Patterns for DevOps Pipeline Quality

Francisco José Costa Silva

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. António Manuel Ferreira Rito da Silva

Examination Committee

Chairperson: Prof. Miguel Leitão Bignolas Mira da Silva
Supervisor: Prof. António Manuel Ferreira Rito da Silva
Member of the Committee: Prof. Ademar Manuel Teixeira de Aguiar

November 2022

Acknowledgments

A special thanks to Prof. António Rito Silva, this thesis supervisor, since he deserves my special gratitude for his insights, guidance, and knowledge sharing, which made this research possible.

I also want to thank my lovely wife, Margarida, for her unconditional support throughout my life. It was crucial to have her assistance and understanding to complete this work.

I also want to show gratitude to my father and mother for all their investment and dedication to me throughout my life and to my brother for his unconditional support.

This work is dedicated to my eternal friend, Silvestre.

Abstract

Software quality is one of the most important characteristics while the development teams build software products and artifacts. DevOps brought speed and agility and the need to create artifacts with quality to avoid unnecessary future work.

This research proposes a qualitative analysis supported by complexity expressions describing pipeline patterns used by small size development teams driven by two of the most common source management strategies, Trunk-based and Feature Branch.

A Systematic Literature Review was executed to assess the current scientific state of the art in DevOps pipeline patterns and Quality Attributes of pipelines. The Design Research Methodology will support the execution of the qualitative analysis, using the pipeline patterns as artifacts.

In this research, we observe that while using Trunk-Based Driven Pipelines (TBDP), the development team can rely on full automation of the delivery process driven by pipelines with higher efficiency, security, and reliability. This pipeline type also gives the developers a faster delivery of new features promoted by the pipeline's high availability. Using Feature-Based Driven Pipelines (FBDP), the development team has more control over the code integration and can rely on isolating new features in a dedicated pipeline during development due to the pipeline's high suitability for the stakeholders. Due to this pipeline type's high availability, the development team will also count on the capacity to respond quickly to unplanned releases.

Keywords

DevOps; Pipeline Quality; Trunk-Based Development; Feature-Based Development

Resumo

A qualidade do software é uma das características mais importantes enquanto as equipas de desenvolvimento constroem produtos e artefactos de software. O *DevOps* trouxe velocidade e agilidade, mas com isso a necessidade de criar artefactos de software com qualidade para evitar trabalhos futuros desnecessários. Este trabalho de investigação propõe uma análise qualitativa suportada por expressões de complexidade que descrevem os padrões de uma *pipeline* de entrega de software, utilizados por equipas de desenvolvimento de pequena dimensão, orientada por duas das estratégias mais comuns de *Code Source Management*, *Trunk-based* e *Feature-Based*. Foi realizada uma Revisão Sistemática da Literatura para avaliar o estado de arte atual dos vários padrõesjá existentes da *pipeline* de *DevOps* e Atributos de Qualidade das mesmas. A metodologia de invertigação utilizada foi *Design Science Research*, e apoiará a execução da análise qualitativa, utilizando os padrões de *pipeline* como artefactos. Nesta pesquisa, observamos que ao usar uma *Trunk-Based Driven Pipeline (TBDP)*, a equipa de desenvolvimento pode contar com a automação total do processo de entrega através de *pipelines* com maior eficiência, segurança e confiabilidade. Esse tipo de *pipeline* também oferece aos developers uma entrega mais rápida de novos recursos promovida pela alta disponibilidade da *pipeline*. Também observámos que usando uma Feature-Based Driven Pipelines, a equipa de desenvolvimento tem mais controlo sobre a integração do código e pode contar com o isolamento de novos recursos numa *pipeline* dedicado durante o desenvolvimento devido à alta adequação da *pipeline* aos stakeholders. Devido à alta disponibilidade desse tipo de *pipeline*, a equipa de desenvolvimento também contará com a capacidade de responder rapidamente a *releases* não planeadas.

Palavras Chave

DevOps; Pipeline Quality; Trunk-Based Development; Feature-Based Development

Contents

List of Figures

List of Tables

Listings

Acronyms

CI	Continuous Integration
DSR	Design Science Research
SLR	Systematic Literature Review
CDY	Continuous Delivery
CDT	Continuous Deployment
SCM	Source Code Management
FBDP	Feature-Based Driven Pipeline
TBDP	Trunk-Based Driven Pipelines

1 Introduction

Contents

1.1 Motivation

Software is part of life. It is prevalent in companies and daily human activities. This importance in our life implies the need to have software products and services that are reliable, useful for the users, and secure every time we use them [4]. With the increased business needs, the developers identified the need to respond quickly to client's needs and get feedback every time new functionality is deployed in production.

DevOps is a recent approach to Software Development. The process has its roots in the agile development mindset, aiming for fast delivery to customers by bringing the development and operation teams closer. DevOps began in 2008 when Patrick Debois, at the Agile 2008 Conference, mentioned the need for an agile infrastructure and interaction between the development and operations teams [5]. Delivery faster is good, but this new approach leads to quality in the final artifacts the software team will deliver to their clients.

The design decisions defined at the beginning of the Software projects can influence the project's long-term success. In an increasingly more automated software world [6], automated delivery pipeline patterns start to appear, and with them, the need for research studies on the pipelines to understand the suitable pipeline pattern for software project requirements. Sustained planning can contribute to lowering the risk of the project's unsuccess [7].

A recent research survey [8] made to a large number of IT professionals regarding the bests practices when implementing DevOps in one organization shows that emphasizing quality early in the process impacts the success of implementing DevOps.

Several studies already focus on the DevOps impact on software development and software quality [9] [4]. However, only some research studies focus on the study of pipeline quality attributes and their implications.

This research motivation is based on almost four years of experience working with DevOps tools and implementing pipelines. It pretends to create a helpful tool for developers during the Design Phase and contributes to increasing research work targeting the pipeline quality attributes as study artifacts.

1.2 Problem

To the author's knowledge of this thesis, no research was conducted to study the impact of pipeline pattern complexity on pipeline quality attributes. The problem identified for this research problem is:

Problem: *The Assessment of the complexity of pipeline patterns driven by the most common source code management strategies to understand their impact on the pipeline quality attributes*.

1.3 Objectives and Deliverables

Having all the needed and accurate information during the design of a pipeline that will support a software delivery process is very important. A pipeline that corresponds to the requirements and is helpful for the development teams can be before starting development can be achievable, knowing what a pipeline pattern will affect in the long term.

This dissertation aims to propose a qualitative analysis of the pipeline patterns driven by two source management strategies, Trunk-based and Feature-Based. The first part of the research will be the proposal of pipeline patterns that will support the implications of delivering software applications that use the most common source management strategy (Trunk-based and Feature-Based). After that, these models (the pipeline patterns) will be transformed into complex expressions. The interpretation of this complexity expression will support the qualitative analysis. For the methodology chosen to support the study of this research artifact, we will use Design Science Research (DSR).

To sustain research, we first will execute a Systematic Literature Review (SLR) to comprehend the state of the art of the DevOps pipeline by understanding the most consensual pipeline stages of its operations. Then we need to understand the already existing pipeline patterns and their quality attributes.

The research questions are:

- RQ1: What is the impact of DevOps in automating the Software Delivery process?
- RQ2: What operational stages define the automated software delivery pipeline that supports the DevOps approach?
- RQ3: What are the existing design patterns and quality attributes of an automated software delivery pipeline that supports the DevOps approach?

1.4 Thesis Outline

Our research document is divided into six chapters. Chapter 1 will exhibit the motivation, research problem, objectives, and deliverables. The literature review performed before this research is disclosed in Chapter 2, highlighting the motivation, research questions, review protocol, and results. In Chapter 3, a theoretical background on the following themes: DevOps, DevOps Pipeline, Software Configuration Management, and Quality Attributes. In Chapter 4 is disclosed in detail the Design Phase belonging to Design Science Research (DSR) methodology, where pipeline patterns are defined and discussed. Chapter 5 delivers the Demonstration and Evaluation phases of the DSR, which reflects the transformation of the pipeline pattern diagrams into complexity expressions that will support the qualitative analysis. Chapter 6 closes the research document structure and presents the Research Conclusions, System limitations, and Future Work.

2

Research Methodology

Contents

A qualitative study of the pipeline patterns driven by the Trunk-based and Feature-Based Source Management strategies is the primary goal of this thesis. In order to be able to study the research artifacts properly, in this chapter, we will define the Design Science Research (DSR) used in this thesis and expose the baseline research work done using a SLR to identify any related existing studies.

2.1 Design Science Research

Firstly suggested by March and Smith in the article "*Design and natural science research on information* " [10] and more recently by Henver et al. [11] , Design Science Research (DSR), is traditionally used in Information System related research. This research methodology qualifies the conducting of studies by understanding the domain and testing "new and innovative artifacts" [11].

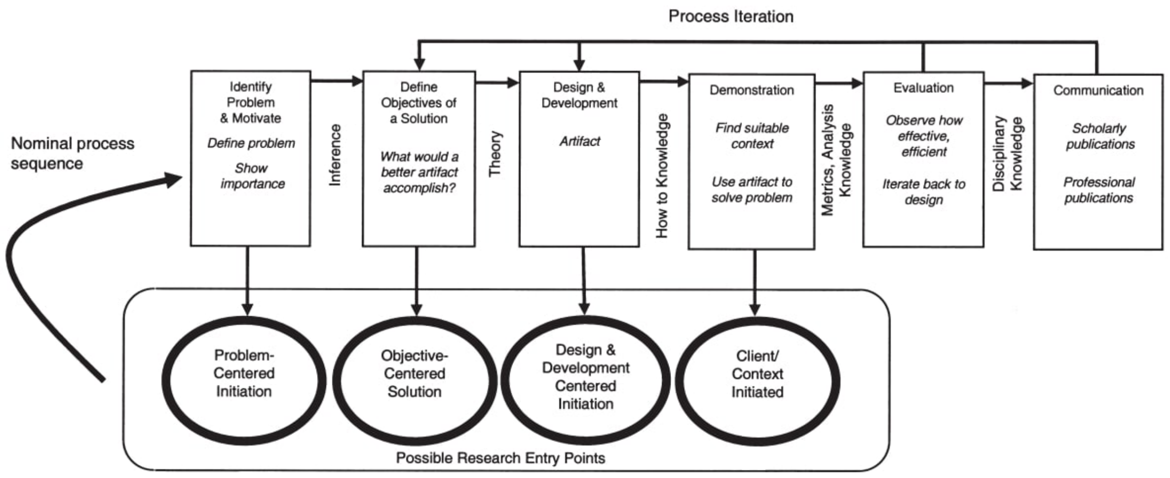

Process Iteration
Nominal process sequence
Identify Problem & Motivate
Define problem
Show importance
Inference
Define Objectives of a Solution
What would a better artifact accomplish?
Theory
Design & Development
Artifact
How to Knowledge
Demonstration
Find suitable context
Use artifact to solve problem
Metrics, Analysis Knowledge
Evaluation
Observe how effective, efficient
Iterate back to design
Disciplinary Knowledge
Communication
Scholarly publications
Professional publications
Problem-Centered Initiation
Objective-Centered Solution
Design & Development Centered Initiation
Client/ Context Initiated
Possible Research Entry Points

Figure 2.1: Figure representing the phases of DSRM process reprinted from [1]

Peffers et al. [1] conferees the following six steps to execute a DSR:

1. Problem identification and motivation,
2. Definition of the objectives for a solution,
3. Designing and development of artifact,
4. Demonstration of the use of the artifact to solve one or more instances of the problem,
5. Evaluation by observing and measuring how well the artifact supports a solution to the problem,
6. Communication of the problem and its importance, the artifact, its utility and novelty, the rigor of its design, and its effectiveness to researchers and practicing professionals.

Henver et al. [11], in the study entitled ”*Design Science in Information System Research*” suggests seven guidelines for the development of DSR in Information Systems Research:

1. Design as an artifact - producing a viable artifact that can be constituted using a construct, a model, a method, or an instantiation providing the vocabulary and symbols to defined problems and solutions

2. Problem relevance- the solution should be technology-based and relevant for business problems, which can be defined as the differences between the goal state and the system’s current state.

3. Design Evaluation - The three dimensions of the artifact target for the research, utility, quality, and efficacy, should be rigorously demonstrated. The Evaluations methods can be categorized as Observational, Analytical, Experimental, Testing, and Descriptive.

4. Research Contributions - The design research should contribute in a verifiable manner to the artifact design functionalities and/or methodologies.

5. Research Rigor - The design research should apply rigorous methods in constructing and evaluating the artifact targeted by the research.

6. Design as a Search process - while searching for a compelling artifact, we should utilize all the means to accomplish the ideal outcomes while fulfilling the laws in problems solving.

7. Communication of Research - The presentation of the Design-science research results should be adequate for technology and management-oriented audiences.

The DSR methodology steps, illustrated in figure 2.1, will be used in this research to support the qualitative study of the pipeline patterns driven by the Trunk-based and Feature-Based Source Management strategies.

The *Problem Identification, Motivation, and Research Objectives* reflected in Section 2 are supported by the initial work executed using the procedures suggested by Kitchenham et al. [12] to perform a Systematic Literature. Chapter 3 presents the *Design and Development* of the artifact disclosure in-depth research to support all the concepts supporting the pipeline patterns. Chapter 4 presents the *Demonstration of the Artifact*, exposing an in-depth study of the principal Trunk-based driven and Feature-Based driven pipelines and their suggested pipeline ramifications using the simulation method. The *Evaluation* will be revealed in Chapter 5 using illustrative scenarios described by complexity expressions representing the artifact models presented in Chapter 4. The *Research Results* will be communicated via submissions to journals and conferences.

2.2 Systematic Literature Review

Systematic Literature Review (SLR) is defined as “a means of identifying, evaluating, and interpreting all available research relevant to a particular research question, or topic area, or phenomenon of interest” [12].

Our SLR was performed with Kitchenham’s Procedures for Performing Systematic Reviews guidance. The procedures used are [12]:

- Planning – In Section 2.2.1 we expose the motivation to perform this research, in Section 2.2.1 the specification of research questions, and in Section 2.2.1 the review protocol.
- Conducting – presented in Section 2.2.2 of this document is the application of the review protocol, resulting in the primary studies and data extraction.
- Reporting the review – this is the last phase of the review. Section 2.2.3 summarizes the extorted information for the selected studies.

We chose to perform this SLR because we needed to summarize state-of-the-art information regarding DevOps Pipelines Structure, existing Patterns, and the Software Quality of the DevOps pipelines. The result o the SLR will help support the Problem, Motivation, and Research Objectives.

2.2.1 Planning

Motivation

DevOps is a relatively new approach to software development. Due to the benefits of this new approach, companies have massively started using it. [6] A recent research survey [8] made to a large number of IT professionals regarding the bests practices when implementing DevOps in one organization shows that emphasizing quality early in the process impacts the success of implementing DevOps. Several studies already focus on the DevOps impact on software development and software quality [9] [4]. However, only some research studies focus on the study of pipeline quality attributes and their implications. We pretend to create a helpful tool for developers during the Design Phase and contribute to increasing research work targeting the pipeline quality attributes as study artifacts.

Research Questions

- RQ1: What is the impact of DevOps in automating the Software Delivery process?
- RQ2: What operational stages define the automated software delivery pipeline that sup- ports the DevOps approach?
- RQ3: What are the existing design patterns and quality attributes of an automated software delivery pipeline that supports the DevOps approach?

Review Protocol

The first step identified in Kitchenham's procedures for defining the review protocol [12] is to create a search string to retrieve the maximum number of studies related to our proposed research questions. All the steps of the review protocol defined are illustrated in Figure 2.2.

The **Search Strings** were: *DevOps AND (patterns OR practices)*; *DevOps AND Software Delivery*; *Devops AND (Quality OR Quality Attributes) AND Software*

The **Data Sources** where the search string was performed were: ACM Digital Library, IEEE Digital Library, Science Direct, Books, ArXiv, Springer Link, and Wiley Online Library.

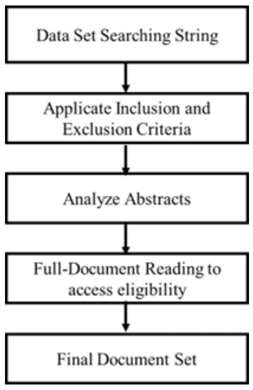

Data Set Searching String
Applicate Inclusion and Exclusion Criteria
Analyze Abstracts
Full-Document Reading to access eligibility
Final Document Set

Figure 2.2: Ilustration of the Review Protocol

The second step is to define the inclusion and exclusion criteria and apply them to the returning articles from a search string. Both criteria are illustrated in Table 2.1.

Inclusion Criteria	Exclusion Criteria
Articles from Journals and Conference Proceedings that relate DevOps and Quality in software Products	Different Subject
Articles from Journals and Conference Proceedings about DevOps approach	No references
Books related to DevOps Approach	Only full Documents
Books related to Software Products	Studies Not Related with DevOps
Thesis or scientific articles where the subject is related to DevOps	No duplicates
Thesis or scientific where the subject is Related to Quality Attributes or Software Quality	Unable to get full documents
Conference Proceedings, and Books	Non-scientific articles from Journals, Conference Proceedings and Books

Table 2.1: Inclusion and Exclusion Criteria

2.2.2 Conducting

Selection of Studies

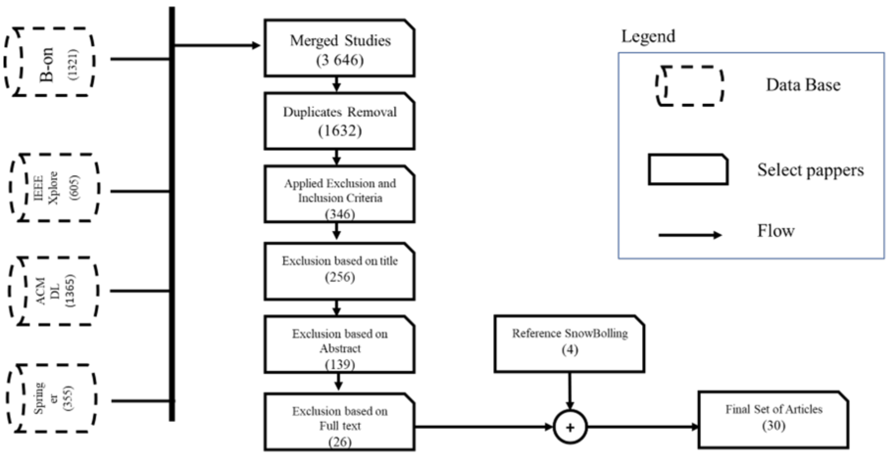

B-on (1321)
IEEE Xplore (605)
ACM DL (1365)
Springer (355)
Merged Studies (3 646)
Duplicates Removal (1632)
Applied Exclusion and Inclusion Criteria (346)
Exclusion based on title (256)
Exclusion based on Abstract (139)
Exclusion based on Full text (26)
Reference SnowBolling (4)
+
Final Set of Articles (30)
Legend
Data Base
Select pappers
Flow

Figure 2.3: Illustration of the Studies Selection process

With the search string int datasets defined in the review Protocol, we found 3646 artifacts. After applying the inclusion and exclusion criteria to the title and abstracts and removing the duplicates, we ended with 139 articles. Each of the 139 articles was read, getting 26 articles relevant to our research. We also added four more that we reached using Reference Snowballing. Figure 2.3 illustrates the studies selections process.

