INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA

Framework DevOps para Projetos de Integração

JORGE GABRIEL FONTES MACIEL AZEVEDO
Junho de 2023

POLITÉCNICO
DO PORTO

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

DevOps Framework for Integration Projects

Jorge Gabriel Azevedo

A dissertation submitted in partial fulfillment of the requirements for the degree of Master of Science, Specialisation Area of Information and Knowledge Systems

ISEP Supervisor: Nuno Ferreira
Deloitte Supervisor: David Forte

Porto, June 29, 2023

Statement of Integrity

I hereby declare having conducted this academic work with integrity.

I have not plagiarised or applied any form of undue use of information or falsification of results along the process leading to its elaboration.

Therefore the work presented in this document is original and authored by me, having not previously been used for any other end.

I further declare that I have fully acknowledged the Code of Ethical Conduct of P.PORTO.

ISEP, Porto, June 29, 2023

Dedicatory

To my family, who supported me during my journey. A special acknowledgment goes to my extraordinary sister, whose constant presence and resilience have served as an inspiration to overcome the countless hurdles we encountered together.

Abstract

Deloitte is a company recognized for its services in Consulting and Audit. Integration Services is one of the teams in Consulting at Deloitte and its expertise is developing integration layers for enterprise applications, enabling decoupled communications between them. Some of the common factors in each project are source code management, moving new developments to production, and the developer's relation with all the remaining teams in the project special the operations team. Historically each project has its own way of doing each of the points mentioned before. These processes are tightly integrated with the factors around the projects when defining it, such as infrastructure, team composition, team experience, etc... As consulting professionals, Deloitte employees usually change between projects which means they need to learn these project-specific processes when they join a new project, and this is a very time-consuming activity until they are completely comfortable.

The goal of this project is to create a standard framework, with its own processes, tools, and rules for these activities. The intent is to reduce the strain on the consultants when changing between projects, with the added benefit of having an asset that can evolve with time. It will be refined in each iteration and built with the knowledge of many professionals. All of this is to be able to adapt to a wider scope of requirements Deloitte can benefit by reusing it. The customers will also receive a much more complete product they can rely on even in more extreme circumstances like migrating the infrastructure where the applications are running.

It was developed as a Proof of Concept for the project, and when compared with the previous implementations, it improved the development implementation time in new projects. Was identified a performance penalty of around 3 minutes in each CI/CD pipeline execution. It was suggested to switch the Proof of Concept implementation from Groovy to Python in order to lessen the problem. The team is very interested in the project and the future benefits that can be derived from it.

Keywords: DevOps, CI/CD, Agnostic Framework, Integration Services

Resumo

A Deloitte é uma empresa reconhecida pelos seus serviços em Consultoria e Auditoria. A equipa de *Integration Services* é uma das equipas de Consultoria da Deloitte e a sua especialidade é o desenvolvimento de camadas de integração para aplicações empresariais, permitindo comunicações desacopeladas entre elas. Alguns dos fatores comuns em cada projeto são o gerenciamento do código-fonte, a promoção de novos desenvolvimentos para a produção e a relação do programador com as restantes equipas do projeto, especialmente a equipa de operações. Historicamente, cada projeto tem sua própria maneira de fazer cada um dos pontos mencionados anteriormente. Esses processos são fortemente integrados com os fatores em torno dos projetos ao defini-los, como infraestrutura, composição da equipa, experiência da equipa, etc... Como é habitual em profissionais de consultoria, os consultores da Deloitte geralmente mudam entre projetos, o que significa que eles precisam aprender esses processos específicos do projeto quando se juntam a um novo projeto, e esta é uma atividade muito demorada até que estejam completamente confortáveis.

O objetivo deste projeto é criar um *framework* padrão, com seus próprios processos, ferramentas e regras para essas atividades. A intenção é reduzir a pressão sobre os consultores ao mudar entre projetos, com o benefício adicional de ter um ativo que pode evoluir com o tempo. Este será refinado em cada iteração e construído com o conhecimento de muitos profissionais. Tudo isto para poder adaptar-se a um leque mais alargado de requisitos que a Deloitte pode beneficiar com a sua reutilização. Os clientes também receberão um produto muito mais completo em que podem confiar, mesmo em circunstâncias mais extremas, como migração da infraestrutura onde os aplicações estão a ser executados.

Foi desenvolvido como uma Prova de Conceito para o projeto, e quando comparado com as implementações anteriores, melhorou o tempo de implementação do desenvolvimento em novos projetos. Foi identificada uma redução de desempenho de aproximadamente de 3 minutos por cada execução do pipeline CI/CD. Foi proposto plano proposto para mitigar o problema migrando a implementação da atual prova de conceito de *Groovy* para o *Python*. A equipa como um todo demonstrou-se bastante interessada no projeto e nos futuros benefícios que podem ser retirados dele.

Acknowledgement

I would like to express my deepest gratitude to the following individuals who have played a significant role in my journey and have contributed to the completion of this thesis.

First and foremost, I am immensely thankful to my family for their unwavering support, love, and encouragement throughout these years. Their belief in me and their constant presence have been invaluable.

I would like to extend a special appreciation to my dear friend Nuno Dinis. Over the past seven years, he has been by my side, providing unwavering support, motivation, and guidance. His friendship has been a constant source of strength and inspiration.

I also want to acknowledge all my friends who have stood by me during this academic pursuit. Your encouragement, understanding, and camaraderie have made this journey more enjoyable and memorable.

Furthermore, I would like to express my gratitude to my colleagues at Deloitte, with a special mention to my mentor Sergio Lopes. His unwavering assistance, invaluable expertise, and visionary perspective have played a crucial role in shaping my research and fostering my professional development.

I am also indebted to my supervisor, Nuno Ferreira, for his invaluable guidance, mentorship, and expertise throughout this thesis. His profound knowledge, dedication, and commitment to academic excellence have significantly influenced the development and quality of my research.

Lastly, I extend my thanks to all the teachers, professors, and mentors who have guided me throughout my academic career. Your knowledge, guidance, and passion for education have profoundly influenced my intellectual development.

To everyone mentioned above and those who have supported me in various ways, I am truly grateful. Your contributions have played an integral part in the realization of this thesis, and I am honored to have had such incredible individuals in my life.

Contents

List of Figures xvii

List of Tables xix

List of Source Code xxi

List of Abbreviations xxiii

1 Introduction 1
- 1.1 Context . . . 1
- 1.2 Problem . . . 2
- 1.3 Project Vision . . . 3
  - 1.3.1 Project Goal . . . 4
  - 1.3.2 Expected final result . . . 4
- 1.4 Project Restrictions . . . 4
- 1.5 Approach to the Problem . . . 6
- 1.6 Code of Ethics . . . 6
- 1.7 Thesis Structure . . . 6

2 State of the Art 9
- 2.1 Research Questions . . . 9
- 2.2 Scientific Research . . . 10
  - 2.2.1 Selection Criteria . . . 10
  - 2.2.2 Research Results . . . 10
- 2.3 Technology Research . . . 11
  - 2.3.1 Term research . . . 11
    - Cloud Platforms . . . 11
    - CI Platforms . . . 11
    - Integration Platforms . . . 12
    - Containerization Platforms . . . 12
  - 2.3.2 90 Days of DevOps . . . 13
- 2.4 Research Answers . . . 13
  - 2.4.1 How is DevOps defined in the literature? . . . 13
    - Collaboration . . . 13
    - Software Development Process . . . 14
    - Software Delivery . . . 15
    - CI/CD . . . 17
    - DevOps Metamodel . . . 19
  - 2.4.2 What GIT workflows are used to manage the different stages of development? . . . 20
  - 2.4.3 How are CI/CD pipelines defined? . . . 22

2.4.4 What CI/CD agnostic processes were already implemented? 23
2.4.5 How does CI/CD pipeline are monitored? 24

3 Value Analysis **25**
3.1 Opportunity Identification 25
3.2 Value Proposition 26
3.2.1 Is already someone trying to do something similar? 28
3.2.2 What are the limitations of this proposed solution? 28
3.3 Quality Function Deployment (QFD) 28

4 Framework Design **31**
4.1 Project Scope 31
4.2 Proposals Presentation 32
4.2.1 Proposal A 32
4.2.2 Proposal B 34
4.3 Solution Selection 36
4.4 Additional Requirements 36

5 Framework Development **39**
5.1 Architecture 39
5.2 Source Code Repository 42
5.2.1 Source Code Structure 42
5.2.2 Branching 42
5.2.3 Git Workflow 42
5.3 Pipeline Shared Library 44
5.3.1 CI/CD Process 44
5.3.2 Library File Structure 47
Internal Folders 47
5.3.3 Script Orchestration 48
Pipeline Manifest files 49
Orchestration Scripts 49
Steps interface 51
Runtime properties 51
Plugins 52
5.3.4 Library Versioning 52
5.3.5 Pipeline Configuration 53
Library Config, Organization Config, and Application Config 53
Configurations file internal structure 54
5.3.6 Pipeline Variables 56
5.4 Deployment Target, Artifact Repository, and Monitoring Service 56
5.5 Considerations 56

6 Evaluation **57**
6.1 Evaluation Methodology 57
6.2 Implementation Overview 57
6.3 Projects for Comparison 58
6.3.1 Project A - Mulesoft on-prem Openshift deployed with Jenkins . . . 58
6.3.2 Project B - Mulesoft in Azure Kubernetes Service deployed with Azure DevOps 59

6.3.3 Project C - Mulesoft in Anypoint CloudHub deployed with GitHub Actions . . . 59
6.3.4 Project D - Tibco on-prem docker deployed with Jenkins . . . 59
6.3.5 Comparison with the proposed solution . . . 59
6.4 Performance Comparison . . . 61
6.4.1 Testing a new hipotesys . . . 62
6.5 Team Pulse . . . 63
6.5.1 Result analysis . . . 64
What is your career level at Deloitte? . . . 64
Did you participate in more than one project since you work at Deloitte? 64
Did you build any DevOps process in any project you participated in since you joined Deloitte? . . . 65
Did any of your projects at Deloitte have any migration such as infrastructure, git server, pipeline orchestrator, or team restructuring during the period you participated in that project? . . . 65
Do you think the teams and/or the customer would benefit from implementing this product in the project environment? . . 66
What do you think will be the benefits for your work if this product was available in more projects at this moment? . . . 66

7 Conclusions **67**
7.1 Project goals . . . 68
7.2 Future work . . . 68

Bibliography **69**

A Orchestration script for the Step "Update Version" **73**

B Configuration Yaml **75**

List of Figures

2.1 Conceptual Map about DevOps impact on peoples collaboration in the team [31] . . . 14
2.2 Conceptual Map about DevOps impact on development process [31] 15
2.3 Conceptual Map about DevOps delivery related impact[31] 16
2.4 Conceptual Map about DevOps delivery related impact [31] 17
2.5 Delivery and Continuous Deployment [42] 18
2.6 Delivery and Continuous Deployment [44] 20
2.7 Git flow [46] 21
2.8 Git flow extended [46] 21
2.9 Metamodel to describe CI/CD pipelines [49] 22
2.10 Pipeline concept described in the book [8] 24
2.11 Monitoring a CI/CD Workflow Using Process Mining [51] 24

3.1 Value Map 28
3.2 QFD - House of Quality 30

4.1 Project Scope 31
4.2 Git workflow for proposed solution A 33
4.3 CI/CD for proposed solution A 33
4.4 Git workflow for proposed solution B 34
4.5 CI/CD for proposed solution B 35

5.1 Component Diagram - Level 1 39
5.2 Component Diagram - Level 2 40
5.3 Component Diagram - Level 3 40
5.4 Pipeline Orchestrator Sequence Diagram 41
5.5 Git Workflow 43
5.6 Pipeline Process 45
5.7 Pipeline Orchestrator Sequence Diagram VS File Structure 48
5.8 Framework Versioning System 53

6.1 Graph distribution for question 1 64
6.2 Graph distribution for question 2 64
6.3 Graph distribution for question 3 65
6.4 Graph distribution for question 4 65
6.5 Graph distribution for question 5 66
6.6 List of most written words in question 6 66

List of Tables

2.1 Search results 10

4.1 Comparison summary between solutions A and B 36

5.1 CI/CD Pipeline Process Jobs and Steps description 46
5.2 List of Pipeline Variables 56

6.1 Comparison summary between presented projects and the solution built in this thesis 58
6.2 Comparison summary between presented projects and the solution built in this thesis 60
6.3 Comparison between presented projects and the solution built 61
6.4 Comparison between presented projects, the solution built, and Python . . 62
6.5 Response Distribution to question 1 64
6.6 Response Distribution to question 2 64
6.7 Response Distribution to question 3 65
6.8 Response Distribution to question 4 65
6.9 Response Distribution to question 5 66

List of Source Code

4.1 Example of git file structuring for MulesSoft project . . . 32
5.1 GIT repository file structure for Mulessoft project . . . 42
5.2 Shared Library file structure . . . 47
5.3 Gitlab's client file source code . . . 49
5.4 Shell command to trigger the Orchestration Scripts . . . 49
5.5 Definition of Step ID and Name . . . 50
5.6 Capturing values from shell arguments . . . 50
5.7 Impoorting configurations . . . 50
5.8 Inject runtime properties . . . 50
5.9 Instanciation of Step Class and plugins . . . 51
5.10 Modules Selection . . . 55
5.11 CI configuration . . . 55
5.12 Plugins configuration . . . 55
5.13 GIT configuration . . . 55
5.14 Maven configuration . . . 55
6.1 Python script to build the package with unit tests . . . 62
6.2 Python script to build the package without unit tests . . . 62
A.1 Groovy script for Update Version task in the Build step . . . 73
B.1 Example of Configuration Yaml . . . 75

List of Abbreviations

API	**A**pplication **P**rogramming **I**nterface
CI/CD	**C**ontinuous **I**ntegration and **C**ontinuous **D**elivery
DB	**D**ata**b**ases
GUI	**G**raphical **U**ser **I**nterface
MVP	**M**Minimum **V**iable **P**roduct
VM	**V**irtual **M**achines

Chapter 1

Introduction

This chapter will be presented a context regarding the project being developed, describing the problem identified and how that affects the team, what solution is proposed, and how that will be implemented to achieve the goal of solving the problem.

1.1 Context

Deloitte is a major company in the Audit and Consulting industries presented in many geographies around the globe and considered a reference in its field of action. Deloitte is divided into many smaller teams dedicated to specific business needs. One such team is Integration Services, which specializes in providing consulting services to build and maintain integration layers for companies in various segments of society. An integration layer is a software layer made to allow information exchange between different applications deployed for a company without relying on specific logic built into each application centralizing the communication between systems and potentially reusing assets that were made before. To accomplish this data sharing, integration platforms and frameworks like MuleSoft[1], Tibco[2], WSO2[3], Apache Camel[4], and many others are available in the market and each count with their pros and cons. Each tool has its characteristics and, consulting companies need to master all of those technologies to expand their offer. Combining this with all the other usual concerns about software development, like code versioning, continuous integration, continuous delivery (CI/CD), documentation, etc. where the market offers a very expressive amount of alternatives forces consultants to keep a large number of technologies behind their radar to be pair with the market. On top of all of this, any process relevant to the company to ensure functionality, quality, security, monitoring, or other increases the complexity even if being automated because the users need to know the process to fix problems when they show up.

Any new approach or methodology that allows these consultants to have a more standard way of working between all of those technologies. It is usually well-seen by professionals that can spend more time building solutions for their clients instead of thinking about how each tool works alone or together with the others.

1.2 Problem

As clarified before, Integration Services professionals need to deal with technologies for various tasks and there is more the one tool for each category considered. Associated with each technology comes a process that tends to differ between implementations due to a lack of standard procedure or at least due to the differences in capabilities of each technology.

The Continuous Integration and Continuous Delivery (CI/CD) is responsible for every step that happens between the moment code is committed to the source-code versioning system and the moment it reaches production. The basic steps of this process are the code being checked in from the source-code versioning system, which usually is a GIT[5] server, the application is built, the automated tests are executed, and the application is deployed to the target environment.

It is intended for these steps to be executed automated to ensure that each build occurs the same way as before, in a predictable way to avoid human error on repetitive tasks. Although it is widely accepted that CI/CD should be an automated process, there isn't much guidance on how the process should look like, what steps should or should not be implemented globally as it is highly dependent on what is being built, and what are the needs.

Even for similar types of projects, inside the same organization, isn't easy to find standards and the most common route is an implementation from scratch of a solution that fits the needs of that project. Some companies have programs to uniformize their CI/CDs, but that is a common challenge with the different technologies and frameworks being used. Non-compatible project file structure, non-compatible GIT repository branching strategy, and many other scenarios make this a highly complex change process. So most companies decide to not implement in the beginning afraid of not getting the return on the investment and when the process is in place it isn't worth the investment of changing all that was built on that base.

For companies with in-house development of products for their own needs, this might not be an issue. Collaborators might not change departments as often which causes issues with a different process than the one that it used to. Although in a consulting firm like Deloitte consultant's rotation happens on average every six months, and having to learn a new process. Even if similar but it will always have its specific nuances which take time to learn. Having a way to minimize this learning process required each time someone joins a project would have a great impact on the consultants' performance and productivity during the onboarding.

Not only this but in cases where the customer goes through a migration that for some reason affects the CI/CD process, migrating the infrastructure where the code is running for instance, usually the solution is to build a brand new process. In the case who is making the new platform decided to change anything which could end up with major changes in the way working people are used to, like branching names, branching strategy, project structure, and other changes that impact the day-to-day work of everyone.

1.3 Project Vision

To handle the problem mentioned before, there were some initiatives to build standalone assets that could be reused. This works for the specific stack of technologies being handled but leads to re-engineering of the process.

Here it is proposed to create a Standard DevOps Framework that works based on the practice and experience of the Integration Services team, addressing all the relevant features required to ensure functionality, quality, security, and monitoring for the process of deploying code.

The Framework, as it will be referred to in this document, will be a combination of a set of rules and a process template that will be instantiated for every project. Combining these two components, teams will be able to share knowledge and assets helping each other in a collaborative way, as intended in the DevOps philosophy.

The Framework needs to be agnostic to the technology behind it, no matter if it is the GIT service, pipeline orchestrator, test framework, or any other variables. This will assist consultants while moving between projects. Additionally, when the customer wants to make core changes in their environment, like changing the platform where applications are deployed, having a Framework like this ready to be implemented with reduced effort, by having the developments or at least the template that only needs to be applied. This is relevant for both Deloitte and the customer.

The concept for the Framework involves more than only CI/CD because CI/CD is influenced by many other variables like GIT repository structure and branching model, pipeline orchestrator, deployment target, artifact repository, the technology used to build the application, and the type of quality and monitoring that is intended to be made. The dependencies mentioned can be summarised in four categories that together define the constraints for the implementation:

- Source-code management server: There are various types of source-code management servers used around the globe, but GIT is an industry standard and the framework will consider GIT as the only option for a source-code management server. This means that any service available that supports standard GIT functions should be fully compatible with the solution being built.
- Repositories: Source-code repositories should follow a standard structure for both files and branching. Having a consistent way to manage the repositories is crucial to the success of the project as it is one of the pieces that developers have the most contact with
- Deployment pipeline: All pipelines should follow the same structure, with similar names in tasks that are equivalent to bring the feeling of familiarity even when the instructions executed behind are different. Pipelines should also perform a similar action in the related component like storing artifacts in repositories in the same way.
- General tools: There are multiple tools and services used in the deployment process. The goal here is not to enforce what tools should be used but rather define what should be actions each tool should be able to perform to be compliant with this framework.

To achieve the goal described before, the Framework will focus on defining what should be the rules to be implemented keeping it agnostic of technologies used instead of focusing on the actual implementation. When the need comes for a new stack of technologies, the framework can then be implemented with that specific stack in mind, but following the

standard process with the benefit of not needing to design whole the process from scratch. All implementations will then later be saved in a GIT repository and versioned to be used whenever it is needed. The Framework itself will be versioned as it is expected that future tools or techniques bring the need to improve the framework. This way will be possible to keep track of the changes made to the framework and what needs to be done to a previous implementation to make it compliant with the newer version.

1.3.1 Project Goal

Building an entire DevOps Framework is a complex task, considering all the variables, processes, and nuances involved in it, especially in a short period of time with limited resources. It requires intensive research and work from the team building it. This isn't practical in the context of this thesis, so the decision to build a Minimum Viable Product (MVP). On the MVP the focus will be concentrated on building a simple version of the product to validate the concept and gather feedback from stakeholders. The feedback will be then used to drive the future developments of the project.

This MVP is divided into two separate parts, the Framework prototype with all the rules and conditions that need to be followed and a Proof of Concept (POC). The POC is an actual Framework's implementation for a specific set of technologies.

1.3.2 Expected final result

As a final result, is expected to have a Proof of Concept of the framework, with the definition of some tooling, processes, and rules. It will not be a complete product, but something that can be used to validate the project before investing more resources in it. It should be compliant with the requirements that will be defined in section 4.4.

This POC should demonstrate how this Framework can in the future help minimize the challenges mentioned before. Demonstrating how consultants change between project inefficiencies Can be mitigated. Also how the Framework helps while defining the DevOps process when starting new projects. And finally, identify the impact of the Framework in the circumstance of platform migrations in the customer environment.

1.4 Project Restrictions

Projects, in special the ones built in an enterprise environment, have specific requirements and restrictions that need to be met in order to be considered a successful implementation and compliant with the policies in place. Some restrictions will need to be taken into consideration during the project.

The first restriction is the scope of projects. The Integration Services team only works on projects related to system integration for enterprise environments and those are the only ones considered. The focus will be applications that implement some sort of Application Programming Interface (API), excluding at least for this initial version, any sort of infrastructure components like Virtual Machines (VM), Databases (DB), or even web applications with Graphical-User-Interface (GUI), mobile or desktop applications.

The second is the technologies supported. During the development of this project, only technologies that are part of the current practice of Deloitte Integration Service and match Deloitte's current strategic vision will be considered for the project.

The third restriction is related to the Proof of Concept (POC) that will be later in this document. The POC will be an actual implementation of the framework with a specific stack of technologies. The technology stack was defined by Deloitte according to the current needs and strategy.

