INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Processo de Integração e Entrega Contínua para Aplicações Baseadas em Análise de Dados

BRUNO DA PONTE VILAR
Outubro de 2023

POLITÉCNICO
DO PORTO

Instituto Superior de Engenharia do Porto

Processo de Integração e Entrega Contínua para Aplicações Baseadas em Análise de Dados

Bruno da Ponte Vilar

Dissertação para obtenção do Grau de Mestre em Engenharia Informática, Área de Especialização em Engenharia de Software

Orientador: Doutor António Constantino Lopes Martins

Porto, outubro 2023

Declaração de Integridade

Declaro ter conduzido este trabalho académico com integridade.

Não plagiei ou apliquei qualquer forma de uso indevido de informações ou falsificação de resultados ao longo do processo que levou à sua elaboração.

Portanto, o trabalho apresentado neste documento é original e de minha autoria, não tendo sido utilizado anteriormente para nenhum outro fim.

Declaro ainda que tenho pleno conhecimento do Código de Conduta Ética do P.PORTO.

ISEP, Porto, 14 de outubro de 2023

Resumo

Processos de entrega de *software* tradicionais não são eficientes para uma empresa que produz *software* se manter competitiva. A crescente necessidade de entregar *software* em ciclos cada vez mais curtos fomentou o crescimento de abordagens e tecnologias que permitam automatizar este tipo de processos.

No entanto, estas abordagens apresentam limitações quando a qualidade de entrega do *software* depende da qualidade dos dados que o mesmo produz.

Com isso em mente, o principal objetivo desta dissertação consiste em adotar um mecanismo de entrega de *software* automático, eficaz, eficiente e fiável dentro da *proGrow S.A.* de forma a que o seu principal negócio, que é a geração de informação a partir de dados recolhidos, seja feito da forma mais eficiente e fiável possível.

Após a análise detalhada do estado de arte atual, foi elaborada uma solução que implementa uma *pipeline* de implantação que incluí uma estratégia de testes que desafia o paradigma de testes tradicional, projetada para o contexto das soluções desenvolvidas. Deste modo, a solução é capaz de entregar *software* automaticamente, garantindo a sua qualidade através de uma estratégia de testes robusta, capaz de validar a qualidade da aplicação através dos seus dados.

Assim, a solução desenvolvida é capaz de reduzir o tempo de desenvolvimento de uma nova versão, bem como garantir que o processo de escrutínio para deteção de erros nos dados gerados é feito de forma automática e repetível, com o objetivo de criar um padrão de qualidade nos dados gerados.

Palavras-chave: Entrega contínua, integração contínua, automatização, testes, qualidade, implantação, *pipeline* de implantação

Abstract

Traditional software delivery processes are not efficient for a company that produces software to remain competitive. The growing need to deliver software in ever shorter cycles has fostered the growth of approaches and technologies that make it possible to automate this type of process.

However, these approaches have limitations when the quality of software delivery depends on the quality of the data it produces.

With this in mind, the main goal of this dissertation is to adopt an automatic, effective, efficient and reliable software delivery mechanism within proGrow S.A. so that its core business, which is generating information from collected data, is done in the most efficient and reliable way possible.

After a detailed analysis of the current state of the art, a solution was devised that implements a deployment pipeline that includes a testing strategy that challenges the traditional testing paradigm, designed for the context of the solutions developed. In this way, the solution is able to deliver software automatically, guaranteeing its quality through a robust testing strategy, capable of validating the quality of the application through its data.

Thus, the solution developed is able to reduce the development time of a new version, as well as ensuring that the scrutiny process for detecting errors in the data generated is done automatically and repeatably, with the aim of creating a quality standard in the data generated.

Keywords: Continuous delivery, continuous integration, automation, tests, quality, deploy, deployment pipeline

Agradecimentos

Em primeiro lugar, queria começar por agradecer aos meus pais, Adélia Vilar e Paulo Vilar por todo o apoio dado no meu percurso académico.

Queria também agradecer a todos os docentes do ISEP que me acompanharam, em especial ao professor António Martins por todo apoio prestado não só durante esta dissertação como em diversos outros momentos, sem a sua mentoria não seria possível entregar esta dissertação.

Quero agradecer também à *proGrow S.A.* como um todo, em especial ao meu supervisor Fábio Santos que se mostrou sempre disponível para me ajudar durante toda a dissertação, tendo tido um contributo crucial para a sua elaboração. Queria também agradecer ao Pedro Cunha e ao Tiago Abreu, pela disponibilidade demonstrada ao longo deste projeto.

Finalmente, gostaria de agradecer a todos os meus colegas de curso que, de uma forma ou de outra, contribuíram para esta longa jornada.

Índice

1 Introdução 1

1.1 Contexto/Problema 1

1.2 Objetivos 2

1.3 Motivação 2

1.4 Resultados Esperados 2

1.5 Metodologia de Investigação 3

1.5.1 Questões de Investigação 4

1.5.2 Revisão de Literatura 4

1.5.3 Estratégias de Investigação 5

1.5.4 Métodos de Geração de dados 5

1.5.5 Análise de Dados 6

1.6 Estrutura do Documento 6

2 Estado de Arte a Nível Conceptual 7

2.1 Principais Conceitos 7

2.1.1 Engenharia de Software Contínua 7

2.1.2 Integração Contínua 7

2.1.3 Entrega Contínua 8

2.1.4 Implantação Contínua 9

2.2 Más Práticas Habituais de Entrega de Software 10

2.2.1 Implantação Manual de Software 10

2.2.2 Implantação Tardia em Staging 10

2.2.3 Configuração Manual de Ambientes de Produção 11

2.3 Boas Práticas de Entrega de Software 11

2.3.1 Processo Repetível e Fiável 11

2.3.2 Automatizar o Máximo Possível 12

2.3.3 Manter Tudo no Controlo de Versões 12

2.3.4 Se Custa, Deve Ser Mais Frequente e Mais Cedo 12

2.3.5 Build Quality In 13

2.3.6 Feito Significa Entregue 13

2.3.7 Todos São Responsáveis Pela Entrega 13

2.3.8 Planeamento e Documentação 14
2.3.9 Melhoria Contínua 14
2.4 Testes 15
2.4.1 Testes Funcionais de aceitação 16
2.4.2 Testes Unitários 17
2.4.3 Testes de Integração 17
2.4.4 Testes de implantação 18
2.4.5 Showcases 18
2.4.6 Testes Exploratórios 18
2.4.7 Testes de Usabilidade 19
2.4.8 Testes de Aceitação Não Funcionais 19
2.4.9 Testes em Aplicações Baseadas na Geração de Nova Informação 20
2.5 Pipeline de Implantação 22
2.5.1 Boas Práticas de Pipelines de Implantação 24
2.6 Scripting de Compilação e Implantação 26
2.6.1 Princípios e Práticas de *Scripting* de Compilação e Implantação 27
2.6.2 Scripting de Implantação 28
2.7 Estratégias de Implantação e Lançamento de Software 30
2.7.1 Implantações *Blue-Green* 31
2.7.2 Lançamento *Canary* 32
2.8 Problemas e Soluções na Adoção de Entrega Contínua 33
2.8.1 Problemas e Soluções de Desenho de Compilação 34
2.8.2 Problemas e Soluções de Desenho do Sistema 34
2.8.3 Problemas e Soluções de Integração 35
2.8.4 Problemas e Soluções de Testes 35
2.8.5 Problemas e Soluções de Lançamento de Software 36
2.9 Casos de Sucesso 37
2.9.1 Caso da HP FutureSmart 37
2.9.2 Caso do Programa de Simplificação da *Suncorp* 37
2.10 Sumário 38
3 Estado de Arte a Nível Tecnológico 39
3.1 Ferramentas de Integração Contínua e Entrega Contínua 39
3.1.1 Jenkins 39
3.1.2 GitHub Actions 40
3.1.3 Team City 40

3.1.4 Comparação entre Jenkins, GitHub Actions e TeamCity 41
3.2 Ferramentas de Testes 41
3.2.1 Jest 42
3.2.2 Mocha 42
3.2.3 Comparação entre Jest e Mocha 42
3.2.4 Selenium 43
3.2.5 Cypress 43
3.2.6 Comparação entre Selenium e Cypress 44
3.2.7 JMeter 45
3.2.8 K6 45
3.2.9 Comparação entre JMeter e K6 46
3.3 Ferramentas Associadas à Implantação 46
3.3.1 Docker 46
3.3.2 Repositório Nexus 47
3.3.3 Amazon Web Services 48
3.3.4 Terraform 49
3.4 Sumário 50
4 Análise de Valor 51
4.1 Processo de Inovação 51
4.1.1 Identificação da Oportunidade 52
4.1.2 Análise da oportunidade 53
4.2 Proposta de Valor 54
4.3 Técnica Sistemática de Análise Funcional 55
4.4 Método de Análise Hierárquica 56
4.5 Sumário 62
5 Análise e *Design* 63
5.1 Análise 63
5.1.1 Requisitos Funcionais 63
5.1.2 Requisitos Não Funcionais 64
5.2 Design 67
5.2.1 Processo Atual 67
5.2.2 Arquitetura 69
5.2.3 Alternativa Arquitetural 76

5.3 Sumário 77

6 Implementação 79

6.1 Estratégia de Testes 79
6.1.1 Testes Unitários 81
6.1.2 Testes de Integração 83
6.1.3 Testes de Aceitação Funcionais 84
6.1.4 Testes de implantação 89

6.2 Infraestrutura 90
6.2.1 Task Definition 90
6.2.2 Serviço ECS 91
6.2.3 Cluster ECS 93
6.2.4 Load Balancer 94

6.3 Pipeline de Implantação 95
6.3.1 Implementação dos Triggers 96
6.3.2 Construção e publicação de Artefactos 97
6.3.3 Testes Automáticos 98
6.3.4 Implantação 100
6.3.5 Testes Manuais 101
6.3.6 Sumário 102

7 Avaliação 103

7.1 Hipótese de Investigação 103

7.2 Indicadores e Fontes de Informação 103

7.3 Metodologia de Avaliação 104

7.4 Avaliação de Resultados 105
7.4.1 Avaliação de Métricas Recolhidas 106
7.4.2 Entrevista Semiestruturada 108
7.4.3 Comparação de Modelos de Maturidade 110
7.4.4 Conclusões da Avaliação 111

8 Conclusões 113

8.1 Objetivos Alcançados 113

8.2 Objetivos Não Alcançados 114

8.3 Trabalho Futuro 114

Lista de Figuras

Figura 1 - Processo de pesquisa(Oates, 2005) 3
Figura 2 – Processo de entrega contínua (Zaiku, 2017) 8
Figura 3 – Diagrama de tipos de teste (Humble and Farley, 2010) 15
Figura 4 – Funcionamento de uma *pipeline* de implantação (Humble and Farley, 2010) 22
Figura 5 – Diagrama de conflitos de *stages* (Humble and Farley, 2010) 25
Figura 6 – Implantação como camadas (Humble and Farley, 2010) 29
Figura 7 – Diagrama do processo de implantação do ambiente e aplicação (Humble and Farley, 2010) 30
Figura 8 – Diagrama com implantação *Blue-Green* 8 (Humble and Farley, 2010) 31
Figura 9 – Implantação *Canary* (Humble and Farley, 2010) 32
Figura 10 – Casos por categorias de problemas na adoção de entrega contínua (Laukkanen, Itkonen and Lassenius, 2017) 33
Figura 11 – Criação de um contentor (MrDevSecOps, 2023) 47
Figura 12 – Ciclo de vida de aplicação ECS (Computer Consulting, 2023) 49
Figura 13- Processo de inovação (Koen *et al.*, 1996) 51
Figura 14 – Diagrama de modelo de desenvolvimento de um novo conceito (Koen *et al.*, 1996) 52
Figura 15 – Proposta de Valor 54
Figura 16 – Diagrama *FAST* 55
Figura 17 – Árvore hierárquica de decisão para seleção de ferramenta CI/CD 57
Figura 18 – Diagrama de casos de uso 64
Figura 19 – Diagrama BPMN do processo de lançamento de versão atual 68
Figura 20 - Diagrama de atividades do processo de *release* novo 69
Figura 21 - Diagrama BPMN da *pipeline* de implantação 71
Figura 22 – Diagrama de sequência “Testes de Aceitação Funcionais” 74
Figura 23 – Diagrama de atividades “Estratégia de Implantação” 75
Figura 24 - Diagrama BPMN da *pipeline* de implantação alternativa 76
Figura 25 – Estrutura de ficheiros do ETL Core 79
Figura 26 – Resultado da execução da estratégia de testes unitários 82

Figura 27 -Resultados de execução da estratégia de testes de integração 84
Figura 28 - Resultados de execução da estratégia de testes de aceitação funcionais 88
Figura 29 - Modelo de Maturidade 104

Lista de Tabelas

Tabela 1 - Problemas de integração 35
Tabela 2 - Problemas de testes 36
Tabela 3 - Comparação *Jenkins vs Gitlab CI/CD vs TeamCity* 41
Tabela 4 - Comparação *Selenium vs Cypress* 44
Tabela 5 - Comparação *JMeter vs K6* 46
Tabela 6 – Comparação de critérios 57
Tabela 7 – Normalização dos valores da matriz de comparação e prioridade relativa 58
Tabela 8 - Valores de índice aleatório para matrizes quadradas de ordem n 59
Tabela 9 - Matriz de comparação do critério de facilidade de aprendizagem e utilização 60
Tabela 10 - Matriz normalizada e prioridade relativa do critério de facilidade de aprendizagem e utilização 60
Tabela 11 - Matriz de comparação do critério de funcionalidades e extensibilidade 61
Tabela 12 - Matriz normalizada e prioridade relativa do critério de funcionalidades e extensibilidade 61
Tabela 13 - Matriz de comparação do critério de custo 61
Tabela 14 - Matriz normalizada e prioridade relativa do critério de custo 62
Tabela 15 – Entrevista semiestruturada 109

Lista de Excertos de Código

Excerto de código 1 – Configuração do Jest 80
Excerto de código 2 - Teste unitário geração de *select custom* 81
Excerto de código 3 – Teste de aceitação à API do *ETL Core* 83
Excerto de código 4 – Exemplo de configuração de ficheiro de entrada JSON 85
Excerto de código 5 – Geração de teste funcional 86
Excerto de código 6 – Inserção de dados 86
Excerto de código 7 – *Strategy* de funções 87
Excerto de código 8 – Comparação de resultados 88
Excerto de código 9 – Testes de implantação 89
Excerto de código 10 – *Task Definition ETL Core* 91
Excerto de código 11 – Configuração serviço *ECS* 92
Excerto de código 12 – Configuração *auto scaling group* 93
Excerto de código 13 – Configuração *health check* 95
Excerto de código 14 – *Trigger* de *push* no ramo de desenvolvimento 96
Excerto de código 15 – *Trigger* de *push* de *tag* de uma versão candidata 96
Excerto de código 16 – *Trigger* manual 97
Excerto de código 17 – Compilação do *software* 97
Excerto de código 18 – Guardar artefactos 98
Excerto de código 19 – *Stage* de construção e publicação de artefactos 98
Excerto de código 20 – Execução de testes unitários 99
Excerto de código 21 – Execução de testes de aceitação funcionais 99
Excerto de código 22 – *Stage* de implantação 100
Excerto de código 23 – *Stage* de testes manuais 101

Lista de Equações

(1) 57
(2) 58
(3) 58
(4) 58
(5) 59
(6) 59
(7) 59
(8) 59
(9) 62

Acrónimos e Símbolos

Lista de Acrónimos

AWS *Amazon Web Services*

AHP *Analytic Hierarchy Process* (Processo de Análise Hierárquica)

API *Application Programming Interface* (Interface de Programação de Aplicação)

CD *Continuous Delivery* (Entrega Contínua)

CI *Continuos Integration* (Integração Contínua)

EC2 *Elastic Compute Cloud*

ECS *Elastic Container Service*

FAST *Function Analysis System Technique*

FFE *Fuzzy Front End*

HTTP *Hypertext Transfer Protocol*

IP *Internet Protocol* (Protocolo de Rede)

JSON *JavaScript Object Notation*

MR *Metamorphic Relation*

MT *Metamorphic Testing*

NCD *New Concept Development Model* (Modelo de Desenvolvimento de um Novo Conceito*)*

QA *Quality Assurance* (Garantia de Qualidade)

SQL *Structured Query Language*

URI *Uniform Resource Identifier* (Identificador Uniforme de Recurso)

1 Introdução

Neste capítulo introdutório, é feito o enquadramento deste projeto, tendo em conta a sua validade técnico-científica assim como o seu valor socioeconómico. É apresentado o problema em detalhe, quais os objetivos a serem atingidos, a motivação pela escolha deste tema e os principais contributos.

1.1 Contexto/Problema

A inerente complexidade no desenvolvimento de *software*, assim como a crescente exigência na entrega de *software* fiável a um elevado ritmo, faz emergir o problema de como entregar *software* o mais depressa possível ao utilizador final.

Assim, as soluções de automatização de processos ganham cada vez mais notoriedade.

Ao nível de engenharia de *software*, existem diversas ferramentas que automatizam os testes, o lançamento de versões e a disponibilização das mesmas ao utilizador final.

Apesar destas ferramentas terem um elevado leque de funcionalidades incorporadas infelizmente não respondem às necessidades de todo o tipo de aplicações desenvolvidas. Em específico de aplicações que trabalham com dados e que o objetivo é garantir que a informação que produzem é correta.

Além da limitação anterior, a adesão a métodos de entrega de *software* automáticos pelas organizações consiste num procedimento demoroso e delicado, enfrentando obstáculos como: ambientes complexos, variados problemas organizacionais e a falta de testes automáticos.

Posto isto, a *proGrow S.A.* pretende automatizar a entrega do seu serviço de análise de dados, chamado *ETL Core*, garantindo a sua qualidade. Este serviço trata-se de uma aplicação de agendamento e processamento de tarefas que, essencialmente, recebe dados, processa-os através de diferentes tipos de operações distintas (como agregações e transformações) e produz indicadores.

1.2 Objetivos

O principal objetivo desta dissertação consiste em desenvolver uma solução que garanta a automatização da entrega de *software* baseado em análises de dados, garantindo a qualidade desta.

De modo a atingir este objetivo, será necessário cumprir com cada um dos seguintes objetivos complementares:

- Estudar estratégias de testes, abordagens e soluções de automatização de entrega de *software* existentes;
- Desenhar e implementar uma solução de entrega de *software* robusta, com uma estratégia de testes adequada, que adote as melhores práticas da área;
- Avaliar a solução com o intuito de validar o quão adequada esta se encontra para resolver o problema.

1.3 Motivação

A motivação para o desenvolvimento do tema desta dissertação reside em aumentar a capacidade que a *proGrow S.A.* precisa de ter para disponibilizar a todos os seus clientes a sua solução de forma eficaz e eficiente garantindo o seu correto funcionamento.

Além do impacto para a organização, a crescente importância dada à entrega de *software*, assim como a dificuldade inerente associada a este desafio, tornam esta proposta de dissertação aliciante.

1.4 Resultados Esperados

Espera-se que o resultado desta tese reduza a quantidade de trabalho necessária para atingir uma entrega de *software* bem-sucedida, contribuindo para um maior ganho de produtividade na equipa e consequentemente na empresa.

Adicionalmente, é esperado que as entregas contínuas possuam uma qualidade superior, uma vez que o processo terá uma estratégia de testes automáticos.

Finalmente, devido à inerente automatização do novo processo, tanto o tempo de desenvolvimento quanto o tempo entre entregas poderão se tornar mais curtos, já que este processo é menos custoso e mais fiável.

1.5 Metodologia de Investigação

A metodologia de investigação adotada no desenvolvimento desta tese consiste num processo composto por múltiplas fases ilustradas na figura 1.

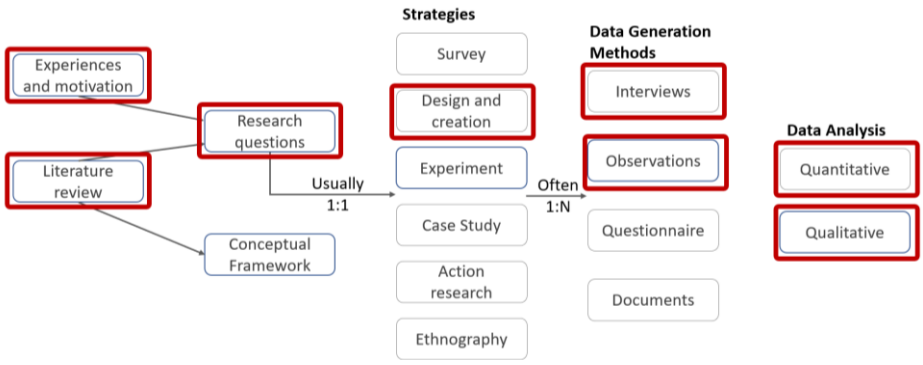

Figura 1 - Processo de pesquisa (Oates, 2005)

Analisando em detalhe o diagrama, é possível perceber o processo de investigação, no âmbito desta dissertação, centra-se nos seguintes tópicos (Pereira, 2022a):

- Seleção de um tópico, sendo este item já abordado amplamente nas secções Contexto/Problema e Motivação;
- Questões de investigação, que guiarão o processo de investigação;
- A revisão da literatura, que consiste na literatura que auxiliará nas respostas às questões de investigação;
- A estratégia de investigação utilizada, que visa resolver o problema de investigação;
- Os métodos de geração de dados, que permitem suportar ou refutar as teorias e hipóteses formuladas;
- A análise de dados, que permite obter conclusões através dos dados obtidos.

Posto isto, nesta secção serão analisados os tópicos mencionados anteriormente que ainda não foram abordados.

1.5.1 Questões de Investigação

Um ponto crítico no sucesso de uma investigação está intimamente ligado à seleção das questões que guiarão todo este processo.

Assim, as principais questões de investigação, cujo este documento visa responder são:

- Quais são as características importantes de uma abordagem de entrega de *software*?
- Como é possível automatizar/melhorar o processo de entrega de *software*?
- Como é possível garantir a qualidade do *software* a ser entregue?
- Que tecnologias devem ser adotadas de forma a atingir uma entrega de *software* automática que garanta a qualidade deste?
- Existe alguma diferença entre as abordagens de entrega de *software* tradicionais quando comparadas a entrega de *software* baseado em análise de dados?

1.5.2 Revisão de Literatura

Com o objetivo de responder às questões de investigação, é necessário analisar o estado de arte atual. Apesar do estado de arte ser analisado nos capítulos 2 e 3 deste documento, é necessário compreender de que forma é feita a revisão da literatura.

Com o objetivo de recolher a informação necessária no que toca ao estado de arte atual, é necessário escolher quais fontes de informação servem como base para tal.

No âmbito desta dissertação, as principais fontes de informação serão artigos com revisão por pares e livros de referência, ambos escritos por especialistas no ramo. Também serão consultados *sites* confiáveis, como por exemplo páginas oficiais de uma determinada tecnologia.

Dito isto, é necessário elaborar critérios mais objetivos para escolher estas fontes de informação. Estes consistem essencialmente em (Pereira, 2022b):

- Avaliar citações através da sua quantidade. Este consiste num bom indicador, apesar de possuir algumas limitações: influenciada pela idade, quanto mais citações mais visibilidade, conduzindo a mais citações, nem todas as citações são boas citações, …
- Avaliar os locais em que artigos são publicados. Estes locais habitualmente indicam que o artigo passou por escrutínio adicional.

1.5.3 Estratégias de Investigação

Com o objetivo de solucionar o problema de investigação, é necessário recorrer a estratégias de investigação. Entre as diversas estratégias disponíveis, a preferida foi a desenho e criação.

Esta estratégia, no âmbito de entrega de *software*, consiste em (Pereira, 2022a):

1. Desenhar um mecanismo de entrega de *software* automático e fiável concordante com as melhores práticas;
2. Criar um artefacto de entrega de *software* tendo por base o desenho efetuado previamente;
3. Avaliar a solução, utilizando métricas recolhidas do funcionamento da solução desenvolvida e de uma entrevista semiestruturada;
4. Concluir de que forma o resultado da avaliação se alinha com os objetivos iniciais.

A razão pela adoção desta estratégia prende-se à necessidade de validar a implementação de um mecanismo de entrega contínua no quotidiano da organização. As restantes alternativas não permitiriam auferir o mesmo grau de validação, já que não seriam estratégias tão específicas para a organização.

1.5.4 Métodos de Geração de dados

Outra questão extremamente importante para todo o processo de investigação, consiste nos métodos de geração de dados. Estes permitem suportar ou refutar as teorias e hipóteses formuladas (Pereira, 2022a).

Os principais métodos adotados neste processo de investigação consistem em:

- Observação, através da recolha de métricas associadas à utilização da solução desenvolvida;
- Entrevistas semiestruturadas, onde são obtidos dados diretamente de um ou mais utilizadores.

Relativamente à observação, o objetivo consiste na recolha de diversas métricas associadas à utilização da solução, de modo a analisar a solução quantitativamente.

A estrutura das entrevistas será um fator crucial na geração de dados, deste modo esta será: curta, com um conjunto de questões pré-definidas e feita apenas a utilizadores que beneficiarão desta solução diretamente.

1.5.5 Análise de Dados

Como dados por si só não possuem nenhum valor significativo, é necessário que após a geração de dados sejam utilizadas técnicas que permitam extrair significado dos dados recolhidos.

Deste modo, no âmbito desta tese a abordagem utilizada será tanto qualitativa como quantitativa, este tema é analisado em maior detalhe no capítulo 7.

1.6 Estrutura do Documento

O documento é iniciado contextualizando o domínio do problema, definindo objetivos, resultados esperados e a metodologia de investigação utilizada.

No capítulo 2 é realizada a análise do estado de arte a nível conceptual, onde são estudados os principais conceitos, metodologias e estratégias na área de entrega de *software*.

No capítulo 3 é elaborada a análise do estado de arte a nível tecnológico, que estuda as principais tecnologias capazes de dar suporte ao analisado no capítulo 2.

No capítulo 4 é realizada a análise de valor, começando por identificar a oportunidade e acabando na proposta de valor, além de serem utilizadas diversas técnicas para auxiliar na construção da solução.

No capítulo 5 é feita a recolha dos requisitos que a solução deve obedecer e, com base nestes, é elaborado e descrito o seu *design*, justificando as decisões mais importantes tomadas.

No capítulo 6 é detalhada a implementação da solução, abordando as suas especificidades e de que forma esta cumpre com o *design* elaborado.

No capítulo 7 é elaborada a avaliação da solução, apresentando a metodologia utilizada assim como os resultados obtidos.

No capítulo 8 são apresentadas as conclusões obtidas, refletindo sobre todo o trabalho efetuado e terminando com a indicações de como pode ser orientado o trabalho futuro.

2 Estado de Arte a Nível Conceptual

Ao longo deste capítulo é realizado um estudo dos principais conceitos e abordagens para o problema e desenho da solução.

Este estudo consiste sobretudo no aprofundamento dos principais conceitos, na análise de boas e más práticas de entrega de *software*, de estratégias de testes e métodos de entrega de *software*. Relativamente aos métodos de entrega de *software*, é também estudado os principais problemas e soluções na sua adoção, assim como alguns casos de sucesso.

2.1 Principais Conceitos

Esta secção incide principalmente nos principais conceitos associados à entrega de software, analisando cada conceito individualmente e fazendo paralelos sobre como estes se correlacionam.

2.1.1 Engenharia de Software Contínua

Engenharia de software contínua consiste num conjunto de atividades continuas que visam entregar *software* rapidamente e com qualidade, de forma a obter *feedback* recorrente dos utilizadores (Shahin, Ali Babar and Zhu, 2017).

Este conjunto de atividades contínuas divide-se em 3 categorias distintas: estratégia de negócio e planeamento, desenvolvimento e operações (Shahin, Ali Babar and Zhu, 2017).

No âmbito desta dissertação, serão analisados com maior enfâse 3 conceitos da categoria de desenvolvimento, sendo estes: integração contínua (CI), entrega contínua (CD) e implantação contínua.

2.1.2 Integração Contínua

Integração contínua consiste numa prática de desenvolvimento de software em que o código desenvolvido por cada elemento de equipa é integrado regularmente num repositório comum (Fowler, 2006).

Cada integração é validada através de um processo automático onde é feita a compilação do *software* e um conjunto de testes com o objetivo de detetar erros o mais depressa possível (Fowler, 2006).

Caso uma determinada alteração ao *software* falhe quer no processo de compilação, quer nos testes, torna-se prioridade máxima corrigir este problema. Deste modo, é garantido que a aplicação se encontra num estado funcional a qualquer altura (Laukkanen, Itkonen and Lassenius, 2017).

Esta abordagem traz como principais benefícios (Humble and Farley, 2010):

- Redução de risco na entrega do *software*;
- Entregas de *software* mais rápidas;
- Maior facilidade em detetar e eliminar *bugs*, resultando num menor número destes.

2.1.3 Entrega Contínua

Devido à inerente complementaridade que existe entre entrega contínua e integração contínua, estas 2 metodologias frequentemente são utilizadas em simultâneo.

Entrega contínua (CD) é uma metodologia de desenvolvimento de *software* em que o *software* se mantém sempre num estado em que poderia ser entregue aos utilizadores a qualquer momento (Fowler, 2013).

Tal é atingido através da otimização, automatização e utilização de processos de compilação, implantação, teste e lançamento de *software*, como representado na figura 2 (Laukkanen, Itkonen and Lassenius, 2017).

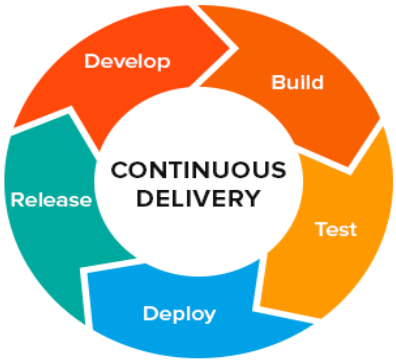

Figura 2 – Processo de entrega contínua (Zaiku, 2017)

Sempre que existe uma alteração no *software*, é desencadeado um conjunto de passos de modo a garantir que o *software* está apto para ser entregue, terminando na possibilidade de implantação. Esta última depende da decisão de um elemento da equipa ou de uma regra de negócio implementada (Shahin, Ali Babar and Zhu, 2017; Shahin *et al.*, 2017).

Assim sendo, é possível caracterizar entrega contínua como uma abordagem *pull-based*, já que existe a decisão de quando e do que fazer uma implantação (Shahin, Ali Babar and Zhu, 2017).

Quando adotada uma abordagem de entrega contínua, os ciclos de desenvolvimento tendem a ser mais curtos e as entregas mais frequentes (Shahin, Ali Babar and Zhu, 2017).

