "additionalProperties": false
                },
                "minItems": 1
              }
            },
            "additionalProperties": false
          }
        },
        "excludedContainers": {
          "type": "Array",
          "metadata": {
            "displayName": "Containers exclusions",
            "description": "The list of InitContainers and Containers to exclude from policy evaluation. The
identify is the name of container. Use an empty list to apply this policy to all containers in all namespaces."
          },
          "defaultValue": []
        },
        "excludedImages": {
          "type": "Array",
          "metadata": {
            "displayName": "Image exclusions",
            "description": "The list of InitContainers and Containers to exclude from policy evaluation. The
identifier is the image of container. Prefix-matching can be signified with `*`. For example:
`myregistry.azurecr.io/istio:*`. It is recommended that users use the fully-qualified Docker image name (e.g.
start with a domain name) in order to avoid unexpectedly exempting images from an untrusted repository."
          },
          "defaultValue": []
        }
      },
      "policyRule": {
        "if": {
          "field": "type",
          "in": [
            "AKS Engine",
            "Microsoft.Kubernetes/connectedClusters",
        }
      }
    },
    "id": "/providers/Microsoft.Authorization/policyDefinitions/95edb821-ddaf-4404-9732-666045e056b4",
    "type": "Microsoft.Authorization/policyDefinitions",
    "name": "95edb821-ddaf-4404-9732-666045e056b4"
}
```

Figure 27 – Policy 1 definition (2nd part)

Policy 2 - SSL connection should be enabled for PostgreSQL database servers

```
{
  "properties": {
    "displayName": "Enforce SSL connection should be enabled for PostgreSQL database servers",
    "policyType": "BuiltIn",
    "mode": "Indexed",
    "description": "Azure Database for PostgreSQL supports connecting your Azure Database for PostgreSQL server
to client applications using Secure Sockets Layer (SSL). Enforcing SSL connections between your database server
and your client applications helps protect against 'man in the middle' attacks by encrypting the data stream
between the server and your application. This configuration enforces that SSL is always enabled for accessing
your database server.",
    "metadata": {
      "version": "1.0.1",
      "category": "SQL"
    },
    "parameters": {
      "effect": {
        "type": "String",
        "metadata": {
          "displayName": "Effect",
          "description": "Enable or disable the execution of the policy"
        },
        "allowedValues": [
          "Audit",
          "Disabled"
        ],
        "defaultValue": "Audit"
      }
    },
    "policyRule": {
      "if": {
        "allOf": [
          {
            "field": "type",
            "equals": "Microsoft.DBforPostgreSQL/servers"
          },
          {
            "field": "Microsoft.DBforPostgreSQL/servers/sslEnforcement",
            "exists": "true"
          },
          {
            "field": "Microsoft.DBforPostgreSQL/servers/sslEnforcement",
            "notEquals": "Enabled"
          }
        ]
      },
      "then": {
        "effect": "[parameters('effect')]"
      }
    }
  },
  "id": "/providers/Microsoft.Authorization/policyDefinitions/d158790f-bfb0-486c-8631-2dc6b4e8e6af",
  "type": "Microsoft.Authorization/policyDefinitions",
  "name": "d158790f-bfb0-486c-8631-2dc6b4e8e6af"
}
```

Figure 28 – Policy 2 definition

Policy 3 - Kubernetes cluster containers should run with a read only root file system