The fourth restriction refers to the resources to develop the project. All resources, including computers, software licenses, and online accounts, will be provided by Deloitte and are the only ones authorized to work with. In case of needing anything else, it should be requested and validated by Deloitte before being used and in case of no authorization, an alternative to that resource should be found.

The fifth restriction is related to the environment where the software will run. All software must be compatible with Linux-based operating systems. Linux-based operating systems are the standard choice for companies to run their integration applications and their pipeline orchestrators. No effort will be made to create a compatible implementation for other operating systems such as Windows, MacOS, or BSD based.

Considering the restrictions already mentioned, it was proposed by Deloitte a workflow of recurrent meetings during the development of the Framework and POC. During those meetings, the developments made since the previous meeting will be reviewed and some feedback will be provided to better adjust the solution to Deloitte's needs. This feedback will be mostly related to high and mid-level design. Nevertheless, there is autonomy to make relevant decisions during the whole project.

1.5 Approach to the Problem

The project referred to is very related to the needs of Deloitte itself. Finding information about exactly this topic isn't trivial so some steps will be followed to pursue the solution to the problem.

First will be research about what was being done globally about the DevOps process. Not specifically related to integration projects, but broader research on the topic. Understanding the needs of the ones who already published their work and comparing those needs with Deloitte ones to understand what does and doesn't make sense in this scenario. preparing an initial plan with the respective value proposition and checking if that fits into the needs and expectations of Deloitte. This will be the State of the Art.

Having a plan defined, and based on the research made the framework will be built in an iterative way, meaning that will exist iterations where the product will be developed progressively. The recurrent meetings mentioned in 1.4 will be used to evaluate the progress made and define the goals for the new iterations.

After considering a framework ready, the POC will start. POC has two goals, first prove that the framework is functional but also to catch some minor details that may prejudice the productivity of the final solution developed.

After this, the results of the work will be presented to all Integration Services team, explaining the problem found, what was the goal, and the solution that end-up being produced. Then requesting everyone to fill out a form about their opinion of the project. Combining this with historical information of other DevOps implementations comparing available features and time to implement to finally evaluate the project.

1.6 Code of Ethics

This project was developed in total compliance with Deloitte's internal regulamentations. All the information provided in this document is approved by Deloitte and all the resources used to develop this project, including laptops, software licenses, and accounts for online services were provided by the firm with the purpose of developing this project.

1.7 Thesis Structure

The document is divided into six chapters, each one dedicated to a specific part of the project.

- The Chapter 1 is made an introduction to the project.
- In Chapter 2 will be presented the research made about other people's work on the topics mentioned here.
- in Chapter 3 will be presented the value analysis and value proposition.
- In Chapter 4 will be described the framework itself, its rules, assumptions, and all the other relevant information to fully understand the product.
- In Chapter 5 will be presented the POC and its final result, what was implemented, and how easy to implement was based on the framework presented.

- Chapter 6 refers to the results, the comparison with historical data of other projects implemented to address the need of having a DevOps practice, and the feedback from the team of Integration Services.
- Chapter 7 will contain the conclusions about the project, what were the bigger challenges, what are the limitations of the current version of the framework, and what are the next steps to continue evolving the product.

Chapter 2

State of the Art

In this chapter will be presented the research made about other people's work on the topics mentioned here and how that research was made. it is followed by a presentation of the literature revision about the structure and flows for GIT repositories, CI/CD pipeline structuring, and CI/CD pipeline monitoring and alarming Then it is finalized with analysis and proposition for this work.

2.1 Research Questions

The research followed the systematic mapping study to get a broad understanding of the DevOps methodologies that are being used. But due to the comprehensiveness of this topic, the question had to be broken down into multiple sub-questions that help clarify the overall culture of DevOps.

- Q1 *How is DevOps defined in the literature?* The intention here is to have a baseline of the definition of DevOps that can be worked on later.
- Q2 *What GIT workflows are used to manage the different stages of development?* The goal here is to understand how developers manage the different branches in their git repositories to work collaboratively.
- Q3 *How are CI/CD pipelines defined?* The purpose here is to understand how pipelines are being built to deliver applications. What are the steps that usually appear in the CI/CD pipelines?
- Q4 *What CI/CD agnostic processes were already implemented?* The goal here is to check if there is already a process that implements an agnostic approach regarding multiple implementation technologies.
- Q5 *How does CI/CD pipeline are monitored?* The goal here is to comprehend the strategies used to extract information from the CI/CD pipeline for reporting and automation purposes.

2.2 Scientific Research

2.2.1 Selection Criteria

To find guidance on what results should or should not be considered, some selection criteria were defined. The criteria are:

- SC1: Should be available on the open web or through ISEP VPN.
- SC2: Publishing date is from 2017 or newer.
- SC3: Publishing language is English or Portuguese.
- SC4: In top 10 results when ordered by relevance using the option in the search engine.
- SC5: Content relevant after reading the abstract.

Besides this criteria is also possible to include resources from reference snowballing [1].

2.2.2 Research Results

After choosing the questions that are relevant to the research, and defining the selection criteria the study was made and the resume is in table 2.1

Table 2.1: Search results

Keywords	**No. of Results**
DevOps	19 300
CI/CD	8 530
Meta Models for DevOps	6 950
DevOps Framework	17 000
Project Agnostic DevOps Framework	2 250
Technology Agnostic DevOps Framework	2 430
GIT Branching Strategy	13 300
Git workflows	23 800
CI/CD Pipeline structure	4 550
CI CD Pipeline structure	18 200
DevOps generic pipeline	2 700
CI/CD Pipeline for enterprise microservices	1 690
Delivery Pipelines DevOps Practices	6 090
Delivery Pipeline Enterprise Architecture Model Evolution	17 200
CI/CD Pipeline for enterprise monitoring and alarming	770
DevOps Best practices	13 000
Selected	09
Reference Snowballing	09
Total	18

[1] A reference snowballing happens when a resource is used and referenced as part of the research even if it respected the search criteria defined above but it is in the references of one or more resources accepted by the same criteria.

From the research made 21 resources including the books Pro Git [6], Microservices for the Enterprise [7] and Generic Pipelines Using Docker [8] were studied to sustain the work that will be presented in the following sections and chapters.

2.3 Technology Research

To complement the scientific research, more research was made on topics that are relevant for a comprehensive understanding of this state-of-the-art and upcoming work. It also introduced an overview of Michael Cade's work who is running the second edition of the project 90 Days of DevOps [9]

The research subjects for this project were divided into groups based on how they relate to one another:

2.3.1 Term research

Cloud Platforms

Microsoft Azure Azure is Microsoft's offering in cloud computing. It is a set of more than 200 products and services [10]. Azure was publicly announced on October 2008 [11]. Originally built on top of the Windows NT base, to directly compete with Amazon EC2 and Google App Engine [11].

Amazon Web Services (AWS) AWS is Amazon's cloud offering suit. It widely used cloud in the world and provides over 200 fully functional services from data centers across the world. [12, 13]

CI Platforms

Jenkins Jenkins is a standalone, open-source automation server that may be used to automate a variety of software development, testing, and delivery operations. Jenkins can be set up using native system packages, Docker, or even operate independently from other software on any machine that has Java Runtime Environment (JRE) preinstalled. [14]

GitHub GitHub is one of the most famous Git hosting service in the world.[15] It was firstly available in 2008 and recently acquired by Microsoft.[16] GitHub also supports the execution of CI activities with their offering of GitHub Actions [17]. GitHub actions also support storing various types of packages in the standard Artifact Repository for various technologies.

GitLab Hitlab is an alternative Git Hosting service[18], well known in the industry for the integrated capacities aside from Git Hosting such as Gitlab CI for Continuous integration.[19]

Azure DevOps Azure DevOps is Microsoft's solution regarding all DevOps teams' needs. Its offering includes Azure Boards for team agile planning; Azure Pipelines to build, test and deploy software in automated processes; Azure Repos for source code hosting in GIT repositories; Azure Test Plans for test management of applications and Azure Artifacts to host artifacts built during the automated pipelines [20].

Integration Platforms

Tibco Tibco is a solution suite for integration designed for business, web, and mobile applications. It uses a visual, model-driven development environment, the program enables you to construct services and combine applications, which can subsequently be deployed in the ActiveMatrix BusinessWorksTM runtime environment. Business processes are defined using the Eclipse graphical user interface (GUI) offered by TIBCO Business StudioTM for BusinessWorksTM, which creates deployable artifacts in the form of archive files. The deployable artifacts can be controlled using an administrative interface, such as TIBCOTM Enterprise Administrator, and deployed and executed in the runtime environment.[21].

Mulesoft Anypoint Mulesoft Anypoint is a set of tools provided by Mulesoft used in the enterprise environment for integration. It is composed various of tools like Anypoint API Designer used to define application interfaces API; Anypoint Studio to develop the applications; Anypoint Exchange to publish and find reusable assets; Anypoint MQ as a dedicated message broker; Anypoint Cloudhub a PaaS service to run Mule Application and API Manager to manage interactions between the APIs in the ecosystem like authentication, service level agreements (SLA) and much more [22].

Azure Functions Azure Functions is a service available in Azure to run custom code in Azure abstracting the need to define any infrastructure to rely, on a model called "serverless". It is compatible with applications written in C#, Java, Python, and many others [23].

Containerization Platforms

Docker Docker is a platform for building, delivering, and executing software. Docker separates the applications from infrastructure, allowing for rapid software delivery. This is achieved with containerization technology that enables the abstraction between the application and the operating system. The application is bundled with all its dependencies in a structure called an image. Images are deployable artifacts that will ensure the application will behave the same way no matter where the application is running [24].

Kubernets Kubernetes is an open-source platform for managing, scaling, and automating the deployment of containerized applications [25]. Kubernetes will manage the life cycle of a container, ensuring that all applications are running in a healthy state according to the configuration defined, rebooting the application in cases of crashes, or even creating more instances of the application in case more demand workloads hit the platform. Kubernetes abstracts traditional infrastructure topics such as networking, load-balancing, and storage for a cohesive and integrated way of working.

OpenShift OpenShift is a platform built by Red Hat[2] [26] on top of Kubernetes with extra features for Developers and Operators. It provides automated upgrades, life cycle management, and installation across the container stack.[27].

Azure Kubernetes Service Azure Kubernetes Services is a managed Kubernetes service provided by Microsoft in Azure [28].

[2] https://www.redhat.com

2.3.2 90 Days of DevOps

90 Days of DevOps[9] is a project from Michael Cade where he describes his journey in the DevOps world for 90 days covering his research, findings, and conclusions. The first iteration of this project happened at the beginning of 2022 and the topics covered were CI/CD, Azure, Automate Configuration Management, Containers, Go programing language, Data storage, Infrastructure as Code, Kubernetes, Linux, Monitoring, and Networking. The second iteration is currently being done with the collaboration of some other members of the community and already addressed DevSecOps, more of CI/CD, and secure coding. Are also planning to touch on topics like Secret Management, AWS, Openshift, Databasesm Serverless, and more.

Michael Cade is the Global Field CTO at Veeam Software and has had a career in IT since 2003. He is being working at Veeam since 2016 [29].

2.4 Research Answers

The results of processing the research made will allow now to have some responses to the questions made before.

2.4.1 How is DevOps defined in the literature?

In the literature, there is not a closed version of what DevOps is or is not [30–32], but all the studies refer to similar points that can be joined together to get an overall definition we can work with. The origin of the DevOps term is related to the Developers and operators and their functions in a project or product team and a relation both teams have with each other[30, 33–35]. There are then some dimensions where it is possible to define how exactly DevOps works, things as collaboration, software development, software delivery, continuous integration, and quality assurance.

Collaboration

DevOps enables efficient teamwork between development and operations teams based on communication [30] bringing people closer. DevOps aims to dismantle silo walls and align incentives throughout the company[31, 36], making people feel they work with a bigger group and that they work as a real impact on other people's life.

Nevertheless, the elimination of silos presents various challenges. How can organizations effectively introduce a cultural shift that assigns developers new responsibilities? How can developers acquire operational expertise? Determining the boundary between developer tasks and operator tasks becomes a pertinent question. Should operators be integrated into cross-functional teams? Should an operator focus solely on one team at a time? What does the term "operator" signify within the context of DevOps?

These inquiries, along with many others, will undoubtedly arise for individuals operating within these teams as they navigate the transition to the DevOps methodology and beyond. The adaptation process will be dynamic and involve continuous adjustments by the team to accommodate the new circumstances.

In Fig. 2.1 is a conceptual map that resumes what DevOps impacts teamwork and collaboration between all team members.

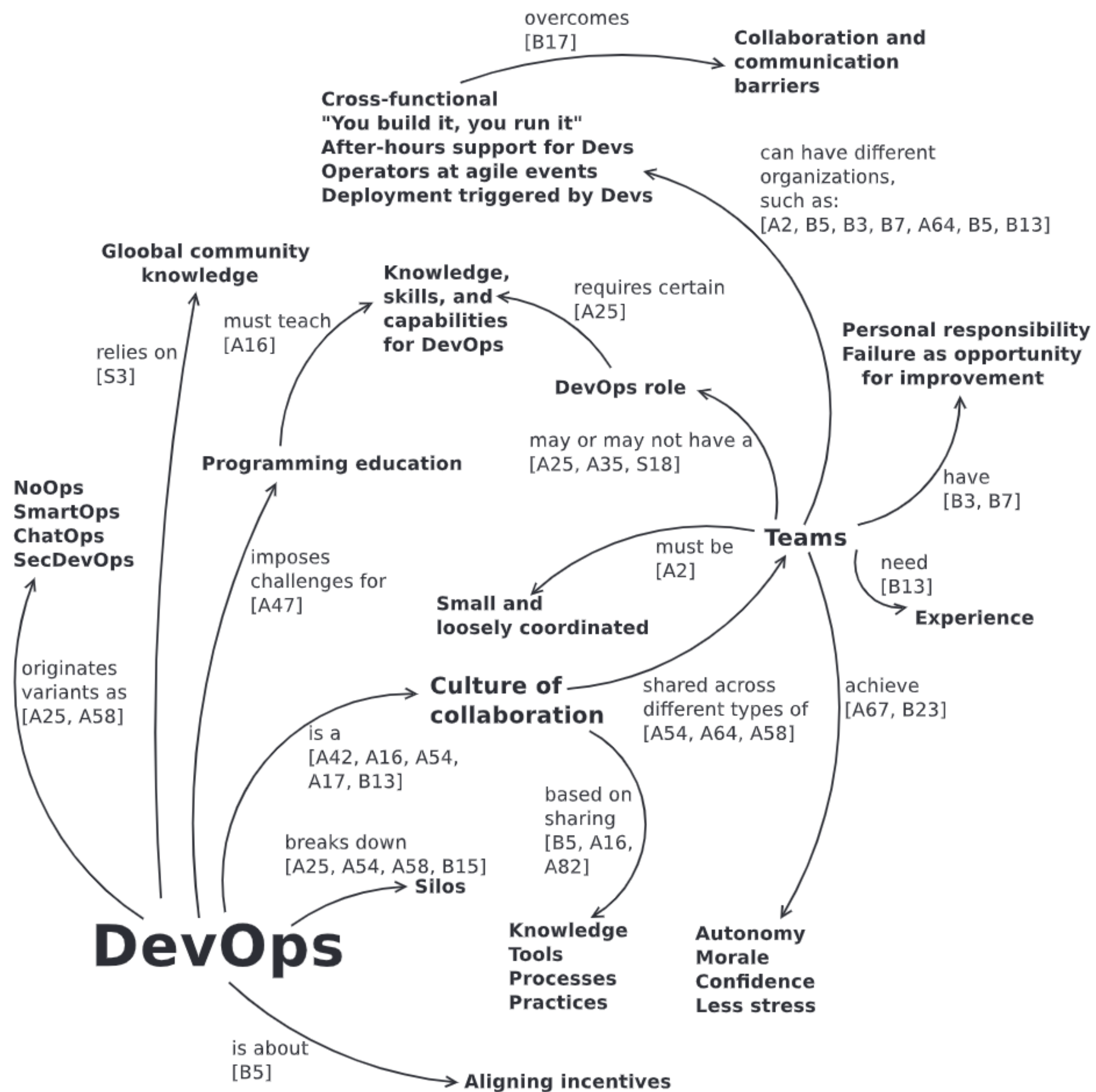

Figure 2.1: Conceptual Map about DevOps impact on peoples collaboration in the team [31]

Software Development Process

DevOps represents a contemporary approach to software development that seeks to resolve the interdependencies between development and operations. By incorporating cutting-edge techniques and tools, it fosters a true integration of developers and operators, enabling them to work together seamlessly[30, 32, 37]. This comes to one final goal: reduce risk and cost with development, while continuously improving product quality and customer satisfaction [31].

All of this contrasts with the traditional way of thinking about reducing the risk and improving quality by having a hierarchical approval process [31] that was many times proven inefficient and slower. Instead of that, the DevOps philosophy embraces change in smaller portions and repeatedly doing so achieves a product much more in line with what is expected by stakeholders and users.

In Fig. 2.2 these concepts are described

Figure 2.2: Conceptual Map about DevOps impact on development process [31]

This aspect holds significance as several authors argue that DevOps can be seen as an extension of agile principles, emphasizing the importance of collaboration and effective communication among team members rather than imposing complex procedures that must be followed by everyone [38, 39]. SSome authors have also pointed out the potential of Agile methodologies to facilitate DevOps thinking [30]. Conversely, project management methodologies like waterfall lack the same level of integration between development and operations as DevOps. Waterfall traditionally follows a sequential approach, dividing projects into distinct phases that must be executed in a predetermined order. This fundamental disparity indicates that waterfall methodologies are fundamentally incompatible with the principles of DevOps.[40]

Software Delivery

DevOps is a model or collection of concepts that focus on delivering software by allowing continuous feedback, responding quickly to changes, and leveraging automated delivery pipelines to shorten cycle times [30]. To achieve continuous delivery, DevOps proposes the concept of a *Deployment Pipline* [31], an automated way to deliver software produced, making it reputable and free from human intervention to avoid error related to it.

In Fig. 2.3 these concepts are described

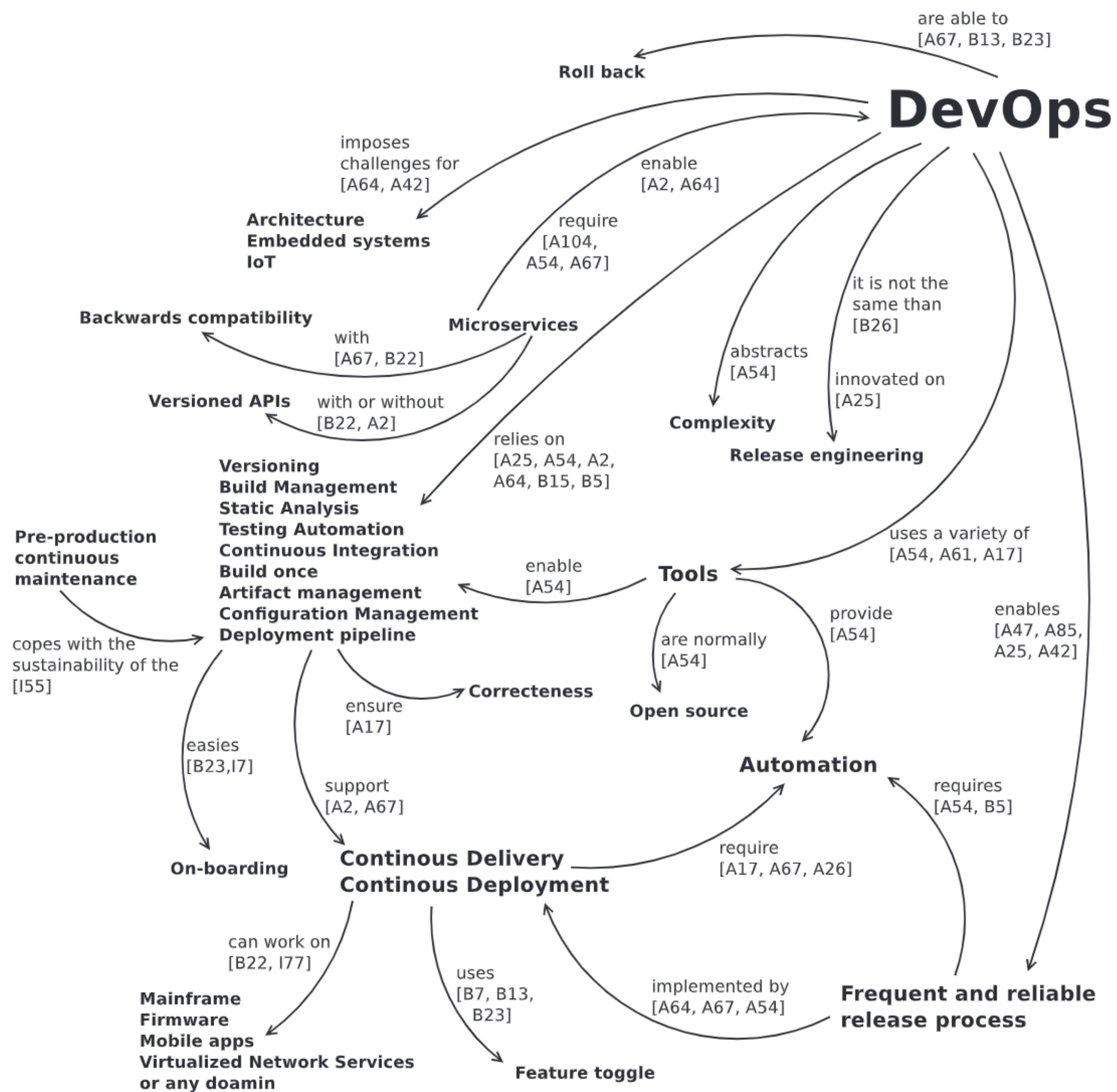

Figure 2.3: Conceptual Map about DevOps delivery related impact[31]

But delivering constant software means nothing if this is not done sustainably. In a way that can ensure things like resilience, reliability, availability performance, and scalability [31]. For this quality, assurance actions are taken all over the DevOps process with multiple and variated runs of tests to ensure everything will work as expected. It also leveraged more flexible ways to deploy the software like virtualization and containerization helped with cloud computing that has the ability and the resources to invite more often than on-premise solutions usually can.