A prática de entrega contínua possuí como principais benefícios (Shahin, Ali Babar and Zhu, 2017; Fowler, 2013):

- Redução do risco associado à implantação;
- Maior perceção de todo o progresso realizado;
- Implantações mais rápidas, menos trabalhosas e mais frequentes;
- Redução de custos.

2.1.4 Implantação Contínua

Apesar de ser um conceito distinto, implantação contínua é confundida muitas vezes com entrega contínua.

Implantação contínua é uma extensão de entrega contínua em que, cada alteração ao *software*, é automaticamente implantada em produção (Shahin, Ali Babar and Zhu, 2017).

Esta abordagem pode também ser caracterizado como uma metodologia *push-based*, uma vez que a implantação é feita automaticamente, sem qualquer tomada de decisão (Shahin, Ali Babar and Zhu, 2017).

Este conceito difere de entrega contínua devido à implantação ocorrer automaticamente. No caso da entrega contínua, esta garante que o *software* pode ser implantado a qualquer momento, no entanto, pode ser tomada a decisão de não o fazer.

Deste modo, é possível afirmar que, para existir implantação contínua é necessário haver entrega contínua, mas o contrário não é verdade (Shahin, Ali Babar and Zhu, 2017).

2.2 Más Práticas Habituais de Entrega de Software

Sem a adoção da prática de engenharia de *software* continua, existe um risco considerável associado ao processo de implantação.

Assim, nesta secção são analisadas algumas más práticas na entrega de *software*, comuns em diversas organizações.

2.2.1 Implantação Manual de Software

Esta consiste numa prática bastante presente nas organizações atualmente. Normalmente, estas organizações possuem aplicações difíceis de implantar, envolvendo muitos passos. Neste processo de implantação, os seus passos tendem a ser separados e atómicos, sendo por vezes efetuados por múltiplos elementos de equipa e até equipas diferentes (Humble and Farley, 2010).

Esta quantidade e diversidade de variáveis no processo de implantação acarretam diferentes riscos. Desde o risco associado ao erro humano na tomada de decisão, como até às possíveis combinações de passos efetuados em alturas diferentes, atingindo resultados possivelmente distintos (Humble and Farley, 2010).

Assim, este tipo de prática aumenta consideravelmente o risco de entregas de *software* falharem, podendo resultar em diversos problemas não previstos (Humble and Farley, 2010).

2.2.2 Implantação Tardia em Staging

Como o nome sugere, esta má prática consiste em efetuar a implantação do *software* num servidor de *staging*, semelhante a produção, apenas após o desenvolvimento ser dado como concluído (Humble and Farley, 2010).

Habitualmente, quando a implantação é feita em *staging*, ninguém além dos desenvolvedores testou a aplicação, sendo que os testes efetuados por estes foram em ambiente local (Humble and Farley, 2010).

Uma vez que a primeira implantação tende a ser mais problemática, quando esta é feita em *staging* é normal serem encontrados novos *bugs*. Como esta implantação foi feita demasiado

tarde, é possível que diversos *bugs* não sejam corrigidos a tempo de serem incluídos na versão (Humble and Farley, 2010).

2.2.3 Configuração Manual de Ambientes de Produção

Quando existe a necessidade de efetuar alguma alteração de configuração ao ambiente de produção, normalmente é feita a alteração manual do parâmetro pretendido (Humble and Farley, 2010).

A título de exemplo, caso seja necessário atualizar o número de conexões da *connection pool* de uma base de dados, este parâmetro é alterado manualmente nos servidores pretendidos.

Alguns indícios apresentados pela adoção desta prática consistem em (Humble and Farley, 2010):

- Após múltiplas implantações em staging, a implantação em produção falha.
- Diferentes servidores de um mesmo *cluster* apresentam comportamentos diferentes.
- Não é possível voltar a uma configuração mais antiga do sistema.
- Alguns servidores dentro de um *cluster*, involuntariamente, apresentam diferentes versões de sistema e infraestrutura.

2.3 Boas Práticas de Entrega de Software

Após a identificação e análise de algumas das más práticas adotadas por grande parte das organizações, é importante compreender o que constitui uma boa entrega de *software*.

Para tal, o objetivo de uma entrega de *software* deve ser claro: entregar *software* de forma eficiente, eficaz e fiável. Deste modo, esta secção apresenta alguns princípios fundamentais para um bom mecanismo de entrega de *software*.

2.3.1 Processo Repetível e Fiável

A repetibilidade e fiabilidade num processo de entrega de *software* deriva essencialmente de dois pontos: da automatização e da existência do controlo de versões para tudo que é necessário utilizar nas fases de compilação, implantação, teste e entrega (Humble and Farley, 2010).

A repetibilidade do processo está intimamente ligada ao seu grau de automatização. Caso o processo não seja automático, a sua repetibilidade é notavelmente condicionada.

No que toca à fiabilidade no processo de entrega de *software*, esta deve ser atingida pela quantidade e qualidade dos testes automáticos efetuados. Naturalmente, quanto maior for a sua utilização, maior será a confiança neste, assumindo que a entrega é bem-sucedida.

2.3.2 Automatizar o Máximo Possível

Idealmente, o processo de implantação de uma solução consiste no simples ato de pressionar um botão. Tal apenas é possível através da automatização do processo de entrega de *software*.

Obviamente, nem tudo pode ser automatizado, como por exemplo testes manuais. No entanto, é possível afirmar que a maioria dos passos num guião de implantação podem e devem ser realizados sem intervenção humana (Humble and Farley, 2010).

Questões como a implantação de *software*, a sua configuração e a maioria dos testes podem ser automatizadas através de processos que recorrem ao controlo de versões, *scripts* e diversas outras ferramentas (Humble and Farley, 2010).

2.3.3 Manter Tudo no Controlo de Versões

Não só o código da aplicação deve estar no controlo de versões. Tudo que é necessário para compilar, testar, configurar e implantar devem estar guardados com um algum tipo de versionamento (Fowler, 2006; Humble and Farley, 2010).

Não só tudo deve estar no controlo de versões, como cada conjunto de alterações deve estar devidamente identificado com a respetiva versão (Humble and Farley, 2010).

2.3.4 Se Custa, Deve Ser Mais Frequente e Mais Cedo

Este princípio pode ser descrito como uma heurística que se resume na ideia de: se entregar *software* é um processo doloroso, deve ser feito o mais cedo possível e com frequência. Adiar a entrega apenas torna o processo mais doloroso do que deveria (Humble and Farley, 2010).

Idealmente, o objetivo deve ser entregar *software* sempre que existe uma alteração que passa em todos os testes automáticos (Humble and Farley, 2010).

Caso tal não seja possível, devem ser estabelecidos objetivos intermédios, como por exemplo uma entrega interna a cada 2 ou 3 semanas. Gradualmente, estes objetivos devem ser atualizados de forma a atingir o resultado pretendido (Humble and Farley, 2010).

2.3.5 Build Quality In

Este princípio é também adotado pelo movimento *lean*, consistindo essencialmente em encontrar erros o mais cedo possível, já que, quanto mais cedo for detetado o erro, mais fácil é de o resolver. Num cenário perfeito, estes erros são detetados antes da alteração chegar ao mecanismo de controlo de versões (Humble and Farley, 2010).

Este princípio possui dois corolários importantíssimos (Humble and Farley, 2010):

- Testar não é apenas uma fase, e certamente não deve ocorrer apenas após a fase de desenvolvimento.
- Toda a gente envolvida no processo de entrega deve testar, não apenas os *testers*.

2.3.6 Feito Significa Entregue

Uma funcionalidade estar terminada pode ter significados diferentes para equipas diferentes. Uma boa prática consiste em considerar uma determinada funcionalidade como feita apenas quando esta estiver a entregar valor aos utilizadores (Humble and Farley, 2010).

Deste modo, uma funcionalidade apenas deve ser dada como feita, quando esta tiver sido corretamente demonstrada e testada por utilizadores, num ambiente semelhante a produção ou até mesmo em produção (Humble and Farley, 2010).

2.3.7 Todos São Responsáveis Pela Entrega

Em diversas organizações, a responsabilidade da entrega fica muitas vezes atribuída à equipa que faz a implantação da solução.

Frequentemente, os desenvolvedores entregam o seu trabalho aos *testers*. Estes, entregam o seu trabalho à equipa responsável pela entrega. Quando a entrega falha, o processo utilizado para resolver o problema é muitas vezes ineficiente (Humble and Farley, 2010).

Alternativamente, a organização como um todo deveria ter um papel no processo de entrega de *software*, uma vez que esta abordagem tende a resultar num melhor processo de implantação (Laukkanen, Itkonen and Lassenius, 2017).

2.3.8 Planeamento e Documentação

A adoção de práticas contínuas deve ser um processo gradual. Deste modo, um bom planeamento com objetivos e passos bem definidos assim como uma documentação ampla consistem em duas práticas essenciais (Shahin, Ali Babar and Zhu, 2017).

A documentação e o desenho no início do planeamento de uma entrega, tem demonstrado uma maior facilidade para equipas manterem a entrega contínua (Shahin, Ali Babar and Zhu, 2017).

Por sua vez, o planeamento devidamente estruturado, constituído por um conjunto de regras definido pela organização, permitiu a esta adotar um conjunto de práticas continuas com confiança, já que todas as atividades estavam devidamente planeadas (Shahin, Ali Babar and Zhu, 2017).

2.3.9 Melhoria Contínua

Este princípio é possivelmente um dos mais importantes abordados nesta secção. A sua importância é tão grande a ponto de ser uma atividade na área de engenharia de *software* contínua (Fitzgerald and Stol, 2017).

Esta atividade é baseada nos princípios *lean* de tomada de decisão com base em informação e redução de desperdício, resultando em pequenas mudanças incrementais que podem ter grandes benefícios ao longo do tempo (Fitzgerald and Stol, 2017).

A primeira entrega de uma solução consiste apenas no primeiro estágio da sua vida. Todas as aplicações sofrem alterações e por consequência, diversas entregas. É importante que o processo de entrega acompanhe a evolução da aplicação (Humble and Farley, 2010).

Essencialmente, isto significa que, toda a equipa, ou mesmo várias equipas, devem se reunir com alguma regularidade e analisar o processo de entrega de *software*. Deve ser feita uma retrospetiva refletindo sobre o que correu bem, o que correu mal e o que deve ser melhorado.

2.4 Testes

Com o objetivo de garantir a qualidade do *software* entregue, esta secção aborda as principais estratégias de teste, também explorando testes em soluções baseadas em análise de dados.

Idealmente, num típico projeto de *software*, metade do tempo e dos custos devem ser alocados a testes. No entanto, isto raramente se verifica (Meyers, Sandler and Badgett, 2012).

Num cenário ideal, desde o início do projeto, *testers* trabalham com desenvolvedores e utilizadores de forma a elaborarem os testes. Estes testes devem ser escritos antes do desenvolvimento começar e englobar requisitos funcionais e não funcionais (Humble and Farley, 2010)

Com as práticas de integração contínua, sempre que uma alteração é feita ao *software*, este conjunto de testes deve ser executado. Quando estes testes passam, significa que o *software* cumpre com o comportamento esperado (Humble and Farley, 2010).

De modo a validar corretamente o comportamento do *software*, devem ser utilizados diversos tipos distintos de testes de forma a auferir um maior grau de confiança à solução. É possível resumir os diferentes tipos de teste na figura 3 (Humble and Farley, 2010).

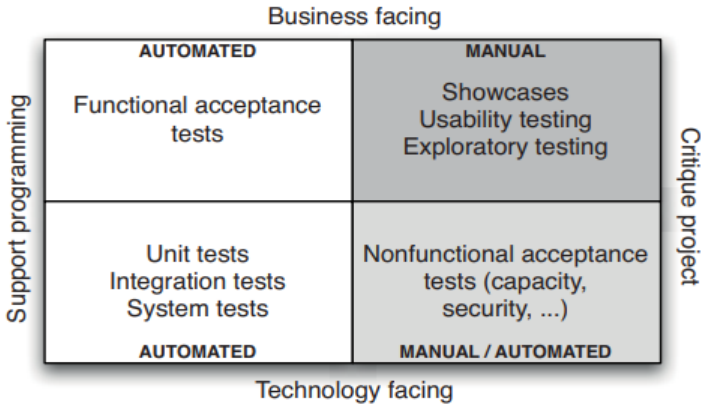

Figura 3 – Diagrama de tipos de teste (Humble and Farley, 2010)

2.4.1 Testes Funcionais de aceitação

Como representado na figura 3, estes testes são direcionados ao negócio e suportam o processo de desenvolvimento. A sua função consiste em garantir que os critérios de aceitação de uma funcionalidade são cumpridos.

Estes testes apresentam um papel crítico quando se trata de garantir a qualidade de *software*, já que quando estes passam, significa que por um lado, os desenvolvedores concluíram a funcionalidade, e por outro lado, os utilizadores tiveram as suas necessidades satisfeitas (Humble and Farley, 2010).

Todos os testes funcionais de aceitação devem ser escritos através da ótica do utilizador, focando-se no comportamento esperado do sistema e não propriamente como é que este o faz (Farley, 2021). Assim, é possível afirmar que, num cenário ideal, este tipo de testes até seria feito pelo próprio utilizador (Humble and Farley, 2010).

De forma a simular corretamente a verdadeira utilização do *software*, os testes funcionais de aceitação devem ser realizados num ambiente semelhante a produção, onde o *tester* irá interagir com a aplicação (Humble and Farley, 2010).

Idealmente, estes testes devem ser automatizados, tendo diversas vantagens como (Humble and Farley, 2010):

- *Feedback* mais rápido, já que os desenvolvedores podem executar um determinado conjunto de testes de forma a perceber se cumpriram com um requisito.
- Redução do trabalho dos *testers*, permitindo a alocação destes em outras atividades.
- Permitem garantir que alterações ao *software* não estragam funcionalidades já implementadas.
- Consoante a implementação dos testes, existem ferramentas que, geram documentação de requisitos automática e mantêm esta atualizada.

Todavia, a automatização dos testes de aceitação funcionais traz uma grande desvantagem: custos de implementação e manutenção (Humble and Farley, 2010).

Sendo assim, é importante perceber quando é que um teste de aceitação funcional deve ser automático, tendo em conta fatores como a importância da funcionalidade a ser testada, o quão difícil é testar a funcionalidade, entre outros (Humble and Farley, 2010).

2.4.2 Testes Unitários

Como representado na figura 3, estes testes são direcionados à tecnologia e suportam o desenvolvimento. O principal objetivo deste tipo de testes consiste em, por um lado, garantir que o programador consegue perceber o caso de uso em questão, por outro lado, contribuir para o desenho do sistema (Crispin and Gregory, 2009).

Essencialmente, estes testes, como o nome indica, testam uma unidade. Esta unidade consiste no excerto de código mais pequeno que pode ser logicamente isolado, e testado, do sistema (Smartbear, 2023).

Este tipo de testes apenas testa uma funcionalidade num dado momento, permitindo saber exatamente o que está a ser testado e onde é que os problemas se encontram (Mackinnon, Freeman and Craig, 2001).

Assim, estes testes permitem contribuir ativamente para um maior grau de confiança no sistema, além de facilitar alterações a código existente, uma vez que, caso os testes passem, a alteração foi bem-sucedida.

2.4.3 Testes de Integração

Tal como os testes unitários, este tipo de testes também pertence ao quadrante direcionado à tecnologia e que suporta o desenvolvimento. Estes testes determinam se unidades independentes de *software* funcionam corretamente quando conectadas.

Enquanto testes unitários tipicamente atuam ao nível da unidade, estes testes são feitos um nível acima, permitindo validar a interação entre as diferentes unidades (Leung and White, 1990).

Estes testes mostram-se especialmente úteis quando a aplicação possui objetos com diferentes ciclos de vida. Já que, apenas quando são testadas partes maiores da aplicação em simultâneo, é possível detetar erros na gestão dos ciclos de vida (Crispin and Gregory, 2009).

Tipicamente, estes testes tendem a ser mais lentos que testes unitários, uma vez que existe um maior tempo de *setup* envolvido, são efetuadas operações I/O, chamadas a bases de dados ou até a outros sistemas (Humble and Farley, 2010).

2.4.4 Testes de implantação

Tal como o nome indica, estes testes acontecem sempre que a aplicação é implantada. Assim como os testes unitários e de integração, estes também estão direcionados à tecnologia e suportam a programação.

A principal responsabilidade deste tipo de testes consiste em verificar que a implantação foi feita corretamente. Por outras palavras, deve ser testado que as partes mais importantes do sistema efetivamente funcionam (Rossel, 2017).

A complexidade deste tipo de testes pode assumir diversas formas, pode ser algo tão simples como verificar que quando a aplicação é implantada, o ecrã principal aparece com o conteúdo esperado, como algo mais complexo, em que são realizados testes funcionais de aceitação ao sistema, como por exemplo validar as credenciais de um *login* (Humble and Farley, 2010).

2.4.5 Showcases

Showcases pertencem ao quadrante de testes direcionados ao negócio que criticam o projeto. Os testes deste quadrante têm a função de verificar que a aplicação entrega o valor que os utilizadores esperam. Isto significa que, não só é validado se o *software* vai de encontro à especificação, como também se a especificação vai de encontro às necessidades do utilizador (Humble and Farley, 2010).

Dito isto, *showcases* consistem na demonstração de funcionalidades ao utilizador, com o intuito de encontrar falhas na especificação ou possíveis mal-entendidos (Humble and Farley, 2010).

Deste modo, este tipo deve ser efetuado cedo e frequentemente, de forma a encontrar as falhas referidas o mais depressa possível (Meyers, Sandler and Badgett, 2012).

2.4.6 Testes Exploratórios

Tal como os *showcases*, estes testes fazem parte dos testes direcionados ao negócio que criticam o projeto. Esta tipo de testes pode ser definida como “simultaneamente aprender, desenhar testes e executá-los”(Bach, 2003).

Por outras palavras, testes exploratórios consistem em testes cujo *tester* tem controlo sobre o desenho dos testes enquanto estes estão a ser executados além de, poder utilizar informação recolhida para, ativamente, desenhar novos e melhores testes (Bach, 2003).

Estes testes distinguem-se dos demais essencialmente pelo grau de planeamento. Enquanto a maioria dos testes segue um guião já planeado, no caso dos testes exploratórios, ideias vão surgindo à medida que os testes são executados, sendo estas ideias exploradas (Bach, 2003).

2.4.7 Testes de Usabilidade

Também no quadrante dos testes direcionados ao negócio que criticam o projeto, estes testes são cruciais para descobrir o quão fácil é para os utilizadores atingirem os seus objetivos dentro da aplicação.

Devido à natural tendência de as pessoas envolvidas no projeto ficarem demasiado confortáveis com o problema e a solução desenvolvida, nem sempre é fácil perceber como o *software* desenvolvido se irá comportar face ao utilizador (Crispin and Gregory, 2009).

Assim, estes testes permitem perceber se a aplicação traz o valor esperado para os utilizadores esperados.

Existem diversas formas de realizar estes tipos de testes, como por exemplo (Crispin and Gregory, 2009):

- Utilizar *personas*. Isto é, inventar diferentes tipos de utilizadores, com níveis de experiência e necessidades diferentes, com o intuito de perceber se a aplicação é capaz de responder com sucesso às necessidades de cada uma das *personas* criadas.
- Extrair diferentes métricas como: quanto tempo os utilizadores precisam para cumprir as suas tarefas, quanto tempo os utilizadores gastam para descobrir os botões e campos adequados, entre outros.

2.4.8 Testes de Aceitação Não Funcionais

Testes de aceitação não funcionais pertencem ao quadrante de testes direcionados à tecnologia que criticam o projeto.

Estes testes, tal como os testes de aceitação funcionais, possuem a função de validar se os critérios de aceitação são cumpridos. No entanto, no caso dos testes de aceitação não funcionais, são validados os critérios de aceitação de atributos de qualidade do sistema, em vez de funcionalidades (Humble and Farley, 2010).

Assim, estes testes irão se focar em atributos de qualidade como: capacidade, desempenho, disponibilidade, segurança, entre outros (Humble and Farley, 2010).

Apesar de muitas vezes negligenciada, a definição de critérios de aceitação de requisitos não funcionais é crucial para o sucesso de um projeto de *software*, sendo necessário especificar da mesma forma que os critérios de aceitação de requisitos funcionais (Humble and Farley, 2010).

É importante também referir que, para validar os critérios de aceitação deste tipo de testes, as estratégias e ferramentas são frequentemente distintas das utilizadas para validar critérios de aceitação funcionais. Obviamente, estas estratégias e ferramentas vão depender de diversos fatores, como o atributo de qualidade a ser testado por exemplo.

2.4.9 Testes em Aplicações Baseadas na Geração de Nova Informação

Embora esta secção não corresponda a um tipo de testes propriamente dito, esta é importante para o âmbito do problema desta dissertação.

Como havia sido referido na secção 1.1 deste documento, existe uma limitação nas ferramentas atuais quando se trata de validar a qualidade de um *software* baseado na qualidade dos dados que este produz.

Essencialmente, todas as estratégias de testes abordadas até então mostram-se limitadas quando é necessário testar *software* através da nova informação gerada por este, como por exemplo aplicações que lidam com um elevado volume de dados, como soluções de análises de dados (Staegemann *et al.*, 2019).

Estas aplicações que lidam com um elevado volume de dados, frequentemente geram conhecimento a partir de informação já existente (Staegemann *et al.*, 2019). Esta geração de conhecimento a partir de informação já existente acentua uma grande diferença entre este tipo de *software* e *software* tradicional.

Enquanto em *software* tradicional os valores de entrada e de saída são conhecidos, isto não acontece para *software* que adota o comportamento referido anteriormente, em que estes

sistemas se assemelham a “caixas pretas”, com valores de entrada e saída desconhecidos. Este problema é habitualmente chamado de problema “*oracle*” (Staegemann *et al.*, 2019).

Existem diversas abordagens para atacar este problema, nesta secção serão abordadas duas destas bastante utilizadas:

- *Metamorphic testing* (MT);
- Inteligência artificial;

2.4.9.1 Metamorphic Testing

Este método consiste em lidar com situações em que o *oracle* não acontece, através da geração de testes, com o objetivo de conseguir associar os dados de entrada com os dados de saída através de relações especificas do domínio chamadas de *metamorphic relations* (*MR*) (Yang, Troup and Ho, 2017).

Após a obtenção de diversas *MRs*, é possível serem efetuados novos casos de teste tendo por base os primeiros testes efetuados e as *MR*s (Yang, Troup and Ho, 2017).

Após a obtenção dos novos casos de teste, tanto os novos testes como os primeiros são executados e validados de acordo com as *MRs.* Se algum par de testes antigos e novos testes não satisfizer uma determinada *MR*, é possível afirmar que o teste foi falhado e que existem erros (Yang, Troup and Ho, 2017).

Um fator importante de ser referido consiste na capacidade do *metamorphic testing* utilizar dados de produção de forma a gerar os primeiros casos de teste, permitindo testar o *software* tendo por base os dados que ele utilizará quando for entregue (Yang, Troup and Ho, 2017).

2.4.9.2 Inteligência artificial

Devido ao vasto leque de técnicas envolvidas no conceito de inteligência artificial, existem diversas formas técnicas distintas capazes de ultrapassar o problema *oracle* (Barr *et al.*, 2015). Estas técnicas, quando comparadas às demais, fazem parte do grupo das abordagens mais recentes a tentar lidar com este problema (Barr et al., 2015).

Fundamentalmente, a ideia consiste em utilizar as capacidades preditivas de inteligência artificial, independentemente da técnica aplicada, com o objetivo de criar testes automaticamente, capazes de ultrapassar este problema (Gao *et al.*, 2019).

2.5 Pipeline de Implantação

Após a análise dos principais conceitos da área, princípios a serem adotados e estratégias de testes, resta perceber como deve ser construído um mecanismo eficiente de entrega contínua, que resulta na automatização de alguns dos conceitos apresentados anteriormente.

A principal abordagem para este fim é a *pipeline* de implantação que, essencialmente, consiste no processo automático de obter *software* do controlo de versões e colocá-lo na mão dos utilizadores (Humble and Farley, 2010).

O típico funcionamento de uma *pipeline* de implantação encontra-se ilustrado na figura 4.

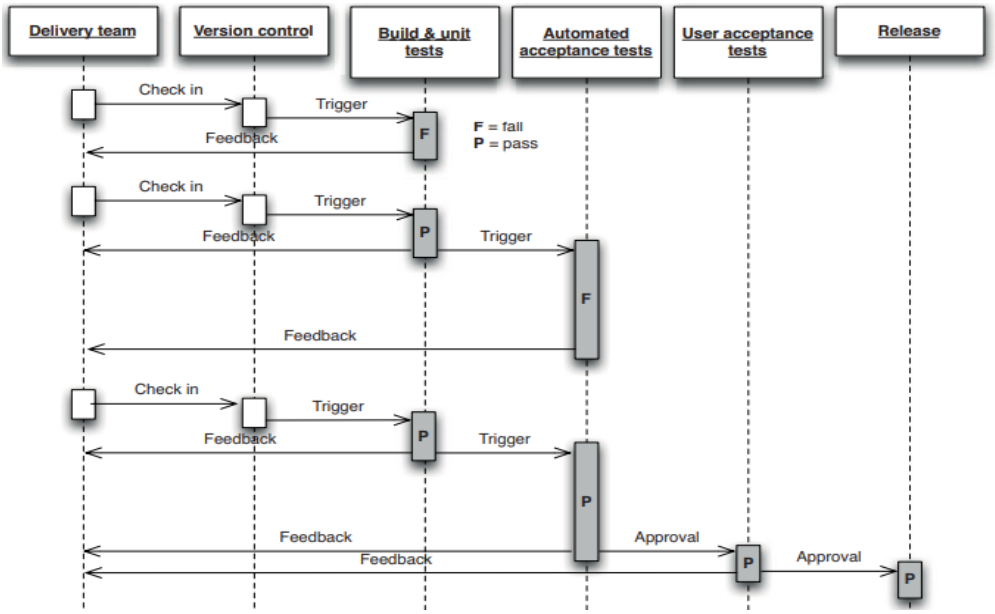

Figura 4 – Funcionamento de uma *pipeline* de implantação (Humble and Farley, 2010)

Analisando o diagrama de sequência da *pipeline* de implantação, é possível compreender o seguinte fluxo (Humble and Farley, 2010):

1. Qualquer alteração feita à aplicação, ou à sua configuração, ou ao seu ambiente desencadeia o funcionamento da *pipeline*.
2. É feita a compilação do *software* (*build*).
3. São executados uma série de testes, com o intuito de garantir que a aplicação se encontra pronta para ser lançada. À medida que a aplicação passa nos testes, cada vez mais estes testes se assemelham a produção.
4. Caso os testes passem, a aplicação pode ser lançada.

A aplicação desta abordagem oferece vantagens como (Humble and Farley, 2010):

- Apenas é lançado *software* que foi testado devidamente e que se encontra adequado ao seu propósito;
- *Bugs* de regressão são evitados, especialmente em correções urgentes, já que estas correções têm de passar pelo mesmo processo.
- Mitiga riscos de implantações que falham devido a problemas de configuração, comunicação, entre outros.

Para a implementação desta abordagem é necessário (Humble and Farley, 2010):

- Adotar uma estratégia de testes adequada que prove que o *software* cumpre o seu objetivo.
- Automatizar a implantação para ambientes de teste, *staging* e produção, já que, quando este processo é manual, tende a ser longo e propenso a erros.

Com estes objetivos em mente, diferentes sistemas apresentam necessidades diferentes, resultando em *pipelines* de implantação distintas, no entanto, o seguinte conjunto de passos (*stages)*, é comum à maioria dos projetos (Humble and Farley, 2010; Farley, 2021)

- *Stage* de commit, responsável por assegurar que o sistema funciona a nível técnico, onde ocorre a fase de compilação, execução de testes unitários e análise de código.
- *Stage* de testes de aceitação automáticos, cuja função consiste em validar que o sistema funciona tanto a nível funcional como não funcional.
- *Stage* de testes manuais, com o objetivo de garantir que o sistema se encontra usável, cumpre os requisitos e disponibiliza valor aos utilizadores. Neste *stage* habitualmente são efetuados testes de aceitação e exploratórios.
- *Stage* de entrega, responsável por entregar o sistema aos utilizadores, quer através da entrega do *software* em si, como através da implantação num ambiente de produção (ou *staging*).

Obviamente, existem outros *stages* que podem e devem ser adicionados a uma *pipeline* de implantação, como por exemplo *stage* de testes de implantação.

É importante mencionar que este processo de entrega de *software*, apesar de possuir um maior grau de automatização, não elimina a interação humana. Este processo assegura que passos complexos, propensos a erros e muitas vezes longos são automatizados.

2.5.1 Boas Práticas de Pipelines de Implantação

Nesta secção, são analisadas as principais práticas a adotar que contribuem positivamente para um aumento da eficácia da *pipeline* de implantação.

2.5.1.1 Compilar Apenas Uma Vez

Não é incomum que muitos sistemas compilem o código diversas vezes em múltiplos contextos distintos: compilar antes dos testes unitários, compilar novamente antes dos testes de aceitação e assim por diante (Humble and Farley, 2010).

Esta prática constitui um erro possivelmente grave, já que, sempre que o código é compilado, existe o risco de ser introduzida uma diferença. Esta diferença pode vir de uma alteração no compilador ou até de uma alteração de um *software* de terceiros (Humble and Farley, 2010).

Outro problema com múltiplas compilações é a falta de eficiência, já que o processo de compilação consome tempo e tende a ser mais demorado à medida que o *software* evoluí (Humble and Farley, 2010).

2.5.1.2 Implantar Da Mesma Forma em Diferentes Ambientes

A confiança na *pipeline* de implantação aumenta cada vez que uma entrega de *software* é bem-sucedida. Como o ambiente de produção normalmente recebe menos implantações, apenas podemos ter confiança que a *pipeline* funcionará corretamente neste ambiente, se o processo de implantação for comum a todos os ambientes (Humble and Farley, 2010).

Obviamente, existem sempre diferenças entre ambientes, no entanto, a solução não deve ser alterar o método de implantação e sim separar as diferentes configurações de cada ambiente (Humble and Farley, 2010).

Uma forma de separar as configurações por ambiente consiste em utilizar um ficheiro de propriedades diferentes por cada ambiente, garantindo que estes diferentes ficheiros são colocados no mecanismo de controlo de versões (Humble and Farley, 2010).

2.5.1.3 Efetuar testes de implantação

Quando é feita a implantação da aplicação, devem existir um conjunto de testes de implantação responsáveis por assegurar que a aplicação está a trabalhar como suposto.

O comportamento deste tipo de testes já foi abordado na secção 2.4.4 deste documento, no entanto, é importante reforçar a importância destes testes principalmente no âmbito de uma

pipeline de implantação, uma vez que estes auferem um maior grau de confiança de que a aplicação foi implantada corretamente.

