



UNIVERSIDADE D
COIMBRA

Gabriel Marco Freire Pinheiro

**CI/CD PIPELINES FOR MICROSERVICE-BASED
ARCHITECTURES**

**Dissertation in the context of the Master's Degree in Informatics Engineering,
Specialization in Software Engineering, advised by Professor Nuno Laranjeiro and
Engenheiro Rui Cunha, presented to the Department of Informatics Engineering
of the Faculty of Sciences and Technology of the University of Coimbra.**

July 2022



FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE
COIMBRA
DEPARTMENT OF INFORMATICS ENGINEERING

Gabriel Marco Freire Pinheiro

CI/CD Pipelines for Microservice- Based Architectures

Dissertation in the context of the Master's Degree in Informatics Engineering,
Specialization in Software Engineering, advised by Professor Nuno Laranjeiro and
Engenheiro Rui Cunha, presented to the Department of Informatics Engineering of the
Faculty of Sciences and Technology of the
University of Coimbra.

July 2022

This Page Was Intentionally Left Blank



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

Gabriel Marco Freire Pinheiro

CI/CD Pipelines para Arquiteturas Baseadas em Micro-Serviços

Dissertação no âmbito do Mestrado em Engenharia Informática, especialização em Engenharia de Software, orientada pelo Professor Nuno Laranjeiro e o Engenheiro Rui Cunha e apresentada ao Departamento de Engenharia Informática da Faculdade de Ciências e Tecnologia da Universidade de Coimbra.

Julho 2022

This Page Was Intentionally Left Blank

Aknowledgments

Above all else, I would like to express my gratitude towards WIT Software. The organization has provided me with an amazing opportunity to work with a business of its magnitude, with amazing people and providing me with an atmosphere that, even during a global pandemic, made me feel integrated and valued.

To the three supervisors at WIT software, Eng. Rui Cunha, Eng. Diogo Cunha and Eng. António Mendes, thank you for your time, support, patience and good spirits in all the meetings and different tasks that I had to carry out. Even when things seemed to not go according to plan, you helped me progress and become a better engineer, a better professional and, above that, a better person.

To my supervisor at the University of Coimbra, Professor Nuno Laranjeiro. Although the first time we made contact through my journey in the master's degree was brief, your influence, attention to detail and readiness to help were duly noted and felt throughout the entire proceedings of the internship and the creation of this report.

To my friends, old and new, who have helped me on throughout this journey, thank you.

Finally, and foremost, this report represents the end of a six-year journey of a student of Informatics Engineering. A boy who finished high school, aspiring to become an engineer. One that still works towards becoming one. One that can be proud of who he is, who he has become, proud of the work that he does, that helps contribute towards the world being a better place, and that looks to the future with high hopes and a glimmer in his eye. This way of being could only be what it is thanks to the family which I have been blessed with.

Although all very special and with their own influence on my path to this goal, most importantly, I would like to give a special word of thanks to my mother, my father and my grandmother, Nini. They are the ones to whom I owe everything. Who I am, what I have become. Without them, without their sacrifice and continuous persistence in raising me to the person I am today, I would definitely not be here.

Thank you.

Abstract

This document looks to present the work carried out by Gabriel Pinheiro, within the context of the Dissertation/Internship in Software Engineering course. The original title of the Project is CI/CD Pipelines for Microservice-Based architectures, promoted and coordinated by WIT Software S.A.

In an effort to adapt to changes and the rapid growth of operating within the software development industry, the goal of the internship consists in creating a framework which establishes a standard within software development teams. This framework creates the Foundation on which current and future projects will base themselves, allowing for customization and alterations where necessary to fit the different contexts. The adoption of this product begins the adoption process of a culture based on automation services, in order to create a different view when facing problems related to the back-stage process of software development.

The internship began with a study of concepts that were considered essential for a better understanding of the project in itself. With this study, a cornerstone was formed on which the remainder of the internship's work would be based on, obtaining in-depth knowledge into how cloud-native technologies operate, what a microservices architecture consists of, what are known automation-based development cultures and how they can be adopted, and, finally, how application observability can be obtained when in production and dealing with microservices. Once concluded, a further study was conducted into understanding which are the best practices associated with creating cloud-native infrastructures, along with the best practices required when creating a CI/CD pipeline, and its adoption into a new organization. This study concluded with establishing which the best suited solutions should be used to apply these practices within an organization.

In order to guarantee a final product which met the expectations of the organization, a list of requirements was established with those responsible for the internship itself, along with a detailed planning of the development project, beginning with the phase of infrastructure provisioning, through to the tools made available to operations teams. Furthermore, a graphic representation was created in order to define the framework's architecture.

The final product of this internship looks to apply a change in the organization's culture of software development, beginning with the adoption of a cloud-native infrastructure, which allows for on-demand usage, billed according to the level of use. This means of provisioning allows computational components to be created and made available in a matter of minutes, contrasting strongly to the traditional reality which would require several days for IT teams to give access to developers. The infrastructure is coupled with the CI/CD pipeline which a bridge between said infrastructure and software developers, a pipeline which automates build tasks, including automated tests, code analysis and dependency verification in a consistent and recurring manner, without the loss of software quality. With this foundation in place, along with the culture of automation becoming more intertwined with the development teams, all stakeholders involved with this process can reap its benefits.

Keywords

Cloud, Microservices, DevSecOps, Kubernetes, Docker Container, Continuous Integration, Continuous Delivery, Rolling Deployments, Monitoring.

Resumo

O presente documento descreve o trabalho realizado por Gabriel Pinheiro, no âmbito da unidade curricular Dissertação/Estágio em Engenharia de Software, com o título original “CI/CD Pipelines para arquiteturas baseadas em micro-serviços”, promovido e sob o acolhimento da empresa WIT Software S.A.

De forma a conseguir adaptar a mudanças e ao crescimento de operações na indústria de desenvolvimento de aplicações e software, o objetivo do estágio consistiu na criação de uma plataforma que permitisse criar um padrão único dentro das equipas de desenvolvimento de software. Esta plataforma estabelece um padrão base para atuais e futuros projetos de desenvolvimento, sendo que permite que sejam feitos ajustes e adaptações ao seu funcionamento consoante o projeto que a mesma integrar. Esta adoção de serviços de automação procura criar uma mudança de cultura dentro da organização, de forma que haja uma sensibilidade diferente quando se aborda problemas relacionados com o processo que decorre nos bastidores da criação de software.

Com o intuito de aprendizagem, o estágio iniciou-se com um estudo de conceitos essenciais ao entendimento do objetivo do projeto. Aqui, criou-se um alicerce sob o qual o restante trabalho se iria fundar, obtendo-se conhecimento relacionado com tecnologias *cloud-native*, arquiteturas baseadas em micro-serviços, culturas de automação, metodologias onde a automação é aplicada e a forma como é aplicada, e, por fim, a análise de informação da aplicação já em produção. Tendo conhecimento destes conceitos, estudou-se as melhores práticas associadas à criação de uma infraestrutura em *cloud* para uma aplicação baseadas em micro-serviços, à criação de uma pipeline CI/CD e como esta deveria ser adotada dentro de uma organização, mantendo o aproveitamento das vantagens de serviços *cloud-native*. A fase de estudo terminou com estudo e análise de possíveis soluções que permitem aplicar estas boas práticas num ambiente de produção.

De modo a garantir um produto final que fosse ao encontro das expectativas da organização, foi feito um levantamento de requisitos e um planeamento do projeto de desenvolvimento, desde a fase de provisionamento de infraestrutura até às ferramentas disponibilizadas às equipas de operações, integrando a análise de métricas e mensagens comunicadas pela aplicação. Para além do planeamento do projeto, foi preparado uma representação gráfica da arquitetura das três facetas da pipeline. Isto é, da pipeline responsável por criar a infraestrutura, da pipeline responsável por entregar a aplicação ao seu ambiente de produção, e, por fim, a arquitetura necessária para garantir observabilidade da aplicação.

A solução final deste estágio pretende realizar uma mudança de cultura na organização, começando pela adoção de uma infraestrutura *cloud-native*, onde o seu provisionamento é feito *on-demand* e taxado consoante o nível de utilização. Este processo permite criar componentes computacionais em meros minutos, ao invés de requerer vários dias para equipas de IT realizarem pedidos de vários colaboradores. A infraestrutura é copulada com uma pipeline CI/CD que estabelece uma ponte de ligação entre *developers* de software e a infraestrutura referida anteriormente. Esta pipeline permite automatizar tarefas de *build*, testes unitários e análise de código de forma recorrente e consistente, acelerando o processo de entrega de software aos seus utilizadores. Com este padrão estabelecido em todos os projetos e esta cultura a tornar-se intrínseca a todos as equipas de desenvolvimento, todos os *stakeholders* envolvidos têm algo a desfrutar desta alteração.

Palavras-Chave

Cloud, Microservices, DevSecOps, Kubernetes, Docker Container, Continuous Integration, Continuous Delivery, Rolling Deployments, Monitorização.

Table of Contents

CHAPTER 1	INTRODUCTION	1
1.1	MOTIVATION AND PROBLEM	1
1.2	GOAL	2
1.3	DOCUMENT STRUCTURE	3
CHAPTER 2	BACKGROUND	5
2.1	CLOUD COMPUTING	5
2.2	MICROSERVICES	7
2.3	DEVOPS	9
2.4	CONTINUOUS INTEGRATION/CONTINUOUS DELIVERY	10
2.5	MONITORING MICROSERVICE	11
CHAPTER 3	STATE OF THE ART	12
3.1	INFRASTRUCTURE PROVISIONING	12
3.2	CONTINUOUS INTEGRATION	16
3.3	CONTINUOUS DELIVERY	20
3.4	MONITORING MICROSERVICES	25
CHAPTER 4	METHODOLOGY	30
4.1	AGILE AND SCRUM METHODOLOGIES	30
4.2	PROJECT PLANNING	33
4.3	RISK MANAGEMENT	37
CHAPTER 5	FRAMEWORK REQUIREMENTS	40
5.1	REQUIREMENT PRIORITIZATION	40
5.2	REQUIREMENT TYPES	41
5.3	FUNCTIONAL REQUIREMENTS	41
5.4	NON-FUNCTIONAL REQUIREMENTS	44
CHAPTER 6	FRAMEWORK ARCHITECTURE	46
6.1	INFRASTRUCTURE PROVISIONING	46
6.2	CONTINUOUS INTEGRATION/CONTINUOUS DELIVERY	54
6.3	MONITORING MICROSERVICES	62
6.4	CHALLENGES AND DIFFICULTIES	66
CHAPTER 7	VERIFICATION AND VALIDATION PROCESS	67
7.1	VERIFICATION PROCESS	67
7.2	VALIDATION PROCESS	69
CHAPTER 8	CONCLUSION	73
8.1	PROJECT OVERVIEW	73
8.2	LIMITATIONS AND FUTURE WORK	77
CHAPTER 9	REFERENCES	78
APPENDIX A	– STATE OF THE ART	III
APPENDIX B	- METHODOLOGY	L
APPENDIX C	– SOFTWARE REQUIREMENT SPECIFICATION	XIV
APPENDIX D	– SOFTWARE ARCHITECTURE AND DESIGN	XXXII
APPENDIX E	– ACCEPTANCE TEST PLAN AND REPORT	LXXXII

Acronyms

API – Application Programming Interface
AWS – Amazon Web Services
CD – Continuous Delivery
CI – Continuous Integration
ECR – Elastic Container Repository
EKS – Elastic Kubernetes Service
ELK – Elasticsearch Logstash Kibana
HTTP – Hypertext Transfer Protocol
HCL – HashiCorp Configuration Language
IaaS – Infrastructure as a Service
IaC – Infrastructure as Code
IT – Information Technology
JSON – JavaScript Object Notation
NIST – National Institute of Standards and Technology
PaaS – Platform as a Service
SaaS – Software as a Service
SDK – Software Development Kit
VM – Virtual Machine
VCS – Version Control System
YAML – Yet Another Markup Language

Table of Figures

Figure 1. Legacy vs. Cloud Delivery Models [6]	6
Figure 2. Monolith vs Microservices Architecture [7]	7
Figure 3. Comparison Between Containers and Virtual Machines [10].....	8
Figure 4. Graphical Representation of the DevSecOps Culture [14].....	9
Figure 5. Graphical Representation of CI/CD [16].....	10
Figure 6. Infrastructure Provisioning Conceptual Diagram.....	13
Figure 7. Continuous Integration Conceptual Diagram	16
Figure 8. Blue-Green Deployment [28].....	21
Figure 9. Canary Release Workflow [28]	22
Figure 10. Microservice Monitoring Conceptual Diagram.....	26
Figure 11. Project Complexity [37]	31
Figure 12. Scrum Overview [38]	33
Figure 13. 2nd Semester Plan	35
Figure 14. Infrastructure Provisioning Pre-Requisites Creation Process.....	47
Figure 15. Automation Pipeline Workflow for Infrastructure Deployment	49
Figure 16. Create Environment Activity Diagram.....	50
Figure 17. Full Infrastructure Components Macro-View	53
Figure 18. Pre-Requisite Elements Created by the Development Team Leader	55
Figure 19. Continuous Integration Workflow Diagram.....	56
Figure 20. Continuous Integration Automated Build Activity Diagram	58
Figure 21. Continuous Integration Automated Build Sequence Diagram	59
Figure 22. CI/CD Workflow Production Diagram.....	60
Figure 23. Blue-Green Deployment Activity Diagram.....	61
Figure 24. Macro-view Microservice Monitoring Solution.....	62
Figure 25. Metrics Monitoring Workflow	63
Figure 26. Grafana Dashboards Example	63
Figure 27. Sequence Diagram of the Process Behind Cluster Metrics Analysis	64
Figure 28. Log Analysis Workflow	65
Figure 29. Sequence Diagram for Viewing Pod Metrics	65
Figure 30. Second Semester Gantt Diagram Reality	76

Table of Tables

Table 1. Symbology	12
Table 2. Infrastructure Provisioning Tool Analysis.....	15
Table 3. Continuous Integration Tool Analysis	20
Table 4. Continuous Delivery Tool Analysis.....	25
Table 5. Microservice Monitoring Best Practices.....	26
Table 6. Microservice Monitoring Competitor Analysis 1	29
Table 7. Microservice Monitoring Competitor Analysis 2	29
Table 8. Task Table.....	34
Table 9. Project Lifecycle Milestones.....	36
Table 10. Project Management Documents	37
Table 11. Risk Priority Table.....	37
Table 12. Risk Identification and Impact.....	38
Table 13. Risk Mitigation	38
Table 14. Priority Description.....	41
Table 15. Requirement Types	41
Table 16. Infrastructure Provisioning Base States	70
Table 17. Infrastructure Provisioning Test Cases	71
Table 18. Example Test Results.....	72
Table 19. Failed Test Report for TC-01.....	72
Table 20. Failed Test Report for TC-11.....	72

This Page Was Intentionally Left Blank

Chapter 1

Introduction

This document is part of the Internship/Thesis course of the MSc in Informatics Engineering, conducted in the Faculty of Science and Technology of the University of Coimbra. The global purpose of the document is to present the work conducted in the internship carried out at WIT Software in the 2021/2022 school year. The internship began on the 23rd of September 2021 and concluded on the 4th of July 2022. The work is orientated and supervised by Professor Nuno Laranjeiro (DEI), Eng. Rui Cunha (WIT Software), Eng. Diogo Cunha (WIT Software) and Eng. António Mendes (WIT Software).

WIT Software is a company with more than 20 years of experience in the telecommunications and software development industry, identifying itself as a software exportation organization, with software running in more than 46 countries around the world. WIT prides itself in the numerous projects that have been developed over its years of operation, with the mission of always guaranteeing software that is of high-availability, secure, high performance, scalable and with special care for the end user's experience.

Having begun as a spin-off in the University of Coimbra in 2001, until 2004 WIT software's headquarters were at the *Pedro Nunes* Institute, in Coimbra. Once the enterprise's capacity became too large for the space, in December 2013 the company found a new home in the *Centro de Empresas de Taveiro, Coimbra* building, just a few kilometres outside the city-centre. Today, the company has offices in Coimbra, Porto, Lisbon, Leiria, Dusseldorf (Germany), Reading (United Kingdom) and in Silicon Valley (California, USA), with more than 200 contracted employees.

1.1 Motivation and Problem

The software development industry has felt an exponential growth over the course of the last 5 years. In Portugal alone, the informatics and communications sector contributed approximately 10% towards the country's GDP, reaching a total market share of approximately 20 billion US Dollars by the end of 2022. Moreover, the industry employs approximately 80 000 workers throughout the entire country. When dealing with growth levels as high as these, having as big an impact on the national economy as this, it is essential for organizations to adapt their operations in order to maintain a relevant stance amongst their clients and also to maintain a high level of competitiveness within their market [1].

Cloud providers offer a platform on which several flexible services can be used to install and give access to software in a secure and efficient manner, whilst following the good practices associated with the software development cultures.

With the rise of many new methodologies and cultures which look to improve and accelerate the software development process, there is a strong impact to be felt regarding the time-to-market necessary to deploy new software solutions to their users. The

evolution of the development process and adoption of cloud-native systems is a growing reality in the world of software development, with more and more organizations looking to adopt automation-based methodologies in efforts to improve the quality of their products [2].

With this internship, WIT software looks to create a means of shortening the time-to-market required to produce software, whilst maintaining and guaranteeing security and quality of service through the use of a framework which creates a foundation for the automated process behind software development within the organization, whilst also reducing costs associated with maintenance and on-site infrastructure updates. Given the rate of growth of the software development market, the need for obtaining an advantage on competitors is essential for maintaining a market quota. This opportunity leads to question how a culture such as this can be implemented, given the use of microservices as the architectural paradigm used when developing an application. In this case, there are questions which define the problems that this internship looks to solve. These problems are [3]:

- “How can an application based on microservices run in a cloud-native environment?”;
- “How can changes made to an application be applied in a secure and consistent manner?”;
- “How can an application based on microservices run in a cloud-native environment?”;
- “How can the changes made to the application be deployed into their production environment, without creating downtime and in a way that tests can be carried out whilst deploying, all in an automated fashion?”;
- “If there are multiple microservices making up an application’s architecture, how can these be observed in a centralised location?”.

In summary, the motivation behind this internship is, above all, to find a way of increasing productivity, reducing time-to-market for application changes, whilst maintaining a high level of rigor and quality regarding the software that is developed.

1.2 Goal

The goal of this internship is to develop a prototype of a generic, CI/CD pipeline framework, on which to build the specificities of the different projects within the organization. The framework includes four main epic stories, these being the following:

- **Infrastructure Provisioning:** workflow involving the creation of cloud-native computational components through the use of configuration files which are maintained and versioned using a git-based repository system;
- **Continuous Integration:** automated process which establishes a link between a software developer and a means of preparing software, providing the possibility of running pre-defined automated tests, static code analysis and vulnerability verification on source code files. The product must be an executable artifact, contained in a containerized image;
- **Continuous Delivery:** means of deploying the new version of the containerized image progressively, configurable to follow different rollout patterns;

- **Monitoring Solution:** offer insights and observability into the application's behaviour by extracting resource consumption metrics and by maintaining application logs in a centralized and persistent manner.

The goal associated with Infrastructure Provisioning will consist of an autonomous pipeline which, with a git repository as the source location of the infrastructure's specification files, creates the infrastructure that the application requires to operate in. In its essence, infrastructure developers will solely have to commit their template files to the source repository, with the remainder of the process being conducted by an automation pipeline.

Regarding Continuous Integration, there will be a mechanism in place which will begin with a git repository as the source where the software code is maintained. Here, an application developer will be required to maintain the applications code, the application's manifest file which contains miscellaneous configurations and information. To trigger the workflow, the developer will have to commit the source code along with the manifest file to the git repository of the project. The code is then built, analysed, and tested autonomously according to a *buildspec* file which defines the steps necessary for the application's image to be prepared.

Continuous Delivery is responsible for deploying the application into its production environment in a progressive manner. This phase of the pipeline releases software through the use of phased rollouts, where traffic made to the application is routed between versions in order to perform dynamic acceptance and operational tests. The framework must be prepared to perform these progressive rollouts according to three different strategies. These are [4]: i) Blue-Green Deployments – two environments are running simultaneously, with equal load between them; ii) A/B Testing – multiple instances have different versions running simultaneously; iii) Canary Releases – two different environments run different versions, with the load between the two being divided, incrementing progressively (5%, 10%, 20%, etc).

Finally, the monitoring solution created offers feedback into how the application is behaving when in its production environment. To achieve this, the framework will offer insights into the application's services and their resource consumption, whilst also offering a centralised means of extracting application logs, which will also give Operations the possibility of understanding what might originate a problem with a certain version of an application, and how it can be mitigated.

1.3 Document Structure

This document will be divided into 6 core chapters:

- **2. Background** – a brief explanation of the global concepts which are considered essential to better understand the internship's final product characteristics and goal;
- **3. State of the Art** –an analysis of the tools which offer a direct solution to the requirements of the internship's final product. The comparison between these will be done in order to establish which offers the most viable solution for the problems associated with the internship;
- **4. Methodology** – an explanation of the methodology used for the development process, as well as the planning of milestones and risks that are associated with the project at hand;

- **5. Framework Requirements** – a definition of the requirements that compose the project;
- **6. Framework Architecture** – an analysis of the product’s architecture, within each of the four main areas of study and their involvement with each other;
- **7. Quality Control** – an explanation of the process used to perform verification and validation of the product.

Chapter 2

Background

In this section, there will be a contextualization and explanation of the concepts deemed essential for understanding the internship's scope. Here, we will be explaining what **Cloud Computing** is; what **Microservice Applications** are; what the **DevSecOps** culture stands for and implements in a development team; what **CI/CD** is and how it helps contribute to adopting a DevSecOps culture; and finally, what **application monitoring** is, and how it is essential when dealing with microservice architectures. Each of these concepts will have a small definition of what they are and have to offer in the world of technology, and also their involvement in the internship.

2.1 Cloud Computing

Considering the evolution of all things within the computing realm, cloud computing and cloud-native systems have received a larger adoption on behalf of enterprises and increased the market value and quota over previous years.

Cloud computing is defined by the NIST as *“Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction [5].”* In other words, it is a means of supplying a number of computing resources, which can be configured and altered according to the user's needs, efficiently, causing little time loss and implicating the least amount of expertise and management on behalf of the user.

Many of the characteristics beknown to Cloud computing are the following:

- The concept of being able to access computing resources without the need for investing in the physical, bare metal resources;
- Being able to maintain a billable model where the user is only required to pay for what they use, and the resources are chosen according to needs;
- The ability to scale the resources according to an application's load, without the need for technical knowledge about how the process is conducted;
- The ability to access these very resources from any location around the world.

Cloud computing has relevance to this internship in the sense that the framework being developed in the internship requires an infrastructure to be provisioned in a cloud-native environment, making a number of computational resources available to software developers to, at a later stage, deploy their software and make it available to its users.

Infrastructure as a Service

The National Institute of Standards and Technology define IaaS as a system “where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications. The consumer does not manage or control the underlying cloud infrastructure but

has control over operating systems, storage, and deployed applications; and possibly limited control of select networking components (e.g., host firewalls) [5].”

Infrastructure as a service is a delivery model associated with cloud-native systems in which computational resources are made available to organizations and developers in a billable, pay-as-you-use model. This reflects a reduction in costs, given the fact that the price paid by the user of the resources is only billed according to the use given to the services. For an organization such as WIT Software, this becomes a viable alternative to manually creating bare-metal servers and resources which may not be used efficiently, requiring a large investment in capital that, at a later time, will require more investment from a maintenance and update point of view.

Infrastructure as Code

When adopting an IaaS model, a positive trade-off to this is the possibility of creating infrastructure with the use of code by creating and accessing infrastructure through the use of script definition files and templates, rather than hardware configuration files. From here, multiple environments can be created, both automatically and on-demand, allowing also for a means of replicating environments easily and safely, without having an effect on the original infrastructure.

Although service-orientated architecture, in its essence, offers the possibility of supplying “Everything as a Service”, cloud computing focuses service delivery according to three more specific delivery models: Infrastructure as a Service, Platform as a Service and Software as a Service. These three models offer different levels of abstraction to their users and are usually represented as *layers* within a stack of software subsystems, as verified in Figure 2.

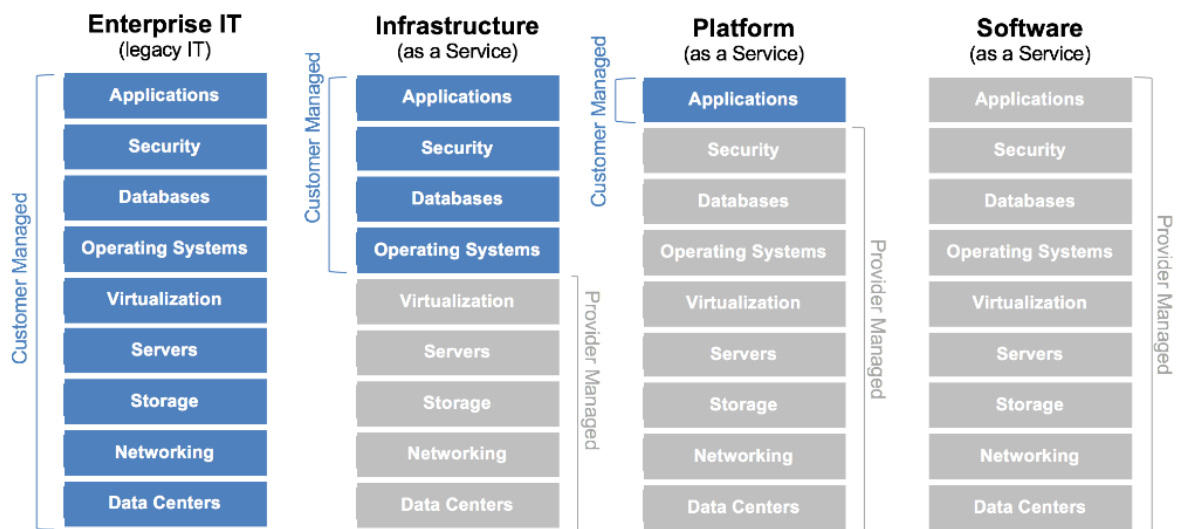


Figure 1. Legacy vs. Cloud Delivery Models [6]

For a more detailed analysis of the different delivery models that cloud computing has to offer, please refer to the document titled Appendix A – State of the Art.

2.2 Microservices

The term Microservices refers to the breaking down of an application into small components which communicate with each other by using lightweight distributed communication methods. Each of these components are a process on their own which together form a suite of small services, each built and defined based on the business capabilities and to what extent they can be deployed independently and operate autonomously [7].

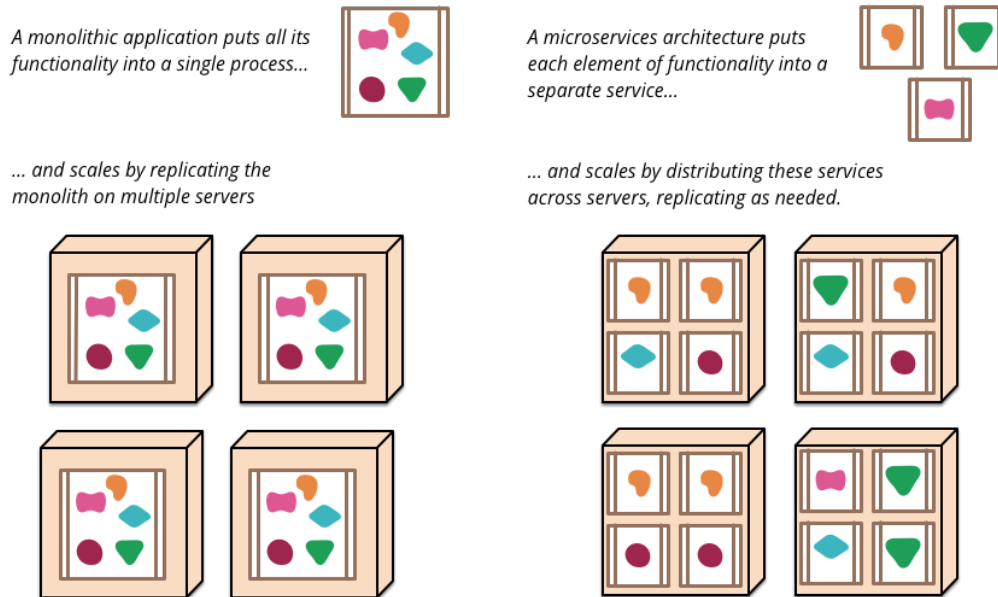


Figure 2. Monolith vs Microservices Architecture [7]

Microservices are a relevant factor toward this internship due to the fact that the architectural model of the application which will run on the cloud-native infrastructure is based on microservices. Given this, the framework developed in this internship must be constructed around this concept whilst guaranteeing that the application itself is operational within each of the development environments that are provisioned. Some of the best practices associated with a microservices architecture are [8]:

- **Network access control:** Implies the restriction of pod-to-pod traffic within a Kubernetes cluster;
- **Patch Management and CI/CD Deployment:** Usage of automated systems which build, test and deploy updates into a Kubernetes cluster

For a question of brevity, two best practices are presented in this document, with the remainder being present and studied with more detail in the document titled Appendix A – State of the Art.

To better understand technical terms associated with microservices, there is a need for a brief understanding of two concepts: **Containerization** and **Kubernetes**.

Containerization

To understand what containerization is, the concept of a Virtual Machine must first be defined. A Virtual Machine (VM) is a computing resource that makes use of software instead of a physical computer to run programs and deploy applications [9]. A VM allocates a suite of

resources for its functionalities to be made available and runs its own operating system separately from other VMs that may be running on the same host. A hypervisor allows multiple VMs to run on a single machine. Each VM includes a full copy of an operating system, the application, necessary binaries and libraries, which occupy large amounts of memory.

The following figure 3 serves as a graphical representation of the differences between containers and Virtual Machines.

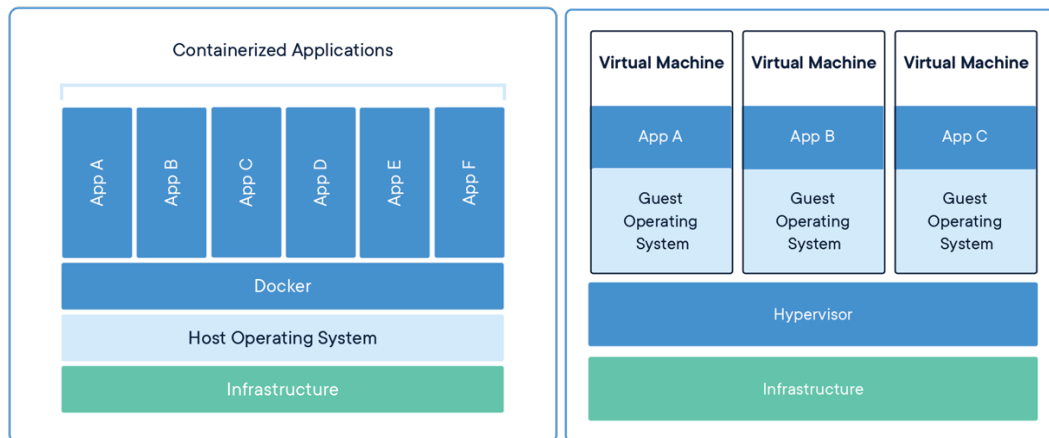


Figure 3. Comparison Between Containers and Virtual Machines [10]

Containers, on the other hand, are standard unit of software which packages application code and all its dependencies into a single executable, allowing the application to run efficiently on multiple computing environments.. Containers take up less space than VMs and can handle more applications and require fewer resources. A container image is a lightweight executable package of software that limits its contents to the bare essentials required by an application to function.

Kubernetes

Kubernetes is an open-source container orchestrations system, responsible for automating software deployments, scaling and management. Kubernetes contains a large number of different components, therefore, for a question of relevance and readability, only those more relevant to the internship will be defined. For a full list of these components and their definitions, please refer to the document Appendix A – State of the Art.

The relevant components are the following:

- **Nodes:** a machine where containers are deployed. Also known as a worker or a minion. Every node must run a container runtime, as well as a component responsible for communication;
- **Pods:** consist of one or more containers that are guaranteed to be co-located on the same node. Each pod in Kubernetes is assigned a unique IP address within the cluster, allowing applications to use ports without the risk of conflict.

2.3 DevOps

The DevOps culture can be defined as the “approach to culture, automation and platform design intended to deliver increased business value and responsiveness through rapid, high-quality service delivery [11]”. In other words, an automated approach to producing and delivering software by means of a large amount of changes being implemented to an application, on a daily basis. Traditionally, software development teams were built and composed of various “silos”. One part of the team would focus on code development, the other on testing, operations, and customer support, *et cetera*. With time, it has been understood that having this separation between teams, although a successful strategy, does not have a long-lasting effect when it comes to quality and customer satisfaction.

By looking at the name itself, DevOps is the literal joining of the terms “Development” and “Operations”. This culture does not limit to just conjugating these words and their consequent departments but also, by including security, collaboration between team members, data analytics, the objective of the DevOps’ culture is to instruct team-members to adopt an attitude of shared responsibility of a product. This increases collaboration in the different tasks that are carried out, thus making code production a much easier task and, subsequently, increasing quality levels within the development process [12].

DevSecOps

This methodology also brings the task of developing security measures to the forefront of development, rather than leaving it at the end of the list of tasks to carry out. Traditionally, security was about exclusion and treated on a “need-to-know basis”. Only those responsible for security knew about it, the remainder accepted what they were told and worked around that. With DevSecOps, as in DevOps, inclusion between the different concepts and groups is encouraged and promoted with the objective of making the three departments work as a team [13]. The following figure 4 represents the workflow and elements involved when putting a DevSecOps culture in practice within an organization.

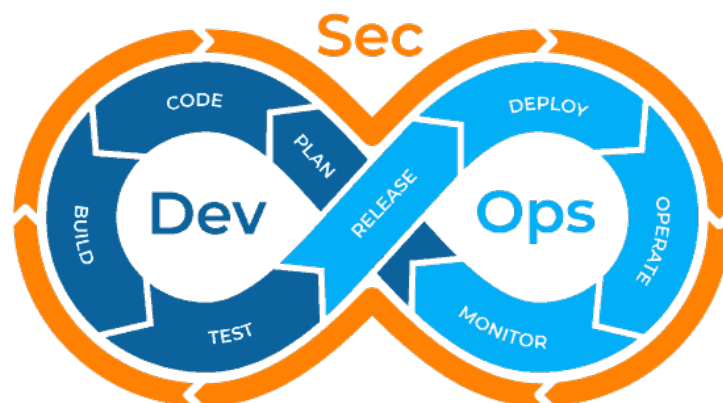


Figure 4. Graphical Representation of the DevSecOps Culture [14]

For this internship, DevSecOps has relevance to the extent that framework that will be developed is the means with which the organization looks to implement a DevSecOps culture into a development team. The adoption of automated tasks that are traditionally time consuming and troublesome into a framework which guarantees a reduction of time-to-market as well as

quality assurance regarding functionalities, performance and security does this culture the most justice when applied in an environment where software needs to be made available in a secure and efficient way.

2.4 Continuous Integration/Continuous Delivery

Continuous Integration/Continuous Delivery is a method used to produce software in an organization, where developers are frequently delivering application changes to customers with the inclusion of automation tools in the development process. CI/CD is a solution to problems created for development and operations teams when looking to integrate new content rapidly, without loss of quality and security [15].

CI/CD looks to introduce ongoing automation and monitoring into an application's lifecycle, beginning at the integration phase, through testing and finally to delivery and deployment. Once together, these two concepts allow for the creation of a CI/CD pipeline which looks to join both development and operations teams. This junction of the two teams creates an adoption of a DevSecOps culture into a development workspace.

Continuous Integration

Continuous Integration is defined as the act of applying multiple changes to an application in a short period of time and having those same changes committed, tested, and verified autonomously in order to be applied in the production environment. The differing element in this concept compared to the customary development process is that the entire process is automated. Traditionally, changes made to an application would be applied every six months, or even once a year. In a market with little competition, this could be a fruitful strategy for introducing new features to application users as the probability of a competitor releasing new software would be somewhat reduced, given that they too would only be delivering 6 months later. Today, given the level of competition within the informatics industry, organizations require a must shorter time to market for their products and software solutions.

Continuous Delivery

Continuous Delivery is the ability to implement all types of changes to software – including new features, configuration changes, bug fixes – into production in a quick, safe and sustainable way. To be able to do this, the software code must always be in a deployable state and ready to be applied in the production environment.

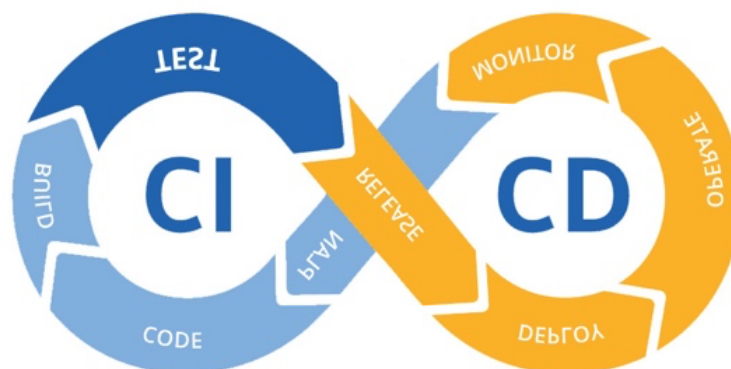


Figure 5. Graphical Representation of CI/CD [16]

Whilst analysing Figure 5, we can see how CI/CD has a very strong resemblance of the workflow involved with DevSecOps. The implementation of this culture is best achieved when adopting the automation of tasks and implementing a framework which offers the possibility for Continuous Integration and Continuous Delivery.

2.5 Monitoring Microservice

Monitoring refers to the process of collecting information about an application in order to help developers audit an application according to the level of availability, bugs that may occur, resource usage and performance changes [17]. This concept is essential when dealing with microservice applications where each service consumes its required resources and produces its own logs.

When looking to manage an application, monitoring is an essential element when an application is put into production. Without this, understanding how the application behaves [18], understanding the state of the various services, readily diagnosing problems, performance and costs management and future planning becomes a much more difficult task as there is no information with which to work. Furthermore, given that the application that is being monitored follows a microservices architecture, there can be situations where dozens or even hundreds of services are put in place to make the application conduct business [19]. In these situations, debugging and error detection become troublesome, and being able to retrieve this information would require having to access each service and perform tests on what may happen.

The final framework produced in the internship will include a solution that offers a means of accessing the information regarding problems associated with application metrics and logging. The solution will offer a centralised location for data analysis and a form of reacting and, in some situation, foreseeing possible errors that may occur.

Chapter 3

State of the Art

In this chapter there will be a general overview of the problems associated with the internship. Each section will begin with an explanation of the problem it looks to resolve, followed by the analysis of the tools that exist in the market at the moment and that look to provide a solution for the three general problems that the framework looks to address: Infrastructure Provisioning, Continuous Integration/Continuous Delivery (CI/CD) and Microservice Application Monitoring. Each section will have a list of the best practices associated with each of these problems, an explanation of the experimentation process carried out with each tool, and finally a comparative analysis of the results obtained from implementing and performing experiences with the tools.

The various tools will be evaluated according to the specific needs of the scope in which they perform. For the evaluation process, the symbols present in the following Table 1 will be used to compare the various tools and offer a better understanding of the possibilities which they offer.

Table 1. Symbology

Symbol	Meaning
✓	Feature is provided.
✗	Feature is not provided.
▲	Feature is provided but is incomplete.

3.1 Infrastructure Provisioning

This section will serve as an explanation of what the intended framework must be able to produce from an infrastructural point of view. “Provisioning denotes the prerequisite steps in managing access to data and resources and facilitating systems and users’ availability. It also refers to the setup of IT infrastructure [20].” In other words, infrastructure provisioning refers to the act of creating computational resources on which to deploy software applications.

When looking to create an infrastructure for an application in a cloud-native environment, there are two methods that can perform this task. One is to manually access the cloud provider’s platform and create the various components, one at a time. The alternative solution is to make use of a trade-off of the adoption of Cloud Computing and the Infrastructure as a Service model. This alternative is creating Infrastructure as Code (IaC). By using IaC, a solution is constructed through the use of configuration templates which are treated in the same manner as software source code. The creating of infrastructure using this model helps mitigate the errors in creation and configuration of application infrastructure, whilst simultaneously saving the developer the time required to manually go to each component and create it. The time used to configure and create a script file, define how the infrastructure must exist, and the deployment of the infrastructure is reduced considerably. Another advantage of provisioning infrastructure

environments with code is the replication of resources can be made by simply taking the contents of the configuration file and creating another file with the same values, reducing the possibility of human error.

The following figure 6 represents a conceptual diagram of IaC, where a developer creates a configuration template file and communicates it to the cloud provider. The cloud provider receives the configuration file and creates the resources according to the values supplied by the developer.

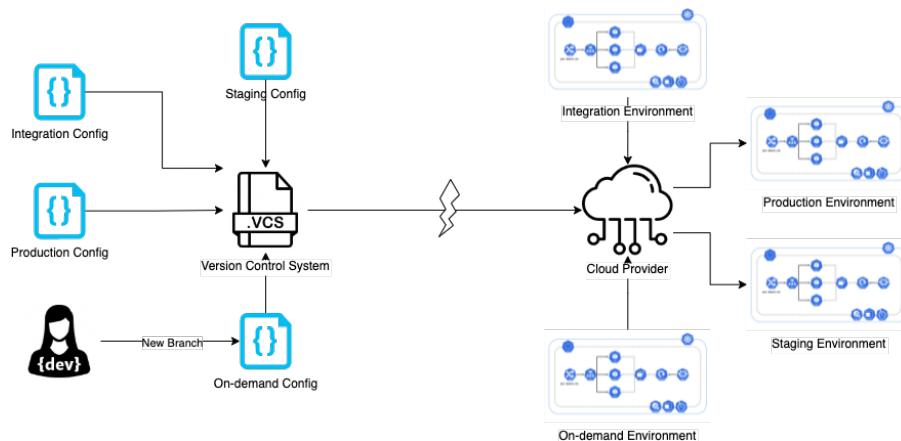


Figure 6. Infrastructure Provisioning Conceptual Diagram

The market offers the possibility of implement a solution using IaC using a large variety of tools, yet as a means of implementing a solution which applies the best practices and also maintains within the scope of the internship, the analysis made in this section will be between AWS's CloudFormation, HashiCorp's Terraform and RedHat's Ansible.

Best Practices

When implementing an Infrastructure as Code framework, the generally accepted best practices [21] are as follows:

- Treat infrastructure as developers treat code (defined format and syntax);
- Declarative configuration;
- All code stored in a source control system;
- Model all production in code;
- Resource monitoring.

When developing this component of the framework, this will be list of practices used to implement the various resources within the infrastructure.

HashiCorp's Terraform

Terraform is an open-source Infrastructure as Code software tool, developed and maintained by HashiCorp [22]. With this tool, users define and provide data centre infrastructure to developers by using a declarative configuration language known as HashiCorp Configuration Language (HCL) or via JSON objects. This tool acts as an abstraction layer over the existent cloud infrastructure provisioning tools' languages, acting as a translator between these said languages and the developer, facilitating the process of provisioning infrastructure.

Terraform offers the following features:

- Open source;
- Multi-cloud integration;
- Supports both HCL and JSON.

AWS CloudFormation Templates

AWS CloudFormation is an Infrastructure as Code service that allows for modelling, provisioning, and managing both AWS and third-party infrastructure resources [23]. What CloudFormation does is allow developers to create an infrastructure through the use of code files by using YAML or JSON languages. It also offers sample templates as a starting point for infrastructure creation.

CloudFormation's more noteworthy features are:

- Not open source, but is free to use;
- Works best with AWS Services;
- YAML and JSON;
- Extensive documentation.

AWS's CloudFormation Templates are a very powerful tool when looking to provision infrastructure for an application, whilst also not being limited to this functionality. The use of CloudFormation Templates to provision infrastructure within AWS is completely free of charge, whilst it also offers a free-tier eligibility for the first 1000 operations that are carried out with third-party resource providers. In this internship, given that all the infrastructure will be created by using AWS resources, it can be considered that there are no costs associated with the use of CloudFormation Templates, yet it is not an open-source tool.

RedHat Ansible

Ansible is the cloud infrastructure configuration tool offered by RedHat, Inc. Similar to Terraform, what Ansible offers is an abstraction layer to the infrastructure provider's traditional declarative language, looking to simplify the process of creating an application infrastructure through the use of "Playbooks". "Playbooks" are a script file which declares which components must construct the infrastructure, allowing the declaration of these elements to be done in a key-value sequence.

Ansible's relevant features toward this internship are the following:

- Open source;
- Multi-cloud integration;
- YAML language support.

Experimentation

When experimenting, two experiences were conducted to evaluate each tool. Firstly, each tool was used to create simple Elastic Cloud Computing (EC2) which would run a test python application docker container. This application would open a service listening on port 8000 and would return an HTML file with the hostname and time at which the request was fulfilled. The second experimentation was the deployment of a simple counter application into an AWS Simple Storage Service (S3) bucket. The amount of support offered when creating the necessary

configuration to deploy the application straight to the bucket would be the determining factor of understanding which tool best performed the task.




When testing this process with Terraform, I felt that the documentation was not clear about the various configurations that could be applied to the EC2 instance, and therefore limited the interaction with the cloud provider. The syntax, although similar to JSON, was not easy to comprehend at first and required a number of trial-and-error events to deploy the application. Regarding the second test case, having experimented previously with the language, understanding how to provision the components was easier, yet once again the lack of configuration possibilities in the documentation was noteworthy.

Ansible and CloudFormation Templates offered a very similar experience when creating the infrastructure. Both offer the possibility of using YAML as the configuration language, both follow a similar means of declaring the components that will constitute the infrastructure itself, with Ansible lacking slightly in terms of documentation, and both tools offer a template verifier which guarantees that syntax errors and misconfigurations are detected before deploying the file itself.

Solution Analysis

The contents present in Table 2 present a more structured analysis of the tools considered for the phase related to infrastructure provisioning.

Table 2. Infrastructure Provisioning Tool Analysis

	Open Source	AWS Integration	Multi-cloud	EKS Support	Rolling Updates	Documentation
 AWS CloudFormation	✗	✓	✗	✓	✓	✓
 Terraform	✓	✓	✓	✓	✗	▲
 Ansible	✓	✓	✓	✓	✓	▲

The experimentation process involved evaluating and comparing the most common open-source tools with the AWS equivalent. After carrying out the experimentation process, the tool chosen for Infrastructure Provisioning was AWS CloudFormation. The extensive documentation and support for developers is the main contributing factor, as it simultaneously offers several functionalities such as state management, rolling updates and rollbacks Out-of-the-Box, along with an extensive documentation and several examples of code snippets which can be used as a starting point for more complex computational components.

3.2 Continuous Integration

This section will serve to better understand what the concept of Continuous Integration is and how it is relevant to the internship's project, along with the problems that the internship looks to provide a solution for. The following figure 8 shows a graphical representation of the workflow involved when creating the Continuous Integration phase of a CI/CD pipeline, which extends from the software developer to a docker image repository, with intermediate steps such as automated image building, functional tests, security verification and validation, and static code analysis. The following Figure 7 presents the workflow involved in a generic Continuous Integration phase of a pipeline, where the developer pushes the changes made to the source code to a Version Control System, and a code build task is carried out. This code build task involves performing static code analysis, security tests and dependency validation, and functional tests. If all is successful, a docker image is built and sent to an image repository.

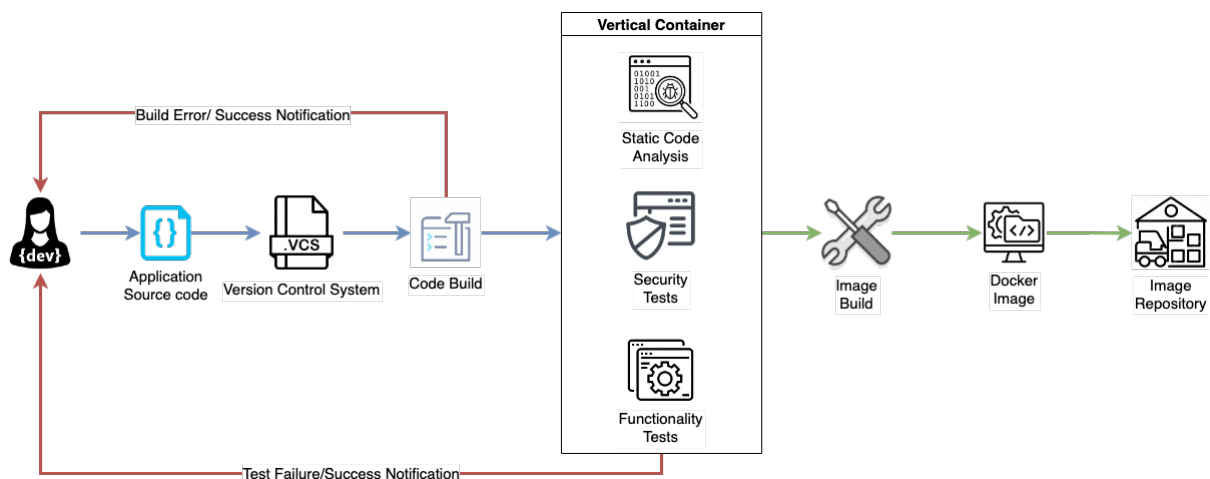


Figure 7. Continuous Integration Conceptual Diagram

In the scope of the internship, Continuous Integration looks to fulfil the need for a faster time-to-market approach when producing software. CI allows development teams to be continuously producing new code and features for an application on a daily basis, without the need for delays with time consuming tasks such as application builds and unit tests. Here, instead of manual, step-by-step testing, all the functionalities, security protocols and code convention measures are guaranteed through an automated sequence of tasks that, that traditionally could stretch out over a 6 month or one year period. In the case of CI, the application's features and source code are updated daily, and a new version of each service is created so that a deployment can be made every day, if it is required to do so.

Best Practices

When implementing a Continuous Integration component within the framework, the generally accepted best practices [24] are as follows:

- Source control system as single point of truth;
- Perform automated tests;
- One build, many steps to achieve the build;
- Perform tests on functionalities which give the most feedback;

- Maintain environment components;
- Monitor build efficiency.

When developing this component of the framework, this will be list of practices used to implement the various resources within the phase responsible for Continuous Integration.

Jenkins

Jenkins is an open-source automation server which allows for automating tasks that compose the building, testing, and deployment phases of software production [25]. Jenkins allows developers to perform static code analysis, unit tests, security dependency verification and validation autonomously, in order to guarantee that a new code commit does not “break the build”. If the new code does not pass the defined tests, the developer who produced the code is notified that there was an error, and the build does not advance. If the tests pass the build phase, a docker image is built and proceeds to the delivery phase.

The main specifications of Jenkins are:

- Open source;
- Multi-cloud integration;
- Supports multiple repository sources for code storage;
- Compatible with a wide variety of tools, including CodePipeline, but requires plugins to achieve integration;

AWS CodePipeline

Amazon’s solution for Continuous Integration (and, subsequently, Continuous Delivery) is AWS’s CodePipeline tool. Considering that CodePipeline offers the possibility for defining tasks both for Continuous Integration and Continuous Delivery, for this sub-chapter, merely the solution for Continuous Integration will be considered [26].

AWS CodePipeline offers the possibility of integrating a source code repository from a number of providers including GitHub and AWS CodeCommit. Once integrated with a source provider, the tool makes use of another tool offered by AWS which is AWS CodeBuild for carrying out tasks such as code analysis, unit testing and dependency verification. AWS CodeBuild is the automation server used for following and executing the tasks present in a *buildspec* file, included in the source repository. If this file’s commands are successfully executed and the application complies with the various defined tasks, an artifact is created and then forwarded to a component where the produced artifact is stored. In the case of this internship, it will be a Docker Image Container, pushed to an AWS Elastic Container Registry.

AWS CodePipeline’s more noticeable functionalities are:

- Full integration with other AWS tools;
- Allows the use of Jenkins as a build provider;
- Compatible with various Version Control Systems;
- Offers pre-built plug-ins with full support;

Being an AWS-native tool, CodePipeline is limited to the Amazon environment. When looking to maintain an agnostic solution for code build automation, this is not a viable solution. In other words, if an organization looks to migrate to a different cloud-provider, the configuration

process must be repeated. AWS CodePipeline does offer integration with multiple VCS repositories, although when looking to go beyond the use of AWS's CodeCommit or GitHub, the configuration process is somewhat troublesome, requiring a much more complex configuration.

To conclude, AWS CodePipeline is not an open-source tool and therefore its use is billed. The first 30 days of experimentation are offered free of charge as a form of encouraging users to adopt the tool and understand its various functionalities. Once the free-trial period concludes, each active pipeline has a cost of \$1 per month. The cost is not limited to this, as each integration of other tools such as AWS CodeBuild, for example, also have their own billing structure which adds onto this.

GitLab CI/CD

GitLab CI/CD is a tool for software development using the continuous methodologies by automatically building, testing, deploying, and monitoring applications [27]. More specifically for this section, GitLab CI is a software automation tool which is provided alongside GitLab's Version Control System. This tool makes use of the Jenkins engine to automate the build process and conduct autonomous tasks which accelerate the process behind software development.

The main specifications of GitLab CI are:

- Open source;
- Multi-cloud integration;
- Supports multiple repository sources for code storage;
- Compatible with a wide variety of tools, including CodePipeline, but requires plugins to achieve integration.

The GitLab CI tool was considered as a possible solution for the CI/CD pipeline due to the fact that part of WIT Software's software code repositories are hosted on a local GitLab enterprise server.

CircleCI

CircleCI is a software automation tool, used by large enterprises such as Spotify, Coinbase and BuzzFeed and is therefore a reference in the market of autonomous software development.

The main specifications of CircleCI are:

- Open source;
- Multiple language and platform support;
- Resource control and management;
- Multi-cloud support
- Compatibility with cloud provider native tools is easily achieved.

CircleCI is an open-source solution which offers a wide variety of possibilities regarding the build and static testing of software. One of the few drawbacks of this tool is the lack of out-the-box features provisioned by the tool itself. Where tools previously mentioned offer different runtime environments along with pre-installed packages such as python language support, docker runtime engine, java runtime engine, etc, CircleCI requires these be provisioned and

installed when the build is being performed. When looking to develop applications which require a large amount of dependencies, these tasks will increase build time and slow down development.

Experimentation

When looking to understand how each of these solutions functioned, I evaluated the complexity required to create a pipeline which would use an existing repository, conduct a static code analysis, unit testing and finally security verification of the module's dependencies. Once these tasks have completed, the code will be built into a docker image and deployed to an AWS Elastic Container Registry. All of these tools make use of a configuration file which defines what is expected from the pipeline and which tasks will be carried out. The elements which will be monitored are:

- Build time (Average time of 50 builds);
- Build state notifications;
- Documentation.

Jenkins performed an average build time of 3 minutes and 42 seconds from the source repository to pushing the new docker image to the elastic container registry. Email notification integration is natively supported by Jenkins where the configuration is somewhat straightforward. The user only needs to provide an email address and an SMTP authentication method. The message received is the build number, the status of the build and a link to the project execution.





AWS CodePipeline had an average build time of 2 minutes and 37 seconds from source to registry. There is no email notification system supported natively as to perform this, the AWS CodeBuild tool must be associated with an AWS Simple Notification Service which receives the message from AWS CodeBuild and then forwards it to an email endpoint. This to say, it is possible to create an email service which delivers the build's state, but it is not native to AWS CodeBuild itself.

Circle CI averaged its build times around 3 minutes and 10 seconds from source to registry. There is a native notification service, where configuration is limited to associating a build project to a specific endpoint. This solution is very attractive, especially given the more complex functionalities that it offers beyond the basic "build and deploy", but the limited documentation leaves me somewhat reluctant to adopt the tool itself.

Solution Analysis

The content of Table 3 presents the different opportunities that each tool offers to create a solution for the problems associated with Continuous Integration and the application of the various best practices associated with this concept.

Table 3. Continuous Integration Tool Analysis

	Open Source	AWS Integration	Multi-Cloud	Notification Service	Documentation	Plug-in Support
 Jenkins	✓	✓	✓	✓	▲	✓
 AWS CodePipeline	✗	✓	✗	✗	✓	✓
 GitLab CI	✓	✓	✓	✓	✓	▲
 circleci CircleCI	✓	✓	✓	✓	▲	✓

Considering that the given documentation and support is an aspect that weighs strongly in favour of the choice of the tool, Jenkins, which operates on a strong plugin basis, requires that those same plugins offer appropriate documentation to assist with developing the framework. These same plugins themselves are not verified and their documentation is somewhat limited. CircleCI does offer a strong campaign when looking to choose the tool responsible for CI in the framework, but its documentation also does lack to an extent, with limited examples. On the contrary to this, AWS's CodePipeline manages to offer these exact features, whilst having to compromise on the solution not being open source and needing manual configuration for the notification service. This compromise means having a more accessible and informed tool available, and therefore, the technology used to conduct the tasks of Continuous Integration will be CodePipeline.

3.3 Continuous Delivery

In this section, the concept of Continuous Delivery will be further developed. There will be an initial definition of the concept itself, followed by an explanation of the needs that the framework will require from the standing point of Continuous Delivery. Furthermore, an explanation and comparative analysis of the solutions that exist for these issues will be made to establish which is the best tool for providing a solution for the problems that compose the internship.

Continuous Delivery has an influence in this internship due to the fact that the changes implemented by the developers need to be deployed into their production environment continuously and progressively. This task could be done manually and with all the alterations to the code and new functionalities being delivered to the entire user population at once, requiring a full deployment plan, during low-traffic hours and with a large level of human

resources being available to carry out the delivery. This is not a viable solution when wanting to guarantee that a new feature or design change is accepted by the application’s users. Given that the idea behind DevSecOps is to perform multiple commits and new versions of software, multiple times a day, this method would be a waste of resources. Moreover, giving users access to an entire new version of an application translates into a more complex strategy when a rollback needs to be performed. The CD module of the pipeline will perform phased rollouts, following these methods of deployments:

- Blue/Green Deployments;
- Canary Deployments;
- A/B Testing

Phased rollouts are a method used to deliver new system implementations and alterations in an incremental fashion. This allows an organization to react accordingly if an issue arises whilst also allowing users to adjust to changes gradually. These methods of deployment are further explained in the document titled Appendix A – State of the Art. For a question of readability, this document will only present the Blue/Green Deployments and Canary Deployments methods.

Blue/Green Deployments are an implementation strategy of a phased rollout. In this case, it uses at least two environments which are as identical as possible, each running a version of the application or service [28]. The application’s traffic, which was originally targeted towards the previous version (blue), is then routed to the second version (green) and acceptance and quality assurance tests are run to guarantee that the new version is ready to receive users. Once the deployment is successful, the green version can then replace the blue. This process is presented in the form of a diagram in Figure 8.



Figure 8. Blue-Green Deployment [28]

A canary deployment is a deployment strategy that releases an application or service incrementally to a subset of users [28]. All infrastructure in a target environment is updated in small phases (ex: 2%, 25%, 75%, 100%). A canary release is the lowest risk-prone, compared to all other deployment strategies, because of this control. The following Figure 9 shows a graphical representation of how a canary release is applied, with the routing of traffic between two application versions.

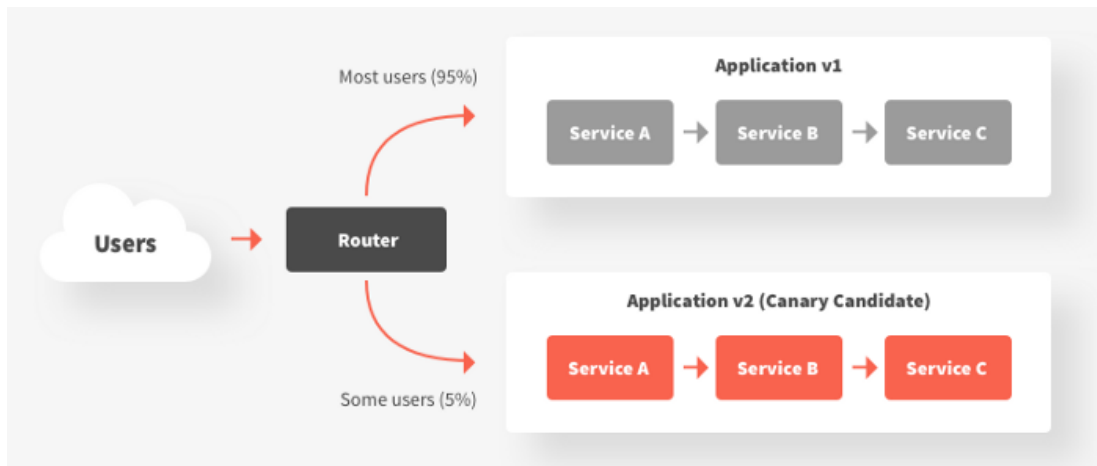


Figure 9. Canary Release Workflow [29]

The pipeline must offer the possibility for these multiple deployment strategies, each being chosen and configurable according to the context into which they are applied. The solutions taken into study which offer a possibility for CD are GoCD, FluxCD, Argo CD and AWS CodeDeploy which is integrated into the AWS CodePipeline tool.

Best Practices

When implementing the Continuous Delivery component in the framework, the generally accepted best practices [24] are as follows:

- Deploy to every environment the same way;
- Inject purpose faults to test pipeline recovery;
- Automate the deployment and rollback process;
- Version control as single point of truth;
- Monitor pipeline statistics.

When developing this component of the framework, this will be list of practices used to implement the various resources within the phase responsible for Continuous Delivery.

GoCD

GoCD is an open-source software delivery tool which is used to help teams automate the process of delivering software to users [30]. GoCD operates based on plugins developed by the community to perform the various tasks necessary to deploy an application into production. GoCD follows a client-server model where the server is running locally on the developer's machine and is responsible for polling the repository for changes. When a change is detected, the corresponding pipeline is triggered.

GoCD offers the following functionalities as valuable for the internship:

- Open source;
- Pipeline Configuration is made using YAML;
- User Management;
- Allows Pipeline Bridging;

- Multi-cloud support;
- Native Kubernetes and Docker support;

FluxCD

FluxCD is a toolset of Continuous Delivery solutions, specializing in Kubernetes and containerized applications [31]. This tool allows for deploying applications into Kubernetes clusters with canary releases, A/B Tests, and blue-green deployments. Exactly the three requisites of the intended product of this internship. Given a YAML manifest file kept in a VCS repository, FluxCD periodically verifies if there are any changes made to the configuration file of the deployment. If there is a change in the configuration, the cluster is updated and synchronized according to the changes made. The current configuration file is kept in cache and used as a term of comparison to verify if any changes were made.

FluxCD's main specifications are:

- Configuration via YAML;
- Command-line tool;
- Out-of-the-Box integration with GitHub;
- Direct integration with EKS;
- Performs bootstrapping within a Kubernetes cluster;

Argo CD

Argo CD is a declarative continuous delivery tool created for the deployment of Kubernetes clusters and microservice applications [32]. This tool's core component is the Application Controller which is responsible for monitoring the current state of an application and compares it to the desired target state that is kept in a Git repository. The existence of this controller allows for automated deployments where the desired application state is pushed into the cluster automatically, a task previously triggered by a Git commit or, in the case of this internship, by a CI pipeline which notifies that a new version of the application is available for deployment.

Argo CD also offers a graphical user interface which brings application state visibility to the developer, whilst also providing a command line tool which can be adopted by more experienced users. Given that an application cluster can have multiple pods running with a number of distributed communications being performed amongst each other, the possibility of having a visual representation of the state of the application brings an increased amount of value to the use of Argo CD as a solution for Continuous Delivery.

Argo CD brings value to the internship with:

- Configuration via YAML;
- Command-line tool;
- Out-of-the-Box integration with GitHub;
- Direct integration with EKS;
- Performs bootstrapping within a Kubernetes cluster;
- Graphic User Interface;

AWS CodeDeploy

AWS CodeDeploy is a fully managed deployment service that automates software deployments to a variety of computational services such as Amazon EC2, AWS Fargate, AWS Lambda, and on-premises servers. AWS CodeDeploy makes it easier to rapidly release new features, helps avoid downtime during application deployment, and handles the complexity of updating applications. AWS CodeDeploy can be used to automate software deployments, eliminating the need for error-prone manual operations.

AWS CodeDeploy brings value to the internship with:

- Configuration via YAML;
- Command-line tool;
- Out-of-the-Box integration with AWS CodePipeline;
- Graphic User Interface with AWS Console;

Experimentation

Given the fact that GoCD does not offer the possibility of integrating phased rollouts, it was not considered in the experimentation phase of the tools involved with Continuous Delivery. Therefore, the comparison made was between FluxCD, Argo CD and AWS CodeDeploy.

The learning curve associated with the three tools that were analysed varied to an extent. Argo CD and Flux CD were the more balanced and understanding their functionality and visualizing the information regarding each deployment was straight forward and, initially did not require an extensive navigation. AWS CodeDeploy did offer several difficulties when performing a canary deployment, requiring that a lambda function be used to process changes and replace the versions of the images running in the cluster.

Essentially, the three tools offer very similar solutions, with the differentiating factor being the GUI that is offered with Argo CD and AWS CodeDeploy. With this in mind, the decision between the tools depended on the experience and feel given when using each solution, with the technical factors being somewhat identical. Also, from a cost point of view, AWS CodeDeploy does have the setback that it is billed according to the amount of deploys performed. This factor reduced the choice to Argo CD.

Solution Analysis

The content shown in Table 4 performs a graphical comparison between the various solutions possible with each of the competitors that were previously analysed.

Table 4. Continuous Delivery Tool Analysis

	Open Source	Blue/Green Deployments	Canary Releases	Notification Service	GUI	Documentation
 GoCD	✓	✗	✗	✓	✓	✗
 FluxCD	✓	✓	✓	✓	✗	▲
 Argo CD	✓	✓	✓	✓	✓	✓
 AWS CodeDeploy	✗	✓	✓	✗	✓	✓

GoCD is a powerful tool for performing automated tests, executing tasks on an application, and deploying software, yet it does not allow the implementation of phased rollouts into its delivery model. Given that this is a compulsory requirement for the final product, this cannot be considered a possible solution for the problem at hand.

AWS CodeDeploy was considered for discussion when analysing the different solutions but, given a larger tendency of the informatics community to adopting FluxCD and Argo CD for Continuous Delivery solutions, the possibility of adding another billed service to the pipeline made the solution much less attractive. Given the fact that other open-source solutions offer very similar functionalities as AWS CodeDeploy, tool was not excluded from the possible choices for the framework.

Being FluxCD and Argo CD the two remaining possibilities, whilst sharing many characteristics and both following a very similar form of operation, Argo CD was found to be the more interesting choice. This is due it also offering a Graphical User Interface, which offers a larger visibility to the state of each container and if the system is in the current desired state or if alterations are necessary. Although a basic reason to differentiate between the two tools, the most noteworthy factor between the two is, in fact, the existence of the GUI.

3.4 Monitoring Microservices

This section explains what Application Monitoring is and what challenges the exist when deploying a microservice application into a cluster. There will also be an analysis of what tools offer a solution for the questions raised and which best suits the questions approached by the internship’s proposal.

Figure 10 presents a graphical representation of the concept of monitoring microservices. The Kubernetes server and the applications running on it are queried for logs and metrics, which DevOps team members can analyse to establish what components can be improved, and if there are any errors occurring that may have gotten through the CI/CD phase, and therefore need to be corrected.

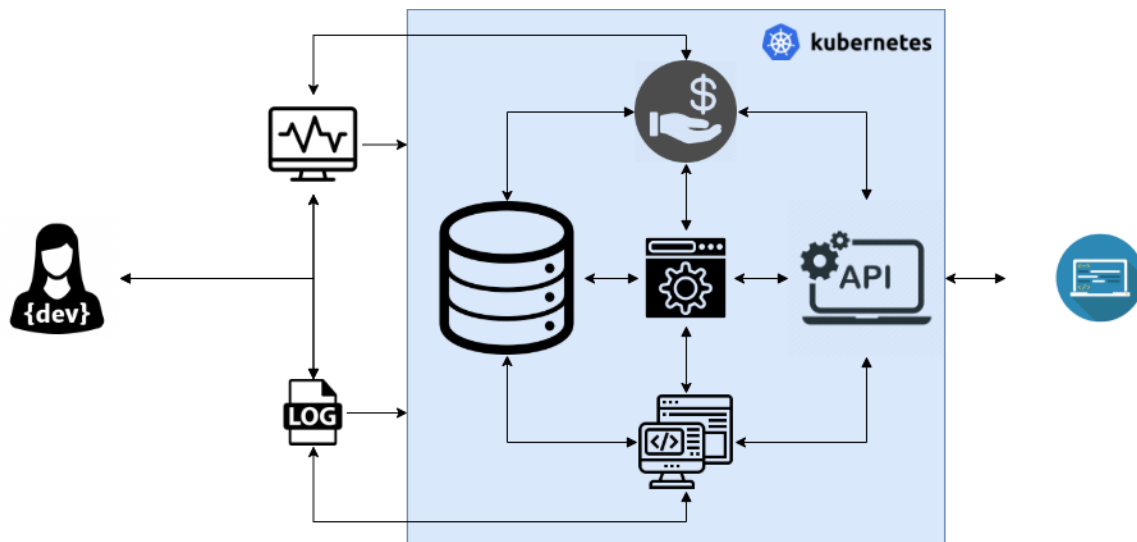


Figure 10. Microservice Monitoring Conceptual Diagram

In order to gain application observability, a monitoring strategy must be implemented that extracts metrics and logs produced by each of the services. Monitoring solutions provide a visual means to observe how the various events within each service are connected and how they behave throughout the production environment. To obtain this information, the best practices for application require that the following components be measured in the cluster, present in Table 5:

Table 5. Microservice Monitoring Best Practices

Metrics	Description
Cluster nodes	Number of nodes that are available. Gives an overview of the resources needed to run the cluster.
Cluster pods	Number of pods running in the cluster. Gives feedback on if there are sufficient nodes for the workload.
Resource utilization	Measures the cluster's memory, CPU, bandwidth and disk consumption.
Container Metrics	Measures the network, CPU and memory usage in a container.
Application Metrics	Measures the business logic metrics such as the number of users accessing the application, user experience, etc.
Kubernetes Scaling and Availability Metrics	Allows to manage how the orchestrator handles each pod, specifically. This includes pod health checks, network data transfer, on-progress deployment, etc.

Best Practices

When implementing the Monitoring component in the framework, the generally accepted best practices [24] are as follows:

- Monitor containers and their various processes;
- Retrieve metrics and alert on service performance;
- Prepare for scalability;
- Monitor application API;
- Perform mapping between application information and organization structure.

When developing this component of the framework, this will be list of practices used to implement the various resources within the phase responsible for Application Monitoring.

AWS CloudWatch

Amazon's solution for application monitoring and observability is AWS CloudWatch. It provides information and insights into an applications functionality and operation, in order to maintain a backlog of applications, whilst allowing to perform actions on the data extracted such as system performance changes and resource optimization. CloudWatch is a unified and centralized information source for obtaining and registering application behaviour [33].

CloudWatch's specifications are as follows:

- Single Observability Platform;
- Performance and Resource Optimization;
- Operational Visibility Direct integration with EKS;
- Derive insights from logs;
- Integration with AWS Tools;

ELK Stack

ELK Stack is a search engine which provides the possibility of extracting, indexing, and presenting application logs. It is a stack composed of three tools: Elasticsearch, Logstash and Kibana [34]. The ELK stack has been updated to support Kubernetes log analysis, maintaining Kibana and Elasticsearch as the visualization tool and centralized search engine for log persistence, respectively. For Kubernetes support, ELK evolved into Elasticstack, where instead of using Logstash as the ideal tool for monolithic log analysis, Elasticsearch introduces the concept of "Beats": lightweight, single-purpose data shippers, which extract specific details about microservices and export them through to Elasticsearch.

The main specifications offered by the ELK stack are:

- Open source;
- Near real-time data extraction;
- Extensive plugin library;
- Plugin verification and evaluation;
- Extensive documentation;

Prometheus and Grafana

Prometheus is an open source and free monitoring solution software responsible for analysing and processing in information regarding application metrics in real time [35]. This tool operates according to a pull model in which the different components within an application are queried for their health, the amount of resources that are being consumed and miscellaneous statistics associated with requests made and tasks that are carried out.

At the same rate that the information is being scraped from the different applications, there must be a means of analysing and viewing the information in a natural and understandable way. The solution for this is to couple Prometheus with Grafana. Grafana is an open-source server application which provides charts, graphs and alerts for the different metrics and occurrences that take place in an application [36].

Prometheus' main value comes from the following:

- Open source;
- Almost real-time data processing;
- Customizable;
- Pull-model;
- Targets are dynamically discovered.

Experimentation

The experimentation process for the tools involved with monitoring the application has the following sequence:

- Deploy a Kubernetes cluster on AWS;
- Deploy a sample application to the cluster;
- Deploy the monitoring tool to the cluster;
- Extract information regarding metrics (Prometheus + Grafana) and logs (ELK Stack).