Fig. 2.4 shows the association of DevOps with all of those concepts.

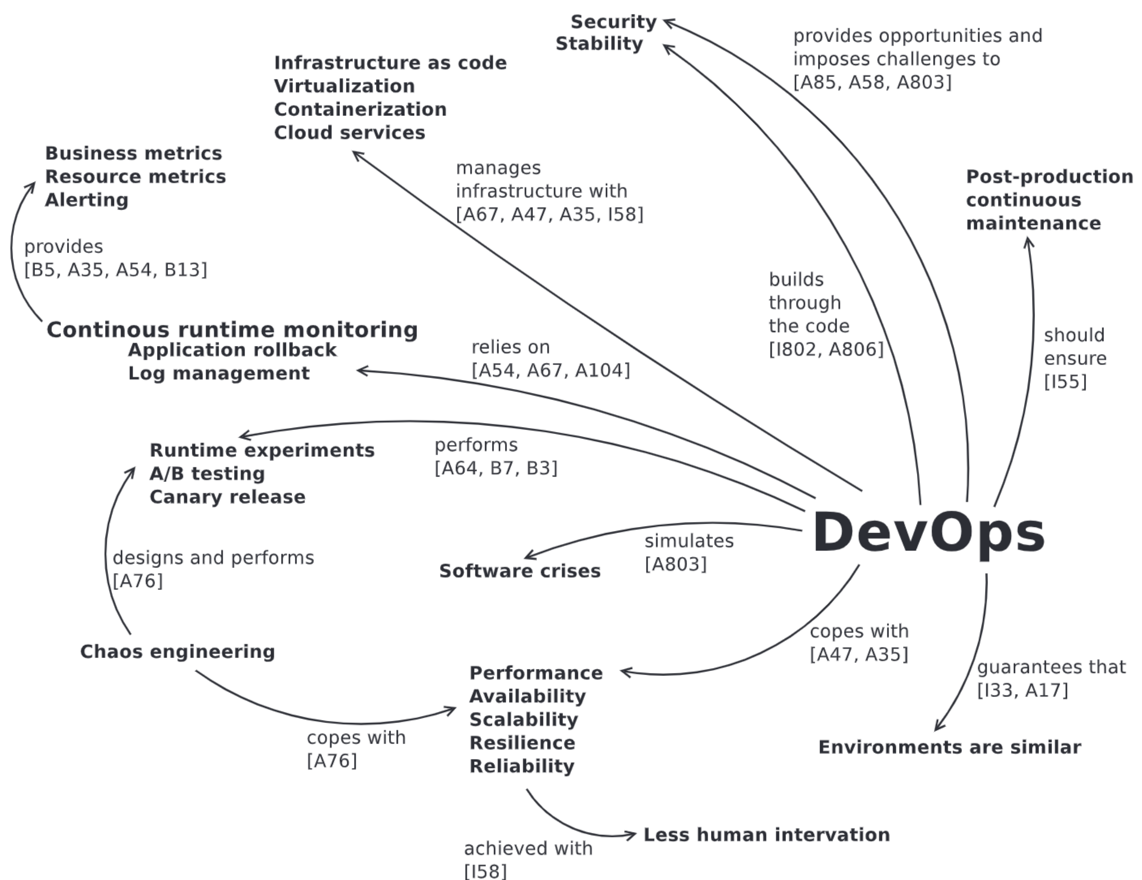

Figure 2.4: Conceptual Map about DevOps delivery related impact [31]

The emphasis on automation has brought about advancements in various related areas, including the configuration of software utilized within the infrastructure and other components surrounding the development of applications. This is the concept of GitOps [41], where configurations are defined in configuration files and stored in Git repositories. These configurations are then utilized in automated processes for provisioning and managing the entire platform. By combining these Git-based configurations with virtualization and/or containerization, it becomes possible to automate the provisioning of infrastructure. This not only facilitates running applications in a stable and controlled environment but also enables the creation of temporary environments for thorough code testing with minimal human intervention. Such automated testing processes ensure more comprehensive, accurate, and repeatable testing, minimizing the occurrence of regression issues and ensuring high quality.

In the end, all this focus on automation and removing the human interface will bring a more predictable and less human-error sceptical process, improving the quality of the software involved [30, 41].

CI/CD

Another topic tightly related to DevOps is the CI/CD.

CI stands for Continuous Integration and is a concept applied to the software development teams when it comes to merging their code changes to the main branch in Git. These changes should be merged regularly [42]. The benefit lies in the ease of integrating smaller

changes into the main branch, as opposed to waiting for a larger code volume that would then require extensive reevaluation after merging numerous changes. As the number of merged changes increases, it becomes increasingly challenging to remember all the decisions made and to accurately assess whether any changes require manual inspection (such as in the case of conflicts). This approach also enhances developers' awareness of others working within the same codebase, aiding in the prevention of conflicts even before they arise.

During the CI phase, some testing steps can already start being executed automatically, for example, unit tests [42]. For this, a CI server will need to be leveraged and these tests can be executed after each commit which will give the developer some information about the state of the code is done.

Moving to the CD part, two possibilities are mutually accepted that is Continuous Delivery and Continuous Deployment. Both share most of the concepts but differ in a precise moment, the deployment to production. For Continuous Deployment, the concept is that after a change is tested and passes all the tests it is ready to be live in production, while Continuous Delivery gives the teams the power of deciding when is the moment to deploy to production [42].

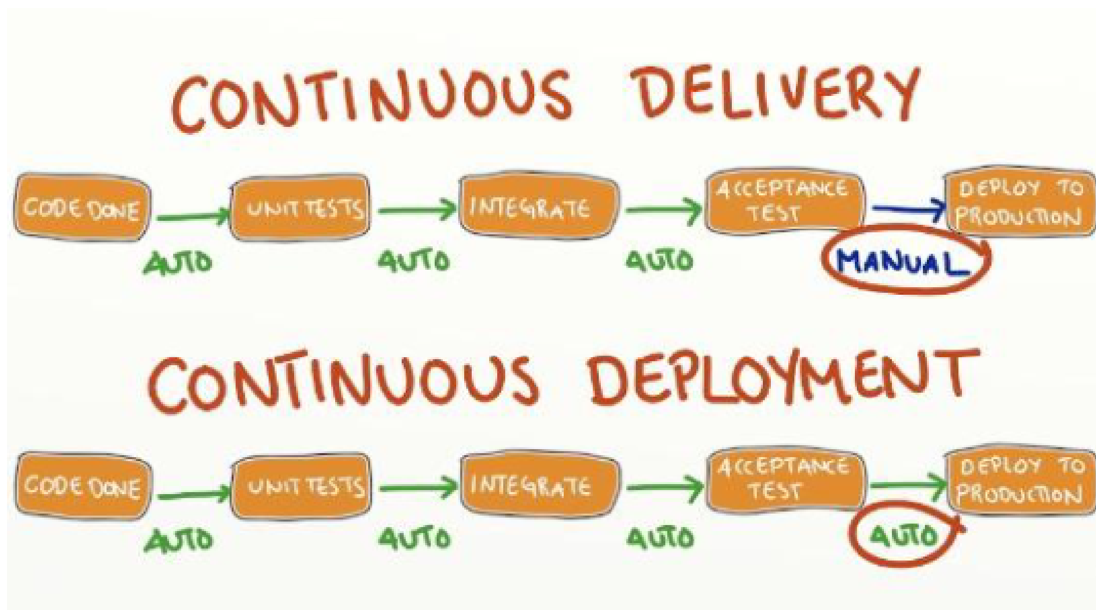

Figure 2.5: Delivery and Continuous Deployment [42]

Although it may seem like a minor distinction, this aspect can have a substantial impact on team dynamics. With Continuous Deployment, once a commit is pushed to the Git server, the developer relinquishes direct control over that code change. This can create apprehension and make it challenging to decide whether to push the code, as there is a fear of introducing new bugs. As a result, the implementation of a Continuous Deployment pipeline becomes crucial, as it requires substantial investment in methods to identify and catch potential issues. The goal is to ensure the production environment remains as stable as possible by proactively addressing and mitigating any potential problems [42]. Continuous Deployment also operates under the assumption that any dependencies, such as other applications requiring updates, will be deployed and live in the production environment before the changes are pushed to the Git server. However, in practice, it is often challenging to achieve this synchronization, and

there are situations where all teams must be prepared to release new software simultaneously. This highlights the need for effective coordination and collaboration among teams to ensure a smooth deployment process in such cases.

Deployment to production Nothing of what is mentioned until now really matters if the goal is reached, the deployment to the production environment. Deploying the code to production can be as simple as purely making the application live, although there are some relevant concepts regarding the code in production that are useful to be aware of:

- Blue/Green Deployment: The updated version will be installed on a completely separate server, no matter it is a physical machine, a virtual one, or even a container. Then is triggered a switch from the older one to the new one, allowing for a zero downtime deployment with the addition of being able to perform more tests in the newer version deployed [42].
- Canary releasing: The newest version of the application is partially deployed on a subset of the machines. Over a week, the update will gradually be made in more devices until all devices are running the latest version. The phased rollout can be stopped when issues with the update arise [43].
- Rollback and Roll-forward: When an issue arises in the production environment, it becomes crucial to take prompt action to restore stability. This is where the concepts of rollback and roll-forward come into play. In a rollback approach, the latest version of the software is replaced with the previous version, effectively undoing all the changes made and reverting the application back to a stable state. This provides the developer with the necessary time to investigate and address the issue, leading to the release of a new version with fixes. On the other hand, a roll-forward approach leverages the automated CI/CD pipeline to deploy a patch or fix directly onto the problematic release. This approach is particularly advantageous when there are database changes involved, as reverting such changes typically requires more effort. Instead of reverting the changes entirely, a roll-forward approach allows for the application of a patch to resolve the issue while still keeping the necessary database changes intact. This is especially beneficial when the original change will eventually need to be re-implemented after the patch is applied. The choice between rollback and roll-forward depends on the specific circumstances and the nature of the issue at hand, providing flexibility and options for resolving production incidents efficiently. [42]

DevOps Metamodel

In the article Towards an IT Governance of DevOps Metamodel, the authors proposed the metamodel of IT Governance of DevOps[44] shown of Fig2.6.

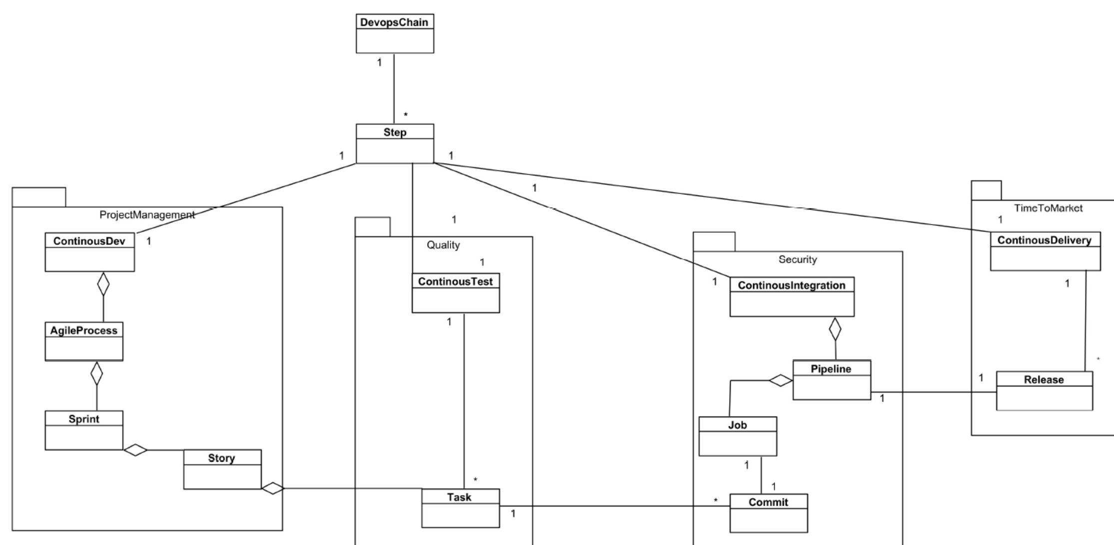

Figure 2.6: Delivery and Continuous Deployment [44]

This diagram helps identify some of the concepts explained before while showing graphically how those concepts relate with each other to create what is called today DevOps.

2.4.2 What GIT workflows are used to manage the different stages of development?

The git workflow will define the rules for the management of the source code repositories. One of the most common processes is Git Flow. In this process exists five types of branches: Master/Main, Develop, Feature, and Hotfix [45].

- Master / Main: is the default branch. In this branch, the source code is in the same state as the application in production. Will be referred to as a master branch for simplicity
- Develop: is the branch with the latest development version of the code
- Feature: Is the branch used by a developer who is developing a feature
- Hotfix: is the branch with the code changes to fix an issue with the production version.
- Release: a branch used to prepare the release.

Gitflow uses branches to represent the different versions of the code throughout the application lifecycle. From the branch *master* is created the branch *develop*. Developers will then create feature branches, usually with the name following the structure *feature/<feature descriptive>*, and make their code changes in it. When the change is ready, it is merged into develop branch. This process repeats itself until the release time. At release time, a release branch is created, from develop branch to perform the final tests. During the release, the code is merged into the master branch. *Hotfix* branches will be used to fix issues in production. When the change is ready to be deployed it is merged into the master to repair the production environment and into development to avoid regression [45, 46]. Fig. 2.7 is a graphical exemplification of this workflow.

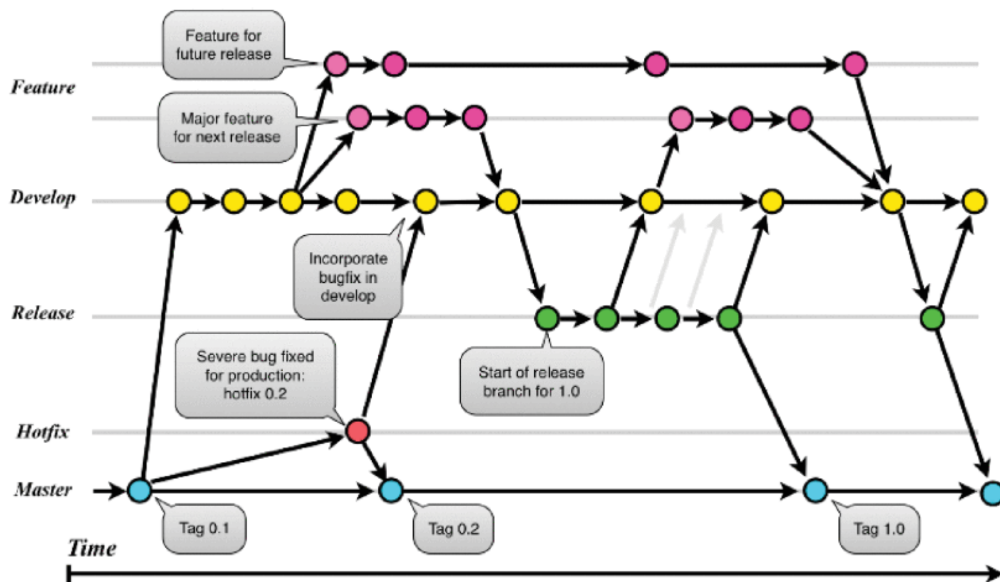

Figure 2.7: Git flow [46]

In the paper Cross-Site Large-Scale Software Delivery with Enhanced Git Branch Model [46], the author mentions an extended version of this workflow to be able to work with multiple teams in the same code base. Instead of a *Develop* branch, each team will receive a *Project* branch. Periodically teams will merge their changes in their project branch into a new branch named *Integration* and pull the changes from other teams also from this branch. *Integration* is then the equivalent of *Develop* branch in the previous mode. The remaining process will remain the same with the release and hotfix branches.

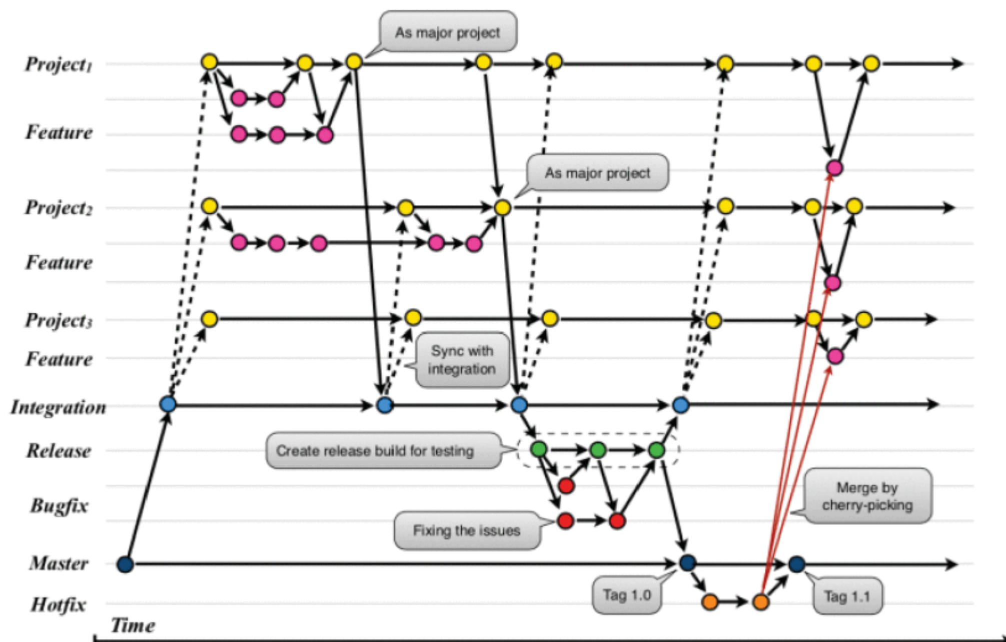

Figure 2.8: Git flow extended [46]

Although is a well-known and well-documented workflow, Git Flow has a complex view of

the process. For projects with good automation and really frequent releases to production, GitHub proposed the GitHub flow [47]. GitHub flow only considered master and feature branches. Changes are made in side branches (we will call them features, but there is not a specific name defined by GitHub). Tests will be made against this feature and when it is ready it is merged, with a pull request, to master to then be deployed to production [47, 48].

2.4.3 How are CI/CD pipelines defined?

To understand how a CI/CD pipeline is defined it is important to understand the key components that compose the pipeline itself. In the article Metamodel to describe CI/CD pipelines, the author mentions the meta-model illustrated in Fig 2.9. It is based on GitHub Actions architecture but generic enough to be compatible with other CI/CD platforms such as GitLab Pipelines [49].

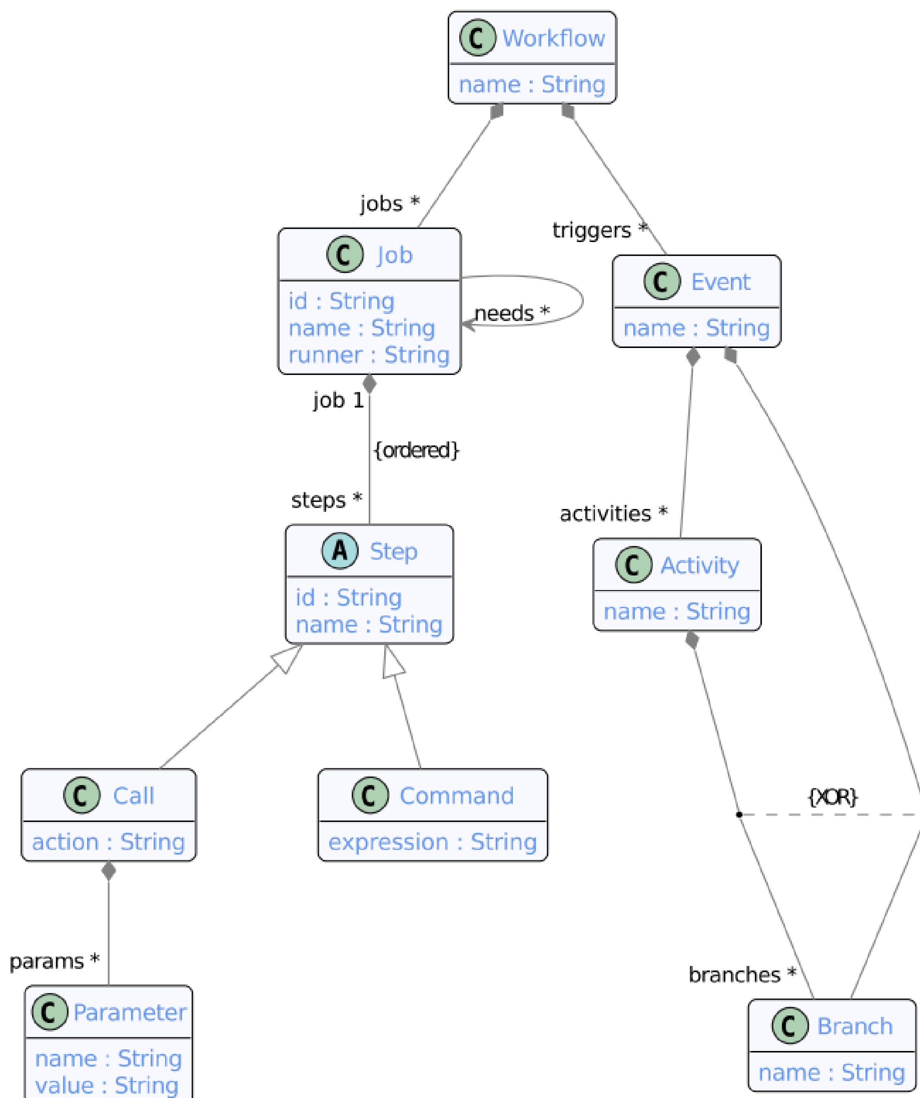

Figure 2.9: Metamodel to describe CI/CD pipelines [49]

This meta-model represents how each of the concepts involved in the CI/Cd pipeline relates to each other.