2.5.1.4 Implantar Numa Cópia de Produção

Um problema frequentemente observado em diversas organizações consiste no ambiente de produção ser consideravelmente distinto do ambiente em que o *software* é desenvolvido e testado (Humble and Farley, 2010).

De forma a aumentar a confiança de que a implantação da aplicação corresponderá às expectativas, é boa prática realizar a implantação previamente num ambiente de *staging* o mais parecido possível com produção (Humble and Farley, 2010).

2.5.1.5 Cada Ação Deve Ser Propagada pela Pipeline Instantaneamente

Não é incomum em diversos projetos serem executadas partes do processo através de um agendamento, por exemplo: testes de aceitação funcionais são feitos uma vez por dia e testes de aceitação não funcionais são feitos uma vez por semana (Humble and Farley, 2010).

No entanto, a *pipeline* de implantação adota uma filosofia distinta: o primeiro *stage* deve ser acionado após cada alteração ao *software*, e cada *stage* após esse apenas deve ser executado quando o anterior terminar e for bem-sucedido (Humble and Farley, 2010). A única exceção a esta regra consiste na execução de *stages* de forma paralela.

No entanto, caso aconteçam múltiplos *check in* no controlo de versões com elevada frequência, é possível existirem alguns conflitos, como o demostrado na figura 5.

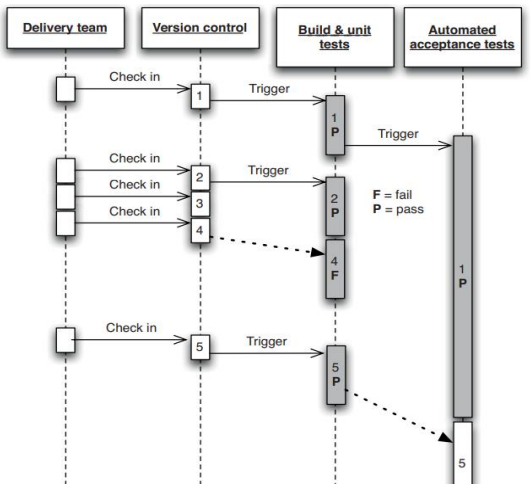

Figura 5 – Diagrama de conflitos de *stages* (Humble and Farley, 2010)

Analisando o conflito presente no diagrama de sequência apresentado, é possível compreender que *check ins* de alterações demasiado frequentes podem causar a execução de um *stage* que já se encontra a ser executado.

Uma possível solução para este problema consiste em garantir que um *stage* nunca é executado por mais do que uma instância em simultâneo. Por outras palavras, quando um *stage* termina a sua execução, o sistema de CI verifica se existem alterações, caso existam, é executado novamente com a última versão (Humble and Farley, 2010).

2.5.1.6 Se Uma Parte da Pipeline Falha, Aborta a Execução

O princípio fundamental de uma *pipeline* de implantação consiste na equipa aceitar que, sempre que é feito *check in* de uma alteração ao *software* no controlo de versões, todos os *stages* serão bem-sucedidos (Humble and Farley, 2010).

Caso ocorra algum erro, é prioridade máxima garantir que este se encontra corrigido e só depois pode ser retomada a normalidade (Laukkanen, Itkonen and Lassenius, 2017).

2.6 Scripting de Compilação e Implantação

Os *stages* de compilação e implantação são completamente imprescindíveis para uma *pipeline* de implantação de qualidade.

À medida que é desenvolvido o *software*, a sua complexidade tende a crescer, assim como a complexidade destes *stages*. Numa aplicação simples, o processo de compilação pode ser algo tão simples como executar um comando. No entanto, a crescente complexidade do *software* conduz a um aumento de componentes ou até mesmo a alguma necessidade distinta na gestão destes, gerando a necessidade de recorrer a scripts (Humble and Farley, 2010).

No que toca ao processo de implantação, este dificilmente consiste apenas em colocar o resultado da compilação do *software* no ambiente pretendido (Humble and Farley, 2010).

Por norma, o processo de implantação é constituído por múltiplos passos como: a configuração da aplicação, configuração da infraestrutura, configuração do sistema operativo, inicialização de dados, entre muitos outros (Humble and Farley, 2010).

Em suma, é necessário que emerja um processo capaz de ultrapassar as dificuldades de compilação e implantação apresentadas, sendo este eficiente e manutenível. A abordagem de utilização de *scripting* nestes *stages* é um método adequado para a resolução deste problema.

2.6.1 Princípios e Práticas de *Scripting* de Compilação e Implantação

Nesta secção, são abordados os principais princípios e práticas de *scripting* de compilação e implantação independentes das tecnologias utilizadas.

2.6.1.1 Criar um Script Para Cada *Stage* da Pipeline de Implantação

Da mesma forma que uma aplicação deve espelhar o domínio em que se insere, os *scripts* devem também refletir os processos que implementam (Humble and Farley, 2010). Deste modo, os *scripts* devem ter uma estrutura bem definida, de fácil manutenção e minimizando dependências entre componentes (Shahin, Ali Babar and Zhu, 2017).

Apesar de inicialmente fazer sentido utilizar um único *script* que contenha todas as operações que serão executadas ao longo da *pipeline* de implantação, posteriormente este deverá ser dividido em múltiplos *scripts*, um por cada *stage* da *pipeline* (Humble and Farley, 2010).

2.6.1.2 Utilizar Tecnologia Apropriada Para a Implantação da Aplicação

Habitualmente, numa *pipeline* de implantação, a maioria dos testes que sucede o *stage* de *commit*, como o *stage* de testes de aceitação automáticos dependem de uma implantação bem-sucedida da aplicação (Humble and Farley, 2010).

Atendendo a óbvia necessidade de automatização da implantação, é importante que seja utilizada a tecnologia adequada este fim. Uma boa escolha significa não só uma maior facilidade em desenvolver o processo de implantação, como também mantê-lo e alterá-lo (Humble and Farley, 2010).

2.6.1.3 Utilizar os Mesmos Scripts de Implantação em Todos os Ambientes

Como já havia sido referido previamente na secção 2.5.1.2 “Implantar Da Mesma Forma em Diferentes Ambientes”, a implantação deve ser a mesma entre ambientes. Como corolário deste princípio, os *scripts* de implantação devem ser os mesmos para todos os ambientes.

Isto traduz-se na utilização dos mesmos *scripts* para cada implantação, onde as diferenças entre estes ambientes, como *IPs* e *URIs* de serviços, são tratadas como configuração gerida independentemente (Humble and Farley, 2010).

2.6.1.4 Garantir que o Processo de Implantação é Idempotente

O processo de implantação deve garantir que o estado deixado no ambiente desejado é sempre o mesmo, independentemente do estado deste ambiente antes da implantação.

A forma mais fácil de atingir este propósito consiste em utilizar ambientes cujo estado base seja o pretendido. Para atingir este fim pode ser utilizada uma abordagem automática, virtualização ou criação de contentores aplicacionais. Este ambiente deve conter tudo o necessário para a aplicação funcionar (Humble and Farley, 2010).

Caso a primeira abordagem não seja possível, uma abordagem alternativa consiste em validar que os pressupostos assumidos quanto ao ambiente onde será implantada a aplicação estão corretos. Caso esta validação falhe, a implantação deve falhar (Humble and Farley, 2010).

Outra possível abordagem consiste na utilização de ferramentas de implantação que são idempotentes (Humble and Farley, 2010).

2.6.1.5 Evoluir o Sistema de Implantação Incrementalmente

O destino desejado consiste num processo de implantação totalmente automático, onde a implantação do *software* se resuma a um simples clique de um botão.

No entanto, qualquer passo tomado nesta direção trará ganhos para a equipa que adota esta abordagem.

Como já havia sido referido na secção 2.3.9, o processo de entrega de *software* deve ser continuamente melhorado, por consequência isto inclui o processo de implantação.

Um possível plano de evolução incremental do sistema de implantação consiste em (Humble and Farley, 2010):

1. Começar por automatizar o processo de implantação para um ambiente de testes;
2. Garantir que o processo de implantação funciona em ambiente de desenvolvimento;
3. Refinar estes *scripts* com o intuito de implantar a aplicação num ambiente de testes de aceitação, de modo que estes testes possam ser executados.
4. Evoluir de modo que outras equipas possam utilizar as mesmas ferramentas para efetuar a implantação em *staging* e produção.

2.6.2 Scripting de Implantação

Um dos principais princípios na gestão de ambientes consiste no facto de alterações a estes apenas serem feitas através de processos automáticos, implicando o uso de *scripts*.

Existem diversas formas distintas de implantar *software* em máquinas remotas como (Humble and Farley, 2010):

1. Escrever um *script* que entra em cada máquina e executa os comandos necessários;
2. Escrever um *script* que corre localmente e possuí agentes responsáveis por correr o *script* em cada uma das máquinas remotas;
3. Empacotar a aplicação utilizando as ferramentas apropriadas e utilizar uma ferramenta de gestão de infraestrutura ou de implantação para lançar novas versões.

A terceira opção apresentada é a preferível, no entanto, caso a terceira opção não seja viável, servidores de integração contínua que possuam um modelo de agentes tornam a adoção da 2ª opção uma escolha viável (Humble and Farley, 2010).

Posto isto, é importante abordar algumas práticas fundamentais a qualquer tipo de estratégia de implantação escolhida.

2.6.2.1 Implantar e Testar Camadas

Um dos principais fundamentos de uma boa abordagem de entrega de *software* consiste na construção em cima de fundações sólidas. Isto significa que, não devem ser testadas alterações que nem sequer compilam, que não devem ser feitos testes de aceitação em alterações que falharam nos testes unitários e assim por diante.

Seguindo esta lógica, antes de ser instalada a aplicação, é necessário saber que o ambiente está preparado para tal. Deste modo, é possível pensar na implantação como uma série de camadas, tal como apresentado na figura 6 (Humble and Farley, 2010).

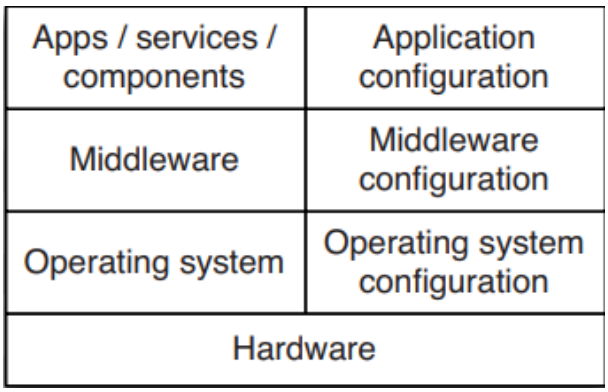

Figura 6 – Implantação como camadas (Humble and Farley, 2010)

De modo a ser possível alcançar uma implantação bem-sucedida, é necessário que, em primeiro lugar, todas as camadas estejam a funcionar como esperado, para só depois ser de facto efetuada a implantação da solução.

2.6.2.2 Testar a Configuração do Ambiente

Como uma implantação bem-sucedida necessita que a implantação de cada camada seja bem-sucedida, é importante testar cada camada à medida que esta é aplicada. Desta forma, caso uma falha ocorra, é possível falhar o processo de implantação o mais depressa possível e direcionar os esforços para resolver o problema (Humble and Farley, 2010).

Estes testes consistem em testes de implantação simples, capaz de cobrir os principais casos de erro, validando a presença ou falta de recursos essenciais. O objetivo consiste em aumentar a confiança de que a camada implantada está a funcionar (Humble and Farley, 2010).

A figura 7 apresenta o processo de implantação e configuração tanto do ambiente como da aplicação além dos testes a serem efetuados.

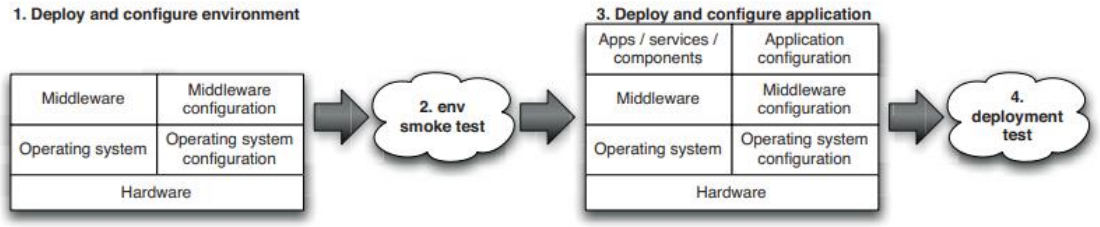

Figura 7 – Diagrama do processo de implantação do ambiente e aplicação (Humble and Farley, 2010)

Os testes de implantação da infraestrutura vão depender do ambiente em questão, no entanto, o objetivo é o mesmo: provar que a configuração do ambiente cumpre as expetativas.

2.7 Estratégias de Implantação e Lançamento de Software

Após a análise da *pipeline* de implantação, e de alguns dos *stages* mais importantes como compilação e implantação, resta perceber que tipo de estratégias de implantação e lançamento de versões podem ser adotadas.

Assim, esta secção aborda 2 técnicas que podem ser utilizadas para atingir lançamentos sem *downtime*: implantação *blue-green* e lançamento *canary*.

Um lançamento sem *downtime* consiste no processo de direcionar os utilizadores de uma versão para outra, praticamente instantaneamente. Se algo correr mal, também deve ser possível redirecionar os utilizadores de volta para a versão anterior (Humble and Farley, 2010).

A chave para um lançamento sem *downtime* bem-sucedido consiste no desacoplamento das várias partes do processo de lançamento, de forma que estes possam ocorrer da forma mais independente possível (Humble and Farley, 2010).

2.7.1 Implantações *Blue-Green*

Este tipo de implantações é uma das técnicas mais poderosas para lidar com a gestão de lançamentos de *software*.

A base deste tipo de implantação consiste na existência de 2 versões distintas do ambiente de produção, chamadas de azul e verde, como ilustrado na figura 8 (Humble and Farley, 2010).

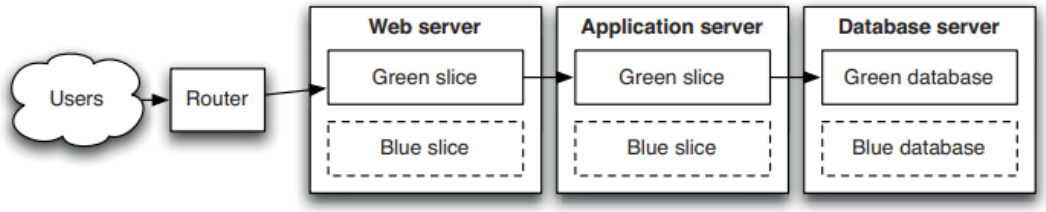

Figura 8 – Diagrama com implantação *Blue-Green* 8 (Humble and Farley, 2010)

Analisando a figura 8, é possível perceber que os utilizadores se encontram a ser direcionados para o ambiente verde, que consiste no atual ambiente de produção.

Quando for necessário implantar a nova versão da solução, esta será feita no ambiente azul, com tempo para a aplicação inicializar. Nesta fase, devem ser corridos testes de implantação de modo a verificar que o ambiente azul funciona como esperado (Yang *et al.*, 2018).

Finalmente, para redirecionar os utilizadores para a nova versão da solução, basta alterar a configuração do *router* para apontar para o ambiente azul. Este redireccionamento normalmente demora menos de 1 segundo (Humble and Farley, 2010).

Caso algo corra mal, é possível redirecionar os utilizadores de volta para o ambiente verde, trocando apenas a configuração do router. Assim, os utilizadores não são afetados pela falha na implantação e é possível investigar o que falhou no ambiente azul (Humble and Farley, 2010).

2.7.2 Lançamento *Canary*

O lançamento *Canary* consiste na implantação da nova versão da aplicação num pequeno conjunto dos servidores de produção, com o objetivo primário de obter *feedback* mais rápido.

A adoção desta abordagem permite reduzir o risco de lançamento de uma nova versão, uma vez que os erros encontrados na nova versão apenas serão encontrados pela minoria dos utilizadores (Humble and Farley, 2010; Rudrabhatla, 2020).

Após este pequeno conjunto de utilizadores validar a nova versão, esta pode ser implantada para os restantes utilizadores (Rudrabhatla, 2020).

A figura 9 captura o funcionamento deste tipo de estratégia.

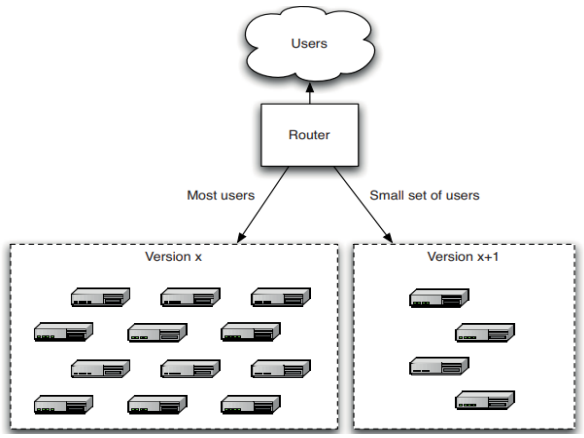

Figura 9 – Implantação *Canary* (Humble and Farley, 2010)

Tal como em implantações *blue-green*, a implantação da nova versão deve ser efetuada em servidores que não estejam a ser utilizados. Apenas após a validação de que a implantação foi bem-sucedida, é que devem ser direcionados utilizadores para a nova versão.

A implementação *canary* apresenta as seguintes vantagens (Humble and Farley, 2010):

- Facilita o processo de retorno à versão antiga;
- É possível comparar versão antiga com a versão atual em tempo real.
- É possível validar os requisitos de capacidade da aplicação, aumentando gradualmente a carga do sistema através do redireccionamento de utilizadores.

Apesar desta abordagem permitir múltiplas versões da aplicação em produção, é aconselhável limitar este valor a duas. Gerir mais do que duas versões, acarretará dificuldades em gerir *bugfixes* e toda a infraestrutura no geral (Humble and Farley, 2010).

2.8 Problemas e Soluções na Adoção de Entrega Contínua

Apesar das inúmeras vantagens descritas em detalhe ao longo deste documento, nem sempre o processo de adoção de entrega contínua é um caminho fácil. Raramente uma organização simplesmente substitui o seu processo de entrega de *software* por uma abordagem de entrega contínua. Normalmente, são dados passos nesse sentido, automatizando o processo de entrega de *software* e adotando práticas de integração contínua e entrega contínua.

Assim, o principal objetivo desta secção é analisar os principais problemas que as organizações enfrentam quando adotam uma abordagem de entrega contínua e demonstrar que as práticas analisadas extensivamente ao longo do documento atenuam ou até mesmo resolvem estes problemas.

No artigo (Laukkanen, Itkonen and Lassenius, 2017) são analisados 30 artigos diferentes que apresentam evidências empíricas da adoção de práticas de entrega contínua.

Neste artigo, os problemas foram separados em sete categorias distintas. Cinco destas categorias estão associadas a diferentes atividades do processo de desenvolvimento de *software*: desenho de *build*, desenho do sistema, integração, testes e lançamento de versões.

As outras duas categorias não se encontram diretamente ligadas ao desenvolvimento de *software*, sendo elas: humanas (e organizacional) e recursos. Como estas 2 categorias não são tão interessantes para o âmbito desta dissertação, não serão analisadas.

O gráfico apresentado na figura 10 mostra o número de casos por categoria dos problemas.

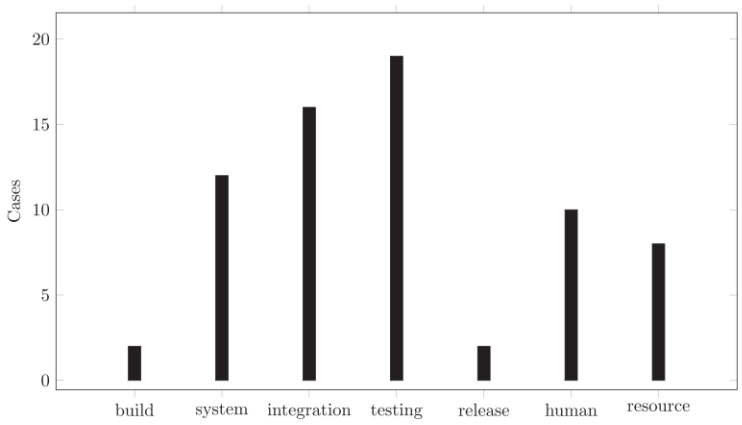

Figura 10 – Casos por categorias de problemas na adoção de entrega contínua (Laukkanen, Itkonen and Lassenius, 2017)

2.8.1 Problemas e Soluções de Desenho de Compilação

Dos 30 artigos utilizados como amostra, apenas dois reportaram problemas neste sentido. Estes problemas incidiram maioritariamente na inflexibilidade e complexidade do processo de compilação (Laukkanen, Itkonen and Lassenius, 2017).

Um dos casos refere que à medida que a aplicação crescia, casos especiais surgiam cada vez mais, tornando os *scripts* de compilação cada vez mais complexos e difíceis de alterar. Noutro caso o problema esteve associado com a modularização da aplicação, que causava uma maior complexidade do processo de compilação (Laukkanen, Itkonen and Lassenius, 2017). Estes problemas tornavam o processo de compilação difícil de manter e modificar.

De forma a solucionar estes problemas, devem ser seguidas as boas práticas abordadas na secção 2.6.1, como por exemplo: utilizar um *script* para cada *stage* da *pipeline* de implantação, utilizar os mesmos *scripts* para implantar a aplicação em todos os ambientes, assim como as restantes práticas.

2.8.2 Problemas e Soluções de Desenho do Sistema

Esta categoria foi uma das que apresentou mais casos, consiste essencialmente em problemas que têm origem em decisões de desenho do sistema.

Os problemas essencialmente dividiam-se em quatro pontos distintos: modularização do sistema, arquitetura inadequada, dependências internas e alterações a *schemas* de base de dados (Laukkanen, Itkonen and Lassenius, 2017).

Os efeitos causados por estes pontos apresentam bastante sobreposição e consistem essencialmente em: maior esforço e dificuldade no processo de implantação, processo de testes mais demorado/complexo e maior complexidade na compilação (Laukkanen, Itkonen and Lassenius, 2017).

Quanto a possíveis soluções para os problemas apresentados nesta categoria, algumas sugestões consistem em:

- Adotar a *pipeline* de implantação, seguindo uma estrutura semelhante à apresentada na secção 2.5.
- Utilizar a abordagem de *scripting* de implantação apresentada na secção 2.6.2, assim como as práticas sugeridas neste.

2.8.3 Problemas e Soluções de Integração

Esta categoria cobre essencialmente os problemas provenientes do código é integrado no fluxo principal. A tabela 1 apresenta os principais problemas assim como a sua descrição (Laukkanen, Itkonen and Lassenius, 2017).

Tabela 1 - Problemas de integração

Problemas	Descrição
Commits excessivamente	*Commits* que continham uma quantidade de alterações demasiado grandes.
Conflitos de *merge*	O *merge* de alterações revelava uma grande quantidade de conflitos.
Build não funcional	A *build* permanece não funcional durante um longo período.
Bloqueio do trabalho	Tarefas ficavam bloqueadas devido a integrações em fila ou até mesmo devido à *build* não estar funcional.
Branches demasiado	*Branches* duram demasiado tempo, aumentando conflitos.
Aprovação de integração lenta	Alterações eram aprovadas lentamente para o ramo principal.

De forma a solucionar os problemas apresentados na tabela 1, a adoção das práticas de entrega de *software* sugeridas na secção 2.3, assim como a adoção da *pipeline* de implantação descrita na secção 2.5, permitem mitigar a maioria destes problemas.

No entanto, um ponto não abordado importante para resolver estes problemas de integração consiste na adoção de uma estratégia de *branching* adequada às necessidades da organização.

2.8.4 Problemas e Soluções de Testes

Tal como o nome indica, esta categoria inclui todos os problemas relacionados com testes de *software*. É importante mencionar que esta consiste na categoria onde foi reportado o maior número de problemas nos casos analisados.

Os principais problemas discutidos consistem em: testes de resultado ambíguos, testes *flaky* e testes que ocupam demasiado tempo. Estes três pontos foram discutidos em pelo menos 6 casos (Laukkanen, Itkonen and Lassenius, 2017).

A tabela 2 sintetiza alguns dos principais problemas no âmbito dos testes, assim como a respetiva descrição de cada um (Laukkanen, Itkonen and Lassenius, 2017).

Tabela 2 - Problemas de testes

Problemas	Descrição
Ambiguidade no resultado dos testes	Os testes não são explícitos o suficiente para perceber se de facto foi um sucesso ou uma falha.
Testes *flaky*	Testes que aleatoriamente falham.
Testes de *hardware*	Testar com *hardware* que nem sempre está disponível.
Testes à interface gráfica	Dificuldades em testar a interface gráfica.
Código instável	Frequentemente, o *software* encontrava-se num estado que não podia ser testado.
Implantação problemática	Implantação demorada e propensa a erros
Testes complexos	Dificuldade em desenvolver testes muito complexos, que por exemplo, necessitam a disponibilização de um ambiente

No que toca aos problemas associados a esta categoria, a adoção da *pipeline* de implantação assim como dos diversos tipos de teste mencionados ao longo da secção 2.4, permitem mitigar a maioria dos problemas referidos acima.

2.8.5 Problemas e Soluções de Lançamento de Software

Apesar destes problemas terem sido reportados apenas num único estudo e o *downtime* durante o processo de implantação ser mencionado em apenas dois, estes apresentam dificuldades que possam vir a ser enfrentadas por qualquer equipa que adote entrega contínua (Laukkanen, Itkonen and Lassenius, 2017).

Os principais problemas reportados consistiam em (Laukkanen, Itkonen and Lassenius, 2017):

- Dificuldade em preservar os dados entre atualizações;
- Dificuldade em manter a documentação atualizada em cada entrega;
- Entregas frequentes causam um maior número de *bugs* em produção;
- Complicação a integrar com *software* de terceiros;
- *Downtime* intolerável devido à frequência de lançamentos.

Com o objetivo de solucionar os problemas acima, a utilização de uma abordagem sem *downtime* como a implantação *blue-green* ou a implantação *canary*, pretendem auxiliar a atingir o fim pretendido.

De modo a solucionar estes problemas, a principal sugestão consiste em dar prioridade à disponibilização de *hardware* que permita cumprir com as necessidades da solução como um todo (Laukkanen, Itkonen and Lassenius, 2017).

2.9 Casos de Sucesso

Na sequência da última secção que abordou as principais dificuldades enfrentadas por organizações que adotaram entrega contínua, assim como soluções para estas, esta secção do documento visa apresentar dois casos de sucesso: o caso da *HP FutureSmart* e o caso do programa de simplificação da *Suncorp*.

2.9.1 Caso da HP FutureSmart

A divisão da *HP LaserJet Firmware* constrói *firmware* que corre todos os seus *scanners*, impressoras e aparelhos multifunções. A equipa é constituída por cerca de 400 pessoas distribuídas entre os Estados Unidos, Brasil e India (Humble, 2023).

Ao fim de três anos, a implementação de entrega contínua, com um foco em: integração contínua, automação de testes, criação de um simulador para correr os testes numa plataforma virtual e reproduzir as falhas dos testes nas máquinas dos desenvolvedores, resultou nos seguintes benefícios (Humble, 2023):

- Redução dos custos de desenvolvimento em cerca de 40%;
- Os custos de desenvolvimento por programa baixaram aproximadamente 78%;
- Recursos que conduziam à inovação aumentaram num fator de 8 vezes.

2.9.2 Caso do Programa de Simplificação da *Suncorp*

O objetivo da *Suncorp* com o plano de simplificação consiste em: desativar os seus sistemas *legacy* de seguros gerais, melhorar a sua plataforma bancária *core* e começar um programa de excelência operacional.

Práticas *lean*, práticas de entrega ágeis e *frameworks* de testes automáticas foram algumas das estratégias essenciais para cumprir com o programa de simplificação (Humble, 2023).

Neste processo, a *Suncorp* foi capaz de reduzir 15 sistemas complexos de seguros pessoais e de vida para 2, assim como desativar 12 sistemas *legacy* (Humble, 2023).

A confiança neste plano foi consideravelmente alta, onde no relatório anual de 2014, foram antecipadas poupanças de $225 milhões para 2015 e $265 milhões para 2016 (Humble, 2023).

2.10 Sumário

Ao longo de todo o capítulo de estado de arte a nível conceptual, foram abordados em detalhe os principais conceitos da área e abordagens, dado ênfase em especial às melhores práticas de entrega de *software*, estratégias de testes e à *pipeline* de implantação.

Esta análise efetuada ao estado de arte a nível conceptual permitiu responder a diversas questões de investigação que haviam sido colocadas:

- Uma boa abordagem de entrega de *software* deve seguir os princípios abordados ao longo deste capítulo como: o processo deve ser repetível e fiável, o mais automático possível, devidamente documentado, entre outros.
- A automatização do processo de entrega de *software* deverá ser feita através da adoção de uma *pipeline* de *implantação*, seguindo as práticas e princípios abordados;
- A qualidade de *software* entregue deve ser garantida através de uma estratégia de testes robusta que inclua testes como: testes unitários, testes de aceitação, testes de aceitação funcionais automáticos e manuais, entre outros;
- A principal diferença entre as abordagens de entrega de *software* tradicionais e as de entrega de aplicações baseadas em análise de dados consiste na estratégia para garantir a qualidade, que lida com problemas adicionais, como o problema *oracle.*

Apesar de nenhuma abordagem analisada solucionar inteiramente o problema desta dissertação, a combinação de algumas das estratégias de teste analisadas assim como a adoção da *pipeline* de implantação, permitiram fornecer uma base em que possa ser construída uma solução apropriada para o problema.

3 Estado de Arte a Nível Tecnológico

Durante este capítulo, são estudadas as principais tecnologias mais relevantes para o âmbito da tese, isto é, que dão suporte ao estudado no capítulo 2.

As tecnologias analisadas encontram-se separadas em três categorias distintas: ferramentas de integração contínua e entrega contínua, ferramentas de testes e ferramentas associadas ao processo de implantação já utilizadas dentro da *proGrow S.A*.

3.1 Ferramentas de Integração Contínua e Entrega Contínua

Devido à crescente popularidade das práticas de integração contínua e entrega contínua, existem cada vez mais ferramentas que visam dar suporte a estas práticas.

Nesta secção são abordadas e comparadas três ferramentas distintas: *Jenkins*, *Github Actions* e *Team City*.

3.1.1 Jenkins

O *Jenkins* é um servidor de automação que pode ser utilizado para automatizar tarefas relacionadas com o processo de compilação, testes e implantação de *software* (Jenkins, 2023).