Data Extraction Analysis

In Data Extraction Analyses, we will present the analysis of different parameters of the selected studies, such as their data sources, this distribution over the years, and a Tag Cloud, which shows the most relevant terms from a list of titles.

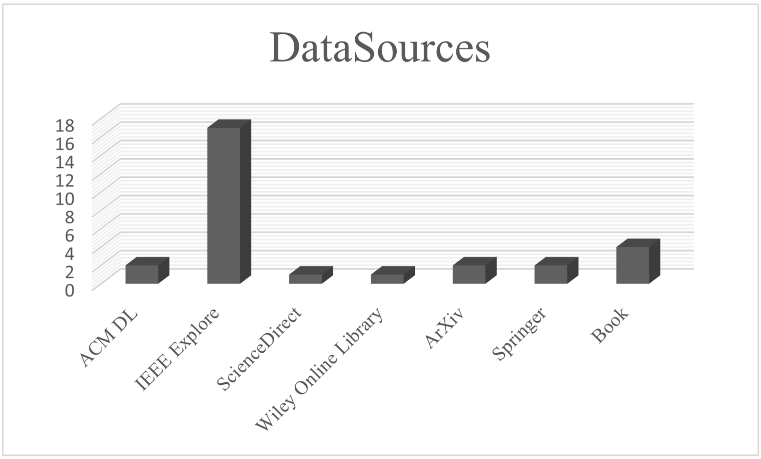


Figure 2.4: Data Sources Bar Graphic

By observing Figure 2.4, we notice that most select studies were from IEEE Explore, followed by the Books. After that, Springer followed with the same number, ScienceDirect, ACM DL, and ArXiv, finishing with the Wiley Online Library.

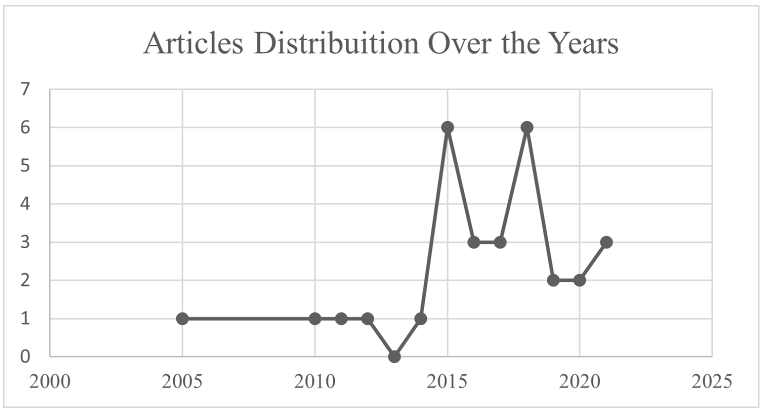


Figure 2.5: Articles Creation Date Distribution

In figure 2.5 is possible to observe the distribution of studies over the years. 2015 and 2018 are the years from which more studies were selected for this research.

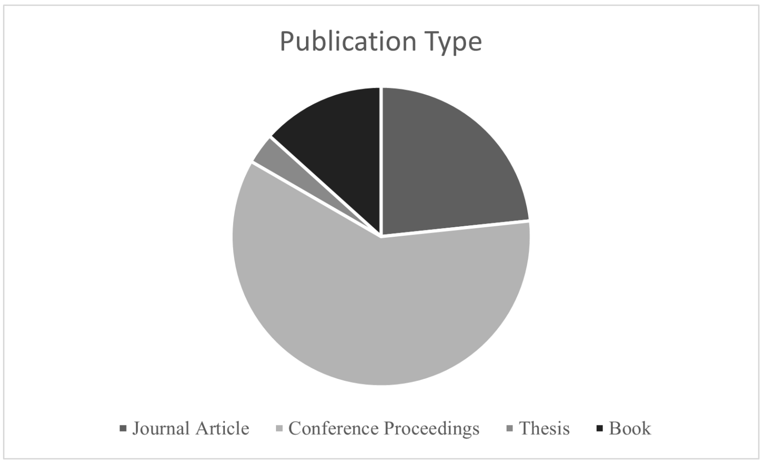


Figure 2.6: Publications types

The most common type among the selected papers in the Conference Proceedings is fifty-eight percent of documents. Papers from journals Articles twenty-nine percent, books twelve percent, and thesis 3 percent were also determined. This distribution I represented in 2.6.

Figure 2.7: Tag Cloud

Figure 2.7 represents a tag Cloud that gives the most relevant terms from a list of titles of the final set of selected papers. The most relevant terms are, with no surprise, “Software,” “Continuous,” “DevOps,” and “Delivery.”

2.2.3 Reporting the Review

RQ1 - What is the impact of DevOps in automating the Software delivery process?

The DevOps approach can help automate the software delivery process. DevOps software delivery automation can be separated into three concepts: Continuous Delivery, Continuous Integration, and Continuous Deployment. Continuous Delivery (CDY) enables operation teams to interactively deliver available software artifacts in a short cycle of time to ensure the software is always in a release state while daily, developers make thousands of changes [13]. Continuous Integration (CI) is the technique that continually merges artifacts, including source code updates from all developers on a team, into a shared mainline to build and test the developed system . [14] Continuous Deployment (CDT) is the automated process of deploying changes to production [14]. CI, CDY, and CDT consummations are called Automated Deployment Pipeline [15]. Four main quality attributes define a DevOps automated pipeline: Continuous Testing, Code Review, Release engineering, and Deployment [16]. Continuous Testing. This characteristic represents a software product’s quality validation, and verification [17]. The pipeline is responsible for executing the tests previously developed by the developers in automation testing tools [16]. Code Review. This characteristic assures the quality of the code written by developers. The automated build of the code delivery process has a notable impact on code. Review participation improves and brings a higher percentage of code reviews [18]. This also increases the involvement and responsibility of developers in the software life-cycle. Release Engineering is a software engineering discipline that develops, implements, and optimizes processes to deploy high-quality software reliably and predictably [19]. This characteristic ensures traceability of changes reproducibility of build, configuration, and release while maintaining a revertible repository of all changes. [16].

RQ2 - What operational stages define the automated software delivery pipeline that supports the DevOps approach?

According to an empirical investigation from Mojtaba Shahin et al. [20], version control, build, unit/integration testing, continuous integration, and production deployment are the most common deployment pipeline stages. Toh et al. [21] quantitative survey introduces three more stages for continuous delivery: delivery to staging, automated testing, and release.

Version Control. This stage, according to Nicolás Paez [22], can be defined considering four specific concerns:

- Artifacts under Version Control. Some examples are the application's source code, the infrastructure's source code, the configuration files, the deployment files, or the application's binary code. Sometimes, the environment variables can be added to the previous list.

- Tools for version control. We must choose a Source Code Management (SCM) tool to store and manage your files. We should separate the source code into three types: Secrets, Binary Artefacts, or Plain Text. We must choose an SCM tool that encrypts the information saved for secrets. In Binary files, we must use the Artifactory repository's

- Identification of versions. We must define Semantic Versions rules to name versions/tags to identify each version.

- Relationships between artifact versions. We must set a traceability strategy if we have one or more source control repositories. An option could be a Tracker Repository.

Build. The build stage generates an executable and testable system from source versions, or baselines [14]. This stage is automated by utilizing build tools that manage the software development and service life cycle. This may involve compiling code, managing dependencies, generating documentation, or running tests. [23].

Unit/Integration Testing. This stage automates the build executable artifacts tests [24]. These tests can be for functionality, integrity, consistency, performance, security, or non-functional requirements. It also enables the creation of quality gates that are defined and implemented to get faster feedback from tests over the whole pipeline up to production environments. [25]. It is intended to share within component teams and integrated beyond component boundaries at the product integration level, validating how the code behaves continuously [26].

Delivery to Staging – Staging is where we can test and experiment with new features merged into the system before releasing them to the Production Environment. The data collected in this stage can be used to prevent software errors. [27].

Automate testing – This Stage main component is the automation tools used to test the application. The test activities in this stage can be test case design, test scripting, test execution, test evaluation, or test result reporting. [28]

Release – In this stage, the organization's teams typically have a release engineer, whose primary responsibilities are dealing with the source code branching and managing source code merges between branches. This stage can be done manually or automatically. [29]

Production Deployment. This is one of the most crucial stages in a DevOps approach pipeline. There are three specific concerns we must take in this stage [24]:

- Early Release testing. Some testing methods are Beta Release, Canary Testing, and A/B testing.
- Error detection. Errors can be either functional or nonfunctional. To identify non-functional mistakes, we can include some indicators of poor behavior in a monitoring system.
- Live Testing. While monitoring is passive, live testing intentionally perturbs the running systems. This method is commonly called Chaos Monkey.

RQ3 - What are the existing design patterns and quality attributes of an automated software delivery pipeline that supports the DevOps approach?

The pipelines supporting the DevOps approach can uniquely reflect the organization's needs. However, with the growing market adoption of automated software delivery, some design patterns started to appear. Typically these patterns reflect the driven goal of the different pipelines.

A report from the software product provider Harness [30] proposes eighth pipeline patterns. With a brief description, the pipeline patterns proposed are:

- *Button Pushing* supports manual approval at the finish of the delivery process.
- *Test Automation* supports the automation of all the code tests
- *Multiservice* supports the automated deployment of multiple artifacts with only one pipeline
- *Multiservice Environment* supports the automated deployment of multiple artifacts with only one pipeline in multiple environments
- *Semi Approval* supports a mix of automation and human approval
- *Full approval* supports full automation of the software delivery

- *Gitops* is the fattest pattern for developers to deploy code changes to environments, and it supports an automated software delivery process without making changes via UI or CLI, being git the only source of true

An article from the software provider Red Hat [31] provides differentiation in two groups of pipeline design patterns, one for application developers and the other for infrastructure operators. *Applications developer's* pipeline pattern supports software delivery automation, starting with a checkout from Source Code Manager, build operations, and delivery in a Virtual Machine. *Infrastructure Operating* pipeline pattern supports the automation of the clone of all the infrastructure files from the Source Code Manager and all operations related to the provisioning or update of the infrastructure.

An article from the online community DZOne [32] suggests seven pipeline patterns that support Continuous Delivery. With a brief description, the proposed pipelines patterns are:

- *Pipeline as code* supports the codifications of the pipeline logic. The resulting code can be stored in the Source Code manager.
- *Externalize Logic into Reusable Libraries* supports using common pipeline logic that can be referenced in multiple pipeline codes.
- *Separate Build and Deploy Pipelines* support the logical separations between a Build pipeline and a deploy pipeline.
- *Trigger the Right Pipeline,* support the trigger pipeline behavior affected by Branch commits, pull requests, and merges to the mainline
- *Fast Team Feedback* supports the trigger of multiple pipelines and omnichannel notifications.
- *Stable Internal Releases* support the deployment of only versioning packages produced by the pipeline supporting the build of automated operations.
- *Buttoned Up Product Releases* support the deployment and audibility of tagged releases to production.

The pipeline quality can be measured by evaluating the pipeline suitability for stakeholders and the Maintainability, Availability, Performance Efficiency, Reliability, and Security attributes [24] [6].

3 Theoretical Background

Contents

In this Chapter, we introduce a more in-depth description of DevOps, DevOps Pipeline, Software Configuration Management, and Quality Attributes

3.1 Devops

DevOps is a collaborative framework that stresses empathy and cross-functional collaboration within and between software development teams to operate resilient systems and accelerate the delivery of changes in an organization [19]. This framework's main elements are Collaboration, Automation, Measurement, and Monitoring [33].

The implementation of DevOps in practice, according to Macarthy et al. 2020 [34], could describe the collaboration between:

- Developers and DevOps teams based on automation;
- Mixed responsibilities of developers without an IT Ops team;
- Developers working with DevOps teams only;
- DevOps teams serve as a bridge between developers and IT Ops teams.

These collaborations between teams help mitigate some problems in the software industry by leading to less fear of change in stakeholders, decreasing the risks of deployment, distributing the responsibility between all teams involved equally, adding feedback loops between developers and operations teams, and reducing silos in the organization [35].

The foundation of DevOps approach practices is automation in standard software Life Cycle steps like development, delivery, deployment, and operations. This automation enables the latency reduction of product releases [36] [26].

3.1.1 Feature Flags

During the automation process, the team must control the feature enablement in different environments. So the team creates a configured file with Feature Flags to achieve that control. The Feature Flags are variables used in a conditional statement to guard code blocks, aiming to enable or disable the feature code in those blocks for testing or release [37].

This approach can be applied in the pipeline using a configuration file that stores and loads the values feature flags introduced by the developers to memory. Each environment has a configuration file. Taking that value into account and using simple if-then instruction, it is decided in run-time whether the feature is active or not. If this solution does not comply with the needs (high number of feature toggles), the team can use other options like a Feature Flag management software or a Feature Flag service.

The Feature Flags can be applied to all application environments, from Development to Production.

3.1.2 Quality Gates and Triggers

To control the workflow of the automated deployment pipelines, the development teams can configure Quality Gates and Triggers.

The development team can use Gates to control the deployment in different environments. There are two condition types of gates: pre-deployment and post-deployment gates. The pre-deployment guarantees no problems in the management system or that all requirements for the deployment are respected. The execution of this gate type happens before the deployment. [38]

Post Deployment gates ensure that incidents from monitoring or incident management systems do not exist before the deployment to the following environments.

Triggers automate the start of the pipeline operations in response to actions or schedule runs (specific times and frequencies). [39]

3.1.3 Tools

The DevOps tools can be aggregated into four domains: Log Monitoring, Resource Monitoring, Build Automation, and Configuration and Deployment Automation. [40]

Log monitoring tools automate the analysis of the logs collected from the system performance and the process running. The developers can also use this type of tool in the debugging processes.

Resource Monitoring tools can be classified as System and Network Monitor tools. System Tools automate, aggregate, and produce graphs from data collected from the system performance and resource usage. Network Tools will monitor the network devices and automate notifications or actions to solve emerging problems.

Build automation tools, and support the automation of the built phase, more precisely, the linting and compiling operations and all software test types' execution.

Configuration and Deployment tools allow the team to automate the system configuration and deployment process.

3.1.4 Team Roles

The DevOps team's primary goal is to close the operations team to the development teams.

The DevOps team structure can have multiple alternatives. Interdepartmental Dev and Ops collaboration, Interdepartmental Dev and Ops team, Boosted (cross-functional) DevOps team, and Full (cross-functional) DevOps team are four suggested team structure types. [2]

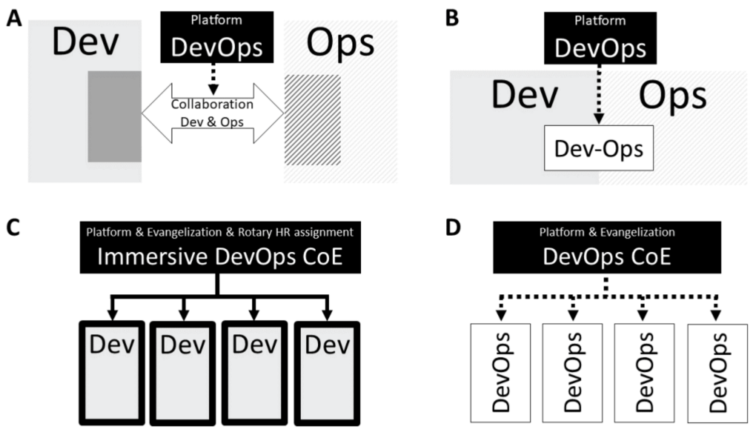


Figure 3.1: DevOps team Roles Diagram reprinted from [2]

In the Interdepartmental Dev and Ops collaboration team structure, represented in 3.1 with the letter A, the developers and operations belong to different departments and collaborate sporadically on a specific project.

The Interdepartmental Dev and Ops team, represented in 3.1 with the letter B, is a stable team that will work on ongoing projects. However, the developers and operations are kept in different departments.

Boosted DevOps team is a team structure, represented in 3.1 with the letter C, where DevOps experts uplift the development teams until they can have full autonomy to apply the DevOps Values and operate the DevOps Tools.

Full DevOps, represented in 3.1 with the letter D, is reached when the team was full autonomy to apply the DevOps Values and operate the DevOps Tools.

3.1.5 Principles

The DevOps approach has four core principles [14]:

- Business or mission first. Focus on the business needs ahead of procedural and technical considerations. This principle helps balance risks and activities to provide the most value to the customer;
- Customer Focus. We apply the right approach if the strategy makes sense to the customer and meets a consumer's need;
- Left shift and continuous everything. Continuous everything meant using the same practices in development and operations teams and maintaining them. In practice, the left shift means lower risks deployments by applying more rigorous release procedures;
- Systems thinking. The approach encourages all members involved to understand the systems from end to end completely.

3.2 Software Configuration Management

3.2.1 Git Operations

To support all interactions with the repository in the Build phase, the development team uses git operations.

Git is a version control system used to manage and track the changes in the source code. The Git repository comprises a local repository that resides in the developer's machine and a remote repository hosted in a Remote Server accessible to all team members. [41]

Every local copy of a version archive is managed through local branches. Remote branches are visible to users with access to the remote repositories. Checking out a branch, making a local change, and committing.

A branch is a foundational mechanism in git. The local copy of the code base is managed using local branches. Remote Branches are available to all users with access to the repository. The mainline (or master) defines the stable code branch where the development team can execute the operations from the Build Phase without problems or blockers. Git branches are a pointer to a specific commit. This pointer moves along with each new commit made by the development team. All git operations can be automated using DevOps Tools and contribute directly to Continuous Integration. [42]

The most typical operations used are [43]:

- Git Push - Operation that uploads the code references and objects to the remote repository.
- Git Merge - Operation that compares and merges the changes between two branches and detects the existence of code conflicts.
- Git Clone - Operations that download the code version from the remote server to the local environment.
- Git Fetch - Operation that synchronizes the branch references and objects from the local repository with the same branch in the remote repository without changing anything in the remote repository.
- Git Pull - Operation that combines a git fetch followed by a git merge in the local environment. The term ”Pull Request” is commonly used when the code contributor asks the remote repository maintainer to review the code generated in the local environment before merging it into the remote repository.
- Git Commit - Operation where all stage changes in the local environment are ready to be pushed to the remote environment.
- Git Fork - Operation that creates a copy of the repository without affecting the upstream repository.