After understanding what are the key components of a CI/CD pipeline, it is possible to define the key activities related to it. In DevOps in practice: A multiple case study of five companies [50], the authors define five categories of activities identified during their study:

- **Development practices**: All new developments happen in dedicated branches of the source code repository and are frequently merged into the main line to avoid extra work fixing conflicts
- **Code review**: Reviewing code before merging into the main line. The main goals were to guarantee quality and provide a forum for discussing coding standards and styles.
- **Continuous Integration**: The CI server runs unit tests and created deployable packages for the system. In some cases, additional tests are performed to ensure the quality of what was developed.
- **Infraestructure as Code**: Scripts and templated used to provide the infrastructure required by the applications defined using specific tools and version controlled.
- **Automated Deployment**: Applications deployed automatically using a dedicated tool such as Jenkins or a custom script
- **Monitoring**: Monitoring systems status using health checks and dedicated platforms.

2.4.4 What CI/CD agnostic processes were already implemented?

There are some works related to getting an agnostic CI/CD process. A good example is described in An Advanced DevOps Environment for Microservice-based Applications [36]. The author referred to the use of templates in GitLab CI that then all repositories pipeline imported.

In the book, Generic Pipelines Using Docker [8] the author described the process of creating a single pipeline that is able to deploy applications written in .NET Core, Java, Node.js, Python, Golang, Angular, and React using shell scripts and docker. The pipeline steps were the same for all of them: Build, Unit Test, Static Code Scan/Security Scan, Packaging/Publishing of Artifacts, Deploying, End to End Tests, and Performance Tests.

Each technology has its own tooling and building a unique platform for all of them is really challenging. To achieve the desired outcome, they employed shell scripts containing the necessary commands for each step in the process, tailored specifically for each technology involved. Additionally, they utilized Docker containers with the required tooling to simplify the complex tasks within the CI server. This approach streamlined the execution of the processes, making them more manageable and less convoluted. By leveraging shell scripts and Docker containers, they were able to achieve the intended results efficiently and effectively.

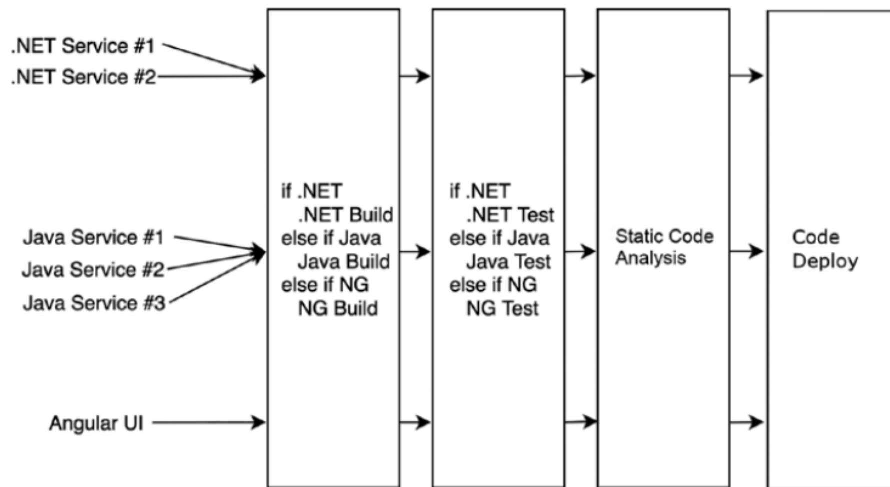

Figure 2.10: Pipeline concept described in the book [8]

2.4.5 How does CI/CD pipeline are monitored?

For CI/CD pipeline the research made offered a very little amount of results, but the project Monitoring a CI/CD Workflow Using Process Mining [51] described a really interesting way of not only monitoring but extracting useful information from the pipeline execution.

The project uses process mining to process information from the pipeline. Process mining is the technique of discovering information about one process by the logs and event logs it produces. There is intense research on techniques for automated process discovery but Event Log Noise, Errors, and Incompleteness raise some challenges to the discovery algorithms, some of which are being addressed by Genetic Minning, Fuzzy Mining, and Heuristic Mining. The author identifies Heuristic Minner as an adequate method to extract the data from CI/CD pipelines.

The strategy used is by having the CI/CD pipeline push messages to a Kafka cluster. Those messages are then picked by a consumer that will process them and save the result into an Elasticsearch server. With the data in the Elasticsearch server, dashboards are fed and users can extract information, not only based on the direct information obtained from the logs but joining the data getting even more information about the process itself and the overall process [51]. Information like time per step, total execution time, the mean value for the execution time, failure rate, failure rate per step, and much more information.

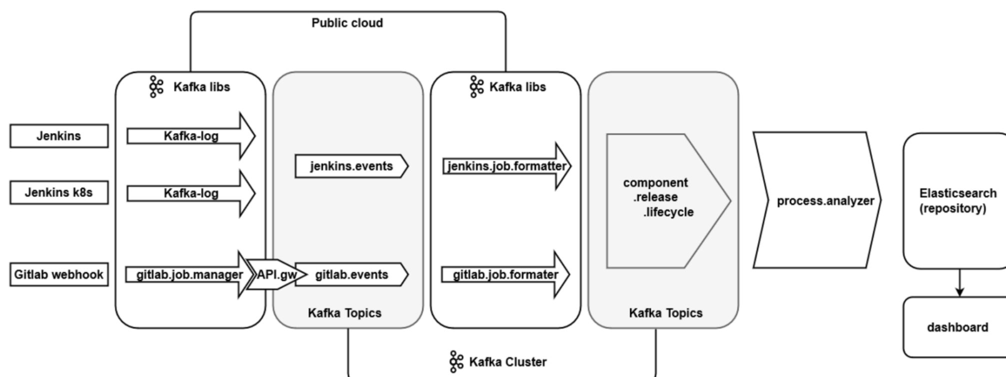

Figure 2.11: Monitoring a CI/CD Workflow Using Process Mining [51]

Chapter 3

Value Analysis

During value analysis, the problem will be revisited, now with more detail identifying the problem and what are the effects it has on the projects around the Integration Services team. A proposed solution is then presented as a way to answer the challenges identifies before and how they will affect business, especially in the areas identified before.

3.1 Opportunity Identification

As a consulting company, Deloitte and subsequently its internal teams like Integration Services work with various amount of technologies and this implies strict management of resources to avoid as many as possible inefficiencies. In the context of Integration Services, some of the categories that change between projects are:

- Git services: GitHub, GitLab, BitBucket Azure DevOps, ...;
- Integration platform: Tibco, MuleSoft, Azure Functions, ...;
- Deployment target: Azure Kubernetes Service, Openshift on AWS, Anypoint Cloudhub, ...;
- Pipeline orchestrator: Jenkins, Azure DevOps, GitHub Actions, GitLab Pipelines, ...

The set of these technologies will be referred to as "technology stack" or "stack". During the preparation of one project, defining the way consultant will deploy their work to production is fundamental, which usually relies on DevOps methodologies since Deloitte works with modern ways to develop software.

Chapter 2 explains how DevOps methodology brings plenty of benefits like the standard practices involved in source code management, testing deploying, and monitoring. Working on this topic each time a new project starts is seen as a standard practice as each project has characteristics that make it hard to share the solution between projects. This tied solutions to the specific context but was hardly reusable forcing teams to re-do the work every time the project starts.

Not only this but teams frequently rotate their team members, which brings additional strain to someone being onboarded to the project. In addition to the project context, understanding what is being developed, what are the constraints, who are the stakeholders, and many other things, people being onboarded need to adapt to a whole new DevOps process while forgetting the previous process because it is an entirely new project. After enquiring with the managers and senior consultants of the projects currently ongoing from

Integration Services, a new member being onboarded in a new project needs around 1 month to become comfortable with the new DevOps process.

Finally, the IT system needs to adjust to business needs and sometimes this implies changes in processes, infrastructure, or platforms used. Depending on what is being changed, it might not be feasible to fully adapt the assets in place due to their tightness with what was the original concept. This brings back the need to refactor and migrate core parts of the whole DevOps process that were previously defined and forces everyone to adapt to the new processes, which might even more than one month because instead of only one or a few new people adapting, now it is the full team dealing with it.

3.2 Value Proposition

Based on what was explained in the previous 3.1, was identified an opportunity to improve this situation by creating a DevOps process standard enough that can be easily adapted to new technology stacks and business contexts. A DevOps process that is able to grow with the business practice and able to share its improvements back so it can be reused and iterated until the moment it becomes a mature and well-built asset.

This would come in the form of a DevOps framework for Integration Services. A model where is defined as how source code is managed, how tests are executed, how the pipelines work, and how monitoring is done. All of this is packaged in a modular solution composed of a base model, where all the basic and generically available concerns are addressed and an ecosystem of modules that extend the built-in functionality to fully adjust to the business needs. Of course, it isn't feasible having an implementation that fits all the needs in all the infinite possibilities of different combinations that might exist. The proposal is to create a standard, a method that can be applied and molded to the need of the customer without losing its own DNA. With this and in the best-case scenario an implementation of a DevOps process would simply be to adjust some settings on the technical side of things, deploy it to the customer environments and make a knowledge transfer to the other teams about the platform. Deloitte consultants wouldn't need this knowledge transfer as this process would be the exact same as they are already used to working with.

Addressing the first problem identified, newer projects would benefit from having ready a structured proposal for the DevOps process, guiding the teams to what is really important in a project: bringing value to the customer. With a reusable asset like this teams will save time designing and trying to anticipate possible points of strain. They would be able to access the same effort already made from previous implementations of the framework, having a much more reliable base to work on. And this will of course be iterative. For each implementation made, feedback will be passed to the time to maintain the framework to make it more modular and efficient for the iteration.

To new people joining the project, a project like this can make a whole difference. These new people will already be familiar with the process from a previous project. The minor adjustment will be required to be done in their mind to be fully operational like *who is the pull request approver?* This will drastically reduce the time needed to be ready to use the DevOps tools, freeing time for the consultant to learn more about the project and focus their energy on delivering value to the customer through their work.

Then, after all this process, achieving a method able to evolve with business becomes much easier. If the process is well built even with a massive change like replacing the current

infrastructure e.g. moving from a legacy on-premise infrastructure to a newer and modern Kubernetes-based infrastructure in the cloud the process can easily be planned and estimated giving a clear view of what is the path without relying on what was there previously but rather with an agnostic framework that will grow with the new requirements that will appear. An example of this difference in the way of thinking would be

- Without this project: *What is the equivalent of this virtual machine running my application in the new Kubernetes infrastructure?*
- With this project: *In the newer infrastructure, the virtual machine running the applications will be replaced with pods. Each pod will execute only one application and Kubernetes will manage everything under the hood.*

In the end, there is another benefit of this approach: actively upgrading the existing platform. The framework will not stagnate. The effect of repeatedly improving the framework will allow to actively upgrade current implementations based on updates made in the framework preparing it for future demands that were not counted on in the first instance. E.g. The framework is implemented into Customer's A environment without any extra features besides the basic functionality. This is a new stack, so the development needs to be made from scratch with version 1.0.0 of the framework. The implementation is done and the asset is versioned with version 1.0.0, following the version of the framework it is based on.

A few months after Customer B starts a project that uses the same technology stack as Customer A but asks for log-based monitoring to monitor pipelines execution time and failure rate that was not in the original version of the framework. The framework is updated to support this feature as an additional plugin that can or cannot be used. The version of the framework is now 1.1.0. Then the implementation made for customer A is updated to version 1.1.0 and the new plugin is deployed. Customer B will receive the implementation much quicker due to the asset reuse and customer A can benefit from the upgrade made to 1.1.0 if it wants to.

Later on, some consultants are onboarded in any of the projects, they will be familiar with the DevOps process and easily start working actively on the project.

It is an example of how this solution can be beneficial for all parties special to Deloitte which was able to reuse assets, save consultants time developing redundant solutions and add more value to the customer with minimal effort.

In Fig 3.1 is presented the value map of this project.

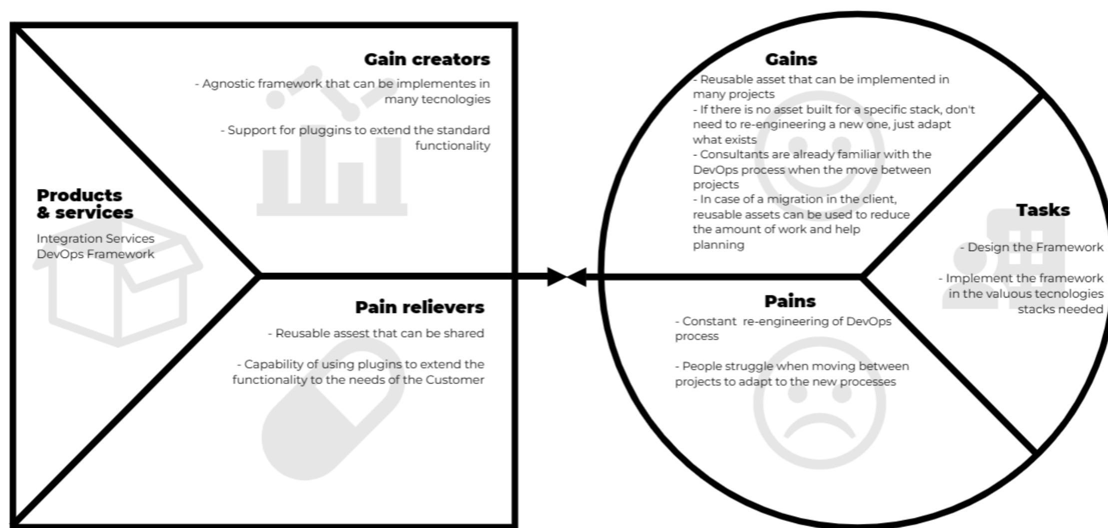

Figure 3.1: Value Map

3.2.1 Is already someone trying to do something similar?

Yes. The previous chapter mentioned two projects, An Advanced DevOps Environment for Microservice-based Applications [36] and Generic Pipelines Using Docker [8]. Both aimed to address some of the challenges already refed but with a big difference. Both mention that the project was to be used internally allowing them to strick the scope to the reality of companies. For this framework, the challenge of having mostly every possible topic configurable makes it a much more complex task to perform.

3.2.2 What are the limitations of this proposed solution?

As good as this solution might seem, there are some limitations that need to be accounted for:

- Assets production and upgrade: The work to keep the framework and its implementation updated will grow with time and more and more resources will be needed to make this maintenance. This will imply effort from everyone involved because the value of this project is intimately connected with the iterative nature and the ability to keep up-to-date.
- Customers with the process already in place: If a customer desires to keep his own process in its own environment, there is nothing Deloitte can do to change that. If even the project being presented doesn't convince the customer, then a consultant will need to learn the process customer wants.
- Time to implement a new stack. If a customer has a completely different stack from the ones already implemented, depending on the number of differences for the other implementations, this might not be very appealing to the customer.

3.3 Quality Function Deployment (QFD)

The idea of QFD (Quality Function Deployment), which was developed in Japan in the late 1960s, is to identify requirements and turn them into useful information for the development

process. As a design tool for the deployment of quality functions, the House of Quality is a component of QFD and is illustrated in Fig.3.2. This approach was created to guarantee quality and potential to the client in the framework of the project[52].

The House of Quality diagram will compare the main features of the product that are:

- Generic Framework for the team
- Implementation reusable between projects
- Possibility to upgrade with newer versions implemented in other customers
- Modularity
- Support for Plugins
- Continuous Development
- Integration of new plugins in previous implementations
- Compatible implementation between technology stacks

with the key characteristics that are needed to solve the problem

- Faster deploy DevOps processes to the client environment
- Save time developing DevOps processes
- Reuse developed assets
- Have faster onboards
- Accelerate migration scenarios
- Provide additional value to the client
- Standard practice in the team

For each character is defined as a weight which means how much important it is to solve the problem. Then each one of the key features is compared with the characteristics to identify how much of a feature is related and contributes to the characteristics needed. Finally, on the top, each feature is compared with the other on how much one feature contributes or not to another.

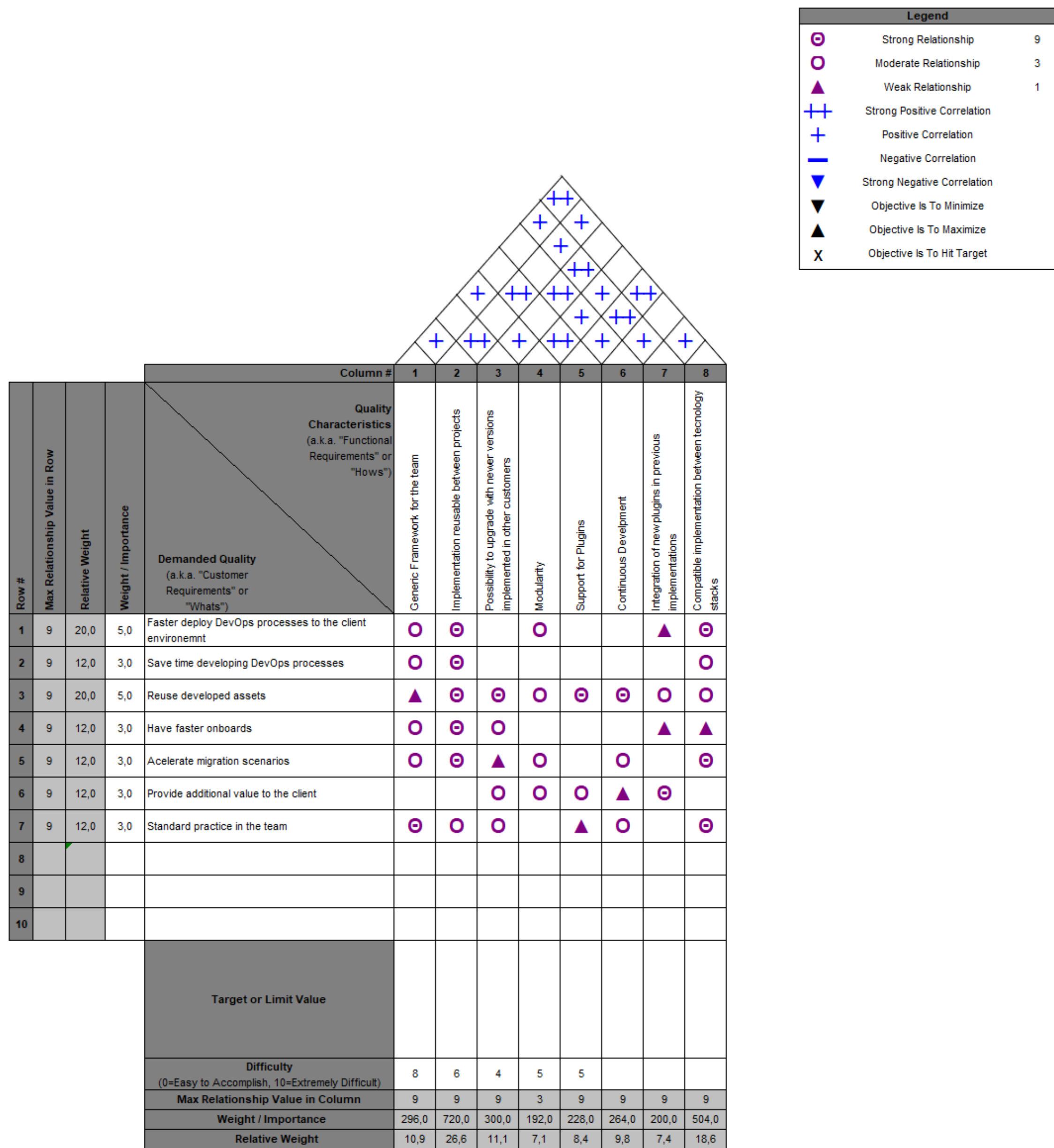

Figure 3.2: QFD - House of Quality

Chapter 4

Framework Design

In this chapter will be presented two alternative proposals for the problem identified. Those solutions will be described in a high-level way, granular enough to make possible the decision-making from Deloitte. After one of the solutions is accepted, a more detailed version of it will be presented together with the requirements (functional and non-functional). This solution will be the one implemented and evaluated in the next chapters.

4.1 Project Scope

Defining an entire DevOps framework robust and agnostic enough to be used on multiple customers from different industries that work with Deloitte isn't possible to be achieved with a single person working during the period of this project.

The subject's complexity and the level of refinement needed is something that requires a collaborative effort from multiple people from different backgrounds working together and sharing experiences.

For this reason, was decided from the beginning that the scope of this project would be reduced. Enough to touch the most critical topics but with a size that a single person could work on it effectively.

In Fig 4.1 is described this project scope when compared with a more detailed vision of DevOps.

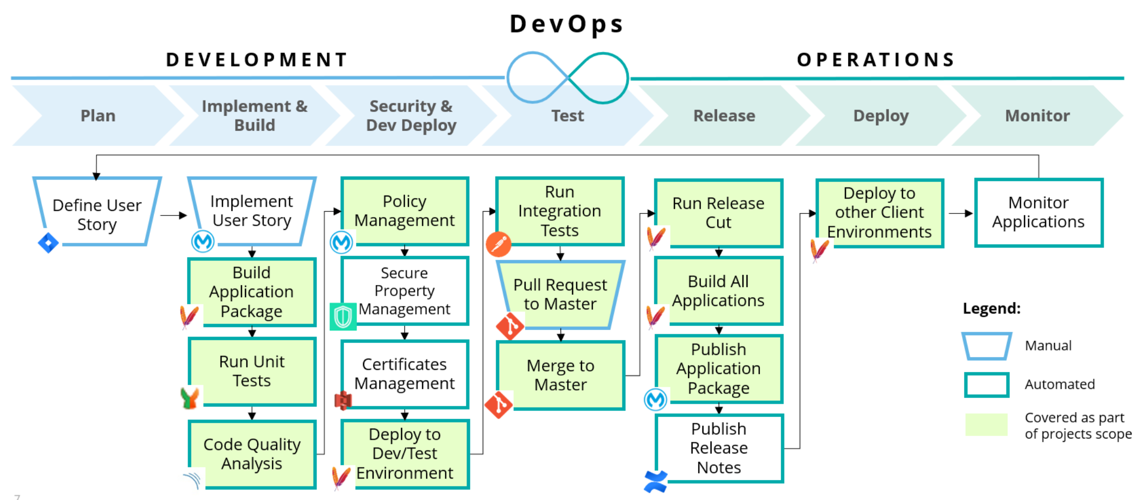

Figure 4.1: Project Scope

Some tasks are inventively manual such as designing user stories, implementing functionalities, and opening and approving pull requests. Others can be automated and from those, a subset was selected that allows validating the MVP.