Este consiste num servidor de integração contínua altamente extensível, através de diversos *plugins*, quer permitem construir e testar praticamente qualquer *software* efetivamente (Rai and Dhir, 2015).

Sendo uma das ferramentas mais dominantes do mercado, o *Jenkins* pode ser adotado por praticamente qualquer equipa, em qualquer projeto, suportando um vasto leque de linguagens de programação e tecnologias (Jenkins, 2023).

Esta tecnologia possui as seguintes vantagens (Jenkins, 2023):

- É gratuita;
- É fácil de instalar, contendo já pacotes para *Windows*, *Linux* e *macOS*;
- É altamente extensível devido à sua arquitetura de *plugins*;
- Permite a distribuição da carga em múltiplas máquinas.

3.1.2 GitHub Actions

O *GitHub Actions* é uma plataforma do *GitHub* destinada a integração contínua e entrega contínua, permitindo a criação de fluxos que automatização diversos processos como: compilação, testes e implantação (GitHub, 2023b).

As principais vantagens oferecidas por esta tecnologia consistem em (GitHub, 2023a):

- Altamente extensível, permitindo a utilização de fluxos desenvolvidos pela comunidade (chamados de *Actions*);
- Permite execução paralela em múltiplos sistemas operativos com diferentes versões;
- Suporte à maioria das linguagens de programação;
- Disponibiliza uma interface gráfica acessível, clara e consistente.

Esta ferramenta torna-se uma escolha ainda mais vantajosa para aquelas equipas que já utilizam ferramentas *GitHub*, como por exemplo o seu mecanismo de controlo de versões.

3.1.3 Team City

Com a premissa de atingir com sucesso integração contínua, entrega contínua e implantação contínua em qualquer projeto, a *JetBrains* desenvolveu o *TeamCity* (JetBrains, 2023).

O *Team City* é uma aplicação CI/CD de uso geral que permite *workflows* flexíveis, colaboração e diversas práticas de desenvolvimento. Essencialmente, esta solução oferece (Melymuka, 2012; JetBrains, 2023):

- Alta escalabilidade, uma vez que um único servidor é capaz de orquestrar diversos agentes de compilação;
- Facilmente integrável com a maioria dos provedores *cloud*;
- Possui um mecanismo inteligente de interpretação de resultados de testes;
- Uma vez que este pode ser integrado com a *IDE*, pode ser feito o compilação e os testes sem a necessidade de ser feito um *commit*;
- Extensibilidade através de *plugins*;

3.1.4 Comparação entre Jenkins, GitHub Actions e TeamCity

Após abordar algumas das principais tecnologias concorrentes no âmbito de integração contínua e entrega contínua, resta compará-las de modo a perceber como cada uma se posiciona entre diversos requisitos.

Deste modo, a tabela 3 apresenta esta comparação (GitHub, 2023a, 2023b; Jenkins, 2023; JetBrains, 2023):

Tabela 3 - Comparação *Jenkins vs Gitlab CI/CD vs TeamCity*

	Jenkins	***GitHub Actions***	***Team City***
CI/CD incluído	Sim	Sim	Sim
Preço	Gratuito (auto-hospedado)	Possui versão gratuita, mas é limitada (exceto quando auto-hospedado)	Possui versão gratuita, mas é limitada (exceto quando auto-hospedado)
Funcionalidades e extensibilidade	Muitas funcionalidades e alta extensibilidade	Muitas funcionalidades e alta extensibilidade	Muitas funcionalidades, mas baixa extensibilidade
Suporte	Não	Sim, mas apenas para utilizadores pagos	Sim, mas apenas para utilizadores pagos
Desempenho	Elevado	Elevado	Elevado
Interface gráfica apelativa	Não	Sim	Sim

É importante mencionar que a secção 4.4 compara novamente estas ferramentas, com o intuito de escolher uma destas tendo em conta a sua adequação ao problema desta dissertação.

3.2 Ferramentas de Testes

Tendo sido abordado em detalhe diferentes estratégias de teste na secção 2.4 desta dissertação, é importante perceber que ferramentas podem dar suporte às estratégias de teste mencionadas.

É importante mencionar que as tecnologias de teste que serão abordadas, estão limitadas pelo fato da *stack* tecnológica adotada pela equipa ter por base *NodeJS* com *TypeScript*.

3.2.1 Jest

O *Jest* é uma *framework* de testes que permite efetuar testes unitários e de integração com foco em manter a simplicidade (JestJS, 2023).

Esta ferramenta é bastante bem documentada, requer pouca configuração e pode ser estendida de forma a cumprir qualquer requisito (JestJS, 2023).

Esta *framework* oferece como principais vantagens (JestJS, 2023):

- Rapidez e segurança, através da execução paralela de testes;
- Cobertura de código sem configuração adicional;
- Facilidade em fazer *mocking*;
- Bom tratamento de exceções, providenciando um contexto adequado quando um teste falha.

3.2.2 Mocha

Tal como o *Jest*, a *Mocha* é uma *framework* de testes habitualmente utilizada para testes unitários e de integração. Esta *framework* é bastante flexível que visa facilitar o desenvolvimento de testes assíncronos (Mocha, 2023).

O *Mocha* possui como principais vantagens (Mocha, 2023):

- Execução de testes em paralelo;
- Suporte a *Browsers*;
- Relatório de cobertura dos testes.

3.2.3 Comparação entre Jest e Mocha

De modo a conseguir efetuar tanto testes unitários como testes de integração, será necessário adotar uma destas tecnologias.

Deste modo, esta secção fará uma comparação destas tecnologias em três critérios distintos: funcionalidades, desempenho e facilidade de adoção.

Relativamente às funcionalidades, o *Jest* inclui *mocking* e *assertion*, além de suportar *snapshots* de testes. Quanto ao *Mocha*, nenhuma destas 3 funcionalidades é suportada nativamente, sendo necessário recorrer a outras bibliotecas. Em suma, enquanto o *Jest* tem mais funcionalidades, o *Mocha* é mais flexível (Pratomo, Schriek and Veen, 2020).

No que toca a desempenho, o estudo (Pratomo, Schriek and Veen, 2020) comparou *Jest* e o *Mocha* neste requisito, concluindo que o *Mocha* era, em média, 34% mais rápido.

Finalmente, no que toca à facilidade de adoção, ambas são tecnologias relativamente simples de adotar. No entanto, para utilizar o *Mocha* na sua plenitude é necessário recorrer a outras tecnologias, tornando a sua curva de aprendizagem maior.

3.2.4 Selenium

O *Selenium* consiste numa ferramenta *open-source* de testes capaz de simular qualquer tipo de interação realizada por um utilizador final, como inserir texto, selecionar *drop-down*, entre outros (Selenium, 2023).

Uma importante característica desta ferramenta consiste no fato desta ferramenta não ser compilada juntamente com o código fonte da aplicação, o que significa que a aplicação testada é exatamente igual à implantada (Selenium, 2023).

Dadas estas características, esta ferramenta torna-se especialmente útil para testes de aceitação funcionais e testes de implantação.

A principal vantagem da adoção desta ferramenta consiste no suporte de uma única interface para todos os principais *browsers*, facilitando imensamente o desenvolvimento de testes, uma vez que um único teste pode ser executado em múltiplos *browsers* diferentes (Singh and Tarika, 2014; Selenium, 2023).

3.2.5 Cypress

Tal como o *Selenium*, esta ferramenta *open-source* permite a elaboração de teste de aceitação funcionais e testes de implantação.

Com o objetivo de facilitar a escrita de testes mais fiáveis, o *Cypress* consiste essencialmente numa aplicação instalada localmente, que eventualmente recorre ao sistema operativo para automatização de algumas tarefas (Cypress, 2023).

Diferente do *Selenium*, esta ferramenta é compilada juntamente com o código fonte da aplicação, permitindo que esta tenha acesso tanto ao *front-end* como ao *back-end* da aplicação (Cypress, 2023).

Esta ferramenta oferece o seguinte conjunto particular de funcionalidades (Cypress, 2023):

- Sistema de *debug* bastante completo;
- Permite consultar o que aconteceu em cada teste, passo por passo;
- Orquestração de testes inteligente;

3.2.6 Comparação entre Selenium e Cypress

Tanto *Selenium* como *Cypress* são duas ferramentas poderosas para efetuar tanto testes de aceitação funcionais como testes de implantação.

A tabela 4 visa comparar as duas tecnologias relativamente a um conjunto de critérios (Cypress, 2023; Selenium, 2023).

Tabela 4 - Comparação *Selenium vs Cypress*

	Selenium	***Cypress***
Facilidade de uso	Relativamente complexo devido à sua sintaxe específica	Simples
Velocidade	Bastante rápido	Bastante rápido
Facilidade de instalação	Instalação um pouco trabalhosa	Apenas executar um comando
***Browsers* suportados**	Todos os *browsers* populares	*Browsers* baseados em chromium e *firefox*
Documentação	Documentação razoável	Boa documentação
Comunidade	Excelente	Ainda em crescimento

3.2.7 JMeter

O *JMeter* é uma aplicação *open-source* desenhada para efetuar testes de carga e medir desempenho, permitindo efetuar testes de aceitação não funcionais com facilidade. Originalmente foi desenvolvida para testar aplicações *web*, sendo posteriormente expandida para suportar outro tipo de testes (JMeter, 2023).

As principais funcionalidades disponibilizadas por esta tecnologia consistem em (Nevedrov, 2006; JMeter, 2023):

- Permite testar diferentes tipos de aplicações, servidores e protocolos como: aplicações e serviços *web*, bases de dados, serviços de e-mail entre outros;
- Uma IDE que permite construir, gravar e fazer *debug* aos testes;
- Permite extrair informação de diferentes formatos de resposta como: *HTML*, *JSON*, *XML*, entre outros formatos textuais;
- Extensível através de uma API disponibilizada.

3.2.8 K6

O *K6* é uma ferramenta de testes *open-source* dedicada a testes de carga que permite efetuar testes de aceitação não funcionais.

A adoção desta tecnologia permite efetuar testes de forma eficiente e fiável, de modo a encontrar possíveis problemas o mais cedo possível, contribuindo para uma aplicação mais resiliente e escalável (K6, 2023).

As principais funcionalidades proporcionadas pelo *K6* são (K6, 2023):

- Ferramenta CLI com *APIs* para auxiliar os desenvolvedores;
- *Scripting* em *javascript*, com suporte a módulos locais e remotos;
- Disponibiliza funcionalidades como *check* e *Thresholds* para facilitar e otimizar o desenvolvimento de testes.

3.2.9 Comparação entre JMeter e K6

No âmbito de tecnologias que permitem efetuar testes de aceitação não funcionais, foram abordadas o *JMeter* e o *K6*, restando agora comprar estas ferramentas.

A tabela 5 compara o JMeter e o *K6* num conjunto de critérios.

Tabela 5 - Comparação *JMeter vs K6*

	JMeter	***K6***
Facilidade de instalação	Necessita de *Java* e compatibilidade com a versão do *JDK*	Não necessita de tecnologias adicionais
Interface gráfica	Sim	Não
Consumo de Recursos	Alto	Relativamente baixo
Manutenção	Um pouco difícil	Fácil
Comunidade	Excelente	Ainda em crescimento

3.3 Ferramentas Associadas à Implantação

O objetivo desta secção consiste em analisar as principais tecnologias associadas com o processo de implantação que estão atualmente em uso dentro da *proGrow S.A.*

De forma a resolver o problema proposto nesta dissertação, é crucial que exista familiaridade com estas tecnologias, uma vez que estas, em princípio, estarão envolvidas no processo de implantação.

3.3.1 Docker

Fundamentalmente, o *Docker* permite não só empacotar uma aplicação e as suas respetivas dependências, como também executar esta, num ambiente relativamente isolado chamado de contentor (Docker, 2023).

Devido à natureza do contentor, não existe dependência com *software* instalado no *host*, garantindo a consistência do comportamento da aplicação em diferentes ambientes (Docker, 2023).

De forma resumida, o processo de criação de um contentor encontra-se ilustrado na figura 11.

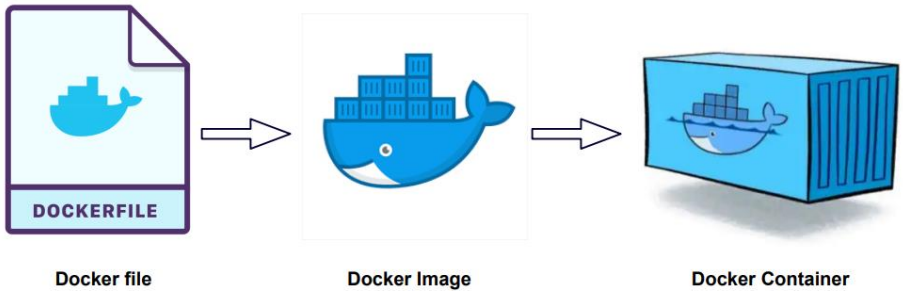

Figura 11 – Criação de um contentor (MrDevSecOps, 2023)

Analisando a figura 11, é possível perceber que um contentor é feito a partir de uma imagem, que essencialmente consiste no conjunto de instruções destinadas à construção de um contentor(Docker, 2023).

Por sua vez, uma imagem é criada a partir de um *Dockerfile*, que se trata do processo habitual para construção de imagens. Este *Dockerfile* consiste num *script* com as instruções necessárias para empacotar a aplicação e as suas dependências, criando assim a respetiva imagem (Docker, 2023).

3.3.2 Repositório Nexus

Cumprindo com a boa prática de entrega de *software* referida na secção 2.5.1.1 "Construir os Binários Apenas Uma Vez", é necessário guardar os artefactos de *software* gerados. Caso estes artefactos não sejam guardados, é necessário construir os binários sempre que se pretenda executar a aplicação.

Dentro da *proGrow S.A.*, os artefactos de *software* gerados são tipicamente imagens *Docker*, que são guardadas num repositório privado utilizando o repositório *Nexus*.

De modo sucinto, o repositório *Nexus* permite gerir artefactos num ambiente privado e controlado. Assim, qualquer utilizador de uma organização é capaz de publicar e obter os artefactos que pretende (Sonatype, 2023).

Este repositório apresenta uma elevada escalabilidade através da utilização de armazenamento dinâmico, políticas de limpeza e resiliência de vários nós (Sonatype, 2023).

3.3.3 Amazon Web Services

Amazon Web Services (AWS) é uma plataforma *cloud* com uma vasta estrutura, que promete segurança, redução de custos e um elevado ritmo de inovação (Amazon, 2023a).

Entre os inúmeros serviços disponibilizados pela AWS, existe um conjunto deles destinados à hospedagem de aplicações.

Posto isto, o objetivo desta secção consiste em analisar alguns destes serviços que são utilizados dentro da *progrow S.A.*

3.3.3.1 Amazon Elastic Compute Cloud

O *Amazon Elastic Compute Cloud* (EC2) é um serviço que disponibiliza capacidade computacional na *cloud*, permitindo a criação e execução de praticamente qualquer aplicação (Amazon, 2023b).

Esta capacidade computacional é disponibilizada ao utilizador através instâncias. Existem diversos tipos de instância de forma a adaptarem-se às diferentes necessidades dos utilizadores, por exemplo, estas instâncias podem ser otimizadas para computação, memória, armazenamento, entre outros (Amazon, 2023b).

Um dos principais pontos fortes deste serviço consiste na sua escalabilidade (vertical). Automaticamente, é possível escalar as instâncias verticalmente em picos de utilização, assim como reduzir as instâncias em períodos de baixa utilização de forma a minimizar custos (Amazon, 2023b).

3.3.3.2 Amazon Elastic Container Service

O *Amazon Elastic Container Service* (*ECS*), é um serviço de orquestração de contentores que facilita todo o processo de implantação, gestão e escalabilidade de aplicações em contentores (Amazon, 2023c).

Estes contentores que são orquestrados são agrupados num grupo lógico chamado *Cluster*. Este *Cluster* pode gerir tanto recursos EC2, como recursos na infraestrutura do utilizador ou até mesmo abstrair os recursos que são utilizados (Amazon, 2023e).

Independentemente dos recursos escolhidos, a sua gestão é abstrata, feita essencialmente a partir de uma *task definition*. Esta especifica de que forma uma aplicação é executada, através de parâmetros como: imagem *Docker* a utilizar, requisitos de *CPU* e memória, entre outros (Amazon, 2023b).

Também é possível gerir um número específico de instâncias de *task definitions* (também conhecidos como *tasks*) simultaneamente, através da utilização de serviços ECS. Estes serviços garantem que, quando uma determinada instância falha, esta é reposta, garantindo que o número pretendido de instâncias se encontra a trabalhar (Amazon, 2023a).

Idealmente, com a utilização de serviços deve ser utilizado um *load balancer*. Este fica encarregue da gestão do tráfego para as instâncias de uma *task definition* de um determinado serviço (Amazon, 2023a).

De uma forma resumida, o ciclo de vida de uma aplicação ECS é apresentado na figura 12.

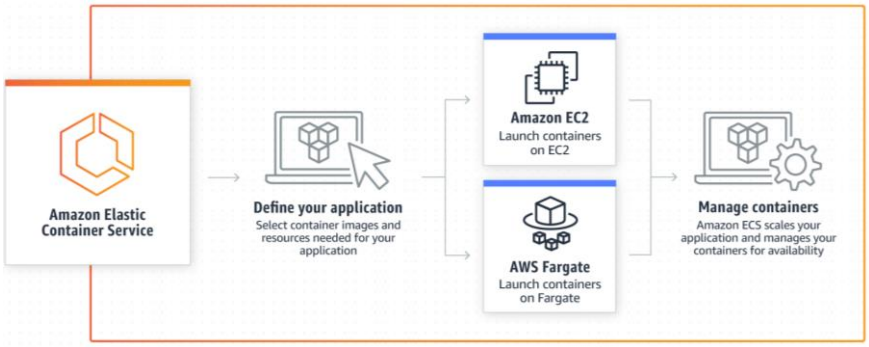

Figura 12 – Ciclo de vida de aplicação ECS (Computer Consulting, 2023)

Observando a figura 12, é possível perceber que o primeiro passo consiste na definição de uma *task definition* responsável por definir a aplicação. Após a sua definição, esta aplicação é implantada numa determinada infraestrutura (como por exemplo contentores *EC2*). Finalmente a aplicação é gerida pelo *ECS*, tendo por base o serviço configurado e a *task definition* da aplicação.

3.3.4 Terraform

A gestão da infraestrutura de uma organização não costuma a ser um processo simples. É comum existirem diversos ambientes, para fins distintos, muitas vezes com tecnologias diferentes.

Posto isto, o *Terraform* apresenta uma abordagem diferente, permitindo automatizar a infraestrutura em diversos provedores *cloud* através de código. Assim, é possível tratar a infraestrutura como *software* tradicional, versionando e partilhando esta (Terraform, 2023).

Como principais vantagens, o *Terraform* apresenta (Terraform, 2023):

- Código para gerir a infraestrutura, desde servidores e base de dados até políticas de *firewall*;
- Suporte a diversos provedores *cloud* como AWS, *Azure* e *Google Cloud*;

3.4 Sumário

O capítulo de estado de arte a nível tecnológico permitiu analisar diversas tecnologias no âmbito de entrega de *software* e de testes, além de abordar tecnologias já utilizadas pela *proGrow S.A.*

Embora não tenha sido encontrada uma única tecnologia que fosse suficiente para atacar o problema desta dissertação, o estudo realizado servirá como base para a escolha das tecnologias que serão adotadas na solução.

Assim, é possível afirmar que a questão de pesquisa que motivou este capítulo encontra-se respondida, uma vez que foram analisadas diversas tecnologias capazes de auxiliar na construção de uma solução que entregue *software* de forma automática garantindo a qualidade deste.

4 Análise de Valor

Neste capítulo é realizada a análise de valor, começando por analisar alguns dos principais conceitos neste domínio, identificar e analisar a oportunidade e terminando na proposta de valor.

Também neste capítulo, é utilizada a análise FAST para analisar se a solução cumpre com as necessidades do projeto.

Finalmente, é utilizado o método de análise hierárquica para comparar as ferramentas *Jenkins*, *GitHub Actions* e *TeamCity*.

4.1 Processo de Inovação

O processo de inovação pode ser dividido em três áreas distintas: o *fuzzy front end* (*FFE*), o processo de desenvolvimento do novo produto (NPD) e a comercialização (Koen *et al.*, 1996), como indicado na figura 13.

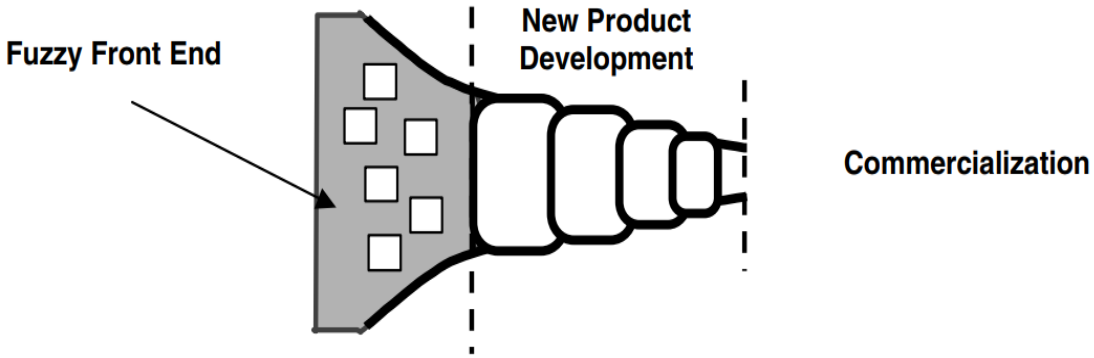

Figura 13- Processo de inovação (Koen *et al.*, 1996)

Devido à dificuldade em comparar práticas de *FFE* entre diversas empresas, foi concebido o modelo de desenvolvimento de um novo conceito (NCD), com o intuito de fornecer conhecimento e uma nova terminologia para *FFE*(Koen *et al.*, 1996).

O modelo de desenvolvimento de um novo conceito (NCD) apresentado na figura 14 consiste em três partes distintas (Koen *et al.*, 1996):

- A *engine*, que corresponde à porção de liderança, cultura e estratégia de negócio da organização que impulsiona os cinco elementos-chave controlados pela empresa.
- A área interior, que define cinco atividades distintas do FFE, sendo estas: identificação da oportunidade, análise da oportunidade, geração da ideia e enriquecimento, seleção da ideia e definição do conceito;
- Os fatores de influência, que consistem em capacidades organizacionais, fatores externos que podem estar envolvidos.

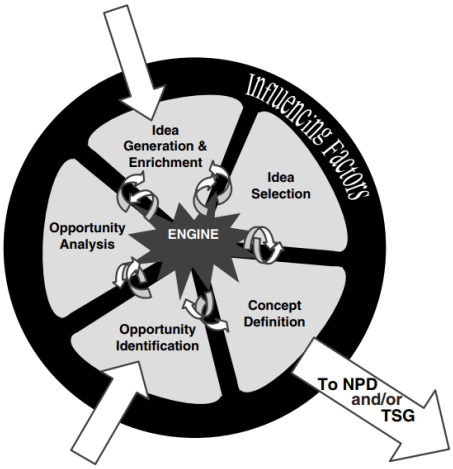

Figura 14 – Diagrama de modelo de desenvolvimento de um novo conceito (Koen *et al.*, 1996)

4.1.1 Identificação da Oportunidade

Como havia sido referido no primeiro capítulo desta dissertação, a necessidade de entregar *software* de forma rápida e fiável traduz-se na necessidade de reformular os processos habituais de entrega de *software* para processos mais elaborados.

No que toca a entrega de *software*, equipas de alto desempenho, que adotam entrega contínua, quando comparadas com equipas de baixo desempenho, mostram-se vastamente superiores em diversas métricas. Estas, além de apresentarem entregas muito mais frequentes, recuperam de *downtime* cerca de 170 vezes mais rápido, além de falharem 5 vezes menos quando fazem uma alteração (Forsgren, Humble and Kim, 2018).

Outro ponto crucial é a estratégia de testes (Forsgren, Humble and Kim, 2018), extensivamente abordado na secção 2.5 deste documento.

A adoção de uma estratégia de testes adequada e automática permite obter ganhos de eficiência, confiabilidade e qualidade, além de permitir a redução de custos e recursos humanos (Rafi *et al.*, 2012; Asfaw, 2015).

Como o atual processo de entrega de *software* dentro da *proGrow S.A.* é manual, trabalhoso, sem uma estratégia de testes automática e propenso a erros, existe uma oportunidade clara em melhorar este processo.

Aliado a isto, existe uma limitação nas abordagens atuais quando se trata de testar aplicações cuja qualidade depende dos dados gerados por esta.

Assim, surgiu a oportunidade de desenvolver um processo de entrega de *software* capaz de colmatar os problemas do processo antigo.

4.1.2 Análise da oportunidade

O atual processo de entrega de *software* dentro da organização é consideravelmente limitado para os padrões atuais no âmbito de engenharia de *software*.

Analisando o atual processo de entrega de *software*, este possui essencialmente duas lacunas distintas:

- Uma entrega manual e trabalhosa, que depende de um ou vários guiões detalhando como deve ser feita a implantação;
- A estratégia de testes consistir essencialmente de testes de aceitação realizados pelo utilizador.

Deste modo, a automatização da entrega de *software* e a adoção de uma estratégia automática de testes permitirá à *proGrow S.A.* aumentar a sua produtividade e frequência de entrega, além de reduzir os erros de *software* e a quantidade de recursos alocados à entrega.

Por outras palavras, a adoção de um processo de entrega contínua permitirá agregar valor à organização tornando-a mais competitiva (Forsgren, Humble and Kim, 2018).

No entanto, é importante mencionar que a adoção destes novos processos na organização traz algumas desvantagens como a necessidade de formar os desenvolvedores nas práticas a adotar, assim como na disciplina para os desenvolvedores manterem os novos processos.

4.2 Proposta de Valor

De forma a avançar com a adoção de um mecanismo de entrega de *software* automático, é necessário perceber se os benefícios desta solução superam os seus custos associados.

Com este objetivo em mente, é necessário perceber que o valor da solução é subjetivo, depende da avaliação geral do consumidor sobre a utilidade da solução com base nos seus benefícios e nos seus custos. Este conceito é conhecido por *perceived* value (Nicola, 2023a).

Deste modo, é necessário elaborar a proposta de valor, que consiste na estratégia especifica para competir por novos clientes.

Assim, a proposta de valor analisará, por um lado, os benefícios, sacrifícios e tarefas do cliente, por outro lado, de que forma a solução alivia a dor e traz ganhos ao cliente. Por outras palavras, a proposta de valor apresenta o conjunto de produtos e serviços que são de valor para o cliente (Nicola, 2023).

No âmbito desta dissertação, o cliente consiste na própria *proGrow S.A.*, mais especificamente nos seus engenheiros de *software*. Assim, será analisado de que forma o projeto traz valor para os desenvolvedores da *proGrow S.A.*

De modo a representar a proposta de valor, será utilizado o modelo de *Osterwalder*, como apresentado na figura 15.

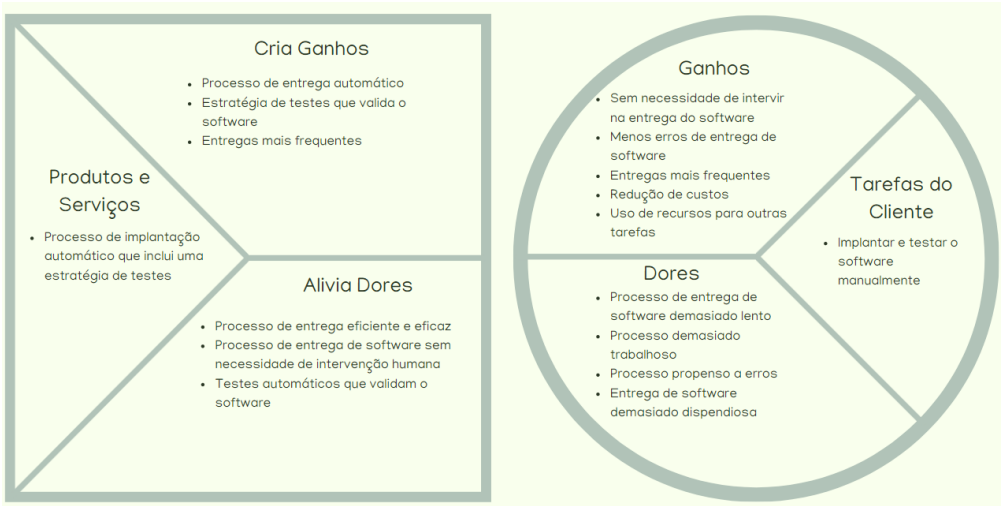

Figura 15 – Proposta de Valor

4.3 Técnica Sistemática de Análise Funcional

A técnica sistemática de análise funcional (FAST) consiste em uma técnica para desenvolver uma representação gráfica das relações entre as funções do projeto, produto, processo ou serviço baseada nas questões “Como?” e “Porquê?” (Nicola, 2023b).

Esta técnica permite analisar se a solução cumpre com as necessidades do projeto, identificando as funções necessárias, assim como funções duplicadas e até desnecessárias (Nicola, 2023b).

A figura 16 apresenta o diagrama FAST para o projeto em pauta.

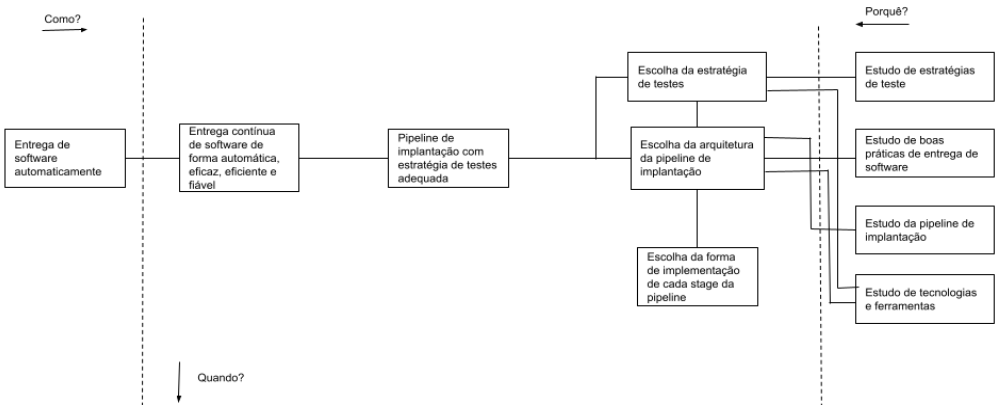

Figura 16 – Diagrama *FAST*

Como qualquer diagrama *FAST*, este pode ser lido da esquerda para a direita, respondendo à pergunta “Como?”, ou pode ser lido no sentido contrário, respondendo à questão “Porquê?”.

