UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

DEV OPS IMPLEMENTAÇÃO DO MODELO DEVOPS EM EQUIPA DE DESENVOLVIMENTO

Artiom Andronic

Mestrado em Engenharia Informática

Trabalho de Projeto orientado por:
Prof. Doutor Pedro Alexandre de Mourão Antunes

2022

Agradecimentos

Ao Prof. Doutor Pedro Alexandre de Mourão Antunes, que, como orientador deste projeto por parte da faculdade, ajudou-me na estruturação e desenvolvimento do documento. Muito obrigado pelo apoio. Ao Eng. Simão Ferreira, que como meu orientador e responsável pelo projeto por parte da empresa, mostrou-se sempre disponível para ajudar nas fases mais difíceis do desenvolvimento do projeto. Sem a sua ajuda e partilha de conhecimentos tudo seria mais difícil. À Ana Guimarães que me deu esta oportunidade e sempre se mostrou disponível para me auxiliar nos tempos mais difíceis. Queria agradecer especialmente toda a sua generosidade, paciência e compreensão. Quero agradecer também a todos os outros membros da TrustSystems pelo acolhimento e ajuda durante o projeto.

Por último quero agradecer à minha família e amigos que me suportaram em toda esta jornada. Sem eles constantemente a me motivarem e indicarem o caminho certo nada disto seria possível. Obrigado por tudo.

Dedicatória.

Resumo

Cada vez mais as pipelines de DevOps estão a evoluir implementando novas técnicas e etapas que garantam que o software é entregue em produção não só com rapidez, mas também com qualidade, resultando em software que passa por várias camadas de testes e verificações antes da entrega, de forma a garantir confiabilidade e estabilidade do mesmo.

Neste estudo de DevOps, que dá ênfase ao problema de automatizar uma fase de testes numa estrutura DevOps, foram realizadas análises da integração de testes automáticos nas pipelines DevOps onde são mostradas as possíveis vantagens desta integração. Posteriormente é analisada uma possível solução que integra testes automáticos dentro de uma estrutura existente, juntamente com uma reflexão acerca dos resultados alcançados.

O estudo foi realizado na empresa TrustSystems, cujo principal objetivo é a integração de uma etapa de testes, totalmente automáticos, na estrutura de desenvolvimento de projetos de software, em que estes verificam principalmente se a qualidade de código do software é aceitável e se este cumpre os requisitos do projeto, cada vez que é entregue numa pipeline de DevOps. Esta nova etapa na estrutura de desenvolvimento deve ser capaz de interromper o processo de entrega de software na pipeline caso o código entregue não cumpra os requisitos definidos pelo quality gate, posteriormente deve notificar as entidades responsáveis dos efeitos ocorridos.

Este estudo é de grande relevância, uma vez que mostra o processo da alteração de uma estrutura de desenvolvimento DevOps bem constituída, para uma estrutura mais moderna que visa entregar código com mais qualidade focando-se na sua estabilidade, confiabilidade e manutenibilidade.

Palavras-chave: DevOps, Testes Contínuos, Quality Gates, Testes Automatizados, Pipeline

Abstract

Increasingly, DevOps pipelines are evolving implementing new techniques and steps that ensure that software is delivered into production not just quickly, but also with better quality, resulting in software that passes through multiple layers of testing and verification before delivery, to guarantee its reliability and stability.

In this DevOps study, which emphasizes the problem of automating a testing phase in a DevOps structure, analyzes the integration of automatic tests in DevOps pipelines, where the possible advantages of this integration are shown. Subsequently, a possible solution that integrates automatic tests within an existing structure is analyzed, together with a reflection on the results achieved.

The study was carried out in the company TrustSystems, whose main objective is the integration of a stage of tests, totally automatic, in the structure of development of software projects, in which they mainly verify if the quality of the code of the software is acceptable and if it complies the project requirements, each time it is delivered in a DevOps pipeline. This new stage in the development structure must be able to interrupt the software delivery process in the pipeline if the delivered code does not meet the requirements defined by the quality gate, later it must notify the responsible entities of the effects that have occurred.

This study is of great relevance, as it shows the process of changing a well-constituted DevOps development structure to a more modern structure that aims to deliver code with better quality, focusing on its stability, reliability, and maintainability.

Keywords: DevOps, Continuous Testing, Quality Gates, Automated Testing, Testing Metrics

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas

API *Application Programming Interface*

CD *Continuous Delivery and Continuous Deployment*

CI *Continuous Integration*

CT *Continuous Testing*

Dev Desenvolvimento de software

DNS *Domain Name System*

HTTP *Hypertext Transfer Protocol*

IDE Ambiente de desenvolvimento integrado

IEC *International Electrotechnical Commission*

ISO *International Organization for Standardization*

Json *JavaScript Object Notation*

KPI *Key Perfomance Indicators*

Ops Operações

QA *Quality Assurance*

REST *Representational State Transfer*

TLS *Transport Layer Security*

Capítulo 1

Introdução

Hoje em dia são poucas as empresas de desenvolvimento de software que não possuem uma pipeline de DevOps[19] para entrega de software em produção. As vantagens de ter uma pipeline DevOps são grandes, pois reduzem bastante o tempo de entrega de software, em geral são relativamente fáceis e simples de implementar e têm ainda o efeito de reduzir o potencial de conflito entre os membros de equipas de desenvolvimento[38]. Com o passar do tempo houve a necessidade de evoluir estas pipelines de forma a que estas não só entreguem o software rapidamente, mas também entreguem software de qualidade, estável e confiável. DevOps é um pipeline com vários processos, um destes processos é o *Continuous Testing* (CT) que veio integrar-se nas pipelines para garantir que estes requisitos de entregas sejam cumpridos, mas impôs-se igualmente a necessidade de equilibrar o tempo de desenvolvimento face à qualidade do código, pois o CT pode vir a ser uma atividade complexa em que o tempo de desenvolvimento de testes é demorado, algo que muitas empresas vêm como desvantagem.

As pipelines DevOps podem ser sempre melhoradas com a integração de processos automáticos que melhoram o processo de desenvolvimento e entrega de software. Este processo levanta desafios, principalmente, o desafio de redesenhar as estruturas destas pipelines DevOps. Especialmente numa pipeline já existente, para melhorá-la é preciso fazer análises às arquiteturas e tecnologias utilizadas e ao perceber exatamente como a pipeline funciona, trata-se então de coletar todos os requisitos e restrições, para se fazer uma boa escolha das tecnologias a serem inseridas dentro da pipeline.
Posteriormente levanta o desafio de garantir que a integração do CT dentro da estrutura DevOps é feita de tal maneira que este processo não interfere com a automatização e continuidade da estrutura.

DevOps é uma pipeline com vários processos(*Continuous Integration* (CI), *Continuous Delivery and Continuous Deployment* (CD)), a integração desses processos é cada vez mais complexo, visto que existem imensos conceitos e tecnologias, que todos os dias

são atualizadas, com o intuito de melhorar as pipelines e tornar o processo de entrega de software mais simples e de melhor qualidade. Especialmente com a instituição de requisitos mínimos que definem a qualidade de código para o desenvolvimento de certos tipos de software. Cada vez mais estes requisitos de qualidade são instituídos em softwares, o que significa que muitas empresas estão a começar a mudar as suas pipelines para integrar o CT[38].

1.1 Objetivos

O principal objetivo deste estudo é o estudo da integração de CT na estrutura de DevOps atual de uma empresa. Esta etapa deve incluir testes, totalmente automáticos, aos quais o software possa ser submetido. Estes testes devem garantir não só que o código do software é de qualidade, mas também devem verificar se o mesmo cumpre os requisitos originalmente desenhados no projeto. A qualidade de código e verificação dos requisitos trará vantagens como confiabilidade, estabilidade, segurança e satisfação do cliente. O processo de CT será feito através de ferramentas que atuam como agentes que irão verificar o código e as funcionalidades do software, estas ferramentas permitem também a utilização de quality gates. Estes quality gates[29] são os que definem os requisitos de qualidade do código entregue e com eles integrados em ferramentas da pipeline DevOps, estes quality gates terão a capacidade de interromper o processo de entrega de software caso o código não cumpra os seus requisitos.

1.2 Ambiente e contexto do projecto

Este estudo apresenta o processo de implementação de uma nova etapa de testes automáticos dentro de uma estrutura DevOps já existente. Esta implementação foi feita na TrustSystems, uma empresa que atua na área da Segurança da Informação, especializada no fornecimento de soluções de modo a garantir a segurança dos ativos de informação. A sua atuação abrange quatro principais áreas: proteção de informação, serviços de segurança, soluções de parceiros e I&D. Visto que, a segurança é uma das áreas desta empresa, faz todo o sentido a inserção desta nova etapa na sua arquitetura, que por certo garantirá que o código desenvolvido é entregue com mais estabilidade e confiabilidade.
A TrustSystems está integrada no grupo Inoweiser, cujo principal objetivo é usar a tecnologia e a inovação para satisfazer as necessidades dos seus clientes. Tem mais de quinze anos de experiência, tanto em mercados nacionais como internacionais, estando presente em mais de 20 países.

A TrustSystems é o exemplo de uma empresa que se deparou com o desafio de integrar CT na sua estrutura DevOps por requisito de um cliente no seu mais recente projeto de desenvolvimento de software. Visto que todos os seus processos de testes ao software até ao momento são feitos de forma manual, existe um grande potencial na integração desta nova etapa na sua arquitetura. Posteriormente dada a grande importância que dão à segurança de software, esta será sem dúvida uma boa maneira de aumentar a confiabilidade do seu código.

1.3 Estrutura do documento

Este estudo irá começar por descrever em algum detalhe o conceito de DevOps, a sua história, para que foi criado, e no que consiste uma pipeline DevOps. De seguida são introduzidos os conceitos de (Continuous integration) CI/ (Continuous delivery and continuous deployment) CD tal como o conceito de CT. Dentro da definição de CT encontramos a listagem das técnicas de testagem de software atualmente mais utilizadas e reconhecidas como mais eficientes. Posteriormente encontraremos a definição de o que é um quality gate e por fim uma argumentação acerca de até onde se deve testar. Depois de o conceito estar explicado passamos para a fase do documento em que se descreve o problema em si que está associado com a arquitetura da empresa. Apresentado o problema, é então discutida essa solução, onde é mostrada a nova arquitetura, as ferramentas que vão ser utilizadas para a solução e argumentação de como devem ser utilizadas e integradas. Finalmente é descrito todo o processo de desenvolvimento da dita solução juntamente com uma reflexão sobre os resultados obtidos.

Capítulo 2

Metodologia

Neste estudo foi adotado a organização do documento problema, intervenção e avaliação (ADR) sugerida por [30]. Esta organização de documento foi adotada por que envolve o design de artefactos, em que neste caso o artefacto alvo é a inserção do processo de CT na estrutura DevOps atual de uma empresa.
De acordo com a metodologia ADR, o estudo foi estruturado nas seguintes fases:

1. A primeira parte do projeto focou-se em fazer uma pesquisa ao estado da arte do DevOps, procurando perceber os seus principais conceitos e perceber o que era e como funciona uma pipeline DevOps. Aqui foram introduzidos os conceitos de CI e CD, onde foi feita a sua descrição e a razão por estas serem úteis. Como o problema tem o seu foco na testagem do código e na sua automatização, foi feita uma pesquisa mais aprofundada do conceito de CT, as técnicas mais utilizadas de testagem e como são feitos os controlos de qualidade dentro das pipelines.