4.2 Proposals Presentation

4.2.1 Proposal A

The first proposed solution is a DevOps process for teams managing multiple applications with no graphical user interface. Each application will have its git repository. The file structure will be the standard file structure for an application project within a given technology. The project files should be directly on the root level of the repository, as in the example below for a Mulesoft project.

```
1  Repository Root
2  +- src
3  |    +- main
4  |         +- mule
5  |         +- resources
6  |    +- tests
7  |         +- mule
8  |         +- resources
9  +- mule-artifact.json
10 +- pom.xml
11 +- .gitignore
```

Listing 4.1: Example of git file structuring for MulesSoft project

The git workflow will be Git Flow, so the repository will have the following mandatory branches:

- master
- develop
- feature/<name of the feature being implemented>
- hotfix/<name of the hotfix being implemented>
- release/<realise version>

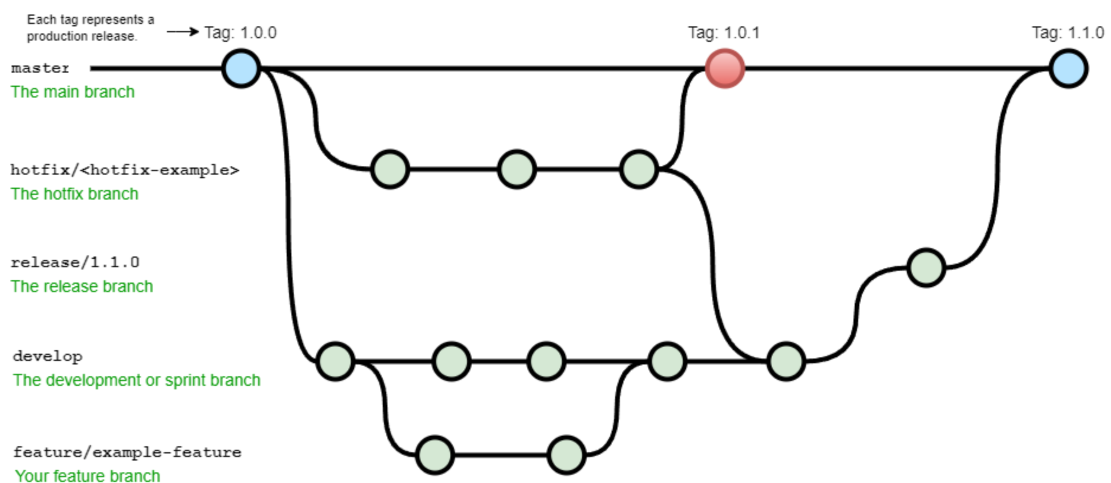

Figure 4.2: Git workflow for proposed solution A

The environment will be defined as part of the project needs, but the recommendation is four environments:

- Development: developers sandbox. Used to test the features being implemented. This is the target of feature branches
- Testing: a more stable version of applications code to be tested together to ensure proper communication between applications. This will be the target of develop branch.
- Pre-prod: Environment for tests final testing before deployment to production. This will be the target of the release and hotfix branch.
- Production: production environment. This will be the target of the master branch.

The code promotion should occur always based on a pull request between the lower branch to the other one.

Regarding the CI/CD pipeline, it will be triggered automatically after a commit from the branch targeting that environment and the steps are:

- Checkout: download source code from the git repository
- Build: Builds the application.
- Automated tests: Run application automated tests.
- Static Analysis: Run the static analysis.
- Push artifacts to Repository: Upload the built artifact (.jar, docker image, nugget package, etc...) to a repository to be used in the deployment.
- Deploy: Deploy the application based on the artifact uploaded.
-

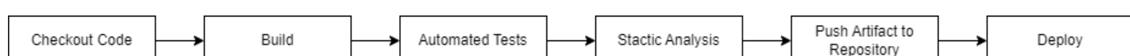

Figure 4.3: CI/CD for proposed solution A

The instructions of the pipeline will be written in shell script files that will later be executed from the pipeline itself. The pipeline should also be able to call an API to report status such as the current step, current step phase, the output of the current step, etc...

4.2.2 Proposal B

The second proposed solution is once more a DevOps process for teams managing multiple applications with no graphical user interface. Each application will have its git repository. The file structure will also be the standard file structure for an application project within a given technology. The project files should be directly on the root level of the repository, as in the example presented before.

The git workflow will be based on GitHub Flow, with some adjustments in the branches naming and appearance:

- master
- feature/<name of the feature being implemented>
- hotfix/<name of the hotfix being implemented>
- release/<realise version>

It will follow the same principle of GitHub flow, with the exception of the existence of a release branch. This branch will be used for code freeze before a release. This will give some time to the project to prepare the release without having to totally stop the code development and/or testing for features going live later on. If any change is required in the release branch to fix some test that is falling, it will be rebased into the master branch.

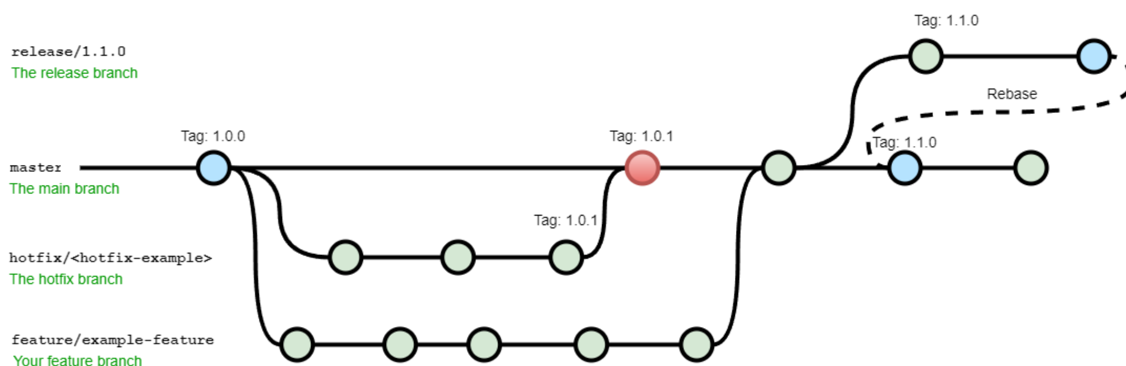

Figure 4.4: Git workflow for proposed solution B

The code promotion should occur always based on a pull request between the lower branch to the other one.

The environments are the same as presented before Development, testing, Pre-prod, and Prod. As for the CI/CD pipeline, it will follow the same logic as the one presented in proposal A, even sharing the same basic steps, but with some restrings when it relates to deployments:

- Deployment to Development: right after a commit is pushed to the central git repository the deployment in Development starts immediately.

- Deployment to Testing: the deployment starts ideally after the pull request from the feature branch to the master branch is approved and merged.
- Deployment to Pre-prod: the deployment starts ideally after the pull request from the feature branch to the master branch is approved and merged and approved by someone else.
- Deploy to Prod: Triggered manually.

Also, the deployment to production will not execute the same steps as the non-prod pipelines. Instead, the pipeline can promote the artifact used in the deployment of the pre-prod environment and use it as a deployment artifact. Then rebase the release branch into master and dele the release branch

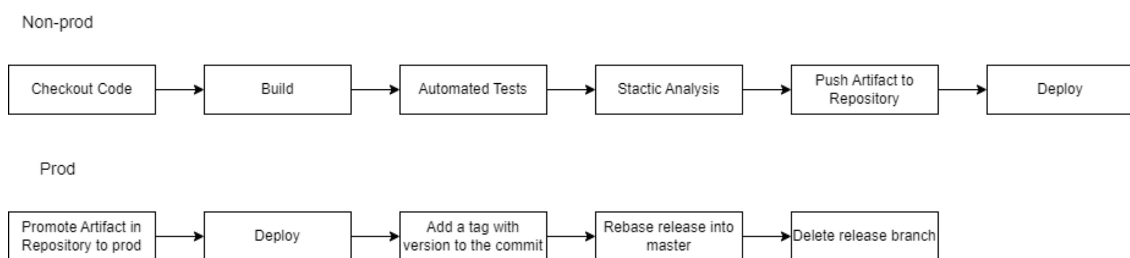

Figure 4.5: CI/CD for proposed solution B

The instructions for the pipeline written using the programming language Groovy. This will allow us to reuse some assets developed previously for other CI/CD implementations, helping to easily validate how viable the project is. Those groovy files will later be executed from the pipeline itself. For monitoring, the pipeline will not call any external system. Instead, the CI/CD server will be configured to send the logs to a linter that will later process the logs and save them in a database for future analysis.

4.3 Solution Selection

In summary, both proposals can be compared in the table 4.1

Table 4.1: Comparison summary between solutions A and B

	Solution A	**Solution B**
Git repository file structure	Standard project file structure	Standard project file structure
Git Workflow	Git Flow	Github flow with release branch
Environments proposed	Development, Testing, Pre-prod and production	Development, Testing, Pre-prod and production
CI/CD pipeline	The same in all environments	A specific set of steps for non-prod and another one for prod
Scripting language	Shell Script	Groovy
Pipeline monitoring	Syncronosly push data to an external API	Have a linter reading the log files from the pipeline and sending it to a database to be processed

Both proposals were presented to Integration Services' responsibilities and all the main points were analyzed. In the meeting, it was decided that Proposal B was the best one, and that will better suit the needs of Deloitte and its clients.

After this decision, started refinement meetings, where more detailed requirements were clarified and the design of the solution started.

4.4 Additional Requirements

From the refinement meetings extracted the following list of requirements that needs to be fulfilled:

Rq 1: The framework should consider the possibility of more than one team working on the same application at the same time. It should provide a way for the teams not to interfere with each other's work.

Rq 2: The git workflow should be the one defined in Proposal B.

Rq 3: The framework should reuse assets already produced for other CI/CD implementations doing the required refactoring.

Rq 4: Developers should be able to define a YAML file with all the configurations they required without needing to change some pipeline orchestrator-specific file.

Rq 5: The configuration files should have three levels: A framework one, where defined what is the gold standard for that repository; an organization one, stored in a different git repository used to define settings specific for the organization and an application one that applies only for the application form were the repository belongs.

Rq 6: All configuration files should share the same structure.

Rq 7: Values in the organization configuration file should override values in the framework default configurations.

Rq 8: Values in the application configuration file should override values in the framework default configurations and organization configuration file.

Rq 9: The framework should support the concept of plugins and pieces of code that can be added to extend functionality. All the code in the standard framework will be part of the framework core. anything else will be plugins.

Rq 10: Will be designed as a plugin to read the logs from the CI/CD pipeline and sent to a dedicated service for analysis.

Chapter 5

Framework Development

In this chapter, the project will be described in further depth. The architecture will be described in the first section, after which the implementation will take place. With all the justifications given throughout the chapter, the challenges encountered will also be presented along with their solutions or other paths taken.

5.1 Architecture

The project will be composed of five major components: Git Server, Pipeline Orchestrator, Deployment Target, Monitoring Service, and Artifact Repository. In the Git Server, source code will be stored and versioned. The Pipeline Orchestrator will use the versioned code in the GIT Server to perform some actions. Those actions will involve storing an artifact (jar file, docker image, etc...) in the Artifact Repository and deploying to the Deployment Target. During this process, the Monitoring Service will receive monitoring data, including metrics and logs. Fig 5.1 represents all major components.

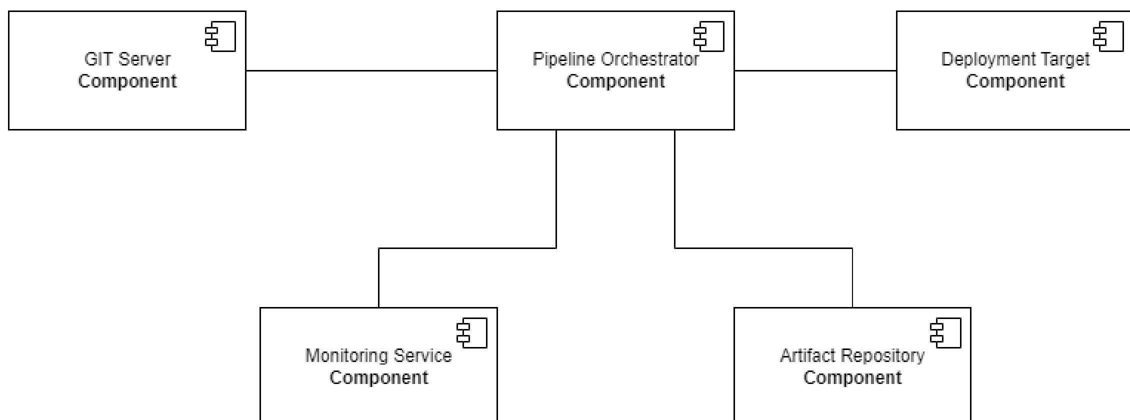

Figure 5.1: Component Diagram - Level 1

The focus of this project is on the pipeline orchestrator. This is the component being customized for each project and makes sense to start standardizing this component first. Inside it, there are two new components, the Pipeline, and the Artifacts. Based on a *trigger*, the Pipeline will execute the jobs, and produce the Artifacts that will be later used.

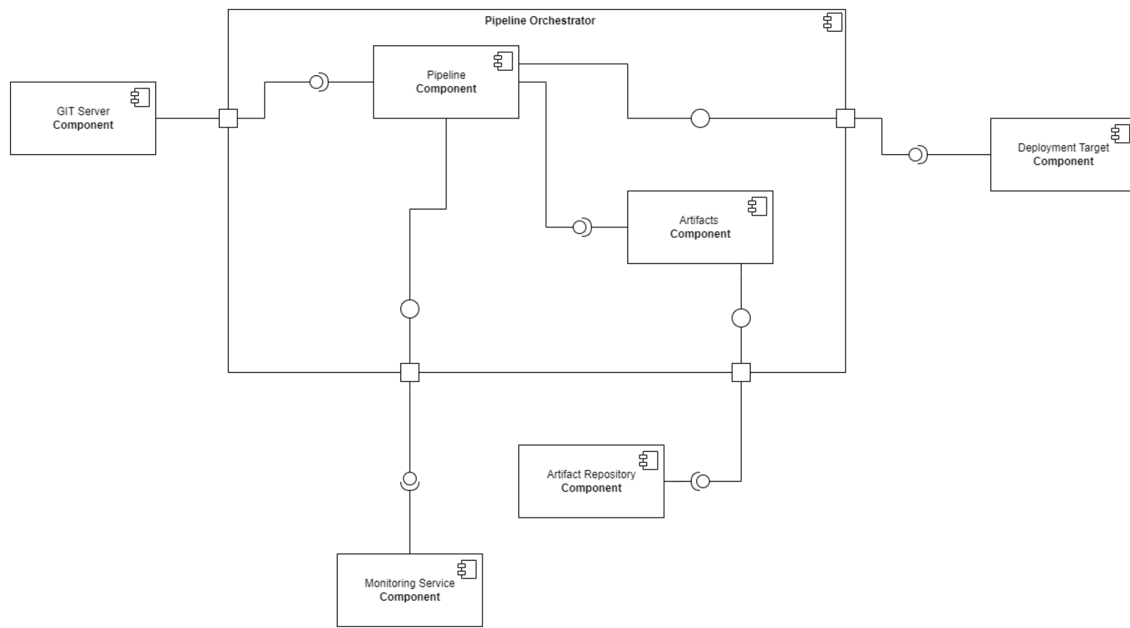

Figure 5.2: Component Diagram - Level 2

Regarding the Pipeline component, it can be described in more detail, defining three more components in it: Pipeline Manifest, Orchestration Scripts, and Step Scripts. Pipeline Manifest will define the conditions for the pipeline trigger (it can be time-based, when a commit happens, manually, or others). Both Orchestration Scripts and Step Scripts can generate metrics and logs that will be sent to the Monitoring but only Step Scripts will generate artifacts, upload them to the Artifact Repository or make a deployment. This is the more granular vision of the system from an architecture standpoint and it is represented in Fig. 5.3.

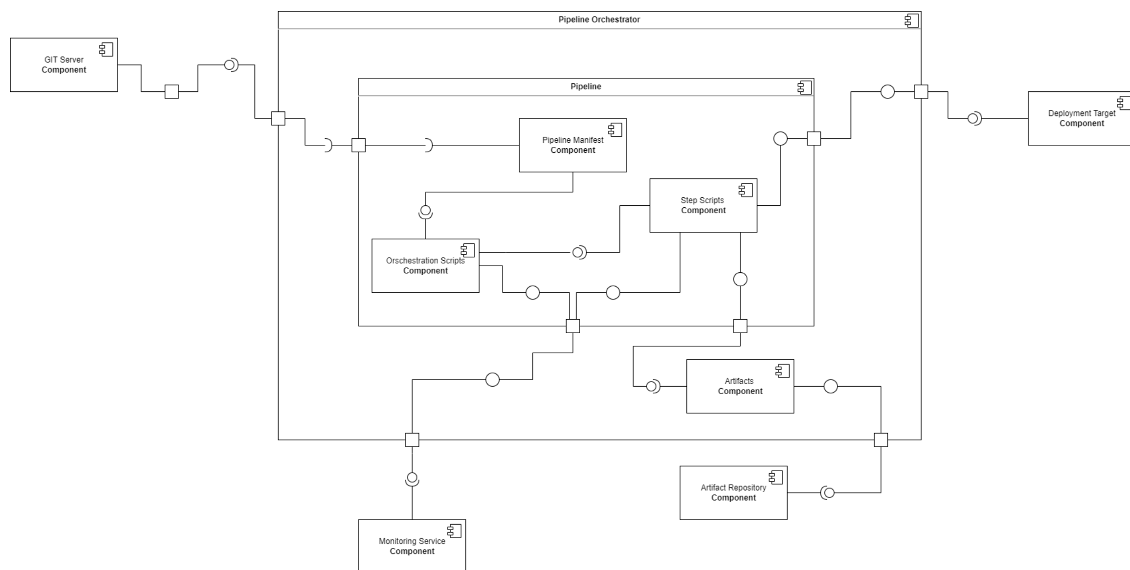

Figure 5.3: Component Diagram - Level 3

Pipeline Manifest are the files each Pipeline Orchestrator requires to have for the pipeline. Some examples of those files are `jenkinsfile` for Jenkins, `gitlab-ci.yml` for GitLab,

or `azure-pipelines.yml` for Azure DevOps. Each Pipeline Orchestrator defines its own standard. To achieve the goal of a technology-agnostic project, this is the first element to be abstracted. To achieve that abstraction, it was decided that all logic will be done in separate script files and the Pipeline will only execute those scripts.

The second challenge regarding technology abstraction is the technology on which the application is built. An approach for this challenge is to have an implementation for each combination of elements Integration Technology, Deployment Target, Artifact Repository, and Monitoring Service. Although this is possible, the reusability will be much harder as the components will be tightly coupled and the most probable outcome in the mid-term is to have multiple implementations that evolve at a very different pace, and with a lot of source code duplicated between projects. The ideal scenario is to have a single library and that library is able to detect what is the combination of technologies and load the adequate modules. To get that, the scripts were separated into two parts, the Orchestration Scripts which will identify the technologies being used and load the requested scripts from the Step Scripts. Besides not having specific step logic, and the orchestration logic being fairly similar for each step, will exist one Orchestrator Script per pipeline step. The reason behind this decision is based on S.O.L.I.D.'s "Single Responsibility Principle" and "Open Closed Principle" [53]. When a new step has to be added, no changes will be made to the already existing ones, reducing the chance of introducing new bugs in features already implemented. This strategy enables the creation of a single library that is independent of the project's chosen technology. Following this approach, the Rq 3: can be considered fulfilled.

To simplify the process, all the steps that can be executed are defined in the Pipeline Manifest file with a condition regarding that step should be enabled for that specific execution. E.g. The build step might not make sense to be executed in the pipeline run to deploy the application in the production environment. The only action in the Pipeline Manifest step is a call for the Orchestration Script. It will load the properties for that pipeline execution, check what steps should be executed on that scope, and load the right scripts to execute the step. Those scripts will perform the actions needed for that particular step. This entire process is represented in the Fig. 5.4

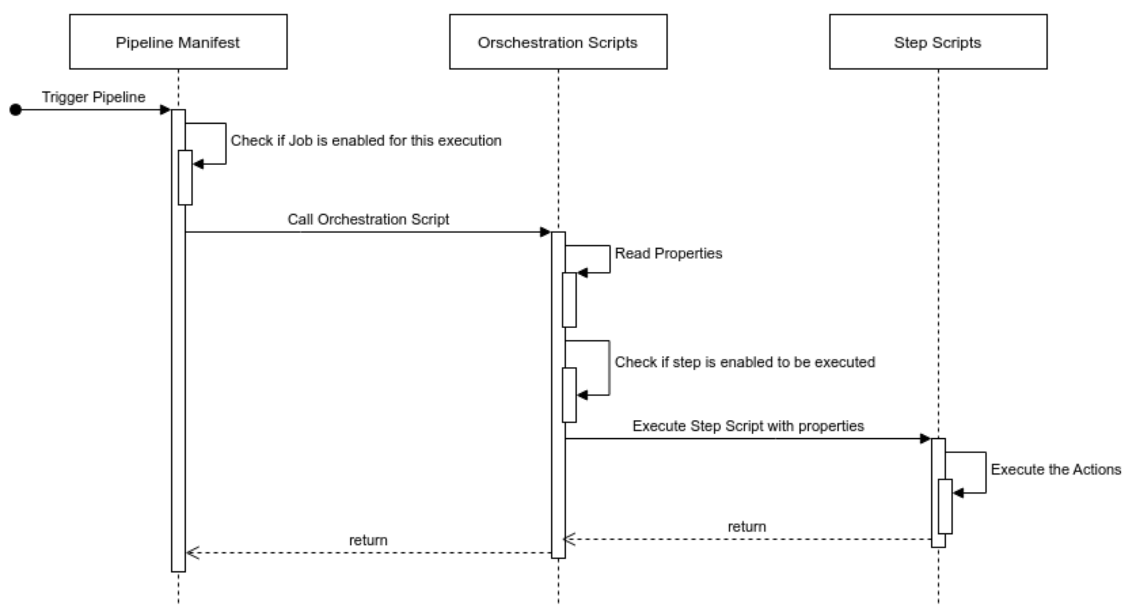

Figure 5.4: Pipeline Orchestrator Sequence Diagram

5.2 Source Code Repository

The source code repository supported is GIT. Any hosting services that support standard GIT as a repository, such as GitHub, GitLab, Azure DevOps, and others will be compatible.