```
{
  "properties": {
    "displayName": "Kubernetes cluster containers should run with a read only root file system",
    "policyType": "BuiltIn",
    "mode": "Microsoft.Kubernetes.Data",
    "description": "Run containers with a read only root file system to protect from changes at run-time with
malicious binaries being added to PATH in a Kubernetes cluster. This policy is generally available for
Kubernetes Service (AKS), and preview for AKS Engine and Azure Arc enabled Kubernetes. For more information, see
https://aka.ms/kubepolicydoc.",
    "metadata": {
      "version": "4.2.1",
      "category": "Kubernetes"
    },
    "parameters": {
      "effect": {
        "type": "String",
        "metadata": {
          "displayName": "Effect",
          "description": "'Audit' allows a non-compliant resource to be created or updated, but flags it as non-
compliant. 'Deny' blocks the non-compliant resource creation or update. 'Disabled' turns off the policy."
        },
        "allowedValues": [
          "audit",
          "Audit",
          "deny",
          "Deny",
          "disabled",
          "Disabled"
        ],
        "defaultValue": "Audit"
      },
      "namespaces": {
        "type": "Array",
        "metadata": {
          "displayName": "Namespace inclusions",
          "description": "List of Kubernetes namespaces to only include in policy evaluation. An empty list
means the policy is applied to all resources in all namespaces."
        },
        "defaultValue": []
      },
      "labelSelector": {
        "type": "Object",
        "metadata": {
          "displayName": "Kubernetes label selector",
          "description": "Label query to select Kubernetes resources for policy evaluation. An empty label
selector matches all Kubernetes resources."
        },
        "defaultValue": {},
        "schema": {
          "description": "A label selector is a label query over a set of resources. The result of matchLabels
and matchExpressions are ANDed. An empty label selector matches all resources.",
          "type": "object",
          "properties": {
            "matchLabels": {
              "description": "matchLabels is a map of {key,value} pairs.",
              "type": "object",
              "additionalProperties": {
                "type": "string"
              },
              "minProperties": 1
            },
```

Figure 29 – Policy 3 definition (1st part)

```
          "matchExpressions": {
            "description": "matchExpressions is a list of values, a key, and an operator.",
            "type": "array",
            "items": {
              "type": "object",
              "properties": {
                "key": {
                  "description": "key is the label key that the selector applies to.",
                  "type": "string"
                },
                "operator": {
                  "description": "operator represents a key's relationship to a set of values.",
                  "type": "string",
                  "enum": [
                    "In",
                    "NotIn",
                    "Exists",
                    "DoesNotExist"
                  ]
                },
                "values": {
                  "description": "values is an array of string values. If the operator is In or NotIn, the
values array must be non-empty. If the operator is Exists or DoesNotExist, the values array must be empty.",
                  "type": "array",
                  "items": {
                    "type": "string"
                  }
                }
              },
              "required": [
                "key",
                "operator"
              ],
              "additionalProperties": false
            },
            "minItems": 1
          }
        },
        "additionalProperties": false
      }
    },
  },
  "policyRule": {
    "if": {
      "field": "type",
      "in": [
        "AKS Engine",
        "Microsoft.Kubernetes/connectedClusters",
        "Microsoft.ContainerService/managedClusters"
      ]
    },
    "then": {
      "effect": "[parameters('effect')]",
      "details": {
        "constraintTemplate": "https://store.policy.core.windows.net/kubernetes/read-only-root-
filesystem/v3/template.yaml",
        "constraint": "https://store.policy.core.windows.net/kubernetes/read-only-root-
filesystem/v3/constraint.yaml",
        "excludedNamespaces": "[parameters('excludedNamespaces')]",
        "namespaces": "[parameters('namespaces')]",
        "labelSelector": "[parameters('labelSelector')]",
        "values": {
          "excludedContainers": "[parameters('excludedContainers')]",
          "excludedImages": "[parameters('excludedImages')]"
        }
      }
    }
  }
 },
 "id": "/providers/Microsoft.Authorization/policyDefinitions/df49d893-a74c-421d-bc95-c663042e5b80",
 "type": "Microsoft.Authorization/policyDefinitions",
 "name": "df49d893-a74c-421d-bc95-c663042e5b80"
}
```

Figure 30 – Policy 3 definition (2nd part)

Policy 4 - Kubernetes clusters should not allow container privilege escalation