2. Depois de se entender o que é o estado da arte do DevOps, na segunda fase, foi feito o design da solução, num domínio abstrato, onde se formalizou e se dissecou o problema, mostrando a sua estrutura e aquilo que se pretende exatamente alcançar com esta solução. Este é um dos passos mais importantes neste projeto, porque foca-se em entender com precisão aquilo que é necessário fazer para solucionar o problema. É feita ainda nesta fase uma reflexão da solução abstrata que reflete sobre a sua exequibilidade, generalização, reutilização, flexibilidade e impacto.

3. A terceira fase, foca-se na implementação da solução no seu domínio concreto. Que começa com a pesquisa das tecnologias utilizadas para a resolução da solução. Esta pesquisa foi feita com base em soluções já existentes no mercado atual de DevOps nos processos de CT. Após a escolha das ferramentas, é feita uma análise das tecnologias utilizadas pela empresa, tal como é feita uma percepção dos ambientes por esta utilizados e algumas das suas práticas.

Posteriormente é apresentada o desenvolvimento concreto da solução que foi dividida em duas iterações. Aqui é apresentado tudo aquilo que foi desenvolvido e implementado, tal como a justificação para algumas das escolhas tomadas.

4. Finalmente, a quarta e última fase é a avaliação da solução no seu domínio onde mostra-se e avalia-se os resultados concretos do desenvolvimento.

Capítulo 3

DevOps: Estado da arte

O DevOps é comumente considerado (Figura 3.1) uma interseção entre os âmbitos de Desenvolvimento de software (Dev), Operações (Ops) e *Quality Assurance* (QA). Apesar da sua atual popularidade esta área da informática não tem uma definição concreta[19] e [31]. É descrito, por muitos, como uma mudança cultural que empresas de informática tomaram com o objetivo de remover ou reduzir a barreira tecnológica entre as equipas de Dev e Ops e permiti-los colaborar com responsabilidades partilhadas [35].

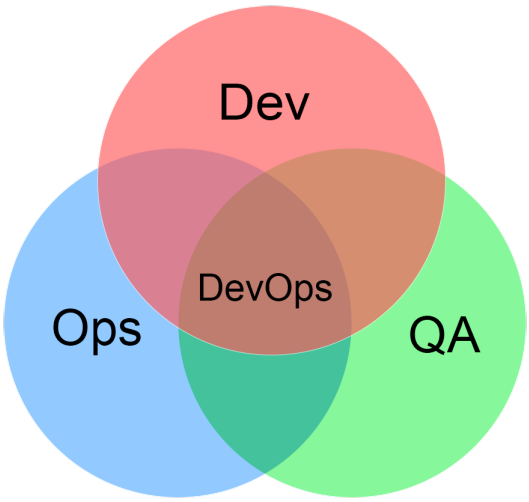

Figura 3.1: Escopo do DevOps [28]

A área de Dev tende a considerar que mudança é o que realmente importa para que o

software esteja ativo no mercado, e que isso incentiva a atualizar o seu software continuamente. Apesar de novas funcionalidades e mudanças para métodos mais estado da arte serem apelativas para o Dev, a área de Ops vê a mudança como um inimigo que aponta prejudicar a estabilidade e confiabilidade do software. Os Ops têm a necessidade de ver que é o software estável que gera benefício, e veem estas mudanças como uma maneira de prejudicar esse rendimento. Ambos têm perspetivas diferentes do que gera valor a uma empresa, e tecnicamente estão ambos corretos.

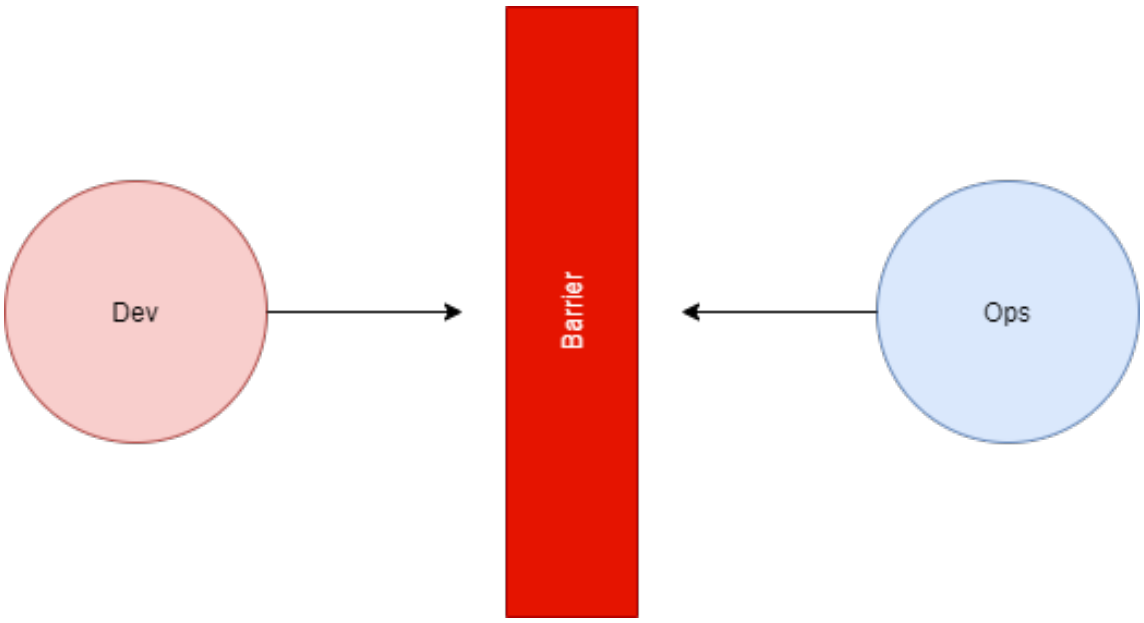

Figura 3.2: Barreira de interresses entre Dev e Ops[38]

Estes conflitos de interesses tendem a criar uma barreira entre estas áreas de informática (Figura 3.2). E é aqui que surgiu o DevOps, que pode ser visto como um conjunto de práticas e ferramentas cujo intuito é simultaneamente quebrar esta barreira entre Dev e Ops e agilizar o processo de entrega de software.

Um conceito importante na área de DevOps é a agilidade, a adopção de técnicas ágeis. Esta agilidade em DevOps faz com que todo o processo de entrega de software seja alinhado, o que significa que o software desde o momento em que está a ser escrito até ser entregue irá passar várias etapas (Figura 3.3) numa ordem específica que assegura que todo este processo seja entregue rapidamente e no seu melhor estado. Este processo traz várias vantagens numa perspetiva de negócio pois reduz o tempo de entrega do software, resultando numa redução de custo e assegura que o software entregue está num estado estável. A maioria da estratégia ágil é feita na primeira etapa que é a de planificação e após ser concluído o ciclo deve-se sempre começar pela planificação da resolução dos problemas detetados.

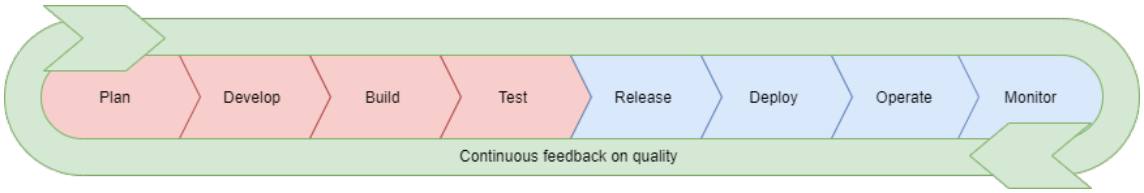

Figura 3.3: Etapas em DevOps baseado em[38]

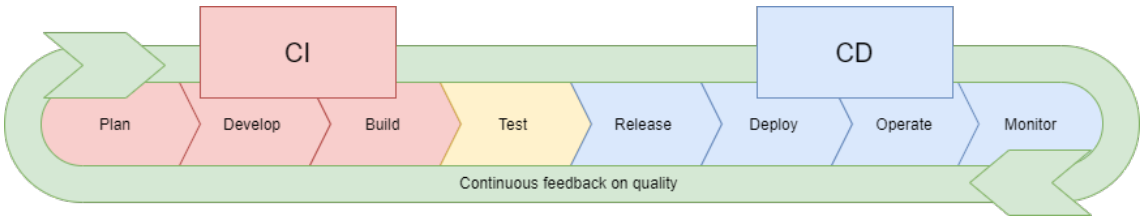

Figura 3.4: CI e CD em DevOps baseado em[38]

Estas etapas podem ser divididas em duas grandes partes, CI – Continuous Integration e CD – Continuous Delivery e Continuous Deployment (Figura 3.4).

3.1 Continuous Integration (CI)

O CI consiste numa forma de desenvolver código ao mesmo tempo em diferentes recursos na mesma aplicação sem terem conflitos. Antigamente as equipas de Dev tinham um dia denominado de "*Merge Day*" que servia para unificar as alterações e os progressos da equipa num local único, este processo era feito manualmente tornando-o entediante e demorado. As ferramentas de CI vieram ultrapassar estes problemas fazendo as equipas frequentemente publicar as suas alterações em ramificações de um repositório partilhado pela equipa em que se integram. Estas ferramentas que gerem o repositório partilhado permitem validar o código verificando se eles têm erros ou se não estão corrompidos, verificando ainda se não existe nenhum conflito nessa junção entre os integrantes das equipas, permitindo, caso haja conflitos, que estes sejam resolvidos de maneira fácil e rápida.

3.2 Continuous delivery e continuous deployment (CD)

Com uma boa base de CI integrada, o próximo passo é
automatizar o processo de unificar o código das várias ramificações num ramo principal do repositório. A isto chama-se Entrega Contínua (Continuous Delivery), cujo foco é garantir que a qualquer momento que existe uma base de código pronta para ser entregue em produção. Aqui é então introduzida a Implementação Contínua (Continuous Deployment) que é a automatização da implementação do código em produção. Esta é a parte

final do que se chama uma pipeline CI/CD.

Uma vez em produção, o software é monitorizado por ferramentas, como o Nagios e o Jhipster para detetar erros e anomalias durante a execução. Aqui serão avaliadas as condições do software em produção e caso seja detetado algo negativo no decorrer da aplicação será reportado e o ciclo volta ao início para que os problemas sejam resolvidos. Outra razão para reiniciar este ciclo será por exemplo adicionar novas funcionalidades ou alterar funcionalidades que não tenham ainda passado os testes de aceitação.

3.3 Continuous testing (CT)

Todo este processo de DevOps deve ser feito meticulosamente para que todo o código que é lançado esteja no nível de qualidade adequado, evitando que código com erros seja inserido na pipeline. Surge assim um outro conceito importante em DevOps que é o de Continuous Testing (CT). Desde o momento em que o código está a ser escrito até ao momento em que código é lançado em produção devem existir testes automatizados que aferem o código em vários aspetos. Como mencionado em [38] estes testes devem fazer um "*Shift left*" e um "*Shift right*" (Figura 3.5) incluindo quality gates que definam *Key Perfomance Indicators* (KPI) para que o software esteja estável.