Lendo o diagrama da esquerda para a direita, temos o principal objetivo da dissertação que consiste em entregar *software* automaticamente. Para tal, é necessário que este processo, além de ser automático seja eficaz, eficiente e fiável. Para cumprir com todos estes quesitos, deve ser utilizada uma *pipeline* de implantação com uma estratégia de testes adequada.

De forma a implementar uma *pipeline* de implantação com uma estratégia de testes adequada, é necessário escolher a estratégia de testes, a arquitetura da *pipeline* de implantação e a forma como cada *stage* da *pipeline* é implementado. Para tal, foi necessário estudar as melhores práticas de entrega de *software*, a *pipeline* de implantação, estratégias de teste e tecnologias e ferramentas que dessem suporte a esta implementação.

Em suma, a elaboração deste diagrama permitiu compreender as principais etapas necessárias para alcançar o objetivo.

4.4 Método de Análise Hierárquica

Quando existe a necessidade de fazer uma escolha entre um conjunto discreto de alternativas, existem diversas técnicas numéricas para este fim, chamadas de métodos de decisão multicritério (Nicola, 2023a).

Um dos principais métodos de decisão multicritério é o método de análise hierárquica (AHP), criado pelo professor Thoma L. Saaty em 1980. Este método permite tanto o uso de critérios quantitativos como qualitativos no processo de avaliação (Nicola, 2023a).

A adoção deste método permite dividir a decisão em diferentes níveis hierárquicos, facilitando não só a sua compreensão como também a avaliação (Nicola, 2023a).

Dito isto, será utilizado o método de análise hierárquica para analisar qual das ferramentas de integração contínua e entrega contínua melhor se adequa ao projeto desta dissertação. Assim, as ferramentas selecionadas para esta análise consistem no: *Jenkins*, *GitHub Actions* e *TeamCity*, uma vez que foram estas as tecnologias estudadas no capítulo 3 "Estado de Arte a Nível Tecnológico".

De forma a selecionar uma destas tecnologias, foi feito um levantamento com a equipa que utilizará a solução desenvolvida sobre os critérios mais relevantes na seleção deste tipo de tecnologias.

Assim, o conjunto de critérios selecionado consiste em:

- Facilidade de aprendizagem e utilização: consiste na facilidade em adotar a ferramenta, usabilidade da ferramenta, documentação e suporte;
- Funcionalidades e extensibilidade: trata-se da quantidade e qualidade das funcionalidades disponibilizadas pela ferramenta e na facilidade em estender a quantidade de funcionalidades;
- Custo: consiste no custo da tecnologia.

Assim, dado o conjunto de alternativas de ferramentas e os critérios de seleção, a figura 17 apresenta a árvore hierárquica de decisão para a seleção da ferramenta de CI/CD.

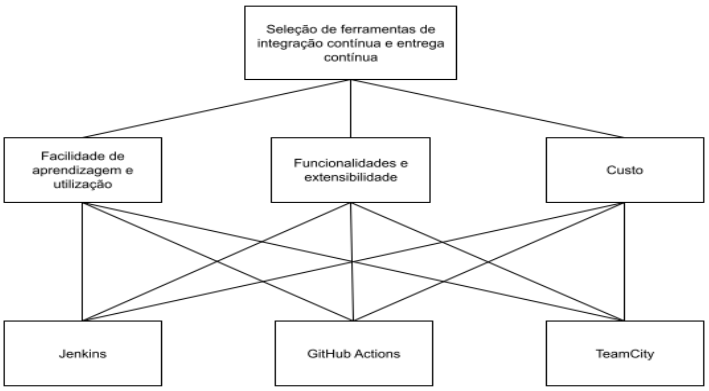

Figura 17 – Árvore hierárquica de decisão para seleção de ferramenta CI/CD

De forma a dar início às comparações, é necessário começar por atribuir níveis de importância nestas comparações. Para tal, será utilizada a escala fundamental de *Saaty*.

Tabela 6 – Comparação de critérios

	Facilidade de aprendizagem e utilização	**Funcionalidades e extensibilidade**	**Custo**
Facilidade de aprendizagem e utilização	1	2	1/2
Funcionalidades e extensibilidade	1/2	1	1/3
Custo	2	3	1
Soma	7/2	6	11/6

A partir dos valores da tabela 6, ignorando a última linha correspondente à soma, é possível escrever os valores em forma de matriz, como representado na matriz A:

$$A = \begin{bmatrix} 1 & 2 & 1/2 \\ 1/2 & 1 & 1/3 \\ 2 & 3 & 1 \end{bmatrix} \quad (1)$$

Após a comparação entre os elementos da hierarquia, é necessário normalizar os valores da matriz de comparação, para tal, cada valor da matriz é dividido pelo total da sua respetiva coluna (Nicola, 2023a). Além da normalização, é necessário também obter a prioridade relativa por critério, através do cálculo da média aritmética de cada linha. A tabela 7 apresenta ambas as operações.

Tabela 7 – Normalização dos valores da matriz de comparação e prioridade relativa

	Facilidade de aprendizagem e utilização	**Funcionalidades e extensibilidade**	**Custo**	**Prioridade relativa**
Facilidade de aprendizagem e utilização	2/7	1/3	3/11	0,2973
Funcionalidades e extensibilidade	1/7	1/6	2/11	0,1638
Custo	4/7	1/2	6/11	0,5390

Analisando a tabela 7, é possível obter o vetor de prioridades, que corresponde à coluna prioridade relativa e identifica a ordem de importância de cada critério, sendo este:

$$X = \begin{bmatrix} 0{,}30 \\ 0{,}16 \\ 0{,}54 \end{bmatrix}$$

(2)

Analisando o vetor de prioridades, é possível deduzir que o custo tem o maior peso de 0,54, de seguida o critério de facilidade de aprendizagem e utilização apresenta um peso de 0,30 e, por fim, o critério de funcionalidades e extensibilidade com menor peso de 0,16.

De seguida, é necessário analisar a consistência das prioridades relativas. Para tal, será calculada a razão de consistência (RC), que permite medir o quanto os julgamentos foram consistentes em relação a grandes amostras de juízos aleatórios (Nicola, 2023a).

Considerando a matriz de comparação não normalizada M e o vetor de prioridades X, o cálculo da razão de consistência é feito de acordo com a seguinte formula:

$$A \times X = \lambda_{max} \times X$$

(3)

Substituindo pelos valores obtidos previamente:

$$\begin{bmatrix} 1 & 2 & 1/2 \\ 1/2 & 1 & 1/3 \\ 2 & 3 & 1 \end{bmatrix} \times \begin{bmatrix} 0{,}30 \\ 0{,}16 \\ 0{,}54 \end{bmatrix} = \lambda_{max} \begin{bmatrix} 0{,}30 \\ 0{,}16 \\ 0{,}54 \end{bmatrix}$$

(4)

Multiplicando a matriz A pelo vetor de prioridade X, é obtido o resultado seguinte:

$$\begin{bmatrix}0{,}89\\0{,}49\\1{,}62\end{bmatrix} = \lambda_{max} \begin{bmatrix}0{,}30\\0{,}16\\0{,}54\end{bmatrix}$$

(5)

De modo a obter o valor próprio λ_{max}, é necessário calcular a média da divisão de cada linha dos vetores, tal como apresentado de seguida:

$$\lambda_{max} = \frac{(0{,}89 \div 0{,}30\ ) + (0{,}49 \div 0{,}16\ ) + (1{,}62 \div 0{,}54\ )}{3} = 3{,}01$$

(6)

Após o cálculo do valor próprio, é possível calcular o índice de consistência da seguinte forma:

$$IC = \frac{\lambda_{max} - n}{n - 1} = \frac{3{,}01 - 3}{3 - 1} = 0{,}01$$

(7)

Para obter a razão de consistência RC, é necessário efetuar a divisão entre o índice de consistência IC calculado anteriormente e um índice aleatório IR. O índice aleatório IR é calculado para matrizes quadradas de ordem n, pelo Laboratório Nacional de Oak Ridge. A tabela 8 define os valores de IR em função do número de critérios.

Tabela 8 - Valores de índice aleatório para matrizes quadradas de ordem n

1	**2**	**3**
0,00	0,00	0,58

Com o valor de IC calculado previamente e considerando o valor de índice como 0.90 é possível calcular o valor da razão de consistência:

$$RC = \frac{IC}{IR} = \frac{0{,}01}{0{,}58} = 0{,}02$$

(8)

Como o valor da razão de consistência é inferior a 0,1, é possível afirmar que os pesos dados aos critérios são confiáveis e consistentes.

Na fase seguinte, definem-se as matrizes de comparação paritária para cada critério, considerando as ferramentas selecionadas.

No que toca à facilidade de aprendizagem e utilização, os principais fatores de diferenciação entre os valores devem-se à já adoção do GitHub como sistema de controlo de versões dentro da organização e à familiaridade de alguns membros com a tecnologia *Jenkins*. Assim, a tabela 9 apresenta a matriz de comparação e a tabela 10 apresenta a matriz normalizada e a prioridade relativa da tecnologia.

Tabela 9 - Matriz de comparação do critério de facilidade de aprendizagem e utilização

Facilidade de aprendizagem e utilização	***Jenkins***	***GitHub Actions***	***TeamCity***
Jenkins	1	1/3	3
GitHub Actions	3	1	5
TeamCity	1/3	1/5	1
Soma	13/3	23/15	9

Tabela 10 - Matriz normalizada e prioridade relativa do critério de facilidade de aprendizagem e utilização

Facilidade de aprendizagem e utilização	***Jenkins***	***GitHub Actions***	***TeamCity***	**Prioridade Relativa**
Jenkins	3/13	1/3	1/3	0,2991
GitHub Actions	9/13	1	5/9	0,7492
TeamCity	1/13	1/5	1/9	0,1293

No que toca ao critério de funcionalidades e extensibilidade, todas as tecnologias possuem as funcionalidades necessárias no âmbito do projeto, no entanto, a extensibilidade da ferramenta *Jenkins* é superior ao *GitHub Actions* que é superior ao *TeamCity* (GitHub, 2023a; Jenkins, 2023; JetBrains, 2023).

As tabelas 11 e 12 apresentam a comparação e cálculos necessários relativamente ao critério de funcionalidades e extensibilidade.

Tabela 11 - Matriz de comparação do critério de funcionalidades e extensibilidade

Funcionalidades e extensibilidade	***Jenkins***	***GitHub Actions***	***TeamCity***
Jenkins	1	3	*5*
GitHub Actions	1/3	1	3
TeamCity	1/5	1/3	1
Soma	23/15	13/3	9

Tabela 12 - Matriz normalizada e prioridade relativa do critério de funcionalidades e extensibilidade

Funcionalidades e extensibilidade	***Jenkins***	***GitHub Actions***	***TeamCity***	**Prioridade Relativa**
Jenkins	15/23	9/13	5/9	0,6333
GitHub Actions	5/23	3/13	1/3	0,2605
TeamCity	3/23	1/13	1/9	0,1062

Relativamente ao custo das tecnologias selecionadas, o *Jenkins* é gratuito quando auto-hospedado tal como o *GitHub Actions* e o *TeamCity*, no entanto, o TeamCity apresenta algumas limitações (como um limite de 100 configurações de *build*).

É importante referir que o *GitHub Actions* apresenta um plano gratuito que permite utilizar os seus servidores na *cloud* durante 2000 minutos por mês.

Posto isto, as tabelas 13 e 14 fazem os mesmos cálculos que as tabelas anteriores, mas para o critério de custo.

Tabela 13 - Matriz de comparação do critério de custo

Custo	***Jenkins***	***GitHub Actions***	***TeamCity***
Jenkins	1	1/3	3
GitHub Actions	3	1	5
TeamCity	1/3	1/5	1
Soma	13/3	23/15	9

Tabela 14 - Matriz normalizada e prioridade relativa do critério de custo

Custo	***Jenkins***	***GitHub Actions***	***TeamCity***	**Prioridade Relativa**
Jenkins	3/13	5/23	1/3	*0,2605*
GitHub Actions	9/13	15/23	5/9	*0,6333*
TeamCity	1/13	3/23	1/9	*0,1062*

Após a obtenção da prioridade relativa para cada critério, tendo em conta as alternativas selecionadas, resta obter a prioridade composta para as alternativas. Para tal é multiplicada a matriz prioridade, onde cada linha corresponde a uma tecnologia e cada coluna a um critério, pelo peso dos critérios obtido previamente (Nicola, 2023a). Assim, a prioridade composta é obtida da seguinte forma:

$$\begin{bmatrix} 0{,}30 & 0{,}63 & 0{,}26 \\ 0{,}75 & 0{,}26 & 0{,}63 \\ 0{,}13 & 0{,}11 & 0{,}11 \end{bmatrix} \times \begin{bmatrix} 0{,}30 \\ 0{,}16 \\ 0{,}54 \end{bmatrix} = \begin{bmatrix} 0{,}33 \\ 0{,}61 \\ 0{,}12 \end{bmatrix} \quad (9)$$

Após a obtenção da prioridade composta para as alternativas, é possível concluir que o *GitHub Actions* (0,61) é a melhor alternativa entre as selecionadas, apresentando-se superior ao *Jenkins* (0,33) por um peso de 0,28.

É importante referir que, o resultado desta análise permite tornar a escolha da tecnologia mais fácil, no entanto, o resultado desta análise pode ser rejeitado, dependendo de considerações adicionais.

4.5 Sumário

A análise de valor conduzida neste capítulo permitiu identificar e analisar claramente a oportunidade existente dentro da organização, de automatizar a entrega de *software* garantindo a sua qualidade. Através desta oportunidade, foi realizada a proposta de valor, onde foram analisados os principais benefícios e custos associados com a adoção da solução.

Também foram utilizadas as técnicas FAST e AHP com o objetivo de auxiliar no processo de desenvolvimento da solução.

5 Análise e *Design*

O principal objetivo deste capítulo consiste em recolher os atributos de qualidade necessários para desenvolver a solução e, com base nestes, desenhar uma solução que seja capaz de cumprir com os atributos recolhidos.

Com o objetivo de cumprir com este objetivo, este capítulo começa por identificar os principais requisitos funcionais e não funcionais necessários para solucionar o problema em questão.

Finalmente, é detalhado todo o processo de desenho da solução, descrevendo as decisões tomadas e contrapondo estas com possíveis alternativas.

5.1 Análise

Como referido previamente, nesta secção são identificados os principais requisitos funcionais e não funcionais necessários para o correto funcionamento do sistema de acordo com os objetivos pretendidos.

De forma a recolher os requisitos funcionais, a estratégia adotada foi a identificação dos principais casos de uso.

Com o objetivo de identificar os requisitos não funcionais, o modelo escolhido para este processo é o FURPS+.

5.1.1 Requisitos Funcionais

Com o objetivo de cumprir com as expetativas dos utilizadores em relação à solução, é necessário compreender de que forma é que estes pretendem interagir com o sistema.

Deste modo, surge a necessidade de recolher os requisitos funcionais que correspondem a uma característica ou funcionalidade que um sistema deve ser capaz de fazer.

Assim, a figura 18 apresenta tanto os casos de uso identificados como os principais atores.

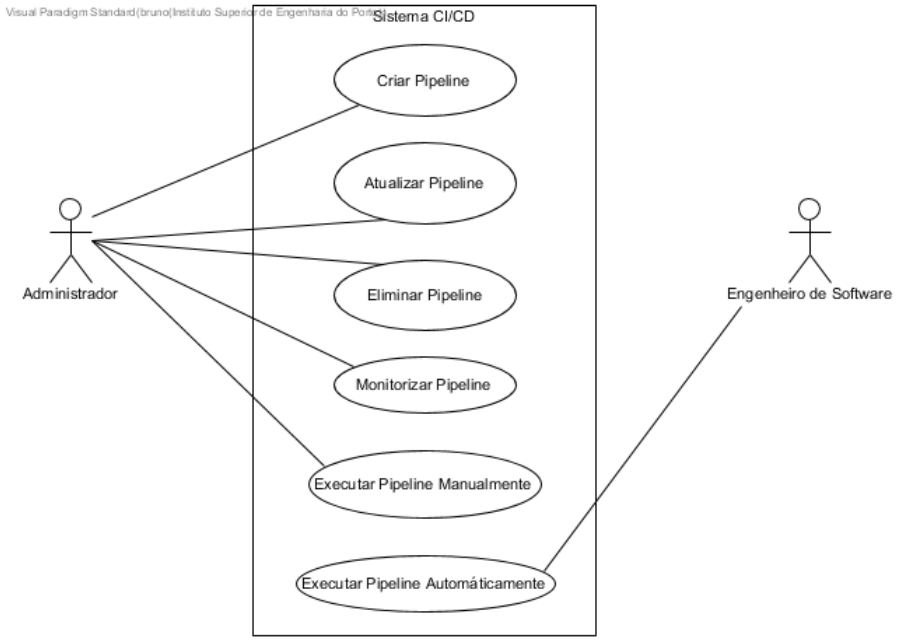

Figura 18 – Diagrama de casos de uso

Analisando em maior detalhe a figura 18, é possível perceber dois atores que farão um uso distinto da aplicação.

Por um lado, o administrador pretende:

- Criar, atualizar e eliminar uma *pipeline*, de forma a poder gerir o sistema CI/CD como pretender, possibilitando inúmeras possibilidades de testar e implantar soluções.
- Executar uma *pipeline* manualmente, para o administrador poder testar e implantar a solução quando pretender.

Por outro lado, o engenheiro de *software* pretende:

- Executar automaticamente a *pipeline*, isto é, que as suas ações despoletem a execução da *pipeline* (como por exemplo, um *commit* para um ramo em específico), com o objetivo de garantir que cada alteração efetuada mantém o *software* pronto a ser entregue.

5.1.2 Requisitos Não Funcionais

O sucesso de uma solução de *software* depende tanto dos requisitos funcionais como dos requisitos não funcionais.

Para muitos autores, a definição do que são requisitos não funcionais não é clara, existindo múltiplas definições deste conceito (Cleland-Huang, 2007). Estes podem ser definidos como "requisitos que especificam propriedades do sistema como restrições ambientais e de implantação" (Jacobson, Booch and Rumbaugh, 1999), por outras palavras, "um requisito que especifica restrições físicas num requisito funcional" (Jacobson, Booch and Rumbaugh, 1999).

Com o objetivo de recolher os requisitos não funcionais, será utilizado o modelo de qualidade FURPS+, que consiste num sistema de classificação que tem em conta as seguintes características: funcionalidade, usabilidade, confiabilidade, desempenho, suporte e restrições adicionais (+) como restrições de desenho, restrições de implementação, restrições de interface e restrições físicas (Janošcová, 2012).

Como a finalidade consiste em recolher os requisitos não funcionais, serão analisadas apenas as características relativas a estes, no caso, todas excetuando a característica funcionalidade (URPS+) (Eeles, 2004; Janošcová, 2012);

5.1.2.1 Usabilidade

A usabilidade consiste essencialmente em fatores humanos, estética, consistência na utilização da interface e documentação (Rafa E. Al-Qutaish, 2010).

A interface disponibilizada deve ser simples, consistente e intuitiva, permitindo assim que a maioria dos utilizadores consigam ser capazes de efetuar uma determinada tarefa num curto espaço de tempo.

5.1.2.2 Confiabilidade

Este tipo de requisitos inclui características como: frequência e impacto de falhas no sistema, disponibilidade, tolerância a falhas, precisão e tempo de média entre falhas (MTBF) (Rafa E. Al-Qutaish, 2010).

No âmbito desta dissertação, é de importância máxima garantir precisão, ou seja, a solução deverá ser capaz de garantir que apenas *software* que possua a qualidade desejada é entregue, resultando numa maior taxa de sucesso de implantações assim como numa redução de tarefas de correções.

Aliado ao referido, a solução deve ser estável, com um baixo número de erros e um elevado MTBF.

5.1.2.3 Desempenho

O desempenho refere-se a características que imponham condições em requisitos funcionais como por exemplo: velocidade, eficiência, tempo de resposta, utilização de recursos, entre outros (Rafa E. Al-Qutaish, 2010).

De forma a controlar e otimizar a utilização de recursos de forma a economizar custos, a *pipeline* deve ser eficiente e a sua execução não deve ser demasiado prolongada, isto é, o seu tempo de execução não deve ultrapassar 1 hora.

5.1.2.4 Suporte

Esta característica é composta por: testabilidade, manutenção, extensibilidade, compatibilidade, entre outros (Rafa E. Al-Qutaish, 2010).

Posto isto, é crucial que a solução desenvolvida seja de fácil manutenção e que o seu funcionamento possa ser estendido, permitindo a adição de novas funcionalidades, novas tecnologias e novos ambientes.

5.1.2.5 Restrições Adicionais (+)

Como já havia sido referido, restrições adicionais consistem em restrições de desenho, implementação, interface e físicas.

No que toca às restrições de implementação, estas especificam como é construído o sistema (Eeles, 2004).

Antes de serem abordadas as restrições de implementação, é necessário abordar um ponto que impacta este tipo de restrições: a aplicação que é testada e entregue, o *ETL Core* não apresenta interface gráfica.

Posto isto, as tecnologias que devem ser utilizadas consistem: no *GitHub Actions* como ferramenta de CI/CD e no *Jest* como ferramenta de testes.

Relativamente à escolha do *GitHub Actions*, esta encontra-se justificada na secção 4.4 “Método de Análise Hierárquica”.

No que toca à escolha do *Jest* esta deve-se essencialmente a duas razões distintas:

1. Permite suportar todos os tipos de testes pretendidos;
2. É extremamente competitiva quando comparada com os seus concorrentes (*Mocha*), no entanto, a sua curva de aprendizagem é menor.

Além da adoção das tecnologias referidas, é necessário que a solução seja:

- Compatível com *Docker*;
- Executada em ambiente composto por serviços AWS.

5.2 Design

Após serem recolhidos todos os atributos de qualidade, é necessário deliberar sobre a arquitetura do sistema.

Assim, esta secção começará por apresentar o processo atual de entrega de *software* existente.

De seguida, será apresentada a arquitetura proposta, contrapondo esta com uma alternativa arquitetural.

5.2.1 Processo Atual

A adoção de uma *pipeline* de implantação não consiste apenas na utilização de uma ferramenta nova, podendo implicar alterações profundas em todo o processo de lançamento de uma versão (processo de *release*).

Assim, o sucesso da adoção de uma *pipeline* de implantação não consiste apenas na sua qualidade intrínseca, mas também no quão bem esta se encaixa no processo atual de lançamento de uma versão de forma a permitir a sua rápida adoção por parte das equipas de engenharia de *software*, bem como cumprir o objetivo que é entregar o *software* que o utilizador conhece sem alterar as suas características.

Deste modo, não só é importante a recolha de todos os atributos de qualidade, como também compreender o processo atual adotado, que se encontra detalhado na figura 19.

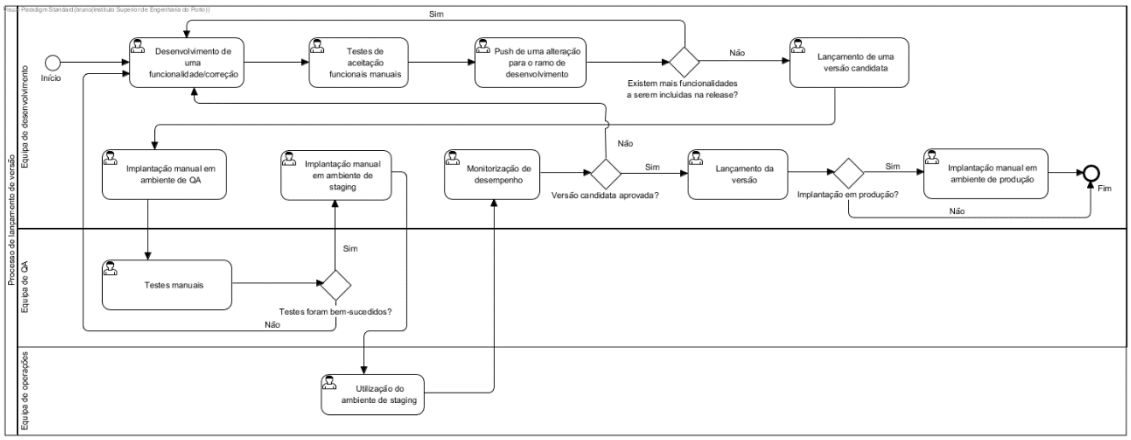

Figura 19 – Diagrama BPMN do processo de lançamento de versão atual

Analisando o diagrama presente na figura 19, é possível perceber que o processo de lançamento de versão atual é o seguinte:

1. São desenvolvidas e testadas manualmente funcionalidades, até a equipa considerar que o conjunto de funcionalidades desenvolvido deve constituir uma versão;
2. É então lançada uma versão candidata;
3. Esta versão candidata é implantada manualmente no servidor de QA;
4. A equipa de QA realiza testes funcionais de aceitação manuais e testes exploratórios manuais;
5. Caso os testes manuais sejam bem-sucedidos, é feita a implantação manual da versão candidata em *staging*. Caso os testes falhem, é necessário corrigir a versão candidata, sendo necessário recomeçar o processo.
6. Posteriormente, é monitorizado o desempenho da versão candidata em *staging*, tendo como objetivo perceber se a aplicação não apresenta nenhum problema de desempenho nem sobrecarrega os recursos que a suportam;
7. Caso a versão candidata seja aprovada, esta é promovida a versão, sendo publicados os seus artefactos no repositório. Caso a versão candidata seja rejeitada, esta é iterada novamente, sendo recomeçado o processo;
8. Finalmente, é dada a possibilidade de implantar a aplicação em produção, sendo este um processo manual.

5.2.2 Arquitetura

Após ser conhecido o processo atual de lançamento de versão, é possível perceber de que forma a nova arquitetura vai, não só alterar o fluxo, como melhorá-lo em pontos chave.

O novo processo de lançamento de versão com a *pipeline* de implantação encontra-se descrito no diagrama de atividades presente na figura 20.

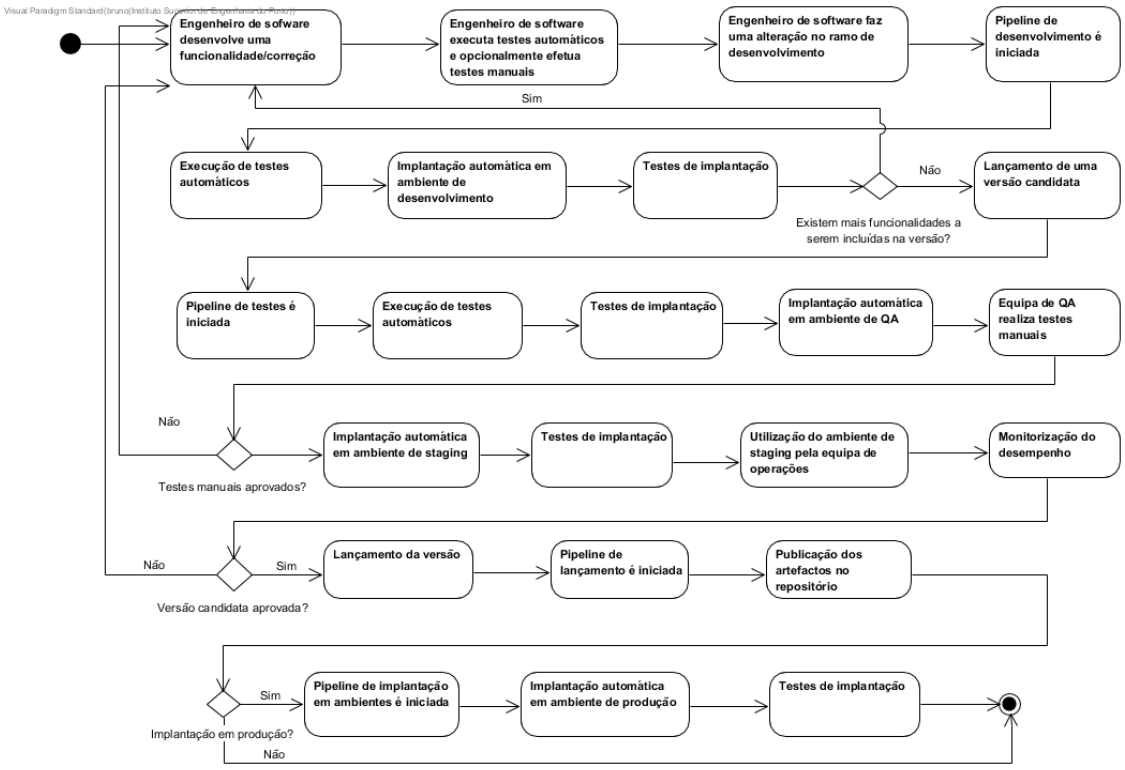

Figura 20 - Diagrama de atividades do processo de *release* novo

Analisando o diagrama de atividades apresentado na figura 20, é possível perceber que o novo processo de lançamento de versão segue um fluxo similar ao processo de lançamento de versão atual, apresentado na figura 19.

As principais diferenças entre os processos apresentados anteriormente consistem na adoção da *pipeline* de implantação e de uma estratégia de testes automáticos, permitindo automatizar todas as implantações e parte dos testes efetuados.

Detalhando o processo apresentado na figura 20, o novo plano de lançamento de versão funciona da seguinte forma:

1. Sempre que um engenheiro de *software* da equipa de desenvolvimento submete uma alteração para o ramo de desenvolvimento, é iniciada a *pipeline* de desenvolvimento, responsável por executar diferentes tipos de testes automáticos e implantar a aplicação no ambiente de desenvolvimento;
2. Este processo é repetido várias vezes até serem concluídas todas as funcionalidades e correções que deverão fechar um ciclo, sendo então lançada uma versão candidata;
3. O lançamento da versão candidata despoleta a execução da *pipeline* de testes, que começa por executar a estratégia de testes automáticos e posteriormente implanta a versão candidata no servidor de QA;
4. Nesta fase, a equipa de QA realiza diferentes tipos de testes manuais como testes funcionais de aceitação e testes exploratórios;
5. Caso os testes realizados pela equipa de QA sejam bem-sucedidos, é feita a implantação da versão candidata em *staging*. Casos os testes falhem, é necessário corrigir a versão candidata, sendo necessário recomeçar o processo.
6. Após a implantação em *staging*, é feita a monitorização do desempenho da solução.
7. De seguida, caso a versão candidata seja aprovada, esta é promovida a versão, despoletando a *pipeline* de lançamento. Caso a versão seja rejeitada, esta é iterada novamente;
8. Assumindo que a versão candidata foi promovida a versão, a *pipeline* de lançamento publica os artefactos no repositório;
9. Finalmente, caso se pretenda implantar a aplicação em um ambiente de produção, é possível fazer *trigger* manual da *pipeline* de implantação em ambientes, que fará a implantação automática no ambiente de produção desejado.