3.2.2 Branching Patterns

In the article *Patterns for Managing Source Code Branches*, published by Martin Flower [3], the author correlates the branching strategies with Continuous Integration, Continuous Deployment, and Continuous delivery. The article gives the following Branching Patterns definitions:

Trunk-based or mainline is a branching pattern where the developer team has one branch (mainline), and as soon they have a healthy commit with new code changes, they integrate them into the mainline. The team can use feature flags to disable unfinished features in the healthy commits. This branching pattern allows the team to have high integration frequency and increases the frequency of merges but reduces their complexity and risk, see Figure 3.2. The high frequency contributes to Continuous Integration.

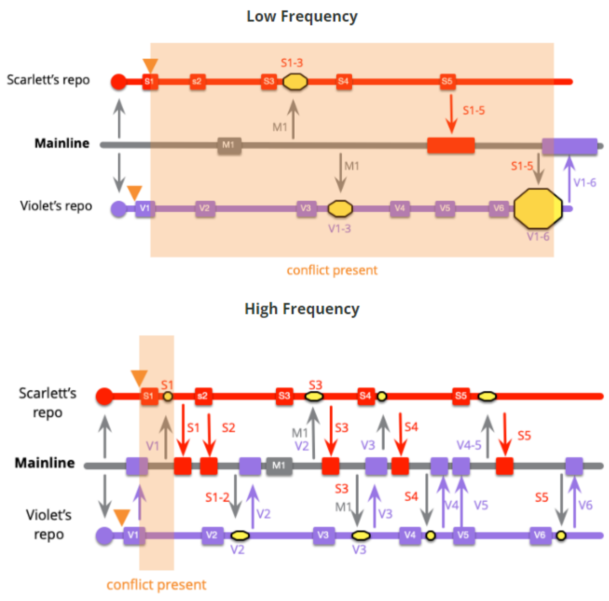


Figure 3.2: Low Frequency vs High-Frequency Integration. Reprinted from [3]

Feature-Based is a branching Strategy where the developer team creates a different branch, and all the changes are pulled to the created branch while developing a new feature. The code base is integrated with the mainline when the feature is complete. This branching pattern is popular in the industry, but the low integration frequency, see Figure 3.2, due to the length of feature branches can increase risk and complexity.

Release Branch is an alternative branching pattern based on Feature-Base. This branch only receives commits to stabilize a version of a product ready for release. This pattern is practical if the team cannot maintain the mainline in a healthy state.

Maturity Branch is an alternative branching pattern based on a Feature-Based strategy, whose head marks the latest code-based version. With this, every team member can access the most stable version of the code.

Environment Branch is an alternative branching pattern based on a Feature-Based strategy. This pattern allows the development team to have a branch containing commits that apply the source code that rearranges the product to run in different environments. A separate branch is created for each environment.

Hotfix Branch is an alternative branching pattern based on a Feature-Based strategy that allows the team to seize all the needed work to fix an urgent production defect. After the fix is applied, it is integrated into the mainline.

Release Train is an alternative branching pattern based on a Feature-Based strategy that allows the team to have regular schedule releases. Each branch created is associated with a different release. After the scheduled time, the branch doesn't suffer more modifications.

Release-Ready Mainline is an alternative branching pattern based on the Trunk-Based strategy, which requires having the mainline healthy enough so the code-based can be deployed directly in production environments. This branching strategy allows for coupled Continuous Integration and Continuous Delivery.

Git-flow is based on a Feature-Based strategy and is one of the most used branching patterns. This pattern is a compilation of most of the patterns mentioned above. It uses the mainline (named develop), feature branches to allow the developers to develop their features separately, a production maturity branch (called master), a hotfix branch, and a release branch. The mainline is the origin of the repository. All the feature branches are cloned from the develop branch when created and integrated with the mainline when finished. The code base present in the mainline is integrated into the release branch. When it is ready for Production, the code is merged from the release branch into the master branch. If a hotfix is needed, the team creates a hotfix branch.

3.2.3 Build Phase and Testing Phase

For this research, the *build operations* refers to transforming a high-level programming language (understandable and written by humans) into a low-level binary language (understood by the computer).

The code compilation is the first operation from the build phase. However, a code repository must be in place to execute this operation. We will explore more around this topic further. The *compilation* operation defines when a computer program translates and tests computer code written in one programming language (source) into another language (target). [44]

Before the code compilation, the *source code can be lint* (an operation that automatically checks the source code for programmatic and stylistic errors).

The *build environment* (where the build workflow operations are executed) can be separated into a *Local Build*, where the system boundary is the developer machine, and *Remote Build*, where the system boundary is a remote server. Two types can also define the *build execution*: *Full Build*, where the code is complied like was the first time, and *Incremental Build*, which only generates the new code changes (by comparing to a previous build).

The development team runs all the necessary code tests in the *Testing phase*. There are four types: Unit, Integration, Functional and Non-Functional.

The *Unit Testing* defines the code tests of total isolated software pieces without dependencies (A single function or a Single class) [45].

When software pieces depend on others, the development team uses *Integration Testing*. It is also an aggregation test strategy for various services or software pieces. To test these dependencies, the team considers all possible side effects when they start writing them.

To validate the code against the function requirements of specifications made by the software client, the team uses *Functional Testing* [46].

The *Non-functional Testing* validates the operation of systems like performance, reliability, scalability, or load capacity rather than code-specific behaviors [46].

Before releasing the code in a Production Environment, the development team can execute a last set of operations to guarantee that we release high-quality code.

End-to-End (E2E) testing is a set of operations that allows validating all software product components, from the origin to the end, to ensure that the application flows are conducted as expected. With these testing operations, the team can simulate a real user scenario, validating if all components integration pieces are working as expected and if there is data integrity. There are two methods to execute E2E testing, horizontal where the testing is executed horizontally across the context of multiple applications, and vertically, where tests happen in all architecture layers of a single application from top to bottom [47].

The tasks workflow to execute end-to-end testing are:

1. Test Planning - where all tasks and resources needed are defined;
2. Test Design - specifying tests, test cases, and risk or usage analysis;
3. Test Execution - execution of all tests;
4. Results Analysis - interpret and evaluate all the test results and perform additional tests if necessary.

The *Continuous Testing* defines the Process of automating the code tests mentioned above. *Continuous Integration* is defined by the creation of automated interactive builds and running of automated software tests. [24]

In short, the build phase aggregates the following operations workflow:

1. Find the source code from the code repository;

2. Use a tool to lint and compile the code;
3. Debug the code generated and check dependencies;
4. Run source code tests.

3.2.4 Deployment Phase

After the build phase, the code repository receives the generated executable code.

The deployment phase aggregates the operations that will allow the development team to retrieve the executable code from the repositories and move from one controlled environment to another.

According to *ITIL 4 practices guides* [48], an environment is: *A subset of IT infrastructure for a particular purpose.*

The most common environments are:

- *Development Environment*: where the developers build the code produced by them and run the unit tests. This code build can be a Local Build or a Remote Build;
- *Integration Environment*: is where all the code from the different team developers is integrated, and all the integration tests are executed;
- *Testing Environment*: where all the functional and non-functional tests are executed;
- *Staging or Pre-Production Environment*: this is where the code tests use real data, but the environment is unavailable to the software client;
- *Production Environment*: where the code runs with real data available for the software clients.

A Container is an operating system (OS) level virtualization that encloses standard OS processes and their dependencies. They are typically used to support the development and deployment of the applications in the infrastructure (A different container supports each environment) [49]. This virtualization helps reduce the time between developing the application code and production deployment and increases environment standardization. [40]

The automation process of the code transition between all the environments before Production is commonly known as **Continuous Delivery**.

3.2.5 Release Phase

According to the *ISO/IEC 20000* [50], a release is:

A collection of one or more new or changed services or service components deployed into the live environment as a result of one or more changes.

In a few words, the release is the operation where features and services are available to the software clients.

The release can be characterized by Major and Minor, which describes the significance of the change in the code. A Release can also be planned or not Planned.

The release by type can be defined by the following;

- *Emergency release or hotfix* defines an emergency deployment of software packages to solve performance or a security issue. (Not Planned);
- *Maintenance releases*, to deploy code packages to fix bugs or patches (Planned);
- *Feature release*, for deploying code packages for new changed functionality (Planned).

The automation of the entire process of Releasing code is commonly known as Continuous Deployment.

3.3 DevOps Pipeline

Software Development Life Cycle 3.3 is the application of standard business practices to support the building of software applications. These business practices can be automated using tools that help to apply the DevOps Approach.

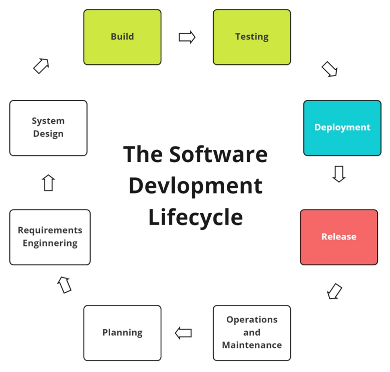


Figure 3.3: SDLC standard business practices relation representation

These standard business practices are Planning, Requirement Engineering, System Design, Build, Testing, Deployment, Releases, and Operation/Maintenance.

Figure Caption

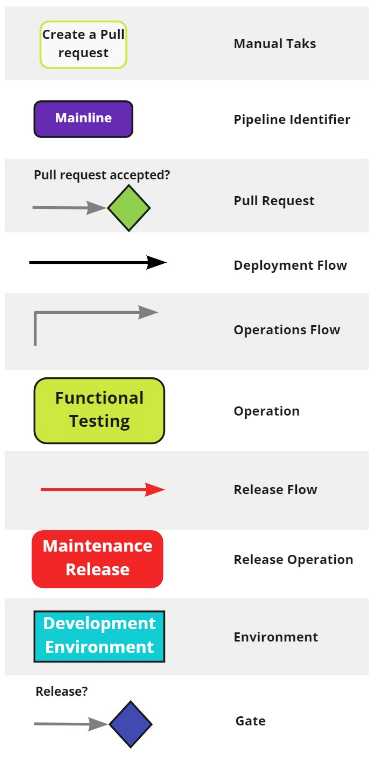


Figure 3.4: Caption for all diagram representations

Figure 3.4 illustrates the caption for all figures used in the pipeline diagram presented in Chapter 4. The black arrow with the green diamond represents pull request triggers. The blue diamond represents the Approval gates in the pipeline. The green blocks illustrate all automated operations in the pipeline, and the red ones are the operations related to the Release. The white rectangles with green borders represent the manual operations The blue blocks represent the different environments. The back arrows display the environment deployment flow, the gray ones show the operations flow in the pipeline and other pipelines, and the red represents the release flow. The purple rectangle represents the pipeline identifier.

3.3.1 DevOps Standard Pipeline

A pipeline in DevOps is composed of a set of stages executed based on a Trigger. These stages have specific responsibilities. Every stage contains multiple tasks executed step by step. [51]

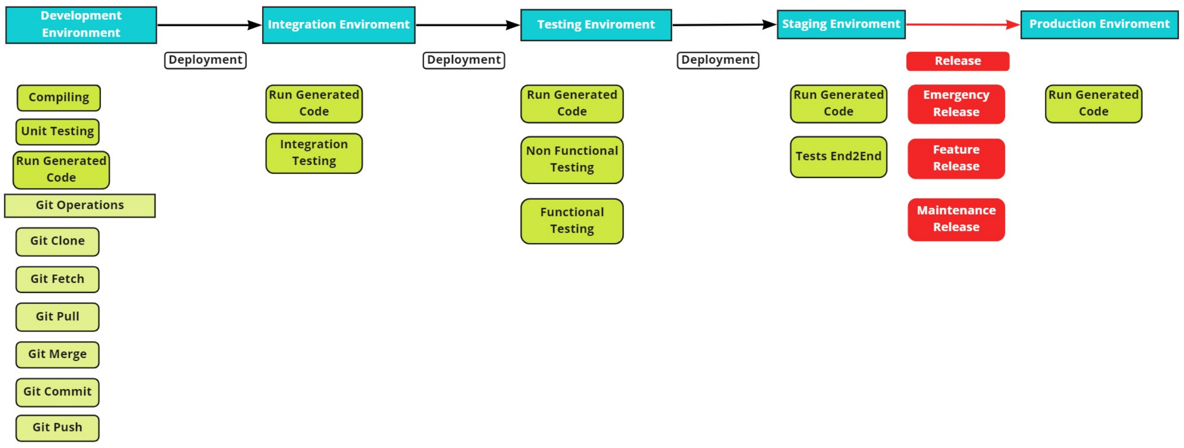


Figure 3.5: Illustration of a Standard Pipeline

For the context of this research, we proposed a standard pipeline representing standard operations of the Build, Deployment, and Release phases. With this, we assume that this standard pipeline supports the development team to achieve Continuous Integration, Continuous Testing, Continuous Delivery, and Continuous Deployment. We also assume that the code base uses only one branch (mainline) and that the pipeline execution runs on a remote server. Figure 3.5 illustrates the Standard DevOps Pipeline.

3.4 Quality Attributes

3.4.1 Software Quality Attributes

A quality attribute is a measurable or testable property of a system that indicates how well the system satisfies the need of its stakeholders [52]. The ISO 9126 model introduced two types of quality attributes [53]:

Internal quality attributes – refers to system properties evaluated without executing them.

External quality attributes - refers to system properties that can be accessed by observing during execution. The ISO 25010 model was an update to ISO 9126 indicates a subdivision of the quality attributes into eight key characteristics [54]:

- *Functional Suitability* - the degree to which a product or system provides functions that meet the stated or implicit requirements when used under specific conditions;
- *Reliability* - The degree to which a system, product, or component performs specified functions under specified conditions for a specified time frame;
- *Performance Efficiency* - this characteristic represents the performance relative to the number of resources used under stated conditions;

- *Operability* - Technical Learnability and Accessibility;
- *Security* - The degree to which a product or system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization;
- *Compatibility* - Degree to which a product, system, or component can exchange information with other products, systems, or components, and/or perform its required functions while sharing the same hardware or software environment;
- *Maintainability* - Degree of effectiveness and efficiency with which a product or system can be modified to improve it, correct it or adapt it to changes in the environment and requirements;
- *Transferability* - Degree to which a product or system can effectively and efficiently be adapted for different or evolving hardware, software, or other operational or usage environments.

3.4.2 Pipeline Quality Attributes

When a DevOps pipeline is responsible for automating the deployment of new features to a system, the developers have in mind the quality attributes of the pipeline to assertively choose the best pipeline pattern that suits the team's needs. .

We can evaluate the DevOps pipeline quality attributes in two ways— the *Pipeline Suitability* for the stakeholders and with software quality attributes that can also be associated with the pipeline. (a full automated pipeline is composed of a set of automated software tasks). [24]

Automated monitoring, automated testing, and code review automation are DevOps activities used to integrate security in the pipeline [55]. We can measure the *Pipeline's Security* by evaluating the efficiency of the tasks related to the activities mentioned. The existence of credentials/tokens in the code and using high privileges running the commands can decrease the pipeline's security. [6]

The *Reliability* and *Availability* evaluate DevOps pipeline delivery mechanism [24]. To measure *Reliability*, the team must consider the probability of having errors when the feature is delivered. If the delivery mechanism fails, the time required to deliver a new feature increases, affecting the *Availability*.

The DevOps pipeline definition strategy impacts their *Maintainability*. Typically, the pipeline is defined as a code using a YAML file. The *Readability* of the code snippets and complexity of the build matrix (several pipeline execution running in parallel) affect the pipeline *Maintainability*. Using Environment Variables can help increase pipeline *Maintainability*. [6]

Pipeline Performance Efficiency is related to pipeline build time. This time is affected by the use or not of parallelization execution of the pipeline, the use of caching in continuous integration operations, and the existence of unneeded components in the build. [6]

For this research context, we will evaluate the DevOps pipeline only on the following quality attributes: Pipeline Suitability, Maintainability, Availability, Readability, Performance Efficiency, Reliability, and Security.

4 Solution

Contents

The source control strategy is ground-based for all software projects. This design decision can affect how the team interacts, the number of steps a pipeline must have, and most importantly, can give the team tools to resolve different challenges. Using the standard pipeline as a reference, in this chapter 4, we will study the impact of the Source Code Management (SCM) strategy on the automated delivery pipeline.

For the Trunk-based and Feature-Based strategies, we will define the principal pipeline, validating some possible ramifications and studying the advantages and disadvantages of the different pipeline patterns. The final Section of this chapter is the Patterns analysis based on all the work presented in all previous sections of this chapter will be summarized.

4.1 Trunk-Based Driven Pipeline

The trunk based is a code version control management approach that enables development teams to merge small and frequent updates to the mainline (or core trunk).

The Pipeline driven by a Trunk-Based code version strategy allows the full automation of the software delivery process. This pipeline is most suitable for projects with small, experienced development teams.

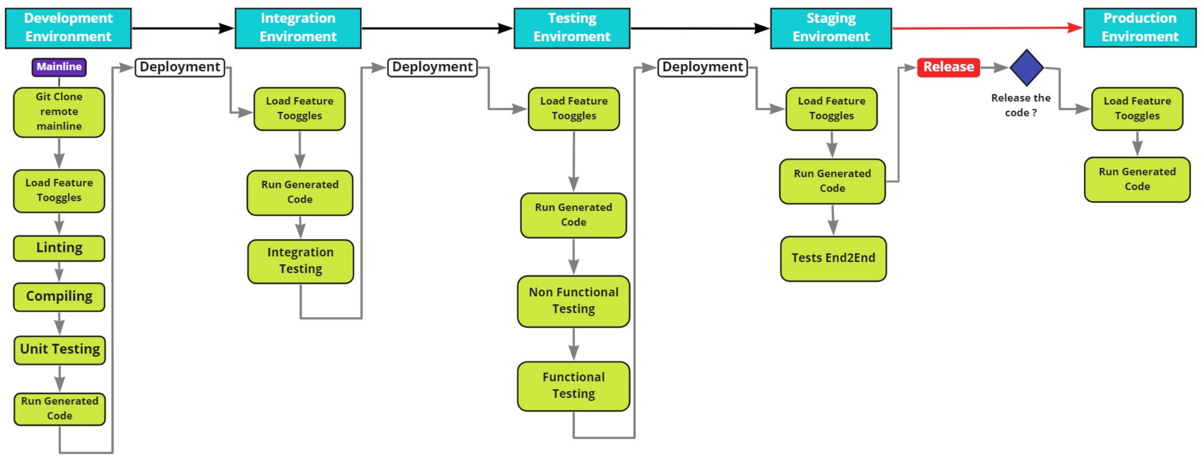


Figure 4.1: Illustration of Principal Pipeline driven by a Trunk-Based code version strategy

The principal pipeline with a very simple pipeline build matrix (uses only the mainline pipeline) can support Continuous Integration, Continuous Testing, Continuous Deployment, and Continuous Delivery.

The figure 4.1 illustrates the principal pipeline that runs starts with a trigger (a code push operation into the mainline branch in the remote repository or a command execution by the development team). The first pipeline step in Development Environment is the code's clone from the remote mainline. After the clone operation, the Linting and Compiling operations start. With the code compiled, the pipeline executes the unit tests. If all unit tests pass successfully, the executable code remains running in the development environment until it is deployed in the integration environment. In this scenario, code is only built once during the pipeline execution. After the development environment, the code generated will not change.