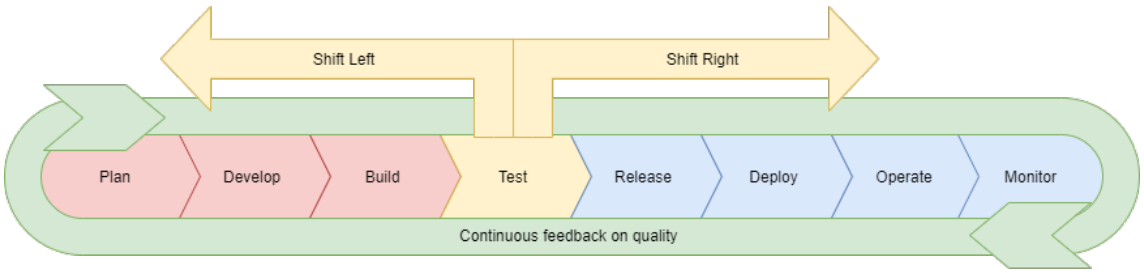

Figura 3.5: CT (*Continuous Testing*) baseado em[38]

3.3.1 Conceito

O CT deve começar quando o código ainda está a ser escrito e deve acabar apenas antes de o código ser entregue. Como visto na (Figura 3.5) existe um conceito [38] que diz que a fase de testes do software num ciclo de DevOps deve abranger quase o ciclo inteiro e não deve ser uma fase só em si. É definido o "*Shift left*" e o "*Shift Right*" que significa como a imagem indica que a fase de testes deve ser dividida pelas outras fases para garantir uma área maior de testes. Se o modelo de DevOps definido na Figura 3.3 fosse cumprido à risca a fase de testes apenas iria englobar testes ao código em si, através de testes unitários e testes de análise estática de código. Hoje em dia com as várias tecnologias

emergentes no mercado de software testes ao código apenas são capazes de cobrir parte dos problemas. Novas normas sugerem que testes de integração, testes de sistema e testes de performance sejam feitos para assegurar a melhor qualidade de software possível [38].

Resumidamente, "*Shift left*" e o "*Shift Right*" vêm definir um conceito segundo o qual os testes devem ser feitos durante ou após cada etapa. Por exemplo durante a escrita de código um desenvolvedor poderá usufruir de linters que poderão ser adicionados como *plug-ins* ao Ambiente de desenvolvimento integrado (IDE) que irão em tempo real indicar possíveis erros no código, assim o desenvolvedor poderá evitar entregar código defeituoso. Mesmo assim, os linters apenas testam por erros de código através de análise estática, os desenvolvedores devem recorrer ao uso de testes unitários para testar funções que achem que têm uma complexidade elevada. Após este código ser entregue deve existir um *quality gate* que defina KPIs, ou alguns requisitos mínimos para garantir que o código é digno de ser passado para a etapa seguinte. Depois de passar os *quality gates* devem ser feitos testes de integração que verificam se todos os componentes integrantes do produto final estão a comunicar corretamente entre eles. Quando a verificação é concluída deve-se então testar o sistema como um todo utilizando scripts que enviem informação numa sequência através do *frontend* para o *backend* verificando as respostas entregues pelo *backend* no *frontend*. E para finalizar as etapas de testes, geralmente, é requisitado um relatório de performance ou até mesmo um tempo de resposta por x utilizadores em simultâneo e para tal ainda são realizados os testes de desempenho.

3.3.2 Técnicas de testagem

Esta secção é dedicada à especificação de um conjunto de técnicas de testagem de software.

Testes de revisão de código Surgiram inicialmente como uma atividade da equipa de QA, onde alguns membros da equipa verificavam o software através da análise de partes do código fonte. Quando esta atividade era realizada o autor do código não podia ser a única pessoa a analisar o código. O objetivo principal desta atividade era encontrar diretamente problemas de qualidade no código [14], no entanto, com o passar do tempo, esta atividade passou a ter como objetivo cumprir uma série de objetivos que, se cumpridos, asseguram qualidade [10][15]:

- Melhor qualidade de código – melhora a qualidade interna do software e reduz o tempo de manutenção do mesmo por melhorar a leitura e a perceção do código.

- Encontra defeitos – melhora a qualidade externa do software e encontra problemas de performance, *bugs* e vulnerabilidades.

- Encontra melhores soluções – ao rever código, geralmente, surgem novas ideias e novos pensamentos de como melhorar a solução atual.

Hoje em dia esta atividade é feita através do uso de ferramentas de análise estática de código, conhecidas por *Linters* ou *Linting tools* que fazem *Linting*.

O *Linting* é a procura de erros de programação no código fonte através de análises automáticas ao mesmo. As ferramentas procuram erros de programação que, muitas vezes, não são detetados pelos compiladores. Procuram estes erros com base em regras das linguagens em causa e uma base de conhecimento que vai sendo atualizada ao longo do tempo, à medida que novos erros vão sendo reportados.

Os erros detetados por estas ferramentas são maioritariamente de 2 tipos: *bugs* ou *code smells*. *Bugs* são erros de programação que podem causar a uma má execução do programa, e que pode até levar à interrupção indesejada do mesmo [26]. *Code smells* são características no código que podem indicar problemas mais profundos no programa. Determinar o que é um *code smell* é subjetivo, varia bastante com a linguagem, o desenvolvedor e até mesmo o método de desenvolvimento [33]. Um *Linter* deteta estes *code smells* muitas vezes como maus hábitos de programação, por exemplo, um *try/catch* de uma exceção genérica ao invés de uma exceção específica. Outro exemplo são *code smells* que podem causar vulnerabilidades que possam ser exploradas por um utilizador maligno, um exemplo notável destes erros são por exemplo o uso da função *printf()* em C, que usada indevidamente pode ser explorada para criar *buffer overflows*, que podem consequentemente terminar a execução do programa, expor os conteúdos em memória e até mesmo manipular a memória de maneira a tornar a execução do programa indesejada.

Apesar de ser eficaz em várias linguagens de programação fazer *linting* é muito mais recomendado para linguagens que são interpretadas (não compiladas) como, por exemplo, o python e javascript. Ao não serem compiladas, erros de sintaxe podem ser corridos, resultando em más execuções de programas onde o erro não é discriminado. Nestas linguagens um *Linter* não vai apenas procurar maus hábitos de código ou erros que não são detetados pelo compilador, mas igualmente erros de sintaxe.

Testes unitários São um método de testagem de software, bastante popular entre desenvolvedores de código, onde unidades individuais de código são testadas para determinar se as mesmas são dignas de ser utilizadas [24], se correspondem ao seu design e se comportam adequadamente [23]. Dependendo do paradigma de programação uma unidade pode ser considerada como uma classe ou um módulo inteiro ou mais comummente uma função ou método individual [37].

De maneira a isolar uma unidade do seu contexto, desenvolvedores geralmente utilizam esboços, simulações e falsificação de objetos e *inputs* de maneira a assistir e criar um contexto para que as funções sejam testadas em vários casos. Podem também recorrer ao uso de testes parametrizados que permitem a execução de múltiplos testes com *inputs* diferentes.

O grande objetivo e a grande vantagem dos testes unitários são a capacidade de isolar cada uma das unidades separadamente e mostrar sob a forma de *log* quais dessas unidades estão corretas e quais estão incorretas. De outro ângulo um teste unitário pode ser visto como um acordo que uma unidade deve cumprir [24].

Resumindo, os testes unitários são uma grande forma de desenvolvedores encontrarem cedo eventuais problemas no código, consequentemente, estes problemas são também resolvidos cedo. O processo de criar um teste unitário também é benéfico para o desenvolvimento do código visto que os desenvolvedores terão de pensar nos possíveis *inputs*, nos *outputs* e nas condições de evitar erros à partida, criando assim código de qualidade numa fase inicial [9].

É de notar que os testes unitários não vão resolver todos os problemas de um programa, isto porque é muito difícil avaliar e percorrer todos os caminhos de execução do mesmo. Aqui surge uma métrica denominada *code coverage* ou *coverage* que mede, em percentagem, a quantidade de código (linhas de código) coberta por testes, sejam estes unitários ou de outro tipo. Um programa com uma grande percentagem de *coverage* sugere que contém menor percentagem de bugs não detetados comparando com um programa com uma *coverage* baixa [16].

Testes de integração São geralmente utilizados logo após os testes unitários e antes dos testes de sistema. O foco dos testes de integração é agregar módulos individuais, definidos nos testes unitários, e testá-los como um grupo. Aqui a intenção é avaliar a conformidade de um componente do sistema com os seus requisitos funcionais [3].

O principal objetivo desta fase é agregar os módulos individuais em grupos, criar uma série de testes que verificam se os requisitos são cumpridos e após verificado indicar no seu *output* se o sistema está pronto para testes de sistema, ou se existe alguma não conformidade em algum dos grupos e indicar onde e qual requisito não é cumprido [27].

Estes testes são muito benéficos para a entrega de software estável. Muitas vezes os desenvolvedores tendem a alterar classes ou estruturas comuns do programa para solucionar um problema único sem se preocupar com o resto do programa, daí resultando que

o problema em causa seja solucionado, mas pode vir a ter um impacto forte no resto das execuções do programa. Ser notificado que uma alteração no módulo x teve impacto na execução da função y é crucial para garantir que o programa que é entregue segue numa linha consistente especialmente quando o desenvolvimento de um software é feito por uma equipa de desenvolvedores.

Testes de sistema São uma técnica de testagem que avalia um sistema totalmente integrado e completo de uma ponta à outra, verificando se o fluxo do software é o esperado. Ultrapassados estes testes podemos afirmar que todas as dependências e todos os sistemas integrados funcionam da maneira expectável [17].

O principal propósito dos testes de sistema é testar a aplicação na perspetiva de um utilizador final, simulando cenários realistas e verificando, se todos os componentes e todas as unidades estão de acordo com a expectativa [17]. Para além de testar um sistema na totalidade, os testes de sistema podem ser utilizados também como testes de aceitação, testes que verificam se o software (neste caso) cumpre com os seus requisitos e especificações contratuais, e ainda podem ser utilizados para correr testes de regressão, que na sua essência são testes que servem para assegurar que todo o software desenvolvido funcione mesmo após uma mudança [13].

Os testes de sistema são feitos através de softwares que correm *scripts* separados numa sequência específica de forma a simular o fluxo de um utilizador, estes softwares geralmente atuam diretamente no *frontend* do software e verificam as respostas. A tarefa de separar o processo em vários *scripts* decorre da necessidade de conseguir obter informação mais detalhada sobre os problemas para que possam ser solucionados rapidamente. Um exemplo desta separação seria criar um *script* que simula um *login* numa página web e outro script a alteração da palavra-chave do utilizador, ao correr estes *scripts* numa sequência poderemos sempre saber onde esteve o problema, ou se no *login* ou se na alteração da sua palavra-chave, neste caso.

Com a complexidade dos sistemas de hoje em dia, os testes de sistema trazem bastantes benefícios às equipas pois expandem a *test coverage*, asseguram que a aplicação é corrigida, detetam *bugs* e reduzem o *time to market* que consequentemente reduz o custo.

Testes de performance Consistem em determinar as capacidades de um sistema em termos de tempo de resposta e estabilidade sobre uma carga de trabalhos. Podem também ser utilizados para validar certas métricas importantes como escalabilidade, confiabilidade e a quantidade de recursos que são utilizados [34].

Existem várias técnicas para testar a performance e cada uma das técnicas tem o seu objetivo específico. Estas técnicas são nomeadamente: testes de carga, testes de stress, testes de imersão, testes de picos, testes de interrupção, testes de configuração, testes de isolamento e testes de internet [34]. Cada um destes testes tem o seu propósito, e são utilizados mediante os objetivos planeados pelas equipas para a performance do sistema, estes objetivos variam bastante de sistema para sistema e de utilização para utilização. Outro objetivo comum para serem feitos testes de performance é comparar dois ou mais sistemas que têm a mesma função a fim de concluir qual deles tem melhor performance. Esta utilização é mais comum para otimizar sistemas testando tecnologias diversas [36].