```
{
  "properties": {
    "displayName": "Kubernetes clusters should not allow container privilege escalation",
    "policyType": "BuiltIn",
    "mode": "Microsoft.Kubernetes.Data",
    "description": "Do not allow containers to run with privilege escalation to root in a Kubernetes cluster.
This recommendation is part of CIS 5.2.5 which is intended to improve the security of your Kubernetes
environments. This policy is generally available for Kubernetes Service (AKS), and preview for AKS Engine and
Azure Arc enabled Kubernetes. For more information, see https://aka.ms/kubepolicydoc.",
    "metadata": {
      "version": "6.0.1",
      "category": "Kubernetes"
    },
    "parameters": {
      "effect": {
        "type": "String",
        "metadata": {
          "displayName": "Effect",
          "description": "'Audit' allows a non-compliant resource to be created or updated, but flags it as non-
compliant. 'Deny' blocks the non-compliant resource creation or update. 'Disabled' turns off the policy."
        },
        "allowedValues": [
          "audit",
          "Audit",
          "deny",
          "Deny",
          "disabled",
          "Disabled"
        ],
        "defaultValue": "Audit"
      },
      "namespaces": {
        "type": "Array",
        "metadata": {
          "displayName": "Namespace inclusions",
          "description": "List of Kubernetes namespaces to only include in policy evaluation. An empty list
means the policy is applied to all resources in all namespaces."
        },
        "defaultValue": []
      },
      "labelSelector": {
        "type": "Object",
        "metadata": {
          "displayName": "Kubernetes label selector",
          "description": "Label query to select Kubernetes resources for policy evaluation. An empty label
selector matches all Kubernetes resources."
        },
        "defaultValue": {},
        "schema": {
          "description": "A label selector is a label query over a set of resources. The result of matchLabels
and matchExpressions are ANDed. An empty label selector matches all resources.",
          "type": "object",
          "properties": {
            "matchLabels": {
              "description": "matchLabels is a map of {key,value} pairs.",
              "type": "object",
              "additionalProperties": {
                "type": "string"
              },
              "minProperties": 1
            },
```

Figure 31 – Policy 4 definition (1st part)

```
                  "operator": {
                    "description": "operator represents a key's relationship to a set of values.",
                    "type": "string",
                    "enum": [
                      "In",
                      "NotIn",
                      "Exists",
                      "DoesNotExist"
                    ]
                  },
                  "values": {
                    "description": "values is an array of string values. If the operator is In or NotIn, the
values array must be non-empty. If the operator is Exists or DoesNotExist, the values array must be empty.",
                    "type": "array",
                    "items": {
                      "type": "string"
                    }
                  }
                },
                "required": [
                  "key",
                  "operator"
                ],
                "additionalProperties": false
              },
              "minItems": 1
            }
          },
          "additionalProperties": false
        }
      },
      "allowedExternalIPs": {
        "type": "Array",
        "metadata": {
          "displayName": "Allowed External IPs",
          "description": "List of External IPs that services are allowed to use. Empty array means all external
IPs are disallowed."
        },
        "defaultValue": []
      }
    },
    "policyRule": {
      "if": {
        "field": "type",
        "in": [
          "AKS Engine",
          "Microsoft.Kubernetes/connectedClusters",
          "Microsoft.ContainerService/managedClusters"
        ]
      },
      "then": {
        "effect": "[parameters('effect')]",
        "details": {
          "templateInfo": {
            "sourceType": "PublicURL",
            "url": "https://store.policy.core.windows.net/kubernetes/allowed-external-ips/v1/template.yaml"
          },
          "apiGroups": [
            ""
          ],
          "kinds": [
            "Service"
          ],
          "excludedNamespaces": "[parameters('excludedNamespaces')]",
          "namespaces": "[parameters('namespaces')]",
          "labelSelector": "[parameters('labelSelector')]",
          "values": {
            "allowedExternalIPs": "[parameters('allowedExternalIPs')]"
          }
        }
      }
    }
  },
  "id": "/providers/Microsoft.Authorization/policyDefinitions/d46c275d-1680-448d-b2ec-e495a3b6cc89",
  "type": "Microsoft.Authorization/policyDefinitions",
  "name": "d46c275d-1680-448d-b2ec-e495a3b6cc89"
}
```

Figure 32 – Policy 4 definition (2nd part)

Policy 5 - Kubernetes cluster services should only use allowed external IPs

For this policy, an array of allowed external IPs was defined in Azure Portal. The array is then sent by parameter to the JSON file to be validated.