The integration testing starts after the pipeline deploys and executes the generated code in the Integration Environment. If all integration tests pass in the Integration Environment, the pipeline deploys the executable code in the Testing Environment.

The pipeline executes the code generated in the Testing environment and starts the Functional and Non-Functional tests. The executable code is deployed in the Staging Environment if all tests pass.

The executable code running in the Staging Environment allows the team to perform tests end-to-end to guarantee the code quality before releasing it in the Production Environment.

After the development team's approval, the release operation can be executed automatically or manually (this will remove the Continuous Deployment capability).

This principal pipeline only supports planned releases. Unplanned releases must follow the pipeline workflow from the beginning to release the code generated with the corrections in Production.

The smaller deployment of code generated and the faster capability to deliver new features are perks of using only one pipeline. It will also force the team members to merge new code with the mainline before running the pipeline. The code merge will drive each team member to guarantee a higher percentage of test code coverage and automated tests in the code produced.

Using only the mainline in the source code repository may reduce the integration time and increase the delivery speed. However, to properly implement this scenario, the development team must be able to enable or disable features on every pipeline execution. By defining feature toggles in the project code-based or a pipeline configuration file, the development team will avoid unfinished features running in not expected environments. Using feature toggles will also allow the team to mitigate risks by gaining the ability to roll back to previous versions rapidly. The downsides of this technique are the increased percentage of dead code (a section of code present in the source code but whose execution result is not used in any computation operation) and the increase of complexity in the pipeline execution.

If the team decides not to apply Continuous Delivery, it should implement a quality gate in the pipeline configuration before the code release operation.

Implementing feature flags and quality gates will decrease the pipeline maintainability and readability.

4.1.1 Common Pipeline Ramifications

Support a High number of Complex Integration Tests

In some software projects (especially those that use Microservices Software architecture), the integration tests run for extended periods and are more suitable to run at concrete timeframes. This limitation affects the feedback cycle proposed by the DevOps mindset. Even though it is essential to have a high testing percentage, if the test duration becomes too long, it can rapidly become an obstacle to the developers.

This pipeline ramification, illustrated in Figure 4.2, proposes one viable proposed solution to deal with automated complex integration tests. The most simple and quick tests are delegated to the primary pipeline. A second pipeline (Alternative Pipeline) needs to be created with the responsibility to run only these complex tests in the integration environment. This pipeline can run after business hours and will be responsible for running the more time-consuming and not convenient tests to be made during work hours (Complex Integration Testing).

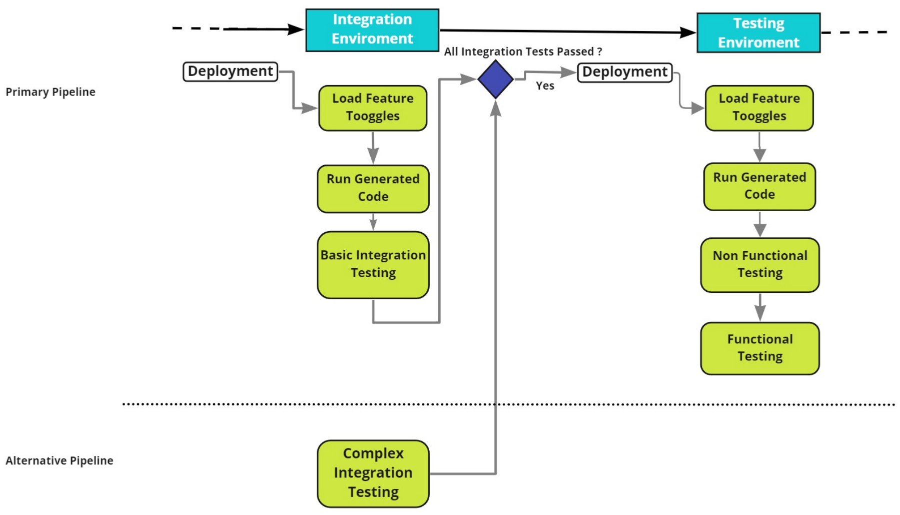


Figure 4.2: Illustration of Pipeline Ramification A

To guarantee that the pipeline workflow is respected, a Quality Gate must be added before the deployment into the Testing environment to grant that the generated code passes all integration tests. This implementation will reduce the pipeline maintainability due to the number of new pipelines, increasing the pipeline build matrix.

However, the performance efficiency of the pipeline will increase due to the parallelization in the integration test executions. It will also reduce Continuous Deployment by having a gate to validate if all tests pass before deploying the code into the Testing Environment.

Lower CPU and memory availability to run the pipeline stages

In development teams lacking compute resources, all CPU or memory available must be used more efficiently to support the automation of the development process.

Typically, the pipeline execution is executed entirely in a remote environment. However, some operations can be executed in a local environment.

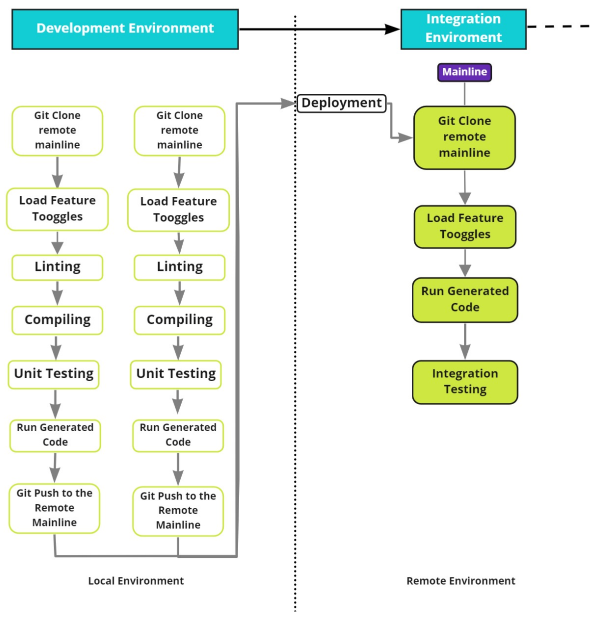


Figure 4.3: Illustration of Trunk-based driven Pipeline Ramification B

Figure 4.3 shows a pipeline using a local build to execute the pipeline operations belonging to the development environment, the team can use the resources from each developer's machine, reducing the resource usage in a remote server. All the tasks performed in a local environment are executed manually.

Local Builds are executed in developers' machines, leading to different Development environments that can cause unexpected errors during or after the compilation operation. Performing the linting, compiling, unit testing, and artifacts in a standardized environment mitigates possible errors and contribute to compatibility.

To archive this standardization, the team can use a virtual machine(VM) or virtual container instances, share it with all team members, and run it in their local machines with all software and needed configurations.

This pipeline ramification supports code generation always in the same environmental conditions, regardless of the configuration or operative system of the host machine. This capability makes the pipeline more reliable. Using the resources from the developer's machines increases the suitability for stakeholders with limited resources.

Standardization has a maintainability cost. The team must support the updates and security patches in the VMs and distribute the image to all team members. If the team uses container images, it will require container technology and a remote repository to store the generated container images.

This pipeline ramification will reduce the pipeline availability due to the build matrix composed of various local pipelines running in the developer's machines. It will also reduce the pipeline efficacy because all the operations in the development environment are performed manually. The pipeline security will decrease because to support the local pipelines, the developers will need their own credentials, used in a non-isolated environment.

Support Inexperience Team

Applying the principal pipeline driven by the trunk-based strategy to an inexperienced team can be challenging. Having a single branch implies that the developers must ensure that the code is mature enough, with a high percentage of unit test code coverage and a low probability of generating bugs. In an inexperienced team, this is hard to accomplish at the beginning.

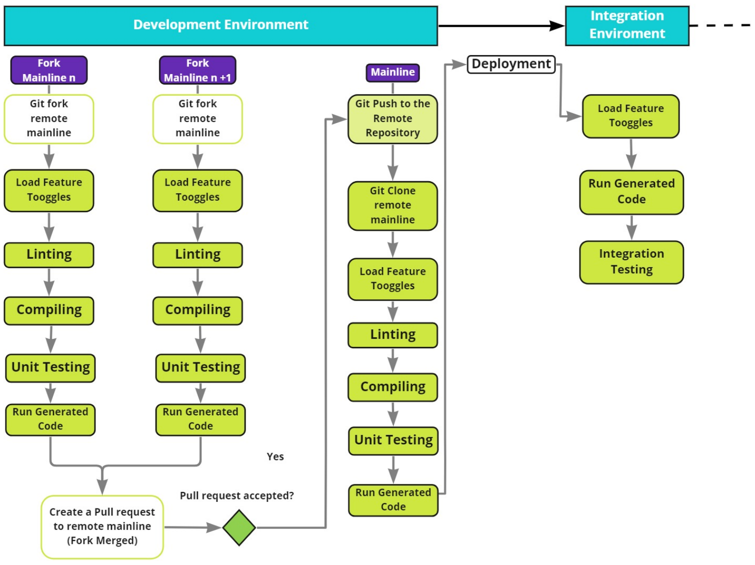


Figure 4.4: Illustration of Trunk-based driven Pipeline Ramification C

As shown in Figure 4.4, the team must guarantee that the code is mature before reaching the mainline pipeline in the development environment. Introducing the pull request operation and creating a pipeline to support mainline forks is one solution to accomplish code maturity in an inexperienced team.

This pipeline ramification supports the ability for developers to use a fork from the mainline, make the needed changes, and integrate without affecting the project code mainline with a pull request, forcing a code peer review from the code maintainer in the remote repository before merging to the code mainline.

Developers may require a more in-depth verification of the code generated in their pipelines supporting the fork before making the pull request. The primary maintainer from the remote repository will spend time reviewing and approving the code merge. These two tasks will improve the pipeline's suitability in code integration.

However, this pipeline ramification will reduce the Continuous deployment and lower the pipeline performance efficiency due to the manual validation operations (increasing the build time of the pipeline).

The workflow of the pipeline ramification is the following:

1. The developer creates forks the mainline it into their local machine, where they will write and test the new code;

2. To support the development in the fork must be created a new pipeline (Where the team can have pipelines templates supported by source code libraries);

3. When the developer wants to merge his fork to the mainline, he creates a pull request to the remote repository;

4. After the creation, the multiple main maintainers of the remote repository are notified to execute a code review;

5. The maintainer will review the code from the fork, make some comments, or request any changes (Send them back to the developer);

6. If everything is OK, the pull request is approved, and the fork is merged into the mainline;

7. After merging the fork with the mainline, the pipeline must be deleted.

We do not consider this pipeline a ramification of a Feature-Based because it is a fork from the mainline, not a creation of a different branch.

Support a Release Ready Mainline

To achieve a release ready mainline, the team must guarantee that the code base present in the mainline is ready to be released in the production environment at any point. This pipeline ramification can be used by high-performance teams and products that need high availability.

As shown in Figure 4.5, the development teams developed the new code in a fork of the mainline in the Development Environment. After the pull request with origin in the fork and destiny mainline, the new code changes from the fork are deployed in the Integration Environment. In this environment, build operations are automatically executed again to guarantee stability during the code delivery after integrating the new code. With these added operations, we can guarantee that the code is not committed directly in the mainline, decreasing the probability of causing instability in the code base and guaranteeing that the code is only merged in the mainline after more controlled code integration.

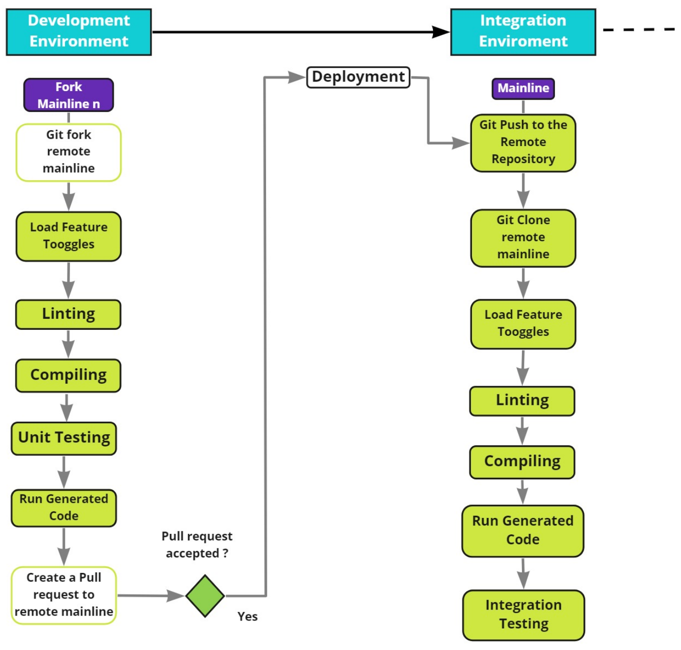


Figure 4.5: Illustration Trunk-based driven Pipeline Ramification D

The code release to the Production Environment is also automated and can be performed automatically without gates.

This pipeline ramification will increase the pipeline's Suitability for the stakeholders due to the added actions (pull request and a release fork to control which release in merge in the mainline) but decrease the Pipeline Performance efficiency due to the increase in build time.

We do not consider this pipeline a ramification of a Feature-Based because it is a fork from the mainline, not a creation of a different branch.

4.1.2 Advantages and downsides

The **advantages** of using a Trunk-Based driven pipeline are:

- A high percentage of test code coverage;
- Reduced number of pipelines to be maintained compared with the other Feature-Based Driven Pipelines;
- Faster capability to deliver new features;
- Smaller Deployment size of the code generated;
- Supports the implementation of Continuous Integration.

The **downsides** of a Trunk-Based driven pipeline are:

- The pipeline workflow must be followed from the beginning to implement Hotfixes or Unplanned releases;
- Implementation of feature toggles to avoid unfinished features running in not expected environments;
- Need for a permission gate in the pipeline configuration before the Production Environment if the team does not desire Continuous Deployment;
- The pipeline needs adaptation to run long-duration Integration testing.

4.2 Feature-Based Driven Pipeline

The Feature-Based is a code version control management approach that allows the developers to have a copy from the main code base (or mainline) where they can work on a new feature until it is completed.

The pipeline driven by a Feature-Branch code version strategy allows the development team to have a more controlled automated delivery, introduce non-planned releases quickly, and have various code-based versions supported by multiple pipelines.

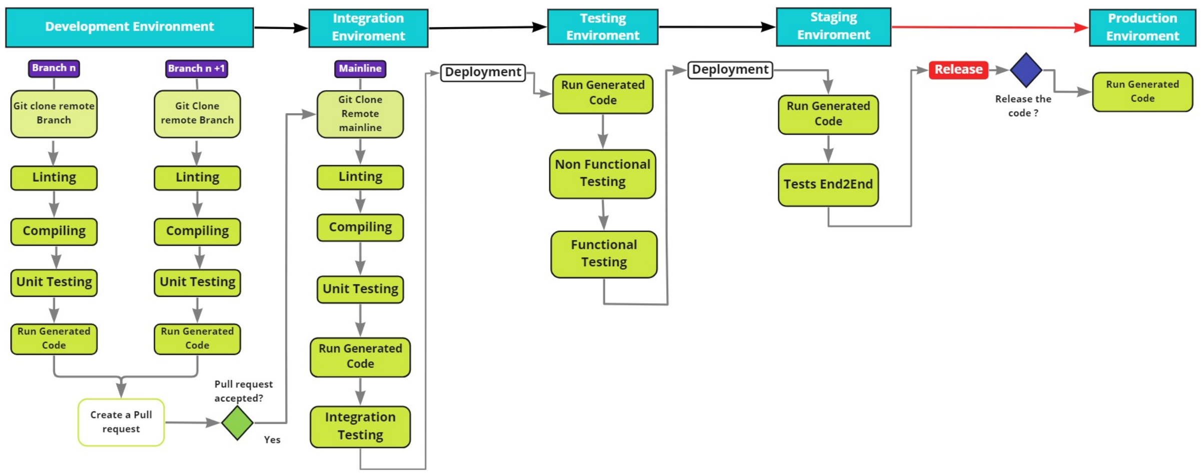


Figure 4.6: Illustration of Principal Pipeline driven by a Feature-Based code version strategy

The principal pipeline illustrated in Figure 4.7 uses n+1 pipelines (n expresses the number of branches created plus the one pipeline for mainline). This principle pipeline supports the team's continuous Testing and Continuous Delivery implementation. The need for having a new branch for each feature removes the ability to integrate and deploy the code continuously.

The development environment supports multiple pipelines that build the source code cloned from each branch. Each pipeline run starts with a trigger (a code push operation into the Feature-Based in the remote repository or a command execution by the development team). It will automatically run linting, compile the code and run the unit tests. The executable code starts running in the development environment only if all unit tests pass successfully.

The developer responsible for the branch creates a pull request when the branch is ready to be merged with the mainline. In this operation, the developers do a significant amount of integration work. The longer the branch lifetime, the more integration work has to be done by the developers. The Feature-Based is deleted after merging with the mainline. From this step forward, all tests are made in the mainline branch.

The mainline pipeline starts with a trigger (a code push operation into the mainline branch in the remote repository or a command execution by the development team). The pipeline will clone the source code from the remote mainline into the Integration Environment, and perform the operations of linting, compiling, and unit tests to guarantee the code stability after the integration. The executable code starts running in the Integration Environment only if all unit tests pass successfully. The Integration Testing starts after the generated code is running in the Integration Environment. If all integration tests run successfully, the pipeline deploys the executable code in the Testing Environment. After the Integration Environment, the code generated will not change unless a hotfix is needed.

The pipeline executes the code generated in the Testing environment and starts the Functional and Non-Functional tests. The executable code is deployed in the Staging Environment if all tests conducted in the Testing Environment pass. In the Staging Environment, the development team can perform end-to-end tests. The release operation can be executed automatically or after the development team's approval. This principal pipeline supports planned and non-planned releases.

This principal pipeline is the most suitable for inexperienced or multiple development teams working on the same project. Creating Pull Requests to Integrate the code in the mainline will force a code peer review to increase code quality.

It will also more rapidly support testing new features (without integration). However, it will force the creation of more complex build matrix pipelines due to the number of code branches needed and their specifications. The number of pipeline stages is proportional to the number of feature branches multiplied by the stages per environment needed. The integration Stage will be a bottleneck to continuous delivery. This stage may contribute to feature delivery delays due to the need to add manual refactoring in the code integration process.

4.2.1 Common Pipeline Ramifications

Release a Hotfix in Production

A hotfix branch allows the team to solve unexpected production environment problems rapidly. However, the delivery pipeline must support the rapid creation of a new environment similar to Production to test the hotfix and execute the build operations to produce the generated code.

If the team uses containers to support the environment, the team can quickly generate the environment automatically using the pipeline.

If containers are not used, the team will need to have a replica of the production environment up and running to solve problems. This replica can be configured using a new pipeline but will increase the complexity of the pipeline's build matrix, reducing the pipeline's maintainability.