3.3.3 Quality gates

Até agora refirmos várias maneiras de testar o software, mas importa entender como integramos estes testes na pipeline DevOps? Isto é possível através de quality gates. Os quality gate são geralmente fornecidos por ferramentas de DevOps que se integram nas pipelines e funcionam como uma porta que controla a qualidade do software. O conceito é fácil de perceber, se o software cumprir com os requisitos de qualidade avança para as etapas de DevOps seguintes; se não, o processo CI/CD é interrompido e o software não avança [29].

A integração de quality gates é essencial para garantir a qualidade do software. Atualmente, a entrega de software tem-se focado mais com a rapidez e não tanto na qualidade e eficácia do software, geralmente levando a entregas de software com *bugs* que estragam a experiência dos utilizadores. Os quality gates vieram integrar-se bastante bem no dia a dia das equipas de software pois funcionam de uma maneira rápida e simples de verificar a qualidade do software. No entanto existe uma desvantagem nos quality gates face ao tempo de entrega/ tempo de entrada em produção aumentando o custo imediato percecionado do software [29].

Sendo clara a relevância dos *quality gates* na verificação de qualidade do software, importa igualmente referir que os próprios *quality gates* têm de ser verificados. Para tal é preciso criar um plano de testes e uma utilização de ferramentas cujos *outputs* sejam relatórios que possam ser analisados e processados pelos *quality gates*. Um exemplo de aplicação de um *quality gate* pode ser definir que o software tem de ter testes unitários suficientes para cobrir 50% das linhas de código, se tiver um número superior a 50% é entregue, se não é parado o processo de entrega e é solicitado mais um conjunto de testes unitários aos desenvolvedores. Quanto mais se investir em testes, maior a qualidade do software na entrega, no entanto também é maior o tempo de entrega. E aqui é importante ponderar o *trade off* entre qualidade de software e tempo de entrega do mesmo [29].

3.3.4 Síntese

Mesmo com a ajuda das novas tecnologias e ferramentas de testes, integrar CT numa estrutura de DevOps pode ser desafiante e consumir muito tempo e a realidade é que não existe uma maneira correta ou mais correta que outra de testar software. Aqui cria-se um debate nas equipas que é, o que se deve testar e como se deve testar. A resposta a esta pergunta depende de software para software, de equipa para equipa e de ambiente para ambiente, mas em todos os casos deve ser feita uma análise cujo foco é responder a esta questão com base na análise de dados de sistema, sejam estes históricos ou atuais, e um inquérito às equipas de desenvolvimento, às equipas de operações e às equipas de QA [21]. Analisar os dados do sistema pode vir a iluminar o caminho para o sucesso do CT, no entanto existem certos objetivos que podem não ser necessários ou requerem um esforço adicional por parte das equipas que não é justificado face ao que se procura. Daqui se fazerem os inquéritos para assegurar que análise é real e possível.

Apontar para uma qualidade de software perfeita quase nunca justifica o esforço por parte das equipas. Vamos tomar o exemplo de testes unitários, estes geralmente são feitos pelos próprios desenvolvedores do código, uma análise de sistema pode determinar que 50% das falhas do sistema ocorrem devido a funções complexas que não são suficientemente testadas. Tal pode resultar numa análise que indica que 100% de cobertura de código é o necessário, no entanto 100% pode não vir a resolver este problema na totalidade e acentua significativamente o tempo de desenvolvimento do software para mais do dobro do tempo, resultando muito provavelmente em prejuízo. Daí a importância de ter um objetivo de cobertura menos exigente, ainda que relevante para deteção e correção da grande maioria dos possíveis erros. (na casa dos 50%, por exemplo).

Concluindo, caso não haja requisitos por parte de um cliente ou normas *International Organization for Standardization* (ISO)/*International Electrotechnical Commission* (IEC) que indiquem que o CT deve ser feito de uma maneira específica, perceber o que testar e como testar é bastante subjetivo e pode variar como referido acima com a equipa, o software e o ambiente.

Capítulo 4

Design da solução: domínio abstracto

Hoje em dia é mandatório ter uma pipeline de DevOps numa empresa de desenvolvimento de software, pois como referido no capítulo anterior acelera bastante o tempo de desenvolvimento e entrega do mesmo. Este capítulo é dedicado à definição e explicação do problema em causa.

4.1 Formalização do Problema

Muitas empresas já têm um pipeline de DevOps implementado na sua arquitetura, poucas são as empresas e os projetos que não usufruem destas tecnologias. Mas apesar de terem processos de CI/CD integrados na sua arquitetura, não são todas as empresas que têm CT integrado. Um destes casos é a TrustSystems que pretende com este estudo integrar e implementar CT na sua arquitetura atual para o seu mais recente projeto.

O problema concreto deste estudo é a automatização de um processo de testes dentro da estrutura DevOps da TrustSystems. Este problema implica que sejam integrados testes de revisão de código, testes unitários, testes de integração e testes de sistema, de uma forma contínua e automática dentro da estrutura DevOps. Devem ser utilizadas uma ou mais ferramentas que sejam capazes de verificar o sucesso dos testes, sejam capazes de gerar um relatório que identifique as razões das falhas e explique sucintamente os problemas ou erros. Finalmente estas ferramentas têm de ser capazes de implementar quality gates que consigam interromper os processos de CI/CD, ou por falta de qualidade do código, ou testes cujos resultados são negativos ou condições do código que não estejam a ser cumpridas.

4.1.1 Estrutura da entrega de software da empresa

Começando pelo início, como podemos observar na Figura 4.1, geralmente, na TrustSystems, os projetos de software começam pelo planeamento que consiste no desenho das

regras de negócio, APIs e modelos de dados, passam depois por várias fases de aprovação até que é definida uma versão que contemple todos os aspectos a definir e permita o avanço do ciclo. É criado então um novo projeto no Jira, um software de gestão, monitorização e acompanhamento de tarefas, para que se possam listar todas as tarefas principais que sejam necessárias no ciclo.

Após a fase de design e planeamento serem concluídas geralmente são criados sucessivamente um ambiente de desenvolvimento na *cloud* e os *scripts* necessários para um ambiente local. Este processo de criação de um ambiente de desenvolvimento consiste na criação de uma base de dados PostgreSQL, neste caso na *cloud*, a criação de um repositório de ficheiros no BitBucket gerido pela ferramenta Nexus Repository e através desta ferramenta adicionar uma outra ferramenta de *build* de software que neste caso é o Jenkins.

Relativamente ao ambiente local de desenvolvimento é necessária a criação de todos os *scripts* Docker para a criação dos containers necessários para criar este ambiente. Os containers Docker são isolados uns dos outros e agrupam as suas próprias configurações, bibliotecas e software [2]. O Docker também permite com facilidade que estes containers comuniquem entre si usando canais de comunicação [2]. Para gerir as redes destes containers, tanto na cloud como num ambiente local, é utilizado o traefik.

O próximo passo principal é avançar com o desenvolvimento propriamente dito, aqui o desenvolvimento divide-se em duas equipas distintas que trabalham em conjunto para a entrega de um produto final - as equipas de BE e as equipas de FE. As equipas de BE são as responsáveis por desenvolver todas as chamadas da API REST garantindo que quando o FE chama, através do protocolo *Hypertext Transfer Protocol* (HTTP), os resultados são provenientes da base de dados. O FE por outro lado é a equipa responsável por mostrar estes resultados no ecrã de uma forma organizada – é a camada com que o utilizador interage e tem de considerar quer o rigor quer a usabilidade.

O BE do projeto será desenvolvido em Java com a *framework* SpringBoot, uma *framework open-source* para a criação de aplicações autónomas de nível de produção que são executados na *Java Virtual Machine*. As suas grandes vantagens são a configuração automática e a capacidade de criar aplicações autónomas [20]. A equipa de BE utiliza a base de dados e um ficheiro OpenAPI para a geração de classes na *framework* de maneira a manterem a consistência e eficácia no desenvolvimento do produto. Utilizam uma abordagem por camadas para o desenvolvimento, essas camadas são o *Rest*, *Data* e *Persistence*. O *Rest* é responsável pela saída e entrada de dados, o *Data* pelas validações e decisões e o *Persistence* pela comunicação com a base de dados, seja esta para guardar ou

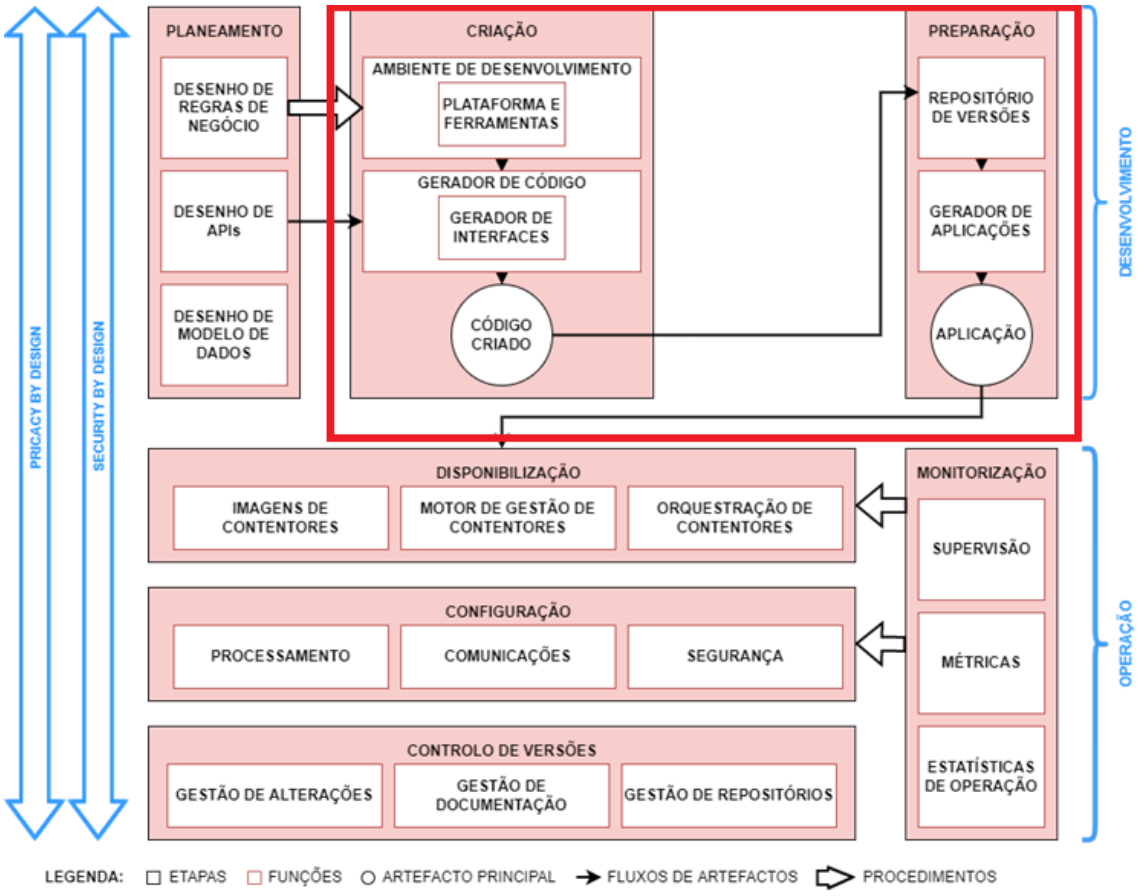

Figura 4.1: Arquitetura atual da TrustSystems

obter dados.