When using AWS CloudWatch, the possibility of having a monitoring tool already included in the cloud provider's solution made for a more complete solution, at first sight. When performing queries to the application and awaiting CloudWatch to update its dashboards to reflect the new information, a delay of around 2 minutes is present, both when analysing application metrics and the logs produced. The fact that it is paid service also contributes to not adopting the tool in the final solution, given that when one is paying for a tool, it is expected that that same tool performs at the highest and most efficient level.

Regarding Prometheus and Grafana, the deployment of these tools was simple. Working as a daemon set deployment to the application. Prometheus runs parallel to the application and extracts information from the pod, the information is extracted in real time and presented to Grafana in the same manner. Given that it is an open-source solution with a much larger efficiency, this solution is a very strong candidate.

The Elasticstack requires a more complex configuration, as there are three tools which must be prepared and orchestrated to retrieve the logging information from the application itself. This process is troublesome but given the immediacy of the information that is extracted from the application, together with the fact that it is an open-source solution, it would seem to be the more correct option regarding application logging.

Solution Analysis

The content of Tables 6 and 7 serve as a graphical comparison between the different tools and what solutions they offer for the problems associated with application monitoring.

Table 6. Microservice Monitoring Competitor Analysis 1







	Open Source	Log Analysis	Health Monitoring	Usage Monitoring
 AWS CloudWatch	✗	✓	✓	✓
 Elasticstack	✓	✓	✗	✗
 Prometheus + Grafana	✓	✗	✓	✓

Table 7. Microservice Monitoring Competitor Analysis 2

	Memory Usage	Network Usage	Number of Pods/Cluster	Notification Service	CPU Usage
 AWS CloudWatch	✓	✓	✓	✗	✓
 Elasticstack	✗	✗	✗	✓	✗
 Prometheus + Grafana	✓	✓	✓	✓	✓

Although CloudWatch does offer a centralized solution for creating application visibility, the fact that it is both a paid service and that it does not offer an integrated notification service reduces its value for the internship. A CloudWatch Agent must be configured and deployed to perform metric extraction, along with application log streaming to operations team members. Furthermore, the lack of dashboard customizability and custom metrics also translates into a much more limited service than that of the Elasticstack and also the junction of Prometheus with Grafana. Due to this, the opted tools for application logging will be the Elasticstack, and for application metrics observability will be Prometheus as the scarping tool together with Grafana as the visualization tool.

Chapter 4

Methodology

This project is going to be integrated into an existing framework and development team's organization tool at WIT Software. To achieve that, an objective planning process must be orchestrated to guarantee that both the internship and the final integration into the enterprise's organization are both a success. The planning is done according to an Agile methodology, based on Scrum, an interactive and incremental software development structure. In this internship, "*Scrum by the Book*" will be the chosen methodology and organization sequence.

This section will be divided into five main sections, beginning with a brief explanation of the agile principles, followed by An explanation of Scrum and its roles, together with an explanation of the methodology and how it works. A graphical representation of the work plan will also be provided, concluding with an explanation of the risks that pertain to the project, with their respective mitigation strategies.

4.1 Agile and Scrum Methodologies

An Agile methodology follows a number of principles that define its culture and serve as a form of differentiation from other methodologies. These principles are the following:

- Individuals and Interactions over processes and tools;
- **Working Software** over comprehensive documentation;
- **Customer Collaboration** over contract negotiation;
- Responding to Change over following a plan.

These principles solver various risks that can occur throughout the software development process. This section will look to give an explanation into the methodology used for the development of the product in this internship. More specifically, the Scrum methodology.

Scrum Definitions

When looking to evaluate a projects complexity, using a Stacey Complexity Matrix gives a good understanding of how close a team is to agreement on what requirements are needed in the project in relation to how certain the team is about the strategy used to implement them. There are three main profiles present when classifying a project according to its complexity, according to Santiago Obrutsky [37]:

- **Simple:** project that are easy to understand and with a simple concept. These are known in their entirety and the developers are 100% sure of what needs to be done and which technologies to use;
- **Complex:** projects with an intermediate level com complexity, requirement knowledge is adequate, as well as the technologies that are required for the project;
- **Chaos:** projects that are very difficult to conclude. Requirements are not understood at all, nor is there knowledge of which technologies to use for development.

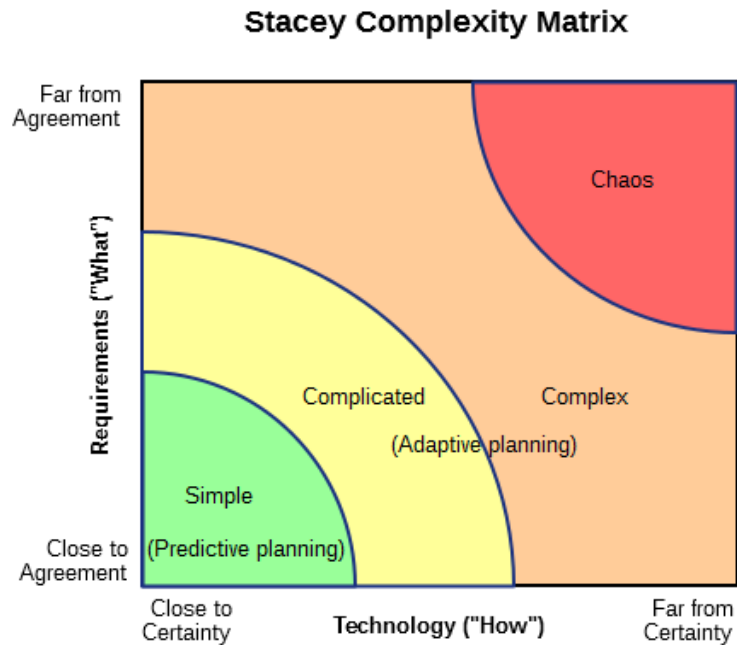


Figure 11. Project Complexity [38]

There is still another level of complexity, which is the area of complex types. This type of project occurs when the number of requirements is too large, with the number of possible solutions also being considerably extensive. In these situations, the system is referred to as a system that is not fully predictable.

Given the number of requirements that are expected for this project, along with their respective solutions, a waterfall methodology is not recommended, and therefore an Agile methodology is the ideal option. The flexibility offered is much larger, which allows the developer to adapt accordingly throughout the product's development process.

Scrum allows the necessary agility for this type of project, given the number of requirements that the framework has, together with the possibility for possible alterations to the product itself. If these suffer changes, there is a certain level of complexity which will require discipline and organization on behalf of the developer.

Scrum Roles

The Scrum methodology has a set of roles that represent those who have a level of commitment towards the project that is being developed. These roles are separated into three definitions: Project Owner, Scrum Master, Development Team.

Product Owner - the Project Owner is the person responsible for analysing business activities, customer communication and management, and product guidance. Given this, their responsibilities fall into the following list:

- Represent and manage the interests of the Stakeholders;
- Own the Product Backlog;
- Establish, nurture and communicate the vision of the product;
- Compares and monitors the project's behaviour, goal and investment;
- Makes decisions regarding when official releases are made.

This role must also ensure that the development team brings value to the project, that requirements are ranked by priority and met by the team, according to the customer's needs. The Product Backlog is a comprehensive and cooperative list of tasks that must be concluded, to which all team members and the project owner can contribute towards.

In the case of this internship, the Product Owner is Mr. Rui Cunha, who also carries out the function of being the project manager. Given his knowledge of the project's scope and the different means possible for achieving a viable solution, the role is well filled to guarantee the project's success.

Scrum Master - the Scrum Master is the element responsible for maintaining a healthy and comprehensive development team. The team's success is defined by this element, being that in the event of problems occurring among team members, they are responsible for dealing with these issues and guaranteeing that all goes according to plan. Mr. Diogo Cunha will be the orientator that is responsible for this task.

Development Team - the Development Team's purpose is to carry out the tasks defined in the Product Backlog. In this internship, the development team is composed exclusively by myself, and therefore my responsibilities are to manage and fully conduct the work which is presented to me. I must also be able to develop the various functionalities of the product, giving the due time to those which have higher priority levels.

How Scrum Works

The Scrum methodology follows an iterative development process where each work plan varies according to the duration of each feature that is to be developed. The Product Backlog is composed of a set of use cases, each having a different level of priority and complexity that allows the iteration of tasks. This also represents the work that has to be carried out throughout the internship.

Specifically in this internship, each iteration will be defined by sprints of 2 weeks, along with daily meetings with the Project Owner and Scrum Master where the following questions will have to be reported:

- What has been done since the last meeting?
- What will be done tomorrow?
- What challenges or issues do I have?

The end of each sprint will present the results and their development proceedings, allowing an adjustment to the plan if there is the need to do so. When closing a work plan, not only do the features need to be implemented, but there must also be an explanation as to what was done to achieve the goal in question. This process is explained in Figure 12.

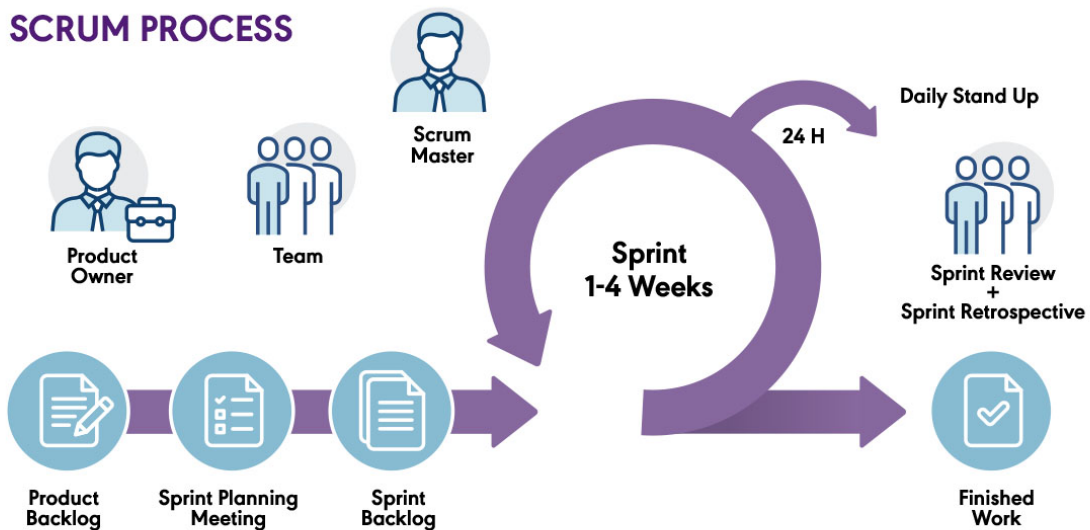


Figure 12. Scrum Overview [39]

The Backlog’s organization is of the Product Owner and the Scrum Master’s responsibility. In theory, this is usually conducted by the Product Owner exclusively, but in the case of this internship, the Backlog will be created and managed by myself, and approved by the other two members, previously mentioned.

In summary, the development process begins with an implementation cycle of two weeks, along with daily meetings with the Scrum Master in order to relay how the work is being carried out in the project and if there are any corrections that need to be made. At the end of each cycle, there is a meeting with the whole team to perform an analysis of the work that was done, what may have gone wrong, and what corrections need to be made to achieve a better result at the end of the next sprint. At the conclusion of these two weeks and once the changes are duly applied, the next iteration of development begins, and the next sprint’s process is initiated.

4.2 Project Planning

The project’s planning structure is intertwined with the Scrum approach in the sense that a Product Backlog will have to be made before development and, to achieve that, a good understanding of the framework requirements is necessary. Having this defined, the project’s goals are defined in a more global outline. From here, the Product Backlog will be altered and have more tasks added as the various requirements are prepared and analysed in more detail.

The Product Backlog’s main objective is to answer the questions “who?”, “what?” and “why?” regarding the requirements, in a simple and straightforward manner. An example of this can be the following: “As a developer, I want to add the possibility to create a new container image which will update and replace an existing image in a Kubernetes cluster, running on an AWS EKS”.

In this requirement, the questions are answered as follows:

- Who: “As a developer (...);”
- What: “(...) add the possibility to create a new container image (...);”

- Why: “(...) update and replace an existing image in a Kubernetes cluster, running on an AWS EKS”.

Each task belongs to a sub-category and has a deadline associated to it. It will follow the following presented in Table 8.

Table 8. Task Table

Category	Sub-Category	Task	Deadline
Implementation	Continuous Integration	Develop method to create infrastructure on command.	5 th February 2022
...

The definition of these components in a task, understanding what is expected with this activity is simple and easy. The deadlines are based on the priority level given by the Product Owner and according to the difficulty of the task.

The Product Backlog will show all the features that are required by the final product. However, given that the final artifact is produced throughout the entire duration of the project, each iteration must have its own definition of a work plan, which will imply that some changes be made to the Product Backlog at the end of each task. The Product Owner and Scrum Master define the work plans which must occupy the previously mentioned 2 weeks.

The tasks that are integrated into each work plan are chosen throughout the duration of the project. Given this, when a task comes to its end, a new work plan will have to be established for the following task. Due to the fact that the organization is the one responsible for creating the project itself, some changes may occur, depending on which are the more prioritized activities at the given moment.

Internship Project Organization

The project’s organization structure defines the different milestones that must be achieved to maintain the project on track and mitigate the possibility of delays occurring. The division of the different tasks allows to explain and establish a threshold to consider a sprint successful. In the case of the internship, the project will have the following organization:

Project Lifecycle:

The following Figure 13, which shows a Gantt diagram, represents the planning of the various tasks and sprints that were planned to be conducted during the second semester:

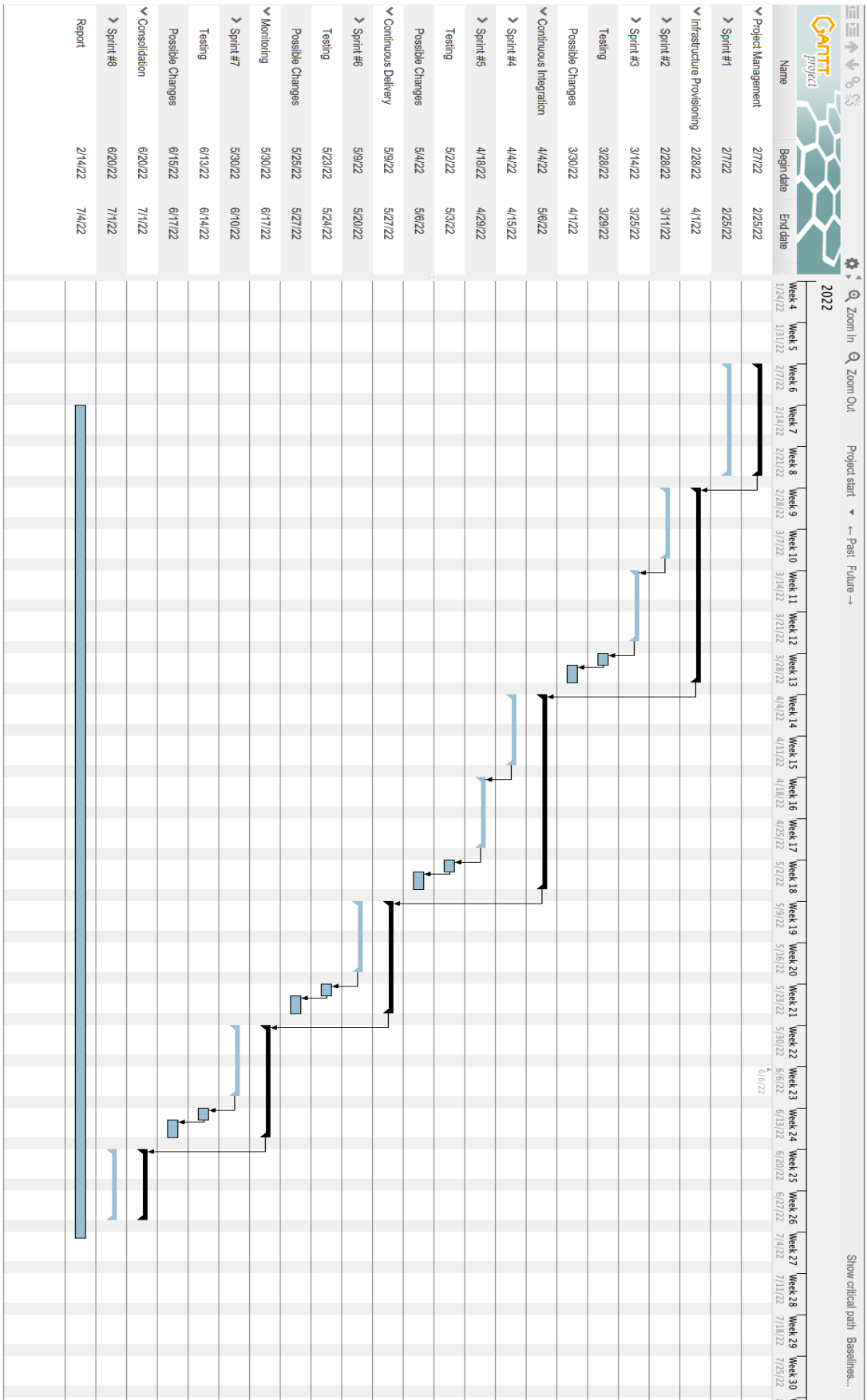


Figure 13. 2nd Semester Plan

The following Table 9 presents the more general view of the project’s lifecycle and milestones. For a more detailed list of dates and deliverables, please refer to the document titled Appendix B – Methodology.

Table 9. Project Lifecycle Milestones

Phase	Sub-phase	Epic Start	Epic End
Project Planning	-	07/02	25/02
Software Development	Infrastructure Provisioning	28/02	01/04
	Continuous Integration	04/04	05/05
	Continuous Delivery	09/05	27/05
	Application Monitoring	30/05	17/06
Consolidation Tests	-	20/06	24/06
Project Revision	-	27/06	01/07
Thesis Delivery	-	07/02	04/07

Project Management will be organized with the following elements and their respective roles:

- Eng. Rui Cunha – Product Owner;
- Eng. Diogo Cunha – Scrum Master;
- Eng. António Mendes – Guidance;
- Gabriel Pinheiro – Developer.

Base Plan and Control

The Base Plan and Control defines when each task will begin, its due date and also the amount of effort that is expected to achieve the timeline that is defined. The effort is measured in hours and is based on each working day being equal to 8 hours. The following Table 10 describes the documents that must be produced by the end of semester, their start and end dates, and the number of hours they must occupy to be produced.

Table 10. Project Management Documents

Document	Start Date	End Date	Effort (Hours)
Software Requirements Specification	10/02/2022	16/02/2022	40 Hours
Software Architecture & Design	10/02/2022	16/02/2022	16 Hours
Risk Plan	21/02/2022	22/02/2022	16 Hours
Quality Assurance Plan	23/02/2022	23/02/2022	8 Hours
Acceptance Test Plan	24/02/2022	24/02/2022	8 Hours
Milestone 1 Report	25/02/2022	25/02/2022	1 Hour

4.3 Risk Management

Whenever developing a large-scale project over a long period of time, the possibility of problems arising is high, and if the team responsible does not have a pre-existing plan for these problems, the project’s integrity and viability may be questioned. In this case, it could lead to a potential failure of the internship. Due to this, the risks associated with this internship have been analysed together with a means of mitigating those same risks. The detection of a risk implies the creation of a strategy which, when applied, will eliminate the possibility of its occurrence.

The risks follow an order of classification which is represented in the following Table 11:

Table 11. Risk Priority Table

Classification	Description
1	Very low impact, easily overcome.
2	Low impact, implies a small change to the project.
3	Medium Impact, requires a certain level of effort to overcome, but manageable.
4	High impact, if changes aren’t made, the project could potentially fail.
5	If this occurs, the project fails completely.

In order to identify each possible risk that has been identified to date, Table 12 and 13 show a list where each risk is defined by an id, type, impact, description and mitigation technique.

Table 12. Risk Identification and Impact

ID	Type	Impact	Description
1	Project Management	4	A sprint does not produce the expected result.
2	Project Management	3	Bad estimation of the duration of a sprint.
3	Technologies	3	Lack of knowledge with the tools used in the project.
4	Project Management	2	Requirement changes.
5	Technologies	4	Tool functionality/support changes.
6	Technologies	4	Organization moves to change cloud provider.
7	Technologies	5	Cloud provider dismantles.
8	Budget	2	The framework oversteps the budgeted cost threshold.

Table 13. Risk Mitigation

ID	Mitigation Technique	Conclusion
1	Adapt the amount of team in each sprint; Perform daily meetings with the Scrum Master.	In the event of a sprint being too long, the amount of time available for disaster recovery is reduced. To mitigate the risk of this happening, the best way to prevent this issue is to define smaller sprints, coupling these with daily meetings with the Scrum Master and Product Owner.
2	- Discard optimism; Resort to the Scrum Master's and the Product Owner's experience; Stick to the tools that are chosen;	The estimation for each sprint is made in a meeting where the Scrum Master, the Product Owner and myself will be present. Given my reduced experience with project estimation, the expertise of those around me will have to be the louder voice and the estimation must be done based on their opinion.

	Perform daily meetings with the Scrum Master.	Maintaining contact with both the Scrum Master and the Product Owner will be crucial for this problem to be mitigated.
3	Analyse and study the technologies before starting development; Communicate troubles with orientators.	Throughout the development of the framework, there will be a number of technologies with which I do not have the required experience. To overcome this issue, there will be time given to studying and prototyping.
4	Maintain a consistent and highly frequent delivery of artifacts to the client.	In the event of a requirement needing to be changed during the project's development, there will have to be a constant contact with the client and the Scrum Master to understand which changes need to be applied and how. This contact helps guarantee the success of the project.
5	Maintain contact and awareness of tool documentation and possible changes to the different APIs and frameworks.	Given that API updates are a reality in the community, there must be a higher level of awareness regarding the changes that may occur. This helps guarantee that the final product does not have errors or failures in its final state.
6	Maintain contact and awareness of tool documentation and possible changes to the different APIs and frameworks.	Given that API updates are a reality in the community, there must be a higher level of awareness regarding the changes that may occur. This helps guarantee that the final product does not have errors or failures in its final state.
7	Maintain contact with product owner to understand if this may become a reality; Look to use agnostic tools where possible.	An unlikely reality, but if the organization chooses to change their cloud provider from Amazon to another competitor in the field, there must be a means of migrating the framework to the other provider with the minimum amount of effort necessary.
8	There is none.	Although a highly improbable occurrence, there exists the possibility of Amazon Web Services dismantling, in which case the provisioned infrastructure will no longer be accessible, and all business must be redefined and migrated to another cloud provider.
9	Stop machines that are not being used; Clean up resources that can lead to an increase in billed values;	Costs are expected when using the framework but must be controlled as much as possible. To control these costs, the saving of computational resources is essential, especially given the fact that the project is a test environment.

Chapter 5

Framework Requirements

This chapter will focus on the requirements of the project. Here, there will be a description of the goals that the internship will have to achieve to consider the project a success. It will also act as a guideline for the functionalities that need to be achieved with the final solution. This chapter will be divided into three subsections: i) Requirement Prioritization; ii) Functional Requirements; iii) Non-functional requirements.

This section will be used to define the different types of requirements that play a role in this project. The sections of Functional Requirements and Non-functional Requirements will contain a list of each requirement associated to the group of requirements in question. Each will contain an ID, title, priority and description. There will also be an explanation regarding the order of priority of each requirement according to its relevance toward the internship's project.

To produce the content of this section, there was a meeting scheduled between the product owner, scrum master and the developer. Here, an initial list of requirements was established, along with a discussion regarding what was expected from the final product. Each requirement was evaluated according to its type and, from here, each requirement was then categorized according to the priority with which it must be developed, establishing a success threshold for the project. The methodology used for prioritizing the different requirements will be explained in further detail in the following subsection.

5.1 Requirement Prioritization

Each requirement has a different priority according to how relevant it is to the project at hand according to MoSCoW Method [40]. The MoSCoW method is a technique used to prioritize the various requirements in a project, where the name defines the four categories that differentiate the different levels of priority that can be given to the requirements. These levels are the following:

- Must Have;
- Should Have;
- Could Have;
- Won't Have Right Now.

Taking this method and these definitions into account, the following Table 14 serves as a way of distinguishing the differences according to a scale of priorities, where each level has its own definition and explanation.

Table 14. Priority Description

Type	Description
Must have	Requirement is essential for the success of the project. These are the core functionalities of the framework.
Should have	Requirement has medium priority. They are important toward the final solution, but in the event that they are not fulfilled, the project won't be considered a failure.
Could have	Low-priority requirements. These have value towards the project but will only be implemented if there is enough time.
Won't Have Right Now	Features that will be considered in future but are not within the scope of the current project.

5.2 Requirement Types

The requirements can be classified according to their type, which allows for creating a more detailed definition of their context throughout the project [41]. These definitions were established according to the value and influence that each requirement brings toward the framework and the complexity that each requirement can assume. Below, Table 15 shows how this differentiation will be made for this project.

Table 15. Requirement Types

Type	Description
Functionality	Requirements that describe the core functionalities of the system.
Usability	Requirements that require the developer's interaction with the framework.
Performance	System performance.
Supportability	Scalability, compatibility, configuration, tests, maintainability.

5.3 Functional Requirements

Functional requirements refer to what the framework must guarantee for the developers from a functional point of view. In other words, the functionalities that the framework must offer.

The definition of these requirements is established using the previously defined functionality type, with their individual priority being established using the MoSCoW method. The following list is a reduced count of the functional requirements that were defined for this project. For a complete list, please refer to the document entitled Appendix C – Software Requirements Specification.

Infrastructure Provisioning

FR_01 – IAM Authorization	
Priority: Must	Type: Functionality
Description: Only DevOps Team Leaders must have access to the provisioning of certain elements within the framework, yet DevOps Team Members must not have these same accessed. Given the need to interact with the AWS CloudFormation service whilst looking to apply least-privilege policies to different levels of responsibility, each group within a DevOps Team will have their own role to assume: DevOps Team Leaders, DevOps Team Members, External DevOps Consultants.	

FR-04: Infrastructure Creation	
Priority: Must	Type: Functionality
Description: The Infrastructure creation must be conducted according to the template files provided in the VCS repository, therefore putting into practice Infrastructure as Code, along with its best practices.	

FR-05: Infrastructure Update	
Priority: Must	Type: Functionality
Description: When there is a change made to the infrastructure's template file, there must be a mechanism in place which performs a change to the infrastructure's state according to the new file's version.	

FR-06: Infrastructure Replication	
Priority: Must	Type: Functionality
Description: There must be a means of replicating the production environment on demand, without causing inconveniences to the replicated environment.	

Continuous Integration

FR-27: Automated Pipeline Creation	
Priority: Must	Type: Functionality
Description: When a new branch is created within the git repository, the framework must create a pipeline which performs the automated build and delivery of the application.	

FR-29: Automated Code Build on Commit**Priority:** Must**Type:** Functionality**Description:** The application's source code must be built autonomously, whenever a new commit is made to the git repository.**FR-32: Automated Unit Testing Module****Priority:** Must**Type:** Functionality**Description:** Each function or unit in the application's source code must be able to produce the correct output, based on a certain input. This output must be evaluated and correct on an individual level and carried out autonomously by the framework's pipeline.**FR-33: Automated Docker Image Build****Priority:** Must**Type:** Functionality**Description:** The pipeline must create a docker container image as a final product.**Continuous Delivery****FR-44: Configurable Rollout Strategy – Canary Releases****Priority:** Must**Type:** Functionality**Description:** to avoid any form of downtime to the application when performing an upgrade, when deploying a new image into the EKS cluster there must be an implementation of Canary Releases within the framework.

Canary Release imply the re-routing of user traffic to new versions of the application in an incremental fashion, and the framework must offer the possibility of this same release strategy.

Monitoring Microservices**FR-47: Container CPU Consumption Visualisation****Priority:** Must**Type:** Functionality**Description:** To gain observability into how applications are behaving within their containerized environments, the framework must offer the possibility to visualise the levels of CPU resource consumption in real-time.

FR-56: Monitor Custom Application Metrics**Priority:** Must**Type:** Functionality**Description:** The monitoring solution must interpret and present the custom metrics created by the application.**FR-58: Application Log Querying****Priority:** Must**Type:** Functionality**Description:** The logs produced by the application must be located in a component which allows querying, in order to filter the outputs according to different fields.

5.4 Non-Functional Requirements

Non-functional requirements refer to what the framework must guarantee for the developers. These requirements do not increase the value of the framework from a functional point of view, but rather address how the framework will behave when analysing merely the system's operation and the user's interaction with the framework.

The definition of these requirements is established using the previously defined types: performance, usability and supportability, with their individual priority being established using the same MoSCoW method used to prioritize the functional requirements. As in the previous section, the following list is a reduced count of the non-functional requirements that were defined for this project. For a complete list, please refer to the document entitled Appendix C – Software Requirements Specification.

NFR-1: Least privileged access**Priority:** Must**Type:** Usability**Description:** Developers must only have access to the tools and functionalities that are essential for their work.**NFR-3: Private EKS endpoints****Priority:** Must**Type:** Usability**Description:** The cluster's endpoints must be private to safeguard the application from external and unauthorised attacks.**NFR-6: Infrastructure Change Processing****Priority:** Must**Type:** Supportability

Description: When a change is made to the infrastructure configuration, these changes must be reflected in the environment when the changes are pushed to the repository.

NFR-10: Multiple nodes with multiple pod replicas

Priority: Must

Type: Supportability

Description: Deploy multiple pod replicas to maintain a highly available application..

NFR-12: Logging Referencing

Priority: Must

Type: Supportability

Description: The logs produced by the application must be produced following JSON notation.

Requirement definition is an essential step when developing and planning a development project. From here, it was possible to perform estimates on how long each epic would require to be produced, along with elaborating a plan regarding acceptance and consolidation testing. Both the acceptance and consolidation test planification documents and reports can be analysed in Appendix E – Acceptance Test Plan and Report.

Chapter 6

Framework Architecture

This chapter provides an explanation into the architectural decisions and features that are incorporated in the framework. These views will be divided into the following sections: i) Infrastructure provisioning; ii) Continuous Integration/Continuous Delivery; iii) Microservice Monitoring.

Finally, there will be one final view which will present the entire framework's overview, defining what the product offers toward its users and the value it offers to the organization. As a means of brevity and facilitating reading, a reduced list of the framework's architecture is included in this section. For a more detailed analysis, please refer to the document Appendix D – Software Architecture and Design.

6.1 Infrastructure Provisioning

In this section, there will be a brief explanation of how the process behind provisioning an infrastructure and how the framework processed its creation. There will be a list of elements that are considered as pre-requisites for the framework to be adopted into a project, defining the foundation on which the framework will operate. The framework offers an automation pipeline which autonomously deploys the infrastructure configuration files to the cloud provider for interpretation and creation. This pipeline will be explored in full, as it is the component that offers the most value to this component of the framework, along with the infrastructure itself. Finally, the infrastructure itself will be presented, showing the various components required to construct the cloud-native resources that are required for the microservice application to be deployed and accessible.

Pre-Requisites

Before applying the framework into a production environment, a number of components are required are required to create a foundation and prepare configurations for the framework to work. To create these same pre-requisites, a DevOps Team Leader is required to have previously installed the AWS CLI tool which will then give access to an account which applies the best practice of least privileged authorization. This account will assume a role named DevOps Team Leader which will have full access both to the AWS CloudFormation service, along with all of the components compose the pre-requisite stack (groups of components which cooperate together) which must be provisioned before developing the infrastructure templates. Given that the Team Leader will only be able to access the AWS Services with this account, the possibility of over-stepping these boundaries does not exist. Those components are the following:

- **AWS CodeCommit Repository:** git repository where the infrastructure templates will be maintained and versioned to control alterations and updates that may occur throughout the Infrastructure as Code development process. This will serve as a single point of truth for the elements that require a source to extract the configuration files from;
- **AWS Lambda Function:** serverless instance, responsible for executing snippets of code, without the need for allocating computational resources. This function will be responsible

for transferring the template files from the AWS CodeCommit repository to the Amazon S3 bucket and the creation of the AWS CodePipeline that will be the means used to autonomously provision the infrastructure. These pipelines will only be created when a branch named develop, staging, lab or main is created in the repository;

- **AWS S3 Bucket:** Simple Storage Service responsible for acting as a source from which the AWS CodePipeline component extracts the configuration templates. This component is compulsory, given that AWS CloudFormation templates limit the source component for an AWS CodePipeline to an S3 Bucket;
- **AWS IAM StackChangeRole Role:** AWS IAM Role responsible for giving AWS CodePipeline the permissions necessary for creating and performing changes on the AWS CloudFormation stacks;
- **AWS IAM CodePipelineRole Role:** AWS IAM Role which gives permissions to the AWS CodePipeline component to perform changes and access information present in the previously mentioned S3 Bucket;
- **AWS IAM LambdaRole Role:** AWS IAM Role responsible for allowing the AWS Lambda function to have read-only access to the AWS CodeCommit repository, to the S3 bucket, and read and write access to the AWS CloudFormation to create the pipeline responsible for provisioning the infrastructure;
- **AWS CloudWatch Event Rule:** event listener which detects changes to the AWS CodeCommit repository in terms of creating and deleting branches, so as to allow infrastructure replication, and communicates those changes to the AWS Lambda function to create a pipeline at the rate that it is necessary to replicate an environment;
- **AWS Lambda Permission:** permission entry which gives the CloudWatch Event Rule permissions to trigger the AWS Lambda function when changes are performed to the repository.

Figure 14 shows the process that must be followed to create the pre-requisite elements which are necessary for the remaining framework to function. The DevOps Team Leader, duly authenticated with an AWS IAM Account, communicates with AWS Services via the AWS CLI tool. With this tool, the only requirement is to run the following command:

```
aws cloudformation create-stack \
--stack-name PRE_REQ_STACK \
--template-body file://PreRequisites.yaml \
--capabilities CAPABILITY_NAMED_IAM
```

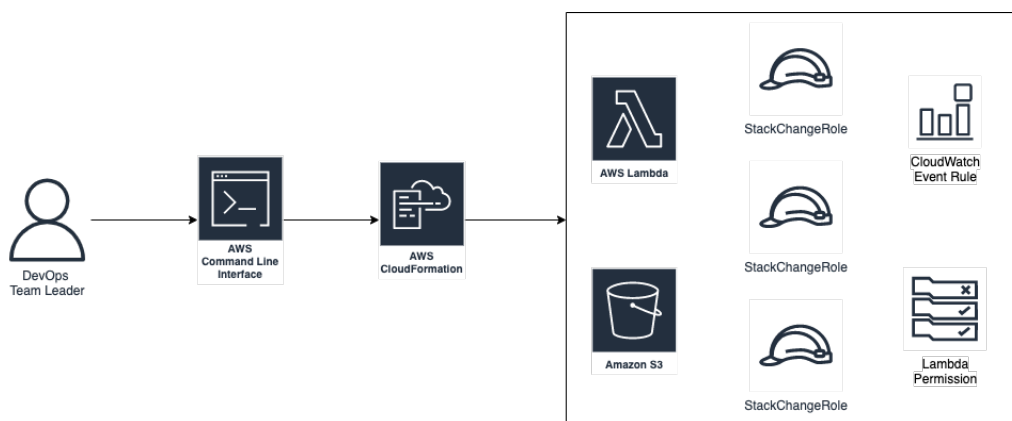


Figure 14. Infrastructure Provisioning Pre-Requisites Creation Process

This command will output the ID of the AWS CloudFormation stack that is created. The value for `PRE_REQ_STACK` is customizable, the file name `PreRequisites.yaml` must match the name of the pre-requisites template and the parameters which are defined in the template file can be overridden by using the `-parameters` flag.

Provisioning Process

The act of provisioning an infrastructure for a microservice begins with template files which are maintained in a Version Control System. These files contain the configuration of the infrastructure written in YAML – a markup language based on key-value pairs which define the desired state of the components where the application will be running in. The tool used for version control is AWS CodeCommit. In the event of a new version of the code being committed to the repository, AWS CodePipeline is notified of this new version and acts as an intermediary between the repository and Amazon’s Web Services, notifying AWS CloudFormation that there is an update pending on the infrastructure’s configuration. The files are then interpreted and used to create or update the desired environment for the application.

The idea of implementing this component in the framework is to develop a means for DevOps Maintainers to make the most of what cloud computing and Infrastructure as Code has to offer. Ultimately, the goal is to prepare a solution that, at the click of a button or by running a single command, the configuration files are uploaded to the code repository and provision an infrastructure, without the need for anymore interaction from the developers and no single, step-by-step configuration is required.

Furthermore, when looking to replicate the components themselves, the process is much easier and robust. A developer is only required to replicate the configuration files or override the parameters to have an identical environment to the one that has been previously deployed.

Automation Pipeline

Once the pre-requisites have been implemented, the means of developing and delivering the infrastructure templates must be implemented. This is accomplished through the use of an automation pipeline which the AWS Lambda function is responsible for creating. Whenever a change is detected in the AWS CodeCommit repository, the Lambda function proceeds to transfer the files from the repository to the S3 Bucket, followed by creating an AWS CodePipeline will then use the files in the S3 Bucket and communicate them to the AWS CloudFormation service and create the infrastructure.

Figure 15 represents the process that begins at the development phase of creating the infrastructure templates which are then committed to the repository.

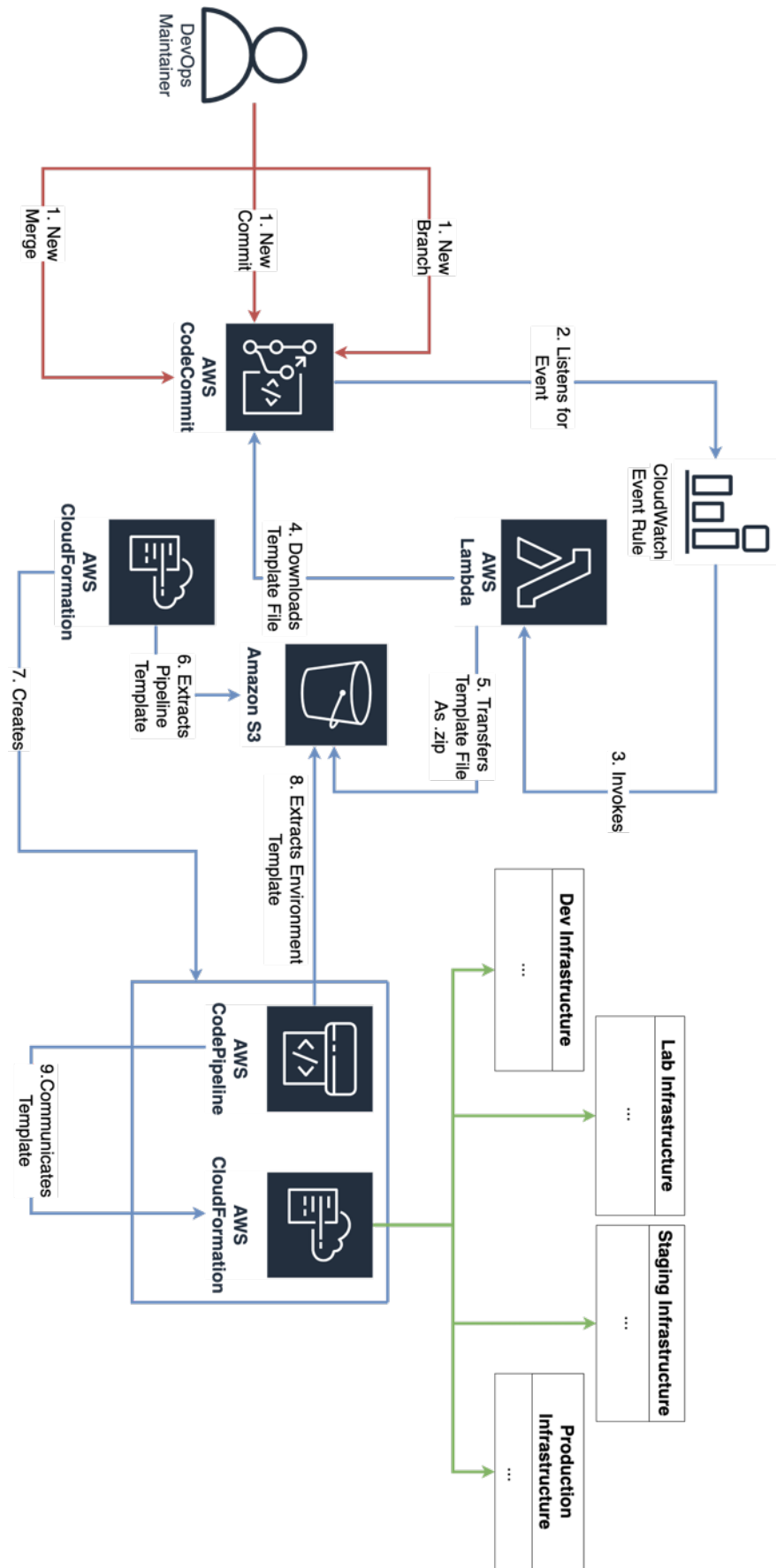


Figure 15. Automation Pipeline Workflow for Infrastructure Deployment

A DevOps Maintainer performs a change to the repository, be it a new commit, a new branch with one of the aforementioned names or a merge between branches, the CloudWatch Event Rule detects this change and deploys the AWS Lambda function. Once invoked, the function then transfers the source templates in the repository to the S3 bucket and, as soon as the change is complete, the AWS CodePipeline is triggered and extracts the infrastructure templates from the bucket and communicates them to the AWS CloudFormation service. After receiving the templates, the AWS CloudFormation service then provisions the different elements that are defined in the template files.

The following Figure 16 presents a representation through an Activity Diagram of the different phases that are necessary for the process from A DevOps Maintainer to the AWS CloudFormation.

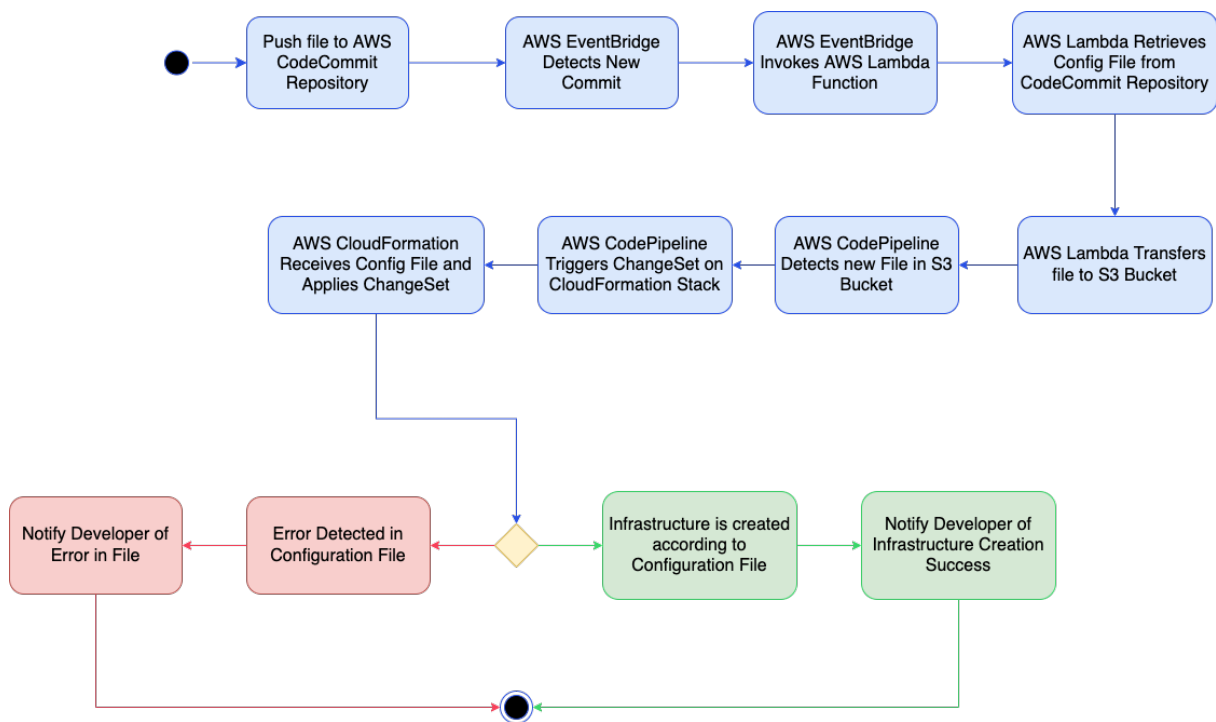


Figure 16. Create Environment Activity Diagram

These components add value to the framework due to the fact that they offer a means of automating the process of provisioning a cloud-native infrastructure in a question of minutes, whilst simultaneously offering the possibility to treat the templates in the same manner as an application’s source code. A DevOps Maintainer can perform changes to an infrastructure that is still in its development phase without interfering with the configuration which is in use in a production environment. This also offers the possibility to create an environment on demand, which offers a solution for the use-case given when an error occurs in the production infrastructure and requires a change without causing down-time to the remaining elements in the stack. The implementation of that change will require testing and evaluation, in which case the framework is prepared to fulfil this requirement.

Infrastructure Elements

Once the automation pipeline was concluded, it was necessary to develop an infrastructure for the microservices application. This infrastructure must be developed in as much a generic fashion as possible in order to have a solution that can integrate into multiple projects without

the need for reconfiguration. The following subsections explain the different components that would provide resources for the application.

The following figure 16 shows a graphical representation of the components that compose the infrastructure on which the microservices application will be deployed. A more in-depth analysis of each element follows.

Networking - The components that compose this section are responsible for allowing network connectivity and communication amongst the various elements that will integrate the infrastructure. Those elements are the following:

- **Virtual private cloud (VPC):** a configurable, centralized and isolated virtualized networking environment into which a range of connectivity tools can be inserted in an effort to provide distributed means of communication;
- **Availability Zones:** AWS Cloud computing resources are housed in highly available data centre facilities. To provide additional scalability and reliability, these data centre facilities are located in different physical locations. Availability Zones are distinct locations within an AWS Region that are engineered to be isolated from failures in other Availability Zones. They provide inexpensive, low-latency network connectivity to other Availability Zones in the same AWS Region [42].
- **Subnets:** a range of possible IP addresses within a VPC which is made available within a specific availability zone. Best practices recommend that the number of subnets that are deployed should always be provisioned in different availability zones, so as to guarantee that, in the event of failure in one of the zones, the other can compensate and mitigate possible down time of the resources. In this solution, there are three different subnets that are created within the VPC, one public subnet on which the bastion host will be deployed, and two private subnets on which the EKS Cluster and the RDS application will be deployed, respectively. The difference between a public and private subnet can be defined as the following [43]:
 - o **Public Subnet:** the subnet traffic is routed to the public internet through an internet gateway or an egress-only internet gateway;
 - o **Private Subnet:** the subnet traffic can't reach the public internet through an internet gateway or egress-only internet gateway. Access to the public internet requires a NAT device. In this case, a NAT Gateway;
- **NAT Gateway:** A NAT gateway is a Network Address Translation (NAT) service. You can use a NAT gateway so that instances in a private subnet can connect to services outside your VPC, but external services cannot initiate a connection with those instances [44]. Essentially, a one-way means of network connectivity which allows elements within a VPC to communicate with external sources, but protects these from external messages;
- **Internet Gateway:** An internet gateway is a horizontally scaled, redundant, and highly available VPC component that allows communication between a VPC and the internet. An internet gateway enables public subnets to connect to the internet if the resource has a public Ipv4 address or an Ipv6 address.

Security Groups - A security group acts as a virtual firewall, controlling the traffic that is allowed to reach and leave the resources that it is associated with. For each security group, *rules* are added that control the traffic based on protocols and port numbers. There are separate sets of rules for inbound traffic and outbound traffic. The infrastructure that is created will have the following security groups:

- **VPC Security Group:** default VPCs and any VPCs that are created come with a default security group. With some resources, if they are not associated with a security group when created the resource, AWS associate these resources with the default security group;
- **Bastion Security Group:** this Security Group is responsible for guaranteeing both which external elements have permission to access the EC2 instance and via which protocol (in this case, a range of IP addresses that belong to the organizations network via an SSH connection) and identifies the host itself when looking to communicate with other components in the Availability Zone;
- **EKS Security Group:** the EKS Security Group defines which components are permitted to access the different nodes within the EKS Cluster. This Security Group allows connections from the elements within the Bastion Security Group, exclusively.

AWS EKS Cluster - Amazon Elastic Kubernetes Service (Amazon EKS) is a managed service that you can use to run Kubernetes on AWS without needing to install, operate, and maintain your own Kubernetes control plane or nodes [45]. This service runs and scales the Kubernetes control plane across multiple AWS Availability Zones to ensure high availability, whilst also automatically scaling control plane instances based on load, detecting and replacing unhealthy control plane instances, and it provides automated version updates and patching for them. It is also able capable of running up-to-date versions of the open-source Kubernetes software, so that all the existing plugins and tooling from the Kubernetes community can be used by developers. The application's distributed communication and orchestration will be managed by the EKS service.

AWS Elastic Load Balancer - The Elastic Load Balancer (ELB) is an intermediary entity which automatically distributes incoming application traffic across multiple targets and virtual components in one or more Availability Zones. The ELB allows for application scaling without requiring complex configurations. An Application Load Balancer makes routing decisions at the application layer (HTTP/HTTPS), supports path-based routing, and can route requests to one or more ports on each container instance in the cluster. For the product of this internship, the ELB will be responsible for distributing and controlling the traffic made to the application, routing the requests to the different nodes of the EKS cluster.

AWS WAF – The Amazon Web Application Firewall (WAF) component is responsible for protecting web applications from common web exploits that may have an effect on the availability or security of the information that is manipulated by the application [46]. It also allows control over the flow of traffic by enabling the possibility of creating security rules that control bot traffic and block common attack patterns, such as SQL injection or cross-site scripting. In the case of the infrastructure that is created, the WAF will be used to filter the request that can be made to the application according to request origin's IP address. The WAF has a direct association with the Load Balancer which controls and manages the flow of requests to the different instances of the application.

AWS Route53 – Given the nature of the application being a website, the AWS Route 53 service is a highly available and scalable service that offers the possibility of Domain Name System (DNS) translation [47]. It is designed to give developers and businesses an extremely reliable and cost-effective means of routing end users to Internet applications by connecting user requests to infrastructure running in AWS and can also be used to route users to infrastructure outside of AWS. In the case of this internship, the service re-routes traffic made to the domain “awsdummy.online” to the endpoint associated with the Elastic Load Balancer, incorporating the HTTPS protocol.

Figure 17 represents a macro-view into the components that compose the infrastructure created for the application to run on.

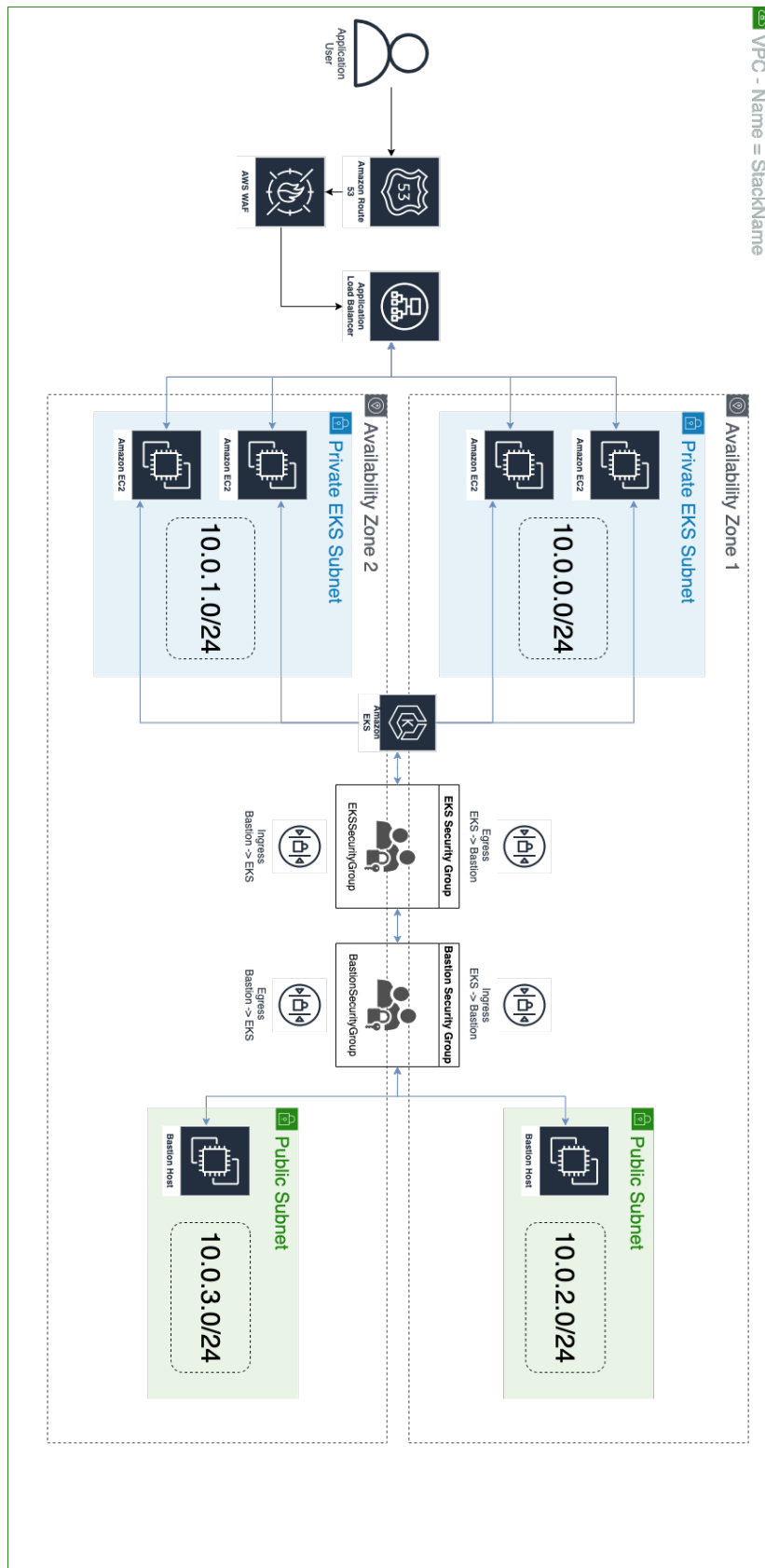


Figure 17. Full Infrastructure Components Macro-View

6.2 Continuous Integration/Continuous Delivery

The contents covered in this section focus on the elements that are responsible for conducting the tasks of Continuous Integration and Continuous Delivery of the framework. In this component of the framework, the source code of the application will have to be built into an executable artifact, with automated tests being performed both on the source code and the artifact itself. Once the Continuous Integration phase concludes with the creation of the artifact, the product must then be deployed autonomously and progressively into the production environment.

Pre-Requisites

Similarly to the Infrastructure Provisioning component of the framework, the CI/CD phase also requires that a list of pre-requisites be provisioned before developers can make use of the autonomous functionalities that the pipeline has to offer. The pre-requisites that must be provisioned are the following:

- **AWS CodeCommit:** git repository where the application source code will be maintained and versioned to control alterations and updates that may occur throughout the development process. This will serve as a single point of truth for the elements that require a source to extract the configuration files from (AWS CodeBuild and Argo CD);
- **AWS Lambda Functions** – 2 Functions (events + emails):
 - **CreatePipelineLambdafunction:** this function is responsible for transferring the template files from the AWS CodeCommit repository to the Amazon S3 bucket and the creation of the AWS CodePipeline that will be the responsible for detecting the changes in the repository and triggering the AWS CodeBuild project which will automate the static code analysis, dependency verification and unit testing on the application source code. These pipelines will only be created when a branch named develop, staging, lab or main is created in the repository;
 - **SendOWASPEmailLambdaFunction:** this function is responsible for extracting the OWASP dependency report from the S3 Bucket responsible for maintaining a persistent version of the report, and emails that same report to the Development Team Leader via email.
- **AWS S3 Buckets** – 2 Buckets (events + emails):
 - **Source Code Bucket:** S3 bucket which stores the application source code and acts as a source for the AWS CodePipeline service which performs the autonomous tasks of Continuous Integration;
 - **OWASP Report Bucket:** S3 bucket to which the dependency verification report is sent and maintained. This bucket is separate to the source code bucket to maintain modularity and separation between different concepts.
- **AWS IAM StackChangeRole Role:** AWS IAM Role responsible for giving AWS CodePipeline the permissions necessary for creating and performing changes on the AWS CloudFormation stacks;
- **AWS IAM CodePipelineRole Role:** AWS IAM Role which gives permissions to the AWS CodePipeline component to perform changes and access information present in the previously mentioned S3 Bucket;

- **AWS IAM LambdaRole Role:** AWS IAM Role responsible for allowing the AWS Lambda functions to have read-only access to the AWS CodeCommit repository, to the S3bucket, and read and write access to the AWS ECR service, to the AWS Systems Manager service and to the CodeBuild service to perform changes during the Continuous Integration phase;
- **AWS IAM CodeBuildRole Role:** AWS IAM Role responsible for allowing the AWS CodeBuild Projects to have read and write access to the AWS CodeCommit repository, to the S3 bucket, and read and write access to the AWS CloudFormation to create the pipeline responsible for provisioning the infrastructure;
- **AWS CloudWatch Event Rule:** event listener which detects changes to the AWS CodeCommit repository in terms of creating and deleting branches, so as to allow infrastructure replication, and communicates those changes to the AWS Lambda function to create a pipeline at the rate that it is necessary to replicate an environment;
- **AWS Lambda Permission:** permission entry which gives the CloudWatch Event Rule permissions to trigger the AWS Lambda functions when changes are performed to the repository or when a new version of the OWASP report is produced.

Figure 18 shows the process required to provision these pre-requisites before making use of the CI/CD pipeline. A Development Team Leader, duly authenticated with an AWS Dev Team Leader account, deploys the contents of the AWS CloudFormation stack in the PreRequisites.yaml file with the same command as in the previous Infrastructure Provisioning Template, with the correct adjustments to not cause conflicts with the previous pre-requisite stack. An example of this command can be the following:

- `Aws cloudformation create-stack \`
`--stack-name CI_PRE_REQ_STACK \`
`--template-body file://PreRequisites.yaml \`
`--capabilities CAPABILITIES_NAMED_IAM`

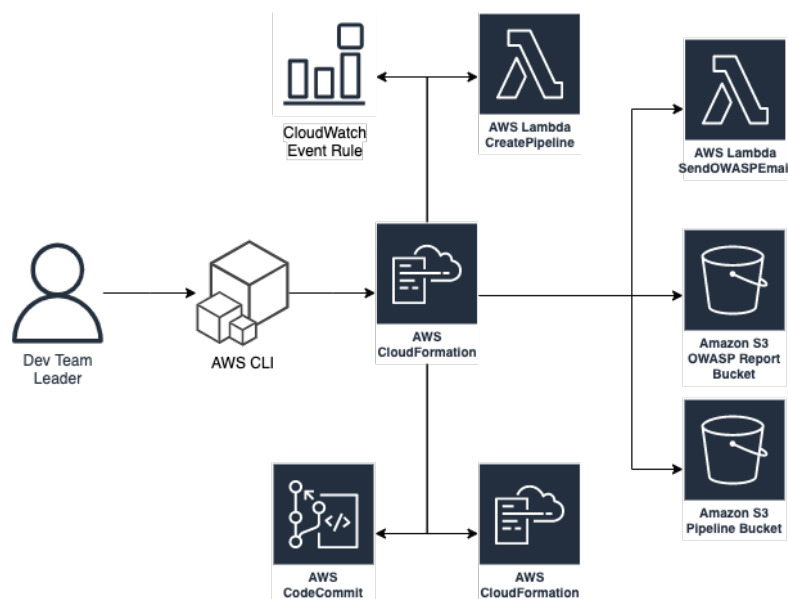


Figure 18. Pre-Requirement Elements Created by the Development Team Leader

This command will output the ID of the AWS CloudFormation stack that is created. The value for CI_PRE_REQ_STACK is customizable, the file name PreRequisites.yaml must match the

name of the pre-requisites template and the parameters which are defined in the template file can be overridden by using the `-parameters` flag, just as in the Infrastructure Provisioning Pre-Requisite phase.

Continuous Integration Process

The software development process begins at the developer's end. The software developer performs changes to the application's source code to perform fixes on errors and bugs that may exist, or to implement new functionalities that may be required to add more value to the application. Figure 19 shows the process that was previously described, and also the steps that are followed after the AWS CodeBuild project is responsible of carrying out. In the event of a new branch being created with `develop`, `staging`, `lab` or `main` as their name, the AWS CloudWatch Event Rule detects this change and communicates with the AWS Lambda service to invoke the function `CreatePipelineLambdafunction` to create a pipeline which begins with the new branch and extends to the AWS CodeBuild project. Each branch has its own AWS CodePipeline service, with its own individual CodeBuild Project. There is no need for an individual S3 Bucket for each branch, as it is possible to create directories which will be associated with each branch.

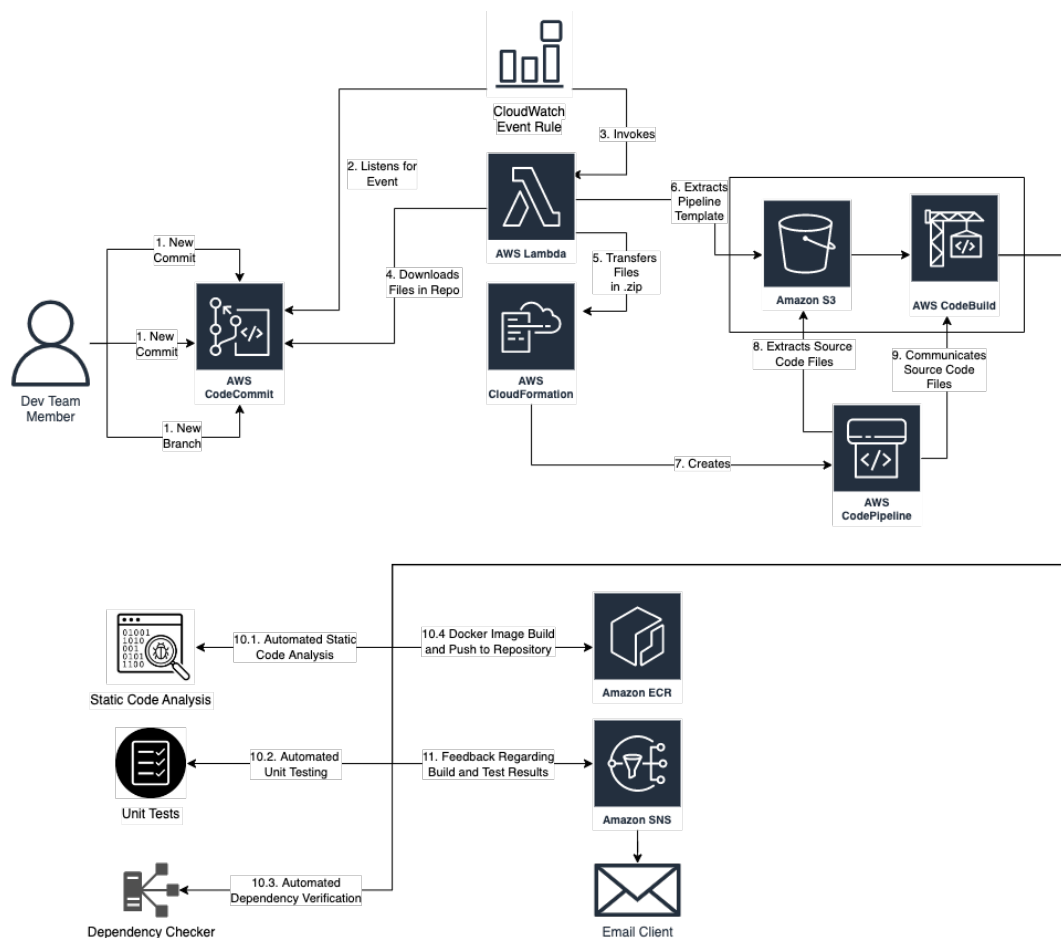


Figure 19. Continuous Integration Workflow Diagram

When a new commit is performed, the AWS CloudWatch Event Rule communicates with the `CreatePipelineLambdafunction` function which receives an event that communicates the event of a new commit being made. The function then clones the files in the repository, compresses

them into a .zip file, and transfers the compressed files to the AWS S3 Bucket. This process is required due to the fact that the CodePipeline element, when created with AWS CloudFormation templates, requires an AWS S3 Bucket as its source, and the files in the Bucket must be in a .zip file to be versioned and managed by the bucket itself. The new version of the .zip file is detected by the CodePipeline service and proceeds to trigger the AWS CodeBuild project associated with the respective branch.

As the project is deployed, the Development Team Leader receives an email notifying that the build began. The Build first analyses the source code with a static code analysis tool to guarantee that coding conventions are followed correctly and that it is ready to be compiled. In the case of the example application source code which is written in Node.js, the tool used is JavaScript-Lint [48]. If an error occurs, the build fails, and the Development Team Leader receives an email with this information. This notification service performs this action in all situations where the build changes its state (“IN PROGRESS”, “FAILED”, “STOPPED”, “SUCCESSFUL”).

If this process is successful, the OWASP Dependency Checker [49] performs an analysis of all the libraries and modules that are used by the application. This tool performs a cross-reference analysis of the application with all known vulnerabilities that may exist in third-party software, with a database that extends to vulnerabilities detected since its first released version in 2013. When the dependency checker fulfils its analysis of the application, a report is produced which explains where vulnerabilities are and if there is a possibility for external attack or data leaks. This event in the AWS CodeBuild project exports this report to the OWASP Report Bucket and then triggers the Lambda Function named SendOWASPEmailLambdaFunction which extracts the dependency report file from the bucket and emails it to the Development Team Leader.

If there are no vulnerabilities detected, the application’s code is then subjected to unit tests to its various functions. These unit tests are conducted by the Unit.js library which runs on the Node.js browser alongside the application and tests the output of the various functions. Once complete, the AWS CodeBuild project then builds a Docker Container Image according to a Dockerfile which is included with the application’s source code. This image is then forwarded to the AWS ECR repository, and the build phase is concluded.

Having concluded this process, the executable is then placed into an Elastic Container Repository from which the image of the service is then pulled by the Elastic Kubernetes Service and deployed into the target environment. Considering that the need for phased rollouts, having finalized and passed all previous tests, the container image will then be transferred into a production ECR from which, when instructed, the production cluster will pull the new version of the application and apply its changes.

The Activity Diagram in Figure 20 serves the purpose of showing the different steps involved in the Continuous Integration phase of the pipeline, when a change is made to the source code of the application. Beginning with the commit on behalf of the developer, the source code is pushed to the AWS CodeCommit repository and then extracted by the Lambda function and uploaded to the S3 bucket. The AWS CodeBuild project then begins the build process of static code analysis, vulnerability verification and unit tests on the source code, and then builds and pushes a new Docker Container Image to the AWS ECR repository.

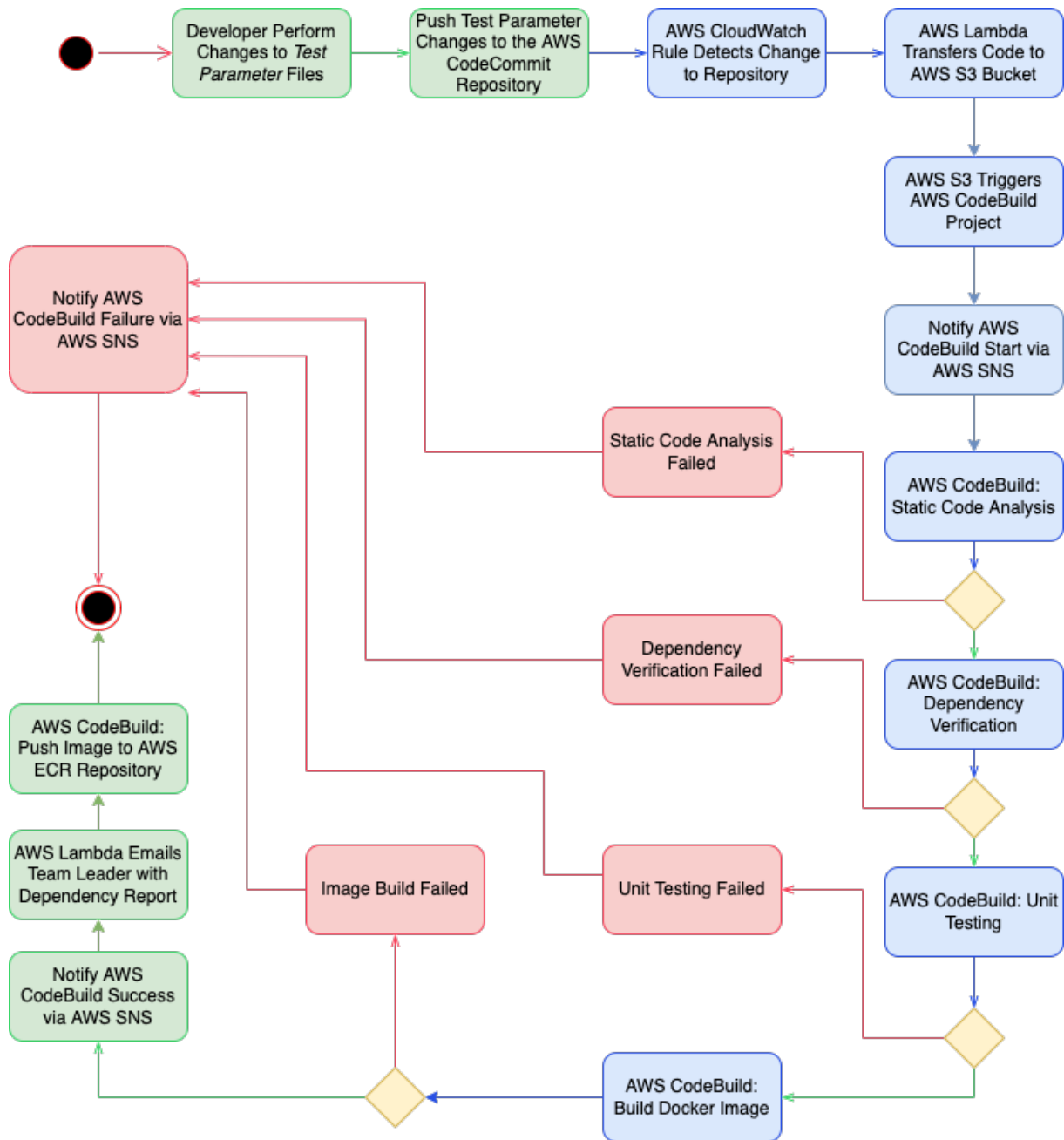


Figure 20. Continuous Integration Automated Build Activity Diagram

The Sequence Diagram in Figure 21 represents the exchange of messages between the various entities during the Continuous Integration phase. The AWS CodeCommit repository reports an alteration in the source code to the AWS CloudWatch service. This service consequently invokes the AWS Lambda function to perform a transfer of the source code files to the project's S3 Bucket. From here, the AWS CodeBuild project is deployed and, if successful, pushes the Docker Image to a repository in the AWS ECR service.



Figure 21. Continuous Integration Automated Build Sequence Diagram

Continuous Delivery Process

The process behind Continuous Delivery begins during the AWS CodeBuild project execution. Once the build phase is complete and the new docker image has been pushed to the ECR repository, the post-build phase begins with an authentication of the AWS CodeBuild instance which extracts the account credentials from AWS Systems Manager. Once authenticated with the correct credentials, the AWS CodeBuild instance communicated with the Argo CD instance, running in the AWS EKS cluster as a Daemon Set and notifies that a new version has been pushed to the AWS ECR repository.

The diagram in Figure 22 represents this workflow, including the CI phase, in efforts to show how the software development pipeline is fully automated and extends from the software developer committing their changes to the source code through to the EKS cluster, without the need for manual intervention. If required, this process can be paused and configured in a way that approval steps may be in place.

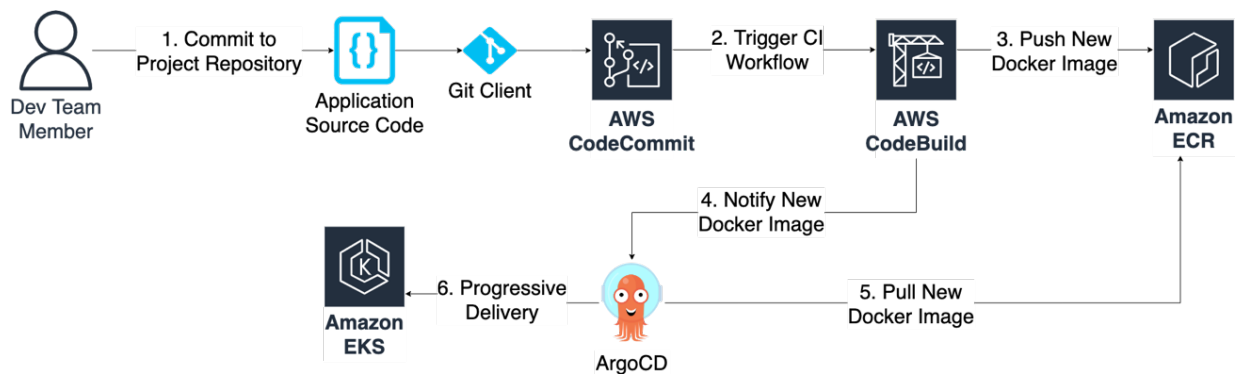


Figure 22. CI/CD Workflow Production Diagram.