5.2.1 Source Code Structure

Each application should exist in its own repository. Inside the repository, all the files should place in the same structure as a standard project of the technology being used. Additionally, some extra files, required for the process can be included.

```
(Repository Root)
+- src
|   +- main
|       +- mule
|       +- resources
|   +- tests
|       +- mule
|       +- resources
+- mule-artifact.json
+- pom.xml
+- .gitignore
```

Listing 5.1: GIT repository file structure for Mulessoft project

5.2.2 Branching

The repository has four specific types of branches that can be used:

- `master`: Contains the latest state of developed stable features. Can be replaced with `main` to follow the modern standard.
- `feature/*`: Used to develop a new feature.
- `bugfix/*`: Used to fix a bug in production.
- `release/*`: Used to make a code freeze to perform final testing before releasing to production.

All branches should also be written in cobra case (all letters lowercase and words separated by dashes "-"). Additional naming rules for the branches should be defined on a project scope, e.g. feature branches should include feature track id ou bugfix branches should include the incident number.

5.2.3 Git Workflow

All the source code in `master` is considered stable and ready for code freeze before testing and release. To develop a new feature, a new branch `feature/<feature-x>` should be created from the latest revision of `master`. To fix a bug in the production system, a `bugfix/<example-incident>` should be created from the latest revision in production. Using these branches will allow other developers to also work on top of the stable version of the application's source code, fulfilling Rq 1:. In any case, once the change is completed, a Pull Request (or Merge Request in the case of GitLab) should be opened and approved by another developer before merging to `master`. The merge is recommended to be done

using *Squash Merge*, which will combine all changes in a single commit for easy tracking and revert in case it is needed in the future.

Released should be managed according to project needs. Once a new release is planned a new branch `release/<release-indicator>` should created from the latest revision of `master`. The source code in this new branch will enter a state of *Code Freeze*, which mean that no new features will be added and only bug fixes will be included before the release. Once the release to production is concluded, all changes in the code should be included in `master` using a rebase process. Using a release branch will also allow other developers to make changes in the application without interfering with the release and the rebase process to ensure that commits are placed in the correct order (changes made for the release appear before changes made for other features), fulfilling also Rq 1:. Finally, this was the Git Workflow defined in Proposal B described in 4.3 fulfilling Rq 2:.

The GIT server should be configured to block commits directly to `master` or of `release` branches.

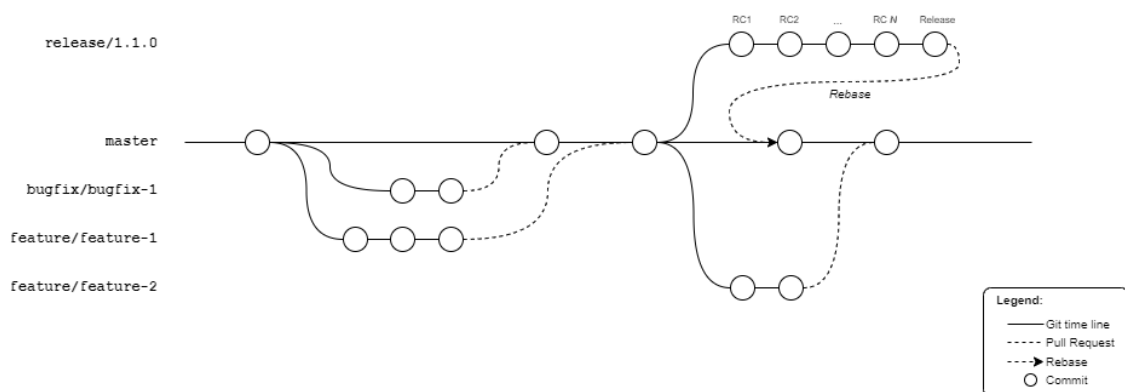

Figure 5.5: Git Workflow

5.3 Pipeline Shared Library

The Framework's CI/CD component will be implemented using a centralised shared library that will later be imported into all projects. This shared library will contain all the logic for the process no matter what the Pipeline Orchestrator or the technologies were involved to develop the application being built and deployed.

5.3.1 CI/CD Process

The first step before designing any feature or implementing any functionality is to understand the scope and the target product that's intended to be built. To avoid the most conflicts with specific Pipeline Orchestrators naming conventions, it will be now defined two specific terms:

- Job: represents a macro-activity in the pipeline. They are abstract enough to be applicable to all the targets on the scope of this project. Some examples of these macro-activities are "Build", "Deploy" or "Integration Tests". Those jobs are usually represented in the Pipeline Orchestrator user interface as separate boxes.
- Step: represents a unit of work inside of a Job. Those are all the related tasks that need to be performed for a Job to be considered.
- Action: the smaller unit of work to be performed. Those are the instructions built into each step to be completed and are specific for each technology.

Although the steps are defined, some may not be implemented in a specific implementation of the final solution. This might happen if some other part of the process takes care of that Task as part of its own behavior. For e.g. GitLab initiates the container from a Docker image and automatically checkouts the application source code when the pipeline starts whereas Jenkins might be configured so that it doesn't need to spin up a container or check out the shared library to be able to access the scripts in it. Those differences should be documented in the specific documentation file for each implementation (will be explained in more detail in section 5.3.5).

The process in Fig. 5.6 represents each of the activities that will be considered in the pipeline. Each lane represents a Job. Inside each job is a set of activities that represent each step that needs to be performed in that Job.

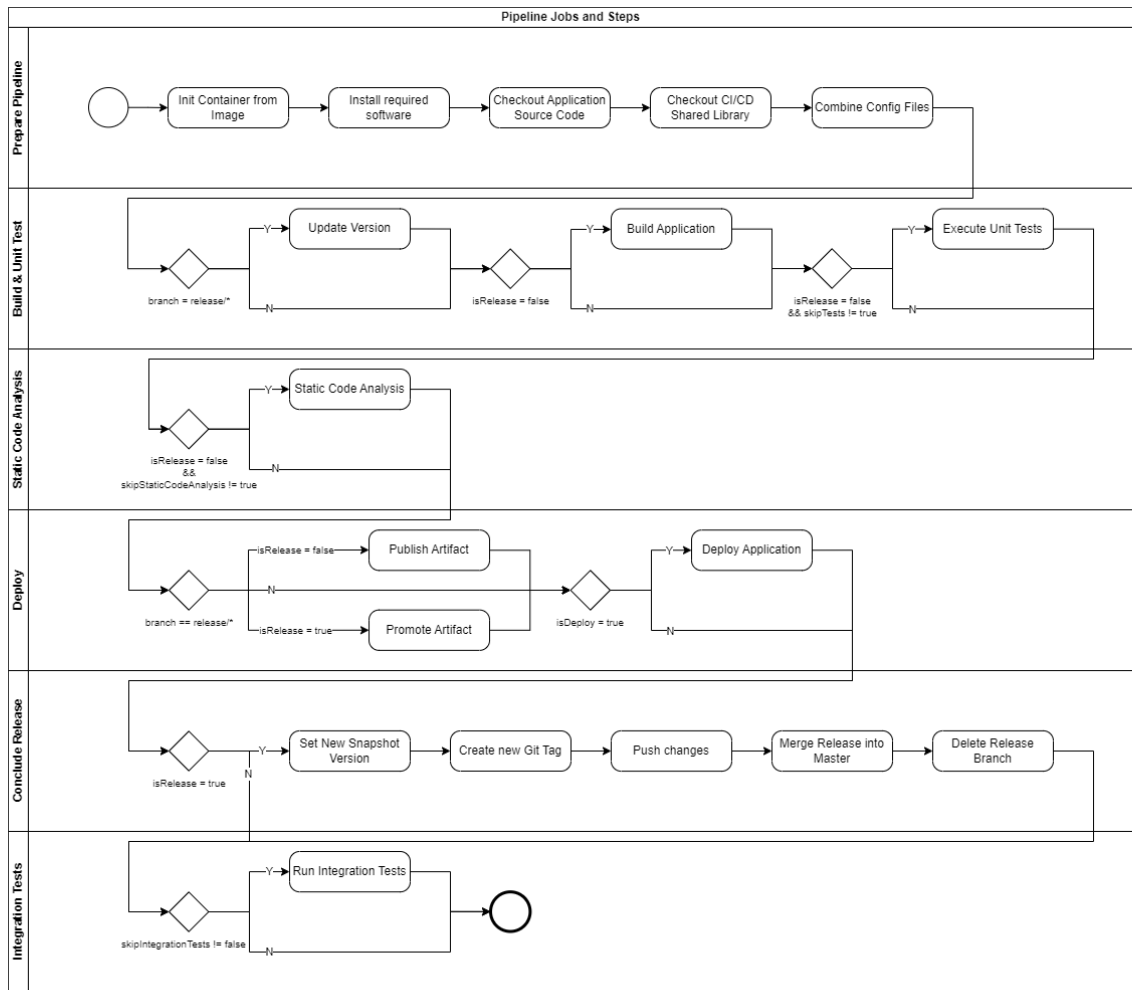

Figure 5.6: Pipeline Process

Bellow, in Table 5.1, is a detailed description of each step in the Job of the process.

Table 5.1: CI/CD Pipeline Process Jobs and Steps description

Job	Step	Description
Prepare Pipeline	Init Container from Image	Initilize Container where all the next instructions will be executed.
	Install Required Software	Install required software that is not available in the image.
	Checkout Application Source Code	Checkout application source code from GIT Server.
	Checkout Shared Library Source Code	Checkout shared library source code to access the required scripts.
	Combine Configuration Files	Combine all the configuration files (will be explained in more detail in 5.3.5).
Build & Unit Test	Update Version	If the branch from where the pipeline is running is `release/*`, update the Release Candite Version.
	Build Application	If the execution is not for a production release, build the application.
	Execute Unit Tests	If the execution is not for a production release or there isn't an indication to skip tests, execute the unit tests.
Static Code Analysis	Static Code Analysis	If it is not a production release and there isn't an indication to skip static code analysis, execute the Static Code Analysis.
Deploy	Publish Artifact	If the source code is from `release/*` branch but it is not a production release, publish the artifact built before.
	Deploy Application	If the source code is from `release/*` branch and it is a production release, promote the artifact in the repository.
	Execute Unit Tests	If deploy action is enabled deploy the artifact (either the built one or the promoted one, depending on what was done before).
Conclude Release	*For all Steps*	All the Steps from this Job only apply in the case is a production release.
	Set New Snapshot Version	Increase the version and add the "`-SNAPSHOT`" prefix.
	Create a new Git Tag	Create a new tag in the latest commit with the version being deployed now.
	Push Changes	Push all the changes to the Git Repository.
	Merge Release into Master	Merge and push the `release/*` branch into `master` branch
	Delete Release Branch	Delete `release/*` branch.
Integration Tests	Run Integration Tests	If there isn't an indication to skip integration tests, execute the integration tests.

5.3.2 Library File Structure

The file structure for the Shared Library was carefully decided to be usable, flexible, and as much future-proof as we can as for now. It is separated into multiple folders, each one with specific responsibilities to deal with.

Internal Folders

The files are organized in five top-level folders.

Documentation `/docs` Contains the documentation for the framework itself and all the implementations. It includes a file `ReadMe.md` as an entry point for the documentation. All the documentation is written in MarkDown, and all diagrams were made as SVG files. Being an SVG file is crucial since text-based SVG files are more suited for use in a Git repository than binary files.

```
(root)
+- docs
   +- ReadMe.md
   +- Implemented Features.md
+- src
   +- com
      +- deloitte
         +- shared
            +- util
               +- Command
               +- ConfigurationYaml
               +- Git
               +- Maven
               +- SemanticVersion
               +- UI
         +- jobs
            +- abstraction
               +- build
               +- unit-test
               +- deploy
               +- ...
            +- implementation
               +- common
                  +- checkout
                  +- checkout-library
                  +- merge-configs
               +- mulesoft
                  +- build
                  +- unit-test
                  +- ...
               +- cloudhub
                  +- deploy
               +- tibco
               +- ...
            +- interfaces
            +- plugins
+- vars
   +- jenkins.groovy
   +- azure-devops.yml
   +- gitlab-ci.yml
   +- ...
+- resources
   +- config
      +- baseconfig.json
+- clients
   +- jenkinsfile
   +- azure-pipelines.yml
   +- .gitlab-ci.yml
   +- ...
```

Listing 5.2: Shared Library file structure

Source Code `/src` Contains all the scripts with logic for the actions implemented. The scripts are exclusively written in Groovy. It was defined as a base package `com.deloitte` and inside of it lives three other packages:

- `shared`: for logic that can be reused across multiple implementations. This is the case of the package `utils` that contains some classes to manage Steps like changing versions according to Semantic Version, performing actions using Git or Maven, etc...
- `jobs.abstraction`: this package is intended to keep all the Orchestration Scripts. One Scrip for each Step and each script has the logic to identify and load the correct script from the other package. These processes will be explored in more detail in section 5.3.3.
- `jobs.implementation`: inside this package will exist a new package for each technology supported. Then, inside those packages, a Groovy class for each of the Steps that specific technology supports. For e.g. `mulesoft` package supports the Job "Build" whereas the package `cloudhub` supports the Job "Deploy". In section 5.3.5 this process will be explored in more detail.

Pipeline Manifests `/vars` Contains all the Pipeline Manifest files that will be imported by the pipeline when the process is triggered. In section 5.3.5 this process will be explored in more detail. This name was chosen to keep the compatibility with Jenkins.

Resources `/resources` This folder is intended to contain resource files such as configuration files or other files like bash scripts. One crucial resource present in this folder is the main configuration file available at `/resources/config/baseconfig.json`. This file must be present for the execution. This file contains all the default pre-sets of the framework and should not be changed. It will be possible to change the pre-sets and that will be explored in more detail in the section 5.3.5.

Pipeline Clients `/clients` This folder is where all supported Pipeline Orchestrators' base Manifest Files are. To enable a repository with this framework, the corresponding file (`jenkinsfile` for Jenkins, `gitlab-ci.yml` for GitLab, or `azure-pipelines.yml` for Azure DevOps) should be copied to the application repository and customized with the required info, such as the Shared Library URL.

5.3.3 Script Orchestration

To understand how the orchestration process was implemented, it is necessary to understand the orchestration fully process described in section 5.1, especially the figure 5.4 and the file structure described in the previous section. Combining both in a single diagram will result in something similar to what is in Fig 5.7.

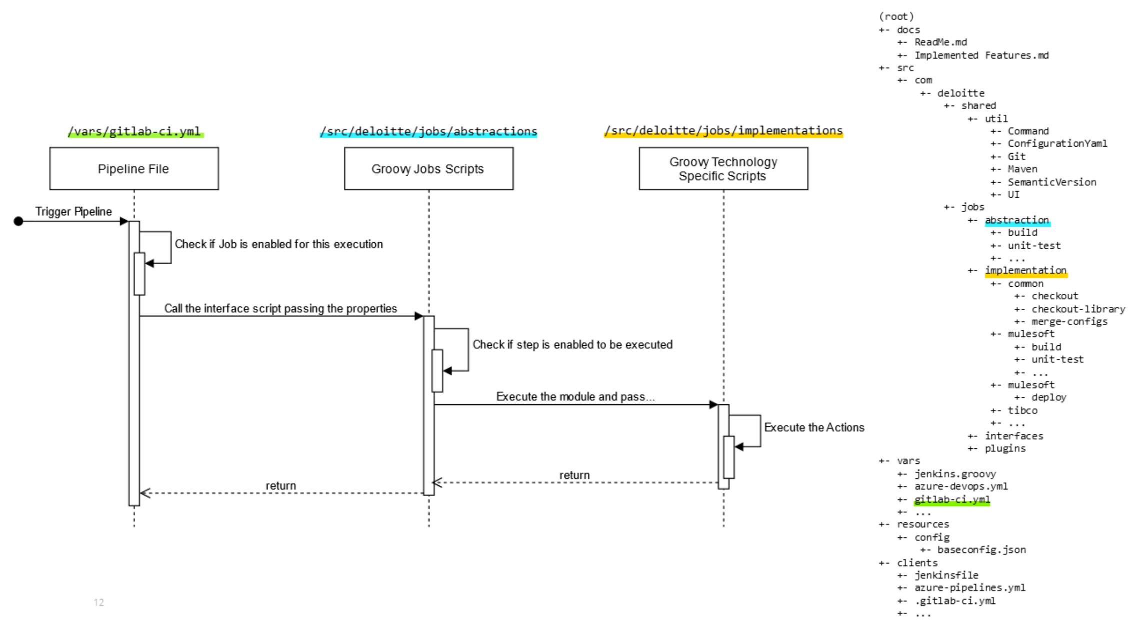

Figure 5.7: Pipeline Orchestrator Sequence Diagram VS File Structure

Pipeline Manifest files

Every application repository will contain a file that will be used to make the connection between the repository and the library. This file should be copied from `/clients` folder and its content should be the equivalent of what is in Listing 5.3 below, which is the import of Pipeline Manifest file from an external repository for the Pipeline Orchestrator in use.

```
include:
  - project: $SHARED_LIBRARY_PROJECT
    ref: $SHARED_LIBRARY_BRANCH
    file: '/vars/gitlab-ci.yml'
```

Listing 5.3: Gitlab's client file source code

Note: `$SHARED_LIBRARY_PROJECT` and `$SHARED_LIBRARY_BRANCH` are two variables defined in the context of the project repository. `$SHARED_LIBRARY_PROJECT` referees to the URL of the GIT repository where the library is stored and `$SHARED_LIBRARY_BRANCH` to the git branch that should be used.

Once the pipeline is triggered, it will load the corresponding file in `/vars`, continuing the previous example, `/vars/gitlab-ci.yml` will be loaded. All the relevant Jobs will be executed. For each job, files inside `/src/com/deloitte/jobs/abstraction` corresponding to each step will be executed. Each might or might not require shell arguments to be passed with contextual information. That information is available in the documentation. Here is also worth mentioning that the execution isn't simply calling the Groovy command line and passing the script path and arguments. Although the scripts were easily reachable from the command line a bigger issue appeared when was the moment of importing and instantiating the classes. The JVM was not able to find classes. This was due to JVM trying to load the files from the folder where the script was located (`/src/com/deloitte/jobs/abstractions`) instead the main source folder (`/src`). The solution for this was to manually define the classpath for the JVM to the main source folder. In the end, each call for the scripts looked something similar with:

```
groovy -classpath cicd-shared-library/src/ \
 cicd-shared-library/src/com/deloitte/jobs/abstractions/
    combineConfigYamlFiles.groovy
```

Listing 5.4: Shell command to trigger the Orchestration Scripts

Orchestration Scripts

Moving to the Orchestration Scripts, all files are divided into four specific blocks:

1: Define Step Id and Name

2: Read shell arguments and save those in variables

3: Import the configurations

4: Inject runtime properties in the configuration object

5: Check is the step that should be executed and if it is, load the correct class from `implementation` folder and call it.

Step Name and ID This block is mandatory and is required as part of the development standard. `STEPID` and `STEPNAME` constants might be needed during the execution of the script.

```
//Define Step id and name
final String STEPID = "updateVersion"
final String STEPNAME = "Update Version"
```

Listing 5.5: Definition of Step ID and Name

Read shell arguments and save those in variables This block is optional, as not all steps will need to receive parameters from outside.

```
//Read shell arguments
def configPath = args[0]
def gitUrl = args[1]
def gitBranch = args[2]
def isMajorRelease = args[3]
def isMinorRelease = args[4]
def isPatchRelease = args[5]
```

Listing 5.6: Capturing values from shell arguments

Import the configurations Importing configurations is a mandatory action because all the Groovy classes that will be instantiated from now have one public method that requires the config to be passed. This action is facilitated with a custom method that abstracts all the logic to read and import the configurations. Is relevant to mention that the variable `configPath` defined in the previous step is used here, making the block read shell arguments mandatory. As part of the project implementation, it was decided that de combined config file would be placed in the repository root folder (more details will be described in section 5.3.5), although it was decided to have this value always passed as an argument without any default values for visibility purposes. In the future, maybe some other technique might be applied to make block 2 truly optional, but for the time being this is the decision made.

```
//Load configMap
def config = new ConfigurationYaml().readConfigYaml(configPath)
```

Listing 5.7: Impoorting configurations

Inject runtime properties in the configuration object This block is optional, as not all steps will need to define runtime parameters. This topic is explained in further detail in this section.

```
//Inject runtime configurations
config.runtime = [:]
config.runtime.gitUrl = gitUrl
config.runtime.gitBranch = gitBranch
config.runtime.isMajorRelease = isMajorRelease
config.runtime.isMinorRelease = isMinorRelease
config.runtime.isPatchRelease = isPatchRelease
```

Listing 5.8: Inject runtime properties

Check is the step that should be executed and if it is, load the correct class from `implementation` folder and call it In this final block, the step will be instantiated and executed. It is here that the plugins are loaded and executed as well as part of the step execution. More details regarding this implementation will be described in the course of this section. Is relevant to mention that Step Name and ID are both used here not only on the banner but also in the plugin call made later in the block.

```
//Update version only for release branches
if (gitBranch.startsWith( config.ci.releaseBranchPrefix )) {
 UI.printBanner(STEPNAME)

 PluginLoader.loadPlugin(STEPID,config ,PluginLoader.BEFORE_EXECUTION)

 Object dynamicInstance = Class.forName("com.deloitte.jobs.
    implementations.${technology}.UpdateVersion").newInstance()

 Result result = dynamicInstance.execute(config)

 PluginLoader.loadPlugin(STEPID,config ,PluginLoader.AFTER_EXECUTION,
    result)
}
```

Listing 5.9: Instanciation of Step Class and plugins

In Appendix A is available an example of Orchestration script for the step "Update Version" in the Build Job of the pipeline.

Steps interface

All classes from the package `/src/com/deloitte/jobs/implementation` have to implement a common interface. This ensures that the orchestration script knows what methods exist in the class and what can be called to perform the required actions. This interface defines a method `execute`. This method returns an object of type `Result` with information about what was done in the step. The content of what is in the object `Result` is the entire responsibility of who develops the implementation.