Para o desenvolvimento do código as equipas de BE utilizam IDE Intellij idea da JetBrains, e as equipas de FE preferem utilizar Microsoft Visual Studio Code para desenvolver Angular (uma *framework* com base em TypeScript), CSS e HTML. A equipa de BE ainda utiliza o Postman para a verificação e validação dos resultados da API REST através da chamada dos pedidos HTTP diretamente, analisando o resultado em *JavaScript Object Notation* (Json).

Em síntese, o projeto vai se basear numa aplicação web, com uma API REST que irá ser desenvolvida em Java Spring Boot, uma base de dados PostgreSQL e o seu frontend irá ser desenvolvido em Angular, CSS e HTML.

4.2 Design, intervenção e avaliação

Como observamos na Figura 4.2 a nova estrutura inclui, logo na etapa de criação, dois agentes, um agente de verificação e um agente de validação. E acrescenta uma nova

etapa, denominada de verificação (CT), que inclui um agente de controlo de qualidade. Esta arquitetura adota a filosofia de código candidato e código aprovado.

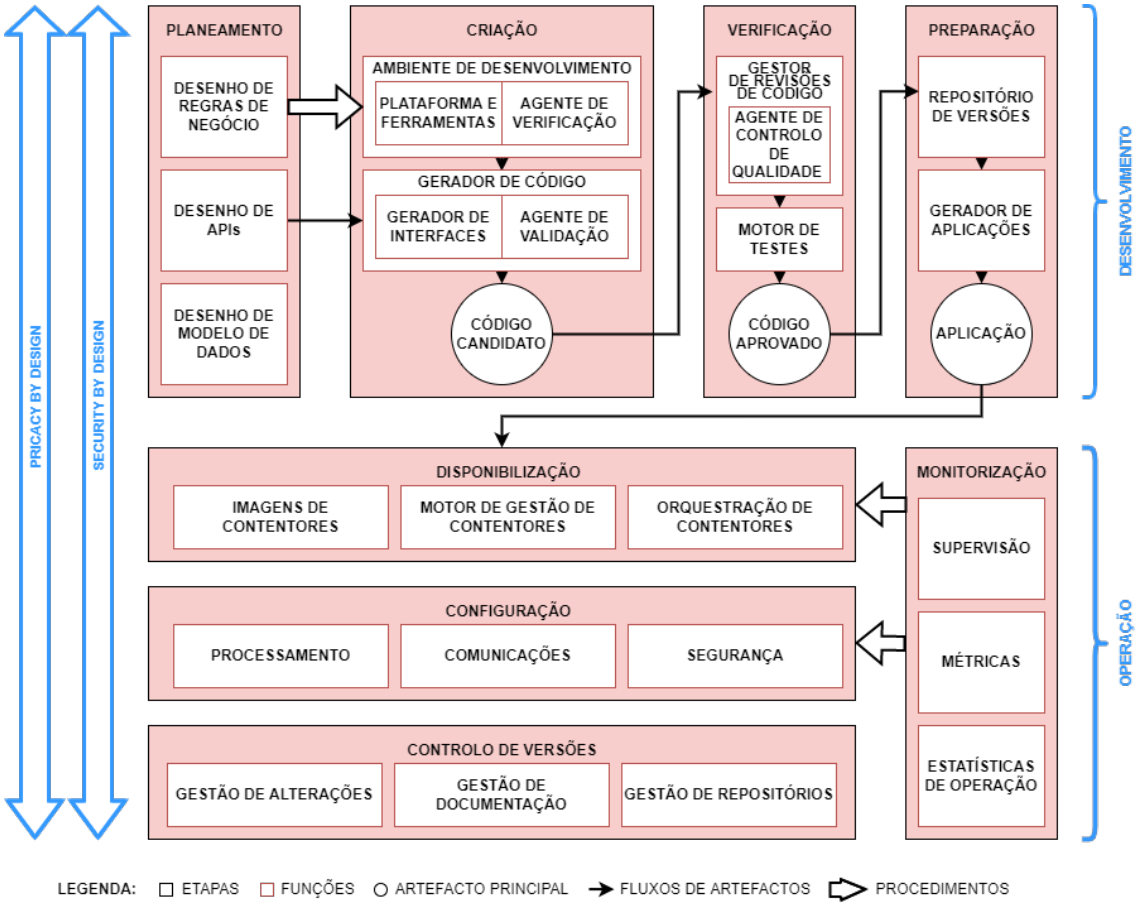

Figura 4.2: Arquitetura incluindo a etapa de verificação

Enquanto o código está a ser escrito este irá sendo assistido pelo agente de verificação para que haja um acompanhamento de qualidade durante a escrita. Terminada a escrita, o código irá passar pelo processo de *build*, o qual irá ser encarregue de gerar o código de maneira que este possa ser compilado. Neste momento o agente de validação irá atuar de maneira a verificar os testes unitários e testes de integração e fazer o respectivo relatório. No fim desta etapa o código obterá então o título de "código candidato" e com ele a permissão para continuar para a próxima etapa que é a de verificação.

Na fase de verificação, o código irá passar em primeiro lugar por um agente de controlo de qualidade que irá definir um *quality gate* cujas condições irão ser verificadas através dos relatórios criados pelos agentes de verificação e agentes de validação da etapa anterior. Caso os requisitos de qualidade sejam cumpridos o código é então passado para o motor de testes, que na sua essência são testes de sistema que irão verificar a qualidade do código totalmente integrado, irão verificar se o código cumpre os seus requisitos e,

por fim, ainda irão verificar se tudo está a funcionar corretamente e se não existem falhas. Aqui o código passa a código aprovado e avança para entrega.

Caso o código candidato não cumpra algum dos requisitos de qualidade, o processo de entrega de código irá ser interrompido, expondo os relatórios com as informações relevantes dos erros e problemas resultantes da interrupção. Isto implicará que os desenvolvedores terão que corrigir todos os problemas indicados pelos relatórios antes de submeter o código.

4.2.1 Avaliação

Após tomadas as primeiras decisões de design, questões foram levantadas no que toca à segurança, à consistência e á usabilidade deste projeto. Nesta secção irão ser descritas algumas das proposições e decisões feitas pela parte destas entidades da empresa, estas que irão ser a base para várias decisões de arquitetura e estrutura do projeto, que irão ser discutidas em maior detalhe mais adiante neste documento.

Começando pelo início da estrutura do estudo, o agente de verificação, que serão os testes de análise estática do código em tempo de desenvolvimento, a ferramenta escolhida, deve ser uma que possa ser utilizada pelos desenvolvedores de código da empresa através do IDE.

Avançando para a estrutura do agente de controlo de qualidade, a ferramenta que cumpre esta função deve ser uma na qual se podem definir quality gates. Iniciais preocupações foram levantadas sobre os quality gates, relativamente aos testes unitários visto que a empresa não queria alargar imenso o seu processo de desenvolvimento de código com testes unitários extensos. No entanto, foi iniciado um movimento dentro da empresa em que os desenvolvedores deveriam ser responsáveis por fazer testes unitários ao seu código. A prioridade dos testes unitários deve ser código que envolve conteúdo sensível critico que possa apresentar vulnerabilidades de segurança na aplicação.

De forma a aumentar as linhas cobertas por testes unitários foi também sugerida a adoção de opções de uma framework utilizada na aplicação que gera testes unitários e testes de integração. Visto que muito do código deste projeto é gerado, e cobertura em código gerado era uma preocupação, esta solução veio a complementar os valores de cobertura pretendidos e ideais.

Relativamente aos quality gates em si, decidiu-se que estes deveriam referir o código novo, ao invés de código global, para adotar a filosofia de que se todo o código novo estiver limpo, em termos gerais, o projeto estará limpo. Para mais sugeriu-se que estes

quality gates fossem passando uma fase experimental, e que fosse ajustando as restrições ao longo do tempo para que, em primeiro lugar se crie um hábito de utilização, e em segundo lugar para que haja uma fase de adaptação entre os desenvolvedores.

Depois de várias reuniões com alguns dos membros da empresa, notou-se que o maior foco seria o motor de testes de sistema. Acerca deste tema várias preocupações foram encontradas quando apresentada a ideia inicial, que seria correr os testes de sistema no ambiente de desenvolvimento, este um ambiente utilizado pelos desenvolvedores para testar manualmente novos recursos que são adicionados durante a fase de desenvolvimento de uma aplicação. Este ambiente contém uma base de dados alocada num servidor que é comum entre todos os seus utilizadores e contém a versão mais atual da aplicação. Imediatamente foram aparecendo preocupações devido ao facto de se utilizar uma base de dados conjunta e que os testes de sistema iriam inundar a base de dados com dados que não seriam utilizados. A primeira sugestão foi tentar eliminar os dados que seriam criados pelos testes de sistema, mas este processo não era eficiente em termos de tempo, pois implicaria a implementação de novos processos a níveis de código ou a níveis de base dados para a própria eliminação. Decidiu-se criar um ambiente individual que tal como o de desenvolvimento, mas no qual as bases de dados pudessem ser descartadas no fim de cada iteração de testes.

Apesar de se ter chegado a estas conclusões rapidamente acerca de como o motor de testes de sistema deveria funcionar, a ideia de criar um ambiente individual gerou ainda mais confusão e preocupação no que toca a segurança, usabilidade e poder computacional. Vai ser então criada um ambiente individual em que outros colaboradores da empresa, cuja especialidade não é a programação de código, fossem capazes de utilizar para criar testes para que estes sejam adicionados à pilha de testes.

4.3 Reflexão

Feita esta avaliação, com membros da empresa, é necessário tomar em conta todos os aspetos e preocupações que foram levantadas durante todas estas discussões e pôr em conta aquilo que é necessário para cumprir a missão que é a integração do CT.

Neste projeto é importante focar na sua exequibilidade, na sua reutilização e na sua generalização para que outros colaboradores possam continuar a utilizar esta estrutura para os seus projetos futuros. A ideia geral não é apenas integrar o CT, mas sim também criar um processo que afete diretamente a estrutura DevOps da empresa para que esta mude de forma a que o CT esteja presente em todos os projetos futuros.

4.3.1 Exequibilidade, generalização, reutilização e flexibilidade

A implementação deste projeto visa tentar cumprir com todos os objetivos propostos pela empresa. No que toca ao agente de verificação e ao agente de controlo de qualidade estas estruturas serão sempre as mesmas para todos os projetos que queiram ser adicionados, pois tratam-se de ferramentas que são adicionadas à estrutura DevOps da empresa em que o seu objetivo é serem facilmente ajustáveis para qualquer tipo de projeto futuro. Em relação ao motor de testes de sistema este será o mais desafiante generalizar e torná-lo em algo que possa ser reutilizável noutros projetos.

A estrutura do motor de testes de sistema utilizará um ambiente individual igual ao ambiente de desenvolvimento do projeto, mas no qual são adicionados bases de dados locais que possam ser descartadas após cada execução. Visto que os ambientes de desenvolvimento já existem, neste ambiente a preocupação é a adição e alteração de recursos como bases de dados e outro tipo de tecnologias que possam fazer sentido alterar, isto faz com que a criação deste ambiente individual seja facilmente exequível para qualquer projeto. É importante perceber que o ambiente individual não é o único componente desta estrutura, o outro componente essencial é o próprio motor de testes. A configuração deste motor terá que ser feita de tal maneira que seja também reutilizável, isto significa que de projeto para projeto o motor seja sempre baseado no mesmo e que execute sempre da mesma forma. O objetivo principal será fazer uma estrutura em que o que varia de projeto para projeto serão apenas os testes em si (o design poderá ser o mesmo, mas cada projeto tem os seus testes únicos) e o ambiente.