Argo CD allows the possibility to deploy the new image using blue-green deployments, A/B Testing or Canary Releases, depending on the intended deployment method. Once the AWS CLI tool is authenticated, it then notifies the Argo CD tool that a new version of the application is ready to be deployed into production and that the application project requires synchronization. This new version is pulled by Argo CD and then deployed progressively according to the rollout's configuration file, which is kept together with the source code repository.

Argo CD is implemented as a Kubernetes controller which is continuously monitoring applications and compares their current state against the desired target state specified in the manifest file which is kept in the AWS CodeCommit repository. A deployed application whose current state deviates from the target state is considered "OutOfSync". Argo CD reports & visualizes the differences, while providing a means of automatically or manually synchronizing the live state back to the desired target state. Any modifications made to the desired target state in the git repository can be automatically applied and reflected in the specified target environments.

Argo Rollouts - Argo Rollouts is a Kubernetes controller [50] and set of CRDs which provide advanced deployment capabilities such as blue-green, canary, canary analysis, experimentation, and progressive delivery features to Kubernetes. Argo Rollouts integrates with ingress controllers and service meshes, performing changes to the traffic they receive and offers the possibility to gradually shift traffic to the new version during an update. The Argo Rollouts controller will manage the creation, scaling, and deletion of ReplicaSets, these being

defined by the spec.template field inside the Rollout resource, which uses the same pod template as the deployment object [51].

Automated Rollbacks - When looking to perform continuous delivery of software, there must be a method of recovering from failure, without causing downtime to the application. Argo Rollouts further offers the possibility of performing an analysis of the metrics of the application as means of performing an automated rollback in the event of the new version failing or producing errors in production. This analysis is performed by an AnalysisTemplate object which carries out an evaluation of the application’s metrics during a news release. The AnalysisTemplate queries the metrics endpoint which is exposed by the application and are harboured in a Prometheus instance.

Figure 23 represents the activities that take place when performing a phased rollout with the strategy being a Blue-Green Deployment.

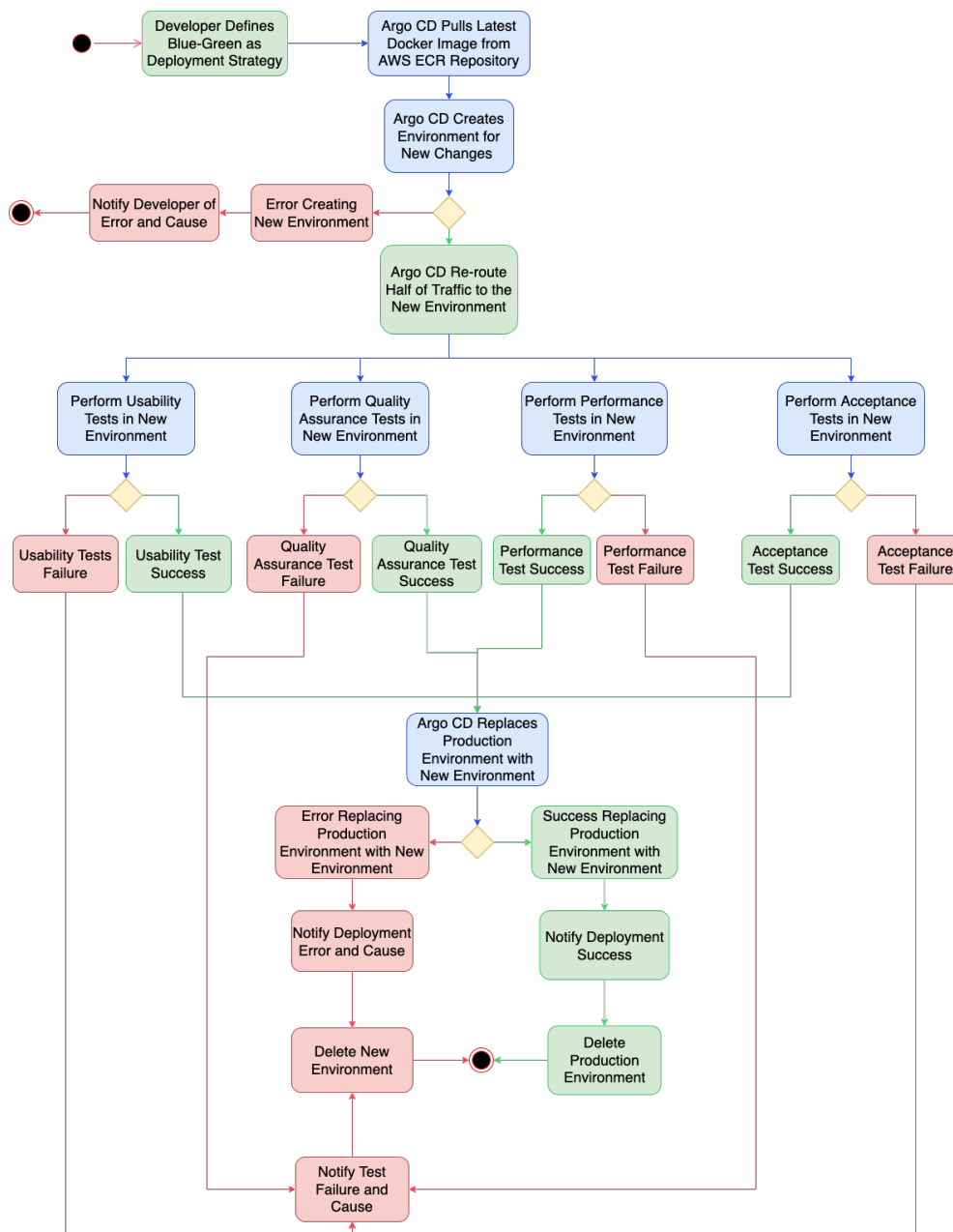


Figure 23. Blue-Green Deployment Activity Diagram

6.3 Monitoring Microservices

This section explains the functionalities implemented to perform the task of monitoring the microservice application and the cloud-native infrastructure. Here, the tools chosen to monitor various metrics that exist within an AWS EKS cluster must offer insights into the metrics of each node, the pods they run and the number of requests that are processed by the service. The metrics exposed by the various services must be presented with a visualization tool in efforts to translate and present the information in a way that is both easy to understand and that offers total observability of the application.

The following figure 24 represents a general overview of how the monitoring solution is implemented, with both Prometheus and Grafana regarding metrics, and the Elasticstack regarding log management.

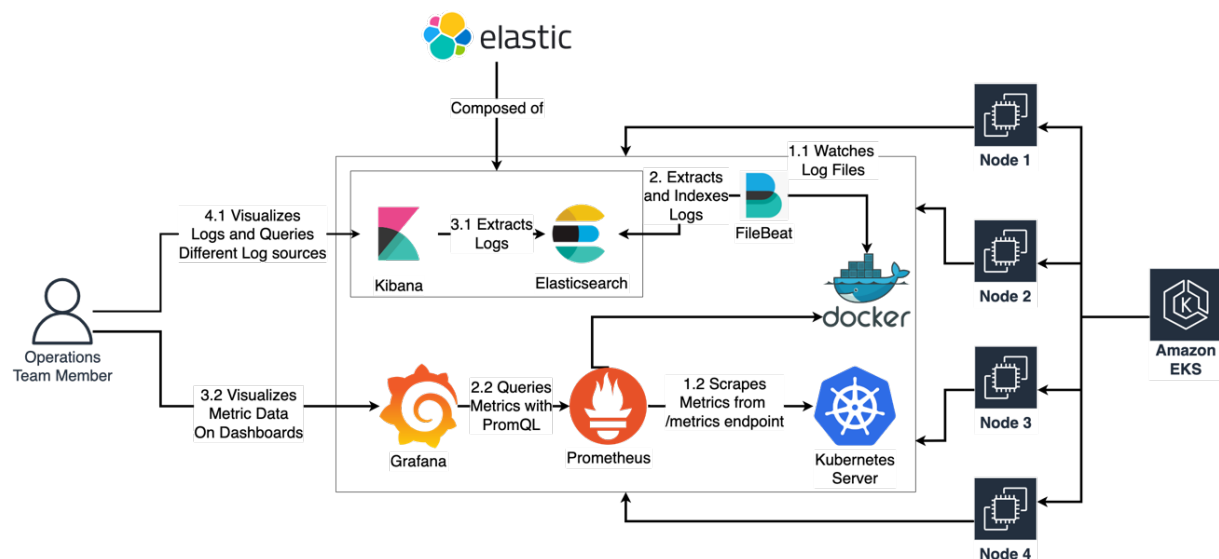


Figure 24. Macro-view Microservice Monitoring Solution

Metrics Analysis

One means of having observability into a microservices application is through the analysis of metrics. Metrics analysis involves observing resource consumption of the application, levels of CPU usage, memory usage and network bandwidth usage of the AWS EKS Cluster, its nodes and the container resources that the different service consume.

Figure 25 represents the workflow that is followed for Operations Teams to analyse and access the metrics of the application and the infrastructure.

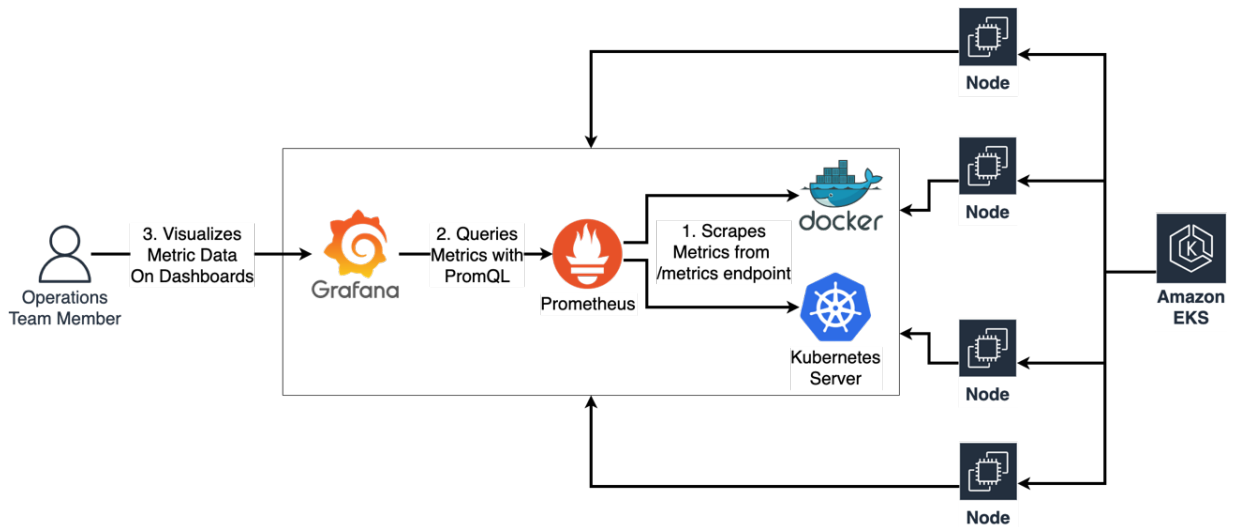


Figure 25. Metrics Monitoring Workflow

Each of the nodes within the AWS EKS Cluster runs an instance of Prometheus which scrapes both the Kubernetes server for cluster and node metrics, whilst also extracting metrics from the application running in the form of a docker image, all from the endpoint names “metrics”. From here, these metrics that are scraped are indexed into a time-series database and can be queried using PromQL: a lightweight, flexible query language which extracts the metrics according to the timestamp at which they are recorded. Grafana makes use of this query language to retrieve the metrics that are stored on within the Prometheus database, and presents them in a much easier way, adding the more information and forms of showing the information, making use of gauges, histograms and counters.

Figure 26 serves as an example for the amount of information that can be viewed by using Grafana as a visualization tool. In this example, there are five panels included in the dashboard: the amount of CPU-time occupied by the image over time; the amount of memory used by the service; the number of requests that each instance of the service has answered to; the number of requests that return a code 400; and finally the number of requests that return a code 200.



Figure 26. Grafana Dashboards Example

Grafana allows the grouping of numerous points of observation in a centralized location. Each panel of the dashboard performs its own query to the Prometheus server and exposes the results obtained, in real time.

The following figure 27 presents a sequence diagram which shows the exchange of messages between Grafana, Prometheus and the AWS EKS Cluster to access Cluster metrics offered by Kubernetes by default.

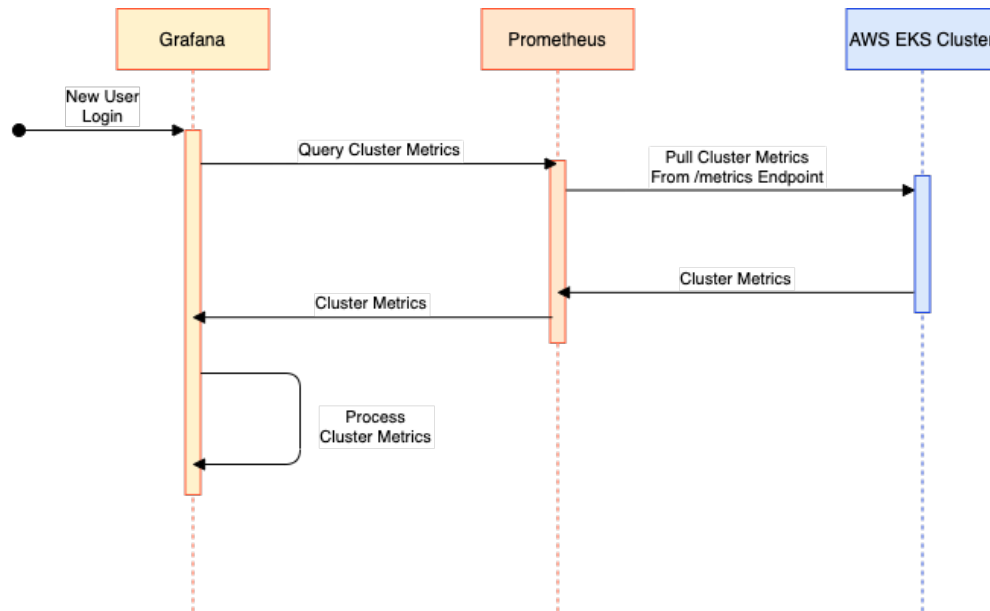


Figure 27. Sequence Diagram of the Process Behind Cluster Metrics Analysis

Application Log Analysis

Regarding service logs, the Elasticstack will be the tool used to process and allow access to the logs produced within the cluster. The stack is composed of three elements, Elasticsearch, Filbeat and Kibana, their respective tasks have been mentioned in chapter 3 – State of the Art. The tool chosen for gaining observability into the applications functionalities is the Elastic Stack. It's comprised of Elasticsearch, Kibana, Beats, and Logstash.

Elasticsearch - allows storage, searching, and analysis of logs with speed at scale.

Kibana - allows data exploring with visualizations, from waffle charts and heatmaps to time series analysis and beyond. Kibana offers preconfigured dashboards for diverse data sources, live presentations to highlight KPIs, and management for deployment in a single User Interface; and Beats which are a free and open platform for single-purpose data shipping. A cluster can have multiple beats operating simultaneously, extracting data from hundreds or thousands of machines and systems to Elasticsearch. For this Framework, Filebeat will be used.

Filebeat - offers a lightweight means of forwarding and centralizing logs and files by shipping with modules for observability and security data sources that simplify the collection, parsing, and visualization of common log formats down to a single command. They achieve this by combining automatic default paths based on your operating system, with Elasticsearch Ingest Node pipeline definitions, and with Kibana dashboards. Plus, a few Filebeat modules ship with preconfigured machine learning jobs.

Figure 28 represents, graphically, the workflow involved when looking to analyse the logs produced by the various services within the microservice application. Filebeat is responsible

for accessing the logs produced by the STDOUT of each service. The STDOUT contents are routed to temporary files in the `/var/log/containers` directory, where each service's log file is identified by the hostname on which the service runs.

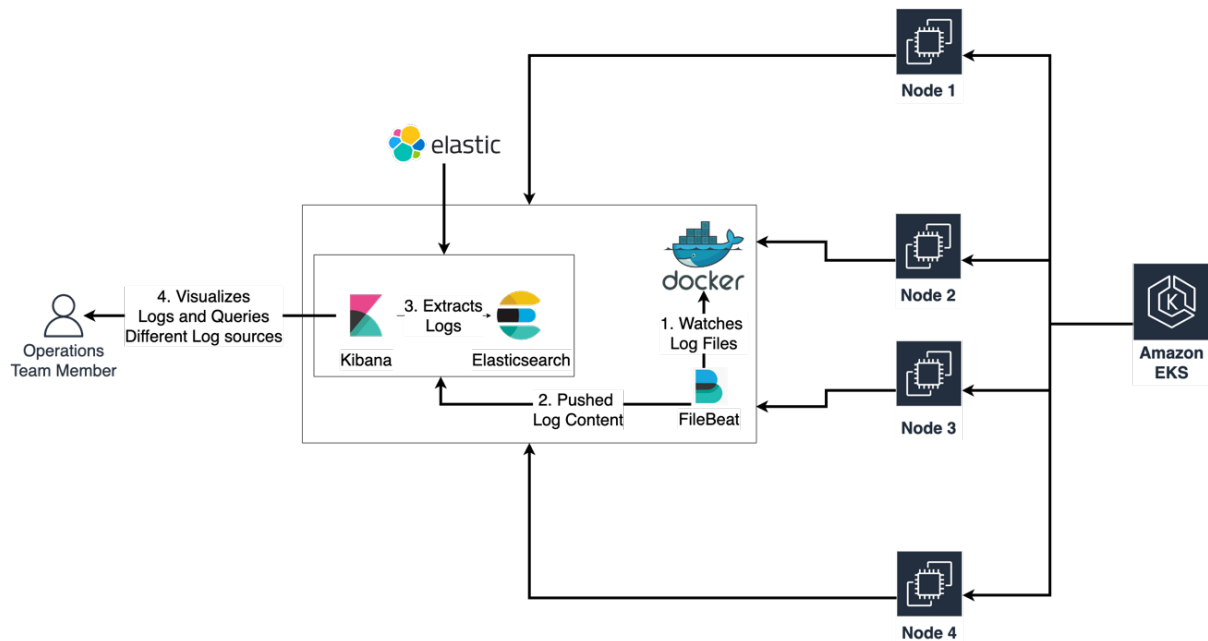


Figure 28. Log Analysis Workflow

The AWS EKS offers the management of the log files by default, therefore not requiring log file rotation nor an extra mechanism to manage the amount of memory occupied by the files. Filebeat captures the contents of these log files and forwards them to Elasticsearch for indexing [52]. Elasticsearch performs indexing, parsing and aggregates the logs received from Filebeat and makes them available for Kibana to perform querying and retrieval of their contents.

The following Figure 29 presents a Sequence Diagram of the messages that are exchanged between the different tools that make it possible to perform log analysis on the microservices that compose the application.

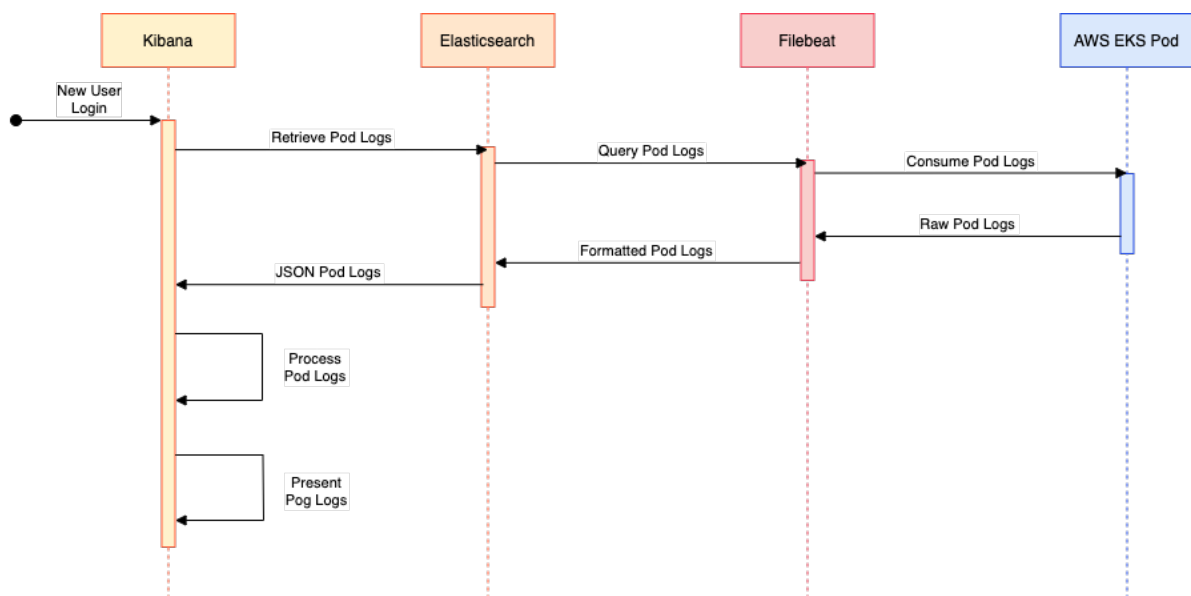


Figure 29. Sequence Diagram for Viewing Pod Metrics

6.4 Challenges and Difficulties

When producing the various components of this framework, many difficulties arose, where I was met by many different challenges and opportunities to grow. The each of the different epics offered their own problems and solutions throughout the development process.

When elaborating the Infrastructure Provisioning components, implementing the different elements within the infrastructure was the most interesting task to accomplish, as it required working with multiple services that are offered by AWS. Having to orchestrate the different tools and conjugate them into a fully working solution for a microservices application was difficult at first given my lack of experience with the microservice architectural pattern. This lack of information was mitigated by the extensive documentation and explanation of how the different services work and their best practices.

The CI/CD components were more straightforward during implementation. Difficulties arose when looking to understand which the more appropriate unit tests were to be carried out during the application build process. Developing with Argo CD brought some difficulties when understanding how to configure repository access and the application's project organization had to be conducted. Having defined the correct organization, associating the AWS CodeCommit repository with Argo CD was simple.

Finally, implementing the monitoring solution did not offer difficulties, given the ease with which the tools are configured.

Chapter 7

Verification and Validation Process

This chapter contains an explanation of the process carried out to guarantee that the produced framework follows the objectives that were defined at the beginning of the internship. The functionalities tested are those defined in the document titled Appendix C - Software Requirements Specification. This chapter presents the formal validation process followed to guarantee the quality with which the framework was developed. For more information, please refer to the document titled Appendix C - Software Requirement Specification Document and Appendix E – Acceptance Test Plan and Report.

As in previous chapters, this chapter will be divided into four sections: Infrastructure Provisioning, Continuous Integration, Continuous Delivery and Application Monitoring. In each of these sections there will be a reduced list of the more essential Base States used to perform tests on the framework, along with the respective Test Cases used to evaluate if the framework performs as expected. All the tests conducted against the framework will be carried out on the computer which WIT Software made available for this project. The tests will be carried out by the author of this document as well.

7.1 Verification Process

Throughout the project, there was a plan of verification in place to guarantee the quality of the developed components during their creation. Whenever a user-story was developed, a progressive evaluation strategy was adopted where each component was required to be compatible with the previously developed components.

At the end of every week, a demonstration was made to the Product Owner and Scrum Master to guarantee that the requirements were being fulfilled correctly, to maintain a coherent and consistent development process, and to introduce a means of verifying that the product was being developed to the organization's expectations. These demonstrations were required to have a list of tasks that were completed during the week, the goals that were achieved, and eventual problems that may have occurred during the development process that week. The demonstration would also have to show the work plan that was in place for the following week, defining the objectives that would have to be met by the next week. The larger demonstrations were shown by the end of each sprint, where the entire state of the framework was presented to the coordinators, along with the expectations for the next sprint.

The verification process was similar in the various components, with a few differences between each other, given their area of study. Regarding Infrastructure Provisioning, each developed component was required to operate with the others within the application's architecture. The infrastructure was required to be deleted and rebuilt on a daily basis, guaranteeing that the components were not dependant on static configurations, which also reinforces the importance of automation and robustness of the infrastructure setup; in Continuous Integration, the pipeline would have to be as generic as possible to be used by different applications. The process would have to be immutable, regardless of the application that was included in the AWS CodeCommit repository; regarding Continuous Delivery, the product of the Continuous Integration phase was the single point of truth when looking to deploy changes into the application's production environment. This product, the Docker Container Image, would then be used in the different

deployment strategies that could be configured according to the project's necessities; and finally the monitoring component was required to make use of the application within its production environment. Once deployed, a need arises to obtain information and observability into the application. To achieve this, the tasks that the application carries out are controlled by accessing the logs that are streamed from the application. The resources that are used by the application allow for improvements from an efficiency point of view.

The verification process occurred naturally, given that each component was dependant on the other. Once presented to the coordinators, and tested for the demonstrations, the following tasks could be executed.

Task #1 – [14/02/2022 – 25/02/2022]: The first task was the creation of the project management documents for the second semester. This task started on the 14th of February and lasted until the 25th of February. The documents were verified by meeting with the Product Owner and the Scrum Master to guarantee that their contents were correct according to the objective of the internship.

Task #2 – [28/02/2022 – 01/04/2022]:

Once the planning process was complete, the development process could begin, starting with the creation of the Infrastructure that would house the microservices application. This task required the creation of the aforementioned elements associated with infrastructure provisioning. This task required a compensation week, given that the development of the components within the environment caused the project to fall behind schedule. This delay would be compensated in the next task. For a more detailed analysis, please refer to the document titled Appendix D – Software Architecture and Design, under the section called Infrastructure Provisioning.

This task was verified by first creating multiple branches within the AWS CodeCommit repository and verifying the creation of the different AWS CodePipeline delivery elements associated with each branch. Each branch was duly updated with new infrastructure templates and performed an update on the existing infrastructure. Finally, each branch was merged into the main branch to visualize the deletion of the infrastructure associated with the original branch and the update of the main branch with the new configuration. These activities were further verified through a demonstration to the Product Owner and Scrum Master where the different elements were shown as created within the AWS console.

Task #3 – [04/04/2022 – 06/05/2022]:

- **Task #3.1 – [04/15/2022 – 15/04/2022]:** Once the infrastructure was in place, the CI/CD pipeline began taking shape, with the repository and pipeline creation taking place over two weeks. Here, the Continuous Integration phase was developed, creating the elements mentioned in the “Pre-Requisites” section of the document titled Appendix D – Software Architecture and Design, along with the automation process responsible for preparing the application's source code for delivery.

This sub task was verified by performing changes to the AWS CodeCommit repository to validate that the different components associated with the AWS CodePipeline worked correctly. As in the previous task, a new commit was made, new branches were created, merged and deleted to verify that the AWS Lambda function was performing the changes to the AWS CodePipeline projects accordingly. This functionality was presented to the Product Owner and the Scrum Master for confirmation and to guarantee that it functioned correctly.

- **Task #3.2 – [18/04/2022 – 29/04/2022]:** Having the initial pipeline in place, the tasks of automated building, static code analysis, dependency verification and push to the ECR Repository had to be defined. Here there was a small decision to be made regarding which tools would be used to perform these tasks. Once these tools were chosen, the pipeline was tested from end to end. A commit made to the AWS CodeCommit repository would trigger the AWS CodePipeline project associated with the branch that was changed. This project begins the AWS CodeBuild project and produces the final Docker Image. The changes made to the application were then verified in the final application.

Task #4 – [09/05/2022 – 27/05/2022]: Once the pipeline was in place to create the final executable Docker Image, Argo CD was deployed into the AWS EKS cluster, under its own namespace. This Continuous Delivery tool performs the progressive releases of the new image into the EKS Cluster, whilst analysing the metrics of the image to perform a rollback when necessary. This process was verified by performing multiple requests to the changed service and visualizing the two different versions returning different results, at the rate that the application load balancer would transfer the traffic between each version.

Task #5 – [30/05/2022 – 17/06/2022]: The final tools that were configured and deployed into the EKS Cluster were Prometheus and Grafana, responsible for metrics analysis and observability from a resource consumption point of view; and Elasticstack, which extracts the log stream from the various containers into a centralised location for querying and processing. The metrics were visualized using Grafana, where a dashboard was composed of multiple panels that made queries to the Prometheus instance running in the AWS EKS cluster.

The logs produced by the application were streamed directly to Kibana. To verify that the logs were being processed correctly, the application was coded in a way that an error message be printed every 15 seconds by the application. This error was printed in the Kibana GUI stream exactly as programmed.

Task #6 – [20/06/2022 – 01/07/2022]: finally, the last task carried out in the internship was the validation of the full framework from one end to another. This task is explained in further detail in the next section.

7.2 Validation Process

The validation process behind the internship consisted of evaluation tests which would follow the contents of the document Appendix E – Acceptance Test Plan and Report. This document defines the due dates for the components to be submitted to functional tests, guaranteeing that they perform correctly and according to their requirements. At the end of each User Epic, the framework was tested, using base states as a foundation on which to perform numerous test cases.

The components of the framework have a reserve of base states and test cases which must be complied with to guarantee that all requirements and functionalities that were implemented are fully operational, and function according to the desired of WIT Software. Base states are a specific snapshot of the framework after a number of tasks have been performed. These states will act as a foundation on which the different test cases will be applied, according to the requirements that are to be evaluated. Test Cases are a series of events that are communicated to the framework as inputs in order to understand if the intended outputs are obtained.

Base States

This section shows the different base states used during the validation process. Each base state is identified by its name, requirements, preconditions that must be met to reach the state in question, and the results which confirm that the current state of the framework is the target base state. In the document Appendix E – Acceptance Test Plan and Report, there is a more detailed explanation of the different steps required to achieve the result of the base state. For more information, please consult that document.

Table 16 shows a list of exemplars that represent how the Base States are constructed. Each exemplar is taken from each of the four elements of the framework. That is Infrastructure Provisioning, Continuous Integration, Continuous Delivery and Monitoring.

Table 16. Infrastructure Provisioning Base States

Name	Requirements	Preconditions	Results
Infrastructure Created State	Internet Access; Access to AWS Console.	BS-06: Infrastructure Templates Created State.	The AWS CloudFormation service console shows “CREATE_COMPLETE” state.
Code Committed State	A command line interface; Access to AWS Console; Internet Access.	BS-14; Git CLI client is installed and configured for the AWS CodeCommit Repository.	AWS CodePipeline status is “Successful” when build completes; AWS CodeBuild project is in “Successful” state when complete
Docker Container Image Pushed to AWS ECR State	Internet Connection; Internet Browser.	BS-15.	The new docker image is pushed in the AWS ECR repository and tagged accordingly.
Grafana Deployed State	The application is running in the AWS EKS Cluster; A Command Line Interface.	Helm deployment tool cli installed; BS-18: Prometheus Deployed to AWS EKS Cluster State.	The DevOps maintainer is successfully logged into the Grafana Server.

Test Cases

Each test case is identified by its name, the requirements that it looks to evaluate and validate, the preconditions that are required to perform the test, and finally the results that are expected by the end of the test. In the document Appendix E – Acceptance Test Plan and Report, there is a more detailed explanation of the different steps required to carry out the test. For more information, please consult that document.

The following Table 17 provides a shortlist of the test cases used to perform the validation process. The examples given each refer to a base case used in each of the four main elements

of the framework: Infrastructure Provisioning, Continuous Integration, Continuous Delivery and Monitoring.

Table 17. Infrastructure Provisioning Test Cases

Name	Requirements	Preconditions	Expected Results
A DevOps Team Leader Creates a New Branch	FR-06.	BS-05; BS-07.	The infrastructure existent in the root branch is replicated, with alterations to the name of the stack which now includes the “staging” branch name.
A Development Team Member performs a commit to a branch	FR-29.	BS-14.	The Docker Container Image produced by the AWS CodeBuild, associated with the AWS CodePipeline, is pushed to the AWS ECR.
Automated Rollout Strategy – Canary Release	FR-44.	BS-17.	The new version of the application replaces the previous version in its entirety.
Application Custom Metrics Visualization	FR-56.	BS-18; BS-19.	Grafana displays the dashboards associated with the custom metrics of the application.

Test Results and Incident Reports

The test cases conducted produced one of two possible results: passed without errors or failed. In the event that the test case passed without errors, no action was necessary, and the requirement was considered as successfully implemented. If a test were to fail, as we can see in Table 18 with Test Case 01 and Test Case 11, each failure required that a report be made of which test case failed, when the test was performed and by who, what incident occurred, and the actions required to correct the issue with the implementation. Once the correction was made, the framework would be re-submitted to the same test case and would be required to pass this repeated test. If the test were passed, the developer would have to report when the change was implemented, by who, and what correction was made that caused the framework to comply with the tests made.

Table 18. Example Test Results

Test Case	Date	Tester	P/F	Comments
TC-01	28/03/2022	GP	F	Failed
TC-10	31/03/2022	GP	P	Passed, without errors.
TC-11	31/03/2022	GP	F	Failed.

The following table 19 and table 20 serve as two examples for the test reports that were made when a failure occurred during the validation process.

Table 19. Failed Test Report for TC-01

Test	TC-01: A DevOps Team Leader Creates the Infrastructure Pre-Requisites
Date; Tester	28/03/2022; Gabriel Pinheiro
Incident	Permissions were missing in the DevOps Team Leader IAM Role.
Actions	Include the permissions required for the DevOps Team Leader to be able to create the pre-requisites for the infrastructure pipeline.
Date; Developer	04/04/2022; Gabriel Pinheiro
Correction	The DevOps Team Leader IAM Role was corrected.

Table 20. Failed Test Report for TC-11

Test	TC-11: SSH Connection from a Bastion Host to a Node in the EKS Cluster
Date; Tester	28/03/2022; Gabriel Pinheiro
Incident	It was not possible to perform an SSH connection between a Bastion Host and a Node in the EKS Cluster. The connection reaches a timeout.
Actions	Correct Security Group ingress settings to include connections from the Bastion Host Security Group.
Date; Developer	04/04/2022
Correction	The EKS Cluster Security Group was updated to allow SSH connections from the Bastion Host Security Group.

Chapter 8

Conclusion

The final chapter of this report marks the end of the document and the end of the internship carried out at WIT Software. This document is divided into two main sections. The first being the presentation of an overview of all the work that was done during the internship. The final section explains the limitation of the framework in its current state, along with the work that could still be done in future to improve the functionalities offered by the product.

8.1 Project Overview

The internship in its entirety had the objective of providing a means of accelerating the process behind developing software. Time-to-market is a fundamental aspect to control when trying to maintain a competitive edge in the software development process. This aspect can be shortened beginning with the creation of infrastructure, all the way through to the deployment time required when making a new version available to the application's users. To achieve this, WIT Software prepared this internship to create a framework which would help developers in creating software in an efficient manner, whilst maintaining the quality with which the organization already creates its applications.

The internship's results focus on three main areas of impact, developed to improve the time-to-market of the software. These three areas are:

- Infrastructure Provisioning;
- Continuous Integration/Continuous Delivery;
- Monitoring.

Infrastructure Provisioning

The first phase which is impacted by the use of this framework is the creation of an infrastructure for an application to run on. In the traditional process of requiring resources from an IT team, on-premises, the amount of time necessary to have access to computational resources could set a project's beginning back a few days. The IT team would create a virtual machine with the resources that were required, prepare the endpoints and user access control, and finally deliver the VM to the development team. This process is a reality which WIT looks to shorten considerably. Furthermore, when needing to scale the resources of the virtual machine, the previous version would have to be destroyed and a new version would have to be re-allocated, repeating the time-consuming process from the beginning.

With this framework in hand, making use of two main features involved with Cloud Computing, Infrastructure as a Service and Infrastructure as Code, the process of creating and infrastructure, along with adjusting and updating its features is improved. In the past, when it would take days to create an infrastructure, now it takes minutes. The infrastructure mentioned in chapter 6 - Framework Architecture takes roughly 30 minutes to be created, including the pre-requisites. When an improvement or change needs to be applied, all that needs to be done is a change in the configuration templates, and the elements affected are updated accordingly, without the need for recreation from scratch. The update process duration is dependent on the amount of components that need to be updated, along with which components are being updated.

This component of the framework proposes a standard for Infrastructure as Code within the organization. This standard begins with the architecture being developed using YAML templates, versioned and controlled by a git repository hosted in AWS CodeCommit. These templates provide the specification for the infrastructure to be provisioned, automating their creation with AWS CodePipeline which establishes the link between the DevOps maintainers who have access to the AWS CodeCommit repository and the AWS CloudFormation service which creates the computational resources.

CI/CD

The second phase of the framework offers an acceleration of the time-to-market, but from the point of view of the developers of the application's source code. Looking into the pillars of the DevSecOps culture, developers should, ideally, perform small commits of code on a daily basis. This means that the changes made to the code are of small impact and allow for continuous testing before the new version is deployed, along with the continuous delivery of those changes into the production environment. In other words, the process behind creating changes and delivering new versions into production is made daily, continuously and autonomously.

The Continuous Integration phase allows automated tests to be conducted on the source code of the application, without the tests needing redefinitions whenever there is a need to repeat them on the source code. The tests are defined once and replicated, without requiring manual intervention. Furthermore, the code itself can be subject to a consistent code analysis, which guarantees that the code itself is robust, easy to read, and that when passing from one developer to another, all coding conventions are maintained. Finally, from a security point of view, the application's code is analysed to make sure that external libraries do not compromise the confidential information of the application. Given that the application suffers small alterations over time, along with automated static tests, the deployment of new versions can be carried out progressively and continuously with the framework. In the Continuous Delivery phase, the possibility of deploying a new version of the application as soon as the CI phase is complete and a new version of the microservice is pushed to the repository becomes a reality.

The CI/CD pipeline establishes a standard for software development teams, applying the best practices of maintaining the software source code in a git repository. This git repository is the single point of truth for the automation of the entire automated sequence of events. The contents of the files are exported by the AWS CodePipeline tool, which retrieves the files from the AWS CodeCommit repository and communicated them to the AWS CodeBuild automation tool. The AWS CodeBuild tool is linked to the Argo CD instance which then performs the rollout according to its configuration file. The standard implemented defines which solutions are to be used when looking to adopt a DevSecOps culture within a development process, along with the organization within a project's repository.

Monitoring

Observability and insight into an application's behaviour in production is essential when dealing with a microservice architectural pattern. This framework allows operations team members to acquire information regarding the application's behaviour within its production environment. Constant improvement is a must when developing software and understanding where to apply changes which best suit and simultaneously look to better the efficiency of the various services begins with providing access to all the information belonging to an application.

The key to mapping out what needs to be changed and how to change it is provided by this framework from two fundamental sources of information: metrics analysis and log analysis.

Metrics analysis is a possibility by combining Prometheus, a metrics scraping tool which performs auto-discovery on the various metrics made available withing a Kubernetes Cluster, with Grafana, a visualization tool which centralizes multiple point of information into one dashboard. Logging analysis is made possible thanks to the implementation of the Elasticstack component. Conjugating the Filebeat service, which extracts logs from the application's STDOUT stream, with Elasticsearch, tool used to centralize the information extracted by Filebeat, and finally with Kibana, which offers a visualization medium through which to view and query the logs of the application.

A standard is presented to the organization by using open-source tools, exclusively. This solution offers a reduction of costs toward monitoring an application, whilst maintaining the same level of insights and access to information which is offered by cloud-hosted services.

Second Semester Overview

In my opinion, the second semester was much more fruitful than the first. The problems that the internship looked to face, the motivation behind it, and the goal which it looked to achieve were still not completely transparent to me at the end of the January. Although some work had been concluded and carried out, the feedback given and my own feelings toward what had been done reflected some confusion and a sense of feeling that some lights were being shone, but the way was not defined. The second semester reflected the complete opposite to this. In the second semester, after having a more practical experience with the different problems that internship brought into scope, after creating a detailed plan of what needed to be implemented, and after defining the objectives that needed to be met by the end of the semester, the process of both learning and implementing the framework became much clearer to me.

Beginning with the infrastructure provisioning phase of the internship, the lack of practical experience regarding creating full-scaled infrastructures was most felt. The experimentation that was required in the first semester was, to an extent, compensated in this section, where I was required to study and understand the different components that are required to create an infrastructure for a microservices application. The need for creating a model which is highly available across multiple Availability Zones, whilst maintaining a clear division between public and private resources and guaranteeing security measures between them caused a small delay in the project's planning. A week of testing and possible changes was kept at the end of each epic pertaining to the project. In this phase, that same week was used to compensate the delay in creating the infrastructure, but then compensated in the Continuous Integration phase, bringing the project back on track.

The creation of the CI/CD pipeline was not problematic. Given some previous experience with the different tools involved in provisioning an AWS CodePipeline service tool in two subjects during the master's degree, the process of creating this component was successful and went according to the projects planning. Although, when creating the CD component, I did require assistance from the Scrum Master, given that the documentation given by Argo CD is somewhat limited for some use cases.

Finally, the monitoring phase of the framework was also successful, without causing any delays. I was requested by the Product Owner to include a specific requirement which was the live streaming and analysis of the application's log from the STDOUT stream of each of the microservices. This change did not cause any interruptions to the work plan, and therefore considered a success given that the component was provisioned on time.

The following Figure 30 represents a Gantt diagram of the second semester which, when compared with Figure 13 of the planned project, shows some differences, particularly due to the delay that occurred in the Infrastructure Provisioning phase.

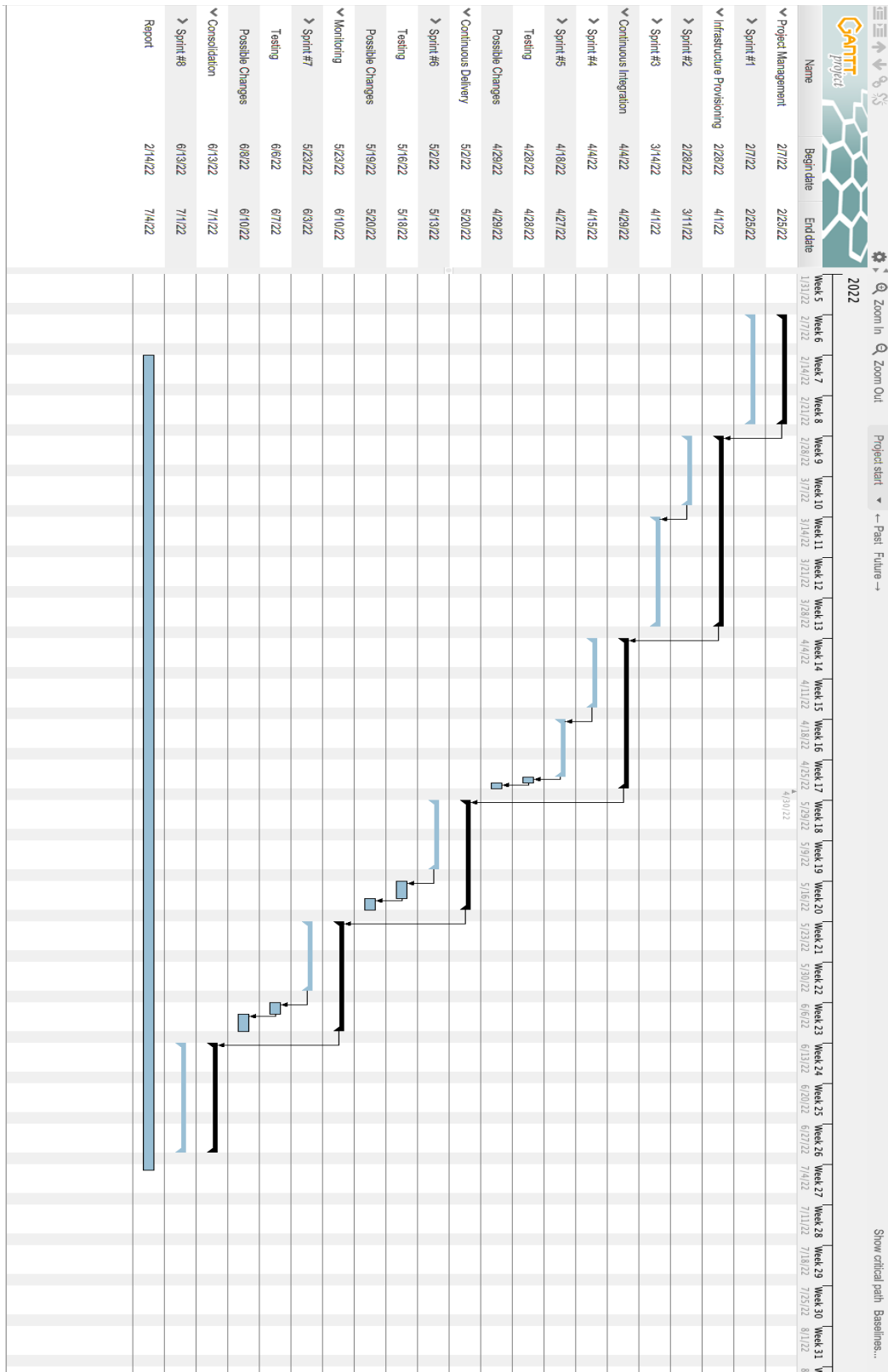


Figure 30. Second Semester Gantt Diagram Reality

8.2 Limitations and Future Work

Infrastructure Provisioning

The components which are provisioned in the infrastructure have one limitation. The EKS Cluster nodes are required to be deployed into a private EKS Subnet, but do not attach to the cluster if this is the case. Although all security measures are put in place for the instances not to be vulnerable to known attacks, they would need to be deployed into a private subnet to guarantee an application of the best practices of creating a Kubernetes cluster hosted on a cloud-native infrastructure.

As future work, the previously mentioned limitation would be corrected, without a workaround but with a concrete standard by which it would be possible to expand the infrastructure to other projects. The infrastructure would be improved to include an RDS Application to manage the database that would be shared amongst the different microservices, as a means of having a centralized point of access for shared information. Furthermore, the resources that are created would also be extended to a third availability zone, to guarantee scalability and availability for different access points.

Finally, the possibility of spanning multiple regions would also be studied to understand how replication could be achieved when looking to expand the application's coverage zones.

CICD

The CI/CD pipeline presents one main flaw in its functionalities, and that is the fact that the source code files must all be kept within the root directory of the repository. This limitation is due to a trade-off of using an AWS Lambda function to extract source code files from the AWS CodeCommit repository to the AWS S3 Bucket, which is the source for the AWS CodePipeline component. Therefore, the CI/CD pipeline is functional as a proof of concept that it is possible to perform the automated CI/CD process, but it requires a correction when conducting the CI phase.

For future work, this use case would be studied and included into the pipeline's functionality, given that project organization, more time than not, requires multiple directories. Once corrected, the framework would then be adapted and introduced into a functioning project as a means of receiving feedback on its behaviour. Initially on a trial basis to guarantee that there are no issues caused by its adoption. The errors that arise would be addressed and corrected to improve the pipeline's possibilities.

Monitoring

From a monitoring point of view, and regarding the requirements that were included in the project, the monitoring component does not offer any limitations as it stands.

For future work, alternatives for the implemented components would be studied to understand if more efficient substitutes exist, and to what extent the ones implemented can also be improved. The Scrum Master took the liberty of providing an alternative for the Filebeat tool by using Fluentbit. This tool is a more lightweight implementation of a file analysis tool, tailored specifically for Kubernetes and microservices. Its implementation would be included in the framework's future improvements.

Chapter 9

References

- [1] N. Solomon, “Global Cloud-Native Adoption Continues to Rise,” 20 05 2022. [Online]. Available: <https://containerjournal.com/features/global-cloud-native-adoption-continues-to-rise/>.
- [2] J. Orme, “Why DevSecOps is key to future-proofing your software development lifecycle,” CloserStill Media Ltd, 20 04 2020. [Online]. Available: <https://www.techerati.com/features-hub/opinions/why-devsecops-is-key-to-future-proofing-your-software-development-lifecycle/>.
- [3] K. Marko, “Follow 6 key steps to deploy microservices in production,” 15 09 2021. [Online]. Available: <https://www.techtarget.com/searchitoperations/tip/Follow-these-6-steps-to-deploy-microservices-in-production>.
- [4] harness.io, “Intro To deployment Strategies: Blue-Green, Canary, And More,” 23 2 2022. [Online]. Available: <https://harness.io/blog/continuous-verification/blue-green-canary-deployment-strategies/>.
- [5] National Institute of Standards and Technology, The NIST Definition of Cloud Computing, Gaithersburg: U.S. Department of Commerce, 2011.
- [6] Unknown, “PAAS AND SAAS – WHAT YOU NEED TO KNOW,” Clarus Financial Technology, 26 March 2019. [Online]. Available: <https://www.clarusft.com/paas-and-saas-what-you-need-to-know/>. [Accessed 22 March 2022].
- [7] M. Fowler and J. Lewis, “Microservices,” 25 March 2014. [Online]. Available: <https://martinfowler.com/articles/microservices.html>.
- [8] VMWare, “Security in Kubernetes,” VMWare, Palo Alto, 2019.
- [9] VM Ware, “What is a virtual machine?,” VMware, 2022. [Online]. Available: <https://www.vmware.com/topics/glossary/content/virtual-machine.html>.
- [10] D. Inc.. [Online]. Available: <https://www.docker.com/resources/what-container/>.
] [Accessed 22 03 2022].
- [11] i. Red Hat, “Understandig DevOps,” 19 April 2018. [Online]. Available: <https://www.redhat.com/en/topics/devops>.
- [12] R. Wilsenach, “DevOpsCulture,” 9 July 2016. [Online]. Available: <https://martinfowler.com/bliki/DevOpsCulture.html>.
- [13] K. Carter, “Francois Raynaud on DevSecOps,” IEEE Software, 2017.
]
- [14] C. B. Pereira, “Bem vindo ao projeto Segurança na pipeline CI/CD - DevSecOps,” 2021.] [Online]. Available: <https://cassideveloper.com.br/devsecops/>.

- [15 Red Hat, inc, “What is CI/CD?,” 31 January 2018. [Online]. Available:
] <https://www.redhat.com/en/topics/devops/what-is-ci-cd>.
- [16 C. C. Acarer, 4 10 2020. [Online]. Available: <https://cacarer.com/ci-cd-continuous-integration-continuous-delivery-and-continuous-deployment/>. [Accessed 22 03 2021].
- [17 humio, “What is Application Monitoring?,” 30 September 2021. [Online]. Available:
] [https://www.humio.com/glossary/application-monitoring/#:~:text=Application%20monitoring%20is%20the%20process,%2Duser%20experience%20\(UX\)..](https://www.humio.com/glossary/application-monitoring/#:~:text=Application%20monitoring%20is%20the%20process,%2Duser%20experience%20(UX)..)
- [18 J. Turnbull, Monitoring with Prometheus, Creative Commons, 2018.
]
- [19 B. Brazil, Prometheus: Up & Running, Sebastopol: O'Reilly Media, Inc, 2018.
]
- [20 Intraway, “Intraway,” 14 July 2021. [Online]. Available:
] <https://www.intraway.com/blog/what-is-infrastructure-provisioning/>.
- [21 A. Inc., “est practices for developing and deploying cloud infrastructure with the AWS
] CDK,” Amazon Inc., [Online]. Available:
] <https://docs.aws.amazon.com/cdk/v2/guide/best-practices.html>. [Accessed 16 05 2022].
- [22 Terraform, “Terraform,” January 2017. [Online]. Available:
] <https://github.com/hashicorp/terraform>.
- [23 Amazon, “What is AWS CloudFormation?,” [Online]. Available:
] <https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/Welcome.html>.
- [24 JetBrains s.r.o, “CI/CD BestPractices,” JetBrains s.r.o, [Online]. Available:
] <https://www.jetbrains.com/teamcity/ci-cd-guide/ci-cd-best-practices/>. [Accessed 15 05 2022].
- [25 J. F. Smart, Jenkins - The Dfinitive Guide, Sebastapol: O'Reilly Media, inc, 2011.
]
- [26 Amazon, “AWS CodePipeline,” [Online]. Available:
] <https://aws.amazon.com/codepipeline/>.
- [27 GitLab Inc, “GitLab CI/CD,” [Online]. Available: <https://docs.gitlab.com/ee/ci/>.
] [Accessed 22 03 2022].
- [28 Harness Author, “Intro To Deployment Strategies: Blue-Green, Canary And More,”
] harness, 20 October 2021. [Online]. Available: <https://harness.io/blog/blue-green-canary-deployment-strategies/>.
- [29 D. Bryant, “A Comprehensive Guide To Canary releases,” Ambassador Labs, 20
] September 2018. [Online]. Available: <https://blog.getambassador.io/cloud-native-patterns-canary-release-1cb8f82d371a>.
- [30 GoCD, “Free & Open Source CI/CD Server,” [Online]. Available: <https://www.gocd.org/>.
]

- [31 Flux, “Flux Documentation,” [Online]. Available: <https://fluxcd.io/docs/>.
]
- [32 Argo CD, “Overview,” [Online]. Available: <https://argo-cd.readthedocs.io/en/stable/>.
]
- [33 Amazon, “Amazon CloudWatch,” [Online]. Available:
] <https://aws.amazon.com/cloudwatch/>.
- [34 Amazon, “The ELK Stack,” [Online]. Available: [https://aws.amazon.com/opensearch-
\] service/the-elk-stack/](https://aws.amazon.com/opensearch-service/the-elk-stack/).
- [35 Prometheus Authors, “From metrics to insight,” 2014. [Online]. Available:
] <https://prometheus.io/docs/introduction/overview/>.
- [36 Grafana Labs, “Get started or start exploring Grafana,” 2014. [Online]. Available:
] <https://grafana.com/docs/>.
- [37 S. Obrutsky, “Comparison and contrast of project management methodologies PMBOK
] and SCRUM,” ResearchGate, New Zealand, 2016.
- [38 R. Stacey, Strategic management and organisational dynamics: The challenge of
] Complexity (5th Edition), Essex: Parson Education Limited, 2007.
- [39 pm-partners, “The Agile Journey: A Scrum Overview,” PM-Partners Group, 23 June 2021.
] [Online]. Available: [https://www.pm-partners.com.au/the-agile-journey-a-scrum-
overview/](https://www.pm-partners.com.au/the-agile-journey-a-scrum-overview/). [Accessed 24 01 2022].
- [40 ProductPlan, “MoSCoW Prioritization,” [Online]. Available:
] <https://www.productplan.com/glossary/moscow-prioritization/>. [Accessed 17 05 2022].
- [41 AcqNotes LLC, “Requirement Types,” AcqNotes LLC, [Online]. Available:
] <https://acqnotes.com/acqnote/tasks/requirement-types>. [Accessed 16 05 2022].
- [42 Amazon Web Services, Inc., “Choosing regions and availability zones,” Amazon Web
] Services, Inc., 2022. [Online]. Available:
[https://docs.aws.amazon.com/AmazonElasticCache/latest/mem-
ug/RegionsAndAZs.html#CacheNode.Memcached.AvailabilityZones](https://docs.aws.amazon.com/AmazonElasticCache/latest/mem-ug/RegionsAndAZs.html#CacheNode.Memcached.AvailabilityZones).
- [43 Amazon Web Services, Inc, “Subnets for your VPC,” Amazon Web Services, Inc, 2022.
] [Online]. Available: [https://docs.aws.amazon.com/vpc/latest/userguide/configure-
subnets.html#subnet-basics](https://docs.aws.amazon.com/vpc/latest/userguide/configure-subnets.html#subnet-basics).
- [44 Amazon Web Service, Inc., “NAT gateways,” Amazon Web Service, Inc., 2022. [Online].
] Available: <https://docs.aws.amazon.com/vpc/latest/userguide/vpc-nat-gateway.html>.
- [45 Amazon Web Service, Inc., “What is Amazon EKS?,” Amazon Web Service, Inc., 2022.
] [Online]. Available: <https://docs.aws.amazon.com/eks/latest/userguide/what-is-eks.html>.
- [46 Amazon Web Services, Inc., “AWS WAF - Web Application Firewall,” Amazon Web
] Services, Inc., 2022. [Online]. Available: <https://aws.amazon.com/waf/>.
- [47 Amazon Web Services, Inc., “Amazon Route 53,” Amazon Web Services, Inc.0, 2022.
] [Online]. Available: <https://aws.amazon.com/route53/>.

- [48 JavaScript Lint , “JavaScript Lint,” JavaScript Lint , [Online]. Available:
] <https://www.javascriptlint.com>.
- [49 OWASP Foundation, Inc., “OWASP Dependency-Check,” OWASP Foundation, Inc.,
] 2022. [Online]. Available: <https://owasp.org/www-project-dependency-check/>.
- [50 Argo CD, “Argo Rollouts - Kubernetes Progressive Delivery Controller,” Argo CD,
] 2022. [Online]. Available: <https://argoproj.github.io/argo-rollouts/>.
- [51 The Kubernetes Authors, “Deployments,” The Linux Foundation, 02 02 2022. [Online].
] Available: <https://kubernetes.io/docs/concepts/workloads/controllers/deployment/>.
- [52 Elasticsearch B.V., “Filebeat overview,” Elasticsearch B.V., 2022. [Online]. Available:
] <https://www.elastic.co/guide/en/beats/filebeat/current/filebeat-overview.html>.
- [53 z. Mahmood, Cloud Computing: Characteristics and Deployment Approaches, Derby:
] University of Derby, 2011.
- [54 C. Richardson, “Pattern: Monolithic Architecture,” 2019. [Online]. Available:
] <https://microservices.io/patterns/monolithic.html>.
- [55 International Trade Administration, “Information and Communications Technology,” 2
] October 2021. [Online]. Available: <https://www.trade.gov/country-commercial-guides/portugal-information-and-communications-technology>.
- [56 A. W. Services. [Online]. Available: <https://aws.amazon.com/pt/types-of-cloud-computing/>. [Accessed 22 03 2022].
- [57 Argo CD, “Overview,” Argo CD, 2022. [Online]. Available: <https://argo-cd.readthedocs.io/en/stable/>.
- [58 Argo CD, “Architectural Overview,” Argo CD, 2022. [Online]. Available: <https://argo-cd.readthedocs.io/en/stable/operator-manual/architecture/>.

Appendixes

Appendix A – State of the Art



FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE
COIMBRA
DEPARTMENT OF INFORMATICS ENGINEERING

Gabriel Marco Freire Pinheiro

Appendix A – State of the Art

Dissertation in the context of the Master's Degree in Informatics Engineering, Specialization in Software Engineering, advised by Professor Nuno Laranjeiro and Engenheiro Rui Cunha, presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

July 2022

This page was intentionally left in blank.

Cloud Computing

Cloud computing is the term that refers to computing services that are offered by a provider, outside their geographical location, through the use of a distributed system over an internet connection. In this paradigm, the user of the cloud service can make use of the provider's infrastructure, ranging from storage, computing capacity or environment deployment without having to consider the investment, instalment and preparation of a full physical infrastructure. The provider, in most cases, follows a 'pay-as-you-go' model, where the user is billed according to the resources that they use and how much they use them. This model opens a door of many benefits regarding how a business' computing infrastructure can be abstracted, turning the focus scope onto business logic, rather than capital costs. But, as all things, there are always advantages and disadvantages to these systems.

Characteristics of Cloud Computing

In this sub-chapter, cloud computing will be analysed in a more technical and in-depth manner, considering the various characteristics of Cloud Computing.

The definition of cloud computing architecture can be summarized into *"multiple cloud components communicating with each other over a loose coupling mechanism such as a messaging queue"*. The following Figure 1 represents a simplified model of how a *cloud-native* system can be orchestrated and architected. Considering the main characteristics of cloud systems, elasticity can be incorporated into a system, requiring intelligence regarding the use of tight or loose coupling of mechanisms within the cloud's structure.

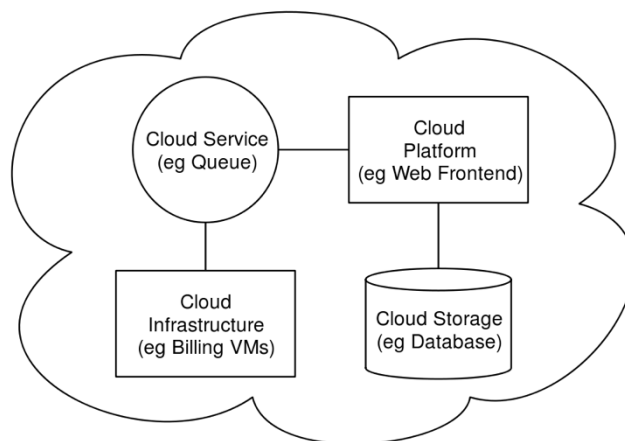


Figure 1. Cloud Computing Sample Architecture [1]

The characteristics that best define Cloud Computing are:

- Agility;
- Device and Location Independence;
- Cost reduction;
- Multitenancy;
- Performance;
- Productivity;
- Redundancy;
- Security.

As aforementioned, cloud computing allows for a reduction in costs, from the point of view of capital investments, but this is not the only characteristic that brings benefits from adopting a cloud-based architecture for development.

The concept of device and location independence is another characteristic brought forward by cloud computing. Considering that access to the infrastructure is made available via the internet, users of the service can access it from any location and from any device that has an internet connection coupled with an internet browser.

Multitenancy is a characteristic belonging to cloud computing that causes some controversy, as both optimization and security requirements are brought into the same conversation. Multitenancy, in short, is the term applied to the sharing of various resources and costs across a larger pool of users. This invites a more optimized use of resources to become a reality, as the same system can be used by multiple users, simultaneously, making available what is strictly necessary according to load. Coupled to this, multitenancy brings into view a more centralized infrastructure, meaning that costs associated with real estate, electricity, amongst other operational costs, are reduced and shared amongst the cloud's users. From a safety point of view, this form of computation opens the possibility to information leaks and open access to sensitive data, if the information is not regulated accordingly. Obviously, this disadvantage only becomes a reality if the system is not correctly configured and regulated. As simple as this may seem, inexperienced and clients with less technological knowledge are more susceptible to gaps in their security protocol.

A trade-off characteristic of this paradigm is that of infrastructure flexibility. Due to the fact that the cloud's computing possibilities are supplied by a cloud provider, all management, installation and maintenance is of the provider's responsibility, hence the reduction regarding capital investment into this infrastructure. And, as the provider is in charge of all the infrastructure, a larger flexibility regarding the addition, expansion and re-provision of resources also falls under the provider's scope of responsibilities, once again reducing the costs associated to these functionalities.

Coupled with agility comes the possibility of an increase in productivity. As aforementioned, cloud architectures are based on a distributed system paradigm where users can access and interact with information from any device with an internet connection and web browser. Allied with this characteristic is the ability of multiple users accessing the same data simultaneously, without requiring a hand-off culture where a product is given over to another user only once the previous has concluded all work required by their department. Naturally, this reduces delays between tasks and increases the amount of work done within the same amount of time. Considering competitive

markets and the necessity for a shorter time-to-market delivery, productivity is essential for a corporation's success.

Redundancy, which in general terms is seen as an expendable feature within a product, is a benefit brought forward by the adoption of cloud computing. Redundancy, in its definition, is the state of not being necessary or useful. When regarding engineering, redundancy translates into a fail-safe, in other words, the inclusion of components that perform the same tasks as others to compensate situations in which a component experiences some sort of a failure and the system's availability is impacted. Cloud computing offers the possibility for redundancy inclusion, acting as a safeguard for these failures.

Within the scope of cloud computing, there are four levels of redundancy that help facilitate readiness for adaptations to these situations.

The first, hardware redundancy, takes into account how faults are tolerated on the machines themselves. How these faults are managed depends on how a user opts for their cloud services. That is, if they are private or public. Privately owned hardware requires that all devices must be configured and orchestrated entirely by the user. Public cloud provisioning is when all design, building and maintenance of environment is trusted to a cloud provider (such as Amazon, Google, IBM, Microsoft, etc). Regardless of the choice, both solutions guarantee that the service is always accessible, and faults are mitigated.

Secondly, process redundancy must be taken into account. In a digital system, a business requires processes to be working to achieve their intended purpose. Some processes may be more critical than others, and a decision must be made to establish which have a higher level of importance over others which are expendable. This opens way to a tempting solution which is that all processes are essential, and all require the same level of availability. Although this may be the easier solution for the problem, it is also the one with the potential for cost escalation. To build a system which is both cost-effective, efficient and that meets the necessary levels of availability, a business must analyse and evaluate each process so as to determine the appropriate level of service reliability accordingly.

Network redundancy is a motive for reflection on a system together with those mentioned previously. The most concrete example for the importance and functionality of this is that of the internet. When one internet carrier has a failure, are there any to compensate the gap? The same train of thought must be followed when considering network redundancy importance in a service. Together with the fact that, all systems that follow a cloud-based architecture require a network connection. If the network fails, and there is nothing to compensate this failure, all the downtime translates into a loss.

Finally, geographic redundancy. The concrete implementation of anywhere, anytime, anyhow. This form of redundancy means that information that is essential for operations is replicated between two or more physically disperse locations. This is a fail-safe that takes into account a more nature-based form of failure. Natural disasters, as regardless of where an enterprise is located, are always a possibility, and when preparing a service, the possibility of these occurring must also be taken into account. Having multiple data centres with operation information is the solution that must be considered and implemented when cloud computing is a potential solution for business. Privately

implemented systems require a more in-depth approach to how the logistics are approached, whilst publicly shared systems generally cover this situation independently. Albeit, some manual configuration may still be necessary, depending on the chosen provider.

Lastly, regarding the characteristics attained by cloud computing, coupled with the need for availability comes the need for scalability and elasticity when supporting a stronger load of requests. Considering how cloud architecture has a fine-grained implementation for resource provision, the system inherits the ability to scale up and down according to the amount of stress that is being applied to resources. Cloud computing offers a very time-efficient form of scaling, which translates into a faster time-to-market, larger flexibility and adaptability to a business due to the fact that resource addition and removal is much more efficient and performed on a much larger speed.

Service Models

Although service-orientated architecture, in its essence, offers the possibility of supplying “Everything as a Service”, cloud computing focuses service delivery according to three more specific delivery models. These are Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). These three models offer different levels of abstraction to their users and are usually represented as *layers* within a stack of software subsystems. Although associated to one another, their relation is extraneous. For instance, SaaS can be provided directly on bare metal machines, without the need for an IaaS as an intermediary, just as a program can be run on an IaaS, without the need for a SaaS Service wrapping.

Infrastructure as a Service (IaaS)

When looking to abstract from low-level network infrastructure such as physical computer components and resources, data partitioning, application scaling and management, and all features associated with this, Infrastructure as a Service is a form of delivering high-level APIs that provide an abstraction layer to these details. A hypervisor is responsible for running instances of virtual machines as guests, whereas in cloud computing, these are grouped together in pools so as to support an increased number of VMs combined with the ability of scaling up and down according to the load that the user will require.

The National Institute of Standards and Technology define IaaS as a system “*where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications. The consumer does not manage or control the underlying cloud infrastructure but has control over operating systems, storage, and deployed applications; and possibly limited control of select networking components (e.g., host firewalls).*” – (*The NIST Definition of Cloud Computing*)

Cloud providers offer access to these resources from large clusters equipment that is installed in multiple data centres, having users access these via either the internet using an internet browser as a point of access, or via a VPN which is usually associated with private cloud implementations.

IaaS looks to deliver a cloud computing infrastructure, as stated before. An infrastructure which includes servers, a full network solution, operating systems and storage solutions are all constituents of a cloud provider offering an infrastructure solution. The best examples of these, and also leading market share holders, are Amazon's AWS, IBM Cloud, Microsoft Azure and Google Cloud.

As all things in technology, and generally in nature, there are counterparts to an IaaS delivery system. The first, as associated to all things *cloud-native*, is that of security. "Is my information safe? Am I and the provider the only ones who can access it?" These are the general questions asked surrounding cloud computing, and IaaS does not go unnoticed. Insider threats or system vulnerabilities may give way to data exposure between the host and Virtual Machines, handing vital and sensitive information to unauthorized entities. Regarding the second question, cloud providers must find a way of guaranteeing that other customers cannot access information belonging to previous users, as is the need for VMs to be adequately isolated within the cloud's multitenant architecture model.

In summary, IaaS is a viable solution for many enterprises that require computational infrastructure on demand and that are not willing to invest in resources and manpower to manage, orchestrate and maintain those same resources. Be it a small start-up company, one experiencing rapid growth, or a large household name, IaaS offers the flexibility and scalability that any new application demands.

Platform as a Service (PaaS)

When developing an application but not wanting to have to manage or control networking, a full-blown operating system and/or storage elements, PaaS offers full control over the deployed applications, together with environment configuration settings.

Cloud providers who offer a PaaS solution deliver a framework for developers to build upon and create customized applications, whilst having servers, storage and networking being managed by the provider, maintaining the developer's management over the application that is being developed. These features are offered over a platform for software creation, accessed via a web application, giving freedom to software construction without the hassle of operating system configuration and development, software updates, storage, or the infrastructure behind all of these tools.

In its essence, PaaS allows for middleware development, with an increased level of scalability and availability to software development. Given the abstraction from all lower-level computational requirements, a more cost-effective model is also brought into play regarding application development and deployment, together with relieving the need for software maintenance when customizing the application accordingly. All of these characteristics, coupled with automation and a significant reduction regarding coding quota, an application being run on a PaaS solution allows for a much more dynamic outlook on how an application is developed, also allowing for a more rapid development and deployment model.

But, as always, PaaS also raises some concerns and limitations. Once again, as is with cloud systems offered by a third-party provider, data security is the first preoccupation regarding a service model. As much as PaaS users run and control their own applications, all the information that resides in the provider's cloud server poses a risk for security

breaches and infiltrations. This also limits options as the vendor's implementation of the service defines how the application is deployed, bringing into play the possibility of restrictions imposed by non-specific hosting policies.

Software as a Service (SaaS)

The SaaS model provides access to application software directly through a web application on a browser. The infrastructure and platform elements of the software are controlled by the cloud provider, usually following a pay-per-use pricing basis or via a subscription fee. All the software that operates the application is installed on the cloud servers, being accessed by users via a cloud client, being that users do not have any worries regarding software installation and effectively running the application, as it is running on the providers servers, being available in an on-demand format.

Analysing from a technical point of view SaaS applications operate in a different manner to those of other models, being that scalability is achieved by cloning tasks onto multiple VMs at runtime, orchestrating the system to achieve the work demand. Load balancers then distribute the workload over the attributed machines, making the process completely transparent to the cloud user, being that the only view seen is that of the access-point. Multitenancy is also a possibility in this model, meaning that any machine can serve multiple users simultaneously.

SaaS raises a number of concerns and limitation with its use, naturally. The main setback being that of interoperability. If an application following this model is not prepared for third-party application integration, organizations must adapt and design their own systems to cooperate with cloud software applications, or reduce their dependencies, amounting for larger costs in operations or a complete re-work of operations. With a lack of integration possibly being resolved, many of times a lack of integration support comes added onto the use of SaaS applications. Many enterprises make use of on-premise applications and infrastructure that SaaS providers may offer little or no support for, leading to a larger cost having to adapt and take into account the complexity of integration and limit how other dependant services are used. More so, a lack of control over the application's features and functionalities implies a redefinition of user's data security and governance merely to take into account the SaaS application. This leads to a limitation in terms of customization since a one-size-fits-all solution doesn't exist. Users many a time are bound to the specific functionalities, performance peaks and integrations provided by the vendor. Finally, as in all service models, data security is one of the major concerns. Large amounts of data are exchanged in the backend data centres in order to perform the necessary functions and functionalities offered by a SaaS application. This may lead to security compromission and compliance if sensitive business information is being transferred on public cloud-based software.

Ultimately, SaaS is most beneficial in situations such as those of start-up and small companies that have the need to launch ecommerce rapidly and don't dispose of the necessary time involved with server issues and software development. Short-term projects can also benefit from this service model as it offers quick, easy and affordable collaboration for its users. Applications with low traffic levels and that require both web and mobile access also take advantage of the characteristics offered by SaaS model applications.

Deployment Models

When adopting a cloud infrastructure, there are three main forms of deployment: private, public and hybrid. There are more, but these are less common and will not be studied within the scope of this project. All of these deployment models follow the point of view as to how they impact business operations, and in which cases their benefits outweigh their limitations and concerns. Let's dive into it.

Private Cloud

A private cloud, as the name suggests, is a cloud infrastructure in which a single organization operates and makes use of, having exclusive access to all the infrastructure, resources and support for the system. Although mostly hosted internally, an external solution can be adopted, together with the option of internal management or allocating this responsibility to a third-party provider. [2]

Undertaking a cloud project comes with the need for a lot of involvement regarding virtualizing the business environment, together with re-evaluating and organizing existing resources and to what extent these need to be changed. Once all of this analysis is complete, business has room for improvement, yet every step taken into adopting this model requires security issues that must be addressed so as to avoid vulnerabilities and unauthorized data access, but when implemented with the correct protocols and policies, a private cloud can bring a more robust business model which influences how the surrounding processes work.

Many a time, when going cloud, the thought of an exclusive cloud solution where all the resources that exist on the system are made available for a single organization, where the system can either be managed by the organization internally or by a third-party. All in all, this model seems like the ideal solution, yet adopting a private cloud project brings a list of requirements that must be taken into account.

Firstly, and above all else, virtualization of the business environment must receive the most attention when bringing a private cloud model into an organization. From the point of view of a start-up or a smaller business, deciding how the different systems and services are going to be made available via cloud can be time consuming and will occupy resources; for larger businesses which have extensive operations, re-evaluating which existing resources work and have to be changed both consumes time and also resources to make the leap onto the cloud.