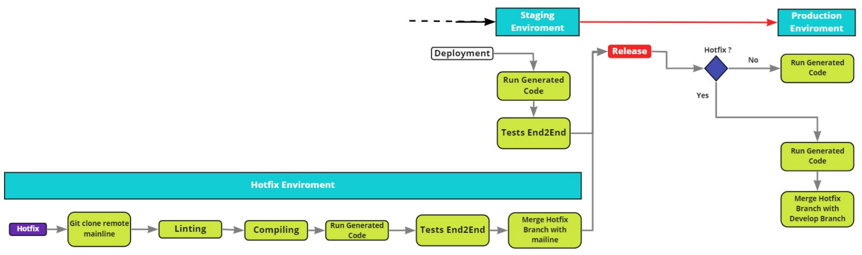


Figure 4.7: Illustration of the Pipeline Ramification A

The new hotfix branch must be a clone from the master/mainline latest commit. The pipeline ramification, represented in Figure 4.7, must be configured to deploy the changes in the hotfix branch in the environment replica and execute the automated build operations for the team to test the solution. When the solution is ready and all the end-to-end tests executed, the generated code is deployed in the Production environment.

The post-deployment process is to merge hotfix code manually to the master branch (resolving any conflicts).

This pipeline ramification can increase pipeline availability and reliability due to the ability to solve problems quickly. Having a second pipeline to support the hotfix decreases the pipeline maintainability due to the need for another pipeline creation, increasing the complexity of the pipeline build matrix.

Experiment a new feature

When the developers want to try out new software approaches, an experiment branch can be used. The pipeline ramification to support that branch must be able to create or deploy in an experimental environment and automate the build operations.

This pipeline ramification that supports the deployment of experimental artifacts, represented in Figure 4.8, runs in parallel with the main pipeline responsible for the delivery process. This pipeline will run the build and deployment operations in the Experimental Environments. All the artifacts running in this environment experiment are not deployed in the main pipeline.

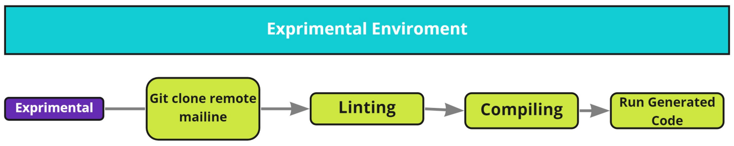


Figure 4.8: Illustration of the Pipeline Ramification B

Using this ramification of the Feature-Based-driven pipeline, the team can, without affecting the main pipeline of the project, automatically deliver the new techniques in a parallel environment.

This pipeline ramification is a straightforward approach that enables the team to test new approaches in code development. It is an excellent opportunity to improve the Functional Suitability of the code by creating specific conditions and testing provided new requirements for the software product. Even though having one more pipeline will increase the team's work by maintaining it.

Support a branch per Environment

Using an environment branching strategy allows the team to have the desired code source version with all configuration changes required to run in a specific environment without using feature branches, feature toggles, or configuration files stored in the code repository.

This pipeline ramification supports all the build operations in each environment (Because the different configurations lead to a different and compatible generated code), all the automated tests, and git operations needed. It should also support the pipeline triggers after the Pull request approval for each branch environment.

This pipeline ramification can have a very simple pipeline build matrix, with only two pipelines, and can become more complex the more environments the team needs to have. (Each Pipeline supports one environment only)

If the team uses only two environments, Development/Quality Assurance and Production, the build matrix becomes very simple with only two parallel pipelines as illustrated in Figure 4.9.

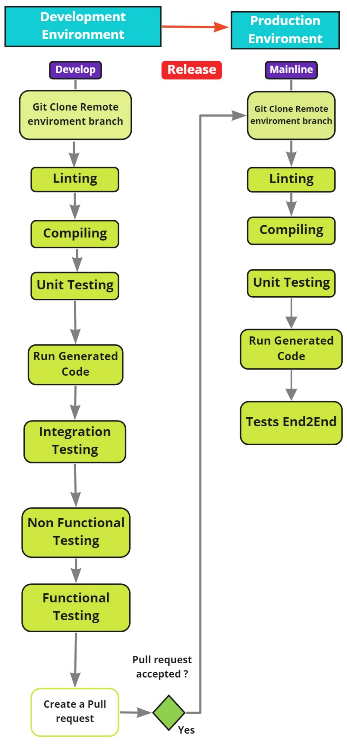


Figure 4.9: Illustration of the Pipeline Ramification C - Simple Version

One pipeline will automate the Development/Quality Assurance operations, where the pipeline supports the build operations, integration, and functional and non-functional tests.

A second pipeline automates the production environment operations where all end-to-end test execution and delivery of the new features to the client occur.

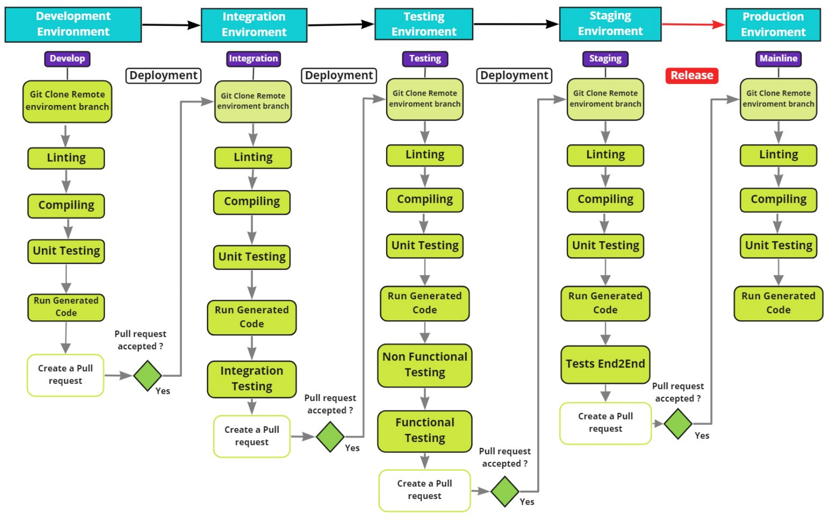


Figure 4.10: Illustration of the Pipeline Ramification C - Complex Version

Represented in Figure 4.10, we can see a pipeline ramification with a more complex build matrix. In this case, we can observe five different pipelines.

When the code is ready to be deployed in a different environment, the development team creates a pull request to merge the new changes.

After the approval, the pipeline executes a clone from the code source, and build and test operations are executed. Each environment has its build operations and testing operations.

Using a simple build matrix can result in an increase in pipeline maintainability compared with the principal pipeline. Nevertheless, this maintainability can decrease with the number of different environments created. This pipeline ramification also contributes to increasing the pipeline suitability to the stakeholders by separating code-building operations in different environments with different configurations. However, the quantity of automated operations decreases pipeline maintainability.

Support Multiple Releases of the Product

Having the ability to have multiple version of the code running allow the development team to execute the needed bug resolution without affecting the mainline, providing the user with the most stable version of the product and efficiently upgrading the version when all the bugs are resolved. This pipeline derivation is more suitable for multiple Version Products, like mobile apps.

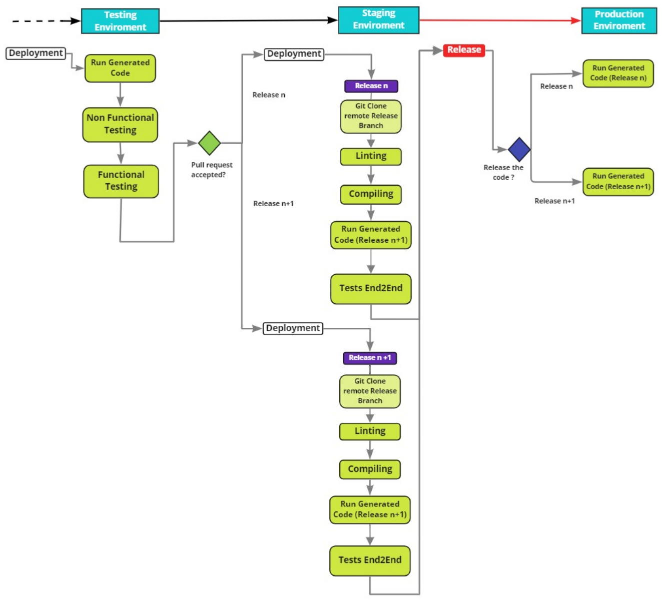


Figure 4.11: Illustration of the Pipeline Ramification D

As we can see in Figure 4.11, each code version will have a respective pipeline, but all the operations are operationalized in the same environments. In the production environment, the team will have multiple version releases.

It is also a suitable pipeline ramification if the team needs to support a manual approval of each version, guaranteeing that each version is frozen until the release manager decides to release the Product user.

This pipeline ramification complex build matrix increases the maintainability. However, the possibility of having multiple versions released increases the pipeline suitability for the stakeholders and increases the pipeline availability due to the ability to release new features quickly and have older versions in production for more time, allowing a quick rollback if needed.

Supporting Git Flow

Git flow is the most used branching pattern used. This pattern merges the feature, hotfix, and release branches. But having multiple branches to support the pipeline is a challenge.

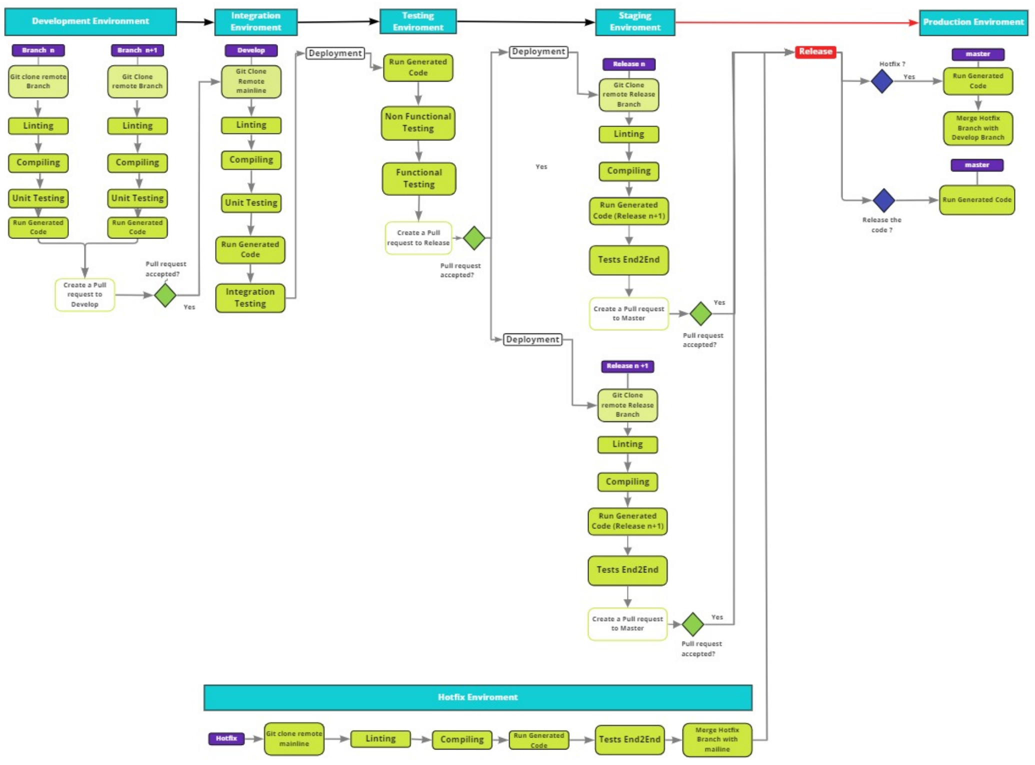


Figure 4.12: Illustration of the Pipeline Ramification E

This pipeline ramification that supports the git flow strategy inherits the characteristics of the two other pipeline ramifications (hotfixes (Figure 4.7) and the releases(Figure 4.11)). The pipeline ramification supporting multiple releases is used to support multiple code releases, The pipeline ramification supporting hotfixes is used to resolve problems in production, increasing this pipeline's availability and suitability for the stakeholders.

Figure 4.12 reflects this pipeline's very complex build matrix, which will decrease the maintainability and readability of the pipeline.

Having multiple different pipelines and manual operations decreases the pipeline performance efficiency.

Programmatic releases

Programmatic Releases (or Release Train) involve all teams developing the product with the same release cadence (can be a release once a quarter, once a month, or even once a week).

This strategy is commonly used in large programs or when individual teams are working as part of a larger whole.

As we can see in Figure 4.13 this pipeline ramification must be an enabler to support a controlled integration between features and releases and between the releases and the mainline.

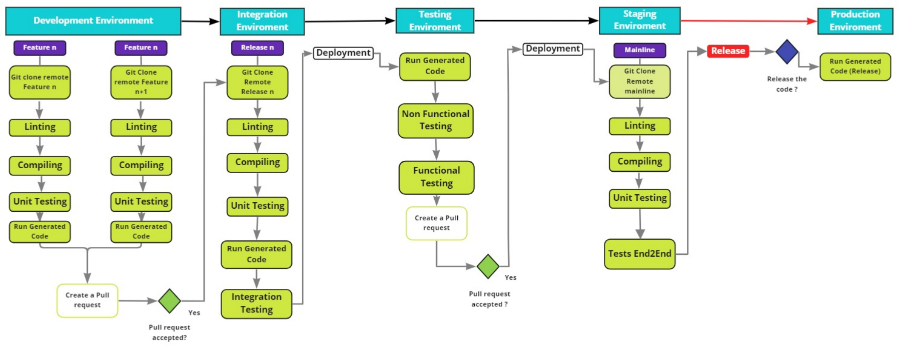


Figure 4.13: Illustration of the Pipeline Ramification F

The feature branches are supported by a dedicated pipeline, allowing the developer's team to build the new features automatically.

To pass the code to the integration environment, first, the code merge of the feature branch with the release branch must be executed.

Another pipeline will be dedicated to supporting the release, being responsible for cloning the code from the release branch execute the automated operations that build and test the code of the features chosen to be released in a predefined deployment time window.

It will also be in the build matrix, a pipeline dedicated to automating the mainline operations. It is triggered after merging the release branch into the mainline branch. This pipeline is responsible for deploying the code in a pre-production environment where the final checks are made before the release of the new code version in the production environment.

This pipeline ramification will also allow having multiple release version pipelines, allowing the development team to start working in parallel on future versions of the product. This characteristic will increase the pipeline's suitability.

Being the team constrained to a standard release schedule will affect the ability to deliver code, reducing the pipeline efficiency and availability.

The pipeline ramification maintainability will also decrease due to the number of pipelines that need to be maintained, increasing the pipeline build matrix.

4.2.2 Advantages and Downsides

The **advantages** of using a Feature-Based Driven Pipeline are:

- Automates the Experimentation of new code features in an isolated environment.
- Support quickly deploys unplanned releases in production.
- Supports the ability to have programmed Releases for the Product.
- Supports having multiple version releases in Productions.
- The manual review before merging features in the mainline gives the pipeline high reliability and availability.

The **downsides** of using Feature-Based Driven DevOps Pipeline are:

- Complex build matrices lead to low pipeline maintainability.
- Cannot implement continuous Integration and Continuous Delivery due to the use of manual operations (Pull Requests).
- The high number of manual operations reduces the pipeline efficiency.

4.3 Patterns Analysis

Having source code a critical role in the software development teams, we can easily conclude that his management strategy directly impacts the automated pipelines that support the software delivery. This analysis pretended to give the development teams important insights before choosing the branching strategy based on the Feature-Based Driven Pipeline (FBDP) and Trunk-Based Driven Pipelines (TBDP) and their respective ramifications.

The drive for having a high percentage of code coverage, the simple and highly maintainable pipeline build matrix, and the faster capability to deploy new features make the Trunk-based driven pipelines the best approach to implement Continuous Deployment, Continuous Integration, and Continuous Delivery.

Suppose that Continuous integration and Continuous delivery are not required, but the team wants to implement Continuous Deployment with the automation of the build and testing process. In that case, the Feature-Based driven pipeline is the best approach.

Using FBDP will contribute to a more controlled delivery process, partially automated, due to the need to support a manual review of the code. With this, the team can have a pipeline dedicated to each feature, supporting the isolation and automation of the build and testing operations. When needed, this pipeline type will also support the direct merge of unplanned features into the mainline and release them quickly in Production. All characteristics make this pipeline type the better approach for a controlled continuous deployment of multiple features.

If the team chooses to use a TBDP will be forced to implement feature toggles to control the feature's availability in the environments. Using feature toggles implies a change in the code structure, increases the code base complexity, and contributes to the percentage of dead code in the code base. In FBDP, because each branch has its dedicated pipeline, and the new feature is "activated" when the code branch merges with the mainline, there is no need to use features toggles.

Unplanned releases take more time to implement in a TBDP than in FBDP due to the need to pass the new code changes in all environments before reaching Production. In FBDP, a parallel pipeline supports the automation of the build operations for unplanned release features and releases them in Production. This ability shorter the time to release a hotfix in production.

Using a TBDP, the development team can support Continuous Integration and Delivery. The team can fully automate the delivery process using a single pipeline that supports all delivery and integration operations. It is impossible to achieve this in FBDP due to the manual Integration and Deployment tasks.

In summary, we can make a simple distinction. Using Trunk-Based Driven Pipelines (TBDP), we have full automation of the delivery process, a higher pipeline efficiency, and faster delivery of new features. Using Feature-Based Driven Pipeline (FBDP), we have more control over the code integration, isolate the development of new features, and respond more quicker to unplanned releases.

5

Evaluation

Contents

5.1 Evaluation Method

The evaluation of each pipeline pattern driven from the defined source code management strategies, illustrated in the pipeline use cases presented in Chapter 4, will be executed in two phases. First, we will use complexity expressions representing four complexity measures: The number of Automated Operations, Possible Pipelines, Quality Gates, and the Number of Manual Operations. This complexity expression will convert the use case illustrations to support the results from the qualitative study of the artifact.

To better understand the complexity measures, the subsections below briefly explain each.

5.1.1 Number of Automated operations

The *Number of Automated Operations* metric measures the number of operations needed to support the automated delivery operations, represented in green and red in Figure 3.4, plus the automated deployment operations, represented in white and red the Figure 3.4. The automated operations supporting the delivery mechanism will be represented in the complexity expressions with the letter **o** and the automated deployment operations with the letter **d**.

A higher number of operations will promote an increase in pipeline efficiency (decreasing the build time), an increase in pipeline reliability (More automation leads to fewer errors), increase in Security (All needed credentials are isolated and only used by the pipeline and supports the Automated monitoring, automated testing, and code review automation) and increases the pipeline availability (reduced the time to recovery if the delivery process fails). The need to maintain and evolve the automated operations will lower the pipeline maintainability and readability. Automatic operation limits the user's freedom to perform tasks, taking into account the steps that best suit their needs, reducing the Pipeline Suitability.