Em termos gerais a implementação da estrutura de testes para cada projeto deverá apenas basear-se na criação do ambiente individual e dos testes de sistema em si. Desta forma a integração do processo de testes automáticos torna-se facilmente adaptável a qualquer projeto.O principal objetivo será criar uma estrutura de testes, em que o foco principal são os testes e não o resto dos componentes.

4.3.2 Impacto

Introduzir CT dentro da estrutura DevOps de uma empresa vem com os seus desafios, implicará que todas os colaboradores envolvidos, especialmente os responsáveis pelo desenvolvimento de código, tenham a noção que o CT está presente e que este influenciará a maneira com que se escreve e testa o código.
Começando por baixo, ter um agente de verificação na forma de uma ferramenta de IDE, vai afetar diretamente os desenvolvedores de código durante o processo de escrita. A ferramente deve sugerir maneiras de melhorar a qualidade do código desenvolvido em tempo real, resultando em que o código seja mais limpo, organizado e seguro. Principalmente,

o foco, é melhorar a segurança pois, por exemplo, isto irá evitar o uso de funções e componentes deprecados que apresentem vulnerabilidades que comprometam a segurança do código.

Obviamente o agente de verificação, não vai evitar todos os problemas, no entanto após ser submetido novo código na pipeline, este irá passar pelo agente de controlo de qualidade. Este agente irá prevenir que código vulnerável seja introduzido na pipeline, e por consequência, fará com que os desenvolvedores retomem ao processo de escrita do código com o intuito de melhorá-lo para que este seja aceite pelo agente.

Outra ocorrência impactante, no processo habitual da empresa, é o facto de os desenvolvedores de código tomarem a responsabilidade da escrita e design de testes unitários e testes de integração. A adoção desta filosofia em si, fará com que os desenvolvedores tenham uma maior atenção ao código que estão a desenvolver.

O agente de controlo de qualidade terá ainda de ter a função de verificar se os testes unitários são executados com sucesso ou não. Qualquer teste unitário falhado deve travar o processo da pipeline e notificar a entidade responsável que corrija esse código. Desta maneira os testes unitários tornam-se uma preocupação e têm impacto direto na entrega de código dentro da pipeline.

Por questões de computação limitada, os testes de sistema terão de ser corridos de maneira diferente do que a maneira planeada inicialmente. Estes não poderão ser corridos após a submissão de código novo, mas sim correr os testes periodicamente fora de horário regular de trabalho, de maneira a juntar várias submissões de código e tentar encontrar problemas dentro de um intervalo de tempo. Apesar de as novas entregas de código não serem imediatamente testadas pelo motor de testes de sistema, esta solução apesar de não ser a ideal, gere uma poupança enorme em termos de computação utilizada.
Depois de analisada esta alteração na solução principal, o balanço que se faz em termos de poupança de recursos face os possíveis erros encontrados por entrega acabam por avantajar a nova solução. Visto que os testes serão corridos na mesma, mas periodicamente, estes irão juntar, por exemplo, dez ou mais entregas de código de uma vez e correr apenas uma vez os testes de sistema. A vantagem de correr uma vez por entrega resultaria em detetar mais facilmente a causa de novos problemas, visto que existe um isolamento maior entre novos problemas face novo código. Por outro lado, juntar entregas e correr apenas uma vez o motor de testes vai aumentar a quantidade de possíveis causas de novos problemas, no entanto apenas é executada uma vez o motor de testes reduzindo assim a quantidade de recursos utilizados. Considerando esta última solução, visto que por vezes muitas das entregas de código baseiam-se em pequenas alterações ou pequenas adições de

novas funções, que por vezes podem nem sequer estar ainda cobertas por testes, a adoção desta solução faz ainda mais sentido.

Os testes de sistema vão ser ainda mais impactantes numa programação com múltiplos componentes, que é o caso do projeto em que se vai aplicar esta nova tecnologia que segue uma arquitetura de micro serviços. Em projetos com uma arquitetura de micro serviços os desenvolvedores tendem a assumir que o desenvolvimento de um micro serviço individual não afeta diretamente outros micro serviços. E em teoria isto nunca deveria acontecer, no entanto existem micro serviços que utilizam os serviços de outros micro serviços, e nestas situações, alterações em micro serviços independentes podem vir a ter um impacto noutros. Os testes de sistema testam sempre o sistema na sua totalidade, portanto problemas em micro serviços causados por outros micro serviços serão quase todos detetados mediante o design dos testes sistemas.

Capítulo 5

Implementação da solução: domínio concreto

Nesta secção será descrita a implementação concreta da solução, incluindo os seus requisitos, as suas restrições, as tecnologias utilizadas, os ambientes e as práticas da empresa em que se integra e os resultados concretos obtidos com a solução.

5.1 Requisitos e restrições

Para a implementação desta solução, e qualquer outra implementação no âmbito do DevOps, é necessário ter em conta vários aspetos relevantes no que toca a ambientes, processos e tecnologias que já estão implementadas e estão em produção na empresa em conta. Isto porque como em qualquer fase de desenvolvimento é preferível reaproveitar e utilizar tudo aquilo que já está implementado para ajudar a encurtar e facilitar o próprio desenvolvimento. Estes aspetos e estas análises ao que já existe, por assim dizer, devem ser todas tomadas em conta no desenvolvimento de soluções de DevOps para moldar a pilha tecnológica e a própria lógica da solução. Nas próximas subsecções irão ser descritas e feitas as análises das tecnologias escolhidas para a solução, os ambientes e processos de desenvolvimento atualmente adotados pela empresa tal como algumas práticas adotas pela empresa correntemente.

5.1.1 Escolha de ferramentas

Para a escolha das ferramentas é necessário tomar vários aspetos em consideração para que a integração na arquitetura atual seja ideal. Deve-se ter em conta a integração das ferramentas escolhidas com as ferramentas já utilizadas na arquitetura de forma que não sejam necessárias alterações na mesma, ou assegurar, caso sejam necessárias, que essas alterações aportam uma melhoria significativa a todo o processo, compensando o investimento necessário. Deve-se também ter em especial atenção as linguagens de programação utilizadas, para que se evite utilizar múltiplas ferramentas para a mesma função. Ferra-

mentas com mais funcionalidades e que conseguem cobrir vários tipos testes serão mais interessantes pois pretende-se, no fundo, que a integração seja simples. Adicionalmente, procura-se utilizar ferramentas *open source* que tenham licenças gratuitas, alinhando com a filosofia da arquitetura atual da Trust Systems.

SonarLint O Sonarlint será o mais forte candidato a adoptar como agente de verificação. É uma extensão de IDE grátis e *open source* que faz, como o nome indica, *linting*. Esta ferramenta tem a vantagem de estar associada ao IDE (por desenho), e faz *linting* em tempo real expondo possíveis problemas no próprio momento e sugere correções para melhorar a qualidade e a segurança do código [8].

O SonarLint funciona com os IDEs mais utlizados do mercado, nomeadamente, os IDEs da JetBrains, o eclipse, o visual studio e o visual studio code. Para além disso, o SonarLint mostra uma grande base de conhecimento para as linguagens mais populares como Java, JavaScript, TypeScript, Python, PHP, HTML, C, C++, etc [8]. Outra grande vantagem do SonarLint é a facilidade com que se instala e se começa a utilizar, pois não necessita de quase nenhuma configuração para o fazer.

Nas próximas imagens estão alguns momentos de execução do Sonarlint no IDE IntelliJ Idea.

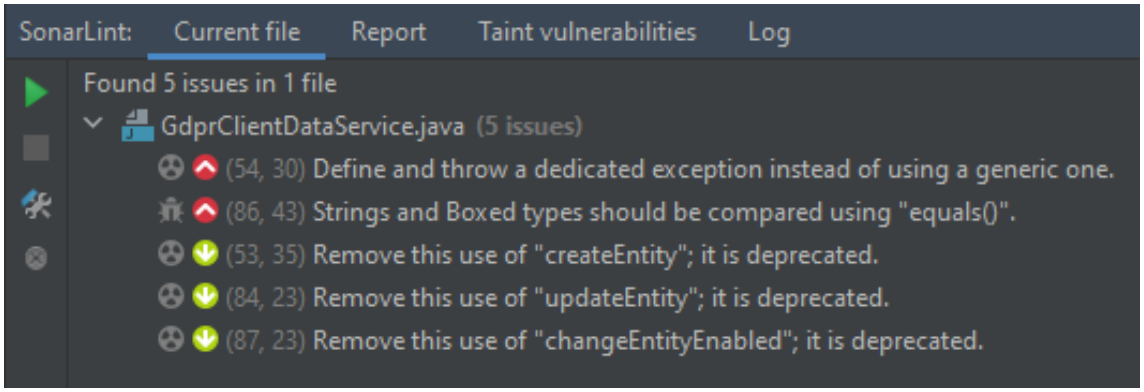

Figura 5.1: Listagem de erros de um ficheiro, através do Sonarlint

Como podemos ver na Figura 5.1, o Sonarlint deteta erros de programação no código, categoriza-os como *bugs* ou *code smells* e atribui a cada um uma importância usando como qualificadores ”*minor*”, “*major*”, “*critical*” ou “*blocker*”.

A Figura 5.2 mostra a descrição de um erro detetado pelo Sonarlint, em que apresenta além de uma breve descrição do problema, o que o poderá causar bem como uma possível solução do erro. Permite também solucionar o problema facilmente através de correções automáticas (Figura 5.3).