No novo processo de lançamento de versão, é importante realçar uma característica importante: a *pipeline* de implantação, abordada extensivamente ao longo deste documento, encontra-se decomposta em 4 *pipelines* distintas: *pipeline* de desenvolvimento, *pipeline* de testes, *pipeline* de lançamento e *pipeline* de implantação em ambientes.

Assim, no âmbito desta arquitetura, o conceito de *pipeline* de implantação refere-se às quatro *pipelines* referidas previamente.

Esta divisão em múltiplas *pipelines* mais pequenas, permite não só a sua utilização de forma individual consoante o estágio no processo de entrega da solução, como também uma adoção mais fácil tendo em conta o processo de lançamento de versões vigente.

Após ser compreendido o novo processo de lançamento de versões, existem duas questões importantes que necessitam de esclarecimento:

- Como é que cada uma das *pipelines*, que compõem a *pipeline* de implantação, são constituídas?
- Qual é a estratégia de testes do novo processo de lançamento de versão?

Com o intuito de abordar as questões anteriores, a figura 21 apresenta um diagrama BPMN que descreve a *pipeline* de implantação.

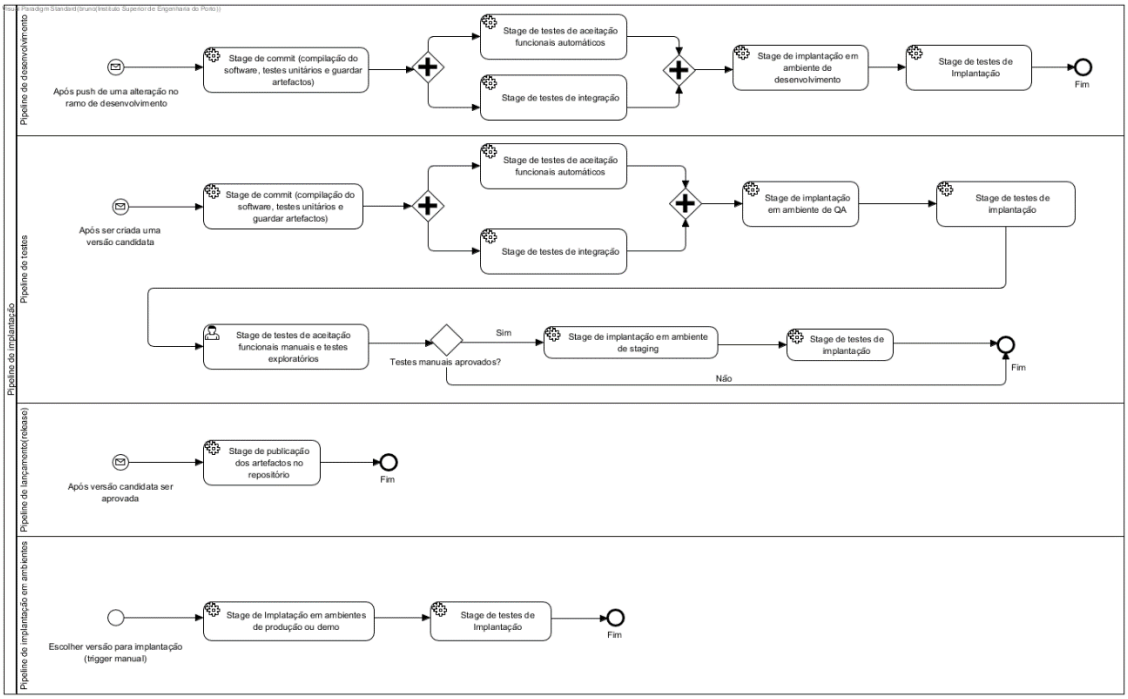

Figura 21 - Diagrama BPMN da *pipeline* de implantação

Após a apresentação da figura 21 onde é possível ver cada *pipeline* e respetivo fluxo de execução, é necessário descrever quais os passos que a compõem e o seu propósito.

A *pipeline* de desenvolvimento é iniciada sempre que é submetida uma alteração para o ramo de desenvolvimento. Esta é constituída pelos seguintes *stages:*

- *Stage* de *commit*: onde é compilada a aplicação, são executados os testes unitários e são guardados os artefactos provenientes da compilação. Este além de consistir no primeiro passo para garantir a qualidade da aplicação, gera e guarda os artefactos que serão utilizados nos restantes estágios, evitando múltiplas compilações;

- *Stage* de testes de integração: que testa a interação entre as diversas unidades lógicas da aplicação;
- *Stage* de testes de aceitação funcionais automáticos: responsável por garantir que os critérios de aceitação de cada funcionalidade são cumpridos;
- *Stage* de implantação em ambiente de desenvolvimento: responsável por testar se a alteração realizada pode ser implantada e que a versão consegue ser atualizada corretamente.
- *Stage* de testes de implantação: que testa se a aplicação foi implantada corretamente.

A *pipeline* de testes é acionada sempre que é lançada uma versão candidata e é constituída pelos seguintes *stages*:

- Todos os *stages* de teste e o *stage* de *commit* utilizados na *pipeline* de desenvolvimento. Embora não seja necessário repetir estes *stages*, uma vez que todas as alterações no ramo de desenvolvimento passam por estes, esta redundância permite garantir que a versão candidata passou efetivamente por todos os testes.
- *Stage* de implantação em ambiente de QA, responsável por implantar a aplicação no ambiente de QA com as configurações pretendidas.
- *Stage* de testes de implantação, que testa se a implantação foi feita corretamente.
- *Stage* de testes de aceitação funcionais manuais e testes exploratórios, responsáveis por validar manualmente se os critérios de aceitação das funcionalidades foram cumpridos, além de explorar potenciais *bugs* na aplicação;
- *Stage* de implantação em *staging*, permitindo não só monitorizar o desempenho da versão candidata, como também disponibilizar esta à equipa de operações.

A *pipeline* de lançamento é iniciada sempre que uma versão candidata é aprovada. Esta publica os artefactos no repositório, permitindo que esta versão possa ser implantada em produção.

A *pipeline* de implantação em ambientes é acionada manualmente por um utilizador com as permissões adequadas. Esta é constituída por apenas 2 *stages*: *stage* de implantação em ambiente de produção (ou demo) e *stage* de testes de implantação.

Posto isto, existem três pontos da arquitetura que necessitam de ser abordados detalhadamente:

- Linhas orientadoras de desenvolvimento de testes automáticos;

- A estratégia de testes de aceitação funcionais automáticos;
- A estratégia de implantação (independentemente do ambiente).

5.2.2.1 Linhas Orientadoras de Desenvolvimento de Testes Automáticos

Embora a estratégia de testes apresentada no *design* seja composta por múltiplos tipos de testes automáticos, existem certas linhas orientadoras que devem ser adotadas por qualquer um destes tipos de testes.

Além destes testes terem de seguir boas práticas e princípios de desenvolvimento de *software*, é importante que estes adotem o padrão *Arrange, Act, Assert* (AAA). De acordo com este padrão, os testes devem adotar a seguinte estrutura (Cheddadi, Motahhir and Ghzizal, 2022):

- Começar por inicializar todos os dados, estruturas de dados e pré-condições necessárias para a realização do test (*Arrange*).
- De seguida, deve ser chamada a função que se pretende testar com os parâmetros pretendidos (*Act*).
- Finalmente, deve ser comparado o resultado obtido com o resultado esperado (*Assert*).

5.2.2.2 Estratégia de Testes de Aceitação Funcionais Automáticos

Existem duas características importantes na aplicação que é testada, o *ETL Core*, que impactam o desenho da estratégia deste tipo de testes: a ausência de uma interface gráfica e o facto desta aplicação ser baseada na geração de novos dados através de informação já existente.

Outro fator que influencia o desenho deste tipo de testes consiste no fato destes estarem intimamente dependentes do utilizador final, uma vez que este define os critérios de aceitação para uma determinada funcionalidade.

Assim, existe uma vantagem clara em tentar atingir o caso “perfeito” de desenvolvimento de testes de aceitação funcionais, onde estes são desenvolvidos pelo utilizador final.

Posto isto, resta perceber o que é um teste de aceitação funcional no âmbito de *ETL Core*.

Uma vez que o *ETL Core* consiste essencialmente numa *pipeline* de dados, que dado um conjunto de dados de entrada, efetua um conjunto de operações (também chamadas por estágios de processamento ou *data sources*) de forma a produzir um conjunto de indicadores, um teste de aceitação funcional deverá ser capaz de testar todo este fluxo.

Assim, um teste de aceitação funcional consiste numa *pipeline*, composta por um ou mais *data sources* (operações), que dado um determinado conjunto de dados de entrada, produz um determinado conjunto de dados de saída válidos.

Com o objetivo de efetuar testes como descrito anteriormente, a estratégia adotada consiste na utilização de dados de entrada e configurações conhecidos para obter dados de saída conhecidos, evitando, no contexto do *ETL Core*, o problema *oracle*.

Assim, combinando a estratégia mencionada anteriormente com a visão de colocar o utilizador como implementador deste tipo de testes, esta estratégia de testes deve seguir o processo mostrado na figura 22.

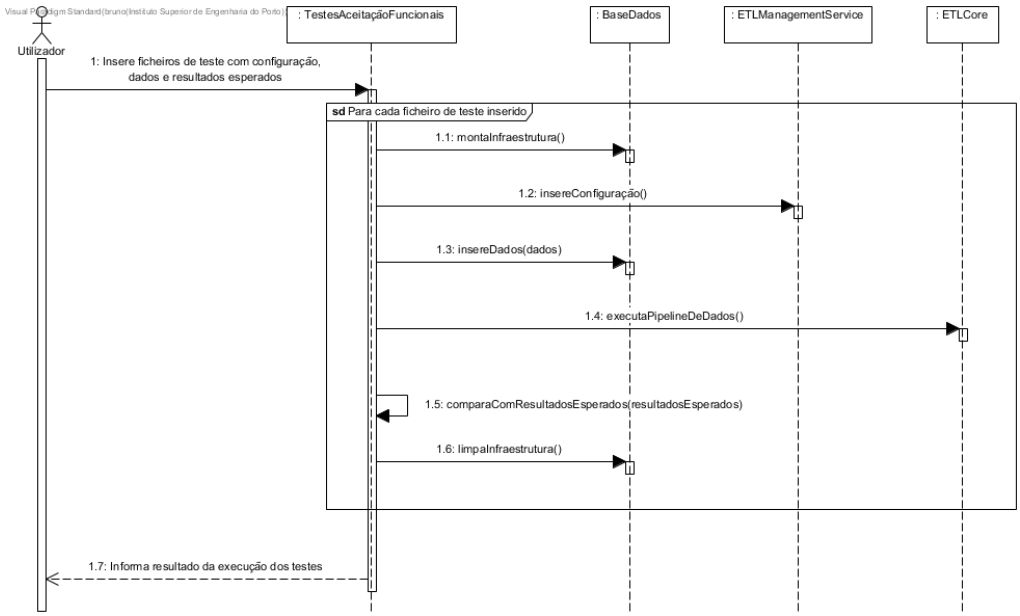

Figura 22 – Diagrama de sequência "Testes de Aceitação Funcionais"

Analisando o diagrama de sequência da figura 22, a interação entre o utilizador e o sistema é feita através de ficheiros, onde cada ficheiro corresponde a um teste de aceitação funcional.

Cada ficheiro de teste é constituído por: configurações referentes à *pipeline* de dados que será executada, dados que alimentarão a *pipeline* de dados e resultados esperados.

Com a informação presente neste ficheiro, os testes de aceitação funcionais têm que:

1. Montar a infraestrutura necessária para executar estes testes;
2. Inserir a configuração que referente à *pipeline* de dados. Para tal, são feitos pedidos a um serviço externo capaz de criar esta configuração, chamado de *ETLManagementService*;

3. De seguida, os dados que alimentam a *pipeline* de dados são inseridos na base de dados;
4. Estando o ambiente de testes montado, é então executada a *pipeline* de dados recorrendo ao *ETL* core para tal;
5. Após a execução da *pipeline* de dados, são comparados os resultados obtidos com os resultados esperados;
6. Finalmente, é limpa toda a infraestrutura.

5.2.2.3 Estratégia de Implantação

Embora a *pipeline* de implantação proposta faça diversas implantações, ainda é necessário especificar em que consiste a estratégia de implantação.

Esta estratégia de implantação tem um objetivo: garantir que a implantação é feita corretamente e sem *downtime*.

Desta forma, a estratégia de implantação desenhada é apresentada na figura 23.

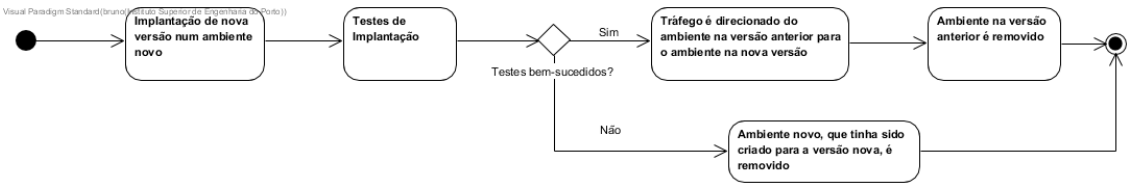

Figura 23 – Diagrama de atividades "Estratégia de Implantação"

Observando o processo de implantação presente no diagrama de atividades da figura 23, este funciona da seguinte forma:

- A versão atual encontra-se em funcionamento num determinado ambiente, sendo este ambiente uma máquina, um contentor ou algo semelhante;
- É feita a implantação da nova versão num ambiente novo;
- A nova versão é validada através de testes de implantação;
- Caso os testes sejam bem-sucedidos, o tráfego é redirecionado do ambiente que se encontra na versão anterior para o novo ambiente na nova versão. Caso os testes falhem, o ambiente que havia sido criado é removido.
- Assumindo que os testes foram bem-sucedidos, o ambiente que se encontra na versão anterior é removido.

Com este processo de implantação é possível garantir, dentro do possível, que a implantação foi feita corretamente e que não ocorreu nenhum *downtime*.

5.2.3 Alternativa Arquitetural

Nesta secção será apresentada uma alternativa arquitetural em relação ao modelo apresentado na secção anterior.

Posteriormente, a alternativa arquitetural formulada será contraposta com a arquitetura escolhida, expondo os pontos positivos e negativos desta, e explicando a razão pela qual esta não foi implementada.

A *pipeline* de implantação alternativa encontra-se representada no diagrama BPMN da figura 24.

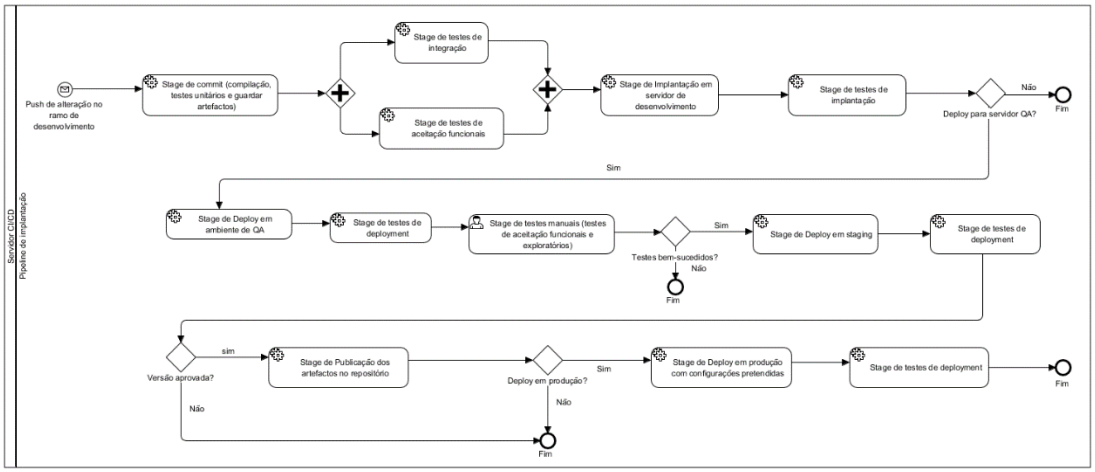

Figura 24 - Diagrama BPMN da *pipeline* de implantação alternativa

Examinando o diagrama da figura 24, é possível perceber que todos os *stages* apresentados coincidem com os *stages* apresentados na figura 21, que corresponde à arquitetura da *pipeline* de implantação escolhida.

Essencialmente, as principais diferenças entre a arquitetura escolhida e a alternativa arquitetural residem:

- No fato de a arquitetura escolhida se encontrar dividida em 4 *pipelines* distintas;
- Na arquitetura alternativa permitir que qualquer alteração possa percorrer todo o fluxo terminando em produção, enquanto a arquitetura escolhida força o lançamento de uma versão candidata.

Embora ambas as arquiteturas sejam de fato bastante similares e permitem atingir um resultado parecido, a arquitetura escolhida apresenta as seguintes vantagens em relação à arquitetura alternativa:

- A sua adoção é mais fácil e simples, uma vez que arquitetura escolhida encaixa no processo de lançamento de versões atual sem quaisquer obstáculos;
- Maior liberdade tecnológica, uma vez que, num cenário ideal, a arquitetura alternativa sugere a necessidade da tecnologia utilizada permitir a escolha de que compilação é implantada em ambiente de QA e *staging*. Diversas tecnologias não apresentam esta capacidade nativamente (como é o caso do *GitHub Actions*);
- Apenas versões candidatas podem ser implantadas no servidor de QA e consequentemente em *staging* e produção. Deste modo, é mitigado o risco de versões da aplicação não pretendidas serem implantadas, desperdiçando recursos desnecessariamente.

No entanto, a arquitetura escolhida apresenta uma desvantagem em relação à arquitetura alternativa: é possível afirmar que a arquitetura alternativa cumpre melhor o conceito de entrega contínua e definitivamente se encontra mais perto de cumprir com o conceito de implantação contínua.

5.3 Sumário

Com base nos requisitos funcionais e não funcionais recolhidos no processo de análise, no processo de *design* é desenhada uma arquitetura teoricamente capaz de cumprir com estes.

A arquitetura consiste numa *pipeline* de implantação altamente modular, dividida em quatro *pipelines*, constituída uma estratégia de teste que incorpora tanto testes automáticos como testes manuais, além de efetuar múltiplas implantações, em ambientes distintos com objetivos diferentes.

Esta arquitetura é ainda contraposta com uma alternativa arquitetural, ilustrando as principais vantagens e desvantagens da arquitetura proposta.

6 Implementação

Tendo sido realizada a recolha dos requisitos, assim como o desenho da arquitetura do sistema e a escolha das tecnologias, é necessário abordar os detalhes de implementação da solução.

Devido à natureza da solução desenvolvida, esta encontra-se essencialmente dividida em 3 segmentos distintos:

- Estratégia de testes;
- Infraestrutura;
- *Pipeline* de implantação.

Desta forma, este capítulo visa abordar em detalhe cada um dos segmentos referidos anteriormente, em que serão justificadas algumas das decisões tomadas a nível de implementação.

6.1 Estratégia de Testes

Na secção 5.2 "*Design*", foi referido que a estratégia de testes escolhida consiste em quatro tipos de testes distintos: testes unitários, testes de integração, testes de aceitação funcionais e testes de implantação. Todos estes testes foram implementados utilizando a *framework Jest*.

Deste modo, todos estes testes encontram-se no mesmo projeto da aplicação que é testada, como é possível perceber pela figura 25, que apresenta a estrutura de ficheiros deste projeto.

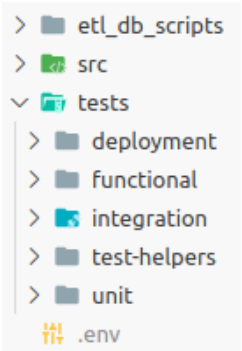

Figura 25 – Estrutura de ficheiros do ETL Core

Dada a estrutura de ficheiros apresentada na figura 25, é importante destacar que o trabalho realizado se centra essencialmente em dois pontos distintos:

- O desenvolvimento, predominantemente realizado na pasta "*tests*", enquanto o restante consistiu em pequenas alterações no código fonte do projeto com o intuito de tornar o código mais testável, localizado na pasta "*src*";
- A configuração, efetuada nos ficheiros "*package.json*" e "*jest.config.js*".

Uma vez que todos os testes implementados utilizam a mesma *framework*, a configuração da mesma é comum para todos os testes e consistiu essencialmente em:

- Adicionar um script de execução de testes ao *"package.json*", uma vez que este ficheiro é responsável por guardar tanto *metadata* quanto configurações do projeto;
- Configurar o ficheiro *jest.config.js*, que permite configurar diversos parâmetros relacionados com a execução dos testes.

No que toca ao *script* de execução de testes, este essencialmente chama a *framework Jest*, que executa os testes implementados. É possível chamar este *script* com parâmetros adicionais, disponibilizados pela própria *framework*, que conferem uma elevada versatilidade na execução dos testes, como por exemplo, executar apenas um conjunto de testes.

Relativamente à configuração do ficheiro "*jest.config.js*", são especificados diversos parâmetros de configuração como: o ambiente de testes, ficheiros de configuração, testes a ignorar, entre outros. O excerto de código 1 apresenta a configuração utilizada neste projeto.

```
module.exports = {
  preset: 'ts-jest',
  testEnvironment: 'node',
  transformIgnorePatterns: ['node_modules/quick-lru'],
  setupFiles: ["dotenv/config"],
  testPathIgnorePatterns: [".d.ts", ".js"]
};
```

Excerto de código 1 – Configuração do Jest

Analisando os pontos chaves da configuração, é possível perceber:

- A utilização da configuração do parâmetro "*preset*" para um valor que permita o suporte de *typescript* para o *Jest*, assim como a utilização do ambiente de testes de *node*, adaptando o *Jest* à aplicação que é desenvolvida em *node.js* com *typescript*;

- A possibilidade de especificação de variáveis de ambiente em ficheiros "*.env*", através da configuração do campo "*setupFiles*";
- Que são ignorados os ficheiros de testes *javascript*, resultantes da conversão (*transpiling*) de código *typescript* em *javascript*, através da configuração do parâmetro "*testPathIgnorePatterns*".

6.1.1 Testes Unitários

A atual estratégia de testes adotada pela *proGrow S.A.* é maioritariamente manual, praticamente não possuindo nenhum teste automático. Assim, a implementação deste tipo de testes tinha dois objetivos distintos:

- Testar as principais funcionalidades do *ETL Core*;
- Servir como base para implementação de futuros testes unitários.

Para atingir os objetivos referidos anteriormente, foram tomadas duas importantes decisões de implementação:

- Priorizar o teste de funções com acesso público em vez de funções com acesso privado;
- São priorizados testes que não recorrem a dados de entrada aleatórios, com o objetivo de manter o carácter determinístico deste tipo de testes;

Posto isto, o excerto de código 2 apresenta um exemplo de um teste unitário feito numa das principais funções. É importante realçar que os restantes testes unitários são análogos a este.

```
test('Check all variables are properly replaced', () => {
                const                expectedSelectQuery                =
`SELECT"start","end","value"from"test_schema"."test1"where"start"<'2023-07-
03T19:48:00.000Z'::timestamp                                AND"end">'2023-07-
03T18:48:00.000Z'::timestamp`;
        const       customParameters:       CustomDataTypeParameters       =
getDefaultCustomParameters(defaults.query);

    const   selectQuery:   string   =   getSelectStatement(customParameters,
{ DB_SCHEMA: defaults.dbSchema, start: defaults.start, end: defaults.end });
  const minifiedSelectQuery = minify(selectQuery, { compress: true });

  expect(minifiedSelectQuery).toBe(expectedSelectQuery);
});
```

Excerto de código 2 - Teste unitário geração de *select custom*

Contextualizando o teste unitário apresentado no excerto de código 5, este teste pretende testar uma função que, dado um conjunto de parâmetros, constrói o respetivo comando SQL *select*.

Analisando o teste desenvolvido, é evidente a adoção do padrão AAA:

- Primeiro são criadas as estruturas que serão utilizadas como dados de entrada (variável "*customParameters*") ou como resultado esperado (variável "*expectedSelectQuery*");
- Depois, é realiza a operação que se pretende testar (função "*getSelectStatement*");
- Finalmente é comparado o resultado obtido com o resultado esperado (função *"expect"*).

O único ponto que não fica transparente no excerto de código 6 é a utilização de um objeto chamado "*defaults*". Este objeto possui um conjunto de valores padrão que são partilhados entre diversos casos de teste, reduzindo a quantidade de código duplicado. Assim, não é necessário que testes diferentes instanciem as mesmas variáveis repetidamente.

O resultado da execução desta estratégia de testes mostra todos os ficheiros de testes executados, gerando um pequeno resumo sobre os testes executados e o seu tempo de execução. Em caso de falha, o *Jest* apresenta o teste que falhou assim como uma mensagem de erro apropriada. A figura 26 apresenta o resultado da execução destes testes unitários.

```
PASS tests/unit/datasources/status/status-data-type.test.ts
PASS tests/unit/datasources/modulo-data-type/hour-util.test.ts
PASS tests/unit/util/util.test.ts
PASS tests/unit/datasources/custom-data-type/custom-sql-generation.test.ts
PASS tests/unit/datasources/modulo-data-type/weekshift-util.test.ts
PASS tests/unit/timeframe/timeframe.test.ts
PASS tests/unit/datasources/modulo-data-type/monthshift-util.test.ts
PASS tests/unit/datasources/modulo-data-type/dayshift-util.test.ts
PASS tests/unit/datasources/modulo-data-type/yearshift-util.test.ts
PASS tests/unit/datasources/modulo-data-type/day-util.test.ts
PASS tests/unit/datasources/modulo-data-type/year-util.test.ts
PASS tests/unit/datasources/modulo-data-type/factoryweek-util.test.ts
PASS tests/unit/datasources/modulo-data-type/week-util.test.ts
PASS tests/unit/datasources/modulo-data-type/month-util.test.ts
PASS tests/unit/datasources/status/internal-status.test.ts
PASS tests/unit/sql-gen.test.ts

Test Suites: 16 passed, 16 total
Tests:       57 passed, 57 total
Snapshots:   0 total
Time:        2.999 s, estimated 5 s
```

Figura 26 – Resultado da execução da estratégia de testes unitários

6.1.2 Testes de Integração

Diversas das funções mais importantes do *ETL Core* possuem uma elevada dependência com serviços externos (mais especificamente com bases de dados), fazendo surgir a necessidade de utilizar outros tipos de testes além de testes unitários.

Aliado ao fator referido anteriormente, a API disponibilizada pelo *ETL Core* tem vindo a ganhar cada vez mais importância dentro da organização, contribuindo para a necessidade de se utilizarem testes de integração.

Assim, os dois fatores referidos anteriormente são os principais pontos que os testes de integração pretendem testar.

Devido à necessidade que estes testes têm de interagir com outros serviços, estes testes tendem a ser iniciados montando toda a infraestrutura necessária para a sua realização, sendo que após a sua execução, o processo deve ser revertido.

O excerto de código 3 apresenta um exemplo de testes de aceitação à API do *ETL Core*.

```
describe('Process Records integration test suite', () => {
  beforeAll(async () => {
    DatasourceRepository.add(simpleCustomDatasource.id,
simpleCustomDatasource);
    DatasourceRepository.add(customDatasource.id, customDatasource);
    supertestApp = supertest(app.getApp);
  });
  afterAll(async () => {
    DatasourceRepository.remove(simpleCustomDatasource.id);
    DatasourceRepository.remove(customDatasource.id);
  });
  test('Get simple custom datasource process query', async () => {
    const expectedQuery = `WITH generate_values AS ( SELECT distinct on (id)
* FROM (SELECT \"start\", \"end\", \"value\" from
\"test_db\".\"ds_qtd_produzida_of\") temp_data ), mark_to_delete AS( UPDATE
\"test_db\".\"simple_ds\" SET \"value\" = NULL INSERT INTO
\"test_db\".\"simple_ds\" SELECT * FROM generate_values `;
    const defaultProcessPayload =
generateDefaultProcessPayload(simpleCustomDatasource.id);

    const res = await supertestApp.post('/datasource').set({ 'Content-Type':
'application/json' }).send(defaultProcessPayload);

    expect(res.statusCode).toBe(200);
    expect(res.text).toBe(expectedQuery);
  });
```

Excerto de código 3 – Teste de aceitação à API do *ETL Core*

Contextualizando o excerto de código 3, o teste de aceitação apresentado testa um *endpoint* disponibilizado pela API do *ETL Core* que, dado o nome de um *data source* (estágio de

processamento) e um intervalo de tempo, retorna a respetiva consulta *SQL* executada por este (caso exista), parametrizada com o intervalo de tempo fornecido.

Como é possível observar no excerto de código 7, o teste apresentado segue o seguinte fluxo:

- Cria a infraestrutura necessária para realizar o respetivo teste de integração (na função "*beforeAll*");
- Gera o conteúdo que será enviado para a API (função "*generateDefaultProcessPayload*");
- Faz o pedido à API, enviando o conteúdo gerado previamente, utilizando a biblioteca *supertest* para tal, que permite facilitar todo o processo de testes de pedidos *HTTP*;
- Compara o estado e conteúdo do pedido, verificando se são iguais ao esperado (função "*expect*").

Tal como no caso dos testes unitários, a execução dos testes de integração gera um pequeno sumário sobre os testes executados e o seu tempo de execução, como ilustrado na figura 24.

Figura 27 -Resultados de execução da estratégia de testes de integração

6.1.3 Testes de Aceitação Funcionais

Tendo em conta o *design* sugerido na secção 5.3.2.2 "Estratégia de Testes de Aceitação Funcionais Automáticos", o fluxo implementado para a execução de testes de aceitação funcionais funciona da seguinte forma:

- É introduzido um ficheiro JSON parametrizado pelo utilizador contendo as configurações da *pipeline*, os dados de entrada e os resultados esperados.
- A ferramenta gera a *pipeline* através das configurações recebidas e processa os dados de entrada.
- Finalmente, são comparados os resultados obtidos com os resultados esperados.