Runtime properties

In order to provide contextual information during execution, such as the branch from which it is running, certain steps require this information to be passed to the instantiated class in the script. Two options were considered for passing this information: the first option involved passing it as a second argument, while the second option involved injecting the values into the `config` object.

The decision was made to proceed with the second approach, injecting the contextual information into the `config` object. This choice was motivated by the fact that all steps must adhere to the same interface, but not all steps necessarily require contextual information. Consequently, the second parameter would be redundant and irrelevant for steps that do not need this information.

By opting for the second approach and injecting the values into the `config` object, the implementation remains consistent across all steps, ensuring that only the necessary information is provided where it is needed, avoiding unnecessary parameters in scenarios where they are not required.

It was decided to create a new entry in the `config`, named `runtime`, and add all the relevant contextual information the step might need. This way it is possible to pass information, during runtime, without changing the interface. Using this technique is achieved a single interface that works for all steps.

Plugins

To support some level of process customization, without touching the main code base, it is introduced the concept of plugins. Plugins are small pieces of code that will be executed at the beginning and at the end of a step if it is enabled. All plugins extend an interface as well that defines two constants `BEFORE_EXECUTION` and `AFTER_EXECUTION` and one entry method named `lauchPlugin`. The method can receive the step id, the config object, one of the constants defined in the interface, and an `Result` object that is optional. The argument of type `Result` is optional as only after the step the object will be available. This feature fulfils the requirements Rq 9:.

Regarding plugin file structure, each plugin should be distributed as a package. This way of distributing the package will allow in the future to externalize the plugins in separated code repositories and be imported as Git Submodules [54]. should contain at least one class named `PluginMain` in the package root. The classes `PluginMain` will be the ones implementing the interface mentioned before and the ones being initialized in runtime.

5.3.4 Library Versioning

Framework versioning will be done using the Semantic Versions standard both for the library itself and for the applications deployed. For the applications being deployed, a class was implemented to deal with the versioning using this standard. It is planned for the future to support other versioning standards, but for now, this is the go-to route.

When it comes to the framework and library, the situation becomes more complex. While the ideal scenario would involve updating all the affected modules to a new version, this is often not feasible. As an alternative, a proposed approach aims to enable the coexistence of multiple versions. The suggestion entails creating a foundational documentation that outlines all the features the framework should support as a conceptual whole. Subsequently, the maintainers would update the modules, while meticulously documenting the implementation details and tracking all the changes made in a changelog.

By adopting this approach, the framework and library can accommodate different versions while still maintaining a comprehensive understanding of the supported features. This documentation-driven method ensures that the changes and updates are well-documented and readily accessible, facilitating the management and maintenance of multiple versions within the framework.

To support this, a table in the documentation will contain all the features and all the modules implemented. Each cell can only have one of three: "N/A" meaning that feature is not applicable for that module, a cross meaning the feature is not implemented and a check mark meaning the feature is implemented. Fig. 5.8 is a concept of what would be this table.

Feature/Implementation	Mulesoft	Cloudhub	Tibco	Docker	Azure DevOps	GitLab
Last update on framework version	1.0.0	1.0.0	1.1.0	1.1.0	1.1.0	1.1.0
Feature A	✔	N/A	✖	N/A	N/A	N/A
Feature B	N/A	N/A	N/A	N/A	✔	✔
Feature C	N/A	N/A	N/A	✔	N/A	N/A
...	...	...	...	...	...	...
Feature X	✖ ✔ N/A	✖ ✔ N/A	✖ ✔ N/A	✖ ✔ N/A	✖ ✔ N/A	✖ ✔ N/A

Figure 5.8: Framework Versioning System

Although this might work in the beginning, this is far from ideal, so the plan for the future is to have this table in a data-oriented format (JSON, XML, CSV, etc...) and a routine will check the data in this file and compare with some type of indication in the feature package itself, making this process much easier to manage in the future when the maintaining team growth. It will also be necessary to develop an update process to articulate not only the update of plugins but the packages and the library itself to ensure a more secure and feature-rich platform.

For now, this feature is out of scope due to the possibility of highly changing the product structure between the phase of POC and the actual implementation of the product.

5.3.5 Pipeline Configuration

The configuration is one of the most important parts of the project. The configuration will enable the combination of all multiple packages allowing the high reusability mentioned before. The configuration file is based on the YAML format as defined in Rq 4:.

Library Config, Organization Config, and Application Config

The project supports three levels of configuration: Library Level, Organization Level, and Application Level (fulfill Rq 5:). In each of these levels a YAML file, with the same structure (fulfill Rq 6:) but possible with different items and/or values.

The Library Level configuration file is placed in `/resources/config/baseconfig.yml`. This file is intended to have all possible keys for each one of the objects and with the values considered and recommended by the implementation team. To avoid having to change this file when the library is installed in the customer platform, another YAML file is in a new GIT repository. The new file will have the same structure but with the values adjusted for what the customer needed. All keys should be placed in the exact same path as in the Library file. During the execution of the CI/CD pipeline, this new repository will be checked-out and all the keys in the Organization Level file will overwrite the values in the Library level file (fulfill Rq 7:). In a similar way, a file named `config-ci.yml` will be placed in each application repository's root. As the organization one, this will be processed during the CI/CD pipeline, this time overwriting both what is defined in Library Level and Organization Level configurations (fulfill Rq 8:). These two files allow fine customization of the CI/CD

pipeline without having to make changes in the library file. This helps to segregate concerns regarding the configuration but also makes it more flexible for day-to-day use.

This process happens on the first job of the pipeline in a step called "Combine Config Files". All the combination logic is separated into a specific class created for it. Once all three files are combined, a new file is created in the path where the pipeline is running and preserved for the next Steps (in the case of Gitlab, this preservation is done by creating an artifact with the combined config file). After this step, all the other steps can use the same class as before to load this new combined configuration YAML file to get access to all the configurations for the execution without the need to make the whole process again.

Configurations file internal structure

In the configuration file, there are some mandatory and some optional elements. The first elements are the module selection ones. This is a group of individual keys to configure all the modules that will be used to build and deploy the application. Is here that the integration technology, the deployment target, and others are defined. These are the values Orchestration Scripts will look for to load each one of the packages to instantiate. Each of those values is the name of the packages that will then be loaded and the classes in it instantiated.

```yaml
pipelineOrchestrator: "gitlab-pipelines"
integrationTechnology: "mulesoft"
deploymentTarget: "cloudhub"
artifactRepository: "maven"
sourceRepository: "git"
builder: "maven"
```

Listing 5.10: Modules Selection

Right next to it comes `ci`. Inside this key will be set values that are relevant for the CI process itself, for example, the `skipTests` value. Here is also possible to change the prefix of the branch that will be then considered the release branch. As for now the value `release/` is set as was defined previously, although it may be useful to change in the future in some customers for some reason or development purposes.

```yaml
ci:
  releaseBranchPrefix: "release/"
  skipTests: false
  skipStaticCodeAnalisys: false
  skiptIntegrationTests: false
```

Listing 5.11: CI configuration

Next is the `plugins` key. Here is possible to enable the plugins and to define what plugins will then be loaded for each one of the steps. The values in the array "before" will be loaded before the step is executed and the values in the array "after" will be loaded after the step were executed. Each one of those strings is the name of the packages that will be loaded as explained before.

```yaml
plugins:
  enable: false
  list:
    updateVersion:
      before:
        - "Plugin A"
      after:
        - "Plugin A"
        - "Plugin B"
```

Listing 5.12: Plugins configuration

After those key, comes specific keys for the packages in use. Those can be any configuration a package needs to run and might have similarities with other keys available already. Although it is possible for a package to use a key outside of its scope, that is not recommended and it is being evaluated the possibility of changing this behavior in the future. Represented below are the keys being used now for the packages GIT and Maven already implemented.

```yaml
git:
  username: "CI-Agent"
  email: "ci-agent@deloitte.com"

```

Listing 5.13: GIT configuration

```yaml
maven:
  cleanPackage: true
  skipTests: false
  munitEnvVars: ""
  settings: ""
```

Listing 5.14: Maven configuration

Appendix B is available as an example of a configuration file.

5.3.6 Pipeline Variables

Besides all the configurations available in the YAML files, some settings depend on the execution itself. So it leveraged the use of Pipeline Variables. Those are values that can change between executions without the need of changing the configuration files. Below is a list of all the variables in use for the moment.

Table 5.2: List of Pipeline Variables

Name	Type	Default Value	Description
DEPLOY_ENV	String	N/A	Environment for the Deployment
IS_RELEASE	Boolean	false	Define if is a release code or not. Only applicable on builds from release branch
IS_MAJOR_RELEASE	Boolean	false	Indicates if the release is a major version one
IS_MINOR_RELEASE	Boolean	false	Indicates if the release is a minor version one
IS_PATCH_RELEASE	Boolean	false	Indicates if the release is a patch version one
IS_DEPLOY	Boolean	true	Define if the build should be deployed or not
SKIP_TESTS	Boolean	false	Define if unit tests should be skipped or not Only applicable on executions that run a build of the application
SKIP_INTEGRATION_TESTS	Boolean	false	Define if integration tests should be skipped or not
GIT_SERVER_URL	String	<Define as Project Variable >	Git Server URL
SHARED_LIBRARY_PATH	String	<Define as Project Variable >	Define if integration tests should be skipped or not
CONFIG_APP_FILE	String	config-ci.yml	Path for the application specific configuration file
CONFIG_BASE_FILE	String	baseconfig.yml	Base configuration filename
CONFIG_BASE_FILE_PATH	String	$ {SHARED_LIBRARY_PATH} /resources/config/ $ {CONFIG_BASE_FILE}	Path for the base configuration file
CONFIG_COMBINED_FILE	String	config-ci-combined.yml	Path where the combined configuration file will be placed

Some of those are planned to be removed, such as `IS_MAJOR_RELEASE`, `IS_MINOR_RELEASE` and `IS_PATCH_RELEASE` and replaced with logic to determine based on the branches where the code came from, but from the implementation of this POC they were used as a shortcut for some complex logic that can suffer major changes in the future.

5.4 Deployment Target, Artifact Repository, and Monitoring Service

Regarding the Deployment Target, the Artifact Repository and the Monitoring Service were considered out of the scope of this project. It was assumed the platforms will be available as standard without any customization made specifically for this project. The only adjustment made was about Monitoring Stack, configuring Log4J to send log data to an Elastic Search server where all the logs were stored. Making this setting made it possible to send logs to an external system avoiding the need for custom pipeline plugins for it. Besides not using a plugin, since logs were sent to the external system Rq 10: was considered fulfilled.

5.5 Considerations

During this chapter was described the various components of the Framework referring to its tooling, processes, and rules. All the requirements mentioned in section 4.4 were fulfilled.

Completed this part of the work, it's time to test the concept, by implementing and comparing it to real-world use cases to understand how it performs and ao well it achieves the goals proposed in the project beginning.

Chapter 6

Evaluation

In this chapter, the project will be evaluated. It will be presented data regarding the time to development, performance, and user feedback that will be used to evaluate the project.

6.1 Evaluation Methodology

The evaluation of this project was dived into three different sections. In the first one, is presented some information regarding the implementation made to establish a baseline. Then those results are compared with prior projects to understand how the situation evolved with the project. Next is a comparison more focused on implementation performance. Finally will be a survey made to the Integrations Services team after an internal presentation explaining the project with the goal of understanding if the team sees value in what was produced and if can be useful in future initiatives.

6.2 Implementation Overview

During the time of this project, four deliverables were achieved:

- Implement the initial Library Codebase, composed of orchestration scripts; utilities for UI, shell integration, Semantic Version, GIT, and Maven clients and basic logging.
- Implement Pipeline Manifest for Gitlab
- Implement Pipeline Manifest for Azure DevOps
- Implement the required packages to deploy Mulesoft to CloudHub
- Implement the required packages to deploy Tibco to an On-Prem Docker server.

The implementation started with the first analysis and design of the whole process. This activity took around 4 weeks. Next started the implementation of the initial Library Codebase together with Pipeline Manifest for Gitlab and required packages to deploy Mulesoft to CloudHub. This phase took around 12 weeks. It was followed by the implementation of the required packages to deploy Tibco to an On-Prem Docker server which took 2 weeks. In this last one, since it was java based it was possible to reuse much of the work made before needing basically to implement only the two packages missing for building Tibco applications and deploying those to the Docker Server. Finally, the Pipeline Manifest for Azure DevOps took around 2 work days. This one was converting the file created for Gitlab and making the adjustments in order to be able to run the Orchestration scripts in the same way. The Azure DevOps implementation was only tested with Mulesoft and CloudHub packages due

to licensing limitations on the Tibco side. Each week the average work in the project was around 20 hours.

Table 6.1 contains a resume of the time spent on each task:

Table 6.1: Comparison summary between presented projects and the solution built in this thesis

Task	Implementation Time	Notes
Analyse and Design the Project	4 weeks	Reuse what was implemented previously. Only needed to implement Tibco and Docker-specific packages
Implement Library Codebase	12 weeks	Implementation of the main code base, with utilities and Mulesoft and Cloudhub packages
Implement Pipeline Manifest for Gitlab		
Implement Required packages to deploy Mulesoft to CloudHub		
Implement Required packages to deploy Tibco to an On-Prem Docker server	2 weeks	Reuse what was implemented previously. Only needed to implement Tibco and Docker-specific packages
Implement Pipeline Manifest for Azure DevOps	2 work days	Reuse what was implemented previously. Only needed to implement Pipeline Manifests files. Only tested with Mulesoft and Cloudhub due to license restrictions

In addition to what was made, another team at Integration Services already started to create the packages to deploy Azure Function with Terraform and Azure DevOps. They started at the end of May but since they cannot reuse as much because there are many differences between those technologies and what was already implemented the time expected to finish the work is longer. Even with these adversities, the team already took a look at what was already made and the feedback was positive, with the most frequent feedback being about how it is easy to plan development with all the structure already built for them. At the time of writing this document, the expected time to deliver the first MVP was mid-June. If their estimations became true, the time to develop the new packages should be around 7 or 8 weeks.

6.3 Projects for Comparison

6.3.1 Project A - Mulesoft on-prem Openshift deployed with Jenkins

This project was implemented in Mulesoft being executed in an on-premise OpenShift installation with four teams working in parallel in the same landscape. The Git workflow was based on the Git flow, with a Develop branch for each team and another one where all the code was merged to be tested before going live. The CI/CD server was a Jenkins server and the pipelines were triggered manually. This project's DevOps process had three revisions that happen between 2019 and 2022. The total amount of time invested in implementing the

CI/CD process was 6 months distributed during the whole time, being the biggest periods, the original implementation that took around 3 months and two other changes that took around 1 month each. The pipeline was able to checkout code from the Git server, run unit tests and static code analysis, push the jar artifact to a Nexus server, and deploy to OpenShift based on that jar file. The releases to production were made Ad-hoc.

6.3.2 Project B - Mulesoft in Azure Kubernetes Service deployed with Azure DevOps

This project was an evolution of the previous one. Now Mulesoft is being executed in Azure Kubernetes Services with the same four teams working in parallel in the same landscape. Kept the same. The CI/CD server was Azure DevOps and the pipelines were triggered manually. This project's DevOps process was implemented between March 2022 and September 2022. During this period all the teams had to adapt to the new process at the same time that the process was evolving as well to accommodate requests from the teams. The pipeline was able to checkout code from the Git server, run unit tests and static code analysis, push a docker image to a Nexus server, and deploy it to Kubernetes using Helm. The releases to production were made Ad-hoc.

6.3.3 Project C - Mulesoft in Anypoint CloudHub deployed with GitHub Actions

This project was implemented in Mulesoft being executed on Mulesoft Anypoint CloudHub with three teams working in the same landscape. The Git workflow was based on the GitHub flow with the release branch. The CI/CD server was the GitHub Actions server and the pipelines were triggered automatically with commits to specific branches. This project's DevOps process had several revisions that happen during the year 2022 due to migrations and the restricting of clients of the company. There was a team permanently working on the Ci/CD setup continuously improving the platform. The pipeline was able to checkout code from the Git server, run unit tests, postman collection tests, and static code analysis, and push the jar artifact to Anypoint CloudHub to deploy it. The releases to production were made at the end of each quarter.

6.3.4 Project D - Tibco on-prem docker deployed with Jenkins

This project was implemented in Tibco being executed on-prem with docker. Two teams worked in the same landscape in parallel - active development and production support. The Git workflow was based on the Git flow. The CI/CD server was built in Jenkins. The pipeline was able to checkout code from the Git server, run unit tests, build a docker image, and deploy it to the application server. The releases to production were made at the end of each month.

6.3.5 Comparison with the proposed solution

The table 6.2 is presented a summary of the projects presented and what was developed in this thesis.

Table 6.2: Comparison summary between presented projects and the solution built in this thesis

	Project A	**Project B**	**Project C**	**Project D**	**Current Project**
Pipeline Orchestrator	Jenkins	Azure DevOps	GitHub Actions	Jenkins	Gitlab and Azure DevOps
Integration Technology	Mulesoft	Muleosft	Mulesoft	Tibco	Mulesoft and Tibco
Infraestructure	OpenShift	Azure Kubernetes Services	Anypoint CloudHub	On Prem	Anypoint CloudHub and Docker
Git Workflow	Git Flow	Git Flow	GitHub Flow with release branch	Git Flow	Custom flow based on GitHub Flow with release branch
Time spent developing the CI/CD	6 Months	6 Months	3 Months until now	6 Months until now	4 Months until now
Code checkout	Yes	Yes	Yes	Yes	Yes
Build	Yes	Yes	Yes	Yes	Yes
Unit Tests	Yes	Yes	Yes	Yes	Yes
Postman tests	No	No	Yes	No	Yes
Static code analysis	Yes	Yes	No	No	Yes
Deployment Artifact	Jar	Docker Image	Jar	Docker Image	Jar and Docker Image
Deployment Method	Jar	Helm	Jar	Docker	Jar and Docker
Log and External Monitoring	No	No	No	No	Yes
Extend features using plugins	No	No	No	No	Yes

6.4 Performance Comparison

Comparing performance between projects isn't a direct process. There are many variables that can influence the final result. From the number of actions performed to the hardware where the pipeline is running or the application involved in the process.

Considering that the pipeline orchestrator is something that can highly impact the performance due to hardware specifications and internal processes involved with the platform, was decided to isolate this variable and execute the actions as defined in the pipeline orchestrator files. For similar reasons, integration technologies and the application itself can have an impact that is not meaningful for this testing, so the test will be with the same target application for all the candidates. Finally, the task used for the test should be something that does not require, external network connections, communication with APIs (to upload artifacts for testing or storage), or deployments as all of those might be affected by the services being called instead of test subjects themselves.

Looking at this scenario, the more suitable test to be done is the Build Step. It will be executed through a series of executions of the Build process both with and without the execution of Unit Tests.

The conditions where the tests were executed are:

- OS: Windows 10 Enterprise
- CPU: Intel(R) Core(TM) i7-1185G7
- RAM: 32GB
- Storage: 512GB NVME SSD
- Java: OpenJDK 1.8.0_372
- Groovy: 4.0.11
- Maven: 3.8.6

Before each execution, all JVM-related processes were killed and were removed the flags to force an update of maven's cache from all commands executed. The cache was pre-filled before any execution.

On the test subjects, all four projects use the command `mvn clean package -U`. The argument `-U` was replaced with `-o` for an offline execution to prevent more to the maven repository. To test without the unit testing, the argument `-DskipTests` was passed as well. The tests were run 10 times for each task. The result is shown below.

Table 6.3: Comparison between presented projects and the solution built

Test	Medium (s)	Standard deviation (s)
Run the maven command from the console with tests	126.74	19.94
Run the maven command from the console without tests	53.82	6.90
Run the Build script with tests	160.72	15.30
Run the Build script without tests	88.94	9.36

From the results above is possible to determine that the execution time increases by about 30 seconds with the Orchestration scripts. It was expected some overhead from using such a complex approach when compared with the raw command from the shell but 30 seconds seems excessive. Some investigation on the matter found that the Groovy script was taking around 25 seconds to start. This might be related to the internal process Groovy needs to perform before being ready to execute the script.

Assuming a delta of extra 25 seconds in each groovy script execution when compared with the action executed with standard command execution it will increase the pipeline time by up to 4 minutes of execution. This is a very expressive value and forces the team to evaluate the viability of the project.

6.4.1 Testing a new hipotesys

During conversations with the team about the issue, it was proposed to test with Python. Python is a programing language known for being easy to learn and being relatively fast for an interpreted language (Groovy is an interpreted language as well).

To test the hypothesis, two Python scripts were created to build the application with and without unit tests. To be precise, this test is not intended to compare the performance of executing the maven command with Python and Groovy. In both implementations maven is an external tool being executed from the script. The goal here is to understand how much faster the Python interpreter is starting and how much of that can influence the overall pipeline's execution time. The Python scripts created are the following.

```
import subprocess
subprocess.Popen('mvn clean
    package -o', shell=True)

```

Listing 6.1: Python script to build the package with unit tests

```
import subprocess
subprocess.Popen('mvn clean
    package -o -DskipTests',
    shell=True)
```

Listing 6.2: Python script to build the package without unit tests

After making 10 runs with both scripts, the results are the following:

Table 6.4: Comparison between presented projects, the solution built, and Python

Test	Medium (s)	Standard deviation (s)
Run the maven command from the console with tests	126.74	19.94
Run the maven command from the console without tests	53.82	6.90
Run the Build Groovy script with tests	160.72	15.30
Run the Build Groovy script without tests	88.94	9.36
Run the Build Python script with tests	141.41	22.32
Run the Build Python script without tests	74.80	5.88

Using Python is still slower than running the command directly in the console, but Python doesn't seams to suffer from the same initiation problem as Groovy. Results show that using Python reduces almost half of the time difference between using a script and the command directly on the shell. Of course, some overhead will always exist due to the intermediate

element but these results indicate an increase in time for the pipeline execution with other scripting languages can be so much reduced that will be barely noticeable in a real-world application.

Finally was decided to investigate more about the language for the product implementation, moving away from Groovy to improve performance.