Esta tecnologia é uma mais-valia importante para a arquitetura tendo em conta as

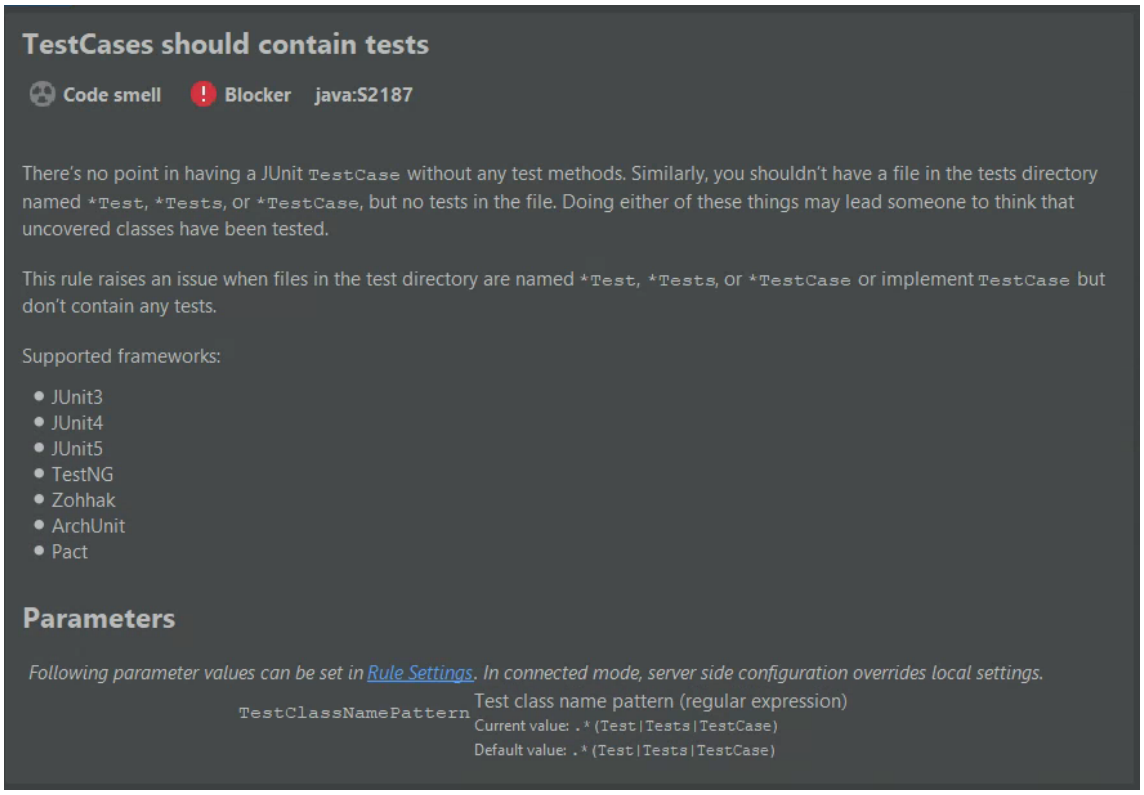

Figura 5.2: Descrição de um erro

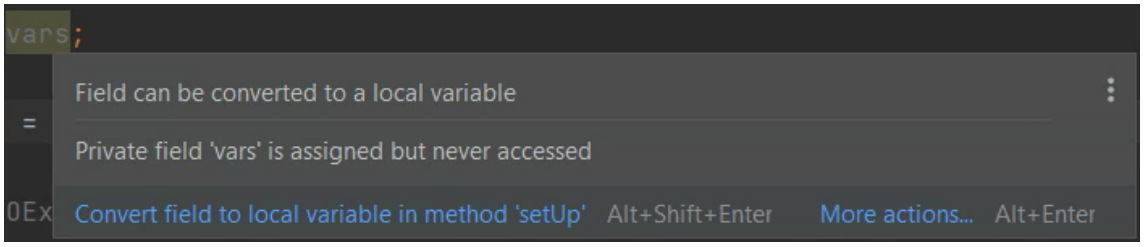

Figura 5.3: Solução automática

tecnologias que são utilizadas para desenvolvimento, nomeadamente os IDEs, IntelliJ Idea e Microsoft Visual Studio Code e as linguagens de programação de Java, com a *framework* Java Spring listadas na base de conhecimento do Sonarlint, tal como o uso de TypeScript e HTML.

SonarQube O SonarQube será o mais forte candidato a adoptar como agente de controlo de qualidade, porque é uma plataforma *open source* que faz inspeções automáticas à qualidade de código através de análise estática ao código fonte de um projeto [1]. Utiliza similarmente ao SonarLint uma estratégia de *linting* para encontrar *bugs*, *code smells* e vulnerabilidades de segurança no código. Para além destas funções o SonarQube tem a capacidade de analisar um projeto inteiro, e fazer um relatório que inclui informações sobre: código duplicado, *coding standarts*, testes unitários, testes de integração, *code co-*

verage, complexidade do código e relatórios de *linting* [1].

O SonarQube funciona com 27 linguagens, incluindo as já mencionadas na ferramenta SonarLint, e fornece análises automáticas e integração para Maven, Ant, Gradle, etc, e ferramentas de CI como o BitBucket, Jenkins, Gitlab, etc [1].

O SonarQube expõe os relatórios na sua plataforma de uma maneira organizada, mostrando de uma forma geral o estado do projeto, e tem também uma aba que discrimina todos os problemas, mostrando onde exatamente se encontram no código, qual o erro, o que alterar, quais as possíveis consequências e o tempo estimado para o resolver. Tal como ao SonarLint, o SonarQube utiliza uma base de conhecimento baseada em regras, todas estas regras podem ser geridas num dos separadores do SonarQube.

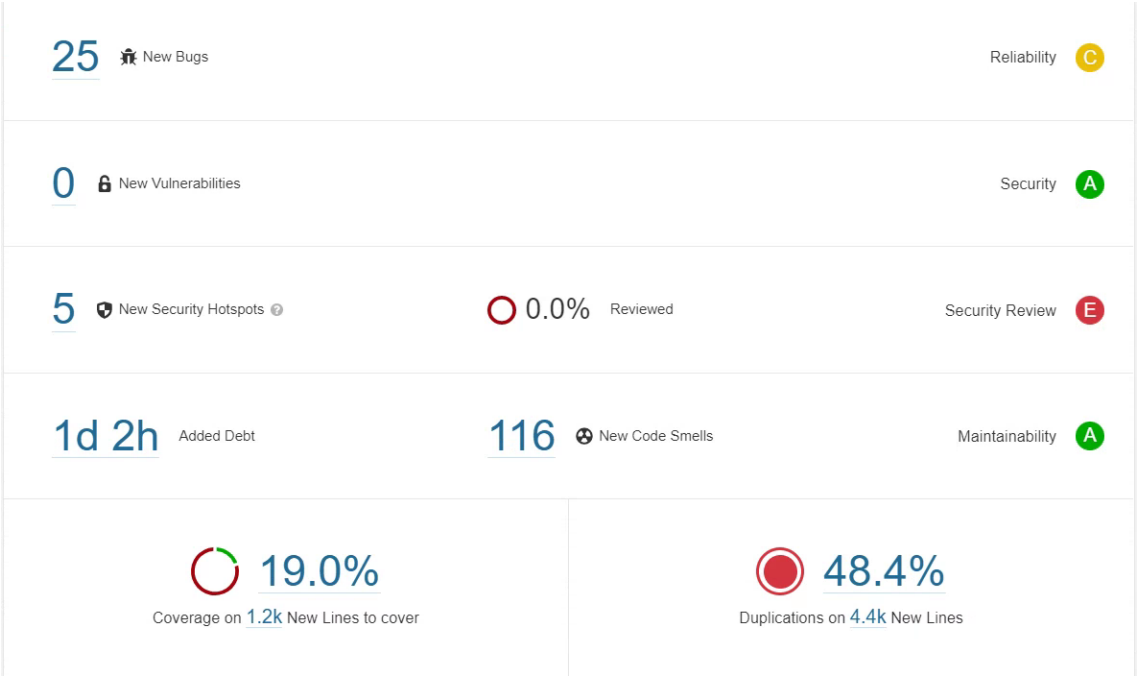

Figura 5.4: Relatório geral de um projeto no SonarQube

Em termos de gestão e administração, o SonarQube consegue prever e dar uma expectativa de quanto tempo será necessário para corrigir os erros, fornece tanto uma expectativa de tempo para cada problema como para o total do projeto (Figura 5.4). O SonarQube permite criar utilizadores, permite que o administrador do SonarQube associe problemas a utilizadores, ignore problemas, marque-os como resolvidos, entre outros. Por fim, o SonarQube permite que se defina uma lista de quality gates (Figura 5.6), que será uma lista de condições que têm de ser cumpridas para que o código seja aprovado e validado.

A utilização desta plataforma com o SonarLint permite que os desenvolvedores estejam a par dos problemas desde que começam a trabalhar no IDE até ao momento que o

Figura 5.5: Listagem dos problemas

Figura 5.6: Exemplo de um quality gate

relatório do SonarQube é feito, facilitando as correcções e reduzindo assim os problemas. Outra grande vantagem do SonarQube é a integração de quality gates com as ferramentas de CI/CD, nomeadamente com o Jenkins que é utilizado na arquitetura atual. Esta integração vai ser o fator que une as duas tecnologias e vai permitir com que se interrompa o processo de CI/CD.

Ferramentas de testes unitários O SonarQube já utiliza uma livraria grátis de java chamada JaCoCo, cujo nome significa *Java Code Coverage*, que será o agente de validação do projeto. Esta livraria gere relatórios de *code coverage* para projetos em Java [11]. O JaCoCo é fácil de configurar no projeto, no entanto requer um esforço de tempo pelo desenvolvedor, para a poder aplicar, sendo que em primeiro lugar terá de fazer uma seleção de todas as funções a testar e para cada uma dessas funções será preciso fazer os testes necessários com a livraria Junit [12]. Com o JaCoCo adicionado ao projeto testar é tão simples como inicializar o SonarQube para observar os resultados na plataforma. Os resultados são expressos de uma forma intuitiva utilizando cores, nomeadamente vermelho, amarelo e verde e providencia principalmente 3 métricas importantes que são: a quantidade de linhas que foram cobertas pelos testes; a percentagem dos percursos corridos no código, geralmente baseado nas expressões *if/else* e *switch*; a cyclomatic complexity que é uma reflexão da complexidade do código baseando-se no número de caminhos necessários para cobrir todos os caminhos possíveis no código utilizando combinações lineares [12].

O JUnit é uma *framework* de Java que providencia um motor de testes unitários e testes de integração. Permite a utilização de funções, métodos e condições para que que possa realizar testes a funções de Java [32].

Selenium O Selenium é uma ferramenta *open source* gratuita de *record and playback* para a criação de testes de sistema sem a necessidade de aprender uma linguagem de *script*. Fornece também a possibilidade da utilização de uma linguagem específica de *scripts* de Selenium (Selenese) para escrever testes em várias linguagens de programação populares, incluindo JavaScript, C #, Java, Perl, PHP, Python, entre outras. Estes testes podem ser executados na maioria dos *browsers web* modernos [7]. Esta ferramenta será o mais forte candidato a adotar como o motor de testes do projeto.

O Selenium permite a escrita de vários tipos de testes através do *browser*, a grande vantagem é que permite interagir diretamente com os elementos do HTML dando controlo total e muitas possibilidades de testes ao desenvolvedor. O Selenium permite criar um bom relatório de testes através da utilização de vários *scripts* executados numa sequência especifica. Utilizar a estratégia de correr os testes de sistema numa sequência permite discriminar todos os testes de maneira que se saiba em que parte da execução falhou. A necessidade de terem de ser corridos numa sequência vem da capacidade do Selenium para agir como um utilizador para o *browser*, permitindo a utilização de *cookies*, sessões, etc.

5.1.2 Síntese da escolha das ferramentas

A solução consiste em utilizar os pontos fortes e as vantagens das ferramentas, que neste caso serão SonarQube em conjunto com SonarLint, JaCoCo e o motor de testes do Selenium.

A utilização do SonarLint servirá como uma mais-valia para os desenvolvedores, tanto de BE como de FE, para identificarem os problemas enquanto estes estão a ser escritos. A utilização desta ferramenta irá identificar através do cálculo da complexidade cognitiva (discutido nos próximos parágrafos) que funções devem ser testadas unitariamente com o JUnit. Para além disso o SonarLint irá permitir, como já mencionado, que a deteção de erros seja feita no momento da escrita o que irá melhorar a qualidade de código para a passagem do quality gate definido pelo SonarQube.

O Jenkins permite a integração de quality gates tanto por parte do SonarQube como do Selenium, portanto irão ser definidos dois tipos de quality gates que irão ser verificados imediatamente após o *build* do programa. Estes quality gates quando não são cumpridos

Métrica	Operador	Valor
Coverage	Menor que	80%
Duplicated Lines (%)	Maior que	3%
Maintanability Rating	Pior que	A
Reliability Rating	Pior que	A
Security Hotspots Reviewed	Menor que	100%
Security Rating	Pior que	A

Tabela 5.1: *Quality gate Sonar Way*

irão interromper a entrega do software.

O primeiro quality gate será o do SonarQube, os quality gates do SonarQube têm duas opções, qualidade de código novo ou qualidade de código global. O *"Sonar Way"* é um *default* da Sonar que implementa boas práticas de uma estratégia chamada "*Clean as you Code*" [6]. Em que o código novo falha se não conseguir cumprir com os requisitos definidos na Tabela 5.1.