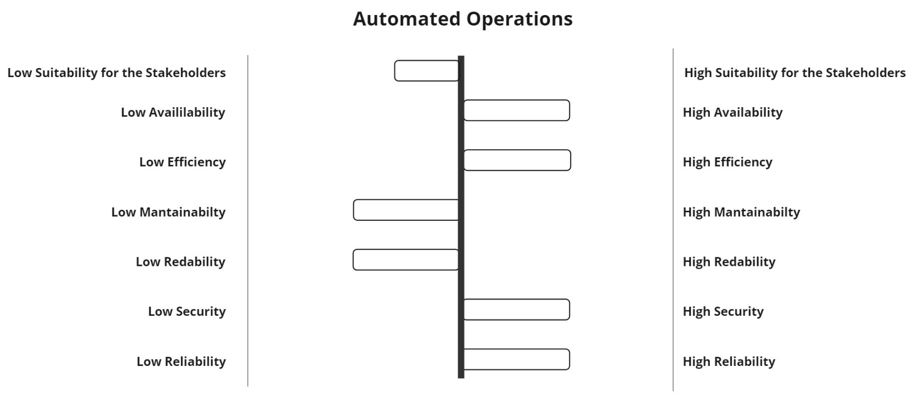


Figure 5.1: Qualitative Analysis for Automated Operations

5.1.2 Possible Number of Pipelines

The *Possible Number of Pipelines* measures the number of pipelines required to support the delivery process, the build matrix. A higher number of pipelines contributes to lower pipeline maintainability, readability, and efficiency (Increases the points of failure and contributes to the existence of pipelines that are occasionally used). The ability to separate the different needs in different pipelines increases the suitability for all stakeholders (the pipeline scope can be delegated to a specific scope) and the pipeline availability (Support of multiple versions of the code running in parallel). The number of pipelines doesn't affect pipeline security and reliability in this context (the Number of pipelines doesn´t directly affect the security [56] nether the probability of having errors when the feature is delivered).

In Feature-Based Driven Pipeline (FBDP), the number of feature branches directly impacts the number of pipelines. A new pipeline must be created for each branch created to support the automated operations of the branch code delivery.

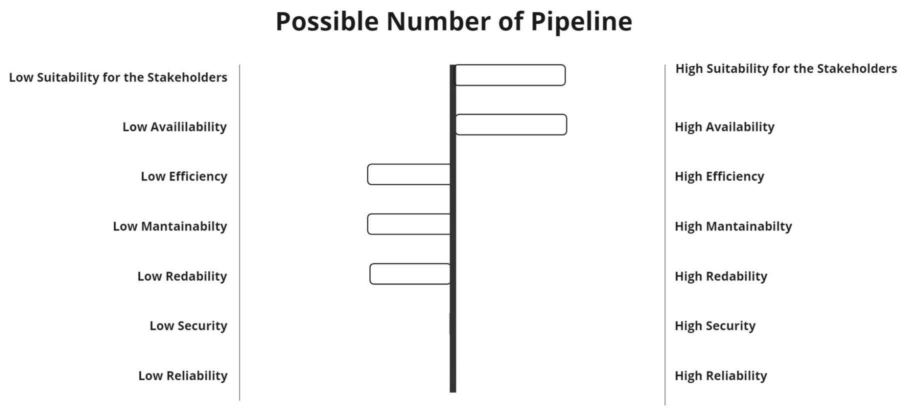


Figure 5.2: Possible Number of Pipelines Impact in pipeline quality

5.1.3 Quality Gates

The *Quality Gates* measures the number of Feature Toggles, Pull requests, and Approvals of a release in Production present in the pipeline. All these decisions can decrease the pipeline readability and maintainability (increase the work done in the pipeline to support the control gates in the pipelines) but increase the pipeline reliability (More control gates, less probability for errors) and pipeline availability (Fewer errors, lower time to deliver). The ability to control the activation of features in different environment increase the Pipeline Suitability for the stakeholders. When used to contribute to support automated monitoring. automated testing, and code review automation, the quality gates increased the pipeline security.

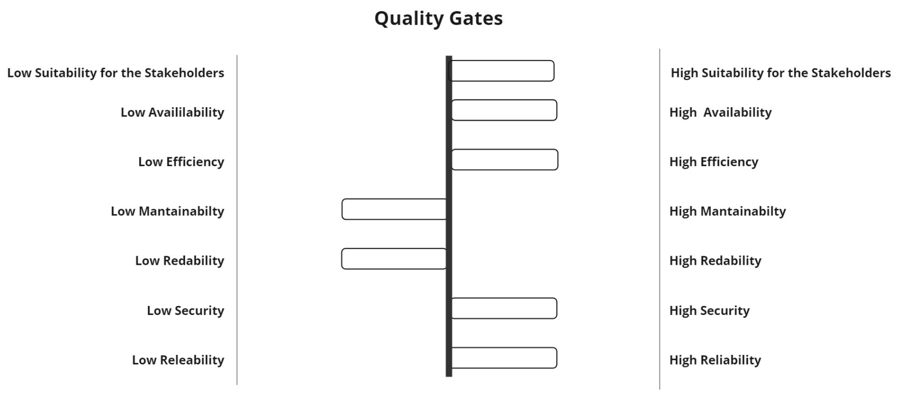


Figure 5.3: Quality Gates Impact in pipeline quality

Features Toggles

As already seen in Section 4.1, all the Trunk-based driven pipelines and respective ramifications use feature toggles to control the state of a feature in specific environments. To understand the complexity of implementing feature toggles in a trunk-based driven pipeline, we will provide an example pipeline that materializes in Figure 4.1. The full version of the code can be accessed in Annex A. This example supports a straightforward python application.

The first step to implementing feature toggles is to add conditionals that will control the code with part of the code that will be applied.

Listing 5.1: Simple Python Application

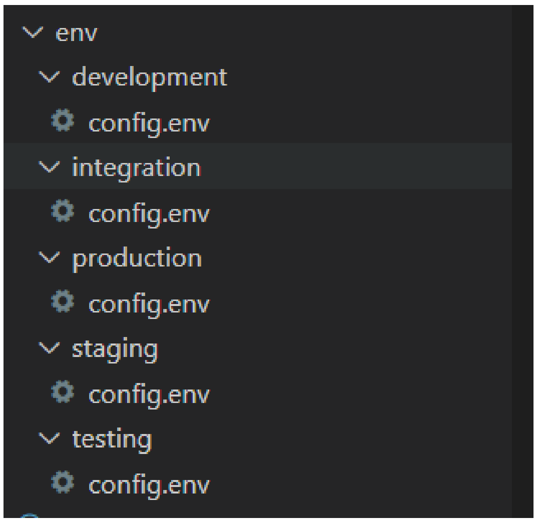
```python
# Import the os library to be able to use the Environment Variables
import os
# If the Feature Toggle represented by "Feature1" is enabled
will print the text inside the
# function print()
if os.environ.get('FEATURE1') == "True":
  print("Running FEATURE1"))
# If the Feature Toggle represented by "Feature2" is enabled will
# print the text inside the function print()
if os.environ.get('FEATURE2') == "True":
  print("Running FEATURE2")
```

In Listing 5.1, we can observe that the conditional IF will increase the cyclomatic complexity of the code. The complexity of the pipeline because a new step must be added to the pipeline to load the Feature Toggle to the environment variables, as seen in the Listing 5.2

Listing 5.2: Load Feature toggle pipeline operation

```
- uses: cardinalby/export-env-action@v1
      name: Load Feature toggles
      with:
        # Will need to set for each environment the path for the config-env
        # containing all the feature toggles with the key-value definitions
        envFile: './env/testing/config.env'
        expand: 'true'
```

The project code Structure will also increase the complexity because it will force to have a `config.env` file for each environment with file structure shown in Figure 5.4



```
env
  development
    config.env
  integration
    config.env
  production
    config.env
  staging
    config.env
  testing
    config.env
```

Figure 5.4: Illustration Project Structure using Feature Toggles

The Feature Toggles will be represented in the Complexity Expressions with the variable **f**.

Pull Requests in a Feature-Based Driven Pipeline

To understand the complexity of implementing pull requests in a Feature-Based Driven Pipeline, we provide an example pipeline illustrated in Figure 4.6. The full version of the code can be accessed in Annex A.

The build matrix to support pull requests is composed of at least two pipeline types, a Feature Pipeline and a Mainline Pipeline

The pull requests are illustrated in Figure 3.4 as a green diamond shape.

Each feature branch must have its own pipeline feature pipeline. This pipeline is only triggered when a push is made to the feature branch. We can observe the implementation of this trigger in Listing 5.3.

Listing 5.3: Feature Pipeline Trigger Example

```
name: Feature Pipeline
on:
# This Definition sets the trigger of the Feature Pipeline
# only when the code is pushed to the feature branch
  push:
    branches:
      - feature/**
```

The Mainline Pipeline is only triggered when the pull request is approved and the code merged in the mainline. We can observe the implementation of this trigger in Listing 5.4.

Listing 5.4: Continuous Deployment pipeline Trigger Example

```
name: CD
on:
  # This Definition sets the trigger of the CD pipeline only when the code
  # is pushed to the mainline
  push:
    branches: [main]
  # and also when the pull request is approved and the code merged in
  # the mainline
  pull_request:
    branches: [main]
```

Additionally to these steps, we need software, in most cases embedded in the source code manager, where the code contributor can create the pull request, validate the changes, and request a review from the remote repository maintainer.

The Pull Requests will be represented in the Complexity Expressions with the variable **p**.

Approval of Releases

If Continuous Delivery is not applicable by the team in the delivery process, a manual intervention must be introduced to control the release of the code in the Production Environment.

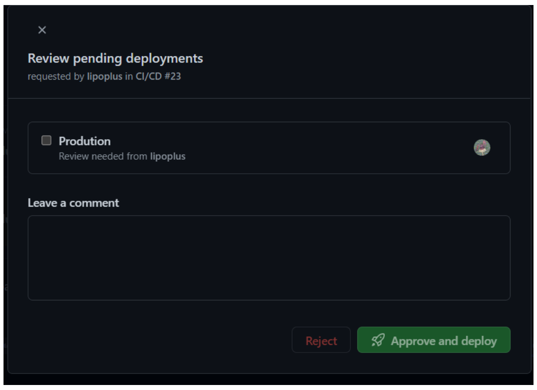
First, the Source Code Manager must be configured for each environment, and the group of users that can approve the pipeline runs access to the environment. Then this configuration is introduced in the pipeline as represented in Listing 5.5.

Listing 5.5: Release Operation definition in the pipeline

```
 release:
      name: Release in Production
      needs: staging_enviroment
      runs-on: ubuntu-latest
      # Using the environment tag, we can configure the place where the code
      # will be deployed and the need for permissions to approve the
      # Releases/Deployment of the code in the environment
      environment:
        name: Prodution
        url: https://github.com
      steps:
        - name: Run Code in Production
          run: echo Coding is deployed
```

Before the pipeline executes the release of the code in the production environment, a notification, represented in Figure 5.5, is launched, and only the team member with the required permission will be able to approve the release.

The Approval Releases will be represented in the Complexity Expressions with the variable **a**.


Review pending deployments
requested by lipoplus in CI/CD #23

Prodution
Review needed from lipoplus

Leave a comment

Figure 5.5: Notification for Approval of the Release

Approval

In some pipeline ramifications, the progression to other environments depends on another pipeline's successful build. In the example pipeline presented in Annex Aand representing Figure 4.4, the quality gates are used to guarantee that the new version of the code is deployed into the testing environment if the principal pipeline and a parallel pipeline running in a scheduled time executing complex integration tests run successfully.

In the Listing 5.6, we have a code snippet of the scheduled pipeline.

Listing 5.6: Example of a schedule pipeline

```
on:
```

```
# In this example, the pipeline runs every 15th minutes, but this value
# can be changed
  schedule:
    - cron:  '*/15 * * * *'

jobs:
  name: Run Complex Tests
      runs-on: ubuntu-latest
      environment:
        name: testing
        url: https://github.com
      steps:
        - uses: actions/checkout@v2
        - name: Running Complex tests
          run: echo Tests are running
```

In the principal pipeline, we need to add a link to this schedule pipeline and configure the dependency on the deployment of the next environment, as we can see in Listing 5.7.

Listing 5.7: Configuration of the schedule pipeline

```
# Link to the scheduled pipeline
  complex_tests:
    name: Run Complex Tests
    runs-on: ubuntu-latest
    environment:
      name: integration
      url: https://github.com
    steps:
    - uses: cardinalby/schedule-job-action@v1
      with:
        ghToken: ${{ secrets.WORKFLOWS_TOKEN }}
        templateYmlFile: '.github-scheduled-workflows/complex_test_example.yml'
  deployment_testing:
    name: Deploy in Testing Environment
    # Configuration of the dependency in the scheduled pipeline and principal pipeline
    needs: [integration_enviroment,complex_tests]
    runs-on: ubuntu-latest
    environment:
      name: testing
      url: https://github.com
    steps:
      - name: deploy
        run: echo Coding is deployed
```

The Quality Gates will be represented in the Complexity Expressions with variable **q**.

Number of Manual Operations

The *Number of Manual Operations* measures the number of operations in the pipeline workflows that need human interaction to be completed. Manually performing the pipeline operations decreases pipeline security by promoting the use of high-privileged commands and multiple credentials in non-isolated environments. The pipeline reliability, efficiency, and availability also decreased due to the increase in the probability of human error and increased build time due to manual operations.

The liberty of manual operations allows the developers to execute the tasks without dependency or restrictions, increasing the stakeholder suitability of the pipeline.

The manual operations don´t affect the pipeline readability or maintainability in this context.

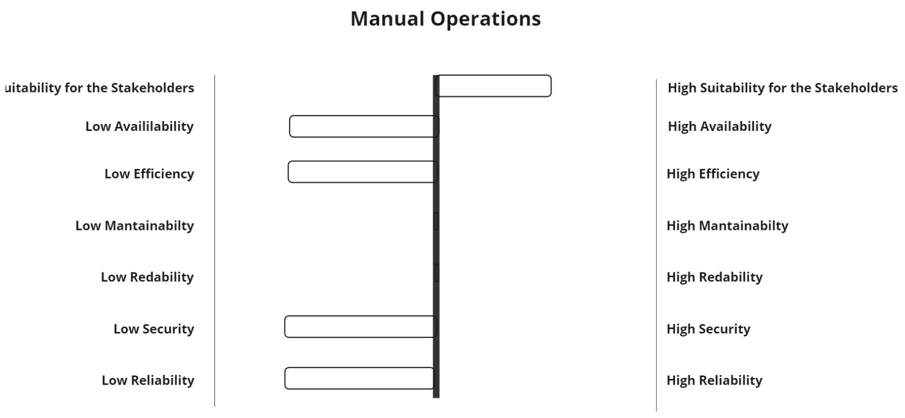


Figure 5.6: Manual Operations Impact in pipeline quality

5.2 Complexity Expressions

In this section, we will present all the resulting complexity expressions and discuss qualitative analyses of each pipeline pattern.

5.2.1 Trunk Based Complexity Expressions

In the table 5.1 the X represents the number of feature toggles, Y is the number of developers, and the Z is the number of forked pipelines.

	Number of Automated Operations	Number of Manual Operations	Possible Pipelines	Quality Gates
Trunk-Based Driven DevOps Pipeline	18o + 4d	0	1	Xf + 1a*
Support a High number of Complex Integration Tests	19o+ 4d	0	2	Xf + 1q +1a*
Lower CPU and memory availability to run the pipeline stages	11o + 4d	7Y	1	1*
Support Inexperience Team	Z(5o) + 18o + 4d	2Z	Z +1	Xf + Zp +1a*
Support a Release Ready Mainline	Z(5o) + 12o + 4d	2Z	Z + 1	Xf+ Zp +1a*

Table 5.1: Trunk Based Complexity Expression

*This operation only exists if the team doesn't use continuous delivery

5.2.2 Feature-Based Complexity Expressions

In the table 5.2 X represents the number of Feature Branch, Y is the number of Hotfixes, the R represents the number of active releases of the code.

	Number of Automated Operations	Number of Manual Operations	Possible Number of Pipelines	Quality Gates
Feature-Based Driven DevOps Pipeline	X(5o) + 12o + 3d	1X	X +1	Xp + 1a
Release a Hotfix in Production	Y(6o) + X(5o) +12o + 2d	1X	X + Y +1	Xp + Ya +1a
Experiment a new feature	4o	0	1	0
Support a branch per Environment (Simple)	14o	1	1	p
Support a branch per Environment (Complex)	29o	4	5	4p
Support Multiple Releases of the Product	X(5o) + 9o + 1d + R(5o) + R(1d)	1X + R	X+1+R	Xp + Rp + Ra
Supporting Git Flow	X(5o) + 11o +2d +R(5o) +Rd + Y(8o)	1X + 2R	X + R + Y +2	Xp + R(2p) + Ya + 1a
Programmatic Releases	X(5o) + 15o + 3d	1X + 1	X + 2	Xp + p + 1a

Table 5.2: Feature-Based Complexity Expressions

5.3 Qualitative Analysis

The following qualitative analysis takes as its target the study of the complexity expression described in section 5.2, and is a visual interpretation of resulting the congregation of the complexity measures presented in section 5.1. Each complexity-expression illustration was interpreted to reflect the impact of each complexity measure in each pipeline quality attribute, supported by the interpretation of the values defined in complexity expressions.