Secondly, when developing a private cloud model, security is one of the more important aspects of development. The aforementioned characteristics all look to improve business, of course, but in every step of developing a self-run datacentre raises security issues that must be addressed along the whole process so as to prevent vulnerability and unauthorized data access to third parties.

Furthermore, a self-run data centre is also generally capital intensive. Their physical footprint requires space allocation for installation, together with the physical, bare-metal hardware and environmental controls. A periodical update and asset refresh are required for the system to work efficiently, translating into further costs. Essentially, you still have to buy, build and manage the infrastructure, thus revoking one of the main benefits of cloud computing which is that of a reduced need for hands-on management.

Public Cloud

On the opposite side of the cloud deployment, public cloud services translate into the delivery of cloud services over the public internet, usually being offered over a paid subscription or free of charge, depending on the provider, support and functionalities that are being made use of.

Although a different concept, public and private cloud services differ very little from each other. In terms of functionalities, both deployment models both offer the same solutions. The main discrepancy between these models is that of the level of security concerns that are raised when sharing computational resources between multiple customers. To address this issue, providers implemented a system where a direct connection service establishes a secure link between users and their resources.

Hybrid Cloud

Cloud computing deployment models offer further solution for an enterprise that offers the best of both public deployment worlds in a hybrid form. This deployment model composes both previously mentioned models, maintaining a private cloud or on-premises resources and public cloud services as separate entities, yet coupling each paradigm, which combines the benefits of both deployment models.

Hybrid cloud computing can also translate into the ability to connect dedicated local services with cloud services, in other words, a combination of public, private and community cloud services, from different providers. This translated into crossing isolation between services and provider boundaries, meaning that all services cannot be simplified into one category of public, private or community cloud service. With all this, one can extend the functionalities of each model so as to extend the capacity and/or capabilities of a cloud service, making use of aggregation, integration or customization with various cloud service models.

The main use case for a hybrid model is when an organization stores sensitive client and business information of a private cloud application, but simultaneously interconnects public cloud-hosted software services with the private infrastructure. This, therefore, extends the capabilities of the enterprise to deliver a much more specific business service by adding external public cloud services.

There are a number of factors that must be taken into consideration when adopting a hybrid cloud model, namely data security and compliance requirements, control levels regarding information management, applications that are used, and so on.

It has been a recurring subject, but security is the characteristic with the most weight and that requires the most investment when dealing with all things *cloud-native*. The best way to implement this is to limit a user to a *need-to-know* model when only the required authorization level for conducting activities is given.

The methods with which the information within an application operates and the levels to which the management is made defines the functionality of the cloud application. Just as in an on-premises model where the infrastructure is locally accessed, cloud implementations have the same planification requirements. Although many details are abstracted, planning is essential to guarantee a system that is consistent and operational, regardless of how an organization proposes to operate.

In its essence, hybrid cloud infrastructure looks to eliminate the limitations which are associated with multi-access relaying. This paradigm translates into the system inheriting enhanced runtime flexibility and adaptive memory processing, which are uniquely available when making use of virtualized interface models.

Security and Privacy

Anything that is shared publicly poses the threat of security privacy issues. As aforementioned, Cloud Computing services pose many concerns regarding privacy issues given the fact that providers have access to all the data stored within the offered infrastructure. [3]

This could give way to information tampering or deletion, both accidentally or deliberately. Having unlimited data access also translates into the concern that providers may share the information to third-parties, many a time for litigation purposes, without the need for a warrant, which is mentioned in privacy policies that are required before the user begins taking part of the provided benefits. Although this is a valid concern, it is strongly advised that users encrypt their data, both that is processed within the clouds component and that is stored on the within the cloud, which prevents unauthorized access to the information.

In the article “*Top Threats to Cloud Computing: Egregious Eleven*” posted by the Cloud Security Alliance, the top three threats in *cloud-native* technologies are “*Data Breaches, Misconfiguration and Inadequate Change Control, and Lack of Cloud Security Architecture, and Strategy*”. When looking into statistics within Cloud Security, 84% of organizations that make use of cloud systems consider traditional security solutions to not work in Cloud Environments [4], 80% of security breaches involve privileged credential access, and, until 2022, at least 95% of Cloud Security failures are predicted to have an origin in customer error. Given these values, there are a set of best practices to take into account, and not just crossing fingers and hoping that a security breach doesn’t occur. *Due diligence is key.*

Redlock.io makes the following suggestions when preparing for eventual breaches and unauthorized data access attempts:

- Educate employees – the first step to securing sensitive information within an organization is by training staff to understand how to identify cyber threats and what to do when they are confronted with them;
- Encrypt Data – this acts as a safety lock with a combination over information, guaranteeing that only those who should have access actually do;
- Implement multi-factor authentication – this adds another layer of protection to data, which in turn makes unauthorized access that more complicated;
- Limit Access Control – Identity and access management technologies allow for user access control, once again dividing users into those who may access valuable information and those who cannot;
- Test Security Measures – ethical hacking, or white hat hacking has become a growing field, alongside security threats and solutions. These ethical

hackers find vulnerabilities within firewalls and security systems and report back on which these are and how they can be mitigated;

- Monitor and remediate resource misconfigurations – make use of robust cloud security solutions that correct misconfigurations and reduce the window of opportunity for malicious actors;
- Detect and remediate anomalous user activity – making use of AI within a business is not just to make chatbots and robots that respond to voice commands. Understanding and implementing AI so as to detect abnormal behaviour and correct these situations is an effective way for threat mitigation and reduction;
- Detect and remediate suspicious network traffic – cloud environment monitoring for threat detection is crucial across all resources;
- Identify vulnerable hosts – vulnerability data must be correlated with configuration data so as to identify vulnerable hosts within a cloud environment;

This may appear to be a seemingly endless list of chores to perform when taking your first steps into Cloud Computing, but given the amount of security threats posed by external factors, they are essential when looking to keep private information and data out of unauthorized third parties.

To conclude, cloud computing is a perfect example of rather being safe than sorry, and implementing correct and effective security measures mitigates a large sum of potential risks that can surely pose a threat to an organization.

Limitations and Disadvantages

As in all technologies, concepts, paradigms, Cloud Computing is coupled with limitations and disadvantages. Bruce Schneier, a cryptographer, computer specialist and writer defines cloud computing as “*A restaurant with a limited menu (...)*” [5]. This was an analogy to explain how a big drawback of Cloud Computing is that it is limited regarding customization options. Together with not meeting legal needs, cloud providers have control over all the back-end infrastructure and policies, which defines what users are able to make use of within the environment. Together with these, users are also limited to how their applications are managed, meaning there may be an opening for data caps where each user is allocated a certain limit of bandwidth with which they can operate with.

The previous chapter mentions the concern which attracts the most attention which is that of privacy and confidentiality. Given that cloud systems work via the internet, confidential information that is not encrypted before being uploaded to cloud services can easily be accessed by third-party organisations if the provider does not implement the necessary measures to protect user data. Data theft, account hijacking, insecure APIs, technology vulnerabilities all compose the concerns associated with security and data within clouds.

Finally, downtime is a big limitation within the cloud, and even more so for smaller enterprises that make use of cloud services. Reboots, network outages, downtime, natural disasters, all these amount to technical problems that can occur within any IT setup. Being dependant on a cloud provider for these occurrences creates a dependency on how this

problem is solved from within their infrastructure. From a private deployment’s point of view, minimising impact and periodic occurrences must be managed so as to ensure that users have the maximum level of service availability, even when an update needs to be implemented.

Market Shares and Organization’s choice

Cloud computing has been an ever-growing ecosystem of technologies, products and services that has evolved into a multi-billion-dollar economy, where market shares are fought over constantly to achieve as much of an edge over the competition as possible. Although a predominant sector with many stakeholders, the industry giants – Amazon Web Services, Microsoft Azure and Google Cloud - have the larger share of the market.

A study conducted in 2019 by the Synergy Research Group, across seven key cloud service providers, operators and market segments reported that revenues associated with all things Cloud exceeded \$150 billion, in the first half of the year alone and reflecting a 24% growth in comparison with the previous year. This growth in investment and revenue shows that an already substantial market still has an even larger scope of expansion into which it can grow.

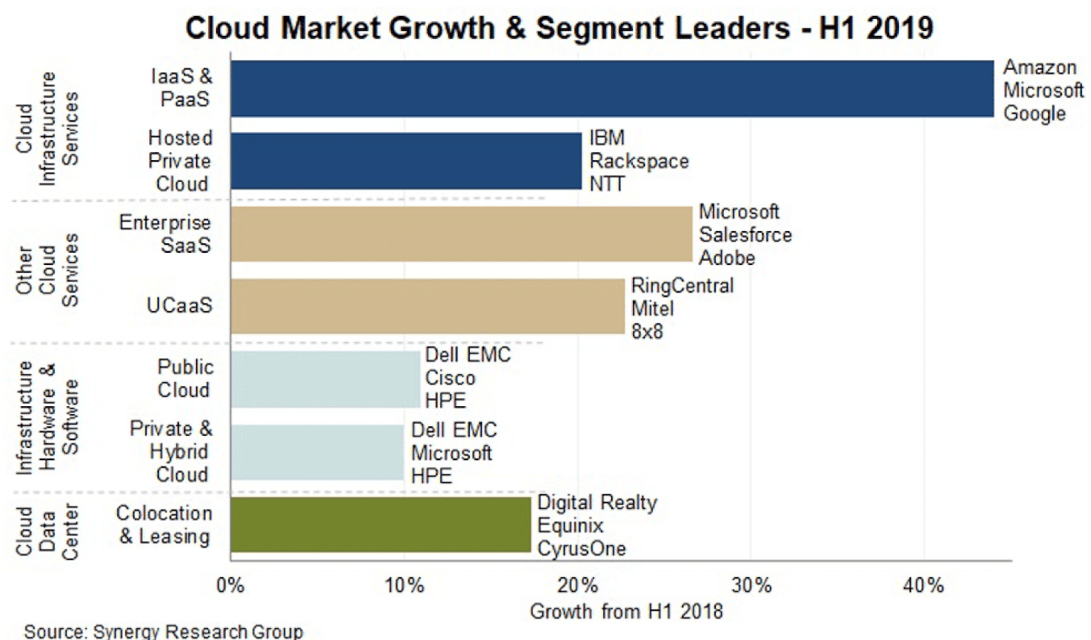


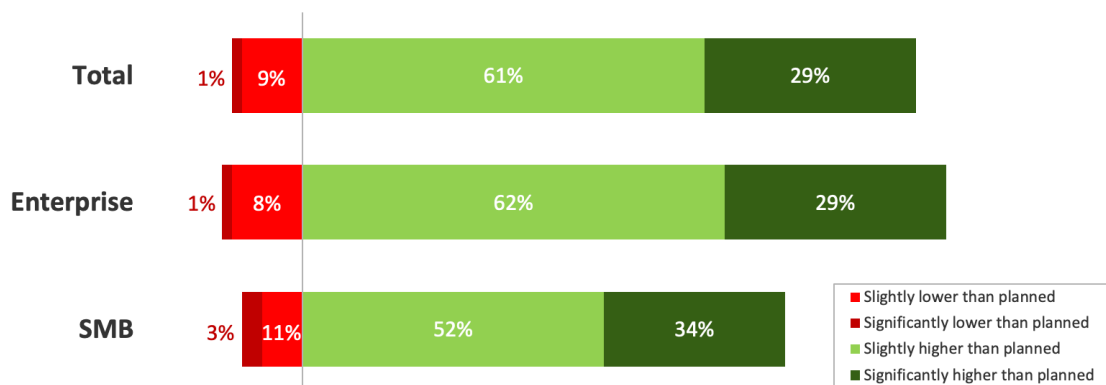
Figure 31. Cloud Market Growth

According to *RightScale’s (Flexera – State of the Cloud Report) 2021 State of Cloud report*, public cloud solutions make up for the majority of the cloud market’s segments. This report explores the opinion of 750 global cloud decision makers, taking into account their thoughts on public, private and multi-cloud markets. In this report, 19% of the inquired businesses confirmed that they made use of exclusive public cloud services, 78% opt for a hybrid model and 2% make use of private solutions. In sum, approximately 97% of the market share is occupied by public cloud solutions and 80% of systems contain private cloud systems.

Even before the global pandemic, cloud services were experiencing a strong level of expansion. But, given the need for change within business operations due to the restricting factors, adopting and integrating cloud functionalities into the business model caused for an even more accentuated level of growth. Businesses plan their adoption to new technologies regularly if they intend to keep up with market trends and R&D levels of competitors. With the arrival of COVID-19, businesses had to react quickly and efficiently to guarantee that they could keep functioning given the limitations. According to business sizes, large enterprises stated that, compared to their business plans, 29% required a significantly higher level of change, 62% a slight increase, 8% a slightly lower need, and 1% a significantly lower level of change. Whilst in SMBs, 34% had a significantly higher change rate, 52% a slightly higher need for change, with 11% and 3% responding to a slightly lower and significantly lower need for change than planned, respectively.

Change from Planned Cloud Usage Due to COVID-19

% of respondents

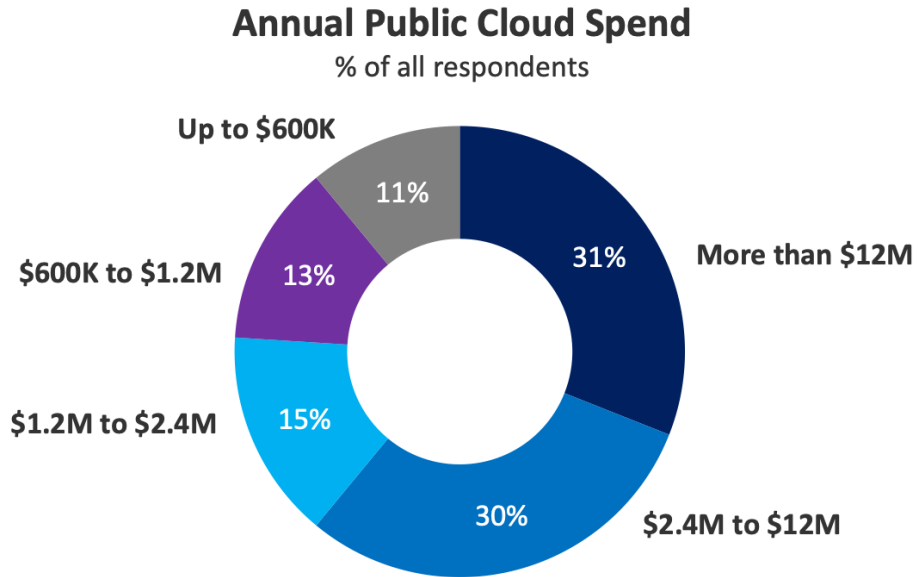


N=750

Source: Flexera 2021 State of the Cloud Report

Figure 32. Change from Planned Cloud Usage Due to COVID-19

From an expenditure point of view, the public cloud has 31% of its users spending more than \$12 million dollars annually. Close to one third of users have a level of investment that exceeds this value of costs, which reflects well how public cloud solutions occupy the delivery model market, and how much investment has been placed in this model.



N=750

Source: Flexera 2021 State of the Cloud Report

Figure 33. Annual Public Cloud Spend

The same survey evaluated the shares of Cloud providers within the public cloud adoption realm, in which each respondent specified if they had applications running on Cloud services, experimenting, planning to use, or did not have any plans to use the service. The document outlines the fact that many are making use of more than one cloud, which leads to percentages being superior to 100. As represented in Figure 34, Amazon Web Services are the majority shareholder of public cloud use, where 50% of respondents have 50% of their significant workloads running on AWS, with Microsoft’s Azure being the runner-up with 41%, and Google’s Cloud retaining 22% of the user’s that were inquired. Half the significant workloads are kept on AWS, 9% clear of its closest competitor which, although seemingly a reduced number, if adapted to the whole market size, accounts for half the share within a \$150 billion dollar market.

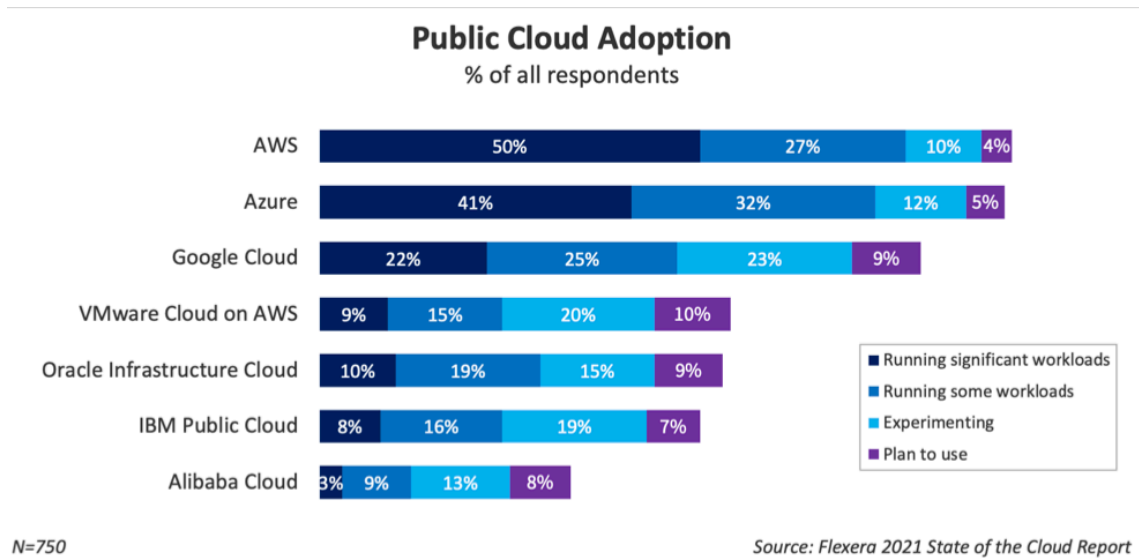


Figure 34. Public Cloud Provider Adoption

This, in the year of 2020 alone. Comparing rankings with the current year of 2021, AWS had a growth of 1%, from 76% to 77%; Microsoft’s Azure registered a 10% growth in adoption, achieving 73% of market share; and Google’s Cloud increasing in 12%, settling with a 47% market share. Smaller providers such as VMware, Oracle and IBM have come to grasp larger numbers within the market of Cloud providers, coming in with 24%, 29% and 24% respectively. The increase of adoption of the smaller players within the Cloud, versus 2020, doesn’t pose a threat to the larger corporations, but is a starting point for a more competitive market, which will reflect a benefit towards users.

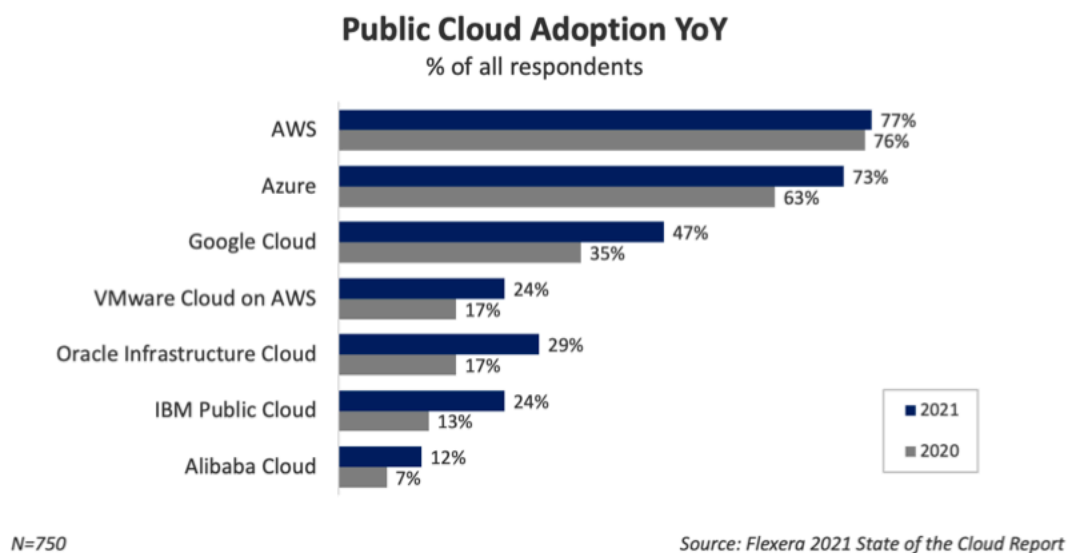


Figure 35. Public Cloud Adoption Year on Year

To achieve the functionalities and objectives of the internship and final project, Wit Software opted to adopt the services provided by Amazon via their Cloud service AWS. Being the leading market shareholder, and also being the ground-breaking provider with the longest track record within the sector, in my opinion, it is the correct decision to take. Not only from a theoretical point of view, but also from that of experience, given that I

myself have also had direct interaction with the functionalities provided. Essentially, all cloud providers offer the same functionalities for their users.

The availability of on-demand servers, being accessed from any location along the globe. Amazon took a big interest into understanding how to offer their services in a “pay-as-you-use” format, which incorporates a lot better into the use of server infrastructure, as these are used in a much more burst-oriented manner. Given the fact that traditional hardware is left on standby for about 90% of its lifetime, Amazon has developed a system which keeps the price of computing infrastructure to the bare essential, avoiding management costs when the system load is inexistant.

Provisioning a hosted web service can be a time-consuming task, even when making use of third-party providers to conduct the task. While making use of traditional means takes a number of days to conduct the task, AWS has developed a form of reducing the deployment time to minutes. Configuration is also much easier due to the fact that during the entire process prior to launching the web service, the user is presented a form which inquires on the various characteristics of the desired service and how it is to operate. The level of configuration is obviously limited, as expected when delegating a task to a third-party cloud provider, but this limitation is compensated by the heavily reduced time-to-market benefit.

Furthermore, as it is regarding *cloud-native* systems, security plays an important role within the use of AWS. This front is covered by the use of Identity and Access Management (IAM) which defines which policies and privileges are given to a user, helping to reduce malpractice and user-error by a large margin. Together with IAM, AWS also offers a Virtual Private Cloud which is oculted from the internet, but can still interact with resources that are located on the same network, acting as a protection barrier from third-party attacks that origin on the internet. The resources that are located on private networks can be accessed by making use of a Virtual Private Network, which Amazon also offers a solution to.

Finally, Amazon prides itself in offering the “five nines of availability”, meaning that the services provided through the use of AWS are online and accessible 99.999% of the time, which translates into an almost inexistent level of downtime. This translates into constant delivery toward the user’s clients’ needs, allowing for consistent and continuous access.

Microservice Architecture

Seen as an ambiguous term without a concrete definition, Microservice Architecture or Microservices have been given a panoply of definitions, which have led to a consensus amongst practitioners within the industry. In this chapter, this architectural paradigm will be explored, with the intention of presenting which contexts it can be applied to, where it has. Rather than giving a formal definition to the term itself, the characteristics that constitute the Microservice Architecture are what have been regarded as the defining factors for the architectural paradigm.

A Monolithic Starting Point

Traditionally, a server-side application would be built to support a wide range of different clients. Internet browsers, native applications, third-party consumers [6], all which partake in the use of a public version of an API. The application will process external requests via an HTTP protocol, for example, triggering business logic tasks, database access, message exchange and finally returning a response for those same clients. In a monolith, all these server-side applications are running on the same server, which is operating on the same VM, with a limited number of resources and little room to manage much more. This is what defines a monolithic application architecture.

This solution offers a huge benefit: simplicity. Simple to develop, simple to deploy, and simple to scale. To an extent. Initially, the application may be reduced in terms of size, without much load and easy to manage. However, with success and adhesion, the application will become larger, with its development team growing in an almost directly proportionate manner, which then raises a number of concerns that become increasingly significant. New team-members are intimidated by the large code base where all the functionalities are implemented, understanding the application and applying modifications become a difficulty, and development begins to slow down, productivity dives and the final product is both delayed and not completely functional.

With the increasing level of competition within the field of software development, the need for continuous deployment and the inclusion of new features must be carried out regularly. Although a possibility when using monolithic applications, it is a difficult task due to the fact that the entire application must be redeployed whenever a change needs to be implemented. When a change is of a large scale, this can be a solution. But, given the need of immediacy within the field, the need for constant, small changes to an application, giving the application a full reboot just to implement a new message code between processes doesn't compensate the downtime.

Even giving the possibility of overcoming all of these obstacles and limitations, development and application scaling bring into scope a number of concerns with the use of monolithic solutions. Once applications begin receiving a larger load of users, the resources that were initially made available will reach a point of rupture, either by fulfilling the intended functionalities in a reduced and time-consuming fashion, giving its users an obstructed experience, or not providing any solutions whatsoever, overloading the system and possibly causing a system crash.

Furthermore, adopting a monolithic architecture implies a long-term commitment to a technological stack. In other words, the architecture forces the use of a certain technology and development tools, which can cause difficulties when wanting to incrementally adopt a different technology. This limitation also implies that, if the framework that is being used becomes obsolete, migration becomes another complication as many a time it is not possible to adopt a more innovative technology without having to rewrite the entire application, which implies a lot more risk with the business. This limitation both withholds an application from innovation and a team from being able to adapt to new cultures and ways of thinking, which, in the long term, can keep an application from growth and from maintaining a competitive quota in its market.

Of course, this situation can be avoided with the correct amount of management and team orientation, but this implies a very strict and fine-grained *modus operandi* within the development scope. Given the evolution of development cultures where DevOps and application flexibility is a requirement within an ever-changing and ever-evolving field of practice, a monolithic architecture is an option best suited for small applications and development teams where, as growth takes place within, an evolution toward microservices becomes a more viable solution.

Microservices

The term *microservices* was first mentioned in 2005 by Peter Rodgers when arguing in favour of REST-services, against SOAP SOA architecture. During this presentation, he went on to say “*Micro-Services are composed using Unix-like pipelines (the Web meets Unix = true loose-coupling). Services can call services (+multiple language run-times). Complex service-assemblies are abstracted behind simple URI interfaces. Any service, at any granularity, can be exposed.*”

The microservice architectural style can be defined as an approach of developing a single application whilst dividing each component and feature into its own process, forming a suite of small services which make use of lightweight communication mechanisms. The way these microservices are built and defined are based on business capabilities and to what extent they can be deployed independently and operate autonomously. As expected, there is a bare minimum in terms of the amount of management that is made from a central entity, but this will act as a component within the whole microservice architecture, acting as a microservice itself. Each of these services can be developed using different programming languages and technologies.

Taking the previously mentioned architectural paradigm into comparison, microservice architecture is essentially the building of application suites which are deployed and scaled independently, forming a firm boundary that defines the module, yet still allowing different services to communicate amongst each other, even when they are developed using different languages. With this, the architectural paradigm can also lead to different teams managing the different modules accordingly.

The form of application construction is strongly compared to Unix-based system design where the different processes communicate with each other by making use of lightweight communication mechanisms, more specifically FIFO pipes. This leads to the

microservice style not being considered a novelty or innovative, but that is it very often overlooked when looking to develop software.

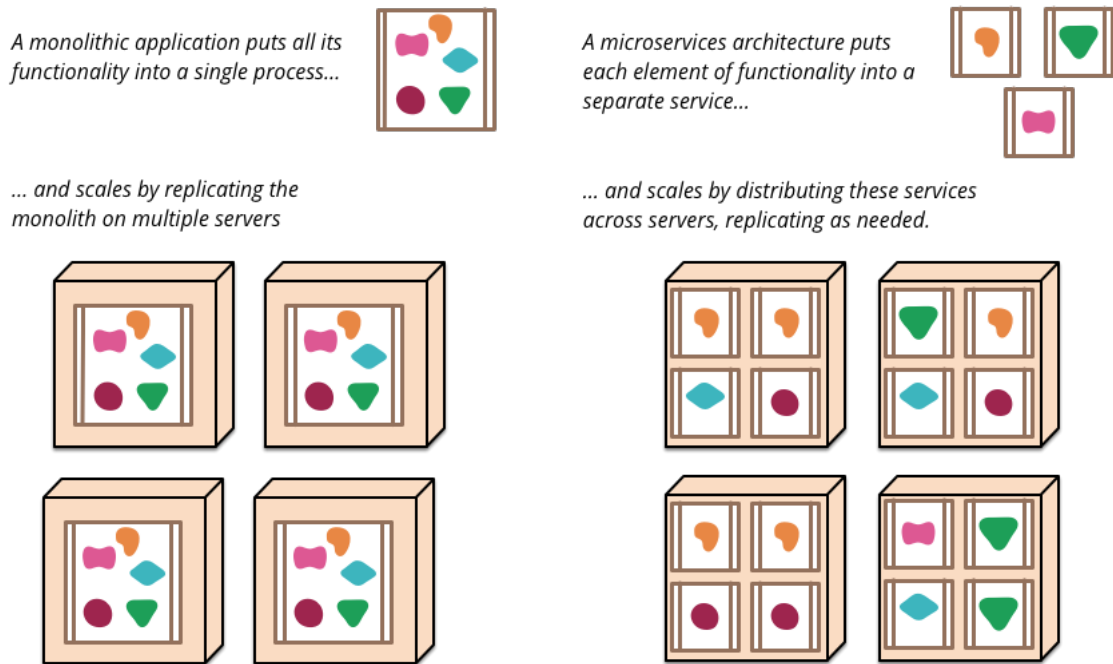


Figure 36. Monoliths vs Microservices

Characteristics

As mentioned previously, there is not a formal definition for the microservices architectural style. Although, it is possible to define its characteristics to be able to find a description of what is common within the label. Given that not all microservices have the all the pertaining characteristics, a common ground between the various applications can be defined. These characteristics are the following:

- Implement componentization via services [https://martinfowler.com/articles/microservices.html];
- They are organized around their business capabilities;
- The intention is to develop products, and not projects;
- Microservices make use of endpoints and pipes, as do Unix systems;
- Decentralised Governance and Data Management;
- Allow for Infrastructure Automation:

The microservice architecture aims to achieve a much simpler form of developing an application. With the evolution of time, developers have been prone to build systems by “plugging in” various components which work together to compose the system in its entirety, and this is very much a possibility when using services as components within a system [https://medium.com/hashmapinc/the-what-why-and-how-of-a-microservices-architecture-4179579423a9].

Traditionally, an application would consist of a single process making use of various libraries to perform, whilst in the case of using services, these can be deployed independently. This difference has a big reflection when looking to apply changes to an application because, as we know, bug fixes are as common as the number of lines of code in a program, and therefore, when looking to apply a change in the application, the idea of a single process making use of various libraries translates into a full reboot of the entire application to be able to apply a new change. Whilst on the opposite side of the coin, the microservice scope makes use of out-of-process components that communicate via web requests, applying an alteration in one service requires that only that service suffers a redeploy. This is not a guaranteed occurrence, of course, as there must be a guarantee of service whilst the application is rebooted, resulting in the need for some coordination and backup solutions, but given the main goal of applying a microservice architecture is to find a way of minimizing these moments through well-established service boundaries.

Another consequence of using services as components means that each service has its own interface, defined according to the remote call mechanisms that are used by the service. Even though the idea behind microservices is to have independent processes acting as building blocks to construct a larger process, a service can be built using various application processes and its own database. In other words, a small equivalent of a monolithic application, but this custom is useful in only a small number of situations.

In the customary application development model of having various parts within the same application and trying to separate these into parts usually leads to trying to optimise around having to have cross-team projects. This strategy helps with organization in the short-term, but when implementing the application, teams will try to force logic into whichever application that the team has access to, which then leads to having business logic in all of these.

When looking into microservices, the division is made in a different manner. The architectural paradigm organizes each service into its intended capabilities. In other words, each service implements its own business logic and capabilities, making use of its own functionalities, with an interface and data persistence being applied to a single service if necessary

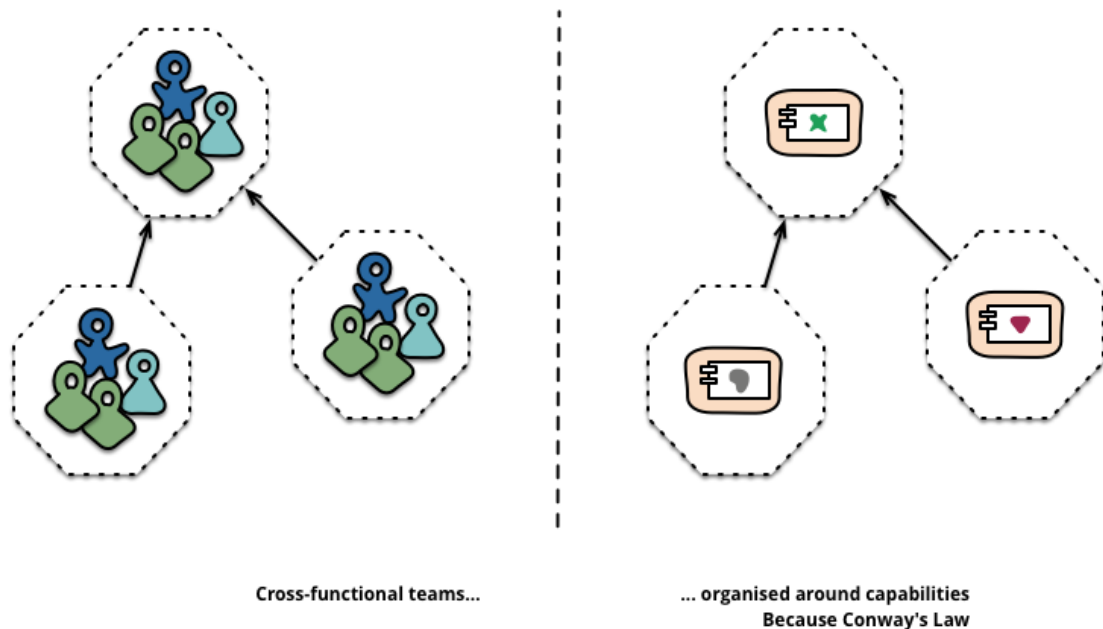


Figure 37. Service Boundaries reinforced by Team Boundaries

Products not projects: Microservices look to create a form of building a product in such a way that the development team then participates in its ownership further than just the development and implementation phase. The development team then takes full responsibility of the software in production, bringing a day-to-day experience with how the application behaves with its users, increasing the contact made and being able to give support from this point of view as well. Essentially, microservices look to make teams adopt the mentality that software isn't just a list of functionalities to implement, but rather an on-going involvement with the application long after it has been developed and delivered to its users. The fact that microservices have a much smaller granularity, creating a closer relationship between developers and users is much easier.

Applications built using microservice architecture look to find the most decoupled form of communication between each service. To achieve this, each has its own domain logic and operates in a request-response sequence where a request is received, the appropriate logic is applied and a response is produced, usually making use of HTTP protocols to achieve these tasks.

Together with using these protocols to communicate with the various endpoints, the technologies used to communicate between the various endpoints are mere asynchronous message routers that convey requests and responses. This reflects the idea of using "smart endpoints and dumb pipes". As in Unix systems, the pipes used are just convoys between processes pass a message, whilst the process that actually receives the message is then the "smart" functionality which interprets the request, actually performs computational tasks and prepares a response.

In a monolith, communication between components is made using method invocations or function calls. This simplicity is lost when looking to convert a monolithic solution into microservices due to the fact that these then require a change in the communication pattern, making a switch from in-memory calls to Remote Procedure Calls, which leads to a more coarse-grained approach, consequently increasing complexity.

When splitting an application into various microservices, there is a possibility of choosing which technology is going to be used to develop each component. In a monolith, the tendency is to use a single technology to be the solution to a problem. This philosophy limits teams and doesn't quite guarantee that the intended problem will be solved in the best way possible. In other words, "*not every problem is a nail and not every solution a hammer.*" [<https://martinfowler.com/articles/microservices.html>]

Decentralizing the governance translates into giving development teams the ability to choose which technology is going to be used to implement each component. For example, needing to implement a service which organizes reports on a web page in AngularJS or having to develop the database communication module in Typescript, there is no problem in each of these being developed with their own technology, as long as the communication between each other is made correctly. This helps join with the fact that teams are responsible for the software when already in distribution, keeping in mind that the maintaining good operations after application delivery guarantees that products are created and not those mere projects that are developed, handed over and forgotten about.

Decentralized Data Management: When it comes to data management, microservices also present a number of different ways to decentralize this component.

In its essence and looking at the highest level of abstraction, the different systems will have a different conceptual perception of the world than their counterparts. The natural correlation between services and their own context boundaries helps define the how each service is separated and, therefore, which elements have a database shared amongst them and which make us of their own. Each service manages its own database, applying the concept of polyglot persistence, where each application has its own instance of the database or an entirely different database system.

As much as applying this rationale does help with componentization and modularity, dealing with updates to resources can become a problem, especially when some information may be contained in different databases and will need to be consistent in every service that uses the information. A possible solution to overcome this difficulty is to use transactions or to share a single database among multiple services. The former solution offers a more complex solution given the difficulty associated with distributed transaction implementation, but allows for a quicker response when looking to satisfy demand. The latter solution is much easier to implement, given that the database is shared amongst services, but this both defeats the purpose of using microservices instead of a monolith and simultaneously gives a slower respond to processed requests, meaning a slower when dealing with higher loads of requests.

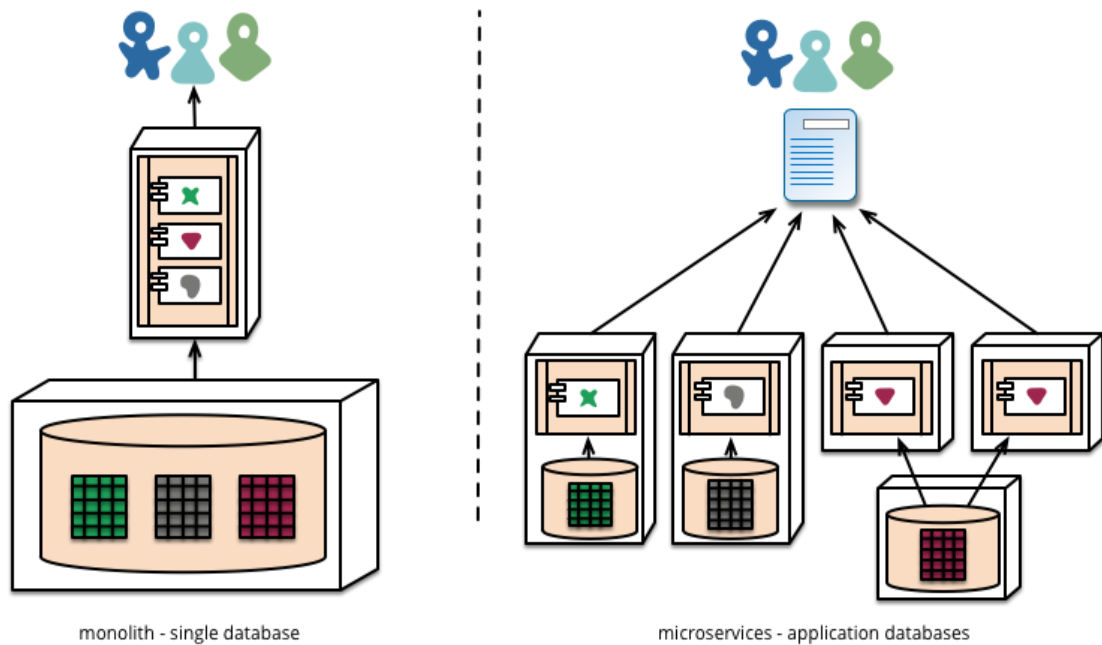


Figure 38. Different implementations of a Database

Automation is a term strongly used within the informatics industry, and in this case, it is also strongly associated with the microservice architecture where building, deploying and operating applications has had a reduction in operational complexity. This architectural paradigm has opened the door for two concepts that have defined how teams develop software. Those two concepts are Continuous Integration and Continuous Delivery, where teams that operate with microservices make use of infrastructure automation to deliver software to their customers. This both speeds up time-to-market and application upgrades. Coupling microservices with Continuous Integration and Continuous Delivery, once the automation is prepared for one service, it can be easily applied to the remaining applications.

Costs

Obviously, as all paradigms and technologies, there is no one-size-fits-all nor a system which doesn't come with drawbacks within its trade-offs. Microservices is no exception, being that the more noticeable shortcomings are:

- Communication Between services is complex;
- Debugging and testing can be troublesome [<https://cloudacademy.com/blog/microservices-architecture-challenge-advantage-drawback/>];
- The fact that it uses distributed systems to improve modularity [<https://martinfowler.com/articles/microservice-trade-offs.html#boundaries>];
- Eventual consistency;
- Operational Complexity;

Considering that each service communicates with its counterparts by using RPCs, each of these must be able to communicate in a consistent and secure manner and when looking at systems with a larger number of service components, guaranteeing these requirements

means an increase in complexity and a need for attention regarding the system's performance. Performance levels are indirectly proportionate to the amount of service within the system.

This problem can be mitigated with the use of more granular calls, reducing the amount that are made. This then complicates the model due to having to group-up the different interactions which ends up not having much of an effect, given the fact that each service that plays a part in the task must be called at least once.

Another possibility for mitigating this issue is the adoption of asynchronous tasks, but this has a big impact on the amount of time that each remote call occupies to achieve a response. Other than the amount of time lost already being defined to the slowest service in a request chain, the amount of wait time becomes defined by this service and the sum of latencies between each communication call, which means another worry when implementing the tasks in a component. Given this distributed nature, debugging is also much more challenging, given that each microservice will have to maintain a clear logging system to understand where eventual problems may be occurring. Debugging and testing walk hand-in-hand, and when looking to test how a system works in its entirety, whilst using distributed components, many a time this cannot be achieved, leaving development teams in the dark.

Eventual consistency translates into the situation where, after applying an update to a website, the page is refreshed, but the change only appears after a few page reloads. In the microservice realm, this inconsistency can be exemplified by a node that received the update being dependant on another which must process a database request. Until the node which communicates with the database receives the requested information, the updated node will only reflect its changes once the chain of custody is complete. Given the fact that microservices use decentralised data management and multiple resource updates, developers must take into account the existence of these inconsistency and produce strategies to mitigate these gaps within the system.

As much as developing and deploying small components is seen as a productive and efficient form of producing software, the weight that a few hundred microservices have on the operations staff is much larger and difficult to handle than a few monolithic applications which perform the same tasks.

This drawback once again brings the role of Continuous Delivery into scope. Automation becomes a tool which is essential when treating microservice as it smoothens and organizes operations a lot more. Continuous Delivery is strongly coupled with the need for service management and monitoring.

The development of microservices requires the application of new skills and tools to be able to keep a consistent operation active and successful. To be able to apply these skills into a development team, adopting a DevOps culture is also required, leading the need for a larger collaboration between development and operations teams, usually combining these to find a more coherent development culture amongst collaborators. When transitioning from a monolithic architecture into microservices, culture change is the biggest challenge associated with the change, which requires that a transition be very well planned and prepared before consideration.

To conclude, microservices as an architectural paradigm are best suited for larger enterprises which have the manpower to be able to create and iterate changes and content on a larger and more efficient scale. Therefore, given the analysis done throughout this chapter, the ideal practice should be to begin with a Monolithic approach to begin the development process. Once the application begins reaching levels of growth where it is no longer viable to maintain this paradigm, the application should begin transitioning towards a microservice architecture, together with the adoption of DevOps culture, making organization resources available at a level where the change can be implemented without suffering a loss to business.

DevOps

In this chapter, we break down the idea of what the DevOps culture is and the various technologies that are associated with this approach to software development. Understanding this concept is fundamental for this thesis as it is the main motivation for the implementation of the final product and its different components. The contents of this chapter will be a formal definition of what DevOps culture is, the processes within a DevOps culture, which are its constituents and what technologies are associated to develop software within a team that is indoctrinated with this approach.

DevOps Culture

In short, the DevOps culture can be defined as the “*approach to culture, automation and platform design intended to deliver increased business value and responsiveness through rapid, high-quality service delivery*”. [<https://www.redhat.com/en/topics/devops>]

Traditionally, software development teams were built and composed of various “silos”. One part of the team would focus on code development, the other on testing, operations and customer support, *et cetera*. With time, it has been understood that having this separation between teams, although a successful strategy, does not have a long-lasting effect when it comes to quality and customer satisfaction. The introduction of agile methodologies has helped to break down these barriers, although deployment, operations and maintenance are activity groups that have been separated from the remaining development process. These so-called silos were broken down on one side yet kept on the other. The DevOps movement and culture looks to remove these borders and increase the collaboration between Development and Operations teams.

By looking at the name itself, DevOps is the literal joining of the terms “Development” and “Operations”. This culture does not limit to just conjuring these extends to more than just the conjugating these words and their consequent departments also but, by including security, collaboration between team members, data analytics, DevOps’ objective is to instruct team-members to adopt an attitude of shared responsibility of a product. This increases collaboration in the different tasks that are carried out, thus making code production a much easier task and, subsequently, increasing quality levels within the development process.

The underlying characteristic within the DevOps culture is increased collaboration. This refers to the sharing of the responsibilities that belong to both the development and operations roles and how this can simplify the deployment and maintenance of software products. From the point of view of the development team, developing software and handing it to a different department to manage and guarantee that it works is easy and leads to a loss of interest in the way the neighbouring department has to carry out their work. With the sharing of responsibility between the two departments, the development team understands the sufferings of the operations team, leading them to find ways of simplifying the deployment and maintenance of the software they hand over. Additionally, observed requirements can also be gained from the monitoring system, which helps feed the development in understanding what can be added, improved or

removed from the system. From an operations point of view, the sharing of responsibilities regarding business goals gives way to a better understanding how the system works and how they can guarantee that the system works as intended.

When looking to adopt a DevOps culture within a development team, there are a number of details and possible malpractices which can seem natural but must be avoided. The idea of the culture is to not have any “silos” between both teams, therefore having handover periods defeats the purpose of teams working together on a software solution from the early stages of development. Therefore, a good practice to implement is co-locating both departments. This will increase the collaborative nature between members and help feed the idea behind having this culture among workers. DevOps also blurs the lines between the two roles, ultimately eliminating the distinction between the two, meaning that both success and failure is shared amongst all those involved. Creating a “DevOps team” is another practice that should be avoided as this is a form of maintaining the same separation that the culture looks to dissolve.

DevOps looks to give teams an autonomous workflow. This organizational shift implies that staff make their own decisions, applying the due changes without the need of following a complex chain of command to get approval for the updates to take effect. Initially, this change within an organization will seem like diving into the deep end without knowing what’s in the water as teams will have the full trust of the organization to manage risks and failures. This practice opens the door for an environment that is free of the fear of failure. Of course, tools to guarantee a coherent workflow will have to be adopted, one being that of Version Control. Changes that are applied can be linked to a ticket-based system where changes can be audited and managed so as to guarantee correct functionality. As a consequence of using Version Control, teams can also look to speed-up testing and automate their deployments.

Continuous Integration/Continuous Delivery

Continuous Integration/Continuous Delivery is a method used to produce software in an organization, where developers are frequently delivering applications and changes to customers with the inclusion of automation tools in the development process. CI/CD is a solution to problems created for development and operations teams when looking to integrate new content. [7]

CI/CD looks to introduce ongoing automation and monitoring into an application’s lifecycle, beginning at the integration phase, through testing and finally to delivery and deployment. Once together, these two concepts allow for the creation of a CI/CD pipeline which looks to join both development and operations teams. This junction of the two teams creates an adoption of a DevOps culture into a development workspace.

Continuous Integration

This section will serve to better understand what the concept of Continuous Integration is and how it is relevant to the internship’s project together with the problems that the internship presents. An explanation of what is expected from this module is explained, together with an analysis of the different tools that offer a solution for the problems at hand.

Continuous Integration is defined as the act of applying multiple changes to an application in a short period of time and having those same changes committed, tested, and verified to be applied to the production environment. The differing element in this concept compared to the customary development process is that the entire process is automated.

In the scope of this internship, the framework that will be created must be able to receive multiple alterations to application code and guarantee that they are ready and functional for testing and deployment. These changes must be notified by a Version Control System repository and must automatically make a pull request for the altered code. Once the request is fulfilled, the code must be built and composed into an executable artefact. This process must maintain the characteristic of being fully autonomous and given that some errors may occur during the production of the executable, there must also be an integrated notification service which is responsible for notifying the developer that there was a problem with the pipeline. This system will roll back the alterations in the code and guarantee that, whenever a change to the code is produced, it will not cause downtime or errors in the application when the new component is included.

In summary, regarding the needs of the framework, as aforementioned, this component of the pipeline will have to be able to detect changes in the application's code, build and compose an image container, and in the event of failure, notify the developer and perform rollback the changes in the code. To achieve this, there will have to be an analysis of the solutions offered on the market from both an agnostic origin and also from Amazon's AWS. In this case, the two tools focused for analysis are Jenkins and AWS's CodePipeline, the leading open-source automation tool and the cloud provider's solution, respectively.

Continuous Delivery

In this section, the concept of Continuous Delivery will be further developed. There will be an initial definition of the concept itself, followed by an explanation of the needs that the framework will require from the standing point of Continuous Delivery. Furthermore, an explanation and comparative analysis of the solutions that exist for these issues will be made to establish which is the best tool for providing a solution for the problems that compose the internship.

Continuous Delivery is the ability to implement all types of changes to software - including new features, configuration changes, bug fixes – into production in a quick, safe and sustainable way. To be able to do this, the software's code must always be in a deployable state and ready to be applied in the production environment.

Traditionally, whenever a change was made to the source-code of an application, testing those same changes and new implementations required manual, time-consuming tests to be conducted to guarantee that all was in order for the delivery of the software. This, together with the possibility of creating “code freezes” which would stop development for some time until a bug was fixed, or a new version was released meant that there was no continuity and consistency in the process of deploying software.

With CD, the testing and integration phases are completely automated, eliminating the possibility of a break in the production line of an application. Whenever a change is implemented, the automated tests are conducted to guarantee that the changed software

is deployed to the production environment without the possibility of a break in application functions or a negative experience for the user, having to operate around bugs and imperfections.

The tool responsible for Continuous Delivery in the pipeline will have to be able to produce a production-ready artifact which, when deployed, will be delivered into an EKS cluster. This deployment will have to be done with the guarantee that the application will not have any downtime for its users, whilst following a strategy of phased rollouts – implementing the changes to a system in an incremental form. In other words, each module will be changed one at a time, so as to safeguard the organization from having to deal with possible implementation issues, false-positive tests, or unexpected bugs.

Once the rollouts begin, there must be at least two environments which are as identical as possible to begin a Blue-Green deployment. In other words, the traffic routed to the previous version of a pod (blue) must be divided and re-routed to the second pod (green) in such a way that the new version can be integrated into production and monitored.

At the rate that the deployment is operational and does not interfere with operations, the traffic that the new pod receives gets incrementally larger. This strategy follows the term of Canary Deployments where a small group of the user population receives the new version of software and acts as a sample for how the application will work in its final environment. At the rate that success is seen in this sample, a larger group of users receive the update, and so on until either an error occurs, or all users receive the update. In the event of an error, there must be a mechanism placed which will perform a rollback of the application to its previous stable state, and the development team must be notified of this occurrence.

State of the Art

In this chapter there will be a general overview of the problems associated with the internship. Each section will begin with an explanation of the problem it looks to resolve, followed by the analysis of the tools that exist in the market at the moment and that look to provide a solution for the three general problems that the framework looks to address: Infrastructure Provisioning, Continuous Integration/Continuous Delivery (CI/CD) and Microservice Application Monitoring. Each section will have a list of the best practices associated with each of these problems, an explanation of the experimentation process carried out with each tool, and finally a comparative analysis of the results obtained from implementing and performing experiences with the tools.

The various tools will be evaluated according to the specific needs of the scope in which they perform. For the evaluation process, the symbols present in the following Table 1 will be used to compare the various tools and offer a better understanding of the possibilities which they offer.

Table 21. Symbology

Symbol	Meaning
✓	Feature is provided.
✗	Feature is not provided.
▲	Feature is provided but is incomplete.

Infrastructure Provisioning

This section will serve as an explanation of what the intended framework must be able to produce from an infrastructural point of view. “Provisioning denotes the prerequisite steps in managing access to data and resources and facilitating systems and users’ availability. It also refers to the setup of IT infrastructure [8].” In other words, infrastructure provisioning refers to the act of creating computational resources on which to deploy software applications.

When looking to create an infrastructure for an application in a cloud-native environment, there are two methods that can perform this task. One is to manually access the cloud provider’s platform and create the various components, one at a time. The alternative solution is to make use of a trade-off of the adoption of Cloud Computing and the Infrastructure as a Service model. This alternative is creating Infrastructure as Code (IaC). By using IaC, a solution is constructed through the use of configuration templates which are treated in the same manner as software source code. The creating of infrastructure using this model helps mitigate the errors in creation and configuration of application infrastructure, whilst simultaneously saving the developer the time required to manually go to each component and create it. The time used to configure and create a script file, define how the infrastructure must exist, and the deployment of the infrastructure is reduced considerably. Another advantage of provisioning infrastructure environments

with code is the replication of resources can be made by simply taking the contents of the configuration file and creating another file with the same values, reducing the possibility of human error.

The following figure 6 represents a conceptual diagram of IaC, where a developer creates a configuration template file and communicates it to the cloud provider. The cloud provider receives the configuration file and creates the resources according to the values supplied by the developer.

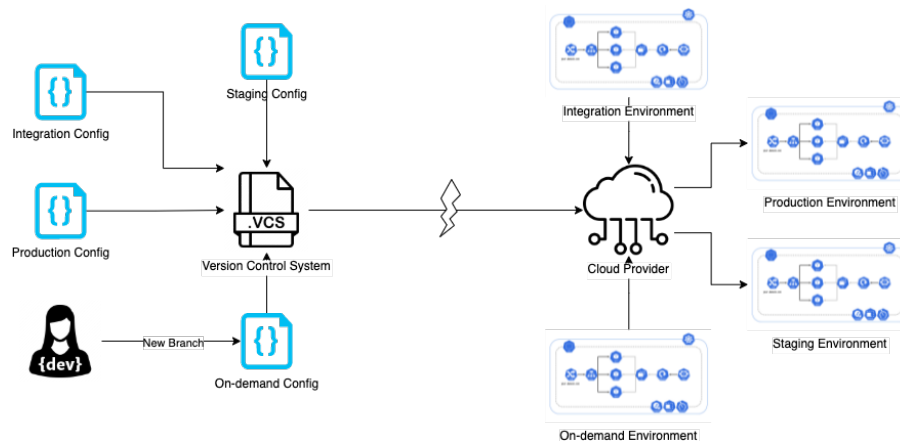


Figure 39. Infrastructure Provisioning Conceptual Diagram

The market offers the possibility of implement a solution using IaC using a large variety of tools, yet as a means of implementing a solution which applies the best practices and also maintains within the scope of the internship, the analysis made in this section will be between AWS's CloudFormation, HashiCorp's Terraform and RedHat's Ansible.

Best Practices

When implementing an Infrastructure as Code framework, the generally accepted best practices [9] are as follows:

- Treat infrastructure as developers treat code (defined format and syntax);
- Declarative configuration;
- All code stored in a source control system;
- Model all production in code;
- Resource monitoring.

When developing this component of the framework, this will be list of practices used to implement the various resources within the infrastructure.

HashiCorp's Terraform

Terraform is an open-source Infrastructure as Code software tool, developed and maintained by HashiCorp [10]. With this tool, users define and provide data centre infrastructure to developers by using a declarative configuration language known as HashiCorp Configuration Language (HCL) or via JSON objects. This tool acts as an abstraction layer over the existent cloud infrastructure provisioning tools' languages, acting as a translator between these said languages and the developer, facilitating the process of provisioning infrastructure.

Terraform offers the following features:

- Open source;
- Multi-cloud integration;
- Supports both HCL and JSON.

AWS CloudFormation Templates

AWS CloudFormation is an Infrastructure as Code service that allows for modelling, provisioning, and managing both AWS and third-party infrastructure resources [11]. What CloudFormation does is allow developers to create an infrastructure through the use of code files by using YAML or JSON languages. It also offers sample templates as a starting point for infrastructure creation.

CloudFormation's more noteworthy features are:

- Not open source, but is free to use;
- Works best with AWS Services;
- YAML and JSON;
- Extensive documentation.

AWS's CloudFormation Templates are a very powerful tool when looking to provision infrastructure for an application, whilst also not being limited to this functionality. The use of CloudFormation Templates to provision infrastructure within AWS is completely free of charge, whilst it also offers a free-tier eligibility for the first 1000 operations that are carried out with third-party resource providers. In this internship, given that all the infrastructure will be created by using AWS resources, it can be considered that there are no costs associated with the use of CloudFormation Templates, yet it is not an open-source tool.

RedHat Ansible

Ansible is the cloud infrastructure configuration tool offered by RedHat, Inc. Similar to Terraform, what Ansible offers is an abstraction layer to the infrastructure provider's traditional declarative language, looking to simplify the process of creating an application infrastructure through the use of "Playbooks". "Playbooks" are a script file which declares which components must construct the infrastructure, allowing the declaration of these elements to be done in a key-value sequence.

Ansible's relevant features toward this internship are the following:

- Open source;
- Multi-cloud integration;
- YAML language support.

Experimentation

When experimenting, two experiences were conducted to evaluate each tool. Firstly, each tool was used to create simple Elastic Cloud Computing (EC2) which would run a test python application docker container. This application would open a service listening on port 8000 and would return an HTML file with the hostname and time at which the request

was fulfilled. The second experimentation was the deployment of a simple counter application into an AWS Simple Storage Service (S3) bucket. The amount of support offered when creating the necessary configuration to deploy the application straight to the bucket would be the determining factor of understanding which tool best performed the task.




When testing this process with Terraform, I felt that the documentation was not clear about the various configurations that could be applied to the EC2 instance, and therefore limited the interaction with the cloud provider. The syntax, although similar to JSON, was not easy to comprehend at first and required a number of trial-and-error events to deploy the application. Regarding the second test case, having experimented previously with the language, understanding how to provision the components was easier, yet once again the lack of configuration possibilities in the documentation was noteworthy.

Ansible and CloudFormation Templates offered a very similar experience when creating the infrastructure. Both offer the possibility of using YAML as the configuration language, both follow a similar means of declaring the components that will constitute the infrastructure itself, with Ansible lacking slightly in terms of documentation, and both tools offer a template verifier which guarantees that syntax errors and misconfigurations are detected before deploying the file itself.

Solution Analysis

The contents present in Table 2 present a more structured analysis of the tools considered for the phase related to infrastructure provisioning.

Table 22. Infrastructure Provisioning Tool Analysis

	Open Source	AWS Integration	Multi-cloud	EKS Support	Rolling Updates	Documentation
 AWS Cloudformation	✗	✓	✗	✓	✓	✓
 Terraform	✓	✓	✓	✓	✗	▲
 Ansible	✓	✓	✓	✓	✓	▲

The experimentation process involved evaluating and comparing the most common open-source tools with the AWS equivalent. After carrying out the experimentation process, the tool chosen for Infrastructure Provisioning was AWS CloudFormation. The extensive documentation and support for developers is the main contributing factor, as it simultaneously offers several functionalities such as state management, rolling updates and rollbacks Out-of-the-Box, along with an extensive documentation and several

examples of code snippets which can be used as a starting point for more complex computational components.

Continuous Integration

This section will serve to better understand what the concept of Continuous Integration is and how it is relevant to the internship's project, along with the problems that the internship looks to provide a solution for. The following figure 8 shows a graphical representation of the workflow involved when creating the Continuous Integration phase of a CI/CD pipeline, which extends from the software developer to a docker image repository, with intermediate steps such as automated image building, functional tests, security verification and validation, and static code analysis. The following Figure 7 presents the workflow involved in a generic Continuous Integration phase of a pipeline, where the developer pushes the changes made to the source code to a Version Control System, and a code build task is carried out. This code build task involves performing static code analysis, security tests and dependency validation, and functional tests. If all is successful, a docker image is built and sent to an image repository.

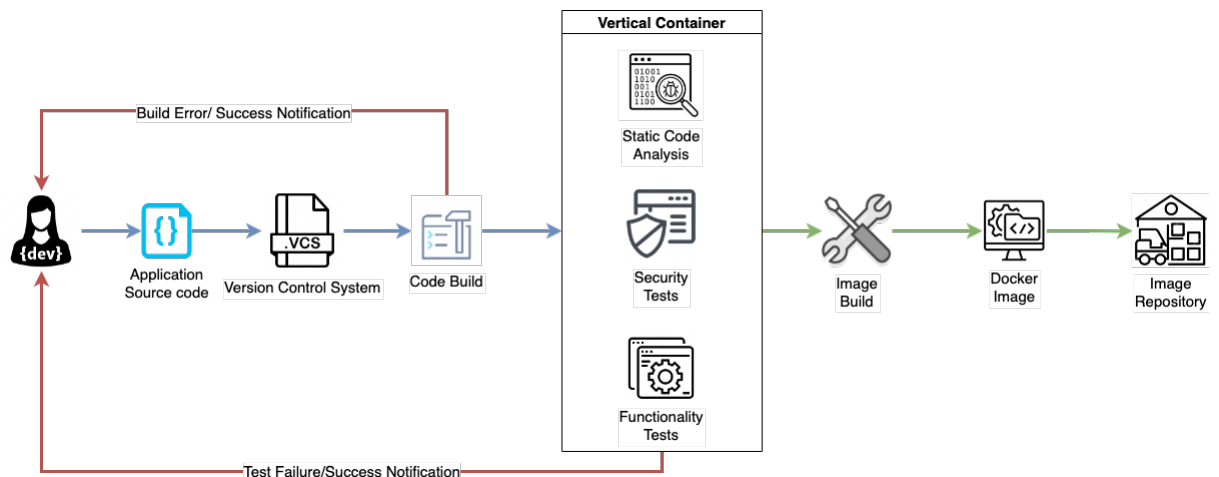


Figure 40. Continuous Integration Conceptual Diagram

In the scope of the internship, Continuous Integration looks to fulfil the need for a faster time-to-market approach when producing software. CI allows development teams to be continuously producing new code and features for an application on a daily basis, without the need for delays with time consuming tasks such as application builds and unit tests. Here, instead of manual, step-by-step testing, all the functionalities, security protocols and code convention measures are guaranteed through an automated sequence of tasks that, that traditionally could stretch out over a 6 month or one year period. In the case of CI, the application's features and source code are updated daily, and a new version of each service is created so that a deployment can be made every day, if it is required to do so.

Best Practices

When implementing a Continuous Integration component within the framework, the generally accepted best practices [12] are as follows:

- Source control system as single point of truth;

- Perform automated tests;
- One build, many steps to achieve the build;
- Perform tests on functionalities which give the most feedback;
- Maintain environment components;
- Monitor build efficiency.

When developing this component of the framework, this will be list of practices used to implement the various resources within the phase responsible for Continuous Integration.

Jenkins

Jenkins is an open-source automation server which allows for automating tasks that compose the building, testing, and deployment phases of software production [13]. Jenkins allows developers to perform static code analysis, unit tests, security dependency verification and validation autonomously, in order to guarantee that a new code commit does not “break the build”. If the new code does not pass the defined tests, the developer who produced the code is notified that there was an error, and the build does not advance. If the tests pass the build phase, a docker image is built and proceeds to the delivery phase.

The main specifications of Jenkins are:

- Open source;
- Multi-cloud integration;
- Supports multiple repository sources for code storage;
- Compatible with a wide variety of tools, including CodePipeline, but requires plugins to achieve integration;

AWS CodePipeline

Amazon’s solution for Continuous Integration (and, subsequently, Continuous Delivery) is AWS’s CodePipeline tool. Considering that CodePipeline offers the possibility for defining tasks both for Continuous Integration and Continuous Delivery, for this sub-chapter, merely the solution for Continuous Integration will be considered [14].

AWS CodePipeline offers the possibility of integrating a source code repository from a number of providers including GitHub and AWS CodeCommit. Once integrated with a source provider, the tool makes use of another tool offered by AWS which is AWS CodeBuild for carrying out tasks such as code analysis, unit testing and dependency verification. AWS CodeBuild is the automation server used for following and executing the tasks present in a *buildspec* file, included in the source repository. If this file’s commands are successfully executed and the application complies with the various defined tasks, an artifact is created and then forwarded to a component where the produced artifact is stored. In the case of this internship, it will be a Docker Image Container, pushed to an AWS Elastic Container Registry.

AWS CodePipeline’s more noticeable functionalities are:

- Full integration with other AWS tools;
- Allows the use of Jenkins as a build provider;
- Compatible with various Version Control Systems;

- Offers pre-built plug-ins with full support;

Being an AWS-native tool, CodePipeline is limited to the Amazon environment. When looking to maintain an agnostic solution for code build automation, this is not a viable solution. In other words, if an organization looks to migrate to a different cloud-provider, the configuration process must be repeated. AWS CodePipeline does offer integration with multiple VCS repositories, although when looking to go beyond the use of AWS's CodeCommit or GitHub, the configuration process is somewhat troublesome, requiring a much more complex configuration.

To conclude, AWS CodePipeline is not an open-source tool and therefore its use is billed. The first 30 days of experimentation are offered free of charge as a form of encouraging users to adopt the tool and understand its various functionalities. Once the free-trial period concludes, each active pipeline has a cost of \$1 per month. The cost is not limited to this, as each integration of other tools such as AWS CodeBuild, for example, also have their own billing structure which adds onto this.

GitLab CI/CD

GitLab CI/CD is a tool for software development using the continuous methodologies by automatically building, testing, deploying, and monitoring applications [15]. More specifically for this section, GitLab CI is a software automation tool which is provided alongside GitLab's Version Control System. This tool makes use of the Jenkins engine to automate the build process and conduct autonomous tasks which accelerate the process behind software development.

The main specifications of GitLab CI are:

- Open source;
- Multi-cloud integration;
- Supports multiple repository sources for code storage;
- Compatible with a wide variety of tools, including CodePipeline, but requires plugins to achieve integration.

The GitLab CI tool was considered as a possible solution for the CI/CD pipeline due to the fact that part of WIT Software's software code repositories are hosted on a local GitLab enterprise server.

CircleCI

CircleCI is a software automation tool, used by large enterprises such as Spotify, Coinbase and BuzzFeed and is therefore a reference in the market of autonomous software development.

The main specifications of CircleCI are:

- Open source;
- Multiple language and platform support;
- Resource control and management;
- Multi-cloud support
- Compatibility with cloud provider native tools is easily achieved.

CircleCI is an open-source solution which offers a wide variety of possibilities regarding the build and static testing of software. One of the few drawbacks of this tool is the lack of out-the-box features provisioned by the tool itself. Where tools previously mentioned offer different runtime environments along with pre-installed packages such as python language support, docker runtime engine, java runtime engine, etc, CircleCI requires these be provisioned and installed when the build is being performed. When looking to develop applications which require a large amount of dependencies, these tasks will increase build time and slow down development.

Experimentation

When looking to understand how each of these solutions functioned, I evaluated the complexity required to create a pipeline which would use an existing repository, conduct a static code analysis, unit testing and finally security verification of the module's dependencies. Once these tasks have completed, the code will be built into a docker image and deployed to an AWS Elastic Container Registry. All of these tools make use of a configuration file which defines what is expected from the pipeline and which tasks will be carried out. The elements which will be monitored are:

- Build time (Average time of 50 builds);
- Build state notifications;
- Documentation.

Jenkins performed an average build time of 3 minutes and 42 seconds from the source repository to pushing the new docker image to the elastic container registry. Email notification integration is natively supported by Jenkins where the configuration is somewhat straightforward. The user only needs to provide an email address and an SMTP authentication method. The message received is the build number, the status of the build and a link to the project execution.





AWS CodePipeline had an average build time of 2 minutes and 37 seconds from source to registry. There is no email notification system supported natively as to perform this, the AWS CodeBuild tool must be associated with an AWS Simple Notification Service which receives the message from AWS CodeBuild and then forwards it to an email endpoint. This to say, it is possible to create an email service which delivers the build's state, but it is not native to AWS CodeBuild itself.

Circle CI averaged its build times around 3 minutes and 10 seconds from source to registry. There is a native notification service, where configuration is limited to associating a build project to a specific endpoint. This solution is very attractive, especially given the more complex functionalities that it offers beyond the basic "build and deploy", but the limited documentation leaves me somewhat reluctant to adopt the tool itself.

Solution Analysis

The content of Table 3 presents the different opportunities that each tool offers to create a solution for the problems associated with Continuous Integration and the application of the various best practices associated with this concept.

Table 23. Continuous Integration Tool Analysis

	Open Source	AWS Integration	Multi-Cloud	Notification Service	Documentation	Plug-in Support
 Jenkins	✓	✓	✓	✓	▲	✓
 AWS CodePipeline	✗	✓	✗	✗	✓	✓
 GitLab CI	✓	✓	✓	✓	✓	▲
 circleci CircleCI	✓	✓	✓	✓	▲	✓

Considering that the given documentation and support is an aspect that weighs strongly in favour of the choice of the tool, Jenkins, which operates on a strong plugin basis, requires that those same plugins offer appropriate documentation to assist with developing the framework. These same plugins themselves are not verified and their documentation is somewhat limited. CircleCI does offer a strong campaign when looking to choose the tool responsible for CI in the framework, but its documentation also does lack to an extent, with limited examples. On the contrary to this, AWS's CodePipeline manages to offer these exact features, whilst having to compromise on the solution not being open source and needing manual configuration for the notification service. This compromise means having a more accessible and informed tool available, and therefore, the technology used to conduct the tasks of Continuous Integration will be AWS CodePipeline.

Continuous Delivery

In this section, the concept of Continuous Delivery will be further developed. There will be an initial definition of the concept itself, followed by an explanation of the needs that the framework will require from the standing point of Continuous Delivery. Furthermore, an explanation and comparative analysis of the solutions that exist for these issues will be made to establish which is the best tool for providing a solution for the problems that compose the internship.

Continuous Delivery has an influence in this internship due to the fact that the changes implemented by the developers need to be deployed into their production environment continuously and progressively. This task could be done manually and with all the alterations to the code and new functionalities being delivered to the entire user

population at once, requiring a full deployment plan, during low-traffic hours and with a large level of human resources being available to carry out the delivery. This is not a viable solution when wanting to guarantee that a new feature or design change is accepted by the application’s users. Given that the idea behind DevSecOps is to perform multiple commits and new versions of software, multiple times a day, this method would be a waste of resources. Moreover, giving users access to an entire new version of an application translates into a more complex strategy when a rollback needs to be performed. The CD module of the pipeline will perform phased rollouts, following these methods of deployments:

- Blue/Green Deployments;
- Canary Deployments;
- A/B Testing

Phased rollouts are a method used to deliver new system implementations and alterations in an incremental fashion. This allows an organization to react accordingly if an issue arises whilst also allowing users to adjust to changes gradually. These methods of deployment are further explained in the document titled Appendix A – State of the Art. For a question of readability, this document will only present the Blue/Green Deployments and Canary Deployments methods.

Blue/Green Deployments are an implementation strategy of a phased rollout. In this case, it uses at least two environments which are as identical as possible, each running a version of the application or service [16]. The application’s traffic, which was originally targeted towards the previous version (blue), is then routed to the second version (green) and acceptance and quality assurance tests are run to guarantee that the new version is ready to receive users. Once the deployment is successful, the green version can then replace the blue. This process is presented in the form of a diagram in Figure 8.



Figure 41. Blue-Green Deployment [16]

A canary deployment is a deployment strategy that releases an application or service incrementally to a subset of users [16]. All infrastructure in a target environment is updated in small phases (ex: 2%, 25%, 75%, 100%). A canary release is the lowest risk-prone, compared to all other deployment strategies, because of this control. The following Figure 9 shows a graphical representation of how a canary release is applied, with the routing of traffic between two application versions.

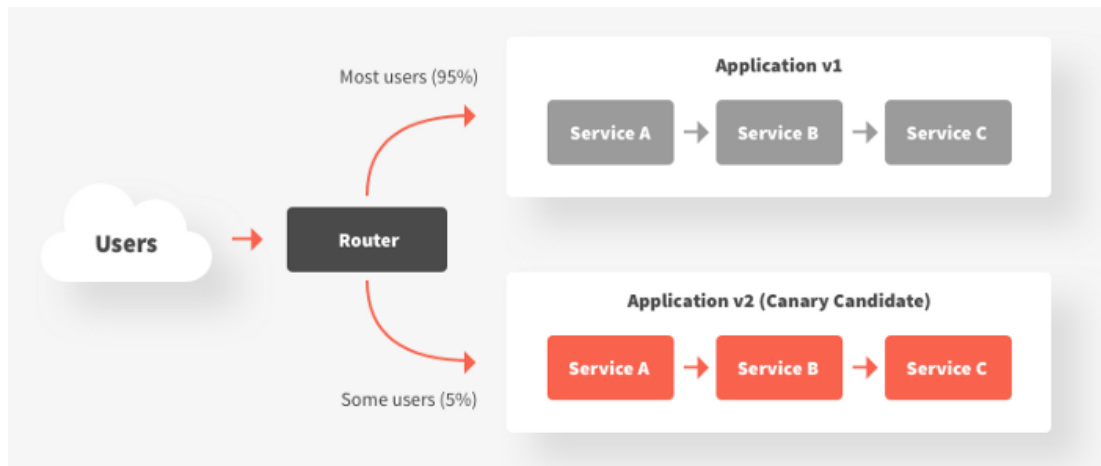


Figure 42. Canary Release Workflow [17]

The pipeline must offer the possibility for these multiple deployment strategies, each being chosen and configurable according to the context into which they are applied. The solutions taken into study which offer a possibility for CD are GoCD, FluxCD, Argo CD and AWS CodeDeploy which is integrated into the AWS CodePipeline tool.