O "*Clean as you Code*" é uma filosofia que se foca apenas no código novo que é inserido no projeto, dizendo que se todo o código novo for cumprindo os requisitos, o projeto numa perspectiva global também irá cumprir esses requisitos. Se esta filosofia for aplicada num projeto desde o seu início, obviamente que os padrões do projeto global irão cumprir os requisitos. No entanto não é possível medir tudo no código novo e deve ser aplicado um quality gate no código global que teste, por exemplo, a eficácia dos testes unitários, visto que o elemento *coverage* dos quality gates de código novo apenas calcula a percentagem de linhas ou funções que são cobertas pelos testes unitários. Para além dos testes unitários, no código global deve-se integrar um limite de complexidade, pois esta, novamente, não pode ser aplicada a código novo mas apenas a código global.

A "*Cyclomatic Complexity*" é calculada com base no número de caminhos no código. Quando o fluxo de uma função é dividido o contador de complexidade aumenta. Cada função tem um mínimo de complexidade de 1. Os cálculos podem variar de linguagem para linguagem, por exemplo, em java a complexidade é incrementada por cada keyword: *if, for, while, case, catch, throw, &&, ——,?* [4].

A *Cognitive Complexity* é o indicador de dificuldade de entendimento do fluxo de código. Os cálculos são feitos especificamente utilizando lógicas e com base no nível de *Cyclomatic Complexity*. A autora do *Cognitive Complexity White Paper, G.Ann* [18], em

resposta a uma pergunta frequente [22] acerca do que seria um bom limite para *cognitive complexity*, indica que 15 de *cognitive complexity* deverá ser o máximo por função. Refere também que este valor é uma referência base e que este número pode variar na linguagem de programação e na necessidade da função. Em geral diz que 15 é um bom número a ter em conta.

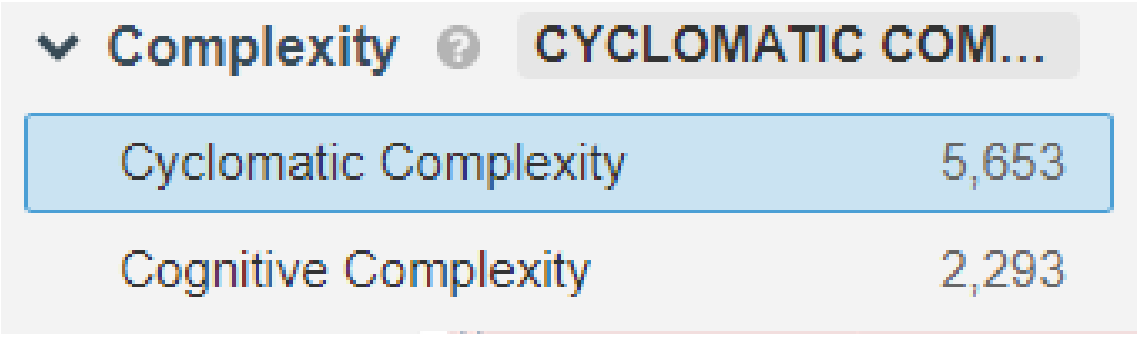

Figura 5.7: Exemplo de complexidade num projeto

Analisando os parâmetros do *Sonar Way quality gate*, em termos de segurança devem se definir os padrões mais altos, sendo a segurança uma prioridade em qualquer desenvolvimento de software. Tomando como verdadeiro que se deve sempre apontar para elevados níveis (ou níveis máximos) de confiabilidade (*reliability*), tal implicaria corrigir todos os *bugs* detetados pelos conjuntos de regras definidos pelas ferramentas.

Em termos de percentagem de *code coverage*, pesquisas dizem que muitos desenvolvedores de testes gostam de definir o mínimo de percentagem de *code coverage* entre os 20% e 30% focando-se apenas em funcionalidades do software que achem relevantes para o contexto [5]. Adquirir 100% de *coverage* parece ser o ideal, mas é uma má ideia segundo Marick, em [25] pois "*Designing your initial test suite to achieve 100% coverage is an even worse idea. It's a sure way to create a test suite weak at finding those all-important faults of omission. Further, the coverage tool can no longer tell you where your test suite is weak - because it's uniformly weak in precisely the way that coverage can't directly detect.*". Mesmo assim, sistemas críticos, como por exemplo software utilizado pelas companhias aéreas nas suas aeronaves, requerem 100% de coverage em todos os aspetos [5]. Muitas empresas optam por escolher um número popular que é os 85% pois foi um número popularizado pela Division X e por outras empresas de renome [25]. No entanto, como referido em [25], é possível enviesar essa taxa de cobertura, obtendo 85% de *coverage*, apenas por olhar para as condições de *coverage* e escolher as que são fáceis de satisfazer para atingir o objetivo. Este comportamento é um processo mais rápido do que pensar em características para testes mais complexos que realmente testam as fraquezas do código. Em [25] definem 70% como um número ideal pois apesar de se

manter uma porção do código que não esteja a ser testada, o custo de cobrir essa zona não compensa os benefícios de encontrar alguns possíveis *bugs*. Daí concluir que deve ser feito um inquérito e uma análise estratégica de que funções são realmente necessário testar e que, com frequência, a percentagem de *coverage* não significa o que as pessoas normalmente entendem [21].

Após um inquérito com alguns membros da empresa concluímos que no quality gate será necessária não o número de linhas que os testes cobrem(*coverage*) propriamente dita, mas sim a taxa de sucesso dos testes unitários que deve ser de 100%, e isto porque não existe tanto interesse na quantificação das linhas que os testes cobrem, mas sobretudo em saber o que está a ser testado e a esse respeito, mediante uma análise de risco, será feito um levantamento para se saber que zonas do código devem ser testadas e como devem ser testadas. Ou seja, é escolhido o código mais problemático ou com mais risco.
A principal razão para esta decisão dos membros de gestão de empresa é o compromisso com a qualidade e a necessidade de considerar o *trade off* que existe entre tempo de desenvolvimento e qualidade código, que não é uma função linear (trata-se de uma análise diferencial). As várias pessoas consultadas acreditam que obter percentagens de *coverage* altas poderá a partir de certa altura aumentar o tempo de desenvolvimento de forma tal que possa deixar de justificar o resultado dos testes adicionais.

Sobre o quality gate do Selenium deve ser feito, tal como no caso dos testes unitários, um levantamento dos testes de sistema. Supõe-se que o *script* em Selenium será feito de forma a imitar o fluxo de um utilizador real, que percorre e interage diretamente com a página web, averiguando sempre se os resultados são os que se esperam. Estes testes devem igualmente testar a página face aos diferentes tipos de utilizadores (admin, normal, por exemplo), caso os tenha, de forma a testar os requisitos de cada um deles, e se estes estão a ser cumpridos ou não.

5.1.3 Tecnologias da empresa

O Jenkins será uma das tecnologias mais importantes nesta solução, devido ao facto de ser o core de toda a pipeline da empresa e pela sua capacidade de servir de condutor e possibilitar a automação desta solução. Outra ferramenta que vai ser utilizada, e que é muito conhecida no mundo do DevOps, é o Docker, que é uma ferramenta que é utilizada para implantar software em pacotes virtualizados chamados de containers. Em conjunto com o Docker que cria containers vai ser utilizado o Traefik que é uma ferramenta utilizada em conjunto com o Docker para fazer a gestão da rede dos containers utilizando o protocolo de *Domain Name System* (DNS) para que os containers possam comunicar e ser acedidos através de URLs definidos. Outra capacidade do Traefik é na gestão e implementação dos

certificados *Transport Layer Security* (TLS) destes containers.

As bases de dados utilizadas serão em PostgreSQL, que são as utilizadas pela empresa nos seus projetos. Em questão às bases de dados ainda será utilizado o Liquibase que é responsável por gerir e aplicar alterações ao esquema das bases de dados, isto será importante para criar o esquema das bases de dados automaticamente quando um serviço de software é iniciado, pois visto que esta solução para os testes de sistema irá utilizar sempre bases de dados vazias é importante garantir que as bases de dados quando são iniciadas mantêm sempre o esquema certo nos seus serviços.

A base do projeto dos testes de sistema vai ser feita em Java com Maven como a sua ferramenta de compilação, associado ao Maven nas suas dependências teremos o Junit que é responsável pelo suporte à criação de testes automatizados em Java. Para além do Junit vão ser utilizados o JaCoCo e o Surefire Report para a criação dos relatórios dos testes que irão ser corridos. Ainda será utilizado o ChromeDriver que é uma ferramenta open-source criada especificamente para testes automáticos em aplicações web, que providencia capacidades de navegação em páginas web, fazer inputs, executar JavaScript, etc. O ChromeDriver atua como um servidor standalone e irá possibilitar executar os testes automáticos criados pelo Selenium.

5.1.4 Fases de desenvolvimento

Como qualquer processo de desenvolvimento de software, especialmente o desenvolvimento de um projeto ao nível do DevOps, que envolve a utilização e configuração de múltiplas tecnologias diferentes é importante ter o conhecimento dessas tecnologias, tal como perceber como cada uma é importante no desenvolvimento da solução.

A primeira fase de desenvolvimento da solução foi essencialmente obter conhecimento das tecnologias que têm um papel importante na solução. Um claro exemplo foi o Jenkins que é a base desta solução pela sua capacidade de automatizar o processo de entrega de software através de scripts que contam com imensos comandos que cumprem os seus propósitos específicos.

Depois de perceber as funcionalidades que o Jenkins oferece, foi então aí que começou a segunda fase do desenvolvimento que foi analisar os atuais processos de entrega de software da empresa, tal como os seus ambientes e maneira com que a implementação das aplicações nestes ambientes é feita.

Aqui começou a teceira fase que foi desenhar uma solução que cumpre os objetivos pretendidos que neste caso é a implementação de uma nova etapa no processo de entrega

de software na pipeline que inclui a testagem do software a entregar e ainda seja capaz de avaliar os resultados e atuar respetivamente, mediante o resultado dos testes, no comportamento da pipeline. O requisito aqui é que a pipeline tem de ser capaz de ser interrompida caso os testes tenham um resultado que não cumpre os padrões definidos. Para além de ser capaz de ser interrompido o processo da pipeline, é importante que o desenvolvedor responsável pelas mudanças que ocorreram na interrupção, seja notificado que mudanças têm de ser feitas para que o código seja entregue na pipeline.

A quarta e última fase de desenvolvimento é o desenvolvimento em si da solução, a implementação nos ambientes da empresa e as configurações necessárias para que este processo na pipeline seja completamente automático.

5.1.5 Práticas da empresa

Há que ter em conta também no desenvolvimento da solução algumas práticas frequentes da empresa nos seus ambientes de desenvolvimento. A TrustSystems tem para cada projeto geralmente quatro ambientes. Um local para cada desenvolvedor utilizar enquanto produz software, um de desenvolvimento que é geralmente o que tem a versão de software mais atual utilizada para testar manualmente a aplicação antes que esta seja passada para o ambiente de QA e só depois de passar a fase de QA é que o software é passado para o ambiente de produção. Todos estes ambientes são compostos por containers Docker que estão dentro de uma rede que é gerida por um outro container Docker com uma imagem do Traefik.

A Figura 5.8 apresenta um exemplo de um ambiente de uma aplicação da TrustSystems em que dentro de uma máquina de produção temos um container que contem o Traefik e dentro da rede do Traefik temos os serviços que compõem a aplicação web dentro de containers. Uma aplicação pode vir a ter vários destes serviços, os serviços de