Compreendido o fluxo implementado, antes de abordar o código, é importante percebermos a estrutura do ficheiro JSON que gera o respetivo teste de aceitação funcional. Para tal, o excerto de código 4 apresenta um pequeno exemplo da configuração deste ficheiro.

```
{
    "config": {
      "referenceDate": "2023-07-14 16:30:00.000"
    },
    "datasources": [{
        "id": "counter_ingest",
        "type": "INGEST-EXTERNAL"
        ...
     }],
    "seeds": {
      "counter_ingest": [{
          "ts": "2023-07-14 16:26:00.000",
          "val": 8
        }]},
    "expected": {
      "ds_counter_aggr": [{
          "value": 14,
          "start": "2023-07-14 16:25:00.000",
          "end": "2023-07-14 16:26:00.000"
        }]}
  }
```

Excerto de código 4 – Exemplo de configuração de ficheiro de entrada JSON

Analisando o excerto de código 4, fica claro que este ficheiro se encontra dividido em quatro configurações distintas:

- "*config*": que possui configurações transversais a todo o teste. Esta possui um parâmetro opcional chamado "*referenceDate*", que informa o sistema que esta data corresponde à data atual que é usada como referência para os testes. Assim, a diferença entre a data de referência e a data atual (no momento de execução do teste), será aplicada a todas as datas do caso de teste, permitindo efetuar testes em tempo real;
- A configuração "*datasources"*, que essencialmente possui a configuração da *pipeline* a ser testada. Este parâmetro consiste numa lista de *data sources* que, como já explicado, é referente a estágios de processamento que efetuam diversos tipos de operação como: agregações, transformações, entre outras;
- O parâmetro "*seeds"*, que consiste nos dados que serão utilizados como dados de entrada da *pipeline*;
- A configuração "*expected"*, que consiste num objeto composto pelos resultados esperados da execução do teste de aceitação funcional.

Compreendida a configuração de um caso de testes, resta compreender os detalhes da sua implementação. O excerto de código 5 apresenta a função de mais alto nível responsável pela geração de um caso de testes.

```
async function generateTestCase(testCase: TestCase, now: Date):
Promise<void> {
  await setUpDatasources(testCase.datasources);
  await insertData(testCase.config, testCase.seeds, now);

  const appLauncher = new AppLauncher();
  await FunctionUtils.delay(APP_DELAY);
  appLauncher.stop();

  await comparesWithExpected(testCase.config, testCase.expected, now);
}
```

Excerto de código 5 – Geração de teste funcional

Analisando o excerto de código 5, fica clara a adoção do padrão AAA também para este tipo de testes. É importante referir que, antes de ser executada esta função, é executado um *script* responsável por montar toda a infraestrutura necessária para estes testes.

Após a criação da infraestrutura necessária, é necessário criar a configuração da *pipeline*, feita pela chamada à função "*setUpDatasources*". Este processo é simples, apenas é obtida a configuração do ficheiro *JSON* e enviada para um serviço externo responsável por validar e criar todos os *data sources*.

De seguida, é feita a inserção dos dados de entrada encarregados de alimentar a *pipeline*. A lógica deste processo de inserção de dados encontra-se exposta no excerto de código 6.

```
async function insertData(config: Config, seedsInput: { [k: string]: any },
now: Date): Promise<void> {
    const seeds: Seed[] = await convertSeedInputListToSeedList(config,
seedsInput, now);
    await Promise.all(
      seeds.map(async (seed) => {
        await Promise.all(
          seed.datasourceRecords.map(async (datasourceRecord) => {
            await
datasourceRecordRep.insertDatasourceRecord(seed.datasourceId,
datasourceRecord, DB_SCHEMA);
        }))}),
    );
}
```

Excerto de código 6 – Inserção de dados

Observando o excerto de código 6, é possível perceber que o processo responsável por alimentar a *pipeline* divide-se em 2 passos:

1. A transformação dos dados configurados em registos prontos para serem inseridos. Este processo é composto por algumas transformações:
    a. As datas destes registos são alteradas consoante a data especificada como data de referência;
    b. São verificadas a existência de funções nos valores configurados, caso existam, estas são executadas.
2. Os valores são inseridos na respetiva base de dados, utilizando o repositório "*datasourceRecordRep*" para tal.

Nos dois passos referidos previamente, é necessário esclarecer em que consiste a execução de funções na transformação dos dados de entrada.

Devido à necessidade de transformar alguns valores (a título de exemplo, converter uma data para o início de uma hora), é disponibilizado um conjunto finito de funções que podem ser chamadas pelo utilizador para atingir esta finalidade. O excerto de código 7 mostra uma pequena parte de como é feita esta implementação.

```
public static getFunctionService(functionName: string):
FunctionService<unknown> {
    switch (functionName.toLowerCase()) {
      case 'start_hour':
        return new StartHourFunctionService();
      case 'end_hour':
        return new EndHourFunctionService();
      case 'start_shift':
        return new StartShiftFunctionService();
      case 'end_shift':
        return new EndShiftFunctionService();
      case 'start_day':
        return new StartDayFunctionService();
      case 'end_day':
        return new EndDayFunctionService();
      default:
        throw new Error(`Function ${functionName} not implemented yet`);
    }
```

Excerto de código 7 – *Strategy* de funções

Com o objetivo não só de disponibilizar um conjunto de diferentes funções ao utilizador, como também facilitar todo o processo de adição de novas funções, foram adotados os padrões *factory*, *singleton* e *strategy* como mostrado no excerto de código 7.

Assim, consoante o nome da função inserida, é instanciado um serviço capaz de traduzi-la, permitindo abstrair o código da lógica de cada uma das funções.

Após a configuração e inserção de dados, é então executada a *pipeline*. Esta execução é feita através do objeto *AppLauncher*, como apresentado no excerto de código 8.

Este *AppLauncher* lança um processo concorrente que essencialmente executa o *ETL Core* que, por sua vez, executa a *pipeline* configurada previamente.

Após o término da execução da *pipeline*, é feita a comparação de resultados através da função *comparesWithExpected* cuja implementação encontra-se descrita no excerto de código 8.

```
async function comparesWithExpected(config: Config, expectedInput: { [k:
string]: any }, now: Date): Promise<void> {
    const expected: SeedResult[] = await
convertExpectedInputToSeedResultList(config, expectedInput, now);
    let result = true;
    for (const seedResult of expected) {
      const resultRecords: DatasourceRecord[] = await
datasourceRecordRep.getAllDataFromDatasource(seedResult.datasourceId);
      result = seedResult.equalsRecords(resultRecords);
      if (result === false) break;
    }
    expect(result).toBeTruthy();
  }
```

Excerto de código 8 – Comparação de resultados

Como exposto no excerto de código 8, esta função começa por ler e transformar todos os resultados esperados, sendo este processo análogo ao presente no excerto de código 6.

Finalmente, é feita a comparação do resultado obtido com os valores esperados. Para fazer esta comparação, são iterados todos os valores esperados e verificados se estes existem, como configurados, nos resultados obtidos.

Tal como nos restantes testes, a execução destes gera um pequeno sumário, como apresentado na figura 28, onde são apresentadas métricas como os testes que passaram, falharam e respetivos tempos de execução.

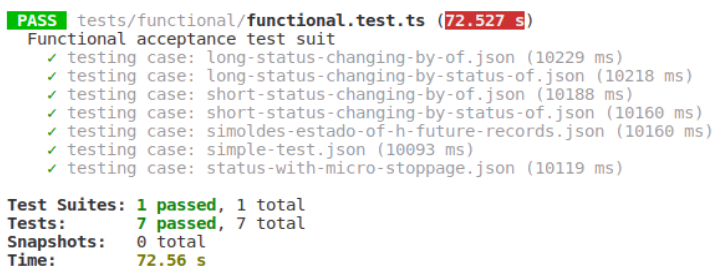

Figura 28 - Resultados de execução da estratégia de testes de aceitação funcionais

6.1.4 Testes de implantação

Com o objetivo de validar se uma implantação é bem-sucedida, surge a necessidade clara de serem implementados testes de implantação.

Devido à natureza deste tipo de testes, a sua implementação difere ligeiramente da dos restantes tipos de testes.

Enquanto todos os testes referidos anteriormente são executados através de um comando de consola, estes testes são executados através de um pedido HTTP/HTTPs para o *ETL C*ore.

Quando recebido o pedido, os testes de implantação fazem essencialmente duas validações distintas:

- Validam se as ligações aos serviços externos estão a funcionar corretamente;
- Executam um teste a uma *pipeline* muito simples, com o intuito de perceber se o serviço funciona como pretendido.

O excerto de código 9 apresenta a função responsável pela execução dos testes de implantação, onde se encontram as validações referidas previamente.

```
public async healthCheck(): Promise<HealthCheckDTO> {
    if (!isPostgresDBConnected() || !redis.isConnected()) {
      return {
        STATUS: 'DOWN',
      };
    }
    const pipelineSuccess: boolean = await this.runSimplePipelineTest();
    return {
      STATUS: pipelineSuccess ? 'UP' : 'DOWN',
    };
  }
```

Excerto de código 9 – Testes de implantação

O processo de validação a serviços externos é relativamente simples, onde se recorrem apenas às bibliotecas utilizadas para interagir com estes sistemas.

Por outro lado, o processo de testar uma *pipeline* já é consideravelmente mais complexo, mas já foi abordado em detalhe ao longo desta secção, não necessitando de esclarecimentos adicionais.

Finalmente, fica claro no excerto de código 9 o retorno da informação que indica se o serviço *ETL Core* se encontra saudável ou não.

6.2 Infraestrutura

Sendo o projeto desenvolvido no âmbito desta dissertação uma prova de conceito sobre entrega de *software*, foi imprescindível dedicar tempo à infraestrutura necessária para a implantação do *ETL Core*.

Esta infraestrutura baseia-se praticamente toda em serviços AWS, sendo utilizado o *Terraform* como ferramenta para gerir esta infraestrutura.

A nível de infraestrutura, para implantar o *ETL Core* é necessário implementar os seguintes passos:

- Criar uma *task definition*;
- Criar um serviço ECS;
- Criar um *cluster* ECS;
- Configurar um *load balancer*.

As secções seguintes vão servir para detalhar a implementação feita em cada ponto mencionado.

6.2.1 Task Definition

Essencialmente, a *task definition* dita de que forma é que o *ETL Core* funciona, parametrizando o contentor onde este serviço corre. O excerto de código 10 apresenta a configuração da *task definition* do *ETL Core*.

```
resource "aws_ecs_task_definition" "service" {
    network_mode = "bridge"
    pid_mode = "task"
    container_definitions = jsonencode([
    {
        name = var.container.name
        image = var.container.image
        portMappings = [{
            containerPort = var.container.port
            hostPort = 0
        }]
        cpu = var.container.cpu
        memoryReservation = var.container.memory
        memory = floor(var.container.memory * 1.25)
        repositoryCredentials = {
            credentialsParameter = var.repository_credentials
        }
        logConfiguration = {
            logDriver = "awslogs"
```

```
            options = {
                awslogs-region = var.aws_region
                awslogs-group = aws_cloudwatch_log_group.service.name
            }}
        environment = concat(var.environment_variables)}])
}
```

Excerto de código 10 – *Task Definition ETL Core*

Examinando o excerto de código 10, é possível compreender que a *task definition* apresentada configura os seguintes aspetos:

- A imagem *Docker* que o contentor irá executar;
- Mapeamento de portas, no caso, o valor da porta dentro do contentor é estático sendo passado pela variável "*var.container.port*" no entanto, a porta que o *host* irá utilizar é dinâmica, podendo ser qualquer valor. Esta configuração é crucial para permitir que mais do que uma instância possa correr na mesma máquina;
- CPU e memória a serem utilizados quando um serviço (no caso, *ETL Core*) é criado através da configuração anteriormente referida;
    - O argumento "*memoryReservation*" especifica a quantidade de memória mínima reservada para o contentor (*soft limit*);
    - O argumento "*memory*" especifica a quantidade de memória máxima que um contentor pode utilizar (*hard limit*);
    - A configuração do *task definition* garante que o *hard limit* é sempre cerca de 25% maior que o *soft limit*.
- As credenciais para o repositório central, de forma a conseguir obter a imagem no repositório *Docker* que será utilizada pelo serviço;
- Configuração de *logs*;
- Especificação de variáveis de ambiente a utilizar no serviço.

6.2.2 Serviço ECS

Com o objetivo de manter um alto nível de disponibilidade no *ETL Core*, foi configurado um serviço ECS.

Neste serviço ECS, são especificados o número mínimo e máximo de instâncias (*tasks*) que deverão ser executadas, assim como qual o número desejado. Esta especificação encontra-se ilustrada no excerto de código 11.

```
resource "aws_ecs_service" "service" {
  cluster         = var.ecs_cluster
  task_definition = aws_ecs_task_definition.service.arn
  ordered_placement_strategy {
    type  = "binpack"
    field = "cpu"
  }
  desired_count                      = 1
  deployment_maximum_percent         = 200
  deployment_minimum_healthy_percent = 100
  deployment_controller {
    type = "ECS"
  }
  load_balancer {
    target_group_arn = aws_lb_target_group.service.arn
    container_name   = var.container.name
    container_port   = var.container.port
  }
}
```

Excerto de código 11 – Configuração serviço *ECS*

No que toca à configuração do serviço ECS para o *ETL Core* presente no excerto de código 11, os principais pontos a serem abordados consistem em:

- Na configuração do *cluster* no qual a instância (ou instâncias) corre;
- Na configuração da *task definition* que irá correr no serviço, sendo a definição desta *task definition* abordada anteriormente;
- Na estratégia de alocação de instâncias, neste serviço é utilizada a estratégia *binpack*, onde as instâncias são alocadas baseadas na disponibilidade da memória e *CPU*, dando prioridade ao *CPU*.
- Na estratégia de implantação, que neste caso garante sempre que uma instância se encontra disponível. Durante uma implantação são duplicadas as instâncias, e apenas quando a nova instância na nova versão se encontrar saudável é que a instância na versão anterior é eliminada.
- Na associação do serviço ao *loadbalancer* (abordado na secção 6.2.4).

6.2.3 Cluster ECS

De forma a correr o serviço ECS configurado na secção anterior, é necessário criar um *Cluster* ECS, dado que qualquer serviço necessita de correr dentro de um *Cluster* ECS.

Na definição de um cluster, a configuração crucial consiste na definição de *capacity providers* responsáveis por gerir a escalabilidade da infraestrutura para tarefas de um determinado *cluster*.

Neste projeto, foi escolhido um único *capacity provider* que gere *tasks* em instâncias EC2. Como serão utilizadas instâncias EC2, é necessário configurar um *auto scaling group*.

Este *auto scaling group* será responsável por tratar diversas instâncias *EC2* como um grupo lógico de forma a permitir a gestão e escalabilidade da infraestrutura automaticamente. Esta configuração é apresentada no excerto de código 12.

```
resource "aws_autoscaling_group" "ecs" {
  name                 = local.resources_name
  launch_configuration = aws_launch_configuration.ecs.name
  vpc_zone_identifier = [
    aws_subnet.main_a.id,
    aws_subnet.main_b.id,
    aws_subnet.main_c.id,
  ]
  health_check_type         = "EC2"
  min_size                  = var.ecs_cluster_min_ec2_instances
  max_size                  = var.ecs_cluster_max_ec2_instances
  desired_capacity          = var.ecs_cluster_desired_ec2_instances
  wait_for_capacity_timeout = 0
  protect_from_scale_in     = false
  capacity_rebalance        = true
}
```

Excerto de código 12 – Configuração *auto scaling group*

Analisando o excerto de código 12, é possível perceber a configuração do número de instâncias mínimo, máximo e desejado (por variáveis) assim como algumas configurações associadas com a gestão da infraestrutura.

Estas configurações permitem que:

- Instâncias sejam terminadas de forma a obedecer aos números mínimos, máximos e desejados especificados;
- Lançar novas instâncias sempre que uma instância estiver com degradação de desempenho, comprometendo assim o correto funcionamento da aplicação.

6.2.4 Load Balancer

Com o objetivo de direcionar o tráfego para o *ETL Core*, principalmente em altura de implantações, em que múltiplas instâncias estão a correr no mesmo *cluster*, é necessário configurar um *application load balancer* (ALB).

Apesar de a configuração de um *application load balancer* abordar diferentes aspetos como configurações de segurança de controlo de tráfego, *subnets*, regras de direcionamento de tráfego, entre outros, neste capítulo será dado ênfase apenas a 2 pontos distintos:

- Associação do *load balancer* ao serviço;
- Definição do *health check*.

No que toca à associação do *load balancer* ao serviço do *ETL Core*, tal é feito na configuração do próprio serviço, como referido na secção anterior.

Assim, sempre que um pedido é enviado para o *load balancer* este fica encarregue de o encaminhar para a instância do *ETL Core* correta. Caso não consiga encontrar a instância pretendida, foi configurada uma ação padrão, que retorna sempre uma resposta com o conteúdo “OK” e o código de *status* 200.

Num cenário de prova de conceito, existe apenas uma única instância do *ETL Core* no *cluster*, para a qual o *load balancer* encaminha todos os pedidos. No entanto, esta situação muda no caso de uma implantação.

Quando ocorre uma implantação, semelhante a uma implantação *blue-green*, é pretendido que numa primeira fase o *load balancer* reencaminhe os pedidos para a instância que se encontra na versão antiga. Quando a nova instância for implantada e se encontrar pronta para receber pedidos, estes devem ser reencaminhados para a nova instância.

De forma ao *load balancer* perceber se a instância se encontra pronta para receber pedidos, é necessário configurar o *health check*, responsável por validar se um serviço se encontra saudável ou não, tal como mostrado no excerto de código 13.

```
health_check {
    enabled             = true
    interval            = 300
    timeout             = 60
    path                = var.health_check_path
    healthy_threshold   = 2
    unhealthy_threshold = 2
  }
```

Excerto de código 13 – Configuração *health check*

De forma sucinta, a configuração do *health check* no excerto de código 13, traduz-se em:

- Uma instância é considerada saudável sempre que dois pedidos seguidos para o *endpoint* de *health check* sejam bem-sucedidos, num intervalo de 300 segundos entre eles.
- Uma instância é considerada não saudável caso dois pedidos seguidos para o *endpoint* de *health check* falhem, havendo um intervalo de 300 segundos entre os pedidos.

6.3 Pipeline de Implantação

Tal como desenhado no capítulo 5 "Análise e *Design*", no âmbito deste projeto foram desenvolvidas 4 *pipelines* de implantação: *pipeline* de desenvolvimento, *pipeline* de testes, *pipeline* de lançamento e *pipeline* de implantação em ambientes.

Embora sejam *pipelines* distintas com objetivos diferentes, existem diversos *stages* que são utilizados em múltiplas *pipelines*.

Posto isto, este capítulo abordará todas as funcionalidades implementadas para as *pipelines* de implantação, especificando quais *pipelines* as usam. Assim, não existirá necessidade de repetir informação entre secções.

Essencialmente, esta secção abordará as seguintes implementações nas *pipelines*:

- *Triggers*;
- Construção e publicação dos artefactos;
- Testes automáticos;
- Implantação;
- Testes manuais.

6.3.1 Implementação dos Triggers

Para implementar o *design* especificado, foi necessário dar suporte a 4 tipos distintos de *trigger*:

1. Sempre que é feito *push* para o ramo de desenvolvimento (necessário para a *pipeline* de desenvolvimento);
2. Sempre que é criada uma *tag* de uma versão candidata (necessário para a *pipeline* de testes);
3. Sempre que é criada uma *tag* de uma versão (necessário para a *pipeline* de lançamento);
4. *Trigger* manual (necessário para a *pipeline* de implantação em ambientes).

Com o objetivo de a pipeline correr sempre que é feito *push* para o ramo de desenvolvimento, a implementação consistiu no apresentado no excerto de código 14.

```
on:
  push:
    branches: [develop]
```

Excerto de código 14 – *Trigger* de *push* no ramo de desenvolvimento

Seguindo a mesma lógica, é possível fazer o mesmo quando o *push* é de uma *tag*, como mostrado pelo excerto de código 15.

```
on:
  push:
    tags: ['*-rc']
```

Excerto de código 15 – *Trigger* de *push* de *tag* de uma versão candidata

Embora o excerto de código 15 apresente apenas o *push* de uma *tag* referente a uma versão candidata (com sufixo "-rc"), é possível aplicar a mesma lógica para uma versão, em que o *trigger* apenas acontece quando o sufixo não é "-rc".

Finalmente, para dar suporte ao *trigger* manual, a implementação consistiu no exibido no excerto de código 16.

```
on:
    workflow_dispatch:
      inputs:
        RELEASE_IMAGE:
                description: 'Image version to deploy:'
                required: true
                default: 0.1.12
```

Excerto de código 16 – *Trigger* manual

Considerando o excerto de código 16, a principal diferença dos restantes *triggers* consiste na possibilidade de permitir ao utilizador inserir informação de entrada, tal como a versão a ser utilizada por exemplo.

6.3.2 Construção e publicação de Artefactos

No que toca à construção e publicação de artefactos, esta pode-se dividir em 3 etapas distintas:

- A primeira etapa consiste na compilação do *software*;
- A segunda etapa consiste em guardar os artefactos resultantes da compilação, de modo a estes poderem ser utilizados pelos restantes *stages* da *pipeline*;
- Na terceira etapa, é feita a construção da imagem *Docker* e respetiva publicação no *Nexus*, que consiste no repositório de artefactos de *software* utilizado pela organização.

As primeiras duas etapas são utilizadas tanto na *pipeline* de desenvolvimento como na *pipeline* de testes. Já a última etapa é utilizada pelas duas *pipelines* referidas anteriormente e também pela *pipeline* de lançamento.

No que toca à primeira etapa, o processo de compilação é simples, como o *ETL* core consiste numa aplicação *node.js* em *typescript*, apenas é necessário instalar todas as bibliotecas necessárias e posteriormente converter o código do projeto para *javacript*, tal como mostrado no excerto de código 17.

```
- name: Build
        run: npm install && npx lerna run build
```

Excerto de código 17 – Compilação do *software*

Relativamente à segunda etapa, o principal objetivo desta é evitar que o *software* seja compilado várias vezes. Assim, foi utilizada uma ação do próprio *GitHub Actions*, que permite

guardar e obter artefactos, sendo estes artefactos facilmente acessíveis por qualquer *stage* em qualquer *pipeline*.

O excerto de código 18 apresenta como é feito o arquivo do *software* compilado.

```
- name: Archive artifacts
        uses: actions/upload-artifact@v3
        with:
          name: development-pipeline-etl-core
          path: |
            packages/**/dist
            !packages/**/dist/*.md
          retention-days: 7
```

Excerto de código 18 – Guardar artefactos

Como apresentado no excerto de código 18, são arquivadas todas as pastas "*dist*", onde está o *software* compilado, sendo ignorados os ficheiros de documentação. Este artefacto é guardado apenas durante 1 semana.

No que toca à terceira e última etapa, é construída imagem *Docker* com base num *Dockerfile* já existente. Posteriormente é publicada a nova imagem no repositório de artefactos *Nexus*. Esta *stage* é apresentado no excerto de código 19.

```
- name: Login to Nexus Registry
        run:      docker      login      registry.progrow.io:5002      -u
"${{       secrets.NEXUS_REPOSITORY_CREDENTIALS_USERNAME       }}"       -p
"${{ secrets.NEXUS_REPOSITORY_CREDENTIALS_PASSWORD }}"
- name: Publish artifacts to nexus
        id: publish-artifacts
        run: |
            ./build_image.sh -p core -v ${{env.RELEASE_IMAGE}}
            ./publish_image.sh -p core -v ${{env.RELEASE_IMAGE}}
            echo
"image=registry.progrow.io:5002/etlmodulecore:$RELEASE_IMAGE"            >>
$GITHUB_OUTPUT
```

Excerto de código 19 – *Stage* de construção e publicação de artefactos

Analisando o excerto de código 19, é possível perceber que o primeiro passo neste *stage* consiste em realizar a autenticação no *Nexus.* Posteriormente é feita a construção e publicação da imagem *Docker* recorrendo a scripts *Shell*. Finalmente, a imagem é guardada numa variável chamada *image*, para que possa ser consultada por outros *stages*.

6.3.3 Testes Automáticos

Abordando os testes automáticos, estes essencialmente são utilizados em duas *pipelines* distintas: na *pipeline* de desenvolvimento e na *pipeline* de implantação.

Como foram implementados quatro tipos de testes distintos (testes unitários, testes de integração, testes de aceitação funcionais e testes de implantação), esta secção irá abordar cada um deles individualmente.

Começando pela execução dos testes unitários, estes são executados no *stage* de *commit*, logo após a compilação, através da execução de um único comando, apresentado no excerto de código 20.

```
- name: Unit test stage
  run: npx lerna run test -- unit
```

Excerto de código 20 – Execução de testes unitários

Uma vez que a execução dos testes de integração é praticamente igual aos testes de aceitação funcionais, serão apenas abordados os testes de integração funcionais. Posto isto, o excerto de código 21 apresenta como é feita a execução deste tipo de testes.

```
- name: Download artifacts
  uses: actions/download-artifact@v3
  with:
    name: development-pipeline-etl-core
    path: ./packages
- name: Sets up environment
  run: |
    docker-compose up -d
    cd packages/management && npm run start &
- name: Functional test stage
  run: cd packages/core && npm run test – functional
```

Excerto de código 21 – Execução de testes de aceitação funcionais

Analisando o excerto de código 21, é possível perceber que a execução dos testes de aceitação funcionais se divide em três passos distintos:

1. Em primeiro lugar, são obtidos os artefactos resultantes da compilação do *software*;
2. De seguida, é criada toda a infraestrutura que dá suporte ao *ETL Core*, que é necessária para a execução deste tipo de testes. Sendo toda esta infraestrutura montado no próprio *runner* do *GitHub Actions*;
3. Finalmente, são executados de facto os testes.

É importante referir que, tal como proposto no *design*, o *stage* de testes de integração e o *stage* de testes de aceitação funcionais são executados em paralelo.

Nesta fase, o único tipo de testes que falta abordar consiste nos testes de implantação.

Aquando da implementação deste tipo de testes, a decisão consistiu na *pipeline* não executar este tipo de testes após a implantação e sim recorrer ao mecanismo de *health check* existente para executar este tipo de testes.

O mecanismo de *health check* já foi abordado na secção 6.2.4 "*Load Balancer*" deste documento, assim, a única coisa que é importante detalhar é que, o pedido feito pelo mecanismo de *health check* irá executar os testes de implantação.

6.3.4 Implantação

No que toca ao *stage* de implantação, este foi implementado em três *pipelines*: *pipeline* de desenvolvimento, *pipeline* de testes e *pipeline* de implantação em ambientes. Em todas as *pipelines* referidas, este processo é análogo ao apresentado no excerto de código 22.

```
- name: Download task definition
  run: |
    aws ecs describe-task-definition --task-definition ${{env.ECS_SERVICE}}
--query taskDefinition > task-definition.json

- name: Fill in the new image ID in the Amazon ECS task definition
  id: task-def
  uses:                                  aws-actions/amazon-ecs-render-task-
definition@c804dfbdd57f713b6c079302a4c01db7017a36fc
  with:
    task-definition: ${{ env.ECS_TASK_DEFINITION }}
    container-name: ${{ env.CONTAINER_NAME }}
    image: ${{ steps.build-image.outputs.image }}

- name: Deploy Amazon ECS task definition
  uses:                                  aws-actions/amazon-ecs-deploy-task-
definition@df9643053eda01f169e64a0e60233aacca83799a
  with:
    task-definition: ${{ steps.task-def.outputs.task-definition }}
    service: ${{ env.ECS_SERVICE }}
    cluster: ${{ env.ECS_CLUSTER }}
    wait-for-service-stability: true
```

Excerto de código 22 – *Stage* de implantação

Analisando o excerto de código 22, é possível perceber que o processo de implantação se divide em três passos:

1. A obtenção da *task definition* mais recente, uma vez que as *task definitions* são definidas pelo *Terraform* e não se encontram no repositório do *ETL Core*;
2. A atualização da *task definition* obtida previamente com a nova imagem gerada;

3. A implantação propriamente dita, sendo necessário especificar o *cluster*, serviço *ECS* e a *task definition* alterada previamente.

Como está a ser utilizado o serviço da AWS ECS para a implantação da aplicação com a configuração já abordada na secção 6.2 “Infraestrutura”, o processo de implantação não apresenta *downtime* além de garantir que a versão implantada se encontra estável.

Assim, apenas quando a implantação é concluída e a nova versão é dada como estável, é que a *pipeline* procede para o *stage* seguinte.

6.3.5 Testes Manuais

Unicamente utilizado pela *pipeline* de testes, este *stage* ocorre após a implantação para o ambiente de QA e visa permitir a execução de testes manuais de aceitação funcionais e testes exploratórios, tal como desenhado previamente.

Dado esta necessidade, o excerto de código 23 apresenta como foi feita a implementação do *stage* de testes manuais.

```
- name: Manual test stage
  uses: trstringer/manual-approval@v1.9.0
  with:
    secret: ${{ github.TOKEN }}
    approvers: bruno-vilar-pg
    minimum-approvals: 1
    issue-title: "Manual tests"
    issue-body: "Please approve or deny the manual functional acceptance
tests and exploratory tests"
    exclude-workflow-initiator-as-approver: false
    additional-approved-words: "sim, aprovado"
    additional-denied-words: "não, desaprovado"
```

Excerto de código 23 – *Stage* de testes manuais

Sucintamente, a implementação presente no excerto de código 27 cria um *issue* que necessita de ser aprovado por alguém com as permissões necessários, neste caso alguém da equipa de QA. Enquanto este *issue* não é aceite ou rejeitado, a pipeline não passa para o *stage* seguinte.

Após os testes manuais de aceitação funcionais e os testes exploratórios serem terminados, alguém da equipa de QA com as permissões necessárias pode aceitar ou rejeitar o *issue*.

6.3.6 Sumário

Concluída a implementação, fica evidente que a solução implementada cumpre com os requisitos funcionais e não funcionais impostos.

A implementação seguiu o *design* elaborado, sendo que o único desvio em relação a este consistiu na implementação dos testes de implantação recorrendo ao mecanismo de *health check* disponibilizado pelo ECS.

7 Avaliação

Neste capítulo é descrita a avaliação da solução anteriormente concebida, com o objetivo de mostrar de que forma é que esta responde aos requisitos que foram colocados.

Assim, este capítulo começa por formular a hipótese de investigação com base nos objetivos propostos. De seguida, são especificados tanto os indicadores e fontes de informação como a metodologia de avaliação. Posteriormente, é feita a avaliação dos resultados. Finalmente, este capítulo termina com a apresentação das principais conclusões do processo de avaliação.

7.1 Hipótese de Investigação

De forma a formular a hipótese de investigação, é necessário utilizar como base os objetivos descritos previamente.

Sendo o principal objetivo desta dissertação a construção de uma solução de entrega de *software* automática e fiável, a hipótese de investigação é: "A adoção de um método de entrega de *software* automático pode melhorar a eficiência, eficácia e fiabilidade do processo de entrega de *software*".

7.2 Indicadores e Fontes de Informação

Para validar a hipótese de investigação especificada é necessário recolher indicadores que permitam aceitar ou rejeitar a hipótese com um elevado grau de confiança. Com este fim em mente, serão recolhidos dois indicadores distintos.

O primeiro tipo de indicador consiste em dados associados à utilização da solução desenvolvida, como por exemplo: tempo de implantação e tempo de desenvolvimento (Lehtonen *et al.*, 2015).