Best Practices

When implementing the Continuous Delivery component in the framework, the generally accepted best practices [12] are as follows:

- Deploy to every environment the same way;
- Inject purpose faults to test pipeline recovery;
- Automate the deployment and rollback process;
- Version control as single point of truth;
- Monitor pipeline statistics.

When developing this component of the framework, this will be list of practices used to implement the various resources within the phase responsible for Continuous Delivery.

GoCD

GoCD is an open-source software delivery tool which is used to help teams automate the process of delivering software to users [18]. GoCD operates based on plugins developed by the community to perform the various tasks necessary to deploy an application into production. GoCD follows a client-server model where the server is running locally on the developer's machine and is responsible for polling the repository for changes. When a change is detected, the corresponding pipeline is triggered.

GoCD offers the following functionalities as valuable for the internship:

- Open source;
- Pipeline Configuration is made using YAML;
- User Management;
- Allows Pipeline Bridging;

- Multi-cloud support;
- Native Kubernetes and Docker support;

FluxCD

FluxCD is a toolset of Continuous Delivery solutions, specializing in Kubernetes and containerized applications [19]. This tool allows for deploying applications into Kubernetes clusters with canary releases, A/B Tests, and blue-green deployments. Exactly the three requisites of the intended product of this internship. Given a YAML manifest file kept in a VCS repository, FluxCD periodically verifies if there are any changes made to the configuration file of the deployment. If there is a change in the configuration, the cluster is updated and synchronized according to the changes made. The current configuration file is kept in cache and used as a term of comparison to verify if any changes were made.

FluxCD's main specifications are:

- Configuration via YAML;
- Command-line tool;
- Out-of-the-Box integration with GitHub;
- Direct integration with EKS;
- Performs bootstrapping within a Kubernetes cluster;

Argo CD

Argo CD is a declarative continuous delivery tool created for the deployment of Kubernetes clusters and microservice applications [20]. This tool's core component is the Application Controller which is responsible for monitoring the current state of an application and compares it to the desired target state that is kept in a Git repository. The existence of this controller allows for automated deployments where the desired application state is pushed into the cluster automatically, a task previously triggered by a Git commit or, in the case of this internship, by a CI pipeline which notifies that a new version of the application is available for deployment.

Argo CD also offers a graphical user interface which brings application state visibility to the developer, whilst also providing a command line tool which can be adopted by more experienced users. Given that an application cluster can have multiple pods running with a number of distributed communications being performed amongst each other, the possibility of having a visual representation of the state of the application brings an increased amount of value to the use of Argo CD as a solution for Continuous Delivery.

Argo CD brings value to the internship with:

- Configuration via YAML;
- Command-line tool;
- Out-of-the-Box integration with GitHub;
- Direct integration with EKS;
- Performs bootstrapping within a Kubernetes cluster;
- Graphic User Interface;

AWS CodeDeploy

AWS CodeDeploy is a fully managed deployment service that automates software deployments to a variety of computational services such as Amazon EC2, AWS Fargate, AWS Lambda, and on-premises servers. AWS CodeDeploy makes it easier to rapidly release new features, helps avoid downtime during application deployment, and handles the complexity of updating applications. AWS CodeDeploy can be used to automate software deployments, eliminating the need for error-prone manual operations.

AWS CodeDeploy brings value to the internship with:

- Configuration via YAML;
- Command-line tool;
- Out-of-the-Box integration with AWS CodePipeline;
- Graphic User Interface with AWS Console;

Experimentation

Given the fact that GoCD does not offer the possibility of integrating phased rollouts, it was not considered in the experimentation phase of the tools involved with Continuous Delivery. Therefore, the comparison made was between FluxCD, Argo CD and AWS CodeDeploy.


The learning curve associated with the three tools that were analysed varied to an extent. Argo CD and Flux CD were the more balanced and understanding their functionality and visualizing the information regarding each deployment was straight forward and, initially did not require an extensive navigation. AWS CodeDeploy did offer several difficulties when performing a canary deployment, requiring that a lambda function be used to process changes and replace the versions of the images running in the cluster.




Essentially, the three tools offer very similar solutions, with the differentiating factor being the GUI that is offered with Argo CD and AWS CodeDeploy. With this in mind, the decision between the tools depended on the experience and feel given when using each solution, with the technical factors being somewhat identical. Also, from a cost point of view, AWS CodeDeploy does have the setback that it is billed according to the amount of deploys performed. This factor reduced the choice to Argo CD.

Solution Analysis

The content shown in Table 4 performs a graphical comparison between the various solutions possible with each of the competitors that were previously analysed.

Table 24. Continuous Delivery Tool Analysis

	Open Source	Blue/Green Deployments	Canary Releases	Notification Service	GUI	Documentation
 GoCD	✓	✗	✗	✓	✓	✗

 FluxCD	✓	✓	✓	✓	✗	▲
 Argo CD	✓	✓	✓	✓	✓	✓
 AWS CodeDeploy	✗	✓	✓	✗	✓	✓

GoCD is a powerful tool for performing automated tests, executing tasks on an application, and deploying software, yet it does not allow the implementation of phased rollouts into its delivery model. Given that this is a compulsory requirement for the final product, this cannot be considered a possible solution for the problem at hand.

AWS CodeDeploy was considered for discussion when analysing the different solutions but, given a larger tendency of the informatics community to adopting FluxCD and Argo CD for Continuous Delivery solutions, the possibility of adding another billed service to the pipeline made the solution much less attractive. Given the fact that other open-source solutions offer very similar functionalities as AWS CodeDeploy, tool was not excluded from the possible choices for the framework.

Being FluxCD and Argo CD the two remaining possibilities, whilst sharing many characteristics and both following a very similar form of operation, Argo CD was found to be the more interesting choice. This is due it also offering a Graphical User Interface, which offers a larger visibility to the state of each container and if the system is in the current desired state or if alterations are necessary. Although a basic reason to differentiate between the two tools, the most noteworthy factor between the two is, in fact, the existence of the GUI.

Monitoring Microservices

This section explains what Application Monitoring is and what challenges the exist when deploying a microservice application into a cluster. There will also be an analysis of what tools offer a solution for the questions raised and which best suits the questions approached by the internship's proposal.

Figure 10 presents a graphical representation of the concept of monitoring microservices. The Kubernetes server and the applications running on it are queried for logs and metrics, which DevOps team members can analyse to establish what components can be improved, and if there are any errors occurring that may have gotten through the CI/CD phase, and therefore need to be corrected.

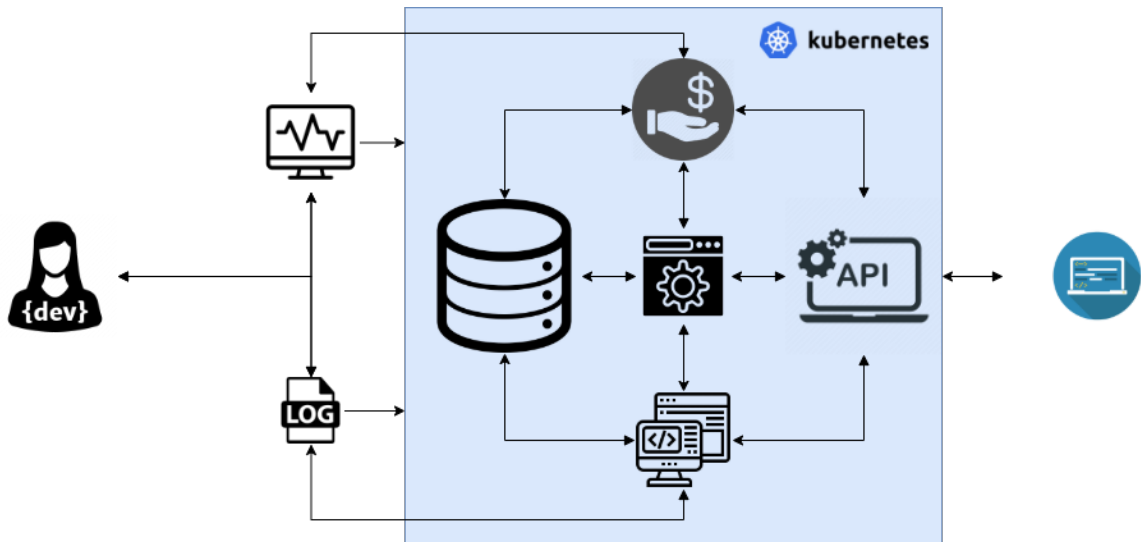


Figure 43. Microservice Monitoring Conceptual Diagram

In order to gain application observability, a monitoring strategy must be implemented that extracts metrics and logs produced by each of the services. Monitoring solutions provide a visual means to observe how the various events within each service are connected and how they behave throughout the production environment. To obtain this information, the best practices for application require that the following components be measured in the cluster, present in Table 5:

Table 25. Microservice Monitoring Best Practices

Metrics	Description
Cluster nodes	Number of nodes that are available. Gives an overview of the resources needed to run the cluster.
Cluster pods	Number of pods running in the cluster. Gives feedback on if there are sufficient nodes for the workload.
Resource utilization	Measures the cluster's memory, CPU, bandwidth and disk consumption.
Container Metrics	Measures the network, CPU and memory usage in a container.
Application Metrics	Measures the business logic metrics such as the number of users accessing the application, user experience, etc.
Kubernetes Scaling and Availability Metrics	Allows to manage how the orchestrator handles each pod, specifically. This includes pod health checks, network data transfer, on-progress deployment, etc.

Best Practices

When implementing the Monitoring component in the framework, the generally accepted best practices [12] are as follows:

- Monitor containers and their various processes;
- Retrieve metrics and alert on service performance;
- Prepare for scalability;
- Monitor application API;
- Perform mapping between application information and organization structure.

When developing this component of the framework, this will be list of practices used to implement the various resources within the phase responsible for Application Monitoring.

AWS CloudWatch

Amazon's solution for application monitoring and observability is AWS CloudWatch. It provides information and insights into an applications functionality and operation, in order to maintain a backlog of applications, whilst allowing to perform actions on the data extracted such as system performance changes and resource optimization. CloudWatch is a unified and centralized information source for obtaining and registering application behaviour [21].

CloudWatch's specifications are as follows:

- Single Observability Platform;
- Performance and Resource Optimization;
- Operational Visibility Direct integration with EKS;
- Derive insights from logs;
- Integration with AWS Tools;

ELK Stack

ELK Stack is a search engine which provides the possibility of extracting, indexing, and presenting application logs. It is a stack composed of three tools: Elasticsearch, Logstash and Kibana [22]. The ELK stack has been updated to support Kubernetes log analysis, maintaining Kibana and Elasticsearch as the visualization tool and centralized search engine for log persistence, respectively. For Kubernetes support, ELK evolved into Elasticstack, where instead of using Logstash as the ideal tool for monolithic log analysis, Elasticsearch introduces the concept of "Beats": lightweight, single-purpose data shippers, which extract specific details about microservices and export them through to Elasticsearch.

The main specifications offered by the ELK stack are:

- Open source;
- Near real-time data extraction;
- Extensive plugin library;
- Plugin verification and evaluation;
- Extensive documentation;

Prometheus and Grafana

Prometheus is an open source and free monitoring solution software responsible for analysing and processing in information regarding application metrics in real time [23]. This tool operates according to a pull model in which the different components within an application are queried for their health, the amount of resources that are being consumed and miscellaneous statistics associated with requests made and tasks that are carried out.

At the same rate that the information is being scraped from the different applications, there must be a means of analysing and viewing the information in a natural and understandable way. The solution for this is to couple Prometheus with Grafana. Grafana is an open-source server application which provides charts, graphs and alerts for the different metrics and occurrences that take place in an application [24].

Prometheus' main value comes from the following:

- Open source;
- Almost real-time data processing;
- Customizable;
- Pull-model;
- Targets are dynamically discovered.

Experimentation

The experimentation process for the tools involved with monitoring the application has the following sequence:

- Deploy a Kubernetes cluster on AWS;
- Deploy a sample application to the cluster;
- Deploy the monitoring tool to the cluster;
- Extract information regarding metrics (Prometheus + Grafana) and logs (ELK Stack).

When using AWS CloudWatch, the possibility of having a monitoring tool already included in the cloud provider's solution made for a more complete solution, at first sight. When performing queries to the application and awaiting CloudWatch to update its dashboards to reflect the new information, a delay of around 2 minutes is present, both when analysing application metrics and the logs produced. The fact that it is paid service also contributes to not adopting the tool in the final solution, given that when one is paying for a tool, it is expected that that same tool performs at the highest and most efficient level.

Regarding Prometheus and Grafana, the deployment of these tools was simple. Working as a daemon set deployment to the application. Prometheus runs parallel to the application and extracts information from the pod, the information is extracted in real time and presented to Grafana in the same manner. Given that it is an open-source solution with a much larger efficiency, this solution is a very strong candidate.

The Elasticstack requires a more complex configuration, as there are three tools which must be prepared and orchestrated to retrieve the logging information from the application itself. This process is troublesome but given the immediacy of the information that is

extracted from the application, together with the fact that it is an open-source solution, it would seem to be the more correct option regarding application logging.

Solution Analysis

The content of Tables 6 and 7 serve as a graphical comparison between the different tools and what solutions they offer for the problems associated with application monitoring.

Table 26. Microservice Monitoring Competitor Analysis 1







	Open Source	Log Analysis	Health Monitoring	Usage Monitoring
 AWS CloudWatch	✗	✓	✓	✓
 Elasticstack	✓	✓	✗	✗
 Prometheus + Grafana	✓	✗	✓	✓

Table 27. Microservice Monitoring Competitor Analysis 2

	Memory Usage	Network Usage	Number of Pods/Cluster	Notification Service	CPU Usage
 AWS CloudWatch	✓	✓	✓	✗	✓
 Elasticstack	✗	✗	✗	✓	✗
 Prometheus + Grafana	✓	✓	✓	✓	✓

Although CloudWatch does offer a centralized solution for creating application visibility, the fact that it is both a paid service and that it does not offer an integrated notification service reduces its value for the internship. A CloudWatch Agent must be configured and deployed to perform metric extraction, along with application log streaming to operations team members. Furthermore, the lack of dashboard customizability and custom metrics

also translates into a much more limited service than that of the Elasticstack and also the junction of Prometheus with Grafana. Due to this, the opted tools for application logging will be the Elasticstack, and for application metrics observability will be Prometheus as the scarping tool together with Grafana as the visualization tool.

Appendix B - Methodology



FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE
COIMBRA
DEPARTMENT OF INFORMATICS ENGINEERING

Gabriel Marco Freire Pinheiro

Appendix B - Methodology

Dissertation in the context of the Master's Degree in Informatics Engineering,
Specialization in Software Engineering, advised by Professor Nuno Laranjeiro and
Engenheiro Rui Cunha, presented to the Department of Informatics Engineering of the
Faculty of Sciences and Technology of the
University of Coimbra.

July 2022

This page was intentionally left in blank.

Table of Contents

METHODOLOGY	I
AGILE AND SCRUM METHODOLOGIES.....	I
PROJECT PLANNING.....	IV
RISK MANAGEMENT.....	X

Methodology

This project is going to be integrated into an existing framework and development team's organization tool at WIT Software. To achieve that, an objective planning process must be orchestrated to guarantee that both the internship and the final integration into the enterprise's organization are both a success. The planning is done according to an Agile methodology, based on Scrum, an interactive and incremental software development structure. In this internship, "*Scrum by the Book*" will be the chosen methodology and organization sequence.

This section will be divided into five main sections, beginning with a brief explanation of the agile principles, followed by An explanation of Scrum and its roles, together with an explanation of the methodology and how it works. A graphical representation of the work plan will also be provided, concluding with an explanation of the risks that pertain to the project, with their respective mitigation strategies.

Agile and Scrum Methodologies

An Agile methodology follows a number of principles that define its culture and serve as a form of differentiation from other methodologies. These principles are the following:

- Individuals and Interactions over processes and tools;
- **Working Software** over comprehensive documentation;
- **Customer Collaboration** over contract negotiation;
- Responding to Change over following a plan.

These principles solver various risks that can occur throughout the software development process. This section will look to give an explanation into the methodology used for the development of the product in this internship. More specifically, the Scrum methodology.

Scrum Definitions

When looking to evaluate a projects complexity, using a Stacey Complexity Matrix gives a good understanding of how close a team is to agreement on what requirements are needed in the project in relation to how certain the team is about the strategy used to implement them. There are three main profiles present when classifying a project according to its complexity, according to Santiago Obrutsky [36]:

- **Simple:** project that are easy to understand and with a simple concept. These are known in their entirety and the developers are 100% sure of what needs to be done and which technologies to use;
- **Complex:** projects with an intermediate level com complexity, requirement knowledge is adequate, as well as the technologies that are required for the project;

- **Chaos:** projects that are very difficult to conclude. Requirements are not understood at all, nor is there knowledge of which technologies to use for development.

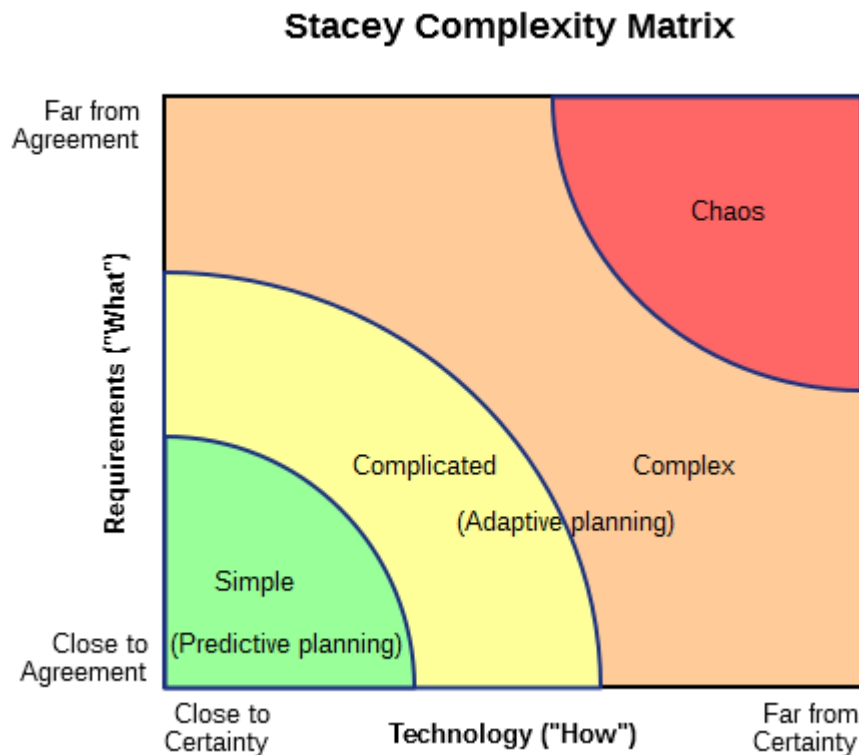


Figure 44. Project Complexity [37]

There is still another level of complexity, which is the area of complex types. This type of project occurs when the number of requirements is too large, with the number of possible solutions also being considerably extensive. In these situations, the system is referred to as a system that is not fully predictable.

Given the number of requirements that are expected for this project, along with their respective solutions, a waterfall methodology is not recommended, and therefore an Agile methodology is the ideal option. The flexibility offered is much larger, which allows the developer to adapt accordingly throughout the product's development process.

Scrum allows the necessary agility for this type of project, given the number of requirements that the framework has, together with the possibility for possible alterations to the product itself. If these suffer changes, there is a certain level of complexity which will require discipline and organization on behalf of the developer.

Scrum Roles

The Scrum methodology has a set of roles what represent those who have a level of commitment towards the project that is being developed. These roles are separated into three definitions: Project Owner, Scrum Master, Development Team.

Product Owner - the Project Owner is the person responsible for analysing business activities, customer communication and management, and product guidance. Given this, their responsibilities fall into the following list:

- Represent and manage the interests of the Stakeholders;
- Own the Product Backlog;
- Establish, nurture and communicate the vision of the product;
- Compares and monitors the project's behaviour, goal and investment;
- Makes decisions regarding when official releases are made.

This role must also ensure that the development team brings value to the project, that requirements are ranked by priority and met by the team, according to the customer's needs. The Product Backlog is a comprehensive and cooperative list of tasks that must be concluded, to which all team members and the project owner can contribute towards.

In the case of this internship, the Product Owner is Mr. Rui Cunha, who also carries out the function of being the project manager. Given his knowledge of the project's scope and the different means possible for achieving a viable solution, the role is well filled to guarantee the project's success.

Scrum Master - the Scrum Master is the element responsible for maintaining a healthy and comprehensive development team. The team's success is defined by this element, being that in the event of problems occurring among team members, they are responsible for dealing with these issues and guaranteeing that all goes according to plan. Mr. Diogo Cunha will be the orientator that is responsible for this task.

Development Team - the Development Team's purpose is to carry out the tasks defined in the Product Backlog. In this internship, the development team is composed exclusively by myself, and therefore my responsibilities are to manage and fully conduct the work which is presented to me. I must also be able to develop the various functionalities of the product, giving the due time to those which have higher priority levels.

How Scrum Works

The Scrum methodology follows an iterative development process where each work plan varies according to the duration of each feature that is to be developed. The Product Backlog is composed of a set of use cases, each having a different level of priority and complexity that allows the iteration of tasks. This also represents the work that has to be carried out throughout the internship.

Specifically in this internship, each iteration will be defined by sprints of 2 weeks, along with daily meetings with the Project Owner and Scrum Master where the following questions will have to be reported:

- What has been done since the last meeting?
- What will be done tomorrow?
- What challenges or issues do I have?

The end of each sprint will present the results and their development proceedings, allowing an adjustment to the plan if there is the need to do so. When closing a work plan, not only do the features need to be implemented, but there must also be an explanation as to what was done to achieve the goal in question. This process is explained in Figure 12.

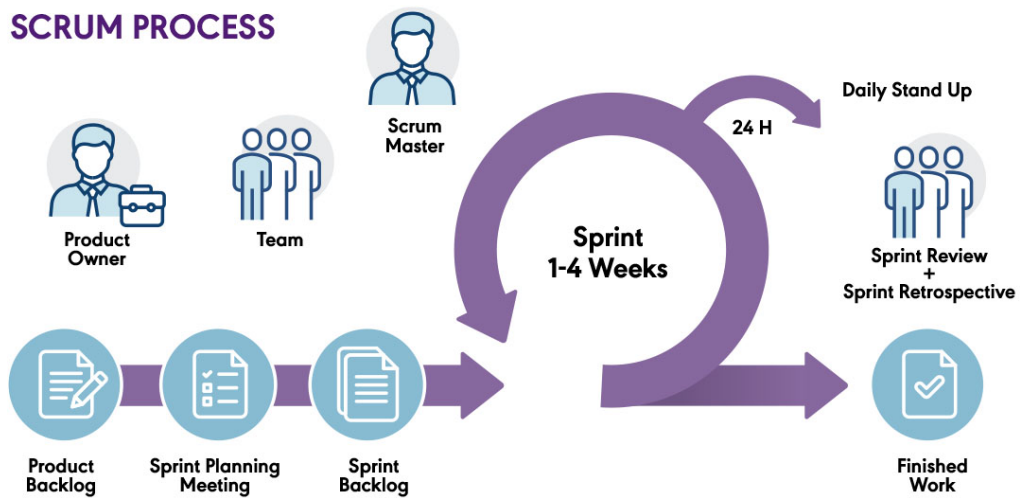


Figure 45. Scrum Overview [38]

The Backlog’s organization is of the Product Owner and the Scrum Master’s responsibility. In theory, this is usually conducted by the Product Owner exclusively, but in the case of this internship, the Backlog will be created and managed by myself, and approved by the other two members, previously mentioned.

In summary, the development process begins with an implementation cycle of two weeks, along with daily meetings with the Scrum Master in order to relay how the work is being carried out in the project and if there are any corrections that need to be made. At the end of each cycle, there is a meeting with the whole team to perform an analysis of the work that was done, what may have gone wrong, and what corrections need to be made to achieve a better result at the end of the next sprint. At the conclusion of these two weeks and once the changes are duly applied, the next iteration of development begins, and the next sprint’s process is initiated.

Project Planning

The project’s planning structure is intertwined with the Scrum approach in the sense that a Product Backlog will have to be made before development and, to achieve that, a good understanding of the framework requirements is necessary. Having this defined, the project’s goals are defined in a more global outline. From here, the Product Backlog will be altered and have more tasks added as the various requirements are prepared and analysed in more detail.

The Product Backlog’s main objective is to answer the questions “who?”, “what?” and “why?” regarding the requirements, in a simple and straightforward manner. An example of this can be the following: “As a developer, I want to add the possibility to create a new container image which will update and replace an existing image in a Kubernetes cluster, running on an AWS EKS”.

In this requirement, the questions are answered as follows:

- Who: “As a developer (...);”
- What: “(...) add the possibility to create a new container image (...);”

- Why: “(...) update and replace an existing image in a Kubernetes cluster, running on an AWS EKS”.

Each task belongs to a sub-category and has a deadline associated to it. It will follow the following format:

Table 28. Task Table

Category	Sub-Category	Task	Deadline
Implementation	Continuous Integration	Develop method to create infrastructure on command.	5 th February 2022
...

The definition of these components in a task, understanding what is expected with this activity is simple and easy. The deadlines are based on the priority level given by the Product Owner and according to the difficulty of the task.

The Product Backlog will show all the features that are required by the final product. However, given that the final artifact is produced throughout the entire duration of the project, each iteration must have its own definition of a work plan, which will imply that some changes be made to the Product Backlog at the end of each task. The Product Owner and Scrum Master define the work plans which must occupy the previously mentioned 2 weeks.

The tasks that are integrated into each work plan are chosen throughout the duration of the project. Given this, when a task comes to its end, a new work plan will have to be established for the following task. Due to the fact that the organization is the one responsible for creating the project itself, some changes may occur, depending on which are the more prioritized activities at the given moment.

Internship Project Organization

The project’s organization structure defines the different milestones that must be achieved to maintain the project on track and mitigate the possibility of delays occurring. The division of the different tasks allows to explain and establish a threshold to consider a sprint successful. In the case of the internship, the project will have the following organization:

Project Lifecycle:

The following Gantt diagram represents the planning of the various tasks and sprints that are to be conducted during the second semester:

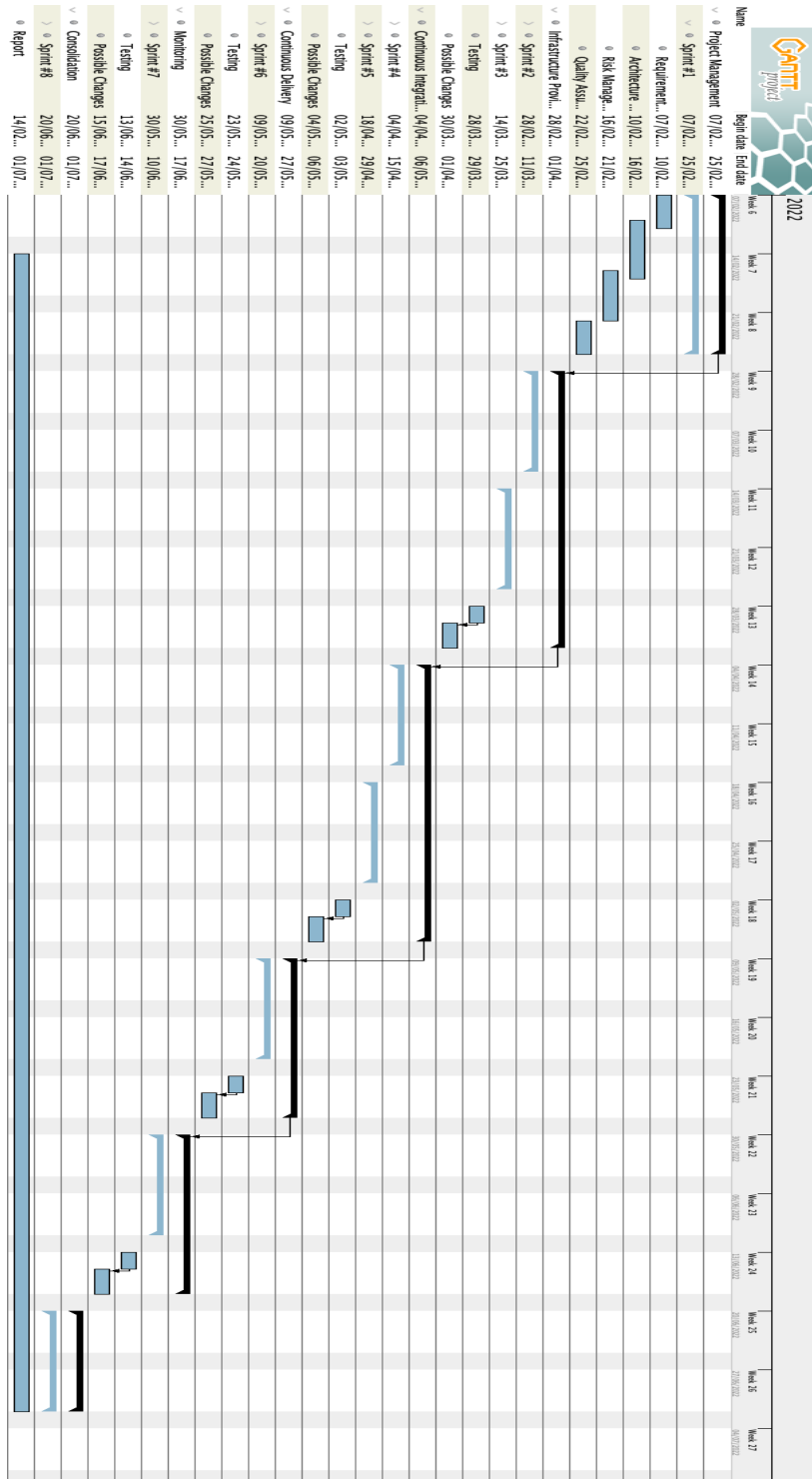


Figure 46. 2nd Semester Overview

Table 29. Project Lifecycle Milestones

Phase	Sub-phase	Epic Start	Epic End
Project Planning	-	07/02	25/02

Software Development	Infrastructure Provisioning	28/02	01/04
	Continuous Integration	04/04	05/05
	Continuous Delivery	09/05	27/05
	Application Monitoring	30/05	17/06
Consolidation Tests	-	20/06	24/06
Project Revision	-	27/06	01/07
Thesis Delivery	-	07/02	04/07

Deliverables:

Table 30. Deliverables 1

Phase	Due Date	Phase	Due Date
Developed Framework	04/07	Acceptance Test Plan	25/02
Quality Assurance Plan	25/02	Architecture and Design	25/02
Software Requirements Specification	25/02	Acceptance Test Report	04/07
Risk Plan	25/02	Quality Assessment Report	04/07

Table 31. Deliverables 2

Phase	Due Date	Phase	Due Date
Milestone 1 Report – Project Management	25/02	Milestone 5 Report – Monitoring	17/06
Milestone 2 Report – Infrastructure Provisioning	01/04	Milestone 6 Report – Consolidation Test	24/06

Milestone 3 Report – Continuous Integration	05/05		-
Milestone 4 Report – Continuous Delivery	27/05		-

Project Management:

- Eng. Rui Cunha – Product Owner;
- Eng. Diogo Cunha – Scrum Master.
- Eng. António Mendes – Guidance
- Gabriel Pinheiro – Developer;

Base Plan and Control

The Base Plan and Control defines when each task will begin, its due date and also the amount of effort that is expected to achieve the timeline that is defined. The effort is measured in hours and is based on each working day being equal to 8 hours.

Table 32. Project Management Documents

Document	Start Date	End Date	Effort (Hours)
Software Requirements Specification	10/02/2022	16/02/2022	40 Hours
Software Architecture & Design	10/02/2022	16/02/2022	16 Hours
Risk Plan	21/02/2022	22/02/2022	16 Hours
Quality Assurance Plan	23/02/2022	23/02/2022	8 Hours
Acceptance Test Plan	24/02/2022	24/02/2022	8 Hours
Milestone 1 Report	25/02/2022	25/02/2022	1 Hour

Table 33. Infrastructure Provisioning Objectives

Document	Start Date	End Date	Effort (Hours)
Demonstration 1	11/03/2022	11/03/2022	30 Minutes
Demonstration 2	25/03/2022	25/03/2022	30 Minutes

Milestone 2 Report	01/04/2022	01/04/2022	1 Hour
--------------------	------------	------------	--------

Table 34. Continuous Integration Objectives

Document	Start Date	End Date	Effort (Hours)
Demonstration 1	15/04/2022	15/04/2022	30 Minutes
Demonstration 2	29/04/2022	29/04/2022	30 Minutes
Milestone 3 Report	06/05/2022	06/05/2022	1 Hour

Table 35. Continuous Delivery Objectives

Document	Start Date	End Date	Effort (Hours)
Demonstration 1	20/05/2022	20/05/2022	30 Minutes
Milestone 4 Report	27/05/2022	27/05/2022	1 Hour

Application Monitoring:

Table 36. Application Monitoring Objectives

Document	Start Date	End Date	Effort (Hours)
Demonstration 1	10/06/2022	10/06/2022	30 Minutes
Milestone 5 Report	17/06/2022	17/06/2022	1 Hour

Table 37. Consolidation Test Objectives

Document	Start Date	End Date	Effort (Hours)
Demonstration 1	24/06/2022	24/06/2022	30 Minutes
Milestone 6 Report	01/07/2022	01/07/2022	1 Hour

Table 38. Miscellaneous Documents

Document	Start Date	End Date	Effort (Hours)
Quality Assessment Report	27/06/2022	29/06/2022	24 Hours

Acceptance Test Report	30/06/2022	01/07/2022	16 Hours
Internship Report	10/02/2022	04/07/2022	1 Hour

Risk Management

Whenever developing a large-scale project over a long period of time, the possibility of problems arising is high, and if the team responsible does not have a pre-existing plan for these problems, the project's integrity and viability may be questioned. In this case, it could lead to a potential failure of the internship. Due to this, the risks associated with this internship have been analysed together with a means of mitigating those same risks. The detection of a risk implies the creation of a strategy which, when applied, will eliminate the possibility of its occurrence.

The risks follow an order of classification which is represented in the following Table 10:

Table 39. Risk Priority Table

Classification	Description
1	Very low impact, easily overcome.
2	Low impact, implies a small change to the project.
3	Medium Impact, requires a certain level of effort to overcome, but manageable.
4	High impact, if changes aren't made, the project could potentially fail.
5	If this occurs, the project fails completely.

In order to identify each possible risk that has been identified to date, there will be a catalogue of this list where each risk will be defined by an id, type, impact, description and mitigation technique.

Table 40. Risk Identification and Impact

ID	Type	Impact	Description
1	Project Management	4	A sprint does not produce the expected result.
2	Project Management	3	Bad estimation of the duration of a sprint.

3	Technologies	3	Lack of knowledge with the tools used in the project.
4	Project Management	2	Requirement changes.
5	Technologies	4	Tool functionality/support changes.
6	Technologies	4	Organization moves to change cloud provider.
7	Technologies	5	Cloud provider dismantles.
8	Budget	2	The framework oversteps the budgeted cost threshold.

ID	Mitigation Technique	Conclusion
1	<p>Adapt the amount of team in each sprint;</p> <p>Perform daily meetings with the Scrum Master.</p>	<p>In the event of a sprint being too long, the amount of time available for disaster recovery is reduced.</p> <p>To mitigate the risk of this happening, the best way to prevent this issue is to define smaller sprints, coupling these with daily meetings with the Scrum Master and Product Owner.</p>
2	<p>- Discard optimism;</p> <p>Resort to the Scrum Master's and the Product Owner's experience;</p> <p>Stick to the tools that are chosen;</p> <p>Perform daily meetings with the Scrum Master.</p>	<p>The estimation for each sprint is made in a meeting where the Scrum Master, the Product Owner and myself will be present.</p> <p>Given my reduced experience with project estimation, the expertise of those around me will have to be the louder voice and the estimation must be done based on their opinion.</p> <p>Maintaining contact with both the Scrum Master and the Product Owner will be crucial for this problem to be mitigated.</p>

3	<p>Analyse and study the technologies before starting development;</p> <p>Communicate troubles with orientators.</p>	<p>Throughout the development of the framework, there will be a number of technologies with which I do not have the required experience.</p> <p>To overcome this issue, there will be time given to studying and prototyping.</p>
4	<p>Maintain a consistent and highly frequent delivery of artifacts to the client.</p>	<p>In the event of a requirement needing to be changed during the project's development, there will have to be a constant contact with the client and the Scrum Master to understand which changes need to be applied and how.</p> <p>This contact helps guarantee the success of the project.</p>
5	<p>Maintain contact and awareness of tool documentation and possible changes to the different APIs and frameworks.</p>	<p>Given that API updates are a reality in the community, there must be a higher level of awareness regarding the changes that may occur.</p> <p>This helps guarantee that the final product does not have errors or failures in its final state.</p>
6	<p>Maintain contact and awareness of tool documentation and possible changes to the different APIs and frameworks.</p>	<p>Given that API updates are a reality in the community, there must be a higher level of awareness regarding the changes that may occur.</p> <p>This helps guarantee that the final product does not have errors or failures in its final state.</p>
7	<p>Maintain contact with product owner to understand if this may become a reality;</p> <p>Look to use agnostic tools where possible.</p>	<p>An unlikely reality, but if the organization chooses to change their cloud provider from Amazon to another competitor in the field, there must be a means of migrating the framework to the other provider with the minimum amount of effort necessary.</p>
8	<p>There is none.</p>	<p>Although a highly improbable occurrence, there exists the</p>

		possibility of Amazon Web Services dismantling, in which case the provisioned infrastructure will no longer be accessible, and all business must be redefined and migrated to another cloud provider.
9	<p>Stop machines that are not being used;</p> <p>Clean up resources that can lead to an increase in billed values;</p>	<p>Costs are expected when using the framework but must be controlled as much as possible.</p> <p>To control these costs, the saving of computational resources is essential, especially given the fact that the project is a test environment.</p>

Appendix C – Software Requirement Specification



FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE
COIMBRA
DEPARTMENT OF INFORMATICS ENGINEERING

Gabriel Marco Freire Pinheiro

Appendix C – Software Requirement Specification

Dissertation in the context of the Master’s Degree in Informatics Engineering,
Specialization in Software Engineering, advised by Professor Nuno Laranjeiro and
Engenheiro Rui Cunha, presented to the Department of Informatics Engineering of the
Faculty of Sciences and Technology of the
University of Coimbra.

July 2022

This page was intentionally left in blank.

Table of Contents

1.	USE CASES.....	II
1.1	INFRASTRUCTURE PROVISIONING	II
1.2	CONTINUOUS INTEGRATION.....	IV
1.3	CONTINUOUS DELIVERY	V
1.4	APPLICATION MONITORING	VIII
2.	FUNCTIONAL REQUIREMENTS.....	XI
2.1	INFRASTRUCTURE PROVISIONING	XI
2.2	CONTINUOUS INTEGRATION.....	XVI
2.3	CONTINUOUS DELIVERY	XXI
2.4	APPLICATION MONITORING	XXIII
3.	NON-FUNCTIONAL REQUIREMENTS	XXVII

Use Cases

Infrastructure Provisioning

	UC-1: Create Infrastructure
Description	The developer performs a change in the application code and must test it to guarantee that it is ready to be placed into production.
Motivation	Developers must have an environment ready to implement and test changes to application code, without having an effect on the production environment.
Users	DevOps Team Members, DevOps Team Manager.
Assumptions	<ol style="list-style-type: none">1. The production environment is active;2. There is an existing VCS repository on AWS CodeCommit;3. There is an infrastructure configuration file in the VCS repository;4. The application's source code is located in the VCS repository;
Order of Events	<ol style="list-style-type: none">1. The developer performs a commit to the VCS repository;2. The VCS repository detects the configuration file;3. The VCS detects that the developer is not on the master branch;4. AWS CodeCommit Actions triggers the event associated with creating an environment;5. The configuration file is communicated to the AWS platform;6. The AWS platform creates the environment together with the defined resources;
Alternate Paths	<ol style="list-style-type: none">4.1 The configuration file has an error;4.2 The notification service communicates the error to the developer that performed the commit;4.3 The commit is rolled back;
Results	<ol style="list-style-type: none">1. The intended environment is created with its respective infrastructure and is ready to receive the application;

	UC-2: Replicate Infrastructure
Description	The developer needs to replicate the production environment to perform a change.
Motivation	Developers must have the possibility to create and/or replicate an environment as a means of adapting to unexpected changes that may happen during production.
Users	DevOps Team Members, DevOps Team Manager.

Assumptions	<ol style="list-style-type: none"> 1. The production environment is active 2. There is an existing VCS repository on AWS CodeCommit 3. There is an infrastructure configuration file in the VCS repository 4. The application's source code is located in the VCS repository;
Order of Events	<ol style="list-style-type: none"> 1. The developer creates a new branch from the master branch; 2. AWS CodeCommit detects the new branch; 3. AWS CodeCommit Actions triggers the event associated with creating an environment; 4. The configuration file is communicated to the AWS platform; 5. The AWS platform creates the environment together with the defined resources;
Alternate Paths	None.
Results	The production environment is replicated and made available for the developer to replicate the error that occurred in production and to test the corrections made.

9.1

	UC-3: Delete Infrastructure
Description	The developer needs to delete the production environment when it is no longer necessary.
Motivation	Developers must have the possibility to the application's infrastructure in order to relieve the unnecessary resources that have been provisioned .
Users	DevOps Team Members, DevOps Team Manager.
Assumptions	<ol style="list-style-type: none"> 1. The production environment is active 2. There is an existing VCS repository on AWS CodeCommit 3. There is an infrastructure configuration file in the VCS repository 4. The application's source code is located in the VCS repository;
Order of Events	<ol style="list-style-type: none"> 1. The developer creates a new branch from the master branch; 2. AWS CodeCommit detects the new branch; 3. AWS CodeCommit Actions triggers the event associated with creating an environment; 4. The configuration file is communicated to the AWS platform; 5. The AWS platform creates the environment together with the defined resources;
Alternate Paths	None.
Results	The production infrastructure is deleted.

Continuous Integration

	UC-5: Process Source Code Changes
Description	The developer creates a change in the application's source code.
Motivation	The changes to the application's source code must be detected by the framework, together with performing automated functional, performance and security tests.
Users	DevOps Team Members.
Assumptions	<ol style="list-style-type: none"> 1. The application source code is in a VCS repository; 2. The buildspec file is together with the application; 3. The test parameters are defined in a test file;
Order of Events	<ol style="list-style-type: none"> 1. Developer performs a new commit of changed code; 2. The AWS CodeCommit Repository updates the version of code; 3. The AWS CodeCommit communicates the changes to the AWS CodePipeline; 4. AWS CodeBuild performs the build of the application into a container image artefact according to the buildspec file; 5. AWS CodeBuild reads the test file and performs the automated functional, performance and security tests; 6. The new artefact is placed into an ECR repository;
Alternate Paths	<ol style="list-style-type: none"> 4.1. The application build fails; 4.2. The developer is notified that an error occurred in the build process of the application; 4.3. The commit is rolled back; 5.1. The application does not comply with a test; 5.2. The developer is notified of the reason for failing the test; 5.3. The commit is rolled back;
Results	<ol style="list-style-type: none"> 1. The image is successfully added to the ECR repository;

	UC-6: Process buildspec file Changes
Description	The developer performs a change in the application's buildspec file.
Motivation	The changes to the buildspec file must be detected by the framework, rebuilding the image and performing the defined tests to the artefact.
Users	DevOps Team Members.
Assumptions	<ol style="list-style-type: none"> 1. The buildspec code is in a VCS repository;
Order of Events	<ol style="list-style-type: none"> 1. Developer commits the changed buildspec file to the repository; 2. The AWS CodeCommit Repository updates the version of the buildspec file;

	<ol style="list-style-type: none"> 3. The AWS CodeCommit communicates the changes to the AWS CodePipeline; 4. AWS CodeBuild performs the build of the application into a container image artefact according to the new buildspec file; 5. AWS CodeBuild reads the test file and performs the automated functional, performance and security tests; 6. The new artefact is placed into an ECR repository;
Alternate Paths	<ol style="list-style-type: none"> 4.1. The application build fails; 4.2. The developer is notified that an error occurred in the build process of the application; 4.3. The commit is rolled back;
Results	The image is successfully added to the ECR repository;

	UC-7: Process Test Parameter Changes
Description	The developer performs a change in the application's test parameter file.
Motivation	The changes to the application's test parameter file must be detected by the framework, together with performing the changes defined in the file automatically.
Users	DevOps Team Members.
Assumptions	<ol style="list-style-type: none"> 1. The test parameter file is in the VCS;
Order of Events	<ol style="list-style-type: none"> 1. Developer commits the changed test parameter file to the repository; 2. The AWS CodeCommit Repository updates the version of the test parameter file; 3. The AWS CodeCommit communicates the changes to the AWS CodePipeline; 4. AWS CodeBuild performs the build of the application into a container image artefact according to the existing buildspec file; 5. AWS CodeBuild reads the new test parameter file and performs the automated functional, performance and security tests; 6. The new artefact is placed into an ECR repository;
Alternate Paths	<ol style="list-style-type: none"> 4.1. The application fails a test in the new test parameter file; 4.2. The developer is notified that an error occurred in the build process of the application; 4.3. The commit is rolled back;
Results	The image is successfully added to the ECR repository;

Continuous Delivery

	UC-8: Phased Rollouts – Blue-Green Deployments
--	---

Description	The chosen deployment strategy is Blue-Green Deployments.
Motivation	Developers must be able to choose different deployment strategies for the delivery of software changes. In this case, the framework must be configurable in a way that Blue-Green Deployments can be used to deliver software changes.
Users	DevOps Team Members.
Assumptions	<ol style="list-style-type: none"> 1. A new container image has been added to the ECR repository; 2. The production environment is operational; 3. The application is running in the production environment;
Order of Events	<ol style="list-style-type: none"> 1. A new version of an image is placed into the ECR repository; 2. This new version is detected by the repository; 3. A replica of the production environment is created; 4. The repository triggers a lambda function which, together with AWS StepFunctions, releases the new version into a new node in the cluster; 5. Quality assurance and user acceptance tests are run in the new environment; 6. Tests conclude, the new environment replaces the previous version in its entirety;
Alternate Paths	<ol style="list-style-type: none"> 5.1.1 Quality Assurance tests fail; 5.1.2 The developer responsible for the new deployment is notified that a test failed; 5.1.3 The container version is rolled back; 5.2.1 Acceptance tests fail; 5.2.2 The developer responsible for the new deployment is notified that a test failed; 5.2.3 The container version is rolled back;
Results	<ol style="list-style-type: none"> 1. The previous environment is replaced, and traffic is rerouted to the new version;

	UC-9: Phased Rollouts – A/B Testing
Description	The chosen deployment strategy is A/B Testing.
Motivation	Developers must be able to choose different deployment strategies for the delivery of software changes. In this case, the framework must be configurable in a way that A/B Testing Deployments can be used to deliver software changes.
Users	DevOps Team Members.
Assumptions	<ol style="list-style-type: none"> 1. A new container image has been added to the ECR repository; 2. The production environment is operational; 3. The application is running in the production environment;

Order of Events	<ol style="list-style-type: none"> 1. A new version of an image is placed into the ECR repository; 2. This new version is detected by the repository; 3. The repository triggers a lambda function which, together with AWS StepFunctions, releases the new version into a pod in the cluster; 4. Part of the traffic is re-routed into the new pod; 5. This deployment into different pods can be performed with various versions of the container image.
Alternate Paths	<ol style="list-style-type: none"> 4.1 An error occurs in the new pod; 4.2 The version of the pod is rolled back to its previous version; 4.3 The developer responsible for the deployment is notified that an error occurred together with the cause of the issue.
Results	<ol style="list-style-type: none"> 1. The previous environment is replaced, and traffic is rerouted to the new version;

	UC-10: Phased Rollouts – Canary Releases
Description	The chosen deployment strategy is Canary Releases.
Motivation	Developers must be able to choose different deployment strategies for the delivery of software changes. In this case, the framework must be configurable in a way that Canary Releases can be used to deliver software changes.
Users	DevOps Team Members.
Assumptions	<ol style="list-style-type: none"> 1. A new container image has been added to the ECR repository; 2. The production environment is operational; 3. The application is running in the production environment;
Order of Events	<ol style="list-style-type: none"> 1. A new version of an image is placed into the ECR repository; 2. This new version is detected by the repository; 3. The ECR repository triggers a lambda function which retrieves the new container image; 4. A percentage of the pods running in the service node receive the new container image; 5. A percentage of the application’s traffic is rerouted to the new container image; 6. Events 3 and 4 are repeated with an increment to the percentage value;
Alternate Paths	<ol style="list-style-type: none"> 3.1 An error occurs in the new container image; 3.2 The pods receive the previous version of the container imager; 3.3 The developer responsible for developing the new version is notified of the error that occurred and the possible cause for the error; <p>The previous alternate paths are the same as those which can occur in the fourth event and therefore are applicable as alternate paths of this event as well.</p>

Results	2. All the pods in the cluster's node are replaced with the latest version of the container image.
----------------	--

Application Monitoring

	UC-11: View Cluster Metrics
Description	A developer can view the metrics pertaining to a specific cluster.
Motivation	As a means of gaining application observability, the developers must be able to view the values of consumed resources on behalf of the Kubernetes cluster as a means of understanding the application's behaviour and obtaining information to perform optimizations and efficiency testing on the cluster.
Users	DevOps Team Members.
Assumptions	<ol style="list-style-type: none"> 1. The application is running in the production environment; 2. The cluster has an instance of Prometheus running as a daemon set; 3. Grafana is coupled with the Prometheus instance and is available by accessing the port 3000;
Order of Events	<ol style="list-style-type: none"> 1. The developer accesses the address and port where Grafana is available on; 2. The developer logs into the Grafana server with the correct credentials; 3. The developer accesses the metrics dashboard; 4. The developer chooses the intended cluster; 5. Grafana presents the dashboard associated with the selected cluster;
Alternate Paths	<ol style="list-style-type: none"> 2.1 The developer inserts incorrect credentials; 2.2 An error message appears notifying that the credentials were incorrect;
Results	<ol style="list-style-type: none"> 1. The metrics associated with the intended cluster appear on the Grafana dashboard;

	UC-12: View Pod Metrics
Description	A developer can view the metrics pertaining to a specific pod.
Motivation	As a means of gaining application observability, the developers must be able to view the values of consumed resources on behalf of the Kubernetes pod as a means of understanding the application's behaviour and obtaining information to perform optimizations and efficiency testing on the cluster.
Users	DevOps Team Members.
Assumptions	<ol style="list-style-type: none"> 1. The application is running in the production environment; 2. The pod has an instance of Prometheus running as a daemon set;

	3. Grafana is coupled with the Prometheus instance and is available by accessing the port 3000;
Order of Events	<ol style="list-style-type: none"> 1. The developer accesses the address and port where Grafana is available on; 2. The developer logs into the Grafana server with the correct credentials; 3. The developer accesses the metrics dashboard; 4. The developer chooses the intended cluster; 5. The developer chooses the intended pod; 6. Grafana presents the dashboard associated with the selected pod.
Alternate Paths	<p>2.3 The developer inserts incorrect credentials;</p> <p>2.4 An error message appears notifying that the credentials were incorrect.</p>
Results	<ol style="list-style-type: none"> 1. The metrics associated with the intended pod appear on the Grafana dashboard.

	UC-13: View Microservice Logs
Description	Perform log analysis into specific pods.
Motivation	Error detection, failure recovery and debugging are an essential task that must be performed to guarantee that an application is successful in production. Given that the sources of logs are dozens, if not hundreds, accessing the logs of each application is troublesome. Accessing one service individually is not a solution.
Users	DevOps Team Members.
Assumptions	<ol style="list-style-type: none"> 1. The application is running in the production environment; 2. An instance of Logstash is running as a side-car deployment on each pod; 3. Elasticsearch is connected to Logstash; 4. Kibana connected to Elasticsearch as a means of interaction. 5. Kibana is made available on port number 5601.
Order of Events	<ol style="list-style-type: none"> 1. The developer accesses the Kibana login page via the desired VPC address and port 5601; 2. The developer inserts the correct credentials and logs into the platform; 3. The developer selects the intended pod to analyse; 4. The developer is presented with the logs belonging to the pod;
Alternate Paths	<p>1.1 The developer accesses the incorrect address and/or port number;</p> <p>2.1 The developer inserts the incorrect credentials;</p> <p>2.2 An alert appears on screen notifying that the credentials are incorrect;</p>
Results	<ol style="list-style-type: none"> 1. The logs belonging to the desired pod are shown to the developer.

	UC-14: Query Microservice Logs
Description	Perform log analysis into specific pods, analysing specific logs from specific timeframes or requests.
Motivation	In the event of an isolated error occurring, there must be a means of accessing the logs produced when the error occurs to better understand what caused the error and what needs to be changed.
Users	DevOps Team Members.
Assumptions	<ol style="list-style-type: none"> 1. The application is running in the production environment; 2. An instance of Logstash is running as a side-car deployment on each pod; 3. Elasticsearch is connected to Logstash; 4. Kibana connected to Elasticsearch as a means of interaction. 5. Kibana is made available on port number 5601; 6. The developer has successfully logged into the Kibana platform.
Order of Events	<ol style="list-style-type: none"> 1. The developer selects the intended pod to analyse; 2. The developer is presented with the logs belonging to the pod; 3. The developer uses the search bar to search for specific fields in the logs that are presented; 4. Kibana performs the filtering according to the text and shows the related logs on screen.
Alternate Paths	4.1 The search query does not return any values;
Results	<ol style="list-style-type: none"> 1. The logs associated with the query are shown according to the selected pod.

Functional Requirements

Infrastructure Provisioning

	FR-01: IAM Authorization - Infrastructure Provisioning
Description	To guarantee that only the DevOps Maintainers in the specified DevOps Team can make changes to the infrastructure's characteristics, the infrastructure must have a policy-based strategy which analyses if the account that is trying to perform changes to the infrastructure has the correct policies which authorise these changes. DevOps Team Leaders must have a larger privilege considering that the Prerequisites of the Infrastructure Provisioning component must be created by them.
Priority	Must
Users	DevOps Team Members.
Assumptions	Version Control System – AWS CodeCommit Repository; UC-1: Create Development Environment; UC-2: Replicate an Environment; UC-3: Create Staging Environment; UC-4: Create Production Environment.

	FR-02: VCS Authentication
Description	As a means of controlling which elements have access to and managing the changes made to the infrastructure's configuration, there must be a strategy in place which requires that a developer authenticates themselves to have access to the code repository. In this case, the team manager must manage which AWS CodeCommit accounts can contribute and access the repository.
Priority	Must
Users	DevOps team manager.
Assumptions	Version Control System – AWS CodeCommit Repository; UC-1: Create Development Environment; UC-2: Replicate an Environment; UC-3: Create Staging Environment; UC-4: Create Production Environment.

	FR-03: Private Version Control System Repository
Description	There must be a Version Control System repository responsible for tracking

	and registering changes made to configuration files.
Priority	Must
Users	DevOps Team Members.
Assumptions	YAML environment configuration file.

	FR-04: Infrastructure Creation
Description	The Infrastructure creation must be conducted according to the template files provided in the VCS repository.
Priority	Must
Users	DevOps Team Members.
Assumptions	Version Control System – AWS CodeCommit Repository; UC-1: Create Production Environment.

	FR-05: Infrastructure Update
Description	When there is a change made to the infrastructure's template file, there must be a mechanism in place which performs a change to the infrastructure's state according to the new file's version.
Priority	Must
Users	DevOps Team Members.
Assumptions	Version Control System – AWS CodeCommit Repository; UC-2: Replicate an Environment.

	FR-06: Infrastructure Replication
Description	There must be a means of replicating the production environment on demand, without affecting the production environment.
Priority	Must
Users	DevOps Team Members.
Assumptions	Version Control System – AWS CodeCommit Repository; UC-2: Replicate an Environment.

	FR-07: Infrastructure Deletion
Description	There must be a means of deleting the production environment when it is no longer necessary.

Priority	Must
Users	DevOps Team Members.
Assumptions	Version Control System – AWS CodeCommit Repository; UC-2: Replicate an Environment.

	FR-08: Daily Infrastructure Deletion
Description	The developed infrastructure must be produced in a way that, on a daily basis, it can be deleted and re-deployed, without requiring configuration alterations.
Priority	Must
Users	DevOps Team Members.
Assumptions	Version Control System – AWS CodeCommit Repository; UC-2: Replicate an Environment.

	FR-09: Repository Approval Scheme - Infrastructure Provisioning
Description	The developed infrastructure must be produced in a way that, on a daily basis, it can be deleted and re-deployed, without requiring configuration alterations.
Priority	Must
Users	DevOps Team Members.
Assumptions	Version Control System – AWS CodeCommit Repository; UC-2: Replicate an Environment.

	FR-10: Create a VPC
Description	The application infrastructure requires a VPC for networking.
Priority	Must
Users	DevOps Maintainers
Assumptions	None

	FR-11: Create a Private Subnet for EKS
Description	The application infrastructure requires a Private Subnet for maintaining security regarding the EKS cluster.
Priority	Must
Users	DevOps Maintainers

Assumptions	None
--------------------	------

	FR-12: Create a Private Subnet for the RDS Application
Description	The application infrastructure requires a Private Subnet for maintaining security regarding the database application.
Priority	Must
Users	DevOps Maintainers
Assumptions	None

	FR-13: Create a Public Subnet for the Bastion Host
Description	The application infrastructure requires a Public Subnet to allow external access to the Bastion Hosts.
Priority	Must
Users	DevOps Maintainers
Assumptions	None

	FR-14: Create an RDS Application
Description	The application infrastructure could have a database application for data management.
Priority	Could
Users	DevOps Maintainers
Assumptions	None

	FR-15: Create an EKS Cluster
Description	The application infrastructure must have an AWS EKS Cluster.
Priority	Must
Users	DevOps Maintainers
Assumptions	None

	FR-16: Create Resources in Two Availability Zones
Description	The application infrastructure must span two availability zones to guarantee

	high availability for the application.
Priority	Must
Users	DevOps Maintainers
Assumptions	None

	FR-17: Create an Elastic Load Balancer
Description	The application infrastructure must an Elastic Load Balancer to manage the flow of traffic between application instances.
Priority	Must
Users	DevOps Maintainers
Assumptions	None

	FR-18: Create a DNS Routing Service
Description	The application must be accessible via a DNS domain.
Priority	Must
Users	DevOps Maintainers
Assumptions	None

	FR-19: Possibility to SSH Connection to Bastion Host from Specific Range of IP Addresses
Description	SSH connections must only be possible when using a machine which has an IP address within a certain range.
Priority	Must
Users	DevOps Maintainers
Assumptions	None

	FR-20: Block to SSH Connection to Bastion Host, from a Machine with a Non-Authorized IP Address.
Description	SSH connections must be blocked when using a machine which does not has an IP address within a certain range.
Priority	Must

Users	DevOps Maintainers
Assumptions	None

	FR-21: Possibility to SSH Connection from Bastion Host to EKS Cluster Nodes
Description	It must be possible to perform an SSH connection from a Bastion Host to a node within the EKS Cluster.
Priority	Must
Users	DevOps Maintainers
Assumptions	None

9.2

	FR-22: Kubectl Tool Access to the EKS Cluster from Within a Bastion Host
Description	The Bastion Host must be able to query the Kubernetes server by using a Kubectl tool.
Priority	Must
Users	DevOps Maintainers
Assumptions	None

Continuous Integration

	FR-23: Private Version Control System Repository
Description	As a means of controlling the changes made to the application's source code, who made the changes and triggering the remaining tasks of the pipeline, there must be a private code repository in place to keep a record of bug fixes, functionality implementations and general code control for the development team.
Priority	Must
Users	DevOps Team Members, DevOps team manager
Assumptions	Application source code; UC-5: Process Source Code Changes; UC-6: Process buildspec File Changes; UC-7: Process Test Parameter Changes

	FR-24: IAM Authorization - Continuous Integration
Description	To guarantee that only the DevOps Maintainers in the specified DevOps Team can make changes to the infrastructure's characteristics, the infrastructure must have a policy-based strategy which analyses if the account that is trying to perform changes to the infrastructure has the correct policies which authorise these changes. DevOps Team Leaders must have a larger privilege considering that the Pre-Requisites of the Infrastructure Provisioning component must be created by them.
Priority	Must
Users	DevOps Team Members.
Assumptions	Version Control System – AWS CodeCommit Repository; UC-1: Create Development Environment; UC-2: Replicate an Environment; UC-3: Create Staging Environment; UC-4: Create Production Environment.

	FR-25: Repository Approval Scheme - Continuous Integration
Description	The developed infrastructure must be produced in a way that, on a daily basis, it can be deleted and re-deployed, without requiring configuration alterations.
Priority	Must
Users	DevOps Team Members.
Assumptions	Version Control System – AWS CodeCommit Repository; UC-2: Replicate an Environment.

	FR-26: Application Source Code
Description	The factors that define what must be defined in the functional requirements FR-16 to FR-23 are dependent on which language is used to develop the application, what the application requires in terms of dependencies, build requirements and tests.
Priority	Must
Users	DevOps Team Members
Assumptions	FR-14: Private Version Control System Repository.

	FR-27: Automated Pipeline Creation
Description	Given that the intention of the CI/CD pipeline is to automate the build

	process of an application, there must be a specification file which defines the steps that AWS CodeBuild must follow to produce an executable artefact.
Priority	Must
Users	DevOps Team Members
Assumptions	FR-14: Private Version Control System Repository; FR-15: Application Source Code.

	FR-28: Application Buildspec file
Description	Given that the intention of the CI/CD pipeline is to automate the build process of an application, there must be a specification file which defines the steps that AWS CodeBuild must follow to produce an executable artefact.
Priority	Must
Users	DevOps Team Members
Assumptions	FR-14: Private Version Control System Repository; FR-15: Application Source Code.

	FR-29: Automated Code Build on Commit
Description	As a means of reducing the time to market occupied by software changes, the process of building the application source code must be done autonomously. To achieve this, there must be a mechanism in place to trigger the build of the application.
Priority	Must
Users	DevOps Team Members.
Assumptions	FR-14: Private Version Control System Repository FR-15: Application Source Code; FR-16: Application buildspec file.

	FR-30: Automated Static Code Analysis Module
Description	As a means of guaranteeing that coding guidelines are met and that there are no syntax errors and weaknesses, there must be a form of analysing application code to reduce time spent debugging.
Priority	Must
Users	DevOps Team Members.
Assumptions	FR-14: Private Version Control System Repository FR-15: Application Source Code;

	FR-17: Automated Application Build on Commit.
--	---

	FR-31: Automated Dependency Verification Module
Description	Working with external dependencies and external plugins can put the application build at risk. Therefore, it is necessary to perform a validation of the dependencies and plugins used in the application to safeguard the final executable from a security point of view and avoid compromising dependencies.
Priority	Must
Users	DevOps Team Members.
Assumptions	FR-14: Private Version Control System Repository FR-15: Application Source Code; FR-16: Application buildspec file; FR-17: Automated Application Build on Commit; FR-18: Automated Static Code Analysis Module.

	FR-32: Automated Unit Testing Module
Description	Each function or unit in the application's source code must be able to produce the correct output, based on a certain input. This output must be evaluated and correct on an individual level.
Priority	Must
Users	DevOps Team Members.
Assumptions	FR-14: Private Version Control System Repository FR-15: Application Source Code; FR-17: Automated Application Build on Commit. FR-18: Automated Static Code Analysis Module; FR-19: Automated Dependency Verification Module.

	FR-33: Automated Docker Image Build
Description	Each function or unit in the application's source code must be able to produce the correct output, based on a certain input. This output must be evaluated and correct on an individual level.
Priority	Must
Users	DevOps Team Members.
Assumptions	FR-14: Private Version Control System Repository FR-15: Application Source Code; FR-17: Automated Application Build on Commit.

	FR-18: Automated Static Code Analysis Module; FR-19: Automated Dependency Verification Module.
--	---

	FR-34: Automated Docker Image Push to AWS ECR Repository
Description	Each function or unit in the application's source code must be able to produce the correct output, based on a certain input. This output must be evaluated and correct on an individual level.
Priority	Must
Users	DevOps Team Members.
Assumptions	FR-14: Private Version Control System Repository FR-15: Application Source Code; FR-17: Automated Application Build on Commit. FR-18: Automated Static Code Analysis Module; FR-19: Automated Dependency Verification Module.

	FR-35: Automated Docker Image Re-Tagging
Description	Each function or unit in the application's source code must be able to produce the correct output, based on a certain input. This output must be evaluated and correct on an individual level.
Priority	Must
Users	DevOps Team Members.
Assumptions	FR-14: Private Version Control System Repository FR-15: Application Source Code; FR-17: Automated Application Build on Commit. FR-18: Automated Static Code Analysis Module; FR-19: Automated Dependency Verification Module.

	FR-38: Build State Change Notification Service
Description	Each function or unit in the application's source code must be able to produce the correct output, based on a certain input. This output must be evaluated and correct on an individual level.
Priority	Must
Users	DevOps Team Members.
Assumptions	FR-14: Private Version Control System Repository FR-15: Application Source Code; FR-17: Automated Application Build on Commit. FR-18: Automated Static Code Analysis Module;

	FR-19: Automated Dependency Verification Module.
--	--

Continuous Delivery

	FR-39: Functional Testing
Description	Before deploying the new container image into production, it must be submitted to functional tests to guarantee that it complies with the defined functional specifications.
Priority	Would
Users	DevOps Team Members
Assumptions	FR-24: Automated Image Build.

	FR-40: Integration Testing
Description	Given that the image will be communicating with other services to achieve, the new image must undergo tests involving other services to guarantee that it does not cause abnormalities in the production environment.
Priority	Would
Users	DevOps Team Members
Assumptions	FR-24: Automated Image Build.

	FR-41: UI Testing
Description	In the event of the service that is being updated being a user interface, it must also undergo evaluation tests to guarantee that the various functionalities work correctly and are adequate.
Priority	Would
Users	DevOps Team Members
Assumptions	FR-24: Automated Image Build.

	FR-42: Configurable Rollout Strategy – A/B Testing
Description	The ability to perform tests to various functionalities and new versions of software in production, in a standard and simple method whilst maintaining a low-cost threshold.
Priority	Must

Users	DevOps Team Members
Assumptions	FR-30: Autonomous ECR Repository Push.

	FR-43: Configurable Rollout Strategy – Blue-Green Deployment
Description	A simple and efficient method to evaluate software changes on quality assurance and user experience level is to use two different environments. One with the previous version of the image, the second with the updated version, with the traffic of the previous version being rerouted to the new version.
Priority	Must
Users	DevOps Team Members
Assumptions	FR-30: Autonomous ECR Repository Push.

	FR-44: Configurable Rollout Strategy – Canary Releases
Description	Canary deployments allow organisations to test in production with real users and use cases and compare different service versions side by side. It's cheaper than a blue-green deployment because it does not require two production environments. And finally, it is fast and safe to trigger a rollback to a previous version of an application.
Priority	Must
Users	DevOps Team Members
Assumptions	FR-30: Autonomous ECR Repository Push.

	FR-45: Automated Rollback on Failure
Description	To avoid error and issue replication, if an error occurs in the production environment, there must be a system in place that removes the new image from the production environment and replaces it with its predecessor as a means of understanding what caused the issue and perform a fix.
Priority	Should
Users	DevOps Team Members
Assumptions	FR-31: Configurable Rollout Strategy – A/B Testing; FR-32: Configurable Rollout Strategy – Blue-Green Deployment; FR-33: Configurable Rollout Strategy – Canary Releases.

	FR-46: System Status Notification Service
--	--

Description	As a means of understanding and viewing the different steps of the deployment phase, the developer must be notified of the different messages that are produced when delivering the software changes (ex. Rollback caused by error, successful canary release).
Priority	Should
Users	DevOps Team Members
Assumptions	FR-24: Automated Image Build Module; FR-25: Functional Testing; FR-26: Security Testing; FR-27: Integration Testing; FR-28: Performance Testing; FR-29: UI Testing; FR-30: Autonomous ECR Repository Push; FR-31: Configurable Rollout Strategy – A/B Testing; FR-32: Configurable Rollout Strategy – Blue-Green Deployment; FR-33: Configurable Rollout Strategy – Canary Releases; FR-34: Automated Rollback on Failure.

Application Monitoring

	FR-47: Container CPU Consumption Visualisation
Description	To gain observability into how applications are behaving within their containerized environments, the framework must offer the possibility to visualise the levels of CPU resource consumption in real-time.
Priority	Must
Users	DevOps Team Members
Assumptions	Application is in the production environment.

	FR-48: Container Memory Consumption Visualisation
Description	To gain observability into how applications are behaving within their containerized environments, the framework must offer the possibility to visualise the levels of memory resource consumption in real-time.
Priority	Must
Users	DevOps Team Members
Assumptions	Application is in the production environment.

	FR-49: Container Bandwidth Consumption Visualisation
--	---

Description	To gain observability into how applications are behaving within their containerized environments, the framework must offer the possibility to visualise the levels of bandwidth resource consumption in real-time.
Priority	Must
Users	DevOps Team Members
Assumptions	Application is in the production environment.

	FR-50: Container Disk Utilisation Visualisation
Description	To gain observability into how applications are behaving within their containerized environments, the framework must offer the possibility to visualise the levels of disk resource consumption in real-time.
Priority	Must
Users	DevOps Team Members
Assumptions	Application is in the production environment.

	FR-51: Container Count
Description	An application container can fail when in the production environment. To understand where there may be an issue with the health of the container running in a pod, there must be a strategy which shows that a pod has stopped working.
Priority	Must
Users	DevOps Team Members
Assumptions	Application is in the production environment.

	FR-52: Container Health Visualisation
Description	An application container can fail when in the production environment. To understand where there may be an issue with the health of the container running in a pod, there must be a strategy which shows that a pod has stopped working.
Priority	Must
Users	DevOps Team Members
Assumptions	Application is in the production environment.

	FR-53: Cluster Nodes Visualisation
--	---

Description	Having access to the number of nodes that are running in the cluster and understanding how the nodes are behaving offers insights into the resources required to run the cluster.
Priority	Must
Users	DevOps Team Members.
Assumptions	Application is in the production environment.

	FR-54: Cluster Pod Visualisation
Description	A form of understanding how the application is functioning in the production environment is by analysing and keeping a toll of the number of pods running in the cluster to determine if there are sufficient nodes to handle the overall workload.
Priority	Must
Users	DevOps Team Members
Assumptions	Application is in the production environment.

	FR-55: Cluster Resource Utilisation
Description	Understanding resource utilisation helps decision making when looking to increase or decrease the number of nodes in a cluster.
Priority	Must
Users	DevOps Team Members
Assumptions	Application is in the production environment.

	FR-56: Custom Application Metrics
Description	Understanding resource utilisation helps decision making when looking to increase or decrease the number of nodes in a cluster.
Priority	Must
Users	DevOps Team Members
Assumptions	Application is in the production environment.

	FR-57: Application Log Visualisation
Description	Being able to understand the number of calls that are made to an application allows developers to better plan and adapt availability and scalability levels to

	their applications. Given the architectural pattern of microservices, each application can scale individually, which makes resource management more micro-manageable.
Priority	Must
Users	DevOps Team Members
Assumptions	Application is in the production environment.

	FR-58: Application Log Querying
Description	Being able to understand the number of calls that are made to an application allows developers to better plan and adapt availability and scalability levels to their applications. Given the architectural pattern of microservices, each application can scale individually, which makes resource management more micro-manageable.
Priority	Must
Users	DevOps Team Members
Assumptions	Application is in the production environment.