5.3.1 Trunk Based Driven Pipelines Qualitative Analysis

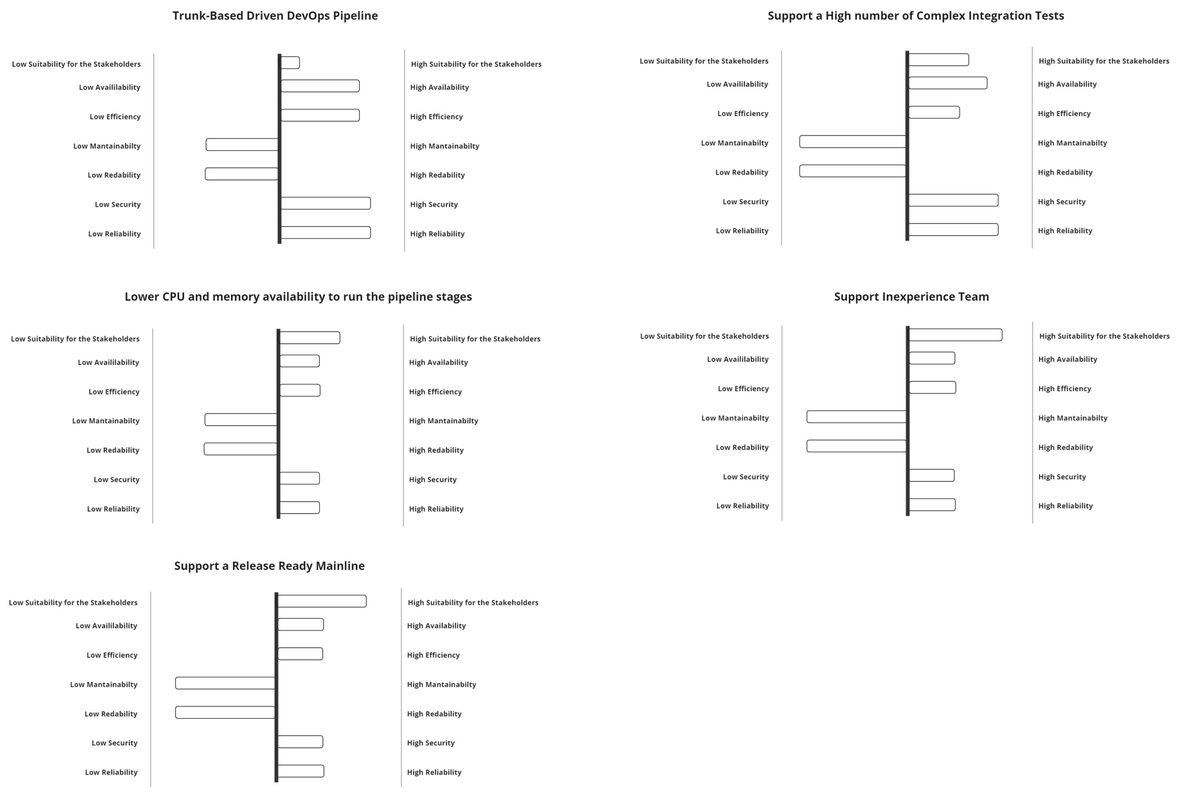


Figure 5.7: Trunk-based Driven Pipelines Qualitative Analysis

Trunk-Based Driven DevOps Pipeline The suitability to the stakeholders is low but will increase the more features toggles we have present. The increment of elements will also progressively decrease the pipeline maintainability and readability. Not having manual operations contributes to increased pipeline availability, efficiency, security, and reliability.

Support a High number of Complex Integration Tests This pipeline ramification will improve the pipeline's suitability due to the presence of another pipeline and approval gate. Having another pipeline will also mean a slight decrease in pipeline efficiency. The other complexity attributes maintain the same values as the principal pipeline.

Lower CPU and memory availability to run the pipeline stages The use of manual operations will allow this pipeline ramification to improve the pipeline's suitability but will slightly decrease the pipeline's security, reliability, efficiency, and reliability. It will not have an impact on the pipeline's Maintainability and Readability.

Support Inexperience Team and Support a Release Ready Mainline These two pipeline ramifications have similar contributions from the complexity attributes. Increasing the number of Forks and Feature toggles, the suitability for the stakeholders tends to increase but will affect the Maintainability and Readability, decreasing them. Having more forks will contribute to an increase in manual operations, which will slightly decrease pipeline availability, efficiency, security, and reliability.

5.3.2 Feature-Based Driven Pipelines Qualitative Analysis

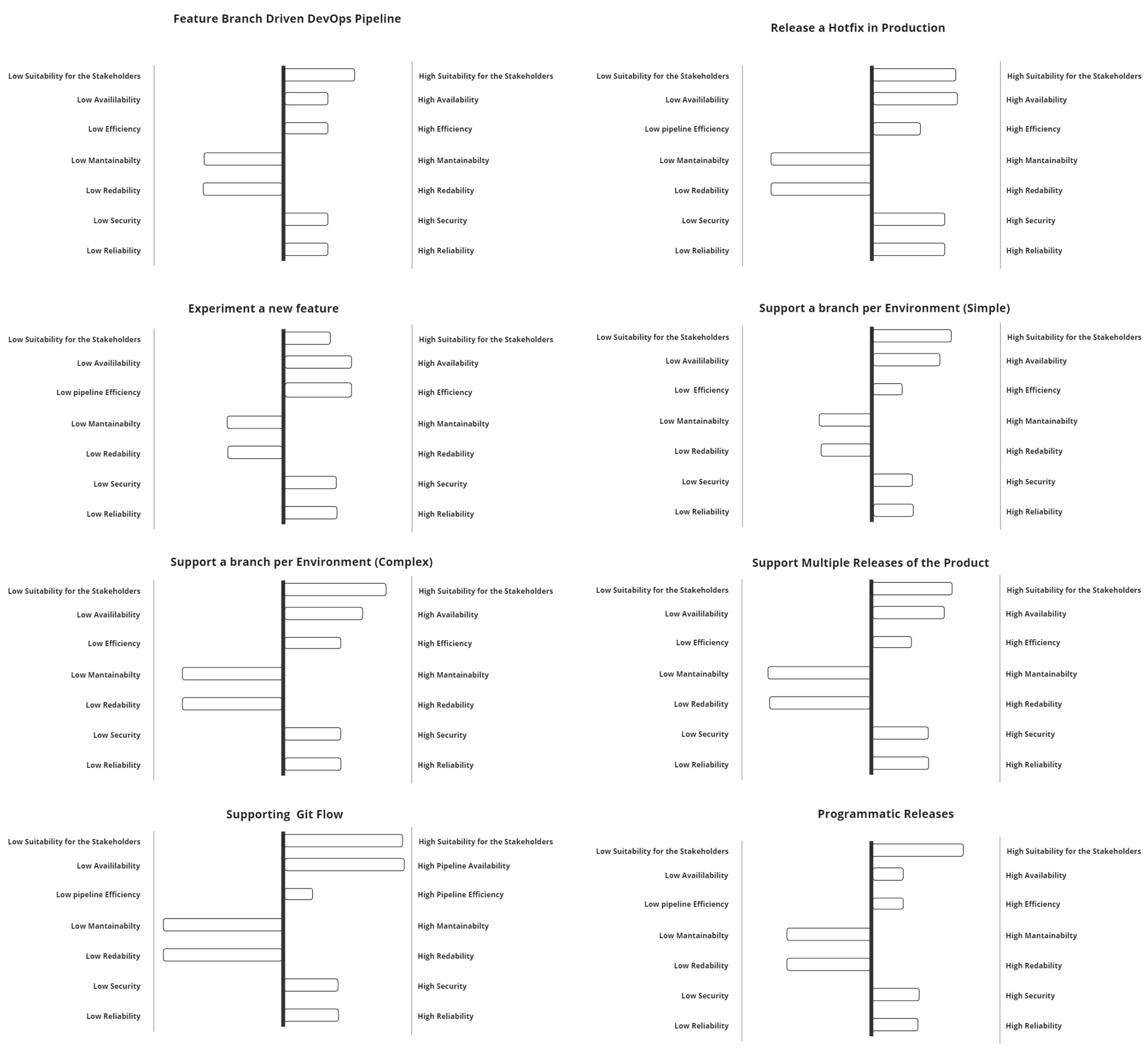


Figure 5.8: Feature-Based Driven Pipelines Qualitative Analysis

Feature-Based Driven DevOps Pipeline This principle pipeline contributes to high pipeline suitability, increasing the more features branches the team creates (Will increase the number of pipelines and Quality gates). This principal pipeline will also have low maintainability and readability due to the number of automated operations and quality gates needed. The number of manual operations will slightly decrease the security and reliability of the pipeline. This pipeline pattern and its ramifications have lower availability and efficiency than the trunk-based driven principle pipelines.

Release a Hotfix in Production This pipeline ramification was similar to quality attributes distribution compared with the principal pipeline. A slight increase in the number of automated operations, possible pipelines, and quality gates set the difference between the principal pipeline, increasing the pipeline suitability for the stakeholders and the pipeline availability. The downside is a decrease in maintainability and readability. The pipeline efficiency is more or less the same (Even though we have more automated operations and quality gates that increase the efficiency, the number of possible pipelines growing and the presence of manual operations contributes to the decrease)

Experiment with a new feature This pipeline ramification is composed of only two complexity attributes: Automated Operations and a Possible Number of pipelines. This composition contributes to increased pipeline availability and efficiency when compared principal pipeline. We can also identify a slight increase in pipeline maintainability and readability (due to the simplicity of the pipeline definition). Regarding pipeline suitability, even though we have automated operations, the ability of the stakeholders to test new features contributes to an increase, making this pipeline ramification highly suitable for the stakeholders.

Support a branch per Environment (Simple and Complex) The use of this pipeline ramifications increases the pipeline suitability influenced by the number of manual operations, pipeline possible number, and quality gates. More elements lead to an increment in pipeline suitability. Nevertheless, the presence of these complex elements in the pipelines decreases pipeline Maintainability and readability. The pipeline efficiency stays the same (same justification as in the Release Hotfix in Production)

Support Multiple Releases of the Product The more features and releases we had supported by this pipeline ramification, the higher the pipeline suitability will be and the lower the pipeline maintainability and readability. The pipeline efficiency will be slightly reduced compared with the principal pipeline, influenced by the number of manual operations and possible pipelines. The security and reliability will stay the same as the principal pipeline. Even thrown we have manual operations that decrease these two complexity attributes, the automated pipelines and quality gates balance by increasing them.

Supporting Git Flow The number of quality gates, manual operations, and possible pipelines in this pipeline ramification contributes to very high pipeline suitability and availability. However, this high number of these complexity attributes contributes to low pipeline maintainability and readability. The pipeline's security and reliability remain the same as the principal pipeline due to the same reason mentioned in the pipeline ramification Support Multiple Releases of the Product.

Programmatic Releases This pipeline ramifications increase the pipeline's suitability compared with the principle pipeline. However, the presence of manual operations and the possible number of pipelines will decrease pipeline availability and efficiency. The pipeline security and reliability slightly decrease due to manual operations. The pipeline maintainability and readability will be lower due to the number of automated operations, a possible number of pipelines, and quality gates.

6

Conclusion

Contents

In this chapter, we conclude the research with general conclusions, limitations and future work

6.1 Conclusions

We believe all the findings will enrich the current knowledge regarding pipeline quality attributes and their implications, contributing valuable insights to academia and the industry.

After modeling the Trunk-Based Driven Pipelines (TBDP) patterns (Principal Pipeline and their ramifications), we reach the following conclusions:

- Trunk-Based Driven Pipeline fully supports Continuous Integration, Continuous Testing, Continuous Deployment, and Continuous Delivery (Figure 4.1).
- The pipeline ramification that supports a High number of Complex Integration Tests enables the use of a parallel pipeline to deal with automated complex integration tests. (Figure 4.2).
- The pipeline ramification that contributes to dealing with Lower CPU and memory availability to run the pipeline stages supports the use of local builds (But operations locally are executed manually) (Figure 4.3).
- The pipeline ramification that supports Inexperience Teams allows the developers to make the needed changes in a fork supported by a pipeline and integrate them into the mainline when the code is ready (Figure 4.4).
- The pipeline that supports a Release Ready Mainline contributes to a new layer of control in the delivery process, allowing the development team only to merge a new release in the mainline when they have a high confidence degree in the new code changes implementation (Figure 4.5).

After modeling the Feature-Based Driven Pipeline (FBDP) patterns (Principal Pipeline and their ramifications), we reach the following conclusions:

- Feature-Based Driven Pipeline enables the development team to have a more controlled automated delivery, introduce non-planned releases quickly and have various code-based versions supported by multiple pipelines (Figure 4.1).
- The pipeline ramification that supports the Release of a Hotfix in Production allows the development team to rapidly deliver in the Production Environment a resolution to an unexpected problem in the software application (Figure 4.7).
- The pipeline ramifications that support the Experimentation of new features allow the development team to automatically deliver experimental artifacts in an isolated environment, not affecting the

principal delivery pipeline. This allows the developers to experiment without affecting the delivery mechanism of the software product (Figure 4.8).

- The pipeline ramification that supports a branch per Environment Simple and Complex gives the development team the possibility of having different environments without using feature branches, feature toggles, or configuration files stored in the code repository (Figure 4.10 and Figure 4.9).
- The pipeline ramification that supports multiple releases of the product contributes to the automation of the delivery of multiple versions of the code-version simultaneity (Figure4.11).
- The pipeline ramification supporting Git Flow aggregates the other two pipeline ramifications (Releases of a Hotfix and Multiple Releases) (Figure 4.12).
- The pipeline pattern supporting Programmatic releases enables a controlled integration between features and releases and between the releases and the mainline (Figure 4.13).

After modeling the pipelines using Trunk-Based Driven Pipelines (TBDP), we observe that the development team with this pipeline pattern can rely on complete automation of the delivery process, higher pipeline efficiency, and faster delivery of new features. When using a Feature-Based-Driven Pipeline (FBDP), the development team has more control over the code integration, can rely on the isolation of new features in a dedicated pipeline during development, and will have the capacity to respond quickly to unplanned releases.

The pipeline quality attributes used to evaluate the pipeline patterns in this research were Pipeline Suitability, Maintainability, Availability, Performance Efficiency, Reliability, and Security.

We have defined four complex attributes categories (Manual Operations, Automated Operations, Number of Possible pipelines, and Quality Gates), and we observed the following:

- The number of automated operations will contribute to increasing pipeline efficiency, reliability, availability, and security. On the other side, it will decrease the pipeline maintainability and readability, and also the suitability for the stakeholders.
- The number of manual operations will contribute to increasing the pipeline's suitability to the stakeholders but decrease the pipeline's reliability, efficiency, and availability. The manual operations do ´not affect the pipeline readability or maintainability.
- The number of pipelines contributes to lower pipeline maintainability, readability, and efficiency. However, the ability to separate the different needs in different pipelines increases the suitability for all stakeholders and pipeline availability. The number of pipelines doesn't have a direct impact on pipeline security and reliability).

- The number of quality gates contributes to increasing the pipeline reliability, availability, security, suitability for the stakeholders, and efficiency. The pipeline's needed code work to support the pipeline's control gates decreases the pipeline's readability and maintainability.

After executing the qualitative study, we can assume the following statements:

- The most stakeholder-suitable pipeline is the pipeline pattern supporting the git flow.
- The principal trunk-based pipeline is the more efficient and is the one with higher availability. It is also the one that guarantees more security.
- The pipeline pattern that supports inexperienced teams and the pipeline patterns that support Release Ready Mainline have very similar quality attributes distribution.
- The use of Feature Toggles instead of feature branches to control the enablement of features in the different environments contributes to a more efficient pipeline. Using feature branches contributes to increasing the availability of the pipeline.

6.2 System Limitations

There is a need to have more research studies with quality attributes of the pipeline that automates the delivery process as target artifacts in academia. Because of that, it is hard to find information submitted for a critical peer-review process that academic research is typically exposed to.

The complexity attributes lack a value for the variables, which can give the results presented in this research an error margin.

6.3 Future Work

In future work, we propose a more extensive study on the quality attributes of the pipeline, studying other possible quality attributes or contributing with more research in this domain.

There are also two new research themes in this study's scope that are worth exploring. The first proposed new research should be performed to calculate the values of the variables for the following complexity measures: Pull requests (p), Feature toggles (f), Approval of releases (a), and Approval (q).

The second is new research with the main objective to apply in an actual use case the complexity expressions expressed in Chapter 5 and then execute a qualitative and quantitative analysis of the results.

Bibliography

[1] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee, “A design science research methodology for information systems research,” *Journal of Management Information Systems*, vol. 24, pp. 45–77, 12 2007.

[2] D. Lopez-Fernandez, J. Diaz, J. Garcia-Martin, J. Perez, and A. Gonzalez-Prieto, “Devops team structures: Characterization and implications,” *IEEE Transactions on Software Engineering*, p. 1, 2021.

[3] M. Flower, “Patterns for managing source code branches,” 5 2020. [Online]. Available: https://martinfowler.com/articles/branching-patterns.html

[4] P. Perera, R. Silva, and I. Perera, “Improve software quality through practicing devops,” 2017, pp. 1–6.

[5] P. Debois, “Agile infrastructure and operations: How infra-gile are you?” 2008, pp. 202–207.

[6] F. Zampetti, S. Geremia, G. Bavota, and M. D. Penta, “Ci/cd pipelines evolution and restructuring: A qualitative and quantitative study,” 2021, pp. 471–482.

[7] M. M. I. Tarawneh, H. AL-Tarawneh, and A. Elsheikh, “Software development projects: An investigation into the factors that affect software project success/ failure in jordanian firms,” 2008, pp. 246–251.

[8] M. A. Akbar, S. Rafi, A. A. Alsanad, S. F. Qadri, A. Alsanad, and A. Alothaim, “Toward successful devops: A decision-making framework,” *IEEE Access*, vol. 10, pp. 51 343–51 362, 2022.

[9] A. Mishra and Z. Otaiwi, “Devops and software quality: A systematic mapping,” *Computer Science Review*, vol. 38, p. 100308, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1574013720304081

[10] S. T. March and G. F. Smith, “Design and natural science research on information technology,” *Decision Support Systems*, vol. 15, pp. 251–266, 1995. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0167923694000412

[11] A. Hevner, A. R, S. March, S. T, Park, J. Park, Ram, and Sudha, “Design science in information systems research,” *Management Information Systems Quarterly*, vol. 28, p. 75, 1 2004.

[12] B. Kitchenham and P. Brereton, “A systematic review of systematic review process research in software engineering,” *Information and Software Technology*, vol. 55, pp. 2049–2075, 2013. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584913001560

[13] L. Chen, “Continuous delivery: Huge benefits, but challenges too,” *IEEE Software*, vol. 32, pp. 50–54, 2015.

[14] “Ieee standard for devops:building reliable and secure systems including application build, package, and deployment,” *IEEE Std 2675-2021*, pp. 1–91, 2021.

[15] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, 1st ed. Addison-Wesley Professional, 2010.

[16] T. Savor, M. Douglas, M. Gentili, L. Williams, K. Beck, and M. Stumm, “Continuous deployment at facebook and oanda.” IEEE Computer Society, 5 2016, pp. 21–30.

[17] M. Young and M. Pezze, *Software Testing and Analysis: Process, Principles and Techniques*. John Wiley amp; Sons, Inc., 2005.

[18] M. M. Rahman and C. K. Roy, “Impact of continuous integration on code reviews,” 2017, pp. 499–502.

[19] A. Dyck, R. Penners, and H. Lichter, “Towards definitions for release engineering and devops.” Institute of Electrical and Electronics Engineers Inc., 2015, p. 3.

[20] M. Shahin, M. A. Babar, M. Zahedi, and L. Zhu, “Beyond continuous delivery: An empirical investigation of continuous deployment challenges,” 1 2017.

[21] M. Z. Toh, S. Sahibuddin, and M. N. Mahrin, “Adoption issues in devops from the perspective of continuous delivery pipeline.” Association for Computing Machinery, 2019, pp. 173–177. [Online]. Available: https://doi.org/10.1145/3316615.3316619

[22] N. Paez, “Versioning strategy for devops implementations,” 2018, pp. 1–6.

[23] C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano, “Devops,” *IEEE Software*, vol. 33, pp. 94–100, 2016.

[24] L. Bass, I. Weber, and L. Zhu, *DevOps: A Software Architect’s Perspective*. Addison-Wesley Professional, 2015.