O segundo tipo de indicador trata-se de uma entrevista semiestruturada feita após a utilização da solução desenvolvida pela equipa, com o objetivo de perceber o grau de satisfação dos utilizadores.

Para esta entrevista, será preparado um conjunto de questões tendo em conta a hipótese de investigação formulada.

Com este par de indicadores, será possível avaliar a solução tanto de forma quantitativa como de forma qualitativa.

7.3 Metodologia de Avaliação

Após a identificação dos indicadores e fontes de informação, é importante perceber como foi desempenhada a avaliação da solução.

A metodologia de avaliação incidirá essencialmente sobre dois pontos distintos:

- No uso de testes estatísticos comparando os mesmos indicadores da solução utilizada atualmente e a nova solução desenvolvida.
- Na elaboração do modelo de maturidade.

Relativamente aos testes estatísticos, tendo os indicadores recolhidos como base, serão formuladas hipóteses que comparem a nova solução com a abordagem existente e efetuados testes estatísticos, como o *t test* e o *Wilcoxon*, para aceitar ou rejeitar estas hipóteses.

No que toca ao modelo de maturidade, este é um aspeto crucial na avaliação da solução. Este modelo ajuda a identificar como é que a organização se posiciona a nível de maturidade dos seus processos e práticas em diversos critérios distintos (Humble and Farley, 2010). A figura 29 apresenta este modelo.

Practice	Build management and continuous integration	Environments and deployment	Release management and compliance	Testing	Data management	Configuration management
Level 3 - Optimizing: Focus on process improvement	Teams regularly meet to discuss integration problems and resolve them with automation, faster feedback, and better visibility.	All environments managed effectively. Provisioning fully automated. Virtualization used if applicable.	Operations and delivery teams regularly collaborate to manage risks and reduce cycle time.	Production rollbacks rare. Defects found and fixed immediately.	Release to release feedback loop of database performance and deployment process.	Regular validation that CM policy supports effective collaboration, rapid development, and auditable change management processes.
Level 2 - Quantitatively managed: Process measured and controlled	Build metrics gathered, made visible, and acted on. Builds are not left broken.	Orchestrated deployments managed. Release and rollback processes tested.	Environment and application health monitored and proactively managed. Cycle time monitored.	Quality metrics and trends tracked. Non functional requirements defined and measured.	Database upgrades and rollbacks tested with every deployment. Database performance monitored and optimized.	Developers check in to mainline at least once a day. Branching only used for releases.
Level 1 - Consistent: Automated processes applied across whole application lifecycle	Automated build and test cycle every time a change is committed. Dependencies managed. Re-use of scripts and tools.	Fully automated, self-service push-button process for deploying software. Same process to deploy to every environment.	Change management and approvals processes defined and enforced. Regulatory and compliance conditions met.	Automated unit and acceptance tests, the latter written with testers. Testing part of development process.	Database changes performed automatically as part of deployment process.	Libraries and dependencies managed. Version control usage policies determined by change management process.
Level 0 – Repeatable: Process documented and partly automated	Regular automated build and testing. Any build can be re-created from source control using automated process.	Automated deployment to some environments. Creation of new environments is cheap. All configuration externalized / versioned	Painful and infrequent, but reliable, releases. Limited traceability from requirements to release.	Automated tests written as part of story development.	Changes to databases done with automated scripts versioned with application.	Version control in use for everything required to recreate software: source code, configuration, build and deploy scripts, data migrations.
Level -1 – Regressive: processes unrepeatable, poorly controlled, and reactive	Manual processes for building software. No management of artifacts and reports.	Manual process for deploying software. Environment-specific binaries. Environments provisioned manually.	Infrequent and unreliable releases.	Manual testing after development.	Data migrations unversioned and performed manually.	Version control either not used, or check-ins happen infrequently.

Figura 29 - Modelo de Maturidade

Interpretando a figura 29, é possível perceber que o modelo de maturidade avalia as práticas e processos da organização em seis critérios diferentes: gestão do processo de compilação e

integração contínua, ambientes e implantação, gestão de lançamento de *software* e se o mesmo está em conformidade com os seus requisitos, testes, gestão de dados e gestão de configurações.

No entanto, no âmbito desta dissertação o foco consiste maioritariamente nos primeiros 4 critérios.

Cada um dos critérios referido é avaliado em 4 níveis distintos (Humble and Farley, 2010):

- Nível -1 ou regressivo: o processo não é repetível e é pouco controlado;
- Nível 0 ou repetível: o processo é documentado e parcialmente automatizado;
- Nível 1 ou consistente: aqui existem processos automáticos aplicados ao longo de todo o ciclo de vida da aplicação;
- Nível 2 ou gerido quantitativamente: neste nível o processo é medido e controlado;
- Nível 3 ou otimização: onde o foco consiste na melhoria do processo.

A elaboração deste modelo permite perceber não só o grau de sucesso da solução, mas também que melhorias ainda podem ser feitas e em que sentido.

No final do processo, foi comparado o modelo de maturidade do estado atual da organização, com o modelo de maturidade após a utilização da solução, de modo a compreender o impacto da solução na organização.

7.4 Avaliação de Resultados

Tal como especificado na metodologia de avaliação, a avaliação de resultados encontra-se separada em 3 etapas distintas:

- A avaliação quantitativa recorrendo a métricas recolhidas tanto da solução utilizada atualmente como da solução desenvolvida.
- A avaliação qualitativa através dos dados recolhidos na entrevista semiestruturada efetuada;
- Na comparação do modelo de maturidade da organização sem o projeto desenvolvido e com o projeto desenvolvido.

7.4.1 Avaliação de Métricas Recolhidas

De forma a efetuar esta avaliação quantitativa, como já referido na secção 7.2 “Indicadores e Fontes de Informação”, serão analisadas as seguintes métricas: tempo de implantação e tempo de implementação.

7.4.1.1 Tempo de Implantação

No que toca ao tempo de implantação, esta métrica consiste no tempo que demora o processo de implantação. Por outras palavras, consiste em todo o tempo necessário dedicar para atualizar uma versão num determinado ambiente e garantir que esta está operacional.

Uma vez que o novo processo de implantação ainda não é utilizado em produção, será comparado o tempo de implantação das últimas dez implantações no ambiente de QA.

É importante relembrar que existe uma certa imprecisão nestes valores, principalmente devido ao fato de o processo existente ser manual e também pelo facto da recolha desta informação ser manual.

Analisando a média dos valores, é possível perceber que o processo anterior durava cerca de 24 minutos, já o novo processo dura cerca de 8 minutos e 13 segundos. Isto significa uma redução de aproximadamente dois terços do tempo com a nova solução implementada, que por sua vez vai permitir à equipa fazer três implantações pelo custo de uma a nível de tempo de implantação gasto.

Por outro lado, recorrendo a testes estatísticos, os dados recolhidos relativos ao processo de implantação atual são considerados o grupo 1, já os dados recolhidos relativos ao processo de implantação desenvolvido são designados de grupo 2.

De seguida, é necessário verificar a possibilidade de utilizar o teste *t* que permite verificar se existe uma diferença substancial entre as médias dos 2 grupos.

Para efetuar um teste *t* é necessário que existam as seguintes suposições:

- Os dados a analisar são contínuos (não discretos);
- Garantir que cada elemento da população tem a mesma probabilidade de ser selecionado;
- O desvio padrão não é conhecido;
- As amostras vêm de uma população normalmente distribuída (ou a amostra é maior que 30);

É possível garantir que as primeiras 3 suposições estão presentes, no entanto, para verificar se a 4ª suposição é verdadeira, é necessário efetuar o teste de *Shapiro-Wilk* para ambos os grupos.

Para este teste são consideradas duas hipóteses:

1. H0: A população é distribuída normalmente;
2. H1: A população não é distribuída normalmente.

Para o grupo 1, o *p-value* obtido foi 0.1826, já para o grupo 2, o *p-value* obtido foi 0.1869. Como o *p-value* de ambos os grupos se encontra acima de 0.5, a H0 é rejeitada.

Como não é possível assumir que a população se encontra distribuída normalmente, não é possível efetuar o teste *t*.

Como alternativa, foi utilizado o teste *Wilcoxon* que consiste num teste não paramétrico. Para este novo teste, são consideradas as seguintes hipóteses:

1. H0: A média do grupo 1 é igual à do grupo 2.
2. H1: A média do grupo 2 é menor que a do grupo 1.

O p-value obtido na execução deste teste foi de 0.0020, é possível rejeitar a hipótese 0 e afirmar que o tempo de implantação do novo processo (grupo 2) é menor que o tempo de implantação do processo utilizado atualmente (grupo 1).

7.4.1.2 Tempo de Implementação

Tal como o nome indica, este indicador consiste no tempo gasto a implementar uma determinada funcionalidade. Por outras palavras, o tempo que demora a uma determinada funcionalidade ser incluída no ramo de desenvolvimento.

Para cada processo foram recolhidos dez tempos de implementação de funcionalidades semelhantes.

Analisando a média dos valores, é possível perceber que o processo anterior, o seu tempo médio de implementação consistia em 6 horas e 6 minutos, já o novo processo, demora cerca de 7 horas e 18 minutos.

Nesta secção, o processo anterior será designado por grupo 1, já o novo processo que foi desenvolvido será chamado de grupo 2.

Tal como no indicador "Tempo de Implantação" abordado na secção anterior, para tornar viável a execução de um teste t, é necessário efetuar um teste de *Shapiro-Wilk* para ambos os grupos. Para este teste, as hipóteses consideradas são:

1. H0: A população é distribuída normalmente;
2. H1: A população não é distribuída normalmente.

Para o grupo 1, o *p-value* obtido foi 0.1704, já para o grupo 2, o *p-value* obtido foi 0.0753. Como o *p-value* de ambos os grupos se encontra acima de 0.5, a H0 é rejeitada.

Como não é possível utilizar o teste t, é utilizado o teste *Wilcoxon* considerando as seguintes hipóteses:

1. H0: A média do grupo 1 é igual à do grupo 2.
2. H1: A média do grupo 1 é menor que a do grupo 2.

A execução do teste *Wilcoxon* obteve o *p-value* de 0.1694. Assim, como o *p-value* não é inferior a 0.5, não é possível descartar a H0 e, consequentemente, não é possível afirmar que o tempo de implementação de uma funcionalidade aumentou.

Embora não seja possível tirar grandes conclusões, é expectável que, com a adoção da estratégia de testes implementada, o tempo de implementação seja maior numa fase inicial, uma vez que é necessário desenvolver testes para todo o código implementado. No entanto, há medida que o projeto cresce, é expectável que a utilização desta estratégia de testes faça o tempo de implementação diminuir, uma vez que alterações ao código se tornarão mais fáceis.

7.4.2 Entrevista Semiestruturada

Relativamente à entrevista semiestruturada, esta consiste num conjunto de quatro perguntas de resposta aberta com o objetivo de perceber o grau de satisfação dos utilizadores.

Assim, esta entrevista foi feita aos três elementos da equipa para a qual a solução foi desenvolvida.

O conjunto de perguntas realizado na entrevista, assim como a análise das respostas é apresentado na tabela 15.

Tabela 15 – Entrevista semiestruturada

Nº da questão	**Questão**	**Análise**
1	Acha que a nova estratégia de testes é eficiente em garantir a qualidade do projeto em que esta se insere?	Existe um consenso entre os entrevistados de que a nova estratégia de testes é capaz de garantir os requisitos de qualidade pretendidos. Uma ideia partilhada entre vários entrevistados consiste na importância que esta estratégia trará para o projeto quando a cobertura de testes aumentar. Foi mencionado o fato desta estratégia já ter detetado 2 *bugs* que estavam em produção.
2	Considera a adição de novos testes um processo complexo?	As respostas a esta questão foram bastante homogêneas, onde todos os entrevistados concordaram que o processo de adição de novos testes não é complexo. No entanto, foi referido por mais do que 1 entrevistado que apesar do processo ser simples, é demorado, principalmente no caso dos testes de aceitação funcionais.
3	Comparando este processo de lançamento de *software* com o anterior, o quão confiante se sente com o lançamento de uma nova versão?	De modo geral, os entrevistados sentem-se mais confiantes com o lançamento de uma nova versão com o projeto desenvolvido no âmbito desta dissertação. No entanto, foi alertado que ainda é necessário realizar trabalho para aumentar a cobertura de testes.
4	Acredita que este novo processo permite reduzir o tempo de entrega de funcionalidades ao utilizador? Se sim, quais são os principais fatores que contribuem para tal?	Todas as respostas foram positivas. Como principais fatores foram indicados: a garantia de que o *software* se encontra sempre testado e a facilidade em fazer implantações, reduzindo assim o tempo gasto nestas fases do ciclo de vida do desenvolvimento de *software*.

7.4.3 Comparação de Modelos de Maturidade

Como mencionado na secção 7.3 “Metodologia de Avaliação” deste documento, parte do processo de avaliação consiste em comparar o modelo de maturidade da organização sem o projeto desenvolvido nesta dissertação e após o desenvolvimento desta solução.

No entanto, não será comparado o modelo de maturidade na sua totalidade, apenas serão comparados os seguintes critérios: gestão do processo de compilação e integração contínua, ambientes e implantação, gestão de lançamento de *software* e se o mesmo está em conformidade com os seus requisitos e testes.

É importante ter em mente que o nível atribuído em cada um dos critérios terá em consideração a figura 29, presente na secção 7.3 “Metodologia de Avaliação”, que apresenta todos os critérios do modelo de maturidade.

Assim, sem o projeto desenvolvido nesta dissertação, o modelo de maturidade é o seguinte:

- Gestão do processo de compilação e integração contínua: nível 0. O processo de compilação não é inteiramente automático, no entanto o processo é repetível e bem documentado.
- Ambientes e implantação: entre o nível -1 e 0. Embora a implantação de *software* seja manual, a configuração é devidamente versionada e a criação de novos ambientes é relativamente fácil e rápida de executar.
- Gestão de lançamento de *software* e se o mesmo está em conformidade com os seus requisitos: nível 0. O processo de lançamento de uma nova versão não é frequente e é demorado, no entanto, este processo é devidamente documentado.
- Testes: nível 0. Os únicos testes que existem são manuais, feitos após o desenvolvimento de cada funcionalidade e no fim de uma versão ser fechada.

Com o projeto desenvolvido nesta dissertação, o modelo de maturidade da organização é o seguinte:

- Gestão do processo de compilação e integração contínua: entre o nível 1 e 2. O processo é executado sempre que uma alteração é efetuada no ramo de desenvolvimento, as dependências são geridas eficientemente e, se uma compilação falhar, é prioridade máxima resolver o problema. No entanto, ainda não são recolhidas métricas associadas ao processo de compilação;

- Ambientes e implantação: entre o nível 1 e 2. Embora o processo de implantação seja totalmente automático, o processo de *rollback* não se encontra bem definido.
- Gestão de lançamento de *software* e se o mesmo está em conformidade com os seus requisitos: entre nível 1 e 2. Apesar do processo de lançamento de uma nova versão ser automático e existirem mecanismos que verifiquem a saúde do ambiente e da aplicação, não existem mecanismos de gestão proativa.
- Testes: nível 1. Testes totalmente automáticos, fazendo os mesmos parte do processo de desenvolvimento. No entanto, não existem testes de aceitação não funcionais automáticos.

Comparando os dois modelos de maturidade, é possível perceber que a adoção do projeto desenvolvido fará a organização evoluir 1 ou 2 níveis em todos os critérios referidos. Tal deve-se à adição de uma estratégia de testes automática e eficaz, assim como a automatização do processo de implantação.

No entanto, é importante referir que, a melhoria do modelo de maturidade depende não só da qualidade do projeto desenvolvido, como também dos processos e práticas presentes dentro da organização. Isto significa que, para o modelo de maturidade evoluir mais ainda, não só a solução deve ser melhorada, como também as práticas e processos internos.

7.4.4 Conclusões da Avaliação

Embora tenha sido possível analisar a solução tanto quantitativamente como qualitativamente, a recolha de métricas quantitativas deixou a desejar, uma vez que a maioria das métricas úteis implicariam o uso do projeto desenvolvido já em produção.

No entanto, tanto a avaliação quantitativa como a avaliação qualitativa apresentaram bons resultados, validando o cumprimento dos requisitos da solução desenvolvida.

As elaborações dos modelos de maturidade evidenciam claramente o ganho da adoção desta solução. No entanto, fica claro que, mesmo com a adoção da solução, ainda existe um longo caminho a ser percorrido para a organização poder ser considerada excelente.

Em suma, esta avaliação mostra que o projeto apresentado é um importante primeiro passo a ser dado no longo caminho de automatizar as práticas e processos de desenvolvimento e lançamento de versões.

8 Conclusões

Após o processo de avaliação da solução desenvolvida, é necessário refletir sobre quais foram as conclusões obtidas. Para tal, é pertinente contextualizar o trabalho efetuado.

O problema que motivou esta dissertação consiste na dificuldade de entregar aplicações baseadas em análise de dados, de forma eficiente e fiável.

Assim, foi necessário estudar o estado de arte atual, onde o foco consistiu no estudo da *pipeline* de implantação e nas diferentes estratégias de testes, sendo analisadas estratégias de testes em aplicações baseadas na geração de informação a partir de dados. Além do estudo destes conceitos, foram também analisadas tecnologias capazes de lhes dar suporte.

Após o estudo do estado de arte, foi analisado de que forma é que esta solução agrega valor à organização.

De seguida, procedeu-se à recolha dos requisitos e ao desenho de uma solução teoricamente capaz de cumprir com estes. A solução consistiu numa *pipeline* de implantação bastante modular, dividida em múltiplas *pipelines*, capaz de entregar *software* de forma eficiente, fiável e sem *downtime*. A fiabilidade encontra-se assegurada através de uma estratégia de testes sólida, capaz de testar o *software* de diferentes formas, dando enfâse à qualidade dos seus dados.

A implementação da solução seguiu o *design* proposto, atingindo o resultado pretendido com apenas um ligeiro desvio.

Por fim, a solução foi avaliada tanto quantitativamente quanto qualitativamente, validando que esta cumpre os objetivos a que foi proposta.

Posto isto, resta perceber quais foram os objetivos atingidos, quais os objetivos que ficaram por atingir e quais linhas condutoras devem ser adotadas na elaboração de trabalho futuro.

8.1 Objetivos Alcançados

O principal objetivo proposto como âmbito desta dissertação consiste no desenvolvimento de uma solução que garanta a automatização da entrega de *software* baseado em análise de dados, garantindo a qualidade da mesma.

Deste modo, este objetivo foi alcançado com sucesso através da criação da *pipeline* de implantação com uma estratégia de testes robusta.

A solução apresentada não só cumpre com o objetivo proposto, como também se adapta aos diversos processos e metodologias presentes na organização, permitindo uma adoção relativamente fácil.

Assim, é possível afirmar que a solução consiste numa base sólida, que deve ser utilizada e melhorada com o objetivo de contribuir para a automatização dos processos de desenvolvimento e lançamento de versões presentes na organização, garantindo a qualidade constante das mesmas.

8.2 Objetivos Não Alcançados

Embora o objetivo principal tenha sido atingido com sucesso, algumas funcionalidades foram deixadas de fora por questões de tempo disponível para o desenvolvimento do projeto, bem como o grau de maturidade da organização relativamente a estes processos.

Posto isto, a estratégia de testes implementada carece da utilização de testes de aceitação não funcionais automáticos que seria importante para garantir a qualidade dos requisitos não funcionais da aplicação testada.

Outro ponto que ficou por explorar foi uma melhor integração entre a *pipeline* de implantação e a execução no *Terraform* de forma a automatizar a gestão da infraestrutura para os diferentes ambientes.

8.3 Trabalho Futuro

No que toca às linhas condutoras para elaboração do trabalho futuro, o primeiro passo deverá consistir em alcançar os objetivos que não foram alcançados.

De seguida, seria importante não só aumentar a cobertura de testes, como também utilizar novas estratégias de testes, como por exemplo *metamorphic testing*, resultando num processo ainda melhor de garantia de qualidade.

Posteriormente, seria importante este processo de entrega de *software* se generalizar para todas as equipas da organização. Deste modo, seria possível acelerar o desenvolvimento e entrega de todo o *software* e não apenas de uma parte dele.

Outra possível melhoria consiste em todo o processo de recolha de métricas, principalmente da ferramenta desenvolvida. Esta melhoria traria informação importante que auxiliaria todo o processo de evolução da solução.

Finalmente, devem ser melhorados cada um dos critérios presentes no modelo de maturidade, permitindo à organização entregar *software* de forma cada vez mais rápida e fiável.

Referências

Amazon (2023a) *Amazon ECS services*. Available at: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/ecs_services.html.

Amazon (2023b) *Amazon ECS task definitions*. Available at: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/task_definitions.html.

Amazon (2023c) *Computação em nuvem com a AWS*. Available at: https://aws.amazon.com/pt/what-is-aws/?nc1=f_cc.

Amazon (2023d) *Recursos do Amazon EC2*. Available at: https://aws.amazon.com/pt/ec2/features/.

Amazon (2023e) *Recursos do Amazon ECS*. Available at: https://aws.amazon.com/pt/ecs/features/.

Asfaw, D. (2015) 'Benefits of Automated Testing Over Manual Testing'.

Bach, J. (2003) 'Exploratory Testing Explained', *Online: http://www. satisfice. com/articles/et-article. pdf*, pp. 1–10. Available at: http://people.eecs.ku.edu/~saiedian/Teaching/Fa07/814/Resources/exploratory-testing.pdf.

Barr, E.T. *et al.* (2015) 'The oracle problem in software testing: A survey', *IEEE Transactions on Software Engineering*, 41(5), pp. 507–525. Available at: https://doi.org/10.1109/TSE.2014.2372785.

Cheddadi, H., Motahhir, S. and Ghzizal, A. El (2022) 'Google Test/Google Mock to Verify Critical Embedded Software', pp. 1–16. Available at: http://arxiv.org/abs/2208.05317.

Cleland-Huang, J. (2007) 'Quality requirements and their role in successful products', *Proceedings - 15th IEEE International Requirements Engineering Conference, RE 2007*, p. 361. Available at: https://doi.org/10.1109/RE.2007.45.

Computer Consulting (2023) *ECS vs Fargate. Qual devo escolher?* Available at: https://www.computerconsulting.com.br/ecs-vs-fargate-qual-devo-escolher/.

Crispin, L. and Gregory, J. (2009) *Agile Testing: A Practicle Guide For Testers And Agile Teams*.

Cypress (2023) Available at: https://docs.cypress.io/guides/overview/why-cypress.

Docker (2023) *Docker Overview*. Available at: https://docs.docker.com/get-started/overview/.

Eeles, P. (2004) 'What is an Architectural Requirement ?', (November 2001), pp. 1–20.

Farley, D. (2021) 'Continuous Delivery Pipelines: How To Build Better Software , Faster'.

Fitzgerald, B. and Stol, K.J. (2017) 'Continuous software engineering: A roadmap and agenda', *Journal of Systems and Software*, 123, pp. 176–189. Available at: https://doi.org/10.1016/j.jss.2015.06.063.

Forsgren, N., Humble, J. and Kim, G. (2018) 'Accelerate: The science of lean software and devops', *Portland, OR: ITRevolution* [Preprint].

Fowler, M. (2006) 'Continuous integration', *Digital Workflows in Architecture* [Preprint]. Available at: https://doi.org/10.1515/9783034612173.228.

Fowler, M. (2013) *ContinuousDelivery*. Available at: https://martinfowler.com/bliki/ContinuousDelivery.html (Accessed: 16 February 2022).

Gao, J. *et al.* (2019) 'Invited paper: What is ai software testing? and Why', *Proceedings - 13th IEEE International Conference on Service-Oriented System Engineering, SOSE 2019, 10th International Workshop on Joint Cloud Computing, JCC 2019 and 2019 IEEE International Workshop on Cloud Computing in Robotic Systems, CCRS 2019*, pp. 27–36. Available at: https://doi.org/10.1109/SOSE.2019.00015.

GitHub (2023a) *GitHub Actions*. Available at: https://github.com/features/actions.

GitHub (2023b) *Understanding GitHub Actions*. Available at: https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions.

Humble, J. (2023) *Evidence and Case Studies*. Available at: https://continuousdelivery.com/evidence-case-studies/ (Accessed: 20 February 2023).

Humble, J. and Farley, D. (2010) *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*.

Jacobson, I., Booch, G. and Rumbaugh, J. (1999) *The Unified Software Development Process*.

Janoščová, R. (2012) *EVALUATION OF SOFTWARE QUALITY*.

Jenkins (2023) *https://www.jenkins.io/*. Available at: https://www.jenkins.io/.

JestJS (2023) Available at: https://jestjs.io/.

JetBrains (2023) *TeamCity*. Available at: https://www.jetbrains.com/teamcity/.

JMeter (2023) Available at: https://jmeter.apache.org/.

K6 (2023) Available at: https://k6.io/.

Koen, P.A. *et al.* (1996) 'Fuzzy Front End : and Techniques', *Industrial Research*, pp, pp. 5–35. Available at: http://www.stevens.edu/cce/NEW/PDFs/FuzzyFrontEnd_Old.pdfNEW/PDFs/FuzzyFrontEnd_Old.pdf.

Laukkanen, E., Itkonen, J. and Lassenius, C. (2017) 'Problems, causes and solutions when adopting continuous delivery—A systematic literature review', *Information and Software Technology*, 82, pp. 55–79. Available at: https://doi.org/10.1016/j.infsof.2016.10.001.

Lehtonen, T. *et al.* (2015) 'Defining metrics for continuous delivery and deployment pipeline', *CEUR Workshop Proceedings*, 1525, pp. 16–30.

Leung, H.K.N. and White, L. (1990) 'A study of integration testing and software regression at the integration level', *Conference on Software Maintenance*, pp. 290–301. Available at: https://doi.org/10.1109/icsm.1990.131377.

Mackinnon, T., Freeman, S. and Craig, P. (2001) 'Endo-Testing : Unit Testing with Mock Objects', *Extreme programming examined*, pp. 287–301. Available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.23.3214&rep=rep1&type=pdf.

Melymuka, V. (2012) *TeamCity 7 Continuous Integration Essentials.*

Meyers, G., Sandler, C. and Badgett, T. (2012) *The Art Of Software Testing*.

Mocha (2023) Available at: https://mochajs.org/.

MrDevSecOps (2023) *Docker Objects*. Available at: https://medium.com/@mrdevsecops/docker-objects-e561f0ce3365.

Nevedrov, D. (2006) 'Using JMeter to Performance Test Web Services', *Dev2Dev*, pp. 1–11. Available at: http://loadstorm.com/files/Using-JMeter-to-Performance-Test-Web-Services.pdf.

Nicola, S. (2023a) 'Análise de Valor'.

Nicola, S. (2023b) 'Análise de Valor: FAST and QFD Techniques'.

Oates, B. (2005) *Researching Information Systems and Computing*.

Pereira, N. (2022a) 'Métodos de Investigação I'.

Pereira, N. (2022b) 'Métodos de Investigação II'.

Pratomo, A.E., Schriek, E. van der and Veen, T. van der (2020) 'Test Driven Development in OWOW's Full-stack Web Development', *International Journal of Industrial Research and Applied Engineering*, 4(2), pp. 46–50. Available at: https://doi.org/10.9744/jirae.4.2.46-50.

Rafa E. Al-Qutaish, P. (2010) 'Quality Models in Software Engineering Literature: An Analytical and Comparative Study', *Journal of American Science*, 6(3), pp. 166–175. Available at: http://www.jofamericanscience.org/journals/am-sci/am0603/22_2208_Qutaish_am0603_166_175.pdf.

Rafi, D.M. *et al.* (2012) 'Benefits and limitations of automated software testing: Systematic literature review and practitioner survey', *2012 7th International Workshop on Automation of Software Test, AST 2012 - Proceedings*, pp. 36–42. Available at: https://doi.org/10.1109/IWAST.2012.6228988.

Rai, P. and Dhir, S. (2015) 'A Prologue of JENKINS with Comparative', pp. 5–9.

Rossel, S. (2017) *Continuous Integration, Delivery, and Deployment Reliable and faster software releases with automating builds, tests, and deployment*. Packt.

Rudrabhatla, C.K. (2020) 'Comparison of zero downtime based deployment techniques in public cloud infrastructure', *Proceedings of the 4th International Conference on IoT in Social, Mobile, Analytics and Cloud, ISMAC 2020*, pp. 1082–1086. Available at: https://doi.org/10.1109/I-SMAC49090.2020.9243605.

Selenium (2023) Available at: https://www.selenium.dev/documentation/.

Shahin, M. *et al.* (2017) 'Beyond Continuous Delivery: An Empirical Investigation of Continuous Deployment Challenges', *International Symposium on Empirical Software Engineering and Measurement*, 2017-Novem, pp. 111–120. Available at: https://doi.org/10.1109/ESEM.2017.18.

Shahin, M., Ali Babar, M. and Zhu, L. (2017) 'Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices', *IEEE Access*, 5(Ci), pp. 3909–3943. Available at: https://doi.org/10.1109/ACCESS.2017.2685629.

Singh, I. and Tarika, B. (2014) 'Comparative Analysis of Open Source Automated Software Testing Tools: Selenium, Sikuli and Watir', *International Journal of Information & Computation Technology*, 4(15), pp. 1507–1518. Available at: https://doi.org/10.13140/2.1.1418.4324.

Smartbear (2023) *What Is Unit Testing?* Available at: https://smartbear.com/learn/automated-testing/what-is-unit-testing/.

Sonatype (2023) *Nexus*. Available at: https://www.sonatype.com/products/sonatype-nexus-repository.

Staegemann, D. *et al.* (2019) 'Exploring the specificities and challenges of testing big data systems', *Proceedings - 15th International Conference on Signal Image Technology and Internet Based Systems, SISITS 2019*, pp. 289–295. Available at: https://doi.org/10.1109/SITIS.2019.00055.

Terraform (2023) *Terraform*. Available at: https://www.terraform.io/.

Yang, A., Troup, M. and Ho, J.W.K. (2017) 'Scalability and Validation of Big Data Bioinformatics Software', *Computational and Structural Biotechnology Journal*, 15, pp. 379–386. Available at: https://doi.org/10.1016/j.csbj.2017.07.002.

Yang, B. *et al.* (2018) 'Service discovery based blue-green deployment technique in cloud native environments', *Proceedings - 2018 IEEE International Conference on Services Computing, SCC 2018 - Part of the 2018 IEEE World Congress on Services*, pp. 185–192. Available at: https://doi.org/10.1109/SCC.2018.00031.

Zaiku (2017) *Continuous Delivery In a Nutshell*. Available at: https://zaiku.medium.com/continuous-delivery-in-a-nutshell-29f4213dabda.