```json
{
  "properties": {
    "displayName": "Kubernetes cluster services should only use allowed external IPs",
    "policyType": "BuiltIn",
    "mode": "Microsoft.Kubernetes.Data",
    "description": "Use allowed external IPs to avoid the potential attack (CVE-2020-8554) in a Kubernetes cluster. For more information, see https://aka.ms/kubepolicydoc.",
    "metadata": {
      "version": "4.0.1",
      "category": "Kubernetes"
    },
    "parameters": {
      "effect": {
        "type": "String",
        "metadata": {
          "displayName": "Effect",
          "description": "'Audit' allows a non-compliant resource to be created or updated, but flags it as non-compliant. 'Deny' blocks the non-compliant resource creation or update. 'Disabled' turns off the policy."
        },
        "allowedValues": [
          "audit",
          "Audit",
          "deny",
          "Deny",
          "disabled",
          "Disabled"
        ],
        "defaultValue": "Audit"
      },
      "namespaces": {
        "type": "Array",
        "metadata": {
          "displayName": "Namespace inclusions",
          "description": "List of Kubernetes namespaces to only include in policy evaluation. An empty list means the policy is applied to all resources in all namespaces."
        },
        "defaultValue": []
      },
      "labelSelector": {
        "type": "Object",
        "metadata": {
          "displayName": "Kubernetes label selector",
          "description": "Label query to select Kubernetes resources for policy evaluation. An empty label selector matches all Kubernetes resources."
        },
        "defaultValue": {},
        "schema": {
          "description": "A label selector is a label query over a set of resources. The result of matchLabels and matchExpressions are ANDed. An empty label selector matches all resources.",
          "type": "object",
          "properties": {
            "matchLabels": {
              "description": "matchLabels is a map of {key,value} pairs.",
              "type": "object",
              "additionalProperties": {
                "type": "string"
              },
              "minProperties": 1
            },
            "matchExpressions": {
              "description": "matchExpressions is a list of values, a key, and an operator.",
              "type": "array",
              "items": {
                "type": "object",
                "properties": {
                  "key": {
                    "description": "key is the label key that the selector applies to.",
                    "type": "string"
                  },
```

Figure 33 – Policy 5 definition (1st part)

```
                  "operator": {
                    "description": "operator represents a key's relationship to a set of values.",
                    "type": "string",
                    "enum": [
                      "In",
                      "NotIn",
                      "Exists",
                      "DoesNotExist"
                    ]
                  },
                  "values": {
                    "description": "values is an array of string values. If the operator is In or NotIn, the
values array must be non-empty. If the operator is Exists or DoesNotExist, the values array must be empty.",
                    "type": "array",
                    "items": {
                      "type": "string"
                    }
                  }
                },
                "required": [
                  "key",
                  "operator"
                ],
                "additionalProperties": false
              },
              "minItems": 1
            }
          },
          "additionalProperties": false
        }
      },
      "allowedExternalIPs": {
        "type": "Array",
        "metadata": {
          "displayName": "Allowed External IPs",
          "description": "List of External IPs that services are allowed to use. Empty array means all external
IPs are disallowed."
        },
        "defaultValue": []
      }
    },
    "policyRule": {
      "if": {
        "field": "type",
        "in": [
          "AKS Engine",
          "Microsoft.Kubernetes/connectedClusters",
          "Microsoft.ContainerService/managedClusters"
        ]
      },
      "then": {
        "effect": "[parameters('effect')]",
        "details": {
          "templateInfo": {
            "sourceType": "PublicURL",
            "url": "https://store.policy.core.windows.net/kubernetes/allowed-external-ips/v1/template.yaml"
          },
          "apiGroups": [
            ""
          ],
          "kinds": [
            "Service"
          ],
          "excludedNamespaces": "[parameters('excludedNamespaces')]",
          "namespaces": "[parameters('namespaces')]",
          "labelSelector": "[parameters('labelSelector')]",
          "values": {
            "allowedExternalIPs": "[parameters('allowedExternalIPs')]"
          }
        }
      }
    }
  },
  "id": "/providers/Microsoft.Authorization/policyDefinitions/d46c275d-1680-448d-b2ec-e495a3b6cc89",
  "type": "Microsoft.Authorization/policyDefinitions",
  "name": "d46c275d-1680-448d-b2ec-e495a3b6cc89"
}
```

Figure 34 – Policy 5 definition (2nd part)