Non-Functional Requirements

	NFR-1: Least privileged access
Description	Developers must only have access to the tools and functionalities that are essential for their work.
Type	Security
Priority	Must

	NFR-2: IAM Roles for multiple users which need identical access
Description	One role for various developers according to the access level required by the tasks to execute.
Type	Security
Priority	Must

	NFR-3: Private EKS endpoints
Description	The cluster's endpoints must be private to safeguard the application from external and unauthorised attacks.
Type	Security
Priority	Must

	NFR-4: Cluster must have its own IAM role
Description	The cluster must have a single role associated with cluster configuration and non-routine changes to the infrastructure (ex. Changing the number of CPU cores available in the cluster).
Type	Security
Priority	Must

	NFR-5: Authorised Repository Access
Description	Only authorised developer accounts can have access to the infrastructure's repository where the configuration file and the application's source code is located.
Type	Security

Priority	Must
-----------------	------

	NFR-6: Infrastructure Change Processing
Description	When a change is made to the infrastructure configuration, these changes must be reflected in the environment when the changes are pushed to the repository.
Type	Reliability
Priority	Must

	NFR-7: No Singleton Pods
Description	If the application is running on a singleton pod and it gets terminated, the entire application goes down and is made unavailable. This cannot be a possibility.
Type	Reliability
Priority	Must

	NFR-8: Automated Test Results
Description	The results of the automated tests must be coherent with reality and reliable to guarantee that the software produced is correct and ready for deployment.
Type	Reliability
Priority	Must

	NFR-9: Automated Software Rollouts
Description	The software changes must be rolled out progressively according to the chosen deployment model, with reliable results originating in those rollouts.
Type	Reliability
Priority	Must

	NFR-10: Multiple nodes with multiple pod replicas
Description	Deploy multiple pod replicas to maintain a highly available application.
Type	Availability
Priority	Should

	NFR-11: Code Standard Compliance
Description	The code in the infrastructure configuration file must be easy to maintain and understand
Type	Supportability
Priority	Must

	NFR-12: Logging Referencing
Description	The logs produced by the application must be produced following JSON notation.
Type	Supportability
Priority	Must

	NFR-13: Pushing Configuration Change
Description	The changes made to the infrastructure's configuration must be implemented autonomously, with the pushing of the changed configuration file being the triggering element for the application of these changes.
Type	Usability
Priority	Must

	NFR-14: Usability – Pushing Application Source Code Changes
Description	The changes made to the application's source code must be implemented autonomously, with the pushing of the changed files being the triggering element for the beginning of the CI/CD Pipeline.
Type	Usability
Priority	Must

	NFR-15: Failed Software Tests
Description	When a test is failed, the system must be able to perform a rollback and notify the developer of the failed test.
Type	Recoverability
Priority	Must

	NFR-16: Error in Production Environment
--	--

Description	In the event of an error occurring in the application's production environment, the system must be able to recover from failure, removing the current version of the application and rolling back to its predecessor.
Type	Recoverability
Priority	Must

	NFR-17: Metric Monitoring Accuracy
Description	The solution used for application monitoring must be able to present metrics with a delay of 1 second from real-time events, 99% of the time.
Type	Performance
Priority	Must

	NFR-18: Logging Referencing
Description	The logging tool must retrieve log results within 10 seconds, 95% of the time, and within 20 seconds 99% of the time.
Type	Performance
Priority	Must

	NFR-19: Infrastructure Creation
Description	There must be a means of verifying that the application's infrastructure was created successfully.
Type	Observability
Priority	Must

	NFR-20: Infrastructure Creation Failure
Description	In the event of the infrastructure's creation failing, there must be a system in place which notifies the developer that there was an error and what the cause was for the error.
Type	Observability
Priority	Must

	NFR-21: Code Build Progress
Description	The progress made when building the application's code must be presented to

	the developer in a format that is easy to understand.
Type	Observability
Priority	Must

	NFR-22: Code Build Failure
Description	In the event of the application's build failing, there must be a system in place which notifies the developer that there was an error and what the cause was for the error.
Type	Observability
Priority	Must

	NFR-23: Code artefact Creation
Description	The creation of an executable artefact, in this case a docker image, must be notified to the developer.
Type	Observability
Priority	Must

	NFR-24: artefact Deployment
Description	The framework must offer a means of visualising the deployment of the new artefact into production according to the deployment strategy that is chosen,
Type	Observability
Priority	Must

Appendix D – Software Architecture and Design



FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE
COIMBRA
DEPARTMENT OF INFORMATICS ENGINEERING

Gabriel Marco Freire Pinheiro

Appendix D – Software Architecture and Design

Dissertation in the context of the Master's Degree in Informatics Engineering,
Specialization in Software Engineering, advised by Professor Nuno Laranjeiro and
Engenheiro Rui Cunha, presented to the Department of Informatics Engineering of the
Faculty of Sciences and Technology of the
University of Coimbra.

July 2022

This page was intentionally left in blank.

Table of Contents

FRAMEWORK ARCHITECTURE	XXXV
INFRASTRUCTURE PROVISIONING.....	XXXV
<i>Identity Access Management</i>	<i>xxxv</i>
<i>Pre-Requisites.....</i>	<i>xxxvi</i>
<i>Provisioning Process.....</i>	<i>xxxviii</i>
<i>AWS CodeCommit Repository Approval Scheme.....</i>	<i>xxxix</i>
<i>Automation Pipeline.....</i>	<i>xl</i>
<i>Infrastructure Elements.....</i>	<i>xlili</i>
CONTINUOUS INTEGRATION/CONTINUOUS DELIVERY	XLVIII
<i>Identity Access Management</i>	<i>xlix</i>
<i>Pre-Requisites.....</i>	<i>li</i>
<i>Continuous Integration Process</i>	<i>liii</i>
<i>Continuous Delivery Process</i>	<i>lviii</i>
MONITORING MICROSERVICES.....	LXVIII
<i>Metrics Analysis.....</i>	<i>lxix</i>
<i>Application Log Analysis.....</i>	<i>lxxiv</i>
CHALLENGES AND DIFFICULTIES	LXXX

Framework Architecture

This chapter provides an explanation into the architectural decisions and features that are incorporated in the framework. These views will be divided into the following sections: i) Infrastructure provisioning; ii) Continuous Integration/Continuous Delivery; iii) Microservice Monitoring.

Finally, there will be one final view which will present the entire framework's overview, defining what the product offers toward its users and the value it offers to the organization. As a means of brevity and facilitating reading, a reduced list of the framework's architecture is included in this section. For a more detailed analysis, please refer to the document Appendix X – Software Architecture and Design.

Infrastructure Provisioning

In this section, there will be a brief explanation of how the process behind provisioning an infrastructure and how the framework processed its creation. There will be a list of elements that are considered as pre-requisites for the framework to be adopted into a project, defining the foundation on which the framework will operate. The framework offers an automation pipeline which autonomously deploys the infrastructure configuration files to the cloud provider for interpretation and creation. This pipeline will be explored in full, as it is the component that offers the most value to this component of the framework, along with the infrastructure itself. Finally, the infrastructure itself will be presented, showing the various components required to construct the cloud-native resources that are required for the microservice application to be deployed and accessible.

Identity Access Management

Identity and access management (IAM) is a framework of business processes, policies and technologies that facilitates the management of electronic or digital identities. In the case for this framework, the DevOps Team leader, DevOps Maintainer and possible third-party entities will have different access to different components within the framework from the Infrastructure Provisioning point of view, with each having their own IAM Role which will be assumed before adopting the framework. These profiles must be provisioned prior to the using the framework in a team. Those three profiles will be the following:

- DevOps Team Leader: role reserved for DevOps team leaders with permissions to read and write information to the AWS CloudFormation service, along with full access to the AWS CodeCommit service;
- DevOps Team Member: role reserve for DevOps maintainers with permissions to read and write infrastructure template files to and from the AWS CodeCommit repository associated with the development project;
- Third-Party Consultant: role reserved for read-.only access to the AWS CodeCommit repository for possible third-party elements that may offer consultation or third-party expertise.

The following figure 47 offers a graphical representation of the organization behind the user roles and their respective policies associated with the AWS CodeCommit repository.

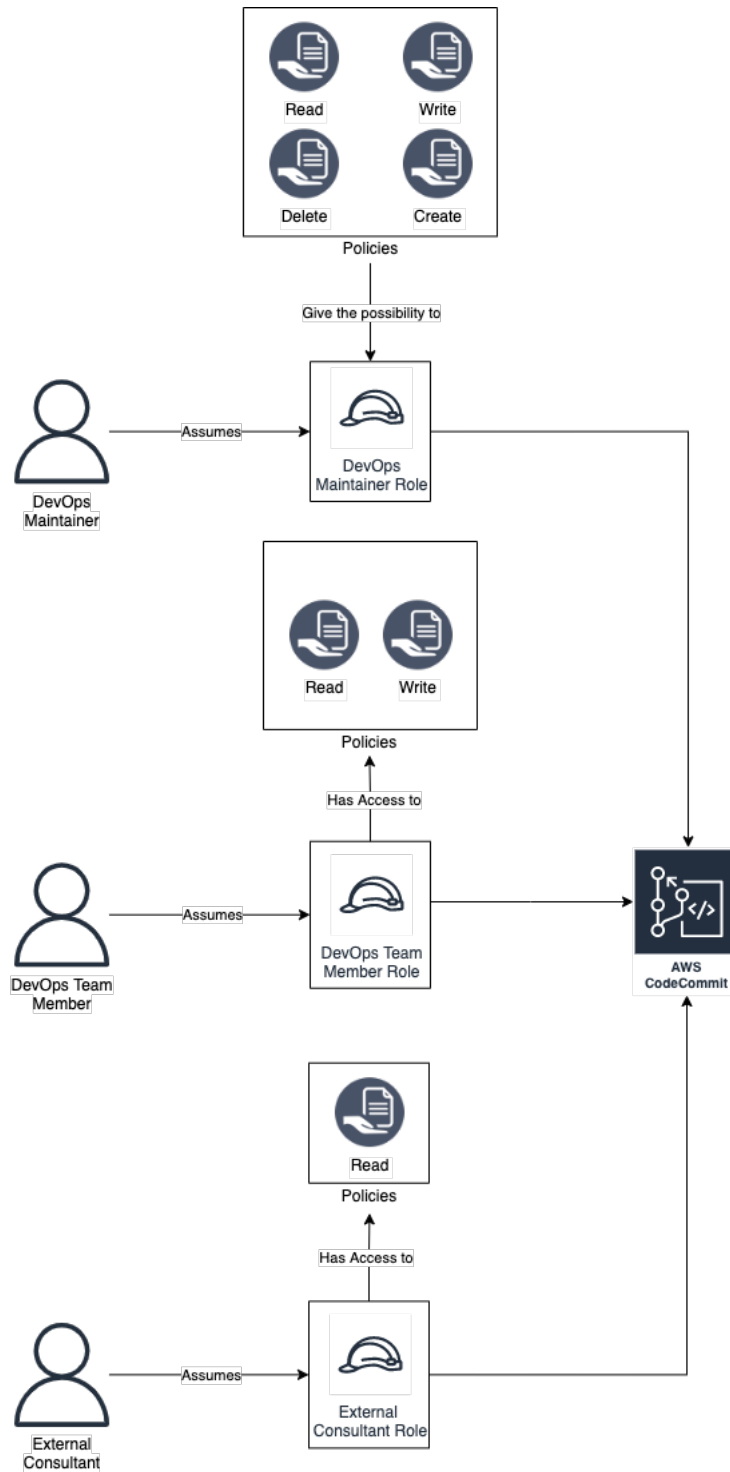


Figure 47. IAM Infrastructure Provisioning Organization

Pre-Requisites

Before applying the framework into a production environment, a number of components are required to create a foundation and prepare configurations for the framework to work. To create these same pre-requisites, a DevOps Team Leader is required to have previously installed the AWS CLI tool which will then give access to an account which applies the best practice of least privileged authorization. This account

will assume a role named DevOps Team Leader which will have full access both to the AWS CloudFormation service, along with all of the components compose the pre-requisite stack (groups of components which cooperate together) which must be provisioned before developing the infrastructure templates. Given that the Team Leader will only be able to access the AWS Services with this account, the possibility of over-stepping these boundaries does not exist. Those components are the following:

- **AWS CodeCommit Repository:** git repository where the infrastructure templates will be maintained and versioned to control alterations and updates that may occur throughout the Infrastructure as Code development process. This will serve as a single point of truth for the elements that require a source to extract the configuration files from;
- **AWS Lambda Function:** serverless instance, responsible for executing snippets of code, without the need for allocating computational resources. This function will be responsible for transferring the template files from the AWS CodeCommit repository to the Amazon S3 bucket and the creation of the AWS CodePipeline that will be the means used to autonomously provision the infrastructure. These pipelines will only be created when a branch named develop, staging, lab or main is created in the repository;
- **AWS S3 Bucket:** Simple Storage Service responsible for acting as a source from which the AWS CodePipeline component extracts the configuration templates. This component is compulsory, given that AWS CloudFormation templates limit the source component for an AWS CodePipeline to an S3 Bucket;
- **AWS IAM StackChangeRole Role:** AWS IAM Role responsible for giving AWS CodePipeline the permissions necessary for creating and performing changes on the AWS CloudFormation stacks;
- **AWS IAM CodePipelineRole Role:** AWS IAM Role which gives permissions to the AWS CodePipeline component to perform changes and access information present in the previously mentioned S3 Bucket;
- **AWS IAM LambdaRole Role:** AWS IAM Role responsible for allowing the AWS Lambda function to have read-only access to the AWS CodeCommit repository, to the S3 bucket, and read and write access to the AWS CloudFormation to create the pipeline responsible for provisioning the infrastructure;
- **AWS CloudWatch Event Rule:** event listener which detects changes to the AWS CodeCommit repository in terms of creating and deleting branches, so as to allow infrastructure replication, and communicates those changes to the AWS Lambda function to create a pipeline at the rate that it is necessary to replicate an environment;
- **AWS Lambda Permission:** permission entry which gives the CloudWatch Event Rule permissions to trigger the AWS Lambda function when changes are performed to the repository.

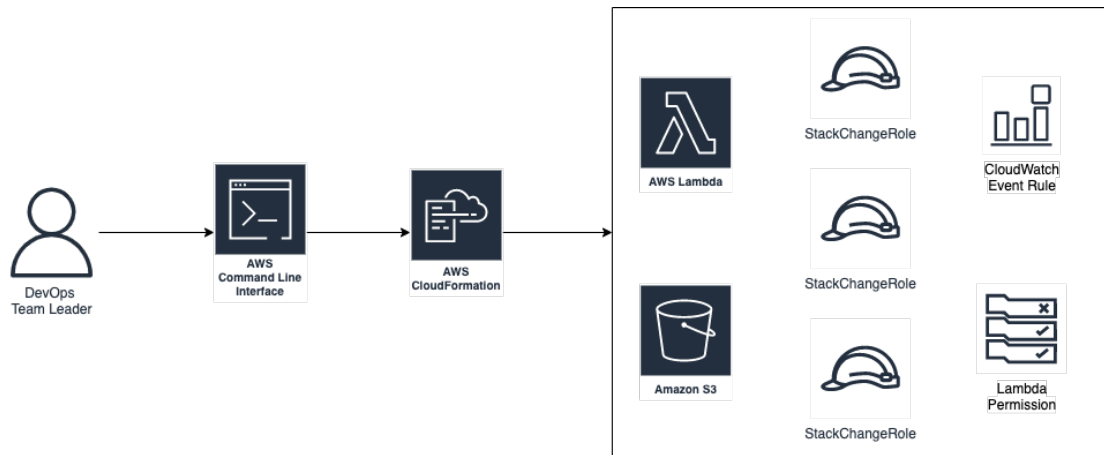


Figure 48. Infrastructure Provisioning Pre-Requisites Creation Process

Figure 12 shows the process that must be followed to create the pre-requisite elements which are necessary for the remaining framework to function. The DevOps Team Leader, duly authenticated with an AWS IAM Account, communicates with AWS Services via the AWS CLI tool. With this tool, the only requirement is to run the following command:

```
aws cloudformation create-stack \
--stack-name PRE_REQ_STACK \
--template-body file://PreRequisites.yaml \
--capabilities CAPABILITY_NAMED_IAM
```

This command will output the ID of the AWS CloudFormation stack that is created. The value for PRE_REQ_STACK is customizable, the file name PreRequisites.yaml must match the name of the pre-requisites template and the parameters which are defined in the template file can be overridden by using the `-parameters` flag.

Provisioning Process

The act of provisioning an infrastructure for a microservice begins with template files which are maintained in a Version Control System. These files contain the configuration of the infrastructure written in YAML – a markup language based on key-value pairs which define the desired state of the components where the application will be running in. The tool used for version control is AWS CodeCommit. In the event of a new version of the code being committed to the repository, AWS CodePipeline is notified of this new version and acts as an intermediary between the repository and Amazon’s Web Services, notifying AWS CloudFormation that there is an update pending on the infrastructure’s configuration. The files are then interpreted and used to create or update the desired environment for the application.

The following figure 16 shows a general overview of the pipeline responsible for provisioning the infrastructure, beginning with the git repository.

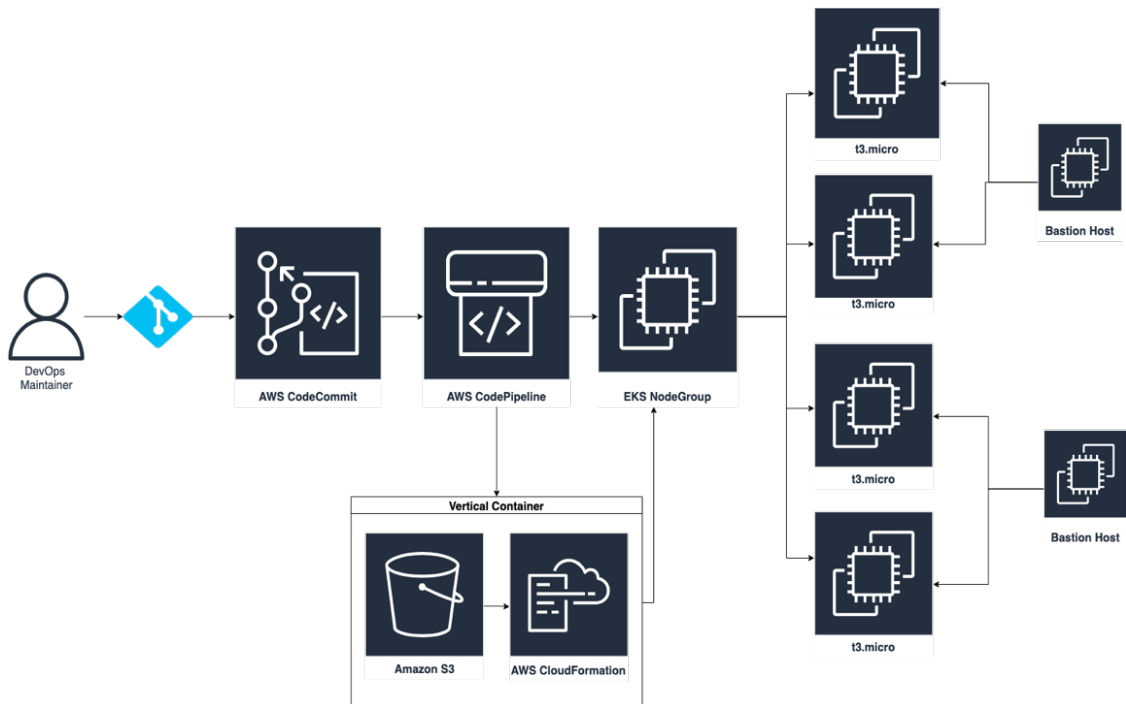


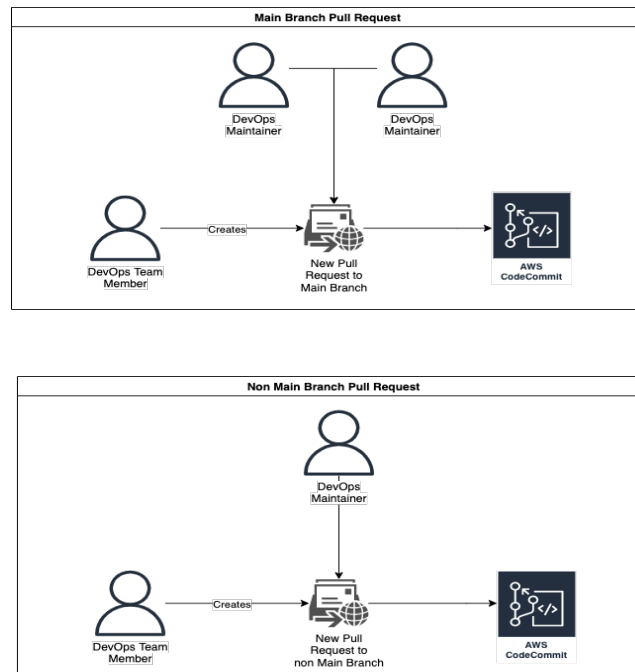
Figure 49. General Overview of the Infrastructure Provisioning Process

The idea of implementing this component in the framework is to develop a means for DevOps maintainers to make the most of what cloud computing and Infrastructure as Code has to offer. Ultimately, the goal is to prepare a solution that, at the click of a button or by running a single command, the configuration files are uploaded to the code repository and provision an infrastructure, without the need for anymore interaction from the developers and no single, step-by-step configuration is required. The diagram in Figure 11 shows a macro-level view of how this process extends from A DevOps Maintainer to the actual environment creation.

Furthermore, when looking to replicate the components themselves, the process is much easier and robust. A developer is only required to replicate the configuration files or override the parameters to have an identical environment to the one that has been previously deployed.

AWS CodeCommit Repository Approval Scheme

Before applying



Automation Pipeline

Once the pre-requisites have been implemented, the means of developing and delivering the infrastructure templates must be implemented. This is accomplished through the use of an automation pipeline which the AWS Lambda function is responsible for creating. Whenever a change is detected in the AWS CodeCommit repository, the Lambda function proceeds to transfer the files from the repository to the S3 Bucket, followed by creating an AWS CodePipeline will then use the files in the S3 Bucket and communicate them to the AWS CloudFormation service and create the infrastructure.

Figure 13 represents the process that begins at the development phase of creating the infrastructure templates which are then committed to the repository. A DevOps Maintainer performs a change to the repository, be it a new commit, a new branch with one of the aforementioned names or a merge between branches, the CloudWatch Event Rule detects this change and deploys the AWS Lambda function. Once invoked, the function then transfers the source templates in the repository to the S3 bucket and, as soon as the change is complete, the AWS CodePipeline is triggered and extracts the infrastructure templates from the bucket and communicates them to the AWS CloudFormation service. After receiving the templates, the AWS CloudFormation service then provisions the different elements that are defined in the template files.

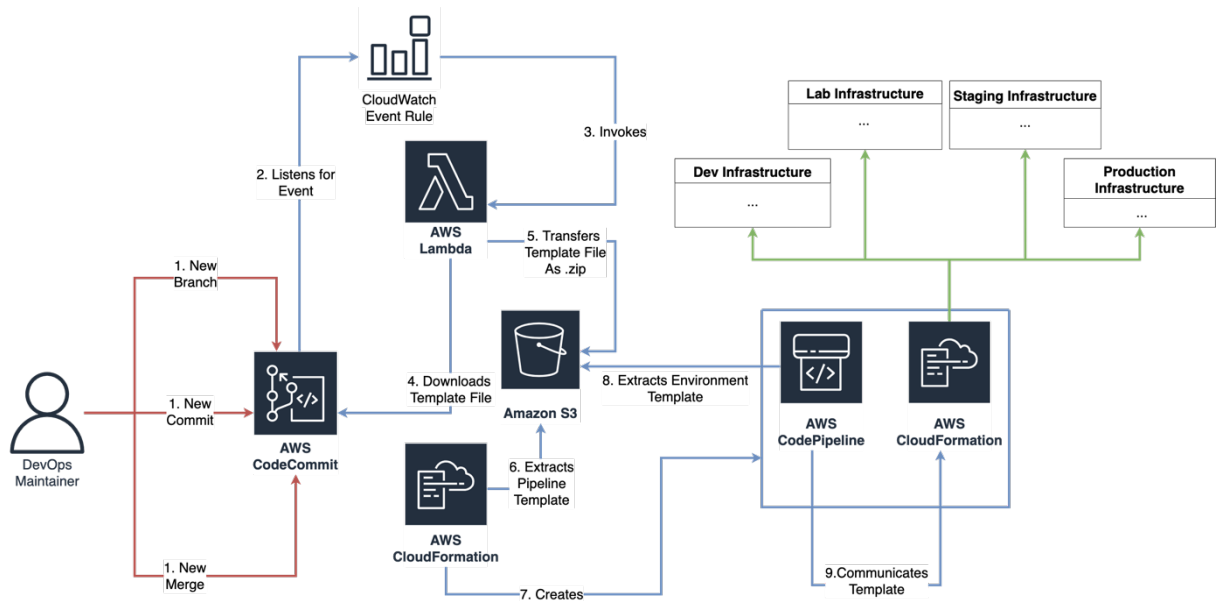


Figure 50. Automation Pipeline Workflow for Infrastructure Deployment

The following Figure 5 presents a representation through an Activity Diagram of the different stages that are necessary for the process from A DevOps Maintainer to AWS CloudFormation to be a possibility. Figure 6 presents a sequence diagram which represents the messages responsible for communicating the necessary changes to create and update an infrastructure.

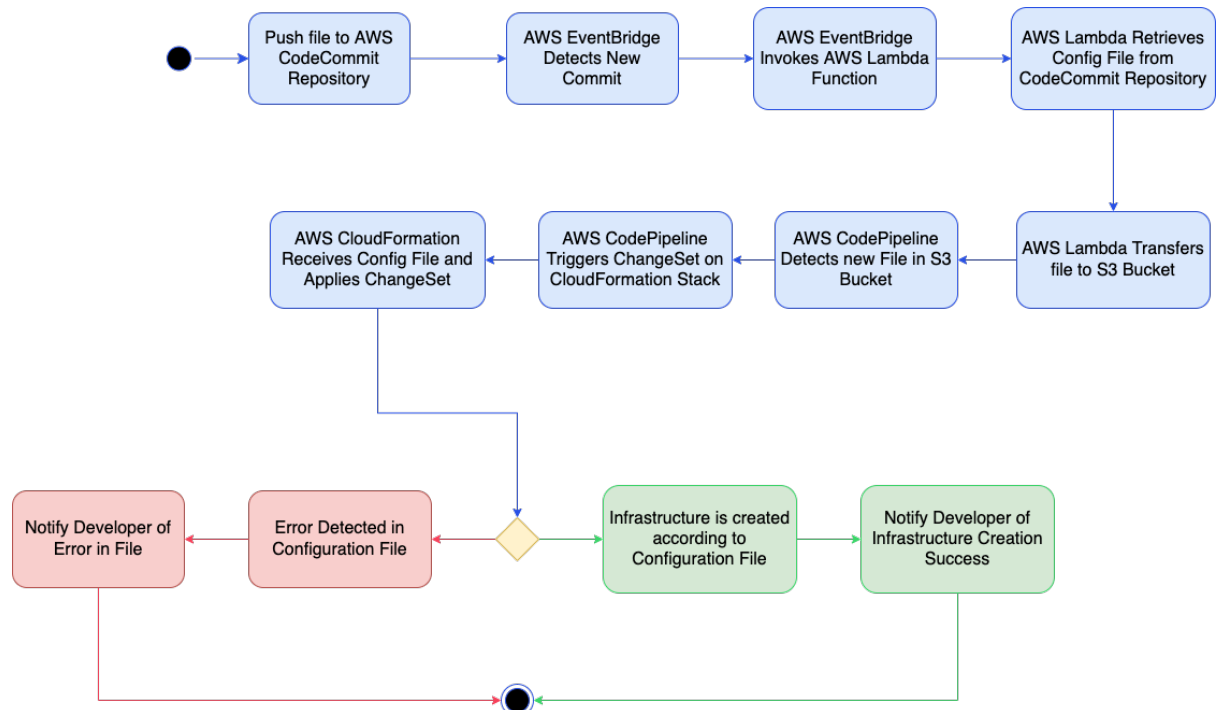


Figure 51. Create And Update Environment Activity Diagram

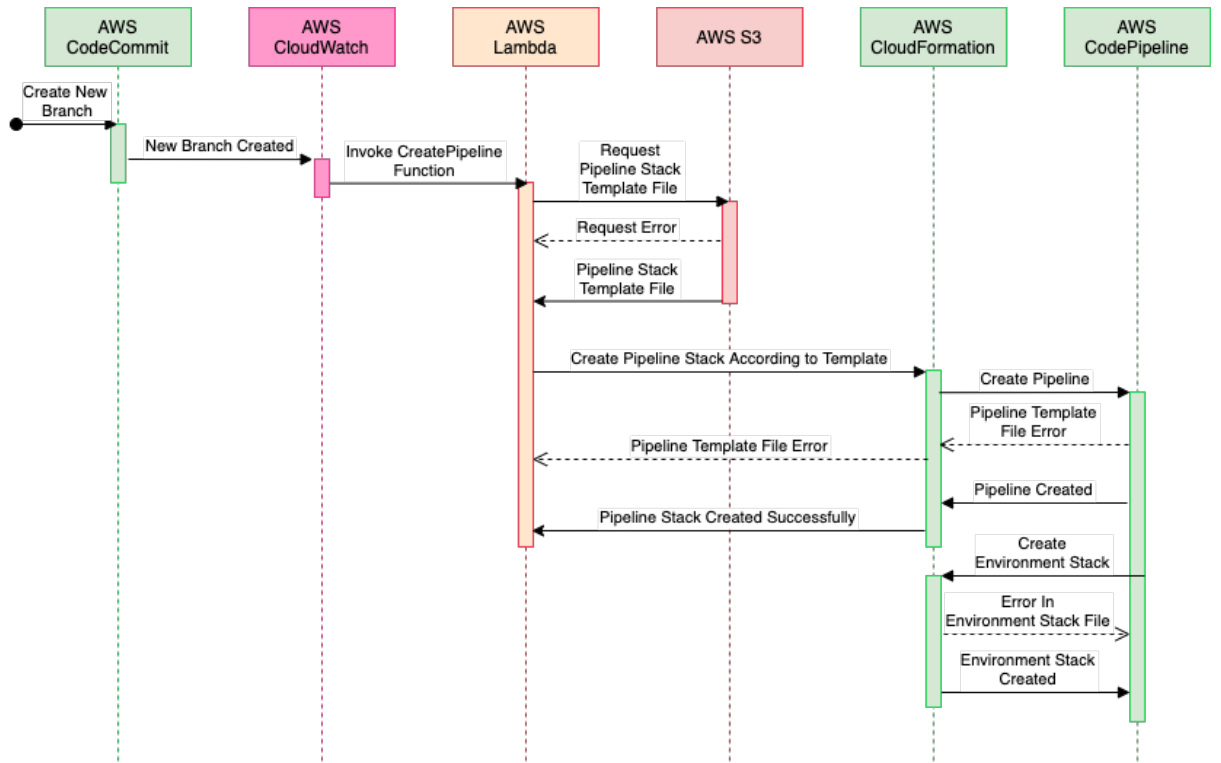


Figure 52. Create And Update Environment Sequence Diagram

The following Figure 7 presents a representation through an Activity Diagram of the different stages that are necessary to perform a replication of the infrastructure.. Figure 6 presents a sequence diagram which represents the messages responsible for communicating the necessary changes to replicate an infrastructure.

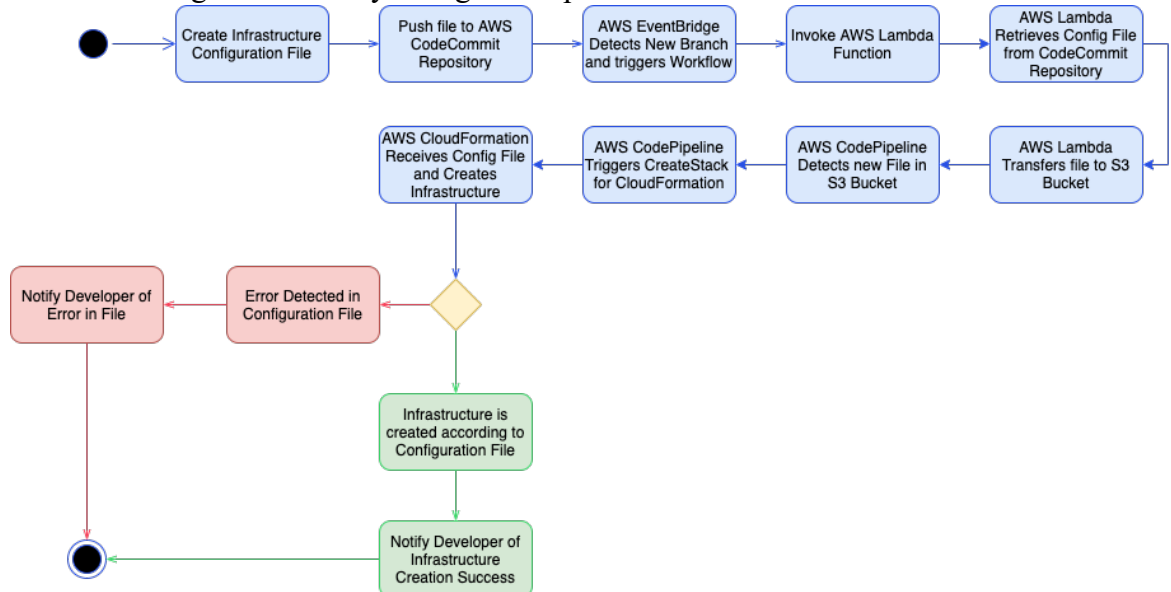


Figure 53. Update Environment Activity Diagram

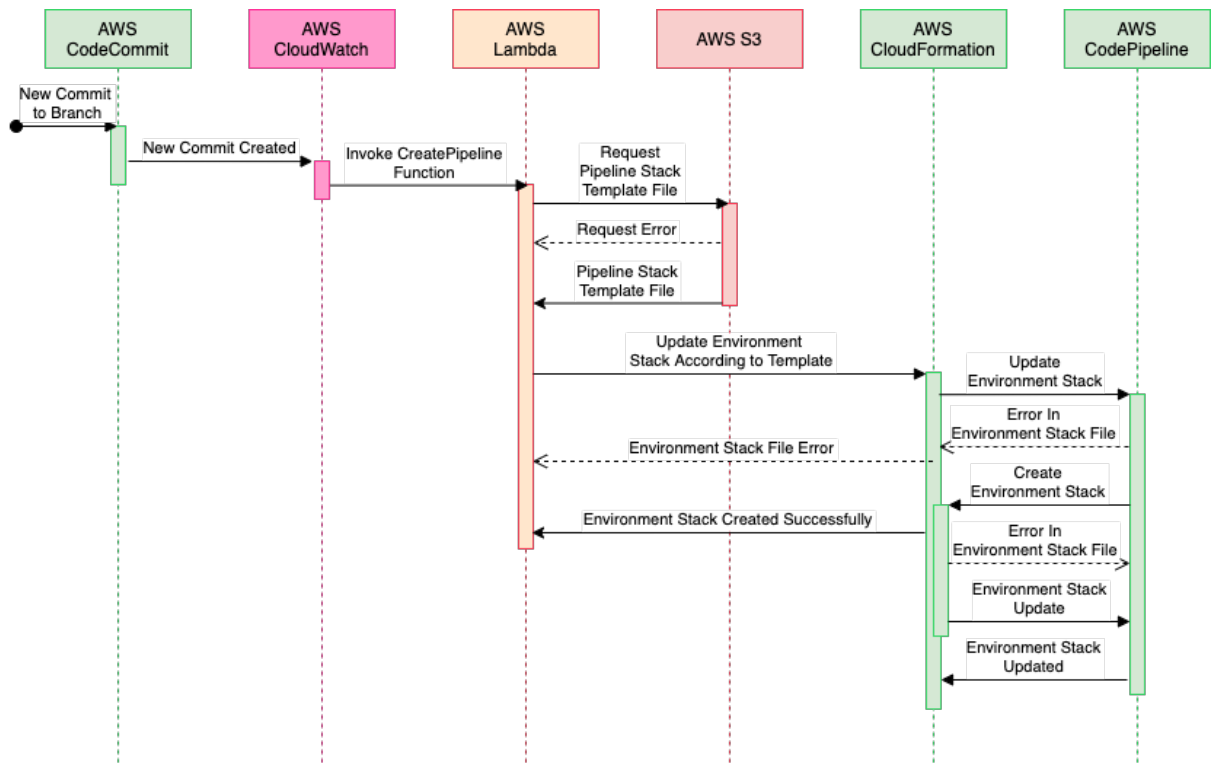


Figure 54. Update Environment Sequence Diagram

These components add value to the framework due to the fact that they offer a means of automating the process of provisioning a cloud-native infrastructure in a question of minutes, whilst simultaneously offering the possibility to treat the templates in the same manner as an application’s source code. A DevOps Maintainer can perform changes to an infrastructure that is still in its development phase without interfering with the configuration which is in use in a production environment. This also offers the possibility to create an environment on demand, which offers a solution for the use-case given when an error occurs in the production infrastructure and requires a change without causing down-time to the remaining elements in the stack. The implementation of that change will require testing and evaluation, in which case the framework is prepared to fulfil this requirement.

Infrastructure Elements

Once the automation pipeline was concluded, it was necessary to develop an infrastructure for the microservices application. This infrastructure must be developed in as much a generic fashion as possible in order to have a solution that can integrate into multiple projects without the need for reconfiguration. The following subsections explain the different components that would provide resources for the application.

The following figure 16 shows a graphical representation of the components that compose the infrastructure on which the microservices application will be deployed. A more in-depth analysis of each element follows.

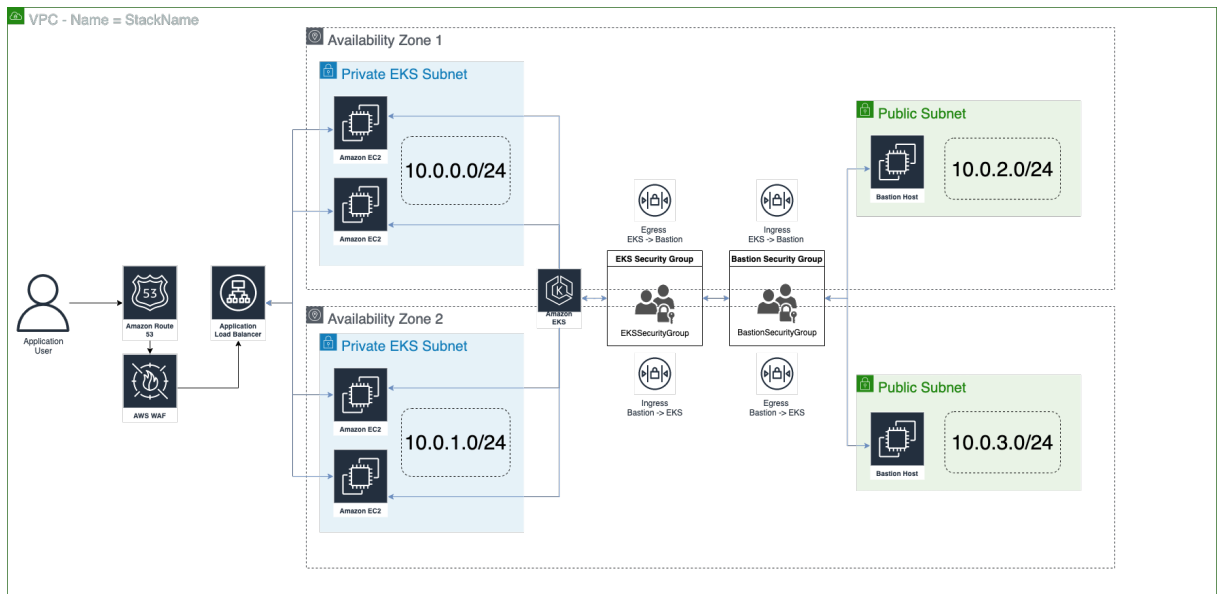


Figure 55. Full Infrastructure Components Macro-View

Networking - The components that compose this section are responsible for allowing network connectivity and communication amongst the various elements that will integrate the infrastructure. Those elements are the following:

- **Virtual private cloud (VPC):** a configurable, centralized and isolated virtualized networking environment into which a range of connectivity tools can be inserted in an effort to provide distributed means of communication;
- **Availability Zones:** AWS Cloud computing resources are housed in highly available data centre facilities. To provide additional scalability and reliability, these data centre facilities are located in different physical locations. Availability Zones are distinct locations within an AWS Region that are engineered to be isolated from failures in other Availability Zones. They provide inexpensive, low-latency network connectivity to other Availability Zones in the same AWS Region [41].
- **Subnets:** a range of possible IP addresses within a VPC which is made available within a specific availability zone. Best practices recommend that the number of subnets that are deployed should always be provisioned in different availability zones, so as to guarantee that, in the event of failure in one of the zones, the other can compensate and mitigate possible down time of the resources. In this solution, there are three different subnets that are created within the VPC, one public subnet on which the bastion host will be deployed, and two private subnets on which the EKS Cluster and the RDS application will be deployed, respectively. The difference between a public and private subnet can be defined as the following [42]:
 - o **Public Subnet:** the subnet traffic is routed to the public internet through an internet gateway or an egress-only internet gateway;
 - o **Private Subnet:** the subnet traffic can't reach the public internet through an internet gateway or egress-only internet gateway. Access to the public internet requires a NAT device. In this case, a NAT Gateway;
- **NAT Gateway:** A NAT gateway is a Network Address Translation (NAT) service. You can use a NAT gateway so that instances in a private subnet can connect to services outside your VPC, but external services cannot initiate a connection with

those instances [43]. Essentially, a one-way means of network connectivity which allows elements within a VPC to communicate with external sources, but protects these from external messages;

- **Internet Gateway:** An internet gateway is a horizontally scaled, redundant, and highly available VPC component that allows communication between a VPC and the internet. An internet gateway enables public subnets to connect to the internet if the resource has a public Ipv4 address or an Ipv6 address.

Figure 17 above presents the architectural organization of the elements responsible for conducting network connectivity and communications between components. Each subnet has a CIDR block of addresses which components deployed into them will take ownership of. Each subnet communicates through an AWS EC2 Route component which, according to a Public Subnet Route Table, performs distributed communications. These Route Tables communicate with a gateway which depending on the type of subnet that is being discussed. When provisioning a public subnet, the communications are made directly with an Internet Gateway, which consequently gives access to the internet; when provisioning a private subnet which needs to communicate with external components, a NAT gateway must be provisioned as a middleman between the route table and the internet gateway, guaranteeing one-way communications.

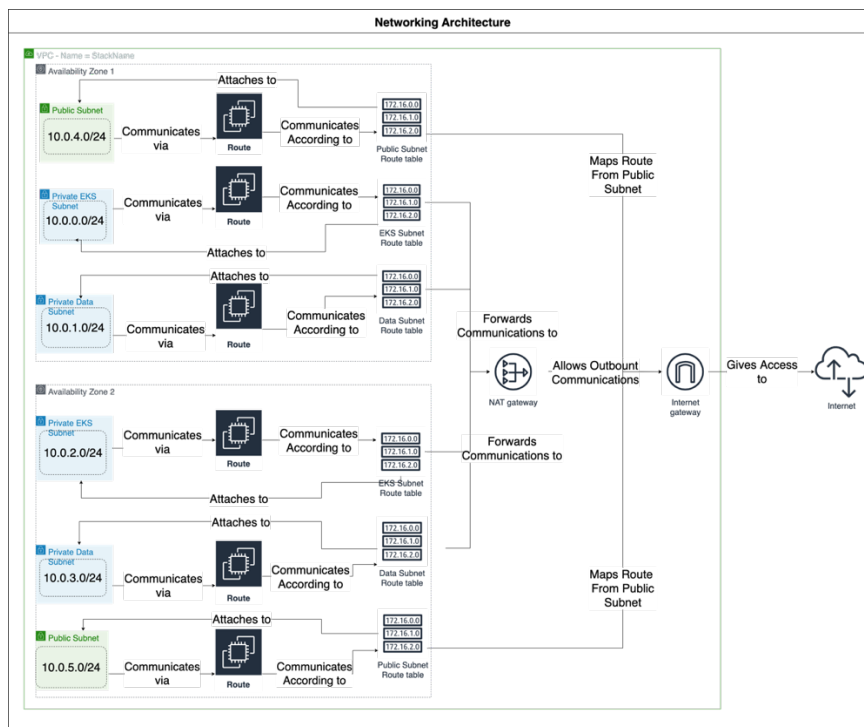


Figure 56. Networking Architecture Diagram

Subnets - As depicted in Figure 17, there will be a total of six subnets provisioned in the cloud infrastructure. The following Figure 18 represents the components that are provisioned in the Private EKS Subnet. Here, the Elastic Kubernetes Cluster will be running, with the computational resources being allocated into an autoscaling EKS NodeGroup. This node group will comprise of four EC2 instances, along for the Cluster Logging API, responsible for managing and orchestrating the components that are deployed into the cluster, the EKS Role, which is an AWS IAM Role which gives the EKS service permission to access and manipulate the different services contained in the cluster, including the EC2 instances and the containers belonging to the application. The

cluster security groups represent the inbound and outbound communications that are allowed between the cluster, the Bastion Hosts and the RDS Application.

The following Figure 19 serves as a graphical representation of the components that are deployed in the public subnets. Each subnet will have an EC2 instance running Linux to serve as the bastion host to access the EKS cluster, with this EC2 instance assuming the Bastion IAM profile which follows the BastionPolicy, along with a LogGroup which allows to maintain track of the logs created in the Bastion Host. Furthermore, the Bastion host will have an Elastic IP Address so as to identify the virtual machine within the subnet, whilst also making use of the previously provisioned Bastion Security group to perform access filtering and communication management with the instance.

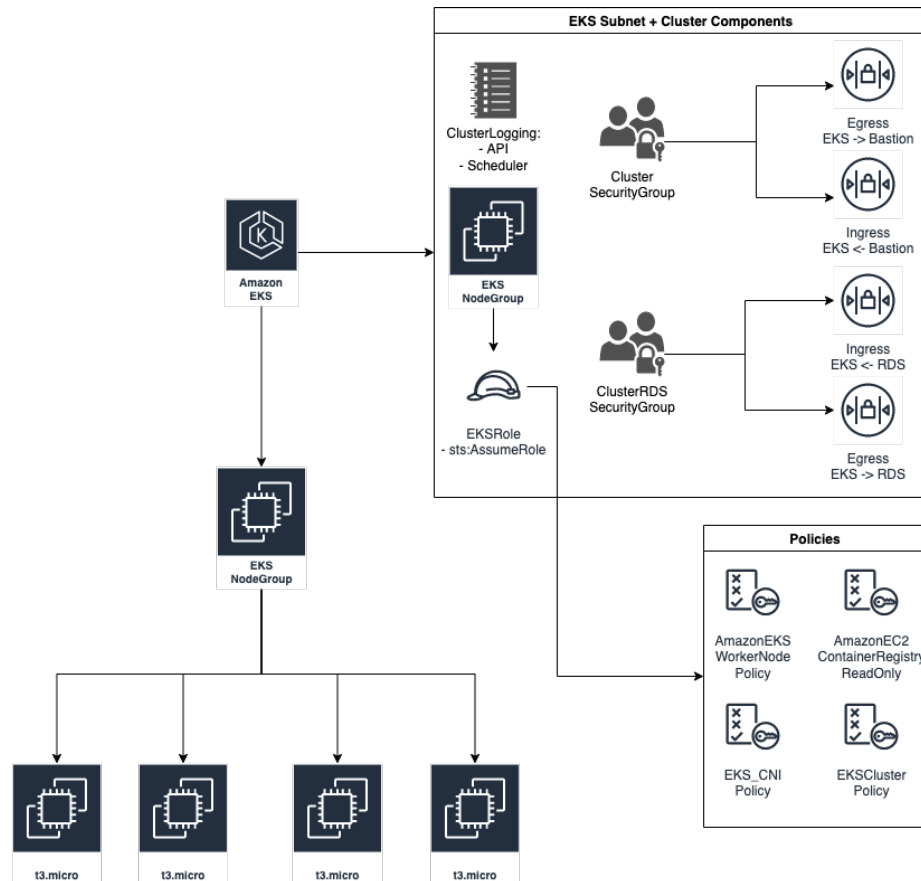


Figure 57. EKS Subnet Components

The Bastion EC2 instance must receive its own configuration. Given the scale of the project and the need to maintain resource consumption to the bare essential, the Linux image running on the machine is an Amazon Linux Image, running on a t2.micro instance. Both the image and the instance are free-tier eligible, therefore, the provisioning of these machines has no cost associated. The configuration must include the security group to which the instance will belong, along with the subnet ID in which the machine will be deployed.

Description of figure 22.

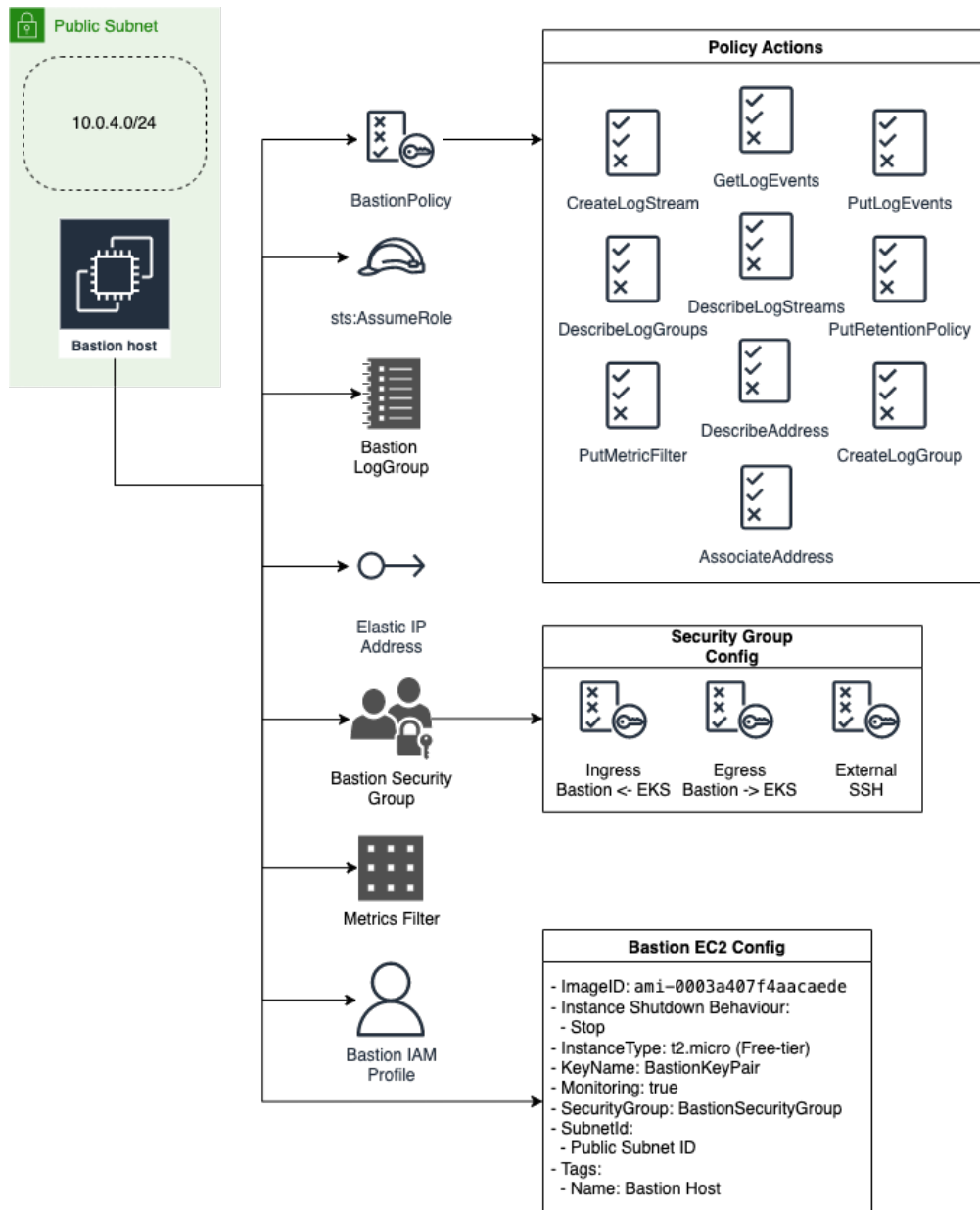


Figure 58. Public Subnet Component Architecture

Security Groups - A security group acts as a virtual firewall, controlling the traffic that is allowed to reach and leave the resources that it is associated with. For each security group, *rules* are added that control the traffic based on protocols and port numbers. There are separate sets of rules for inbound traffic and outbound traffic. The infrastructure that is created will have the following security groups:

- **VPC Security Group:** default VPCs and any VPCs that are created come with a default security group. With some resources, if they are not associated with a security group when created the resource, AWS associate these resources with the default security group;
- **Bastion Security Group:** this Security Group is responsible for guaranteeing both which external elements have permission to access the EC2 instance and via which protocol (in this case, a range of IP addresses that belong to the organizations network

via an SSH connection) and identifies the host itself when looking to communicate with other components in the Availability Zone;

- **EKS Security Group:** the EKS Security Group defines which components are permitted to access the different nodes within the EKS Cluster. This Security Group allows connections from the elements within the Bastion Security Group, exclusively.

AWS EKS Cluster - Amazon Elastic Kubernetes Service (Amazon EKS) is a managed service that you can use to run Kubernetes on AWS without needing to install, operate, and maintain your own Kubernetes control plane or nodes [44]. This service runs and scales the Kubernetes control plane across multiple AWS Availability Zones to ensure high availability, whilst also automatically scaling control plane instances based on load, detecting and replacing unhealthy control plane instances, and it provides automated version updates and patching for them. It is also able capable of running up-to-date versions of the open-source Kubernetes software, so that all the existing plugins and tooling from the Kubernetes community can be used by developers. The application's distributed communication and orchestration will be managed by the EKS service.

AWS Elastic Load Balancer - The Elastic Load Balancer (ELB) is an intermediary entity which automatically distributes incoming application traffic across multiple targets and virtual components in one or more Availability Zones. The ELB allows for application scaling without requiring complex configurations. An Application Load Balancer makes routing decisions at the application layer (HTTP/HTTPS), supports path-based routing, and can route requests to one or more ports on each container instance in the cluster. For the product of this internship, the ELB will be responsible for distributing and controlling the traffic made to the application, routing the requests to the different nodes of the EKS cluster.

AWS WAF – The Amazon Web Application Firewall (WAF) component is responsible for protecting web applications from common web exploits that may have an effect on the availability or security of the information that is manipulated by the application [45]. It also allows control over the flow of traffic by enabling the possibility of creating security rules that control bot traffic and block common attack patterns, such as SQL injection or cross-site scripting. In the case of the infrastructure that is created, the WAF will be used to filter the request that can be made to the application according to request origin's IP address. The WAF has a direct association with the Load Balancer which controls and manages the flow of requests to the different instances of the application.

AWS Route53 – Given the nature of the application being a website, the AWS Route 53 service is a highly available and scalable service that offers the possibility of Domain Name System (DNS) translation [46]. It is designed to give developers and businesses an extremely reliable and cost-effective means of routing end users to Internet applications by connecting user requests to infrastructure running in AWS and can also be used to route users to infrastructure outside of AWS. In the case of this internship, the service re-routes traffic made to the domain “awsdummy.online” to the endpoint associated with the Elastic Load Balancer, incorporating the HTTPS protocol.

Continuous Integration/Continuous Delivery

The contents covered in this section focus on the elements that are responsible for conducting the tasks of Continuous Integration and Continuous Delivery of the framework. In this component of the framework, the source code of the application will

have to be built into an executable artifact, with automated tests being performed both on the source code and the artifact itself. Once the Continuous Integration phase concludes with the creation of the artifact, the product must then be deployed autonomously into the production environment.

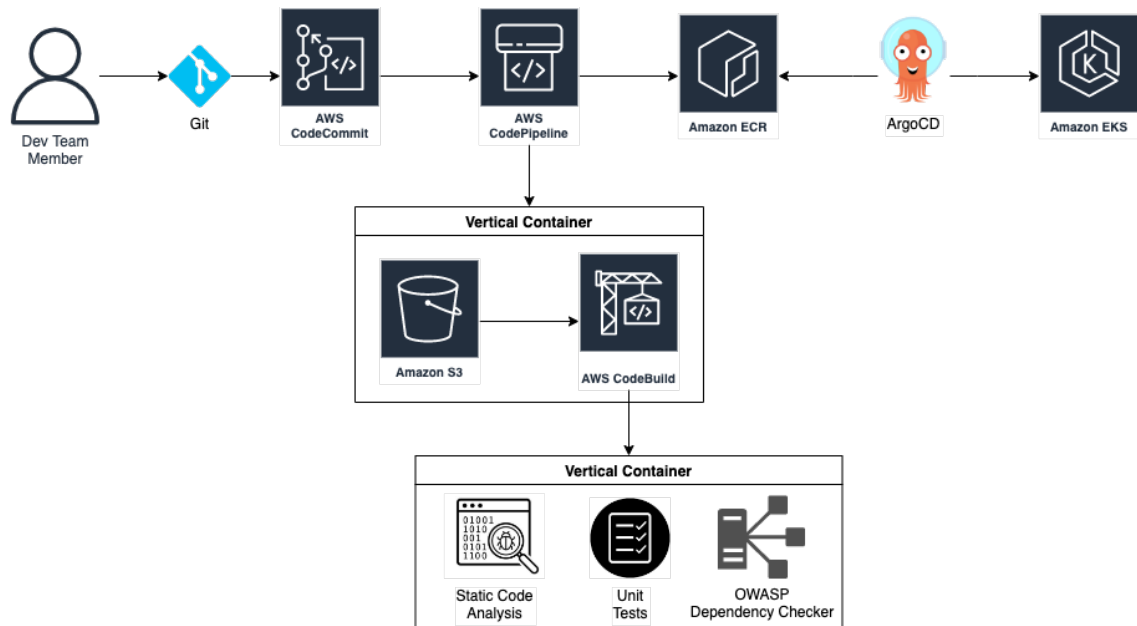


Figure 59. Macro-view of the CI/CD Pipeline Process

The framework is ready to incorporate multiple alterations to application's source code and perform functional tests, static code analysis and dependency vulnerability analysis on the code in order to foresee possible errors that may occur and avoid their propagation in production. If this phase is successful, an artifact is produced and made available as a Docker Container Image and kept in an AWS Elastic Container Repository (ECR). The second phase, Continuous Delivery, is implemented in a manner that automated deployments of the Docker Container Image can be performed in a progressive manner. When a new version of the docker image is committed to the ECR, the phase CD phase begins, where the tool responsible for carrying out the deployment pulls the image from the repository and deploys in progressively into the EKS cluster.

Identity Access Management

In the case for this framework, the Development Team leader, Development Team Member and possible third-party entities will have different access to different components within the framework from the Infrastructure Provisioning point of view, with each having their own IAM Role which will be assumed before adopting the framework. These profiles must be provisioned prior to the using the framework in a team. Those three profiles will be the following:

- Development Team Leader: role reserved for development team leaders with permissions to read and write information to the AWS CloudFormation service to create the pre-requisites for the development pipeline, along with full access to the AWS CodeCommit service;
- Development Team Member: role reserve for development team members with permissions to read and write source code files to and from the AWS CodeCommit repository associated with the development project;

- Third-Party Consultant: role reserved for read-only access to the AWS CodeCommit repository for possible third-party elements that may offer consultation or third-party expertise.

The following figure 60 offers a graphical representation of the organization behind the user roles and their respective policies associated with the AWS CodeCommit repository.

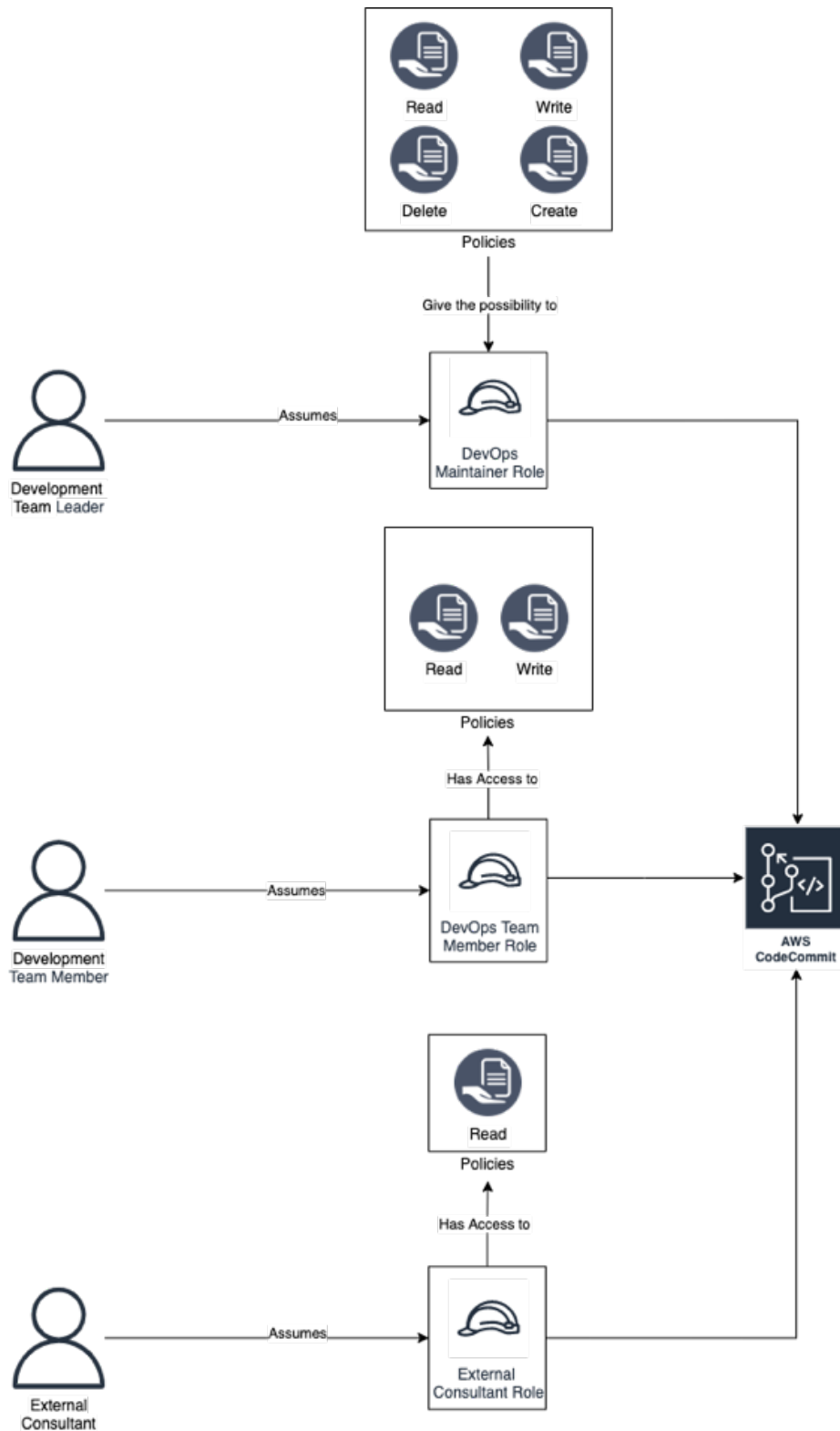


Figure 60. IAM Organization for the Software Development Team

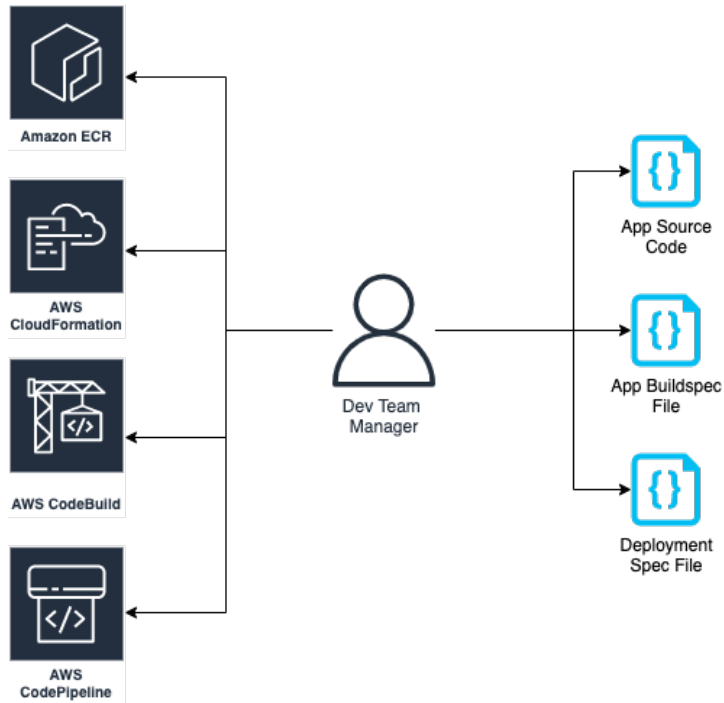


Figure 61. Dev Team Leader Responsibilities

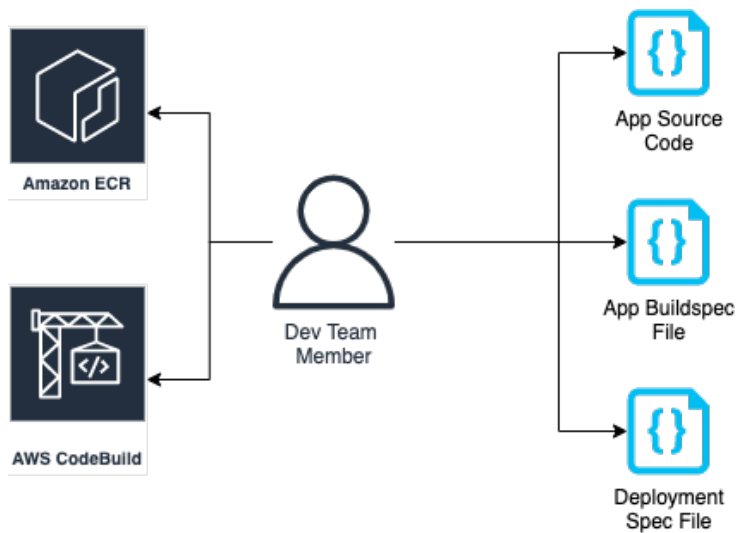


Figure 62. Dev Team Member Responsibilities

Pre-Requisites

Similarly to the Infrastructure Provisioning component of the framework, the CI/CD phase also requires that a number of pre-requisites be provisioned before developers can make use of the autonomous functionalities that the pipeline has to offer. The pre-requisites that must be provisioned are the following:

- **AWS CodeCommit:** git repository where the application source code will be maintained and versioned to control alterations and updates that may occur throughout the development process. This will serve as a single point of truth for the elements that require a source to extract the configuration files from (AWS CodeBuild and Argo CD);

- **AWS Lambda Functions** – 2 Functions (events + emails):
 - **CreatePipelineLambdafunction:** this function is responsible for transferring the template files from the AWS CodeCommit repository to the Amazon S3 bucket and the creation of the AWS CodePipeline that will be the responsible for detecting the changes in the repository and triggering the AWS CodeBuild project which will automate the static code analysis, dependency verification and unit testing on the application source code. These pipelines will only be created when a branch named develop, staging, lab or main is created in the repository;
 - **SendOWASPEmailLambdaFunction:** this function is responsible for extracting the OWASP dependency report from the S3 Bucket responsible for maintaining a persistent version of the report, and emails that same report to the Development Team Leader via email.
- **AWS S3 Buckets** – 2 Buckets (events + emails):
 - **Source Code Bucket:** S3 bucket which stores the application source code and acts as a source for the AWS CodePipeline service which performs the autonomous tasks of Continuous Integration;
 - **OWASP Report Bucket:** S3 bucket to which the dependency verification report is sent and maintained. This bucket is separate to the source code bucket to maintain modularity and separation between different concepts.
- **AWS IAM StackChangeRole Role:** AWS IAM Role responsible for giving AWS CodePipeline the permissions necessary for creating and performing changes on the AWS CloudFormation stacks;
- **AWS IAM CodePipelineRole Role:** AWS IAM Role which gives permissions to the AWS CodePipeline component to perform changes and access information present in the previously mentioned S3 Bucket;
- **AWS IAM LambdaRole Role:** AWS IAM Role responsible for allowing the AWS Lambda functions to have read-only access to the AWS CodeCommit repository, to the S3bucket, and read and write access to the AWS ECR service, to the AWS Systems Manager service and to the CodeBuild service to perform changes during the Continuous Integration phase;
- **AWS IAM CodeBuildRole Role:** AWS IAM Role responsible for allowing the AWS CodeBuild Projects to have read and write access to the AWS CodeCommit repository, to the S3 bucket, and read and write access to the AWS CloudFormation to create the pipeline responsible for provisioning the infrastructure;
- **AWS CloudWatch Event Rule:** event listener which detects changes to the AWS CodeCommit repository in terms of creating and deleting branches, so as to allow infrastructure replication, and communicates those changes to the AWS Lambda function to create a pipeline at the rate that it is necessary to replicate an environment;
- **AWS Lambda Permission:** permission entry which gives the CloudWatch Event Rule permissions to trigger the AWS Lambda functions when changes are performed to the repository or when a new version of the OWASP report is produced.

Figure 22 shows the process required to provision these pre-requisites before making use of the CI/CD pipeline. A Development Team Leader, duly authenticated with an AWS Dev Team Leader account, deploys the contents of the AWS CloudFormation stack in the

PreRequisites.yaml file with the same command as in the previous Infrastructure Provisioning Template, with the correct adjustments to not cause conflicts with the previous pre-requisite stack. An example of this command can be the following:

- aws cloudformation create-stack \
 - stack-name CI_PRE_REQ_STACK \
 - template-body file://PreRequisites.yaml \
 - capabilities CAPABILITIES_NAMED_IAM

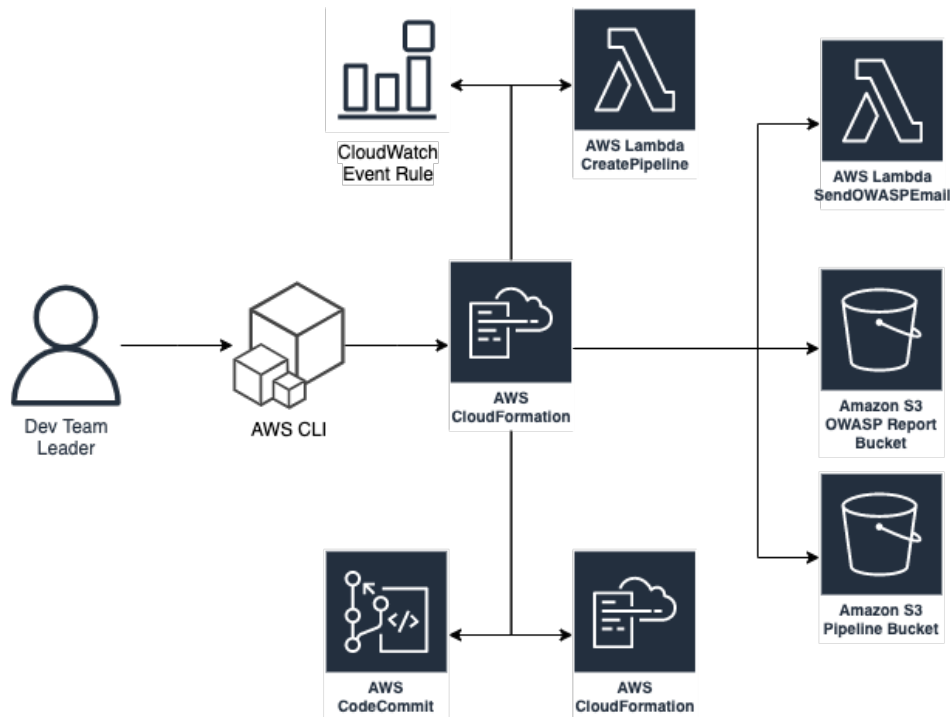


Figure 63. Pre-Requirement Elements Created by the Development Team Leader

This command will output the ID of the AWS CloudFormation stack that is created. The value for CI_PRE_REQ_STACK is customizable, the file name PreRequisites.yaml must match the name of the pre-requisites template and the parameters which are defined in the template file can be overridden by using the `-parameters` flag, just as in the Infrastructure Provisioning Pre-Requirement phase.

Continuous Integration Process

The software development process begins at the developer's end. The software developer performs changes to the application's source code to perform fixes on errors and bugs that may exist or to implement new functionalities that may be required to add more value to the application. The diagram in Figure 23 resembles the process once the developer performs a change to the application's source code AWS CodeCommit repository, be it a new commit, new branch or new merge between branches. In the event of a new branch being created with one of the aforementioned names, the AWS CloudWatch Event Rule detects this change and communicates with the AWS Lambda service to invoke the function CreatePipelineLambdafunction to create a pipeline which begins with the new branch and extends to the AWS CodeBuild project. Each branch has its own AWS CodePipeline service, with its own individual CodeBuild

Project. There is no need for an individual S3 Bucket for each branch, as it is possible to create directories which will be associated with each branch.

When a new commit is performed, the AWS CloudWatch Event Rule communicates with the CreatePipelineLambdafunction function which receives an event that communicates the event of a new commit being made. The function then clones the files in the repository, compresses them into a .zip file, and transfers the compressed files to the AWS S3 Bucket. This process is required due to the fact that the CodePipeline element, when created with AWS CloudFormation templates, requires an AWS S3 Bucket as its source, and the files in the Bucket must be in a .zip file to be versioned and managed by the bucket itself. The new version of the .zip file is detected by the CodePipeline service and proceeds to trigger the AWS CodeBuild project associated with the respective branch.