6.5 Team Pulse

To complement the analysis made in the previous sections it was requested to team members of Integration Services provide feedback on the work done. This was made by having a presentation of the project for all analysts, consultants, senior consultants, and managers of the team. The problem that motivated the project was described and then the solution was explained, describing the benefits for the audience in it, which consisted in explaining how can this be implemented in various clients and what differentiated this project from other implementations made before, how the framework was implemented and how can the teams operate it when in a project context. At the end of the presentation, it was reserved a time slot for questions from people in the audience, where everyone had the opportunity to make their own questions and provide some feedback already.

After the presentation, a form was sent to everyone asking the following questions to evaluate the team value perception on the project developed.

- *What is your career level at Deloitte?* Close response question, with the options "Consultant", "Senior Consultant" and "Manager". This question will be used to differentiate the populations in the analysis made after.
- *Did you participate in more than one project since you work at Deloitte?* Close response question with the options "Yes" or "No".
- *Did you build any DevOps process in any project you participated in since you joined Deloitte?* Close response question with the options "Yes" or "No".
- *Did any of your projects at Deloitte have any migration such as infrastructure, git server, pipeline orchestrator, or team restructuring during the period you participated in that project?* Close response question with the options "Yes" or "No".
- *Do you think the teams and/or the customer would benefit from implementing this product in the project environment?* Close response question with the options "Yes" or "No".
- *What do you think will be the benefits for your work if this product was available in more projects at this moment?* Open response question.

6.5.1 Result analysis

Starting by understanding the responses to each individual question.

What is your career level at Deloitte?

For the first question, the goal was to understand the population replying to the form.

Table 6.5: Response Distribution to question 1

Analyst	Consultant	Senior Consultant	Manager
30 %	33 %	26 %	11 %

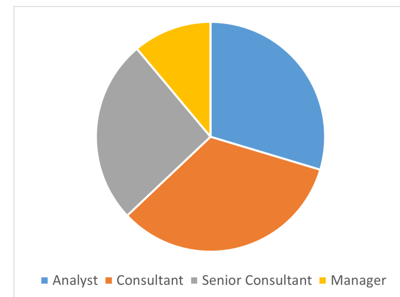

Figure 6.1: Graph distribution for question 1

Did you participate in more than one project since you work at Deloitte?

Moving forward to understand how much of Deloitte employees worked already in different projects and whether the results were as expected the majority.

Table 6.6: Response Distribution to question 2

Yes	No
63 %	37 %

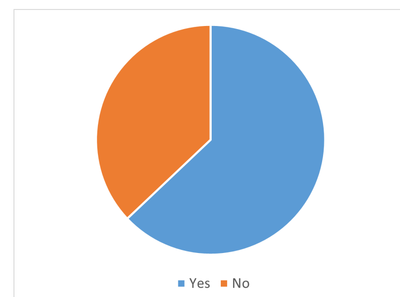

Figure 6.2: Graph distribution for question 2

And this value only increases if the responses are grouped by career level, with 67% of the consultants replying positively and 100% of the senior consultants and managers. This shows how much people could benefit from the solution presented.

Did you build any DevOps process in any project you participated in since you joined Deloitte?

Moving to a more specific question, here the intention was to understand how many people had to deal with developing their own DevOps process.

Table 6.7: Response Distribution to question 3

Yes	No
30 %	70 %

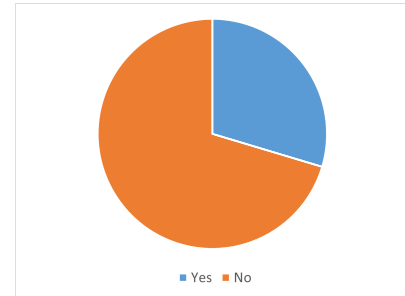

Figure 6.3: Graph distribution for question 3

Here the population reduces fairly, but still, 30% of the population is a relevant amount. And once more filtering by career level shows that 44.5% of the consultants and 43% of the senior consultants worked in these activities.

Did any of your projects at Deloitte have any migration such as infrastructure, git server, pipeline orchestrator, or team restructuring during the period you participated in that project?

On the question regarding platform migrations, 59% of the inquiries replied positively, once more describing the potential of this project. Zooming on the career level, this number increases to 55.6% of the consultants, and 85.7% of the senior consultants. This shows that almost all of the Deloitte Integration Consultants had to perform migration of some kind and this once more shows how useful this tool can be to the team.

Table 6.8: Response Distribution to question 4

Yes	No
59 %	41 %

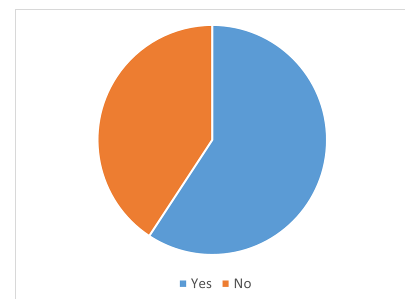

Figure 6.4: Graph distribution for question 4

Do you think the teams and/or the customer would benefit from implementing this product in the project environment?

To get feedback from the team about this project, all the inquired people replied positively about the usefulness of this project.

Table 6.9: Response Distribution to question 5

Yes	No
100 %	0 %

Figure 6.5: Graph distribution for question 5

What do you think will be the benefits for your work if this product was available in more projects at this moment?

This was an open-answer question. The most written word was "time" appearing in 22% of answers. Other relevant words/expressions were "deployment", "framework", "DevOps" and "CI/CD"

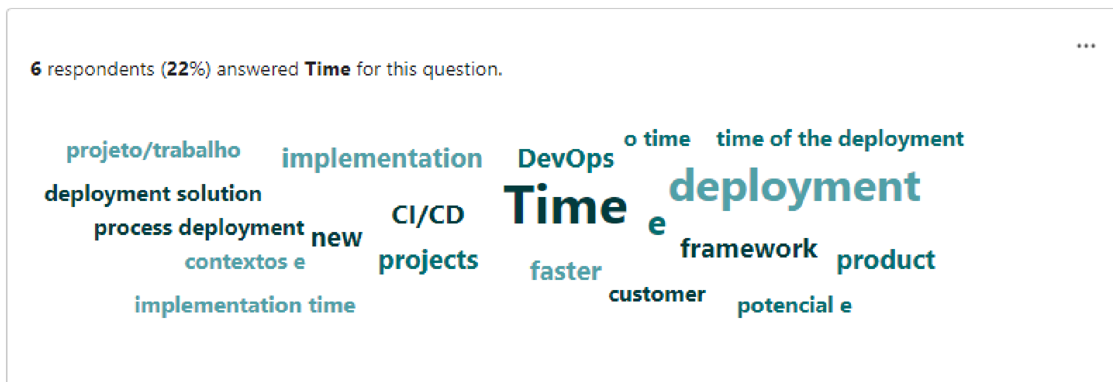

Figure 6.6: List of most written words in question 6

Chapter 7

Conclusions

This project appears as a response to the problem of a non-standardization of DevOps processes in Deloitte Integration Services projects. The goal was to come up with a solution of building a standard that could be adopted and reused by many projects avoiding reengineering previous solutions and merging together work for many professionals combining their experience in a product that will fit better every project.

The first step was to do a state-of-the-art analysis, of what are the people doing on their own DevOps processes and to understand what level of abstraction is put into their work. Was found that even with some abstraction, there is a tendency to make some of the elements static such as the pipeline orchestrator. The research was made on the technologies that are relevant for the Integration Services team as part of a state-of-the-art study aiming to build a solution that fits the needs of this specific type of software development, ignoring factors that are not relevant, e.g. graphical user interface testing.

After the state-of-the-art study, two proposals for the solution were designed and presented to the Deloitte Integration Services responsible which decided on one of the proposals and included some extra requirements that had to be included in the final version of the project.

After analyzing and choosing one of the options, it was time to start defining the architecture of the process with all of its rules and concerns. It led to the development of a PoC for a shared library to be used in CI/CD pipelines together and a set of rules to manage the Git repositories. Those steps were completed with the creation of packages for building Mulesoft and Tibco, deployment to Anypoint Cloudhub and Docker, and pipeline operation on GitLab and Azure DevOps.

Finished the PoC, was compared with previous DevOps projects made before, concluding that the implementation time can be highly improved. Although it is good news, the execution times suffered due to the use of Groovy which take longer times to start and hurt the performance of the overall execution. It was proposed as a future step to migrate the project to a more modern and fast scripting language like Python. This project was presented to the Integration Services team. The results showed the team's confidence in the value of the project and what it can bring to everyone.

7.1 Project goals

In section 1.3.2 the goals for the project were defined. It was defined successfully a basic Framework that was used to validate the concept together with the team. This Framework also complied with all the requirements defined in section 4.4. On a macro scale, their main goals were defined to be an answer to the three pains identified as part of the problem: consultants' migration between projects, projects' initial DevOps process, and migrations in the customer environment. All of those are also achieved with the result of this project. Based on a modular architecture, where all the implementation packages are behind a common interface, the transition of all the consultants should be easier due to the familiarity they already should have with the Framework itself. With the capacity of developing each one of the packages separately and reuse them as required, it is possible to achieve simply take the ones needed for a new project or to use them in a new infrastructure after the migration. Even when a package is missing, the effort is only to implement the missing package instead of the full process. In the POC itself, some inefficiencies were identified. It was expected until a certain level the additional overhead due to the extra components needed although Groovy is not the best tool due to its own performance issues. It was a good starting point to reuse some libraries already implemented to speed up the POC's implementation, but the transition to a more performant technology is a much-required step for the success of the project in the long term.

7.2 Future work

Concluding the project and making a retrospective, there are some improvements to be made in the future. The first is to research about the language to implement the project and make the migration of what was done so far to that new language to minimize the performance impact and future work redoing parts of the code. Next will be the implementation of all the components from the DevOps process that were mentioned to be out of scope in chapter 4. Another important topic to deal with is the process update of the library and its components like core utilities, packages, and plugins and distributing those updates to the customer environment. Together with a way of automatically validating the versions of packages and plugins to prevent issues running the CI/CD pipeline. Finally, the last improvement points identified were a change in the process to block packages to be able to read properties from other packages and support for more versioning methods besides Semantic Version.

Bibliography

[1] *MuleSoft | Integration Platform for Connecting SaaS and Enterprise Applications*. url: https://www.mulesoft.com/ (visited on 11/22/2022).
[2] *TIBCO Homepage | TIBCO Software*. url: https://www.tibco.com/ (visited on 11/22/2022).
[3] *Create awesome digital experiences quickly, easily, and securely*. url: https://wso2.com/ (visited on 11/22/2022).
[4] *Home - Apache Camel*. url: https://camel.apache.org/ (visited on 11/22/2022).
[5] *Git*. url: https://git-scm.com/ (visited on 12/09/2022).
[6] Scott Chacon and Ben Straub. "Pro Git". In: *Pro Git* (2014). doi: 10.1007/978-1-4842-0076-6.
[7] Kasun Indrasiri and Prabath Siriwardena. *Microservices for the Enterprise*. Apress, 2018. isbn: 978-1-4842-3857-8. doi: 10.1007/978-1-4842-3858-5.
[8] Brandon Atkinson and Dallas Edwards. *Generic Pipelines Using Docker*. Apress, 2018. isbn: 978-1-4842-3654-3. doi: 10.1007/978-1-4842-3655-0.
[9] Michael Cade. *90 Days Of DevOps*. url: https://github.com/MichaelCade/90DaysOfDevOps (visited on 04/27/2023).
[10] *What is Azure—Microsoft Cloud Services | Microsoft Azure*. url: https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-azure/ (visited on 04/28/2023).
[11] *History of Microsoft Azure*. url: https://techcommunity.microsoft.com/t5/educator-developer-blog/the-history-of-microsoft-azure/ba-p/3574204 (visited on 04/28/2023).
[12] *Amazon Web Services (AWS)*. url: https://aws.amazon.com/pt/ (visited on 04/28/2023).
[13] *What is AWS*. url: https://aws.amazon.com/what-is-aws/ (visited on 04/28/2023).
[14] *Jenkins User Documentation*. url: https://www.jenkins.io/doc/ (visited on 04/28/2023).
[15] *GitHub*. url: https://github.com/ (visited on 04/28/2023).
[16] *Microsoft acquires Github*. url: https://news.microsoft.com/announcement/microsoft-acquires-github/ (visited on 04/28/2023).
[17] *GitHub actions*. url: https://github.com/features/actions (visited on 04/28/2023).
[18] *Gitlab*. url: https://gitlab.com/ (visited on 04/28/2023).
[19] *Gitlab CI*. url: https://docs.gitlab.com/ee/ci/ (visited on 04/28/2023).
[20] *Azure DevOps Services | Microsoft Azure*. url: https://azure.microsoft.com/en-us/products/devops (visited on 04/28/2023).
[21] *Tibco Documentation*. url: https://docs.tibco.com/pub/activematrix_businessworks/6.5.0/doc/html/GUID-F08094D0-4FC0-4D77-B7DB-5D6E857A895F.html (visited on 11/22/2023).
[22] *MuleSoft Anypoint Platform*. url: https://www.mulesoft.com/pt/platform/anypoint-platform-features (visited on 11/22/2023).

[23] *Azure Functions documentation | Microsoft Learn*. url: https://learn.microsoft.com/en-us/azure/azure-functions/ (visited on 04/28/2023).

[24] *Docker overview | Docker Documentation*. url: https://docs.docker.com/get-started/overview/ (visited on 04/29/2023).

[25] *Kubernetes*. url: https://kubernetes.io/ (visited on 04/29/2023).

[26] *Red Hat OpenShift makes container orchestration easier*. url: https://www.redhat.com/en/technologies/cloud-computing/openshift (visited on 04/29/2023).

[27] *What is OpenShift? How to do OpenShift monitoring | Dynatrace news*. url: https://www.dynatrace.com/news/blog/what-is-openshift-2/ (visited on 04/29/2023).

[28] *Managed Kubernetes Service (AKS) | Microsoft Azure*. url: https://azure.microsoft.com/en-us/products/kubernetes-service/ (visited on 04/28/2023).

[29] *Michael Cade | LinkedIn*. url: https://www.linkedin.com/in/michaelcade1/ (visited on 04/27/2023).

[30] Ramtin Jabbari et al. “What is DevOps? A systematic mapping study on definitions and practices”. In: *ACM International Conference Proceeding Series* 24-May-2016 (May 2016). doi: 10.1145/2962695.2962707. url: https://dl.acm.org/doi/10.1145/2962695.2962707.

[31] Leonardo Leite et al. “A Survey of DevOps Concepts and Challenges”. In: *ACM Computing Surveys (CSUR)* 52 (6 Nov. 2019). issn: 15577341. doi: 10.1145/3359981. url: https://dl.acm.org/doi/10.1145/3359981.

[32] Maximilien De Bayser, Leonardo G. Azevedo, and Renato Cerqueira. “ResearchOps: The case for DevOps in scientific applications”. In: *Proceedings of the 2015 IFIP/IEEE International Symposium on Integrated Network Management, IM 2015* (June 2015), pp. 1398–1404. doi: 10.1109/INM.2015.7140503.

[33] Dario Bruneo et al. “CloudWave: Where adaptive cloud management meets DevOps”. In: *Proceedings - IEEE Symposium on Computers and Communications* Workshops (Sept. 2014). issn: 15301346. doi: 10.1109/ISCC.2014.6912638.

[34] Constantine Aaron Cois, Joseph Yankel, and Anne Connell. “Modern DevOps: Optimizing software development through effective system interactions”. In: *IEEE International Professional Communication Conference* 2015-January (Jan. 2015). issn: 21581002. doi: 10.1109/IPCC.2014.7020388.

[35] Barbara Kitchenham and Pearl Brereton. “A systematic review of systematic review process research in software engineering”. In: *Information and Software Technology* 55 (12 Dec. 2013), pp. 2049–2075. issn: 0950-5849. doi: 10.1016/J.INFSOF.2013.07.010.

[36] Stefan Throner et al. “An Advanced DevOps Environment for Microservice-based Applications”. In: *Proceedings - 15th IEEE International Conference on Service-Oriented System Engineering, SOSE 2021* (Aug. 2021), pp. 134–143. doi: 10.1109/SOSE52839.2021.00020.

[37] Shigeru Hosono and Yoshiki Shimomura. “Application lifecycle kit for mass customization on PaaS platforms”. In: *Proceedings - 2012 IEEE 8th World Congress on Services, SERVICES 2012* (2012), pp. 397–398. doi: 10.1109/SERVICES.2012.80.

[38] Manish Virmani. “Understanding DevOps & bridging the gap from continuous integration to continuous delivery”. In: *5th International Conference on Innovative Computing Technology, INTECH 2015* (July 2015), pp. 78–82. doi: 10.1109/INTECH.2015.7173368.

[39] B. S. Farroha and D. L. Farroha. “A framework for managing mission needs, compliance, and trust in the DevOps environment”. In: *Proceedings - IEEE Military Communications Conference MILCOM* (Nov. 2014), pp. 288–293. doi: 10.1109/MILCOM.2014.54.

[40] Floris Erich, Chintan Amrit, and Maya Daneva. “Cooperation between information system development and operations: A literature review”. In: *International Symposium on Empirical Software Engineering and Measurement* (Sept. 2014). issn: 19493789. doi: 10.1145/2652524.2652598.

[41] Florian Beetz and Simon Harrer. “GitOps: The Evolution of DevOps?” In: *IEEE Software* 39 (4 2022), pp. 70–75. issn: 19374194. doi: 10.1109/MS.2021.3119106.

[42] Long Nguyen Ba. “Enterprise-grade CI/CD pipeline for mobile development”. In: (2022).

[43] Eberhard Wolff. “A practical guide to continuous delivery”. In: (). url: https://www.oreilly.com/library/view/a-practical-guide/9780134691626/.

[44] Ibrahim Hamzane and Badr EL Khalyly. “Towards an IT Governance of DevOps Metamodel”. In: IEEE, Dec. 2021, pp. 824–827. isbn: 978-1-6654-1634-4. doi: 10.1109/DASA53625.2021.9682334.

[45] *Gitflow Workflow*. url: https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow (visited on 04/05/2023).

[46] Che Yu Chang, Pei Pei Ou, and Der Jiunn Deng. “Cross-site large-scale software delivery with enhanced git branch model”. In: *Proceedings of the IEEE International Conference on Software Engineering and Service Sciences, ICSESS* 2019-October (Oct. 2019), pp. 153–156. issn: 23270594. doi: 10.1109/ICSESS47205.2019.9040834.

[47] *GitHub Flow*. url: https://githubflow.github.io/ (visited on 04/05/2023).

[48] *GitHub flow - GitHub Docs*. url: https://docs.github.com/en/get-started/quickstart/github-flow (visited on 04/05/2023).

[49] Corinne Pulgar. “Eat your own DevOps”. In: (Oct. 2022), pp. 225–228. doi: 10.1145/3550356.3552395. url: https://dl.acm.org/doi/10.1145/3550356.3552395.

[50] Lucy Ellen Lwakatare et al. “DevOps in practice: A multiple case study of five companies”. In: *Information and Software Technology* 114 (Oct. 2019), pp. 217–230. issn: 0950-5849. doi: 10.1016/J.INFSOF.2019.06.010.

[51] Ana Filipa Nogueira and Mário Zenha-Rela. “Monitoring a CI/CD Workflow Using Process Mining”. In: *SN Computer Science* 2 (6 Nov. 2021), pp. 1–16. issn: 26618907. doi: 10.1007/S42979-021-00830-2/FIGURES/9. url: https://link.springer.com/article/10.1007/s42979-021-00830-2.

[52] *QFD Online*. url: http://www.qfdonline.com/ (visited on 12/11/2022).

[53] R.C. Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Martin, Robert C. Prentice Hall, 2018. isbn: 9780134494166.

[54] *Git Tools - Submodules*. url: https://git-scm.com/book/en/v2/Git-Tools-Submodules (visited on 05/13/2023).

Appendix A

Orchestration script for the Step "Update Version"

```
package com.deloitte.jobs.abstractions

import com.deloitte.jobs.implementations.common.PluginLoader
import com.deloitte.shared.internal.Result
import com.deloitte.shared.utils.ConfigurationYaml
import com.deloitte.shared.utils.UI

//Define Step id and name
final String STEPID = "update-version"
final String STEPNAME = "Update Version"

//Read shell arguments
def configPath = args[0]
def gitUrl = args[1]
def gitBranch = args[2]
def isMajorRelease = args[3]
def isMinorRelease = args[4]
def isPatchRelease = args[5]

//Load configMap
def config = new ConfigurationYaml().readConfigYaml(configPath)

//Inject runtime configurations
config.runtime = [:]
config.runtime.gitUrl = gitUrl
config.runtime.gitBranch = gitBranch
config.runtime.isMajorRelease = isMajorRelease
config.runtime.isMinorRelease = isMinorRelease
config.runtime.isPatchRelease = isPatchRelease

//Define technology for dynamic class instantiation
def technology = (String) config.integrationTechnology

//Update version only for release branches
if (gitBranch.startsWith( config.ci.releaseBranchPrefix )) {
    UI.printBanner(STEPNAME)

    PluginLoader.loadPlugin(STEPID, config, PluginLoader.
        BEFORE_EXECUTION)
```

```
39
40     Object dynamicInstance = Class.forName("com.deloitte.jobs.
    implementations.${technology}.UpdateVersion").newInstance()
41     Result result = dynamicInstance.execute(config)
42
43     PluginLoader.loadPlugin(STEPID, config , PluginLoader.
    AFTER_EXECUTION, result)
44 }
```

Listing A.1: Groovy script for Update Version task in the Build step

Appendix B

Configuration Yaml

```
pipelineOrchestrator: "gitlab-pipelines"
integrationTechnology: "mulesoft"
deploymentTarget: "cloudhub"
artifactRepository: "maven"
sourceRepository: "git"
builder: "maven"

ci:
  releaseBranchPrefix: "release/"
  skipTests: false
  skipStaticCodeAnalisys: false
  skiptIntegrationTests: false

plugins:
  enable: false
  list:
    updateVersion:
      before:
        - "Plugin A"
      after:
        - "Plugin A"
        - "Plugin B"

git:
  username: "CI-Agent"
  email: "ci-agent@deloitte.com"

maven:
  cleanPackage: true
  skipTests: false
  munitEnvVars: ""
  settings: ""

mulesoft:
  skipTests: false
```

Listing B.1: Example of Configuration Yaml