[25] P. Zimmerer, “Strategy for continuous testing in idevops.” Association for Computing Machinery, 2018, pp. 532–533. [Online]. Available: https://doi.org/10.1145/3183440.3183465

[26] M. Virmani, “Understanding devops bridging the gap from continuous integration to continuous delivery.” 2015, pp. 78–78–82. [Online]. Available: https://search.ebscohost.com/login.aspx?direct=true&db=edb&AN=110068582&site=eds-live

[27] A. Capizzi, S. Distefano, M. Mazzara, L. J. P. Araùjo, M. Ahmad, and E. Bobrov, “Anomaly detection in devops toolchain,” 2019. [Online]. Available: http://arxiv.org/abs/1909.12682

[28] Y. Wang, M. Pyhäjärvi, and M. V. Mäntylä, “Test automation process improvement in a devops team: Experience report,” 2020, pp. 314–321.

[29] N. Kerzazi and I. E. Asri, “Release engineering: From structural to functional view.” Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3289402.3289547

[30] H. Inc., “8 deployment pattern structures to transform your ci/cd,” 2022.

[31] B. Son, “An architect’s guide to devops pipelines: Continuous integration continuous delivery (ci/cd),” 11 2020.

[32] C. Belyea, “The top 7 pipeline design patterns for continuous delivery,” 6 2020.

[33] Pasi, O. M. L. L. Ellen, and Kuvaja, “Dimensions of devops,” Torgeir, P. M. L. Casper, and Dingsøyr, Eds. Springer International Publishing, 2015, pp. 212–217.

[34] R. W. Macarthy and J. M. Bass, “An empirical taxonomy of devops in practice.” Institute of Electrical and Electronics Engineers Inc., 8 2020, pp. 221–228.

[35] A. Wahaballa, O. Wahballa, M. Abdellatief, H. Xiong, and Z. Qin, “Toward unified devops model,” vol. 2015-November. IEEE Computer Society, 11 2015, pp. 211–214.

[36] R. Jabbari, N. bin Ali, K. Petersen, and B. Tanveer, “What is devops? a systematic mapping study on definitions and practices.” Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2962695.2962707

[37] J. Meinicke, C.-P. Wong, B. Vasilescu, and C. Kästner, “Exploring differences and commonalities between feature flags and configuration options,” 2020, pp. 233–242.

[38] M. M. A. Ibrahim, S. M. Syed-Mohamad, and M. H. Husin, “Managing quality assurance challenges of devops through analytics.” Association for Computing Machinery, 2019, pp. 194–198. [Online]. Available: https://doi.org/10.1145/3316615.3316670

[39] I. Karamitsos, S. Albarhami, and C. Apostolopoulos, “Applying devops practices of continuous automation for machine learning,” *Information*, vol. 11, 2020. [Online]. Available: https://www.mdpi.com/2078-2489/11/7/363

[40] H. L. Akshaya, J. Vidya, and K. Veena, “A basic introduction to devops tools,” *International Journal of Computer Science Information Technologies*, vol. 6, pp. 5–6, 2015.

[41] D. Spinellis, “Git,” *IEEE Software*, vol. 29, pp. 100–101, 2012.

[42] S. Just, K. Herzig, J. Czerwonka, and B. Murphy, “Switching to git: The good, the bad, and the ugly,” 2016, pp. 400–411.

[43] E. R. A. N. D. W. G. B. J. D. and Davenport, “A quick introduction to version control with git and github,” *PLOS Computational Biology*, vol. 12, pp. 1–18, 9 2016. [Online]. Available: https://doi.org/10.1371/journal.pcbi.1004668

[44] S. McIntosh, B. Adams, T. H. D. Nguyen, Y. Kamei, and A. E. Hassan, “An empirical study of build maintenance effort,” 2011, pp. 141–150.

[45] P. Runeson, “A survey of unit testing practices,” *IEEE Software*, vol. 23, pp. 22–29, 2006.

[46] F. Häser, M. Felderer, and R. Breu, “Software paradigms, assessment types and non-functional requirements in model-based integration testing: A systematic literature review.” Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2601248.2601257

[47] W. T. Tsai, X. Bai, R. Paul, W. Shao, and V. Agarwal, “End-to-end integration testing design,” 2001, pp. 166–171.

[48] C. AGUTTER, *ITIL® Foundation Essentials – ITIL 4 Edition: The ultimate revision guide*, 2nd ed. IT Governance Publishing, 2019. [Online]. Available: http://www.jstor.org/stable/j.ctvckq658

[49] P. Sharma, L. Chaufournier, P. Shenoy, and Y. C. Tay, “Containers and virtual machines at scale: A comparative study.” Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2988336.2988337

[50] C. Engle, J. Brewster, and G. Blokdijk, *ISO/IEC 20000 Certification and Implementation Guide - Standard Introduction, Tips for Successful ISO/IEC 20000 Certification, FAQs, Mapping Responsibilities, Terms, Definitions and ISO 20000 Acronyms*. Emereo Pty Ltd, 2008.

[51] T. F. Düllmann, C. Paule, and A. van Hoorn, “Exploiting devops practices for dependable and secure continuous delivery pipelines,” 2018, pp. 27–30.

[52] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 3rd ed. Addison-Wesley Professional, 2012.

[53] J. Miguel, D. Mauricio, and G. Rodriguez, “A review of software quality models for the evaluation of software products,” *International journal of Software Engineering Applications*, vol. 5, pp. 31–54, 1 2014.

[54] I. 25010, “Iso/iec 25010:2011, systems and software engineering — systems and software quality requirements and evaluation (square) — system and software quality models,” 2011.

[55] A. A. U. Rahman and L. Williams, “Security practices in devops.” Association for Computing Machinery, 2016, pp. 109–111. [Online]. Available: https://doi.org/10.1145/2898375.2898383

[56] ——, “Security practices in devops.” Association for Computing Machinery, 2016, pp. 109–111. [Online]. Available: https://doi.org/10.1145/2898375.2898383

A
Code of Project

In this Annex, we provide all the code used in the development of this thesis

A.0.1 Trunk-Based Pipelines Code

Listing A.1: Simple Trunk-Based Driven Pipeline Example.

```

name: CI/CD

on: [push]

jobs:
  development_evironment:
    name: Development Enviroment
    runs-on: ubuntu-latest
    environment:
      name: development
      url: https://github.com
    steps:
    # Clone of the code
    - uses: actions/checkout@v1
      name: Clone code
    - uses: cardinalby/export-env-action@v1
      name: Load Feature toogles
      with:
        envFile: './env/development/config.env'
        expand: 'true'
    - name: Linting
      run: echo Linting code.
    - uses: actions/setup-python@v2
      name : Build
    - name: Unit Testing
      run: echo Run Unit testing
    - uses: jannekem/run-python-script-action@v1
      name : Run Code
      with:
       script: |
          import os
          if os.environ.get('FEATURE1') == "True":
              print(os.environ.get('FEATURE1'))
          if os.environ.get('FEATURE2') == "True":
              print("Running FEATURE2 a")
  deployment_integration:
    name: Deploy in Integration Environment
    needs: development_evironment
    runs-on: ubuntu-latest
    environment:
      name: integration
      url: https://github.com
    steps:
      - name: Deploy in Integration
        run: echo Coding is deployed
  integration_enviroment:
    name: Integration Environment
    needs: deployment_integration
    environment:
      name: integration
      url: https://github.com
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v1
      name: Clone code
    - uses: cardinalby/export-env-action@v1
      name: Load Feature toogles
      with:
        envFile: './env/integration/config.env'
        expand: 'true'
    - uses: jannekem/run-python-script-action@v1
      name : Run Code
      with:
       script: |
          import os
          if os.environ.get('FEATURE1') == "True":
              print(os.environ.get('FEATURE1'))
          if os.environ.get('FEATURE2') == "True":
              print("Running FEATURE2")
    - name: Integration Testing
      run: echo Run Unit testing
  deployment_testing:
    name: Deploy in Testing Environment
    needs: integration_enviroment
    runs-on: ubuntu-latest
    environment:
      name: testing
      url: https://github.com
    steps:
      - name: deploy
        run: echo Coding is deployed
  testing_enviroment:
    name: Testing Environment
    needs: deployment_testing
    runs-on: ubuntu-latest
    environment:
      name: testing
      url: https://github.com
    steps:
    - uses: actions/checkout@v1
      name: Clone code
    - uses: cardinalby/export-env-action@v1
      name: Load Feature toogles
      with:
        # Will need to set for each environment path for the config-env
        # containing all the feature toggles with the key-value definitions
        envFile: './env/testing/config.env'
        expand: 'true'
    - uses: jannekem/run-python-script-action@v1
      name : Run Code
      with:
       script: |
          # Import the from the library to be able to use the Environment Variables
          import os
          # If the Feature Toggle represented by "Feature1" is enable
          # will print the text inside the function print()
          if os.environ.get('FEATURE1') == "True":
              print("Running FEATURE1")
          # If the Feature Toggle represented by "Feature2" is enabled will print
          # the text inside the function print()
          if os.environ.get('FEATURE2') == "True":
              print("Running FEATURE2")
    - name: Non-Functional Testing
      run: echo Run Non-Functional Testing
    - name: Functional Testing
      run: echo Run Functional Testing
  deployment_staging:
    name: Deploy in Staging Environment
    needs: testing_enviroment
    runs-on: ubuntu-latest
    environment:
      name: staging
      url: https://github.com
    steps:
      - name: deploy
        run: echo Coding is deployed
  staging_enviroment:
    name: Staging Environment
    needs: deployment_staging
    runs-on: ubuntu-latest
    environment:
      name: staging
      url: https://github.com
    steps:
    - uses: actions/checkout@v1
      name: Clone code
    - uses: cardinalby/export-env-action@v1
      name: Load Feature toggles
      with:
        envFile: './env/staging/config.env'
        expand: 'true'
    - uses: jannekem/run-python-script-action@v1
      name : Run Code
      with:
       script: |
          import os
          if os.environ.get('FEATURE1') == "True":
              print(os.environ.get('FEATURE1'))
          if os.environ.get('FEATURE2') == "True":
              print("Running FEATURE2")
    - name: Non-Functional Testing
      run: echo Run Non-Functional Testing
    - name: Functional Testing
      run: echo Run Functional Testing
  release:
      name: Release in Production
      needs: staging_enviroment
      runs-on: ubuntu-latest
      environment:
        name: Prodution
        url: https://github.com
      steps:
        - name: Run Code in Production
          run: echo Coding is deployed
```

Listing A.2: Trunk-Based Driven Pipeline with complex tests Example.

```

name: CI/CD With Complex Tests

on: [push]

jobs:
  development_evironment:
    name: Development Enviroment
    runs-on: ubuntu-latest
    environment:
      name: development
      url: https://github.com
    steps:
    # Clone of the code
    - uses: actions/checkout@v1
      name: Clone code
    - uses: cardinalby/export-env-action@v1
      name: Load Feature toogles
      with:
        envFile: './env/development/config.env'
        expand: 'true'
    - name: Linting
      run: echo Linting code.
    - uses: actions/setup-python@v2
      name : Build
    - name: Unit Testing
      run: echo Run Unit testing
    - uses: jannekem/run-python-script-action@v1
      name : Run Code
      with:
       script: |
          import os
          if os.environ.get('FEATURE1') == "True":
              print(os.environ.get('FEATURE1'))
          if os.environ.get('FEATURE2') == "True":
              print("Running FEATURE2 a")
  deployment_integration:
    name: Deploy in Integration Environment
    needs: development_evironment
    runs-on: ubuntu-latest
    environment:
      name: integration
      url: https://github.com
    steps:
      - name: Deploy in Integration
        run: echo Coding is deployed
  integration_enviroment:
    name: Integration Environment
    needs: deployment_integration
    runs-on: ubuntu-latest
    environment:
      name: integration
      url: https://github.com
    steps:
    - uses: actions/checkout@v1
      name: Clone code
    - uses: cardinalby/export-env-action@v1
      name: Load Feature toogles
      with:
        envFile: './env/integration/config.env'
        expand: 'true'
    - uses: jannekem/run-python-script-action@v1
      name : Run Code
      with:
       script: |
          import os
          if os.environ.get('FEATURE1') == "True":
              print(os.environ.get('FEATURE1'))
          if os.environ.get('FEATURE2') == "True":
              print("Running FEATURE2")
    - name: Integration Testing
      run: echo Run Unit testing
  # Link to the scheduled pipeline
  complex_tests:
    name: Run Complex Tests
    runs-on: ubuntu-latest
    environment:
      name: integration
      url: https://github.com
    steps:
    - uses: cardinalby/schedule-job-action@v1
      with:
        ghToken: ${{ secrets.WORKFLOWS_TOKEN }}
        templateYmlFile: '.github-scheduled-workflows/complex_test_example.yml'
  deployment_testing:
    name: Deploy in Testing Environment
    # Configuration of the dependency in the scheduled pipeline and principal
    needs: [integration_enviroment,complex_tests]
    runs-on: ubuntu-latest
    environment:
      name: testing
      url: https://github.com
    steps:
      - name: deploy
        run: echo Coding is deployed
  testing_enviroment:
    name: Testing Environment
    needs: deployment_testing
    environment:
      name: testing
      url: https://github.com
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v1
      name: Clone code
    - uses: cardinalby/export-env-action@v1
      name: Load Feature toogles
      with:
        # Will need to set for each environment the path for the config-env
        # containing all the feature toogles with the key-value definitions
        envFile: './env/testing/config.env'
        expand: 'true'
    - uses: jannekem/run-python-script-action@v1
      name : Run Code
      with:
       script: |
          # Import the os library to be able to use the Environment Variables
          import os
          # If the Feature Toggle represented by "Feature1" is enable
          # will print the text inside the function print()
          if os.environ.get('FEATURE1') == "True":
              print("Running FEATURE1")
          # If the Feature Toggle represented by "Feature2" is enable
          # will print the text inside the function print()
          if os.environ.get('FEATURE2') == "True":
              print("Running FEATURE2")
    - name: Non-Functional Testing
      run: echo Run Non-Functinal Testing
    - name: Functional Testing
      run: echo Run Functinal Testing
  deployment_staging:
    name: Deploy in Staging Environment
    needs: testing_enviroment
    runs-on: ubuntu-latest
    environment:
      name: staging
      url: https://github.com
    steps:
      - name: deploy
        run: echo Coding is deployed
  staging_enviroment:
    name: Staging Environment
    needs: deployment_staging
    environment:
      name: staging
      url: https://github.com
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v1
      name: Clone code
    - uses: cardinalby/export-env-action@v1
      name: Load Feature toogles
      with:
        envFile: './env/staging/config.env'
        expand: 'true'
    - uses: jannekem/run-python-script-action@v1
      name : Run Code
      with:
       script: |
          import os
          if os.environ.get('FEATURE1') == "True":
              print(os.environ.get('FEATURE1'))
          if os.environ.get('FEATURE2') == "True":
              print("Running FEATURE2")
    - name: Non-Functional Testing
      run: echo Run Non-Functinal Testing
    - name: Functional Testing
      run: echo Run Functinal Testing
  release:
      name: Release in Production
      needs: staging_enviroment
      runs-on: ubuntu-latest
      environment:
        name: prodution
        url: https://github.com
      steps:
        - name: Run Code in Production
          run: echo Coding is deployed
```

Listing A.3: Trunk-Based Driven Parallel Pipeline with complex tests Example.

```

name: CI/CD With Complex Tests

```

```
on:
# In this example the pipeline runs  every 15th minute, but this value can be changed
  schedule:
    - cron:  '*/15 * * * *'

jobs:
  name: Run Complex Tests
      runs-on: ubuntu-latest
      environment:
        name: testing
        url: https://github.com
      steps:
        - uses: actions/checkout@v2
        - name: Running Complex tests
          run: echo Test are running
```

A.0.2 Feature-Based Pipelines Code

Listing A.4: Simple Feature branch Mainline Pipeline Example.

```

name: Mainline Pipeline
on:
  # This Definition sets the trigger of the CD pipeline only
  # when the code is pushed to the mainline
  push:
    branches: [main]
  # and also when the pull request is approved and, the code merged in the mainline
  pull_request:
    branches: [main]
jobs:
  integration_enviroment:
    name: Integration Environment
    environment:
      name: integration
      url: https://github.com
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v1
      name: Clone code
    - name: Linting
      run: echo Linting code.
    - uses: actions/setup-python@v2
      name : Build
    - name: Unit Testing
      run: echo Run Unit testing
    - uses: jannekem/run-python-script-action@v1
      name : Run Code
      with:
       script: |
          print("Hello World")
    - name: Integration Testing
      run: echo Run Unit testing
  deployment_testing:
    name: Deploy in Testing Environment
    needs: integration_enviroment
    runs-on: ubuntu-latest
    environment:
      name: testing
      url: https://github.com
    steps:
      - name: deploy
        run: echo Coding is deployed
  testing_enviroment:
    name: Testing Environment
    needs: deployment_testing
    environment:
      name: testing
      url: https://github.com
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v1
      name: Clone code
    - uses: jannekem/run-python-script-action@v1
      name : Run Code
      with:
       script: |
         print("Hello World")
    - name: Non-Functional Testing
      run: echo Run Non-Functional Testing
    - name: Functional Testing
      run: echo Run Functional Testing
  deployment_staging:
    name: Deploy in Staging Environment
    needs: testing_enviroment
    runs-on: ubuntu-latest
    environment:
      name: staging
      url: https://github.com
    steps:
      - name: deploy
        run: echo Coding is deployed
  staging_enviroment:
    name: Staging Environment
    needs: deployment_staging
    environment:
      name: staging
      url: https://github.com
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v1
      name: Clone code
    - uses: jannekem/run-python-script-action@v1
      name : Run Code
      with:
       script: |
          print("Hello World")
    - name: Non-Functional Testing
      run: echo Run Non-Functinal Testing
    - name: Functional Testing
      run: echo Run Functinal Testing
  release:
      name: Release in Production
```

```
      needs: staging_enviroment
      runs-on: ubuntu-latest
      # Using the environment tag, we can configure the place
      # where the code will be deployed
      # and the need for permissions to approve the Releases/Deployment
      # of the code in the environment
      environment:
        name: prodution
        url: https://github.com
      steps:
        - name: Run Code in Production
          run: echo Coding is deployed
```

Listing A.5: Simple Feature branch Pipeline Example.

```

name: Feature Pipeline
on:
# This Definition sets the trigger of the CI pipeline
# only when the code is pushed to the feature branch
  push:
    branches:
      - feature/**
jobs:
  development_evironment:
    name: Development Environment
    environment:
      name: development
      url: https://github.com
    runs-on: ubuntu-latest
    steps:
    # Clone of the code
    - uses: actions/checkout@v1
      name: Clone code
    - name: Linting
      run: echo Linting code.
    - uses: actions/setup-python@v2
      name : Build
    - name: Unit Testing
      run: echo Run Unit testing
    - uses: jannekem/run-python-script-action@v1
      name : Run Code
      with:
      # This is a simple python script that prints in the console
      # the values present inside the function print()
       script: |
              print("Hello World")
              print("Hello Feature 1")
```