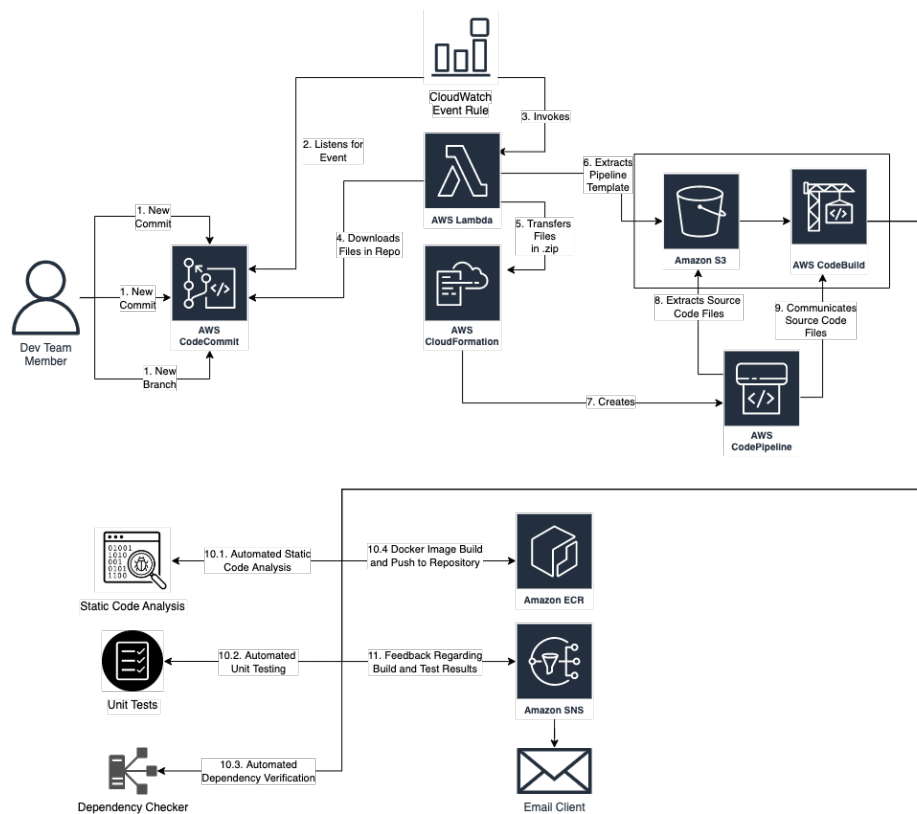


Figure 64. Continuous Integration Workflow Diagram

Figure 23 above shows the process that was previously described, and also the steps that are followed after the AWS CodeBuild project is responsible of carrying out. As the project is deployed, the Development Team Leader receives an email notifying that the build began. The Build first analyses the source code with a static code analysis tool to guarantee that coding conventions are followed correctly and that it is ready to be compiled. In the case of the example application source code which is written in Node.js, the tool used is JavaScript-Lint [47]. If an error occurs, the build fails, and the Development Team Leader receives an email with this information. This notification service performs this action in all situations where the build changes its state (“IN PROGRESS”, “FAILED”, “STOPPED”, “SUCCESSFUL”).

If this process is successful, the OWASP Dependency Checker [48] performs an analysis of all the libraries and modules that are used by the application. This tool performs a cross-reference analysis of the application with all known vulnerabilities that may exist

in third-party software, with a database that extends to vulnerabilities detected since its first released version in 2013. When the dependency checker fulfils its analysis of the application, a report is produced which explains where vulnerabilities are and if there is a possibility for external attack or data leaks. This event in the AWS CodeBuild project exports this report to the OWASP Report Bucket, followed by triggering the SendOWASPEmailLambdaFunction which transfers the report file from the bucket and emails it to the Development Team Leader.

If there are no vulnerabilities detected, the application's code is then subjected to unit tests to its various functions. These unit tests are conducted by the Unit.js library which runs on the Node.js browser alongside the application and tests the output of the various functions. Once complete, the AWS CodeBuild project then builds a Docker Container Image according to a Dockerfile which is included with the application's source code. This image is then forwarded to the AWS ECR repository, and the build phase is concluded.

Having concluded this process, the executable is then placed into an Elastic Container Repository from which the image of the service is then pulled by the Elastic Kubernetes Service and deployed into the target environment. Considering that the need for phased rollouts, having finalized and passed all previous tests, the container image will then be transferred into a production ECR from which, when instructed, the production cluster will pull the new version of the application and apply its changes.

The Activity Diagram in Figure 24 and the Sequence Diagram in Figure 25 serve the purpose of showing the different steps in the Continuous Integration phase of the pipeline, when a change is made to the source code of the application. Beginning with the commit on behalf of the developer, the source code is pushed to the AWS CodeCommit repository and then extracted by the Lambda function and uploaded to the S3 bucket. The AWS CodeBuild project then begins the build process of static code analysis, vulnerability verification and unit tests on the source code, and then builds and pushes a new Docker Container Image to the AWS ECR repository.

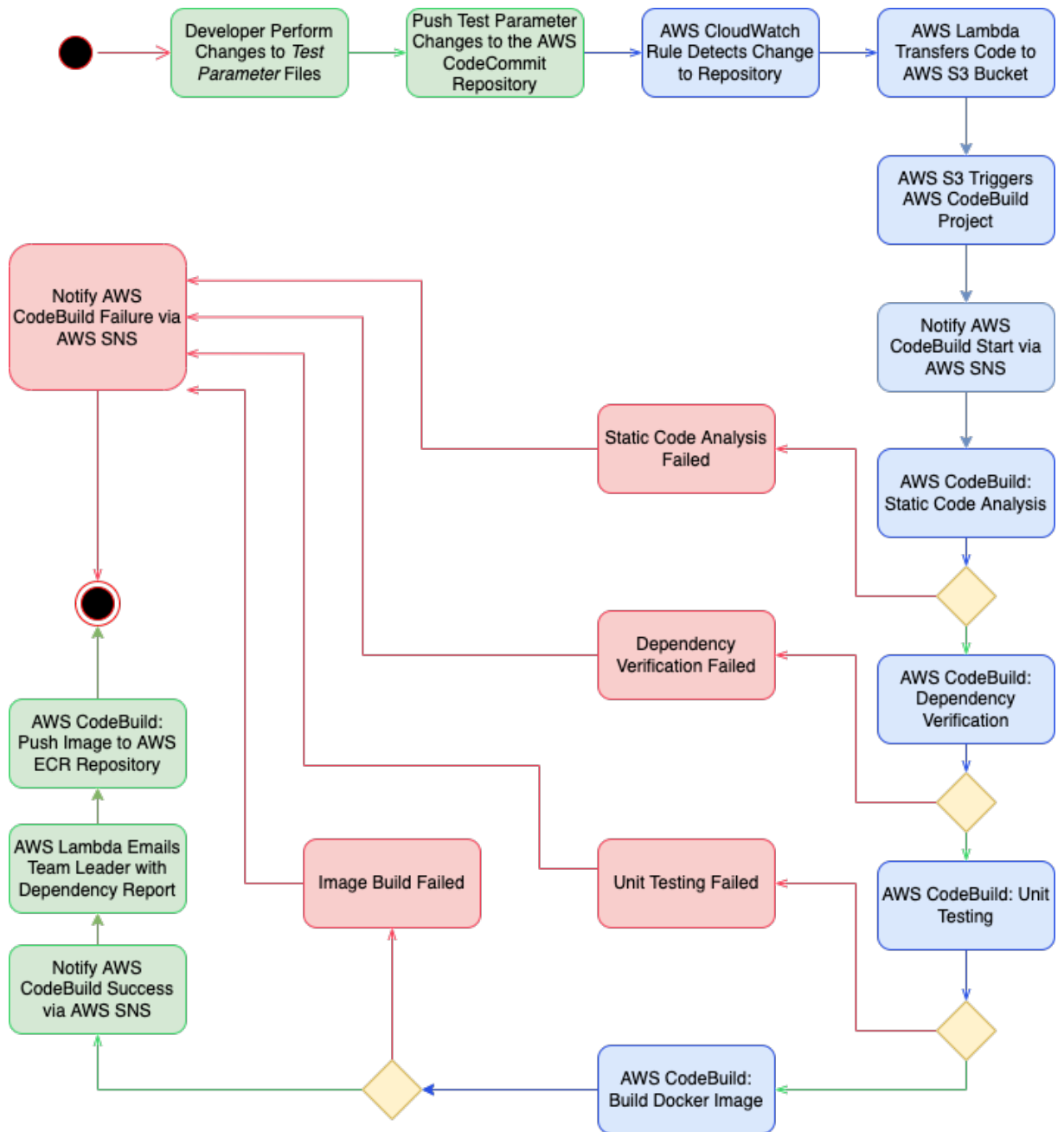


Figure 65. Continuous Integration Automated Test Definition Integration Activity Diagram

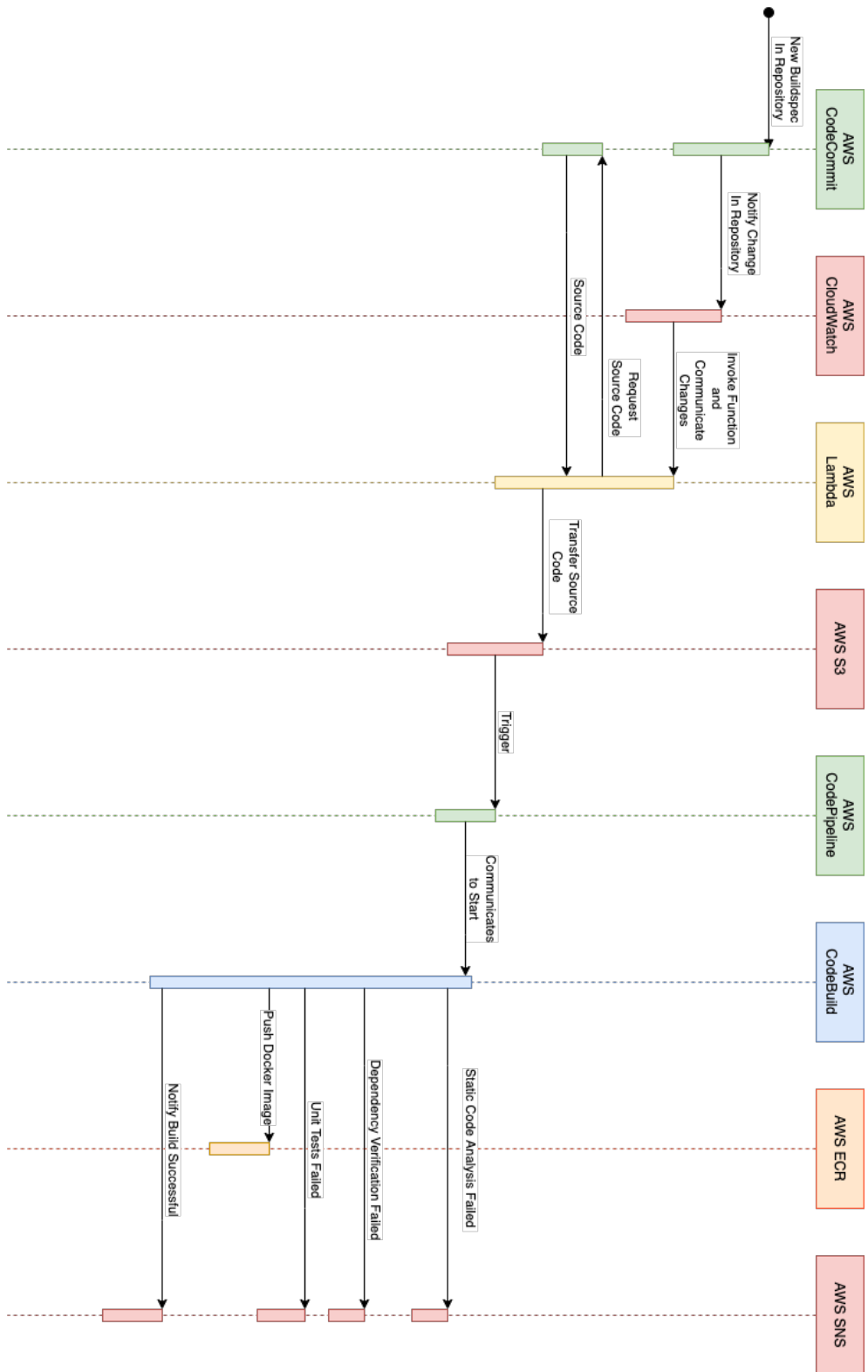


Figure 66. Process Changes to Repository Sequence Diagram

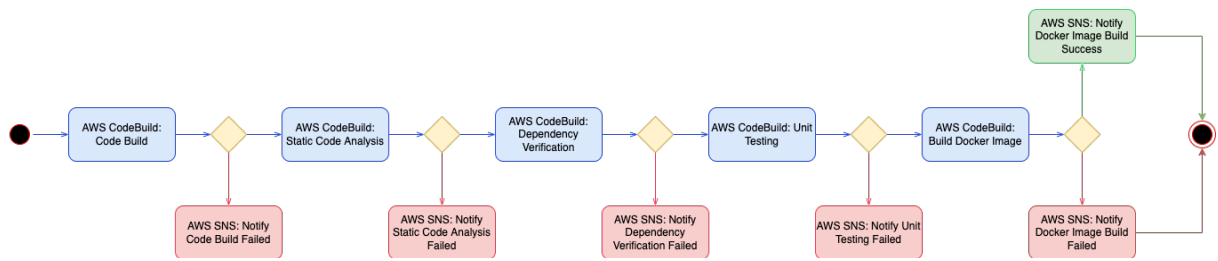


Figure 67. System Status Alert for CI Phase Activity Diagram

Continuous Delivery Process

The process behind Continuous Delivery begins still during the AWS CodeBuild project execution. Once the build phase is complete and the new docker image has been pushed to the ECR repository, the post-build phase begins with an authentication of the AWS CodeBuild which extracts the account credentials from AWS Systems Manager.

The diagram in Figure 26 represents this workflow, including the CI phase, in efforts to show how the software development pipeline is fully automated and extends from the software developer committing their changes to the source code through to the EKS cluster, without the need for manual intervention. If required, this process can be paused and configured in a way that approval steps may be in place.

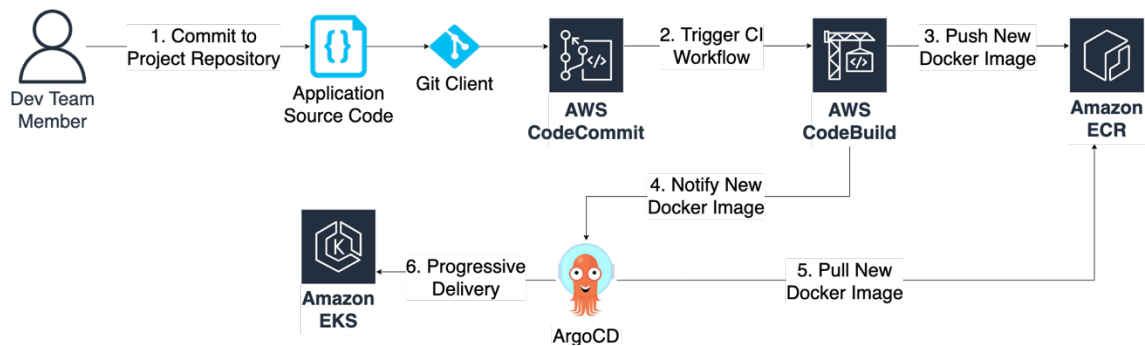


Figure 68. CI/CD Workflow Production Diagram.

Argo CD allows the possibility to deploy the new image using blue-green deployments, A/B Testing or Canary Releases, depending on the intended deployment method. Once the AWS CLI tool is authenticated, it then notifies the Argo CD tool that a new version of the application is ready to be deployed into production and that the application project requires synchronization. This new version is pulled by Argo CD and then deployed progressively according to the rollout’s configuration file, which is kept in the source code repository.

Argo CD - Argo CD is an open-source continuous delivery tool, which follows the GitOps pattern of using Git repositories as the source of truth for defining the desired application state. Argo CD automates the deployment of the application according to the specified target environments. Argo CD can track updates to branches, tags, or a specific version of manifests in a Git commit.

Argo CD is implemented as a Kubernetes controller which is continuously monitoring applications and compares their current state against the desired target state specified in the manifest file which is kept in the AWS CodeCommit repository. A deployed application whose current state deviates from the target state is considered “OutOfSync”.

Argo CD reports & visualizes the differences, while providing a means of automatically or manually synchronizing the live state back to the desired target state. Any modifications made to the desired target state in the Git repo can be automatically applied and reflected in the specified target environments.

Figure 27 serves as a graphical representation of how Argo CD operates from an architectural [49] point of view. Argo CD makes use of a REST API server is deployed into the Kubernetes cluster and connects to the Kubernetes server, offering application management and status reporting, invokes application operations, repository and credentials management and listens for Git webhook events as a trigger.

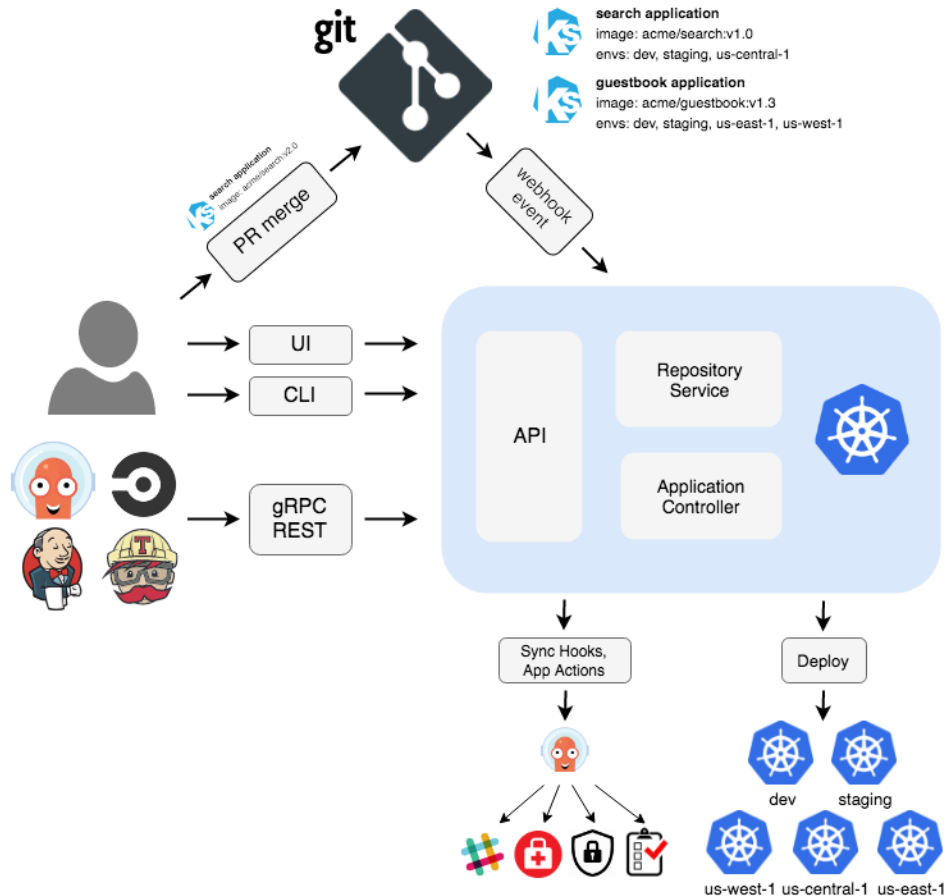


Figure 69. Argo CD Architecture Overview [50]

Argo CD makes use of a repository server [49] which is an internal service that maintains a local cache of the Git repository holding the application manifests. It is responsible for generating and returning the Kubernetes manifests.

Finally, Argo CD also has an application controller [49] which is a Kubernetes controller that continuously monitors running applications and compares the current, live state against the desired target state (as specified in the repo). It detects the “OutOfSync“ application state and optionally takes corrective action. It is responsible for invoking any user-defined hooks for lifecycle events.

Argo Rollouts - Argo Rollouts is a Kubernetes controller [51] and set of CRDs which provide advanced deployment capabilities such as blue-green, canary, canary analysis, experimentation, and progressive delivery features to Kubernetes. Argo Rollouts integrates with ingress controllers and service meshes, performing changes to the traffic

they receive and offers the possibility to gradually shift traffic to the new version during an update.

Argo Rollouts controller will manage the creation, scaling, and deletion of ReplicaSets, these being defined by the `spec.template` field inside the Rollout resource, which uses the same pod template as the deployment object [52]. When the `spec.template` is changed, that signals to the Argo Rollouts controller that a new ReplicaSet will be introduced into the AWS EKS cluster and the controller will use the strategy set within the `spec.strategy` field in order to determine how the rollout will progress from the old ReplicaSet to the new ReplicaSet. Once that new ReplicaSet is scaled up and the controller will mark it as "stable".

Figure 28 shows an example of an Argo Rollouts configuration file. As previously mentioned, the `spec` field is responsible for defining how the rollout must be carried out. In the case of the code in Figure 28, there must be five replicas of the application present in each ReplicaSet, the release strategy is a Canary Release with five different steps that must take place. The rollout would begin with re-routing 20% of the traffic between the two new versions, during an unlimited time. This occulting of the duration allows for dynamic testing on the application such as acceptance tests, performance evaluation or possible bug detection that may have not been caught during the Continuous Integration phase. Once the deployment is considered ready to advance, a Development Team Manager will inform Argo CD to promote the deployment to its next step.

```
deployment.yaml 1 X
CI Files > CICDPipelineProjectRepository > deployment > deployment.yaml > {} spec > {} strategy > {} canary > [ ] steps
io.k8s.api.core.v1.Service (v1@service.json)
1  apiVersion: argoproj.io/v1alpha1
2  kind: Rollout
3  metadata:
4    name: my-app-rollout
5    labels:
6      #app: guestbook
7      app.kubernetes.io/instance: my-canary
8  spec:
9    replicas: 5
10   strategy:
11     canary:
12       steps:
13         - setWeight: 20
14           pause: {}
15         - setWeight: 40
16           pause: {duration: 1200}
17         - setWeight: 60
18           pause: {duration: 1200}
19         - setWeight: 80
20           pause: {duration: 1200}
21       analysis:
22         templates:
23           - templateName: success-rate
24             startingStep: 3
25         args:
26           - name: service-name
27             value: guestbook
28           - name: vrs
29             value: '0.1.99'
30     revisionHistoryLimit: 0
31   selector:
32     matchLabels:
33       app: guestbook
34       tier: my-app-rollout
35   template:
36     metadata:
37       labels:
38         app: guestbook
39         tier: my-app-rollout
40     spec:
41       containers:
42         - name: 0-1-99
43           image: 168473906286.dkr.ecr.eu-west-3.amazonaws.com/testinfraprojectname-testprojectecr:0.1.99
```

Figure 70. Example of an Argo Rollouts Configuration File

The following steps contain a “duration” field, which acts as a timer between increments of the traffic between the two versions. If there is no issue with the new version, the counter will decrease until reaching zero, at which point the traffic between the versions will then pass from, for example, 40% to 60% between the new version of the application and the previous version, respectively.

Finally, if another change occurs in the `spec.template` during a transition from a stable ReplicaSet to a new ReplicaSet (i.e., you change the application version in the middle of a rollout), then the previously new ReplicaSet will be scaled down, and the controller will try to progress the ReplicaSet that reflects the updated `spec.template` field.

Automated Rollbacks - When looking to perform continuous delivery of software, there must be a method of recovering from failure, without causing downtime to the application. Argo Rollouts further offers the possibility of performing an analysis of the metrics of the application as means of performing an automated rollback in the event of the new version failing or producing errors in production.

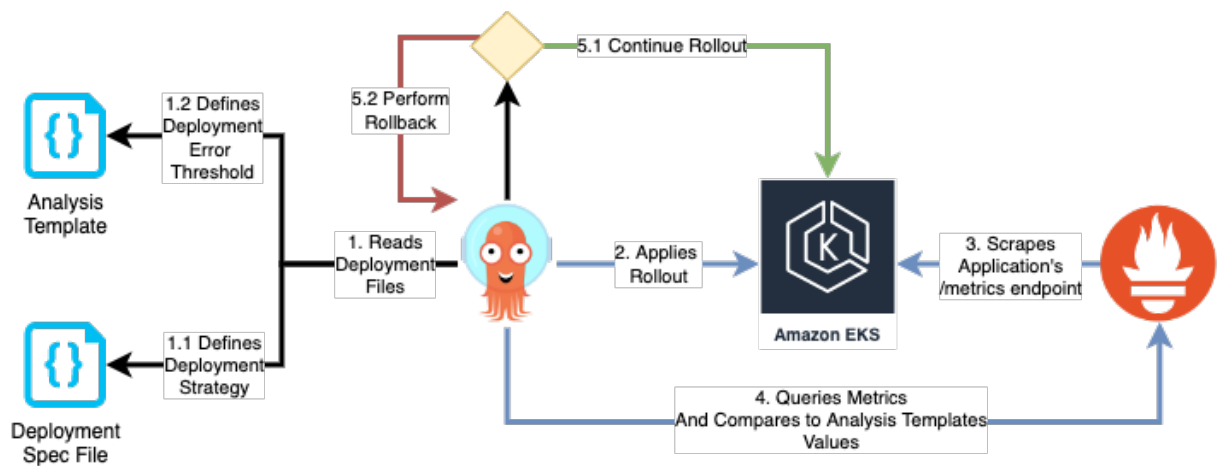


Figure 71. Automated Rollback Workflow

Figure 29 is an example of an AnalysisTemplate object which is responsible for carrying out an evaluation of the application’s metrics during a new release. In the example, under the `spec.metrics` field, every 15 seconds, there is a query made to the metrics server which is exposed by the application. These metrics are harboured in a Prometheus instance, which will be explained in further detail in the next section. In the event of the result of the query not adhering to the “`successCondition`” after 10 attempts, the version is automatically rolled back to the previous version, with the release of the new version being labelled as “degraded”. The query made returns the sum of requests that return a code 400 request, divided by the total amount of requests made to the application. Given that Prometheus returns an array when returning results for a query, the first position is reserved for the number result of the query itself, and therefore if the amount of code 400 requests oversteps the 10% mark, the release is cancelled, and a rollback takes place.

```
deployment.yaml 1 X
CI Files > CICDPipelineProjectRepository > deployment > deployment.yaml > ...
io.k8s.api.core.v1.Service (v1@service.json)
1  apiVersion: argoproj.io/v1alpha1
2  kind: AnalysisTemplate
3  metadata:
4    name: success-rate
5  spec:
6    args:
7    - name: service-name
8    - name: vrs
9    metrics:
10   - name: success-rate
11     interval: 15s
12     count: 10
13     # NOTE: prometheus queries return results in the form of a vector.s
14     # So it is common to access the index 0 of the returned array to obtain the value
15     successCondition: result[0] >= 0.1
16     provider:
17       prometheus:
18         address: http://ada5b2168fb084abc8cf4be8e05dd40e-833956970.eu-west-3.elb.amazonaws.com:80
19         query: sum(code_400_count{version="0.1.99"})/sum(code_X_count{version="0.1.99"})
20
```

Figure 72. Canary Releases Delivery Model Sequence Diagram

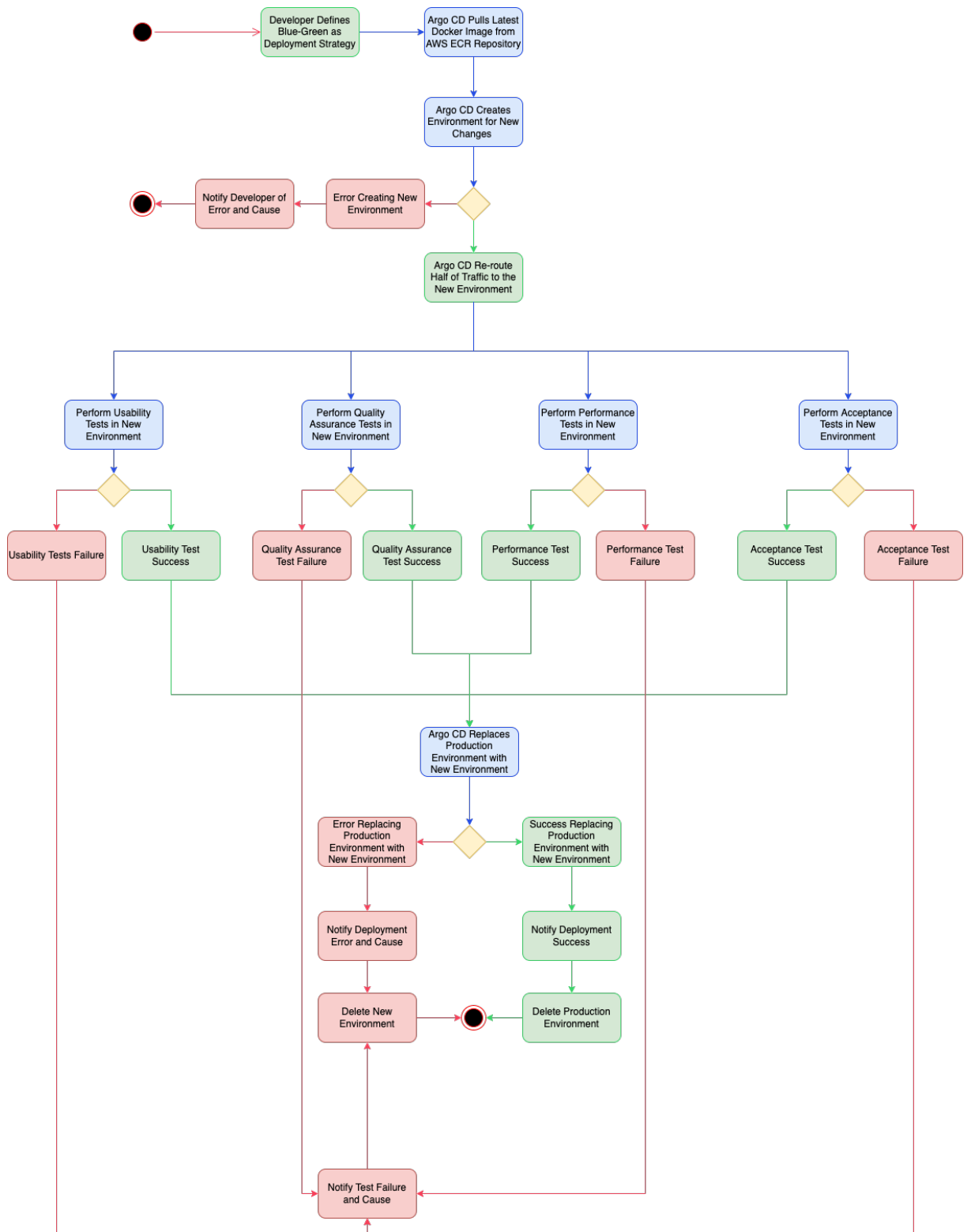


Figure 73. Blue-Green Deployments Activity Diagram

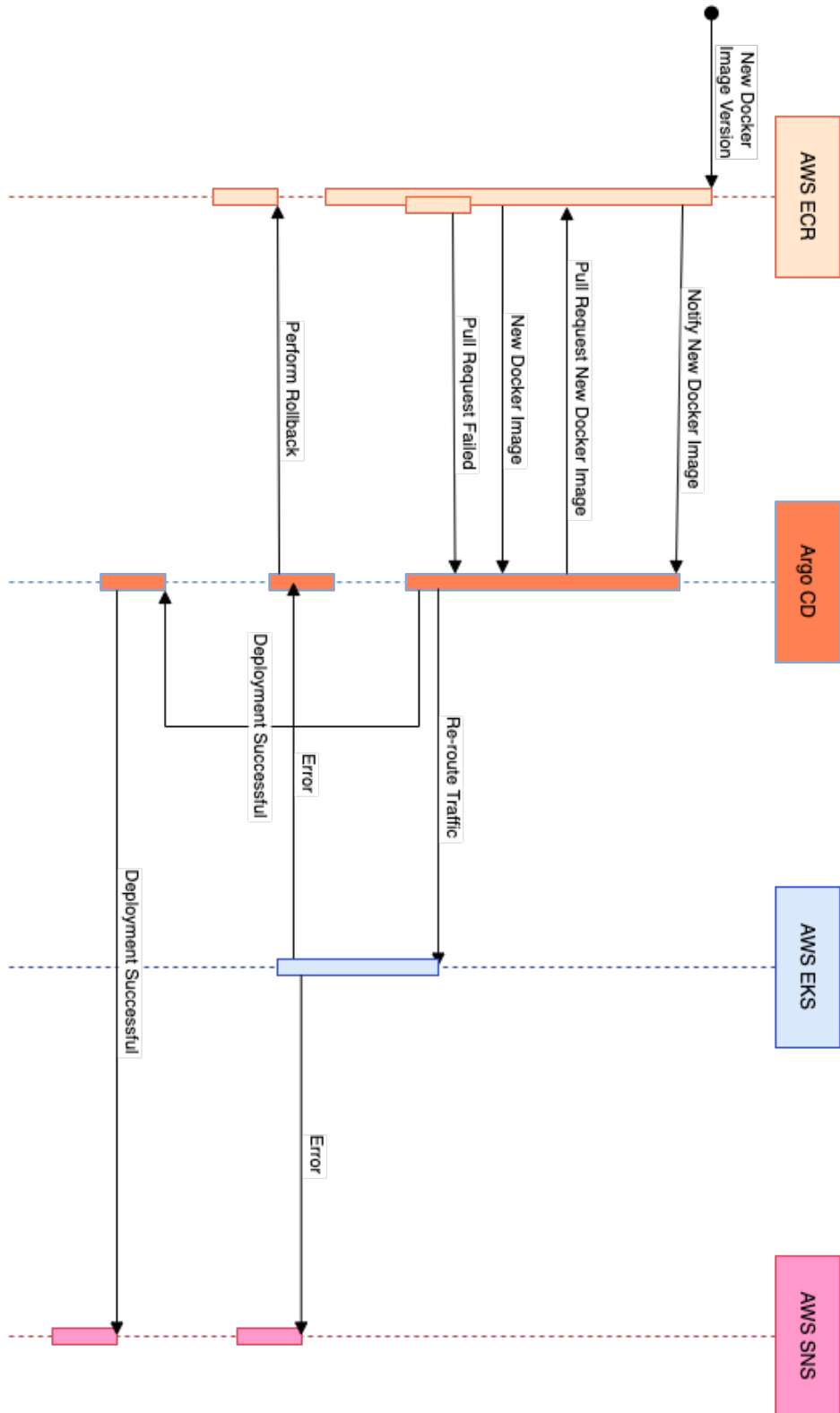


Figure 74. Blue-Green Deployment Sequence Diagram

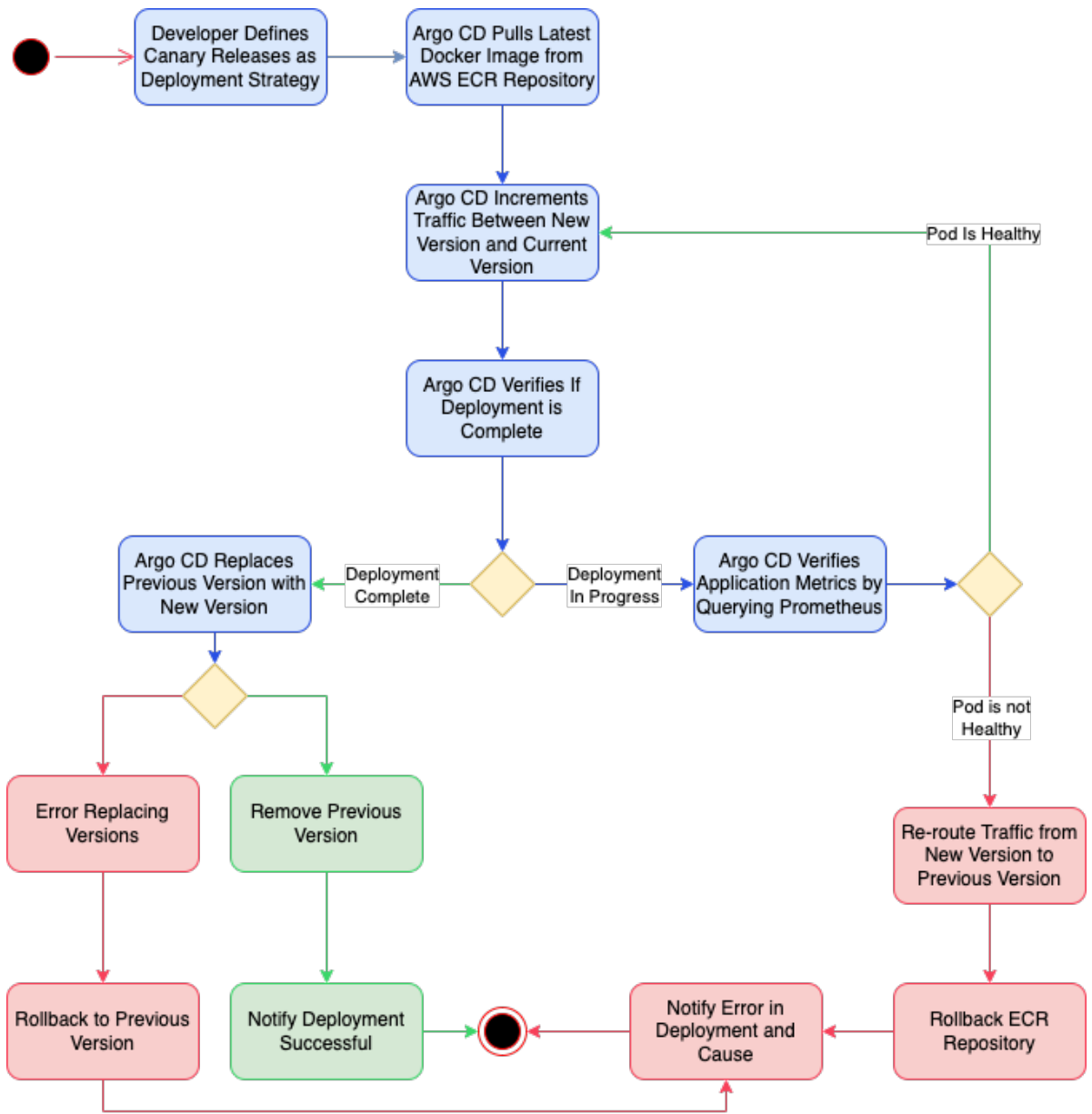


Figure 75. Canary Releases Activity Diagram

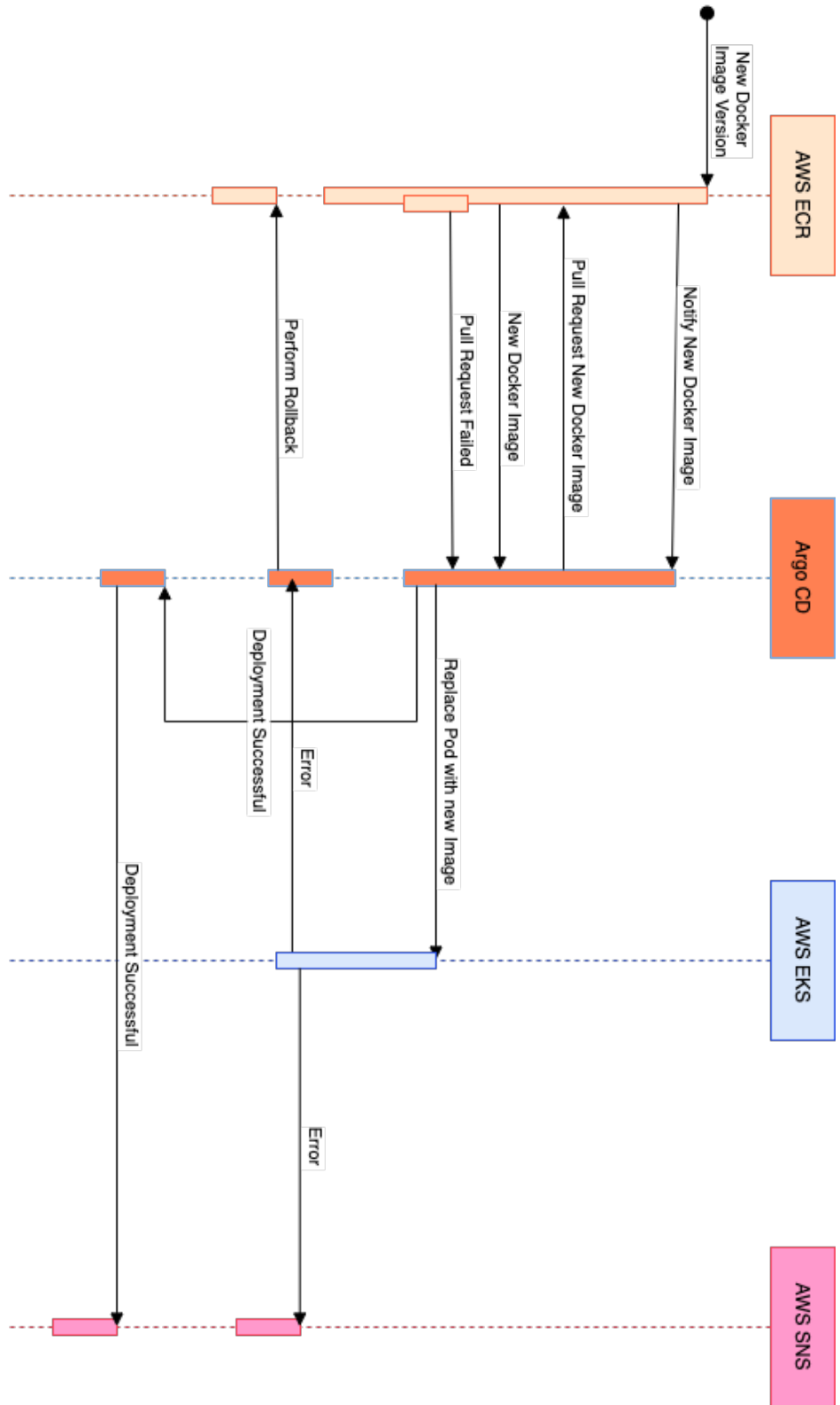


Figure 76. Canary Releases Sequence Diagram

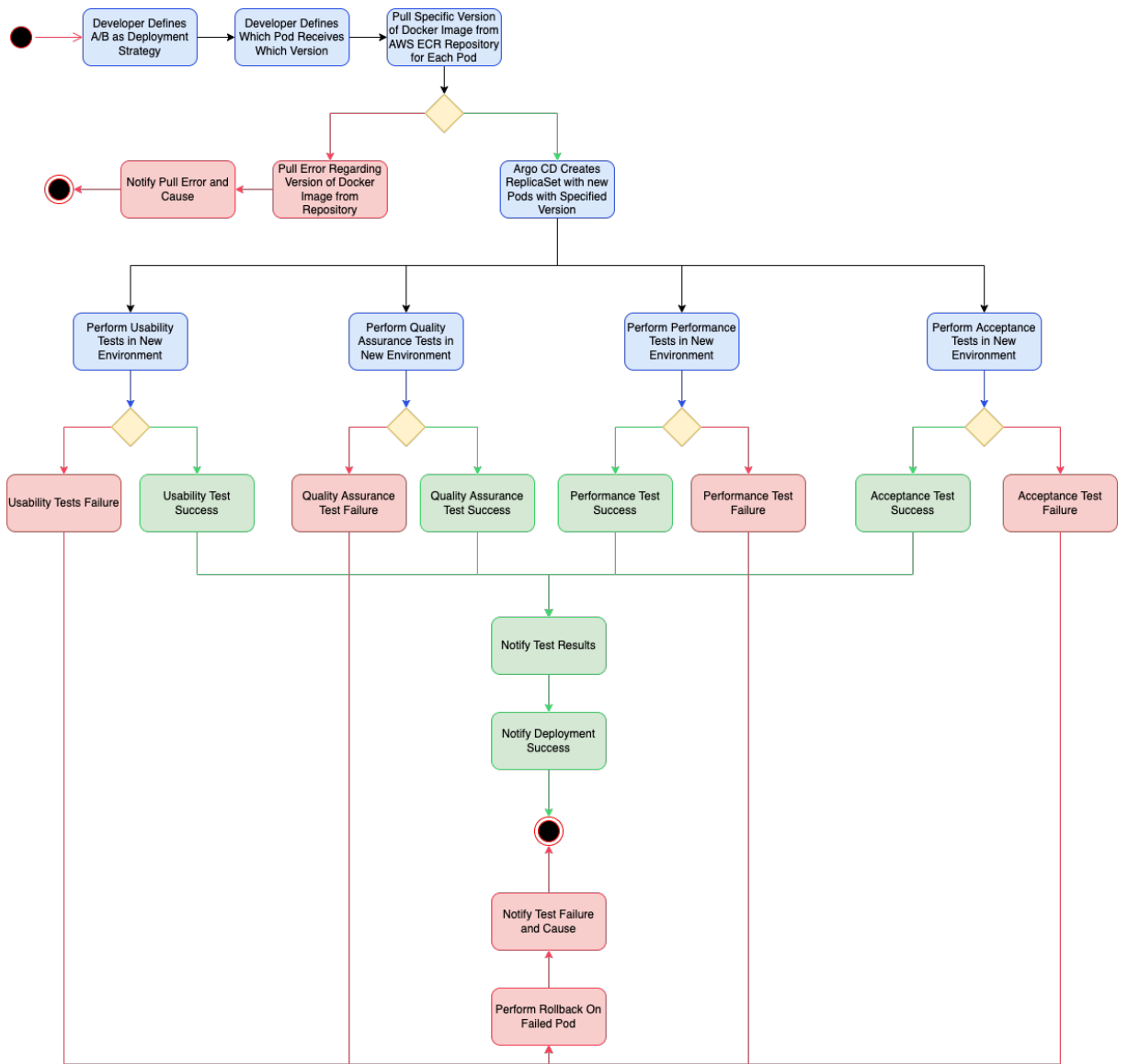


Figure 77. A/B Testing Activity Diagram

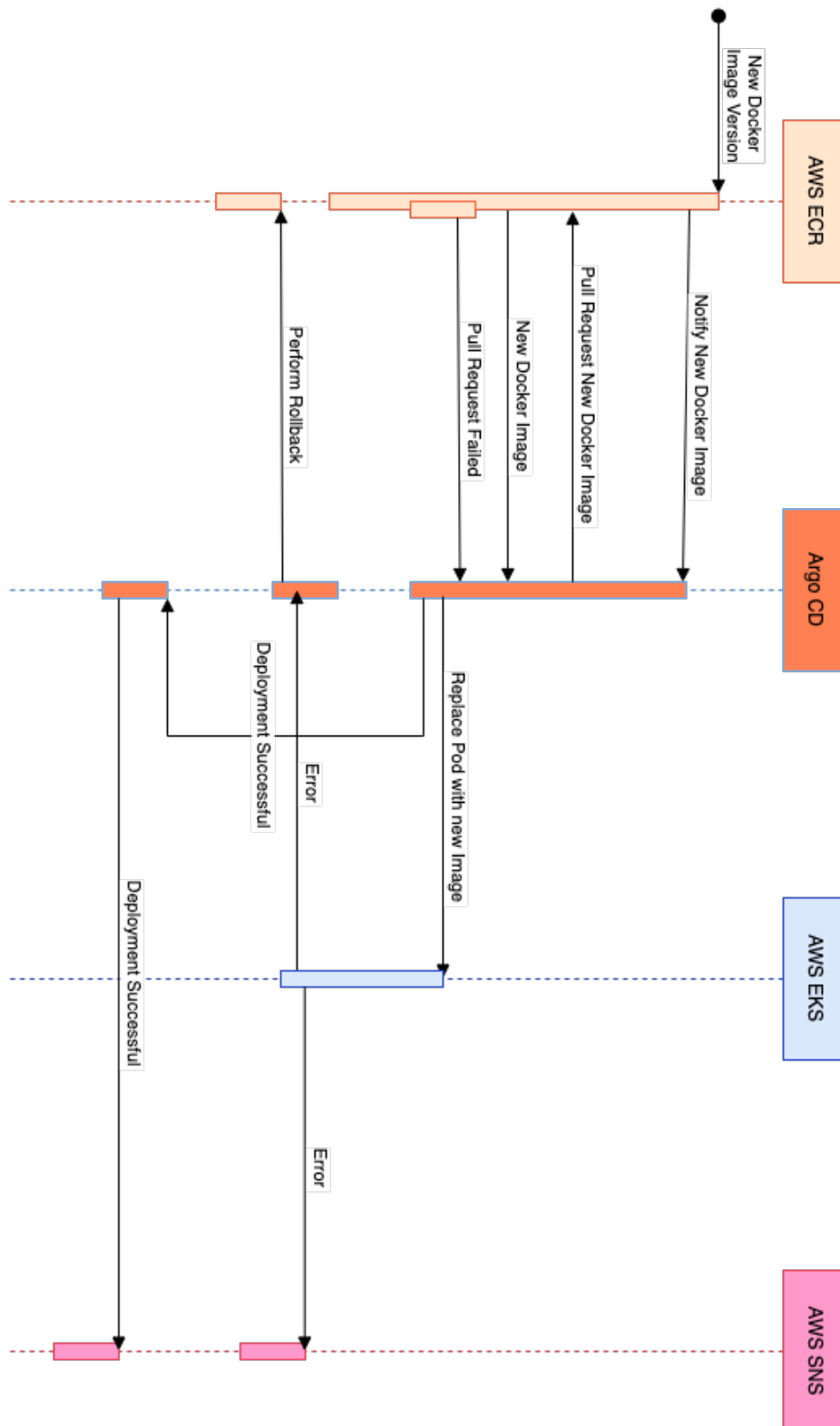


Figure 78. A/B Testing Sequence Diagram

Monitoring Microservices

This section explains the functionalities implemented to perform the task of monitoring the microservice application, along with the infrastructure in which it is deployed. Here,

the tools chosen to monitor various metrics that exist within an AWS EKS cluster must offer insights into the metrics of each node, the pods they run and the number of requests that are processed by the service. The metrics exposed by the various services must be presented with a visualization tool in efforts to translate and present the information in a way that is both easy to understand and that offers total observability of the application.

Figure 30 represents a general overview of how the monitoring solution is implemented, with both Prometheus and Grafana regarding metrics, and the Elasticstack regarding log management.

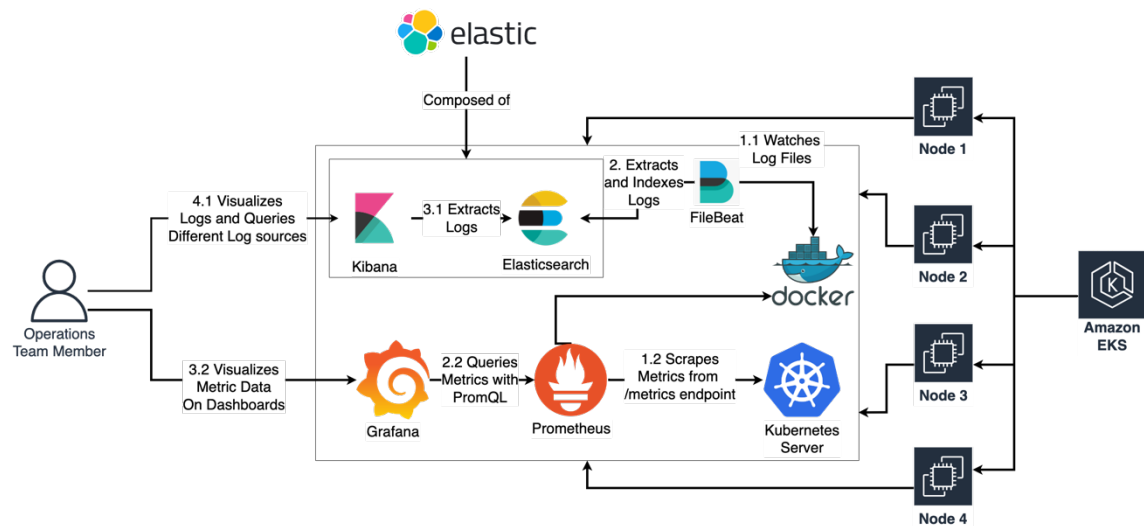


Figure 79. Macro-view Microservice Monitoring Solution

Metrics Analysis

One means of having observability into a microservices application is through the analysis of metrics. Metrics analysis involves observing resource consumption of the application, levels of CPU usage, memory usage and network bandwidth usage of the AWS EKS Cluster, its nodes and the container resources that the different service consume.

Figure 31 represents the workflow that is followed for Operations Teams to analyse and access the metrics of the application and the infrastructure. Each of the nodes within the AWS EKS Cluster runs an instance of Prometheus which scrapes both the Kubernetes server for cluster and node metrics, whilst also extracting metrics from the application running in the form of a docker image, all from the endpoint names “metrics”. From here, these metrics that are scraped are indexed into a time-series database and can be queried using PromQL: a lightweight, flexible query language which extracts the metrics according to the timestamp at which they are recorded. Grafana makes use of this query language to retrieve the metrics that are stored on within the Prometheus database, and presents them in a much easier way, adding the more information and forms of showing the information, making use of gauges, histograms and counters.

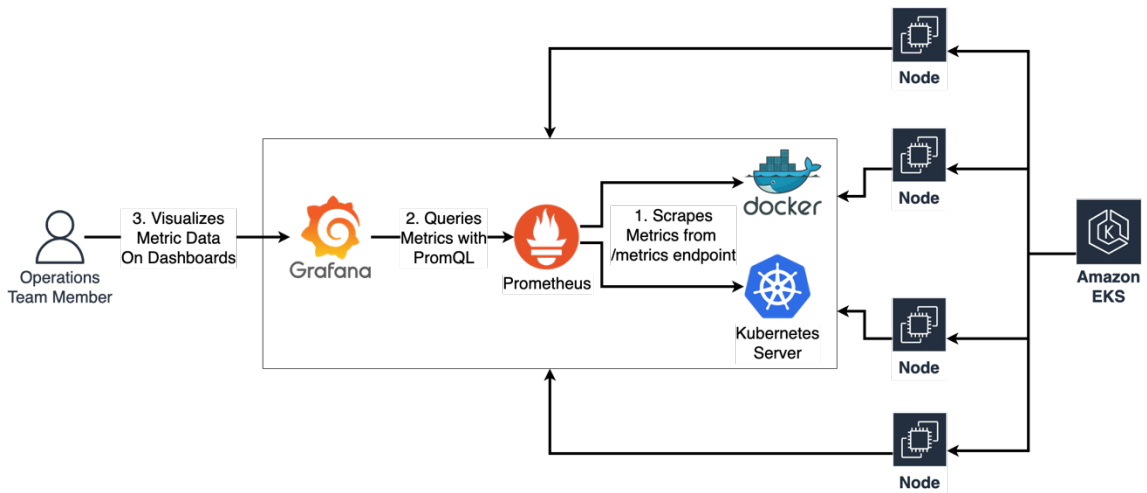


Figure 80. Metrics Monitoring Workflow

Figure 33 shows an example of the possibilities offered by Grafana when looking to analyse the metrics scraped by Prometheus, in comparison to Figure 32, where the amount of information that can be viewed is limited to one query, along with only one graph. It must also be said, Grafana is not a competitor to Prometheus, but rather a tool that compliments the metrics scraper.

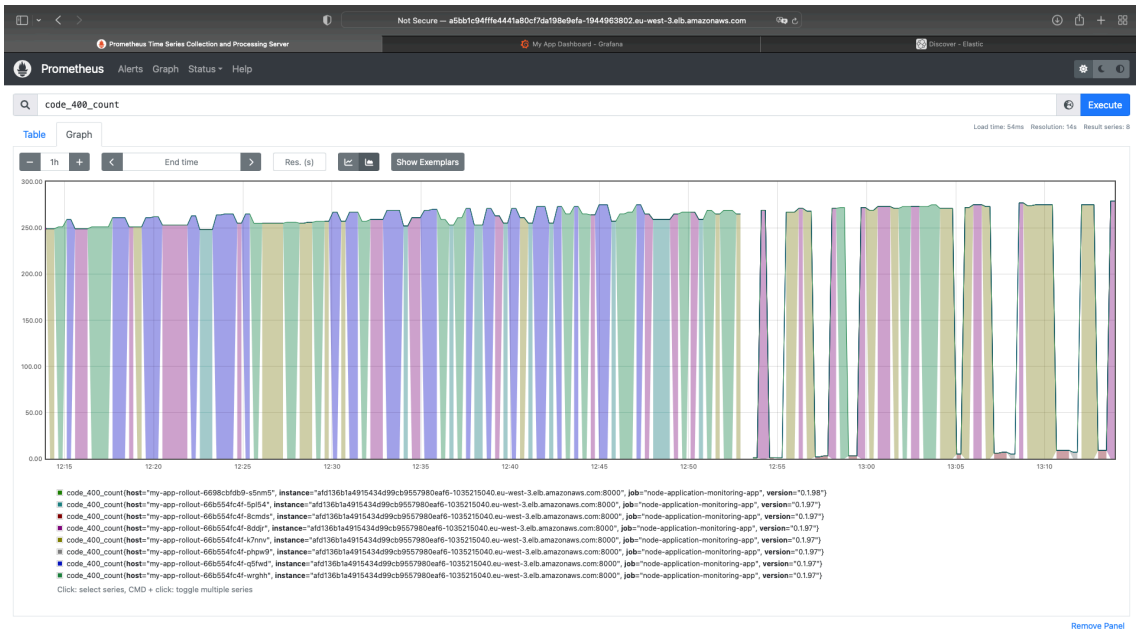


Figure 81. Prometheus Dashboard



Figure 82. Grafana Dashboards Example

Figure 34 presents a sequence diagram which shows the exchange of messages between Grafana, Prometheus and the AWS EKS Cluster to access Cluster metrics offered by Kubernetes by default.

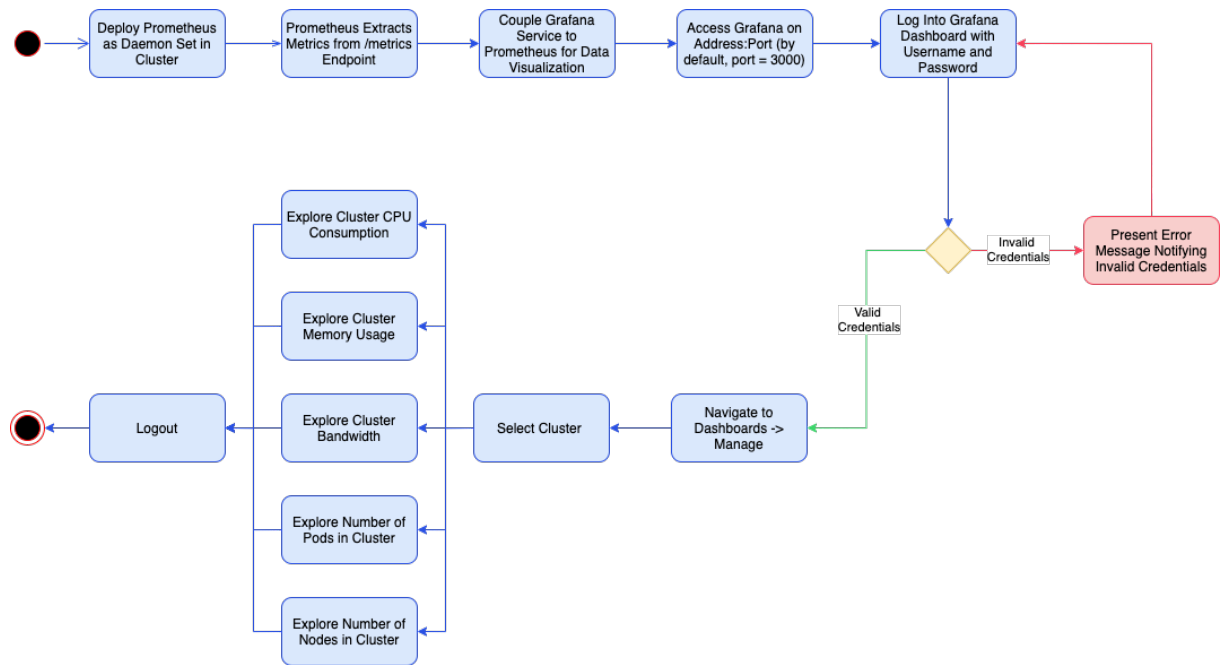


Figure 83. Cluster Metrics Analysis Activity Diagram

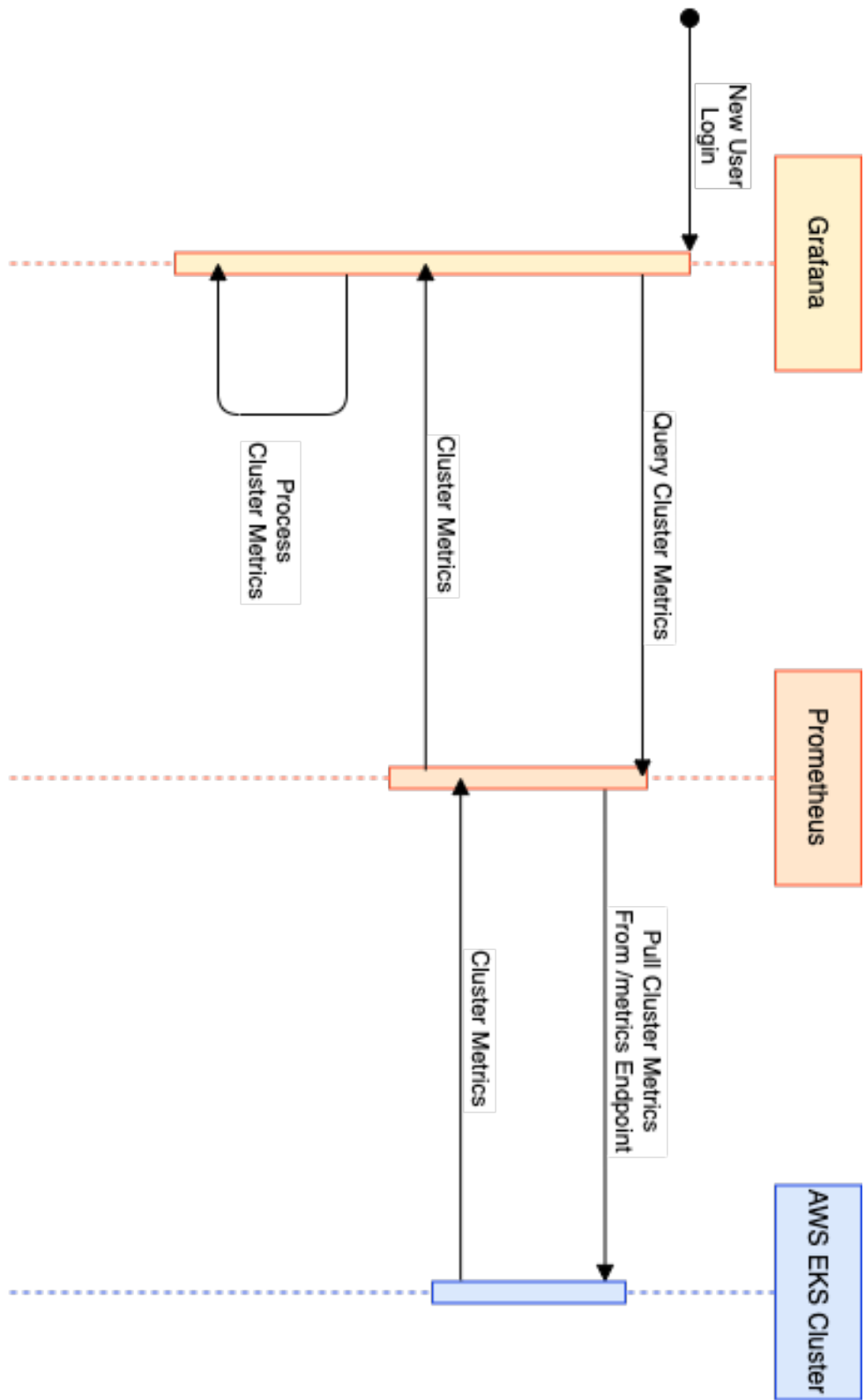


Figure 84. Cluster Metrics Analysis Sequence Diagram

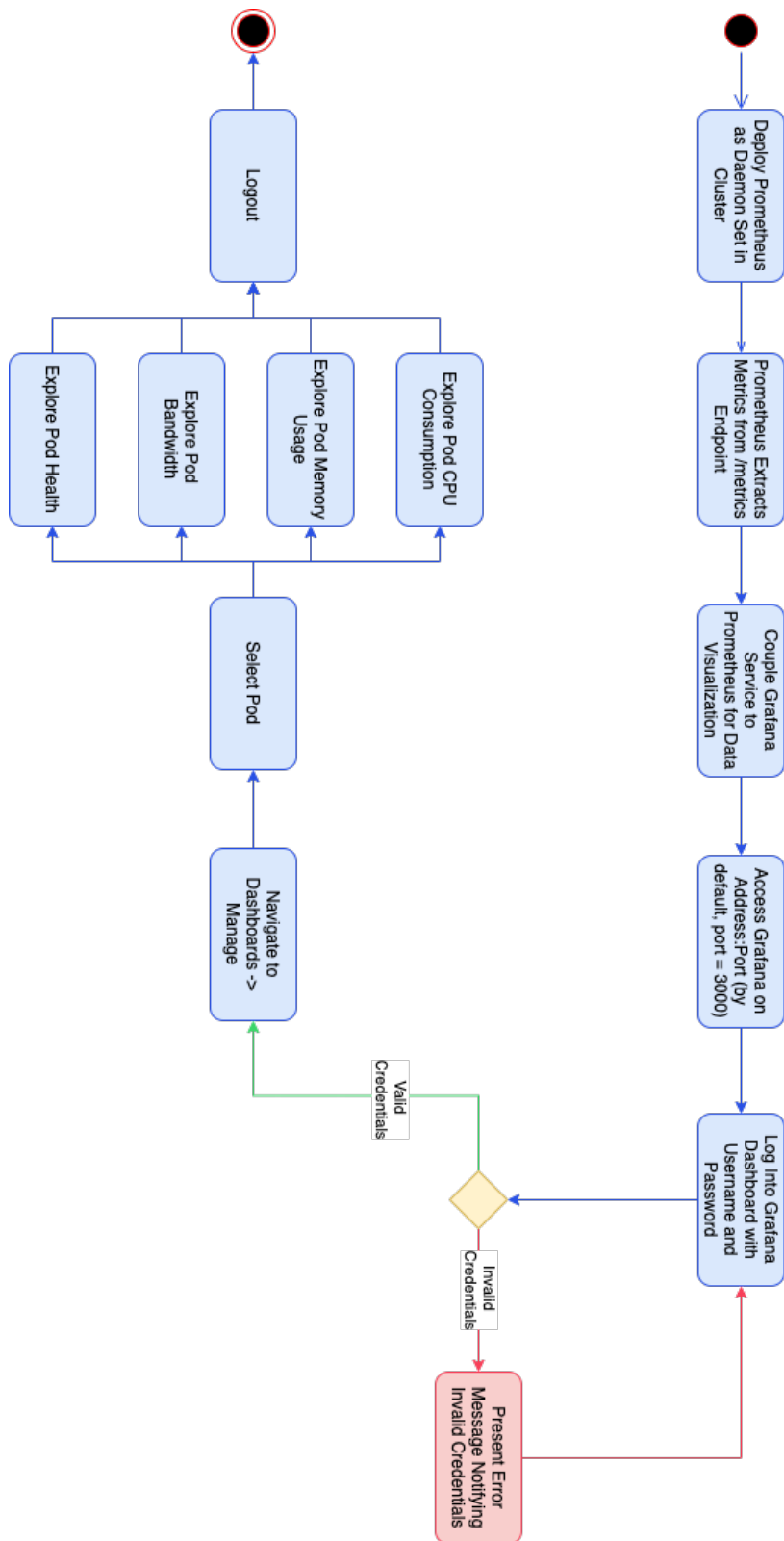


Figure 85. Pod Metrics Analysis Activity Diagram

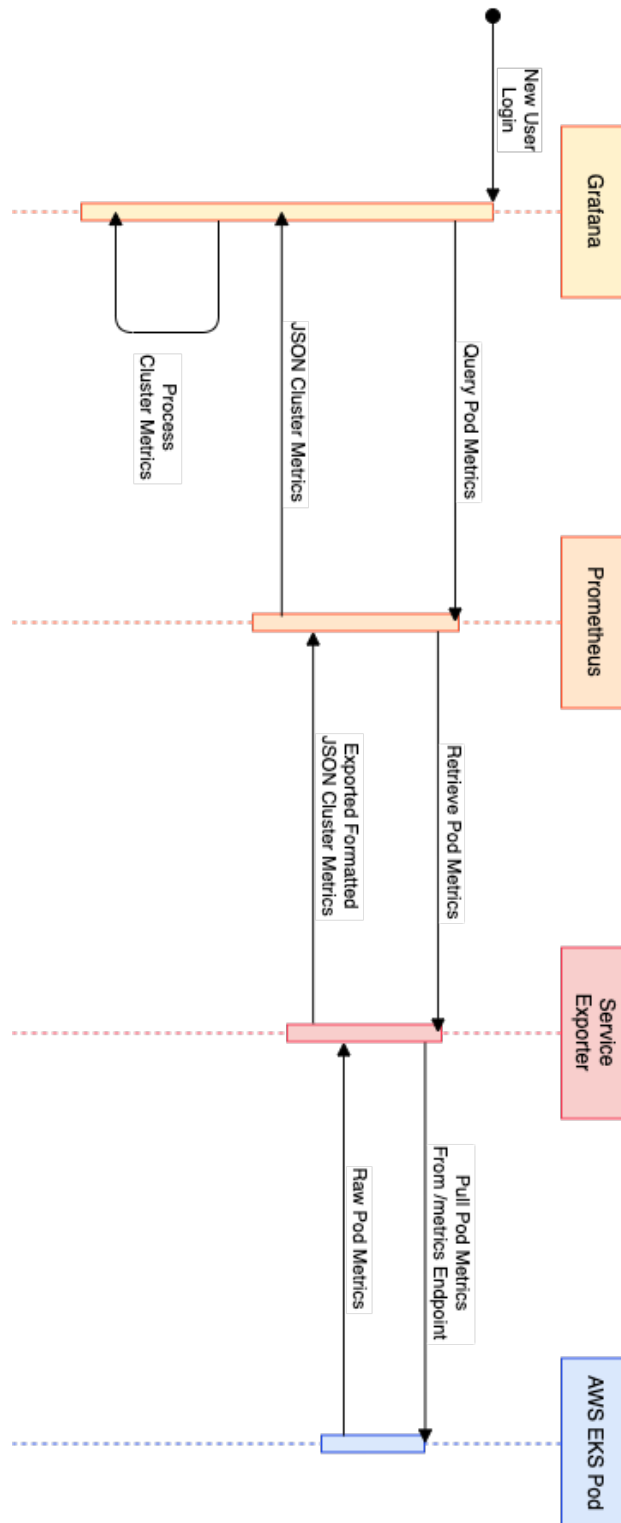


Figure 86. Pod Metrics Analysis Sequence Diagram

Application Log Analysis

Regarding service logs, the ELK stack will be the tool used to process and allow access to the logs produced within the cluster. The stack is composed of three elements, Elasticsearch, Logstash and Kibana, their respective tasks have been mentioned in chapter 3 – State of the Art.

The tool chosen for gaining observability into the applications functionalities is the Elastic Stack. It's comprised of Elasticsearch, Kibana, Beats, and Logstash.

Elasticsearch allows storage, searching, and analysis of logs with speed at scale.

Kibana allows data exploring with visualizations, from waffle charts and heatmaps to time series analysis and beyond. Kibana offers preconfigured dashboards for diverse data sources, live presentations to highlight KPIs, and management for deployment in a single User Interface; and Beats which are a free and open platform for single-purpose data shipping. A cluster can have multiple beats operating simultaneously, extracting data from hundreds or thousands of machines and systems to Elasticsearch. For this Framework, Filebeat will be used.

Filebeat offers a lightweight means of forwarding and centralizing logs and files by shipping with modules for observability and security data sources that simplify the collection, parsing, and visualization of common log formats down to a single command. They achieve this by combining automatic default paths based on your operating system, with Elasticsearch Ingest Node pipeline definitions, and with Kibana dashboards. Plus, a few Filebeat modules ship with preconfigured machine learning jobs.

Figure 35 represents, graphically, the workflow involved when looking to analyse the logs produced by the various services within the microservice application. Filebeat is responsible for accessing the logs produced by the STDOUT of each service. The STDOUT contents are routed to temporary files in the /var/log/containers directory, where each service's log file is identified by the hostname on which the service runs. The AWS EKS offers the management of the log files by default, therefore not requiring log file rotation nor an extra mechanism to manage the amount of memory occupied by the files. Filebeat captures the contents of these log files and forwards them to Elasticsearch for indexing [53]. Elasticsearch performs indexing, parsing and aggregates the logs received from Filebeat and makes them available for Kibana to perform querying and retrieval of their contents.

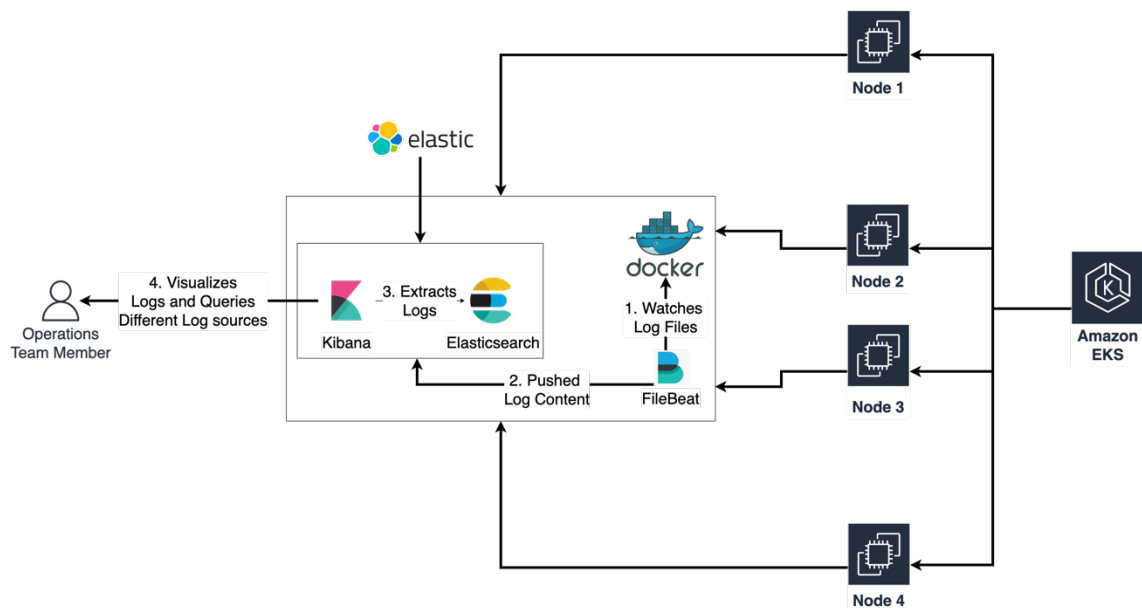


Figure 87. Log Analysis Workflow

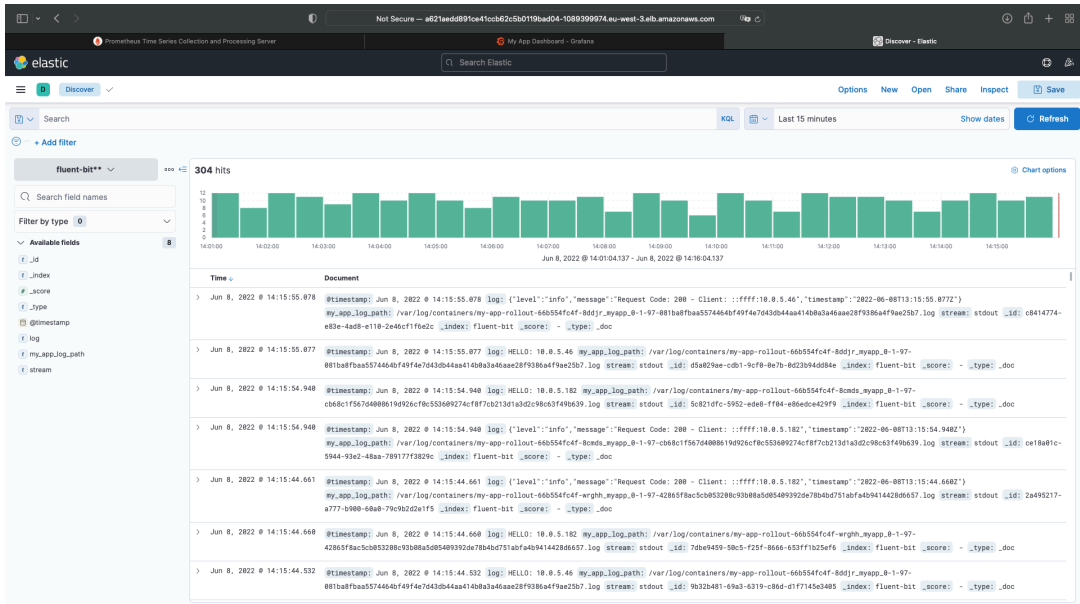


Figure 88. Example of Log Analysis in Kibana

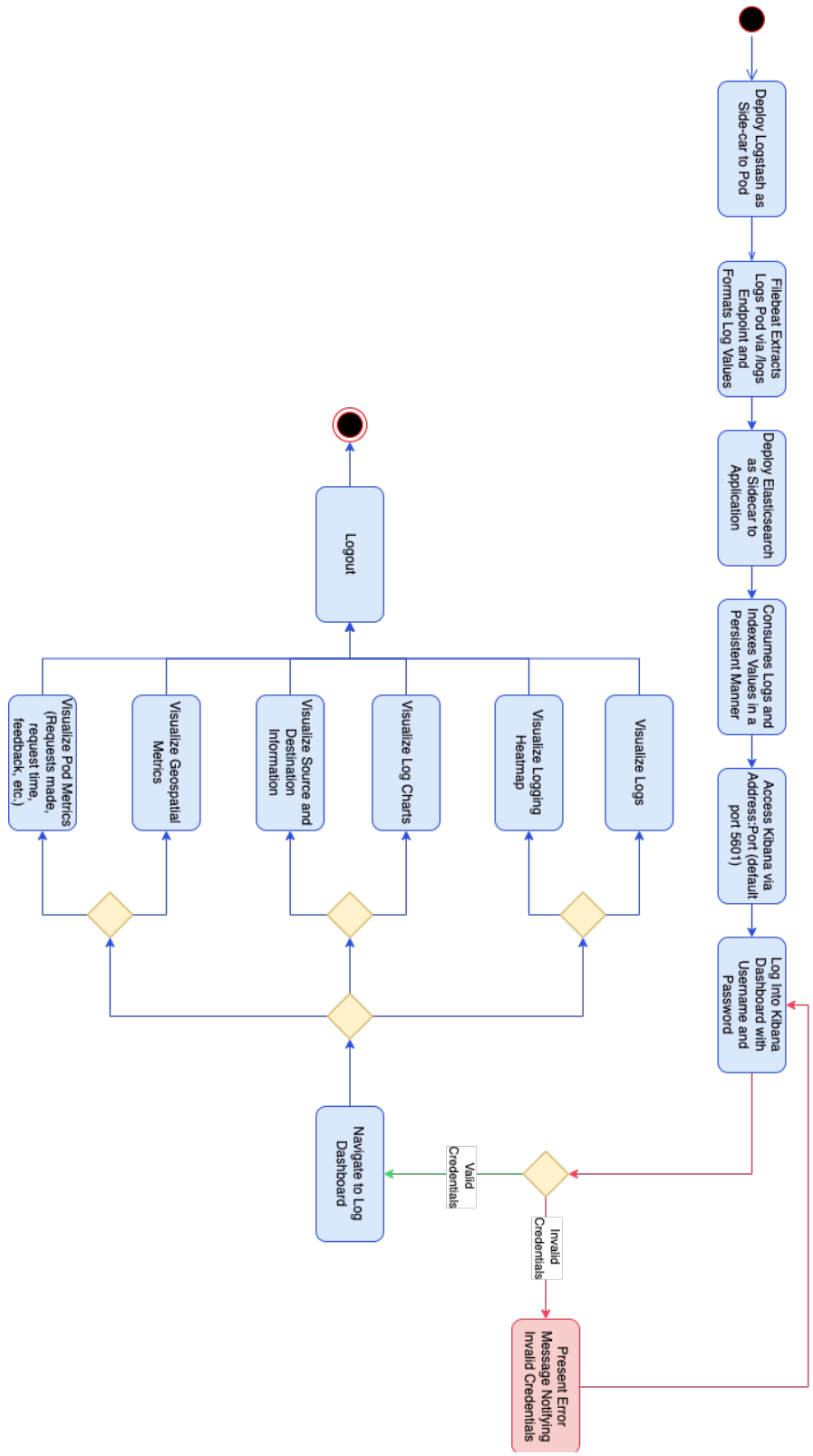


Figure 89. View Microservice Logs Activity Diagram

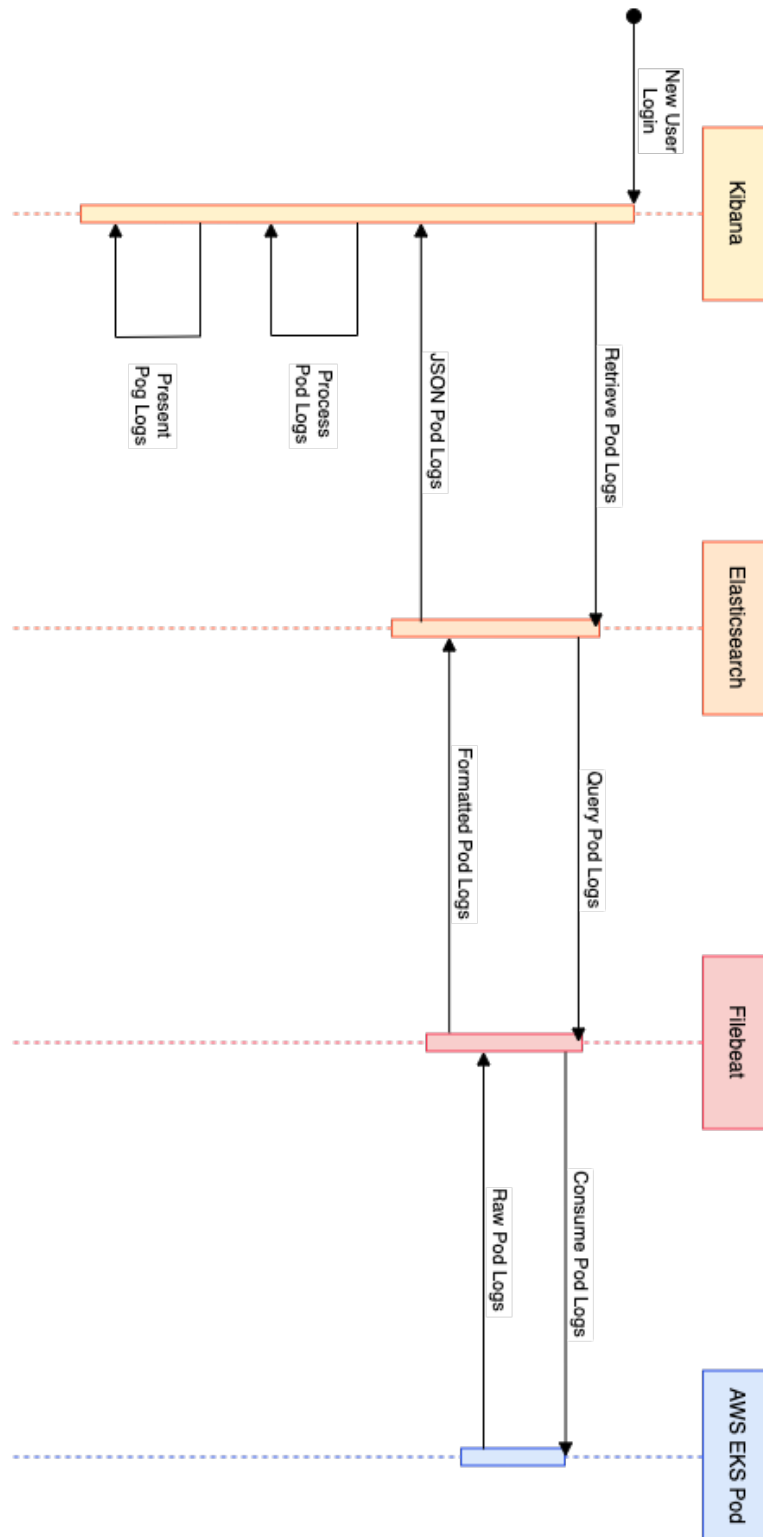


Figure 90. View Microservice Logs Sequence Diagram

The following Figure 37 presents a Sequence Diagram of the messages that are exchanged between the different tools that make it possible to perform log analysis on the microservices that compose the application.

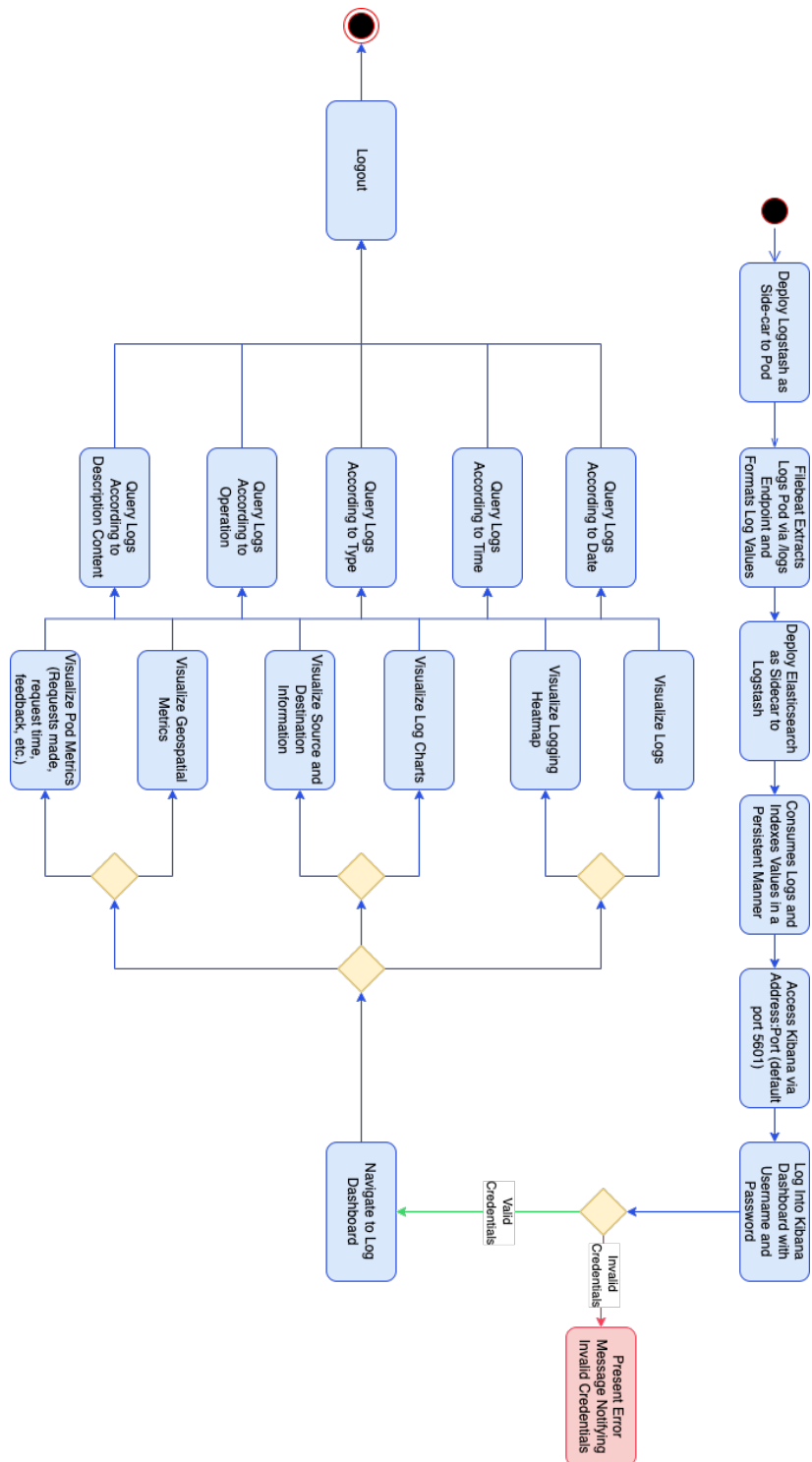


Figure 91. Query Microservice Logs Activity Diagram

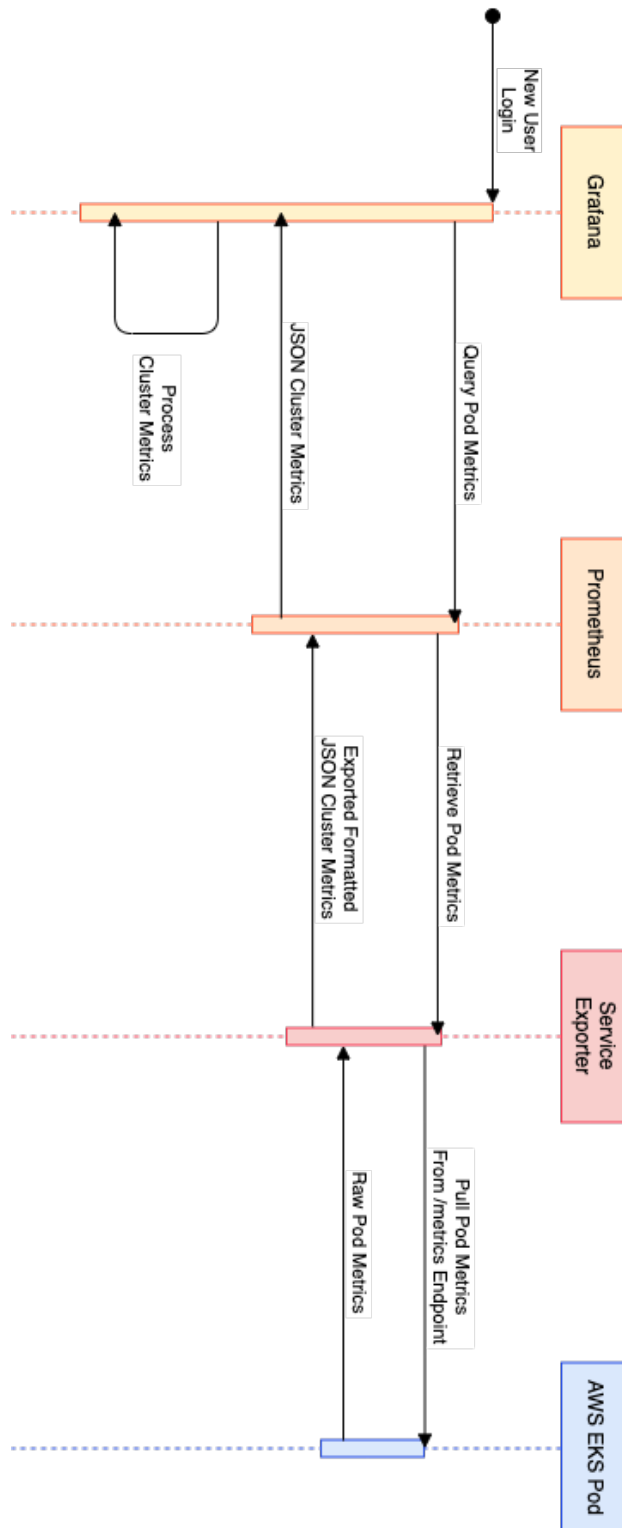


Figure 92. Query Microservice Logs Sequence Diagram

Challenges and Difficulties

When producing the various components of this framework, many difficulties arose, where I was met by many different challenges and opportunities to grow. The each of the

different epics offered their own problems and solutions throughout the development process.

When elaborating the Infrastructure Provisioning components, implementing the different elements within the infrastructure was the most interesting task to accomplish, as it required working with multiple services that are offered by AWS. Having to orchestrate the different tools and conjugate them into a fully working solution for a microservices application was difficult at first given my lack of experience with the microservice architectural pattern. This lack of information was mitigated by the extensive documentation and explanation of how the different services work and their best practices.

The CI/CD components were more straightforward during implementation. Difficulties arose when looking to understand which the more appropriate unit tests were to be carried out during the application build process. Developing with Argo CD brought some difficulties when understanding how to configure repository access and the application's project organization had to be conducted. Having defined the correct organization, associating the AWS CodeCommit repository with Argo CD was simple.

Finally, implementing the monitoring solution did not offer difficulties, given the ease with which the tools are configured.

Appendix E – Acceptance Test Plan and Report



FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE
COIMBRA
DEPARTMENT OF INFORMATICS ENGINEERING

Gabriel Marco Freire Pinheiro

Appendix E – Acceptance Test Plan and Report

Dissertation in the context of the Master’s Degree in Informatics Engineering,
Specialization in Software Engineering, advised by Professor Nuno Laranjeiro and
Engenheiro Rui Cunha, presented to the Department of Informatics Engineering of the
Faculty of Sciences and Technology of the
University of Coimbra.

July 2022

This page was intentionally left in blank.

Table of Contents

ACCEPTANCE TESTING.....	LXXXVI
VERIFICATION PROCESS	ERROR! BOOKMARK NOT DEFINED.
VALIDATION PROCESS	ERROR! BOOKMARK NOT DEFINED.
ACCEPTANCE TEST PLAN.....	ERROR! BOOKMARK NOT DEFINED.
TEST RESULTS	CXIX
INCIDENT REPORTS	CXXI

Acceptance Testing

This chapter contains an explanation of the process carried out to guarantee that the produced framework follows the objectives that were defined at the beginning of the internship. The functionalities tested are those defined in the document titled Appendix C - Software Requirements Specification. This chapter presents the formal validation process followed to guarantee the quality with which the framework was developed. For more information, please refer to the document titled Appendix C - Software Requirement Specification Document and Appendix E – Acceptance Test Plan and Report.

As in previous chapters, this chapter will be divided into four sections: Infrastructure Provisioning, Continuous Integration, Continuous Delivery and Application Monitoring. In each of these sections there will be a reduced list of the more essential Base States used to perform tests on the framework, along with the respective Test Cases used to evaluate if the framework performs as expected. All the tests conducted against the framework will be carried out on the computer which WIT Software made available for this project. The tests will be carried out by the author of this document as well.

Verification Process

Throughout the project, there was a plan of verification in place to guarantee the quality of the developed components during their creation. Whenever a user-story was developed, a progressive evaluation strategy was adopted where each component was required to be compatible with the previously developed components.

At the end of every week, a demonstration was made to the Product Owner and Scrum Master to guarantee that the requirements were being fulfilled correctly, to maintain a coherent and consistent development process, and to introduce a means of verifying that the product was being developed to the organization's expectations. These demonstrations were required to have a list of tasks that were completed during the week, the goals that were achieved, and eventual problems that may have occurred during the development process that week. The demonstration would also have to show the work plan that was in place for the following week, defining the objectives that would have to be met by the next week. The larger demonstrations were shown by the end of each sprint, where the entire state of the framework was presented to the coordinators, along with the expectations for the next sprint.

The verification process was similar in the various components, with a few differences between each other, given their area of study. Regarding Infrastructure Provisioning, each developed component was required to operate with the others within the application's architecture. The infrastructure was required to be deleted and rebuilt on a daily basis, guaranteeing that the components were not dependant on static configurations, which also reinforces the importance of automation and robustness of the infrastructure setup; in Continuous Integration, the pipeline would have to be as generic as possible to be used by different applications. The process would have to be immutable, regardless of the application that was included in the AWS CodeCommit repository; regarding Continuous Delivery, the product of the Continuous Integration phase was the single point of truth when looking to deploy changes into the application's production environment. This product, the Docker Container Image, would then be used in the different deployment strategies that could be configured according to the project's necessities; and finally the

monitoring component was required to make use of the application within its production environment. Once deployed, a need arises to obtain information and observability into the application. To achieve this, the tasks that the application carries out are controlled by accessing the logs that are streamed from the application. The resources that are used by the application allow for improvements from an efficiency point of view.

The verification process occurred naturally, given that each component was dependant on the other. Once presented to the coordinators, and tested for the demonstrations, the following tasks could be executed.

Task #1 – [14/02/2022 – 25/02/2022]: The first task was the creation of the project management documents for the second semester. This task started on the 14th of February and lasted until the 25th of February. The documents were verified by meeting with the Product Owner and the Scrum Master to guarantee that their contents were correct according to the objective of the internship.

Task #2 – [28/02/2022 – 01/04/2022]:

Once the planning process was complete, the development process could begin, starting with the creation of the Infrastructure that would house the microservices application. This task required the creation of the aforementioned elements associated with infrastructure provisioning. This task required a compensation week, given that the development of the components within the environment caused the project to fall behind schedule. This delay would be compensated in the next task. For a more detailed analysis, please refer to the document titled Appendix D – Software Architecture and Design, under the section called Infrastructure Provisioning.

This task was verified by first creating multiple branches within the AWS CodeCommit repository and verifying the creation of the different AWS CodePipeline delivery elements associated with each branch. Each branch was duly updated with new infrastructure templates and performed an update on the existing infrastructure. Finally, each branch was merged into the main branch to visualize the deletion of the infrastructure associated with the original branch and the update of the main branch with the new configuration. These activities were further verified through a demonstration to the Product Owner and Scrum Master where the different elements were shown as created within the AWS console.

Task #3 – [04/04/2022 – 06/05/2022]:

- **Task #3.1 – [04/15/2022 – 15/04/2022]:** Once the infrastructure was in place, the CI/CD pipeline began taking shape, with the repository and pipeline creation taking place over two weeks. Here, the Continuous Integration phase was developed, creating the elements mentioned in the “Pre-Requisites” section of the document titled Appendix D – Software Architecture and Design, along with the automation process responsible for preparing the application’s source code for delivery.

This sub task was verified by performing changes to the AWS CodeCommit repository to validate that the different components associated with the AWS CodePipeline worked correctly. As in the previous task, a new commit was made, new branches were created, merged and deleted to verify that the AWS Lambda function was performing the changes to the AWS CodePipeline projects accordingly. This functionality was presented to the Product Owner and the Scrum Master for confirmation and to guarantee that it functioned correctly.

- **Task #3.2 – [18/04/2022 – 29/04/2022]:** Having the initial pipeline in place, the tasks of automated building, static code analysis, dependency verification and push to the ECR Repository had to be defined. Here there was a small decision to be made regarding which tools would be used to perform these tasks. Once these tools were chosen, the pipeline was tested from end to end. A commit made to the AWS CodeCommit repository would trigger the AWS CodePipeline project associated with the branch that was changed. This project begins the AWS CodeBuild project and produces the final Docker Image. The changes made to the application were then verified in the final application.

Task #4 – [09/05/2022 – 27/05/2022]: Once the pipeline was in place to create the final executable Docker Image, Argo CD was deployed into the AWS EKS cluster, under its own namespace. This Continuous Delivery tool performs the progressive releases of the new image into the EKS Cluster, whilst analysing the metrics of the image to perform a rollback when necessary. This process was verified by performing multiple requests to the changed service and visualizing the two different versions returning different results, at the rate that the application load balancer would transfer the traffic between each version.

Task #5 – [30/05/2022 – 17/06/2022]: The final tools that were configured and deployed into the EKS Cluster were Prometheus and Grafana, responsible for metrics analysis and observability from a resource consumption point of view; and Elasticstack, which extracts the log stream from the various containers into a centralised location for querying and processing. The metrics were visualized using Grafana, where a dashboard was composed of multiple panels that made queries to the Prometheus instance running in the AWS EKS cluster.

The logs produced by the application were streamed directly to Kibana. To verify that the logs were being processed correctly, the application was coded in a way that an error message be printed every 15 seconds by the application. This error was printed in the Kibana GUI stream exactly as programmed.

Task #6 – [20/06/2022 – 01/07/2022]: finally, the last task carried out in the internship was the validation of the full framework from one end to another. This task is explained in further detail in the next section.

Validation Process

The validation process behind the internship consisted of evaluation tests which would follow the contents of the document Appendix E – Acceptance Test Plan and Report. This document defines the due dates for the components to be submitted to functional tests, guaranteeing that they perform correctly and according to their requirements. At the end of each User Epic, the framework was tested, using base states as a foundation on which to perform numerous test cases.

The components of the framework have a reserve of base states and test cases which must be complied with to guarantee that all requirements and functionalities that were implemented are fully operational, and function according to the desired of WIT Software. Base states are a specific snapshot of the framework after a number of tasks have been performed. These states will act as a foundation on which the different test cases will be applied, according to the requirements that are to be evaluated. Test Cases

are a series of events that are communicated to the framework as inputs in order to understand if the intended outputs are obtained.

Base States

This section shows the different base states used during the validation process. Each base state is identified by its name, requirements, preconditions that must be met to reach the state in question, and the results which confirm that the current state of the framework is the target base state. Table 16 shows a list of exemplars that represent how the Base States are constructed. Each exemplar is taken from each of the four elements of the framework. That is Infrastructure Provisioning, Continuous Integration, Continuous Delivery and Monitoring.

operations teams insight into the application’s metrics and logging behavior when in production.

Name	BS-01: Initial State – Infrastructure Provisioning
Requirements	A command line interface; Internet Access.
Preconditions	AWS Command Line Interface (CLI) Client is installed.
Workflow	<p>According to the DevOps Team Leader’s System:</p> <ul style="list-style-type: none"> - Windows: Run the following command – msiexec.exe /i https://awscli.amazonaws.com/AWSCLIV2.msi - Mac: Run the following command – curl "https://awscli.amazonaws.com/AWSCLIV2.pkg" -o "AWSCLIV2.pkg" sudo installer -pkg AWSCLIV2.pkg -target / - Linux: Run the following command – curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip" unzip awscliv2.zip sudo ./aws/install
Results	The AWS CLI tool is successfully installed.

Name	BS-02: DevOps Team Leader Logged In State
-------------	--

Requirements	<p>A DevOps Team Leader must be given the credentials which give access to the respective AWS User;</p> <p>A command line interface;</p> <p>Internet Access.</p>
Preconditions	BS-01: Initial State – Infrastructure Provisioning.
Workflow	<ol style="list-style-type: none"> 1. The DevOps Team Leader adds their credentials to the systems environment variables with the following: <ul style="list-style-type: none"> - export AWS_ACCESS_KEY_ID=ASIAIOSFODNN7EXMPL; - export AWS_SECRET_ACCESS_KEY= /K7MDENG/EXMPLKEY; - export AWS_SESSION_TOKEN=AQoDYXdzEJr...<remainder of session token>- 2. The DevOps Team Leader confirms his credentials with the following command: <ul style="list-style-type: none"> - aws sts get-caller-identity. <p>Alternatively:</p> <ol style="list-style-type: none"> 1. The DevOps Team Leader configures their AWS CLI credentials with the command ‘aws configure’; 2. The DevOps Team Leader inserts their Access Key ID; 3. The DevOps Team Leader inserts their Secret Access Key; 4. The DevOps Team Leader opens the file ~/.aws/credentials file; 5. The DevOps Team Leader adds a line with: <pre>aws_session_token = AQoDYXdzEJr...<remainder of session token>;</pre> 6. The DevOps Team Leader confirms his credentials with the following command: <ul style="list-style-type: none"> - aws sts get-caller-identity.
Results	The DevOps Team Leader Credentials will be printed in the terminal.

Name	BS-03: DevOps Maintainer Logged In State
Requirements	A DevOps Maintainer must be given the credentials which give access to the respective AWS User;

	A command line interface; Internet Access.
Preconditions	BS-01: Initial State – Infrastructure Provisioning
Workflow	<ol style="list-style-type: none"> 1. A DevOps Maintainer adds their credentials to the systems environment variables with the following: <ul style="list-style-type: none"> - export AWS_ACCESS_KEY_ID=ASIAIOSFODNN7EXMPL - export AWS_SECRET_ACCESS_KEY= /K7MDENG/EXMPLKEY - export AWS_SESSION_TOKEN=AQoDYXdzEJr...<remainder of session token> 2. A DevOps Maintainer confirms his credentials with the following command: <ul style="list-style-type: none"> - aws sts get-caller-identity <p>Alternatively:</p> <ol style="list-style-type: none"> 1. A DevOps Maintainer configures their AWS CLI credentials with the command ‘aws configure’; 2. A DevOps Maintainer inserts their Access Key ID; 3. A DevOps Maintainer inserts their Secret Access Key; 4. A DevOps Maintainer opens the file ~/.aws/credentials file; 5. A DevOps Maintainer adds a line with: <pre>aws_session_token = AQoDYXdzEJr...<remainder of session token>;</pre> 6. A DevOps Maintainer confirms his credentials with the following command: <ul style="list-style-type: none"> - aws sts get-caller-identity
Results	A DevOps Maintainer Credentials will be printed in the terminal.

Name	BS-04: Pre-Requisites Created State – Infrastructure Provisioning
Requirements	A command line interface; Internet Access.

Preconditions	BS-2: DevOps Team Leader Logged In State
Workflow	<ol style="list-style-type: none"> 1. Navigate to folder where PreRequisites.yaml is located; 2. Run command: <pre>aws cloudformation create-stack \ --stack-name TestName \ --template-body file://PreRequisites.yaml \ --capabilities CAPABILITY_NAMED_IAM</pre> 3. Navigate to AWS CloudFormation Console; 4. Verify that the pre-requisites Stack is being created with the name 'TestName'; <p>Success tag appears when the creation is complete.</p>
Results	<p>A stack ID will be printed in the command line interface when the command is run;</p> <p>The infrastructure is created and ready to receive an application.</p>

Name	BS-05: Ready For Development State – Infrastructure Provisioning
Requirements	<p>A command line interface;</p> <p>Access to AWS Console;</p> <p>Internet Access.</p>
Preconditions	BS-02: DevOps Team Leader Logged In State
Workflow	<ol style="list-style-type: none"> 1. Navigate to AWS S3 management page; 2. Select upload file; 3. Select PipelineTemplate.yaml file; 4. Confirm upload is successful; 5. Perform commit of local infrastructure files to CodeCommit repository; 6. Navigate to AWS CloudFormation management page; 7. Confirm infrastructure stack is in “Create In Progress” state; 8. Confirm infrastructure stack reaches “Creation Complete” state.

Results	AWS CodePipeline infrastructure components are placed in the S3 bucket successfully for the Lambda Function to access and create the infrastructure accordingly.
---------	--

Name	BS-06: Infrastructure Templates Created State
Requirements	A command line interface; Access to AWS Console; Internet Access.
Preconditions	BS-02: DevOps Team Leader Logged In State Or BS-03: DevOps Maintainer Logged In State; BS-04: Pre-Requisites Created State – Infrastructure Provisioning; BS-05: Ready For Development State – Infrastructure Provisioning; Git CLI client is installed and configured for the AWS CodeCommit Repository.
Workflow	1. Commit changed infrastructure templates; 2. Verify Lambda Logs to confirm alteration detection; 3. Verify in AWS CodePipeline that pipeline is in “In Progress” state.
Results	Confirm infrastructure creation is successful with stack in AWS CloudFormation service console being in “Create Complete” state. The infrastructure is updated and displays the new configuration.

Name	BS-07: Infrastructure Created State
Requirements	Internet Access; Access to AWS Console.
Preconditions	BS-06: Infrastructure Templates Created State.
Workflow	1. Login to AWS Console; 2. Navigate to AWS CloudFormation service;

	3. Verify that Stack is in “CREATE_IN_PROGRESS” state.
Results	Confirm infrastructure creation is successful with stack in AWS CloudFormation service console being in “CREATE_COMPLETE” state. The infrastructure is updated and displays the new configuration.

Name	BS-08: SSH Connection Established to Bastion Host State
Requirements	A command line interface; A DevOps Maintainer must have access to the SSH key-pair which allows access to the Bastion Host EC2 machine; SSH CLI installed; Internet Access.
Preconditions	A DevOps Team Leader/Team Member must have authenticated their AWS CLI (BS-2/BS-3).
Workflow	A DevOps Maintainer runs the following command: ssh -i KeyPair.pem ec2-user@bastion_host_DNS.amazonaws.com.
Results	The terminal user and hostname are altered to match the EC2 user and the Bastion Host’s hostname.

Name	BS-09: Initial State – Continuous Integration
Requirements	A command line interface; Internet Access.
Preconditions	AWS Command Line Interface (CLI) Client is installed.
Workflow	According to the Development Team Leader’s System: <ul style="list-style-type: none"> - Windows: Run the following command – msiexec.exe /i https://awscli.amazonaws.com/AWSCLIV2.msi; - Mac: Run the following command –

	<pre>curl "https://awscli.amazonaws.com/AWSCLIV2.pkg" -o "AWSCliv2.pkg" sudo installer -pkg AWSCliv2.pkg -target; - Linux: Run the following command – curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip" unzip awscliv2.zip sudo ./aws/install.</pre>
Results	The AWS CLI tool is successfully installed.

Name	BS-10: Development Team Leader Logged In State
Requirements	<p>A Development Team Leader must be given the credentials which give access to their respective AWS User;</p> <p>A command line interface;</p> <p>Internet Access.</p>
Preconditions	BS-09: Initial State – Continuous Integration.
Workflow	<ol style="list-style-type: none"> 1. A Development Team Leader adds their credentials to the systems environment variables with the following: <ul style="list-style-type: none"> - export AWS_ACCESS_KEY_ID=ASIAIOSFODNN7EXMPL; - export AWS_SECRET_ACCESS_KEY= /K7MDENG/EXMPLKEY; - export AWS_SESSION_TOKEN=AQoDYXdzEJr...<remainder of session token>; 2. A Development Team Leader confirms his credentials with the following command: <ul style="list-style-type: none"> - aws sts get-caller-identity. <p>Alternatively:</p> <ol style="list-style-type: none"> 1. A Development Team Leader configures their AWS CLI credentials with the command ‘aws configure’; 2. A Development Team Leader inserts their Access Key ID; 3. A Development Team Leader inserts their Secret Access Key; 4. A Development Team Leader opens the file ~/.aws/credentials file;

	<p>5. A Development Team Leader adds a line with:</p> <pre>aws_session_token = AQoDYXdzEJr...<remainder of session token>;</pre> <p>6. A Development Team Leader A Development Team Leader confirms his credentials with the following command:</p> <ul style="list-style-type: none"> - aws sts get-caller-identity.
Results	The credentials belonging to the Development Team Leader in question will be printed in the terminal.

Name	BS-11: Development Team Member Logged In State
Requirements	<p>A Development Team Member must be given the credentials which give access to the respective AWS User;</p> <p>A command line interface;</p> <p>Internet Access.</p>
Preconditions	BS-09: Initial State – Continuous Integration.
Workflow	<p>1. A Development Team Member adds their credentials to the systems environment variables with the following:</p> <ul style="list-style-type: none"> - export AWS_ACCESS_KEY_ID=ASIAIOSFODNN7EXMPL; - export AWS_SECRET_ACCESS_KEY= /K7MDENG/EXMPLKEY; - export AWS_SESSION_TOKEN=AQoDYXdzEJr...<remainder of session token>; <p>2. A Development Team Member confirms his credentials with the following command:</p> <ul style="list-style-type: none"> - aws sts get-caller-identity. <p>Alternatively:</p> <ol style="list-style-type: none"> 1. A Development Team Member configures their AWS CLI credentials with the command ‘aws configure’; 2. A Development Team Member inserts their Access Key ID; 3. A Development Team Member inserts their Secret Access Key; 4. A Development Team Member opens the file ~/.aws/credentials file;

	<p>5. A Development Team Member adds a line with:</p> <pre>aws_session_token = AQoDYXdzEJr...<remainder of session token>;</pre> <p>6. A Development Team Member confirms his credentials with the following command:</p> <ul style="list-style-type: none"> - aws sts get-caller-identity.
Results	The credentials belonging to the Development Team Member in question will be printed in the terminal.

Name	BS-12: Pre-Requisites Created State – Continuous Integration
Requirements	A command line interface; Internet Access.
Preconditions	BS-10: Development Team Leader Logged In State.
Workflow	<ol style="list-style-type: none"> 1. A Development Team Leader must navigate to folder where PreRequisites.yaml is located; 2. Run command: <pre>aws cloudformation create-stack \ --stack-name CIPreRequisites \ --template-body file://PreRequisites.yaml \ --capabilities CAPABILITY_NAMED_IAM</pre> 3. A Development Team Leader must navigate to AWS CloudFormation Console; 4. A Development Team Leader must verify that the pre-requisites Stack is being created with the name 'TestName'; <p>Success tag appears when the creation is complete.</p>
Results	<p>A stack ID will be printed in the command line interface when the command is run;</p> <p>The infrastructure is created and ready to receive an application.</p>

Name	BS-13: Ready For Development State – Continuous Integration
-------------	--

Requirements	A command line interface; Access to AWS Console; Internet Access.
Preconditions	BS-12: Pre-Requisites Created State – Continuous Integration.
Workflow	<ol style="list-style-type: none"> 1. A Development Team Leader must navigate to AWS S3 management page; 2. A Development Team Leader must select upload file; 3. A Development Team Leader must select PipelineTemplate.yaml file; 4. A Development Team Leader must confirm upload is successful; 5. A Development Team Leader must perform commit of local infrastructure files to CodeCommit repository; 6. A Development Team Leader must navigate to AWS CloudFormation management page; 7. A Development Team Leader must confirm infrastructure stack is in “Create In Progress” state; <ul style="list-style-type: none"> - Confirm infrastructure stack reaches “Creation Complete” state.
Results	AWS CodePipeline infrastructure components are placed in the S3 bucket successfully for the Lambda Function to access and create the infrastructure accordingly.

Name	BS-14: Application Source Code Created State
Requirements	A command line interface; Access to AWS Console; Internet Access.
Preconditions	BS-11: Development Team Member Logged In State; Git CLI client is installed and configured for the AWS CodeCommit Repository.
Workflow	A Development Team Member develops the application source code.

Results	Application Source Code is ready to be committed to the AWS CodeCommit repository.
---------	--

Name	BS-15: Code Committed State
Requirements	A command line interface; Access to AWS Console; Internet Access.
Preconditions	BS-14: Application Source Code Created State; Git CLI client is installed and configured for the AWS CodeCommit Repository.
Workflow	<ol style="list-style-type: none"> 1. A Development Team Member commits application source code to AWS CodeCommit Repository; 2. A Development Team Member verifies Lambda Logs to confirm alteration detection; 3. A Development Team Member verifies in AWS CodePipeline service that pipeline is in “In Progress” state.
Results	<ol style="list-style-type: none"> 1. AWS CodePipeline status is “Successful” when build completes; 2. AWS CodeBuild project is in “Successful” state when complete.

Name	BS-16: Docker Container Image Pushed to AWS ECR State
Requirements	Internet Connection; Internet Browser.
Preconditions	BS-15: Code Committed State
Workflow	Once the CI phase is complete, the new docker image is pushed to the AWS ECR repository autonomously by the AWS CodeBuild project.
Results	The new docker image is present in the AWS ECR repository and tagged accordingly.

Name	BS-17: Application Deployed with Argo CD State
Requirements	Internet Connection; Internet Browser.
Preconditions	BS-15: Argo CD and Argo Rollouts Deployed State
Workflow	<ol style="list-style-type: none"> 1. A Development Team Leader must login to the Argo CD server; 2. The credentials of this server are configurable, but by default are: Username: admin Password: the name of the server's hostname encoded in a base-64 format; This password can be retrieved by running the following command: <code>export ARGOCD_SERVER=`kubectl get secret argo-initial-admin-secret -n argocd -o jsonpath="{.data.password}" base64 -d`</code> 3. The Development Team Leader must navigate to the "Manage your repositories, projects, settings" tab; 4. The Development Team Leader must navigate to the "Repositories" tab; 5. The Development Team Leader must navigate to the "Connect Repo using SSH"; 6. The Development Team Leader must insert a name for the repository; 7. The Development Team Leader must include the repository ssh URL; 8. The Development Team Leader must include the SSH private associated with the AWS CodeCommit Repository; 9. The Development Team Leader must navigate to the "Connect" button. 10. The Development Team Leader must click on the "Sync" button; 11. The Development Team Leader must click on the "Synchronize" button.
Results	The application is deployed into the AWS EKS Cluster and accessible via the Kubernetes Service associated with the deployment.

Name	BS-18: Prometheus Deployed to AWS EKS Cluster State
Requirements	The application is running successfully in the AWS EKS Cluster;

	<p>The application is configured to export its custom metrics to the “/metrics” endpoint;</p> <p>A Command Line Interface;</p>
Preconditions	<p>Helm deployment tool cli installed.</p> <p>values.yaml configuration file including the application endpoint from which Prometheus must scrape custom metrics.</p>
Workflow	<ol style="list-style-type: none"> 1. A DevOps Maintainer creates a namespace for Prometheus with the following command: <code>kubectl create namespace prometheus</code> 2. A DevOps Maintainer deploys prometheus by using the helm deployment tool with the following command: <code>helm upgrade -I Prometheus Prometheus-community/Prometheus \</code> <code>--namespace Prometheus \</code> <code>--values values.yaml \</code> <code>--set</code> <code>alertmanager.persistentVolume.storageClass="gp2",server.persistentVolume.storageClass="gp2"</code> 3. A DevOps Maintainer exposes the Prometheus Server with an endpoint by using the following command: <code>kubectl patch svc prometheus-server -n prometheus -p</code> <code>'{"spec": {"type": "LoadBalancer"}}'</code>
Results	<p>Prometheus is successfully deployed into the AWS EKS cluster.</p>

Name	BS-19: Grafana Deployed to AWS EKS Cluster State
Requirements	<p>The application is running successfully in the AWS EKS Cluster;</p> <p>A Command Line Interface;</p>
Preconditions	<p>Helm deployment tool cli installed;</p> <p>BS-18: Prometheus Deployed to AWS EKS Cluster State.</p>
Workflow	<ol style="list-style-type: none"> 1. A DevOps Maintainer creates a namespace for Grafana with the following command: <code>kubectl create namespace grafana</code> 2. A DevOps Maintainer deploys grafana by using the helm deployment tool with the following commands: <ul style="list-style-type: none"> - <code>helm repo add grafana https://grafana.github.io/helm-charts --namespace grafana;</code> - <code>helm repo update;</code>

	<ul style="list-style-type: none"> - helm install grafana grafana/grafana --namespace grafana; <p>3. A DevOps Maintainer exposes the grafana Server with an endpoint by using the following command:</p> <pre>kubectl patch svc grafana -n grafana -p '{"spec": {"type": "LoadBalancer"}}'</pre> <p>4. Roughly 2 minutes later, the grafana server endpoint will be accessible. A DevOps Maintainer must access the endpoint a perform a login with the following credentials:</p> <ul style="list-style-type: none"> - Username: admin; - Password: the default password is exposed as the Grafana service hostname encoded in a base64 encoding. To retrieve the password, in a command line terminal, use the following command: <pre>kubectl get secret --namespace grafana -o jsonpath=".data.admin-password" base64 --decode ; echo</pre>
Results	<ol style="list-style-type: none"> 1. Grafana is successfully deployed into the AWS EKS cluster; 2. The DevOps maintainer is successfully logged into the Grafana Server.

Name	BS-20: Elasticsearch Deployed to AWS EKS Cluster State
Requirements	The application is running successfully in the AWS EKS Cluster; A Command Line Interface;
Preconditions	Helm deployment tool cli installed;
Workflow	<ol style="list-style-type: none"> 1. A DevOps Maintainer must create a namespace for the Elasticstack elements with the following command: Kubectl create namespace elasticstack 2. A DevOps Maintainer must install elasticsearch into the AWS EKS Cluster with the helm deployment tool, with the following command: helm install elasticsearch elastic/elasticsearch --namespace elasticstack --values values-elasticsearch.yaml <p>*The file called values-elasticsearch.yaml contains a definition of the path in which the STDOUT log files are kept within the node;</p> <ol style="list-style-type: none"> 3. A DevOps Maintainer must expose the elasticsearch-master service to an endpoint with the following command: <pre>kubectl patch svc grafana -n grafana -p '{"spec": {"type": "LoadBalancer"}}'</pre>

	4. After about two minutes, the elasticsearch endpoint will be provisioned, and accessible.
Results	Elasticsearch is successfully deployed into the AWS EKS cluster;

Name	BS-21: Filebeat Deployed to AWS EKS Cluster State
Requirements	The application is running successfully in the AWS EKS Cluster; A Command Line Interface;
Preconditions	Helm deployment tool cli installed; BS-20: Elasticsearch Deployed to AWS EKS Cluster State
Workflow	A DevOps Maintainer must install filebeat into the AWS EKS Cluster with the helm deployment tool, with the following command: <pre>helm install filebeat elastic/filebeat --namespace elasticstack --values values-filebeat.yaml</pre> *The file called values-filebeat.yaml contains a definition of the path in which the STDOUT log files are kept within the node, how the logs must be processed from the log files and the elasticsearch endpoint to which the logs must be forwarded;
Results	Elasticsearch is successfully deployed into the AWS EKS cluster;

Name	BS-22: Kibana Deployed to AWS EKS Cluster State
Requirements	The application is running successfully in the AWS EKS Cluster; A Command Line Interface;
Preconditions	Helm deployment tool cli installed; BS-20: Elasticsearch Deployed to AWS EKS Cluster State; BS-21: Filebeat Deployed to AWS EKS Cluster State.
Workflow	1. A DevOps Maintainer must install Kibana into the AWS EKS Cluster with the helm deployment tool, with the following command:

	<pre>helm install kibana elastic/kibana --namespace elasticstack --values values-kibana.yaml</pre> <p>2. A DevOps Maintainer must expose the kibana-kibana service to an endpoint with the following command:</p> <pre>kubectl patch svc kibana -n elasticstack -p '{"spec": {"type": "LoadBalancer"}}'</pre> <p>*The file called values-kibana.yaml contains a definition of the elasticsearch endpoint to which the logs must be extracted and presented on the Kibana server;</p> <p>3. After about two minutes, the kibana endpoint will be provisioned, and accessible.</p>
Results	Kibana is successfully deployed into the AWS EKS cluster;

Test Cases

Each test case is identified by its name, the requirements that it looks to evaluate and validate, the preconditions that are required to perform the test, and finally the results that are expected by the end of the test. The following Table 17 provides a shortlist of the test cases used to perform the validation process. The examples given each refer to a base case used in each of the four main elements of the framework: Infrastructure Provisioning, Continuous Integration, Continuous Delivery and Monitoring.

Name	TC-01: A DevOps Team Leader Creates the Infrastructure Pre-Requisites
Requirements	FR-01: IAM Authorization – Infrastructure Provisioning
Preconditions	BS-02: DevOps Team Leader Logged In State.
Steps	<ol style="list-style-type: none"> The DevOps Team Leader authenticates their AWS CLI with the credentials of a DevOps Team Leader account; The DevOps Team Leader executes the following command: <pre>aws cloudformation create-stack \ --stack-name PreRequisitesInfra \ --template-body file://PreRequisites.yaml \ --capabilities CAPABILITY_NAMED_IAM</pre>
Expected Results	<ol style="list-style-type: none"> The Stack ID of the pre-requisites stack is printed in the terminal; The pre-requisites stack is successfully created.

Name	TC-02: A DevOps Maintainer Attempts to Create the Infrastructure Pre-Requisites
Requirements	FR-01: IAM Authorization.
Preconditions	BS-03: DevOps Maintainer Logged In State.
Steps	<ol style="list-style-type: none"> 1. A DevOps Maintainer authenticates their AWS CLI with the credentials of a DevOps Maintainer account; 2. A DevOps Maintainer executes the following command: <pre>aws cloudformation create-stack \ --stack-name PreRequisitesInfra \ --template-body file://PreRequisites.yaml \ --capabilities CAPABILITY_NAMED_IAM</pre>
Expected Results	<ol style="list-style-type: none"> 1. An error message appears in the terminal saying the following: 2. The pre-requisites stack is not created.

Name	TC-03: A DevOps Maintainer performs the first commit to the “develop” branch
Requirements	FR-04: Infrastructure Creation.
Preconditions	BS-03: DevOps Maintainer Logged In State; BS-04: Pre-Requisites Created State – Infrastructure Provisioning; BS-05: Ready For Development State – Infrastructure Provisioning; BS-06: Infrastructure Templates Created State.
Steps	<ol style="list-style-type: none"> 1. A DevOps Maintainer develops the infrastructure templates: 2. A DevOps Maintainer pushes the infrastructure templates to the AWS CodeCommit repository.
Expected Results	The infrastructure is automatically created according to the configuration templates.

Name	TC-04: A DevOps Maintainer performs a commit to the “develop” branch
Requirements	FR-05: Infrastructure Update.
Preconditions	BS-07: Infrastructure Created State.
Steps	<ol style="list-style-type: none"> 1. A DevOps Maintainer develops the infrastructure templates; 2. A DevOps Maintainer pushes the infrastructure templates to the AWS CodeCommit repository.
Expected Results	The infrastructure is automatically updated with the new configuration.

Name	TC-05: A DevOps Team Leader Creates a New Branch named “staging”
Requirements	FR-06: Infrastructure Replication.
Preconditions	BS-05: Ready For Development State; BS-07: Infrastructure Created State.
Steps	<ol style="list-style-type: none"> 1. A DevOps Maintainer creates a new branch in the git repository locally, named “staging”, stemming from the “main” branch; 2. A DevOps Maintainer publishes the new branch to the AWS CodeCommit repository.
Expected Results	The infrastructure existent in the root branch is replicated, with alterations to the name of the stack which now includes the “staging” branch name.

Name	TC-06: A DevOps Team Leader Deletes a Branch in the AWS CodeCommit Repository
Requirements	FR-07: Infrastructure Deletion.
Preconditions	BS-07: Infrastructure Created State.

Steps	A DevOps Team Leader deletes a branch in the AWS CodeCommit Repository in the AWS CodeCommit console.
Expected Results	The Infrastructure associated with the deleted branch is also deleted.

Name	TC-07: A DevOps Maintainer performs a Pull Request to the “main” branch
Requirements	FR-09: Repository Approval Scheme – Infrastructure Provisioning.
Preconditions	BS-06: Infrastructure Templates Created State; BS-07: Infrastructure Created State.
Steps	<ol style="list-style-type: none"> 1. A DevOps Maintainer creates a new pull request between the “develop” branch and the “main” branch; 2. A DevOps Team Leader approves the pull request.
Expected Results	The production infrastructure is updated according to the new infrastructure Templates.

Name	TC-08: Verify Infrastructure Components
Requirements	FR-10: Create VPC – FR-18: Create a DNS Routing Service.
Preconditions	BS-07: Infrastructure Created State.
Steps	<ol style="list-style-type: none"> 1. A DevOps Maintainer verifies the AWS CloudFormation console to guarantee that the infrastructure components were successfully created; 2. A DevOps Maintainer navigates to the “Template” tab in the desired stack; 3. A DevOps Maintainer navigates to the “View In Designer” button.
Expected Results	<ol style="list-style-type: none"> 1. The infrastructure components are all in the “CREATE_COMPLETE” state;

	2. The infrastructure is shown in a diagram in the AWS CloudFormation Designer tool.
--	--

Name	TC-09: SSH Connection to a Bastion Host With an Approved IP Address
Requirements	FR-19: Possibility to SSH Connection to Bastion Host from Specific Range of IP Addresses.
Preconditions	BS-07: Infrastructure Created State.
Steps	<ol style="list-style-type: none"> 1. The DevOps Maintainer must guarantee that their machine is either connected to the WIT network directly on-premise or by using the organization's VPN; 2. The DevOps Maintainer must have an authorized key-pair which allows access to the Bastion Host EC2 instance; 3. The DevOps Maintainer executes the following command: <pre>ssh -i KeyPair.pem ec2-user@bastion_host_DNS.amazonaws.com.</pre>
Expected Results	The terminal now has direct access to the Bastion Host EC2 instance, with the ec2 user and host now resembling ec2-user@bastion_hostname \$.

Name	TC-10: SSH Connection to a Bastion Host Without an Approved IP Address
Requirements	FR-20: Block to SSH Connection to Bastion Host, from a Machine with a Non-Authorized IP Address.
Preconditions	BS-07: Infrastructure Created State.
Steps	<ol style="list-style-type: none"> 1. The DevOps Maintainer is not connected to the WIT network and does not have an authorized IP address; 2. The DevOps Maintainer attempts an SSH connection to the Bastion Host with the following command: <pre>ssh -i KeyPair.pem ec2-user@bastion_host_DNS.amazonaws.com.</pre>

Expected Results	A connection timeout error is presented on the screen after 60 seconds.
------------------	---

Name	TC-11: SSH Connection from a Bastion Host to a Node in the EKS Cluster
Requirements	FR-21: Possibility to SSH Connection from Bastion Host to EKS Cluster Nodes.
Preconditions	BS-08: SSH Connection Established to Bastion Host State.
Steps	<p>Within the ssh connection, the DevOps Maintainer executes the following command to perform an SSH connection to the EKS Cluster node with verbose messaging to use an authorized SSH key-pair:</p> <pre>ssh -v ec2-user@eks_node_ip_address</pre>
Expected Results	The terminal now has direct access to the specified EKS Node EC2 instance, with the ec2 user and host now resembling ec2-user@bastion_hostname \$.

Name	TC-12: Kubectl Tool Access to the EKS Cluster from Within a Bastion Host
Requirements	FR-22: Kubectl Tool Access to the EKS Cluster from Within a Bastion Host.
Preconditions	BS-08: SSH Connection Established to Bastion Host State.
Steps	<ol style="list-style-type: none"> 1. A DevOps Maintainer installs the Kubectl tool with the following commands: <ul style="list-style-type: none"> - curl -LO https://dl.k8s.io/release/\$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl - sudo install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl 2. A DevOps Maintainer configures the Kubectl tool context with the EKS cluster with the following command: <ul style="list-style-type: none"> - aws eks update-kubeconfig --region region-code --name cluster-name 3. A DevOps Maintainer uses the following command to retrieve the pods running on the following command:

	- kubectl get pods.
Expected Results	A list of the current running pods are shown in the terminal.

Name	TC-13: A Development Team Leader Creates the Infrastructure Pre-Requisites
Requirement	FR-24: IAM Authorization – Continuous Integration.
Preconditions	BS-10: Development Team Leader Logged In State.
Steps	<ol style="list-style-type: none"> 1. The Development Team Leader authenticates their AWS CLI with the credentials of a Development Team Leader account; 2. The Development Team Leader executes the following command: <pre>aws cloudformation create-stack \ --stack-name PreRequisitesCI \ --template-body file://PreRequisites.yaml \ --capabilities CAPABILITY_NAMED_IAM.</pre>
Expected Results	<ol style="list-style-type: none"> 1. The Stack ID of the pre-requisites stack is printed in the terminal; 2. The pre-requisites stack is successfully created.

Name	TC-14: A Development Team Member Attempts to Create the Infrastructure Pre-Requisites
Requirements	FR-24: IAM Authorization – Continuous Integration.
Preconditions	BS-11: Development Team Member Logged In State.
Steps	<ol style="list-style-type: none"> 1. A Development Team Member authenticates their AWS CLI with the credentials of a DevOps Maintainer account; 2. A Development Team Member executes the following command: <pre>aws cloudformation create-stack \ --stack-name PreRequisitesCI \ --capabilities CAPABILITY_NAMED_IAM.</pre>

	<pre>--template-body file://PreRequisites.yaml \ --capabilities CAPABILITY_NAMED_IAM.</pre>
Expected Results	<ol style="list-style-type: none"> 1. An error message appears in the terminal saying the following: 2. The pre-requisites stack is not created.

Name	TC-15: A Development Team Leader Creates a New Branch named “develop”
Requirements	FR-27: Automated Pipeline Creation.
Preconditions	BS-13: Ready For Development State – Continuous Integration.
Steps	<ol style="list-style-type: none"> 1. A Development Team Member creates a new branch in the git repository locally, named “develop”, stemming from the “main” branch; 2. A Development Team Member publishes the new branch to the AWS CodeCommit repository.
Expected Results	A new AWS CodePipeline is created with the name of the project and the branch name, separated by a hyphen, as its name.

Name	TC-16: A Development Team Leader Creates a New Branch named “staging”
Requirements	FR-27: Automated Pipeline Creation.
Preconditions	BS-13: Ready For Development State – Continuous Integration.
Steps	<ol style="list-style-type: none"> 1. A Development Team Member creates a new branch in the git repository locally, named “staging”, stemming from the “main” branch; 2. A Development Team Member publishes the new branch to the AWS CodeCommit repository.
Expected Results	A new AWS CodePipeline is created with the name of the project and the branch name, separated by a hyphen, as its name.

Name	TC-17: A Development Team Leader Creates a New Branch named “lab”
Requirements	FR-27: Automated Pipeline Creation.
Preconditions	BS-13: Ready For Development State – Continuous Integration.
Steps	<ol style="list-style-type: none"> 1. A Development Team Member creates a new branch in the git repository locally, named “lab”, stemming from the “main” branch; 2. A Development Team Member publishes the new branch to the AWS CodeCommit repository.
Expected Results	A new AWS CodePipeline is created with the name of the project and the branch name, separated by a hyphen, as its name.

Name	TC-18: A Development Team Member performs a commit to the “develop” branch
Requirements	FR-29: Automated Code Build on Commit.
Preconditions	BS-14: Application Source Code Created State.
Steps	<ol style="list-style-type: none"> 1. A Development Team Member develops the application source code; 2. A Development Team Member pushes the application source code to the AWS CodeCommit repository.
Expected Results	<ol style="list-style-type: none"> 1. The AWS CodePipeline service associated with the “develop” branch is alerted of the new version of the application source code; 2. The AWS CodeBuild project associated with the “develop” branch is triggered and builds the application source code according to the project’s <i>buildspec</i> file; 3. The Docker Container Image produced by the AWS CodeBuild is pushed to the AWS ECR repository.

Name	TC-19: A Development Team Member performs a commit to the “staging” branch
-------------	---

Requirements	FR-29: Automated Code Build on Commit.
Preconditions	BS-14: Application Source Code Created State.
Steps	<ol style="list-style-type: none"> 1. A Development Team Member develops the application source code; 2. A Development Team Member pushes the application source code to the AWS CodeCommit repository.
Expected Results	<ol style="list-style-type: none"> 1. The AWS CodePipeline service associated with the “staging” branch is alerted of the new version of the application source code; 2. The AWS CodeBuild project associated with the “staging” branch is triggered and builds the application source code according to the project’s <i>buildspec</i> file. <p>The Docker Container Image produced by the AWS CodeBuild is pushed to the AWS ECR repository.</p>

Name	TC-20: A Development Team Member performs a commit to the “develop” branch
Requirements	FR-29: Automated Code Build on Commit.
Preconditions	BS-14: Application Source Code Created State.
Steps	<ol style="list-style-type: none"> 1. A Development Team Member develops the application source code; 2. A Development Team Member pushes the application source code to the AWS CodeCommit repository.
Expected Results	<ol style="list-style-type: none"> 1. The AWS CodePipeline service associated with the “lab” branch is alerted of the new version of the application source code; 2. The AWS CodeBuild project associated with the “lab” branch is triggered and builds the application source code according to the project’s <i>buildspec</i> file. <p>The Docker Container Image produced by the AWS CodeBuild is pushed to the AWS ECR repository.</p>

Name	TC-21: A Development Team Member performs a Pull Request to the “main” branch
Requirements	FR-25: Repository Approval Scheme – Continuous Integration.
Preconditions	BS-14: Application Source Code Created State.
Steps	<ol style="list-style-type: none"> 2. A Development Team Member creates a new pull request between the “develop” branch and the “main” branch; 3. A Development Team Leader approves the pull request.
Expected Results	The production infrastructure is updated according to the new application source code files.

Name	TC-22: Static Code Analysis
Requirement	FR-30: Automated Static Code Analysis Module.
Preconditions	BS-15: Code Committed State.
Steps	Given the automated nature of the pipeline, there are no steps required other than specifying in the terminal commands in <i>buildspec</i> that specify which tool to perform the static code analysis with and on which files. It is possible to use wildcards such as “*.js” to specify that the analysis must be made on all files what follow a JavaScript format.
Expected Results	10/10 evaluation for successful code analysis.

Name	TC-23: OWASP Dependency Verification
Requirement	FR-31: Automated Dependency Verification Module.
Preconditions	BS-15: Code Committed State.
Steps	Given the automated nature of the pipeline, there are no steps required other than specifying in the terminal commands in <i>buildspec</i> that specify which files the OWASP dependency tool must perform the

	dependency verification. It is possible to use wildcards such as “*.js” to specify that the analysis must be made on all files what follow a JavaScript format.
Expected Results	<ol style="list-style-type: none"> 1. Successful dependency verification with exit code 0; 2. Email with dependency verification report.

Name	TC-24: Unit Testing
Requirement	FR-31: Automated Unit Testing Module.
Preconditions	BS-15: Code Committed State.
Steps	The Unit tests which will run and the functions and scripts which will be evaluated must be defined in the Unit.js file which includes the different results that are expected as outputs with the module.assert("result") function.
Expected Results	Successful Unit testing validation.

Name	TC-25: Docker Container Image Creation
Requirement	FR-33: Automated Docker Image Build. FR-34: Automated Docker Image Push to AWS ECR Repository.
Preconditions	BS-15: Code Committed State.
Steps	Given the automated nature of the pipeline, there are no steps required other than specifying in the terminal commands in <i>buildspec</i> that specify that there must be a Docker Container Image built according to the Dockerfile, included with the application source code. There must also be commands in the <i>buildspec</i> file that must perform the automated push to the AWS ECR Repository.
Expected Results	<ol style="list-style-type: none"> 1. A Docker Container Image artifact; 2. Docker Container Image pushed to the AWS ECR Repository.

Name	TC-26: AWS ECR Repository Re-Tagging
Requirement	FR-35: Automated Docker Image Re-Tagging.
Preconditions	BS-14: Code Committed State.
Steps	Given the automated nature of the pipeline, there are no steps required other than specifying in the terminal commands in <i>buildspec</i> that specify how the re-tagging process must be performed on the Docker Container Image.
Expected Results	The Docker Container Image pushed to the AWS ECR Repository must be tagged as “latest”, with the previous version following the same sequence as the previous versions. For example, if the previous version was 0.1.9, the version that is retagged must have 0.1.10 as its tag.

Name	TC-27: Build State Change E-mail Notification
Requirement	FR-38: Build State Change Notification Service.
Preconditions	BS-14: Code Committed State.
Steps	In the event of the build failing, the pre-requisite stack provisions a means of subscribing an e-mail address to an AWS Simple Notification Service which notifies the subscription of changes in the state of the AWS CodeBuild project.
Expected Results	Email notification alerting if the build Project had a state change (“IN_PROGRESS”, “STOPPED”, “FAILED”, “SUCCESSFUL”).

Name	TC-28: Automated Rollout Strategy – Canary Release
Requirement	FR-4: Configurable Rollout Strategy – Canary Release.
Preconditions	BS-17: Application Deployed with Argo CD State.

Steps	This process is conducted autonomously by the Argo Rollouts plugin.
Expected Results	The new version of the application replaces the previous version in its entirety.

Name	TC-27: Automated Rollback on Failure
Requirement	FR-45: Automated Rollback on Failure.
Preconditions	BS-17: Application Deployed with Argo CD State.
Steps	<p>This process is conducted autonomously by the Argo Rollouts plugin. The Argo Rollouts plugin is configured with a YAML file which defines which conditions must be maintained for the rollout to be successful.</p> <p>In the event of the condition not being met, the Argo Rollouts plugin scales the new, unstable version down to zero, and re-scales the previous, stable version back to its production level.</p>
Expected Results	The previous, stable version continues in production, whilst the new version is considered degraded.

Name	TC-28: Resource Consumption Visualization
Requirement	FR-47: Container CPU Consumption Visualisation; FR-48: Container Memory Consumption Visualisation; FR-49: Container Bandwidth Consumption Visualisation; FR-50: Container Disk Utilisation Visualisation; FR-51: Container Count; FR-52: Container Health Visualisation;
Preconditions	BS-18: Prometheus Deployed to AWS EKS Cluster State BS-19: Grafana Deployed to AWS EKS Cluster State
Steps	An Operations Team Member or DevOps Maintainer must create the various dashboards associated with the metrics that are required.

Expected Results	Grafana displays the various dashboards associated with the queried metrics.
------------------	--

Name	TC-29: Application Custom Metrics Visualization
Requirement	FR-56: Custom Application Metrics
Preconditions	BS-18: Prometheus Deployed to AWS EKS Cluster State BS-19: Grafana Deployed to AWS EKS Cluster State
Steps	An Operations Team Member or DevOps Maintainer must create the dashboards associated with the metrics that are created by the application with the specified metrics.
Expected Results	Grafana displays the dashboards associated with the custom metrics of the application.

Name	TC-30: View Application Logs
Requirement	FR-57: Application Log Visualisation.
Preconditions	BS-20: Elasticsearch Deployed to AWS EKS Cluster State; BS-21: Filebeat Deployed to AWS EKS Cluster State; BS-22: Kibana Deployed to AWS EKS Cluster State.
Steps	An Operations Team Member or DevOps Maintainer must create the dashboards associated with the metrics that are created by the application with the specified metrics.
Expected Results	Grafana displays the dashboards associated with the custom metrics of the application.

Name	TC-29: Query a Specific Container Application Logs
Requirement	FR-58: Application Log Querying

Preconditions	BS-20: Elasticsearch Deployed to AWS EKS Cluster State; BS-21: Filebeat Deployed to AWS EKS Cluster State; BS-22: Kibana Deployed to AWS EKS Cluster State.
Steps	An Operations Team Member or DevOps Maintainer must create the dashboards associated with the metrics that are created by the application with the specified metrics.
Expected Results	Grafana displays the dashboards associated with the custom metrics of the application.

Test Results

The test cases conducted produced one of two possible results: passed without errors or failed. In the event that the test case passed without errors, no action was necessary, and the requirement was considered as successfully implemented. If a test were to fail, each failure required that a report be made of which test case failed, when the test was performed and by who, what incident occurred, and the actions required to correct the issue with the implementation. Once the correction was made, the framework would be re-submitted to the same test case and would be required to pass this repeated test. If the test were passed, the developer would have to report when the change was implemented, by who, and what correction was made that caused the framework to comply with the tests made.

Infrastructure Provisioning

Table 41. Infrastructure Provisioning Test Results

Test Case	Date	Tester	P/F	Comments
TC-01	28/03/2022	GP	F	Failed
TC-04	29/03/2022	GP	P	Passed, without errors.
TC-08	30/03/2022	GP	P	Passed, without errors.
TC-10	31/03/2022	GP	P	Passed, without errors.
TC-11	31/03/2022	GP	F	Failed.

Continuous Integration

Table 42. Continuous Integration Test Results

Test Case	Date	Tester	P/F	Comments
TC-13	11/04/2022	GP	P	Passed, without errors.
TC-14	11/04/2022	GP	P	Passed, without errors.
TC-21	12/04/2022	GP	P	Passed, without errors.
TC-22	13/04/2022	GP	P	Passed, without errors.
TC-23	13/04/2022	GP	P	Passed, without errors.
TC-26	14/04/2022	GP	F	Failed.
TC-27	14/04/2022	GP	P	Passed, without errors.

Continuous Delivery

Table 43. Continuous Delivery Test Results

Test Case	Date	Tester	P/F	Comments
TC-28	26/05/2022	GP	P	Passed, without errors.
TC-29	26/05/2022	GP	F	Failed.

Monitoring Microservices

Table 44. Microservices Monitoring Test Results

Test Case	Date	Tester	P/F	Comments
TC-30	02/06/2022	GP	P	Passed, without errors.
TC-31	02/06/2022	GP	P	Passed, without errors.
TC-32	03/06/2022	GP	F	Failed partially. The logs presented were not correctly parsed at the time this test was conducted. This incident was corrected <i>a posteriori</i> .
TC-33	03/06/2022	GP	F	Failed Partially. Due to the previous test case (TC-32). The possibility to query the logs was still possible, but not in the manner that in its entirety.

Incident Reports

Test	TC-01: A DevOps Team Leader Creates the Infrastructure Pre-Requisites
Date; Tester.	28/03/2022; Gabriel Pinheiro
Incident	Permissions were missing in the DevOps Team Leader IAM Role.
Actions	Include the permissions required for the DevOps Team Leader to be able to create the pre-requisites for the infrastructure pipeline.
Date; Developer	04/04/2022; Gabriel Pinheiro
Correction	The DevOps Team Leader IAM Role was corrected.

Test	TC-11: SSH Connection from a Bastion Host to a Node in the EKS Cluster
Date; Tester.	28/03/2022; Gabriel Pinheiro
Incident	It was not possible to perform an SSH connection between a Bastion Host and a Node in the EKS Cluster. The connection reaches a timeout.
Actions	Correct Security Group ingress settings to include connections from the Bastion Host Security Group.
Date; Developer	04/04/2022
Correction	The EKS Cluster Security Group was updated to allow SSH connections from the Bastion Host Security Group.

Test	TC-26: AWS ECR Repository Re-Tagging
Date; Tester.	14/04/2022; Gabriel Pinheiro
Incident	The previous Docker Container Image was not successfully retagged. When a new version of the image is pushed to the repository, the previous version is left in an “<untagged>” state.
Actions	Correct re-tagging process to reflect the correct versions of the images.
Date; Developer	18/04/2022; Gabriel Pinheiro
Correction	The images are now retagged successfully.

Test	TC-29: Automated Rollback on Failure
-------------	---

Date; Tester.	26/05/2022; Gabriel Pinheiro
Incident	An error was purposely included in a new version, and it still followed through to production.
Actions	Correct method of automatic rollbacks.
Date; Developer	25/05/2022; Gabriel Pinheiro
Correction	Rollbacks are now dependant on application metrics and if an error is discovered in the analysis of the logs, the rollback is automatically carried out.

Test	TC-32: View Application Logs
Date; Tester.	03/06/2022; Gabriel Pinheiro.
Incident	The logs are produced by the application in JSON but are not correctly parsed to Kibana according to the different tags.
Actions	Correct JSON parsing to reflect the different tags in the logs.
Date; Developer	06/06/2022; Gabriel Pinheiro
Correction	The logs are now parsed correctly and indexed according to their tags.

Test	TC-33: Query a Specific Container Application Logs
Date; Tester.	03/06/2022; Gabriel Pinheiro.
Incident	Given the lack of parsing of the logs, it is not possible to query the logs according to the tags produced by the application.
Actions	Correct JSON parsing to reflect the different tags in the logs.
Date; Developer	06/06/2022; Gabriel Pinheiro
Correction	The logs are now parsed correctly and indexed according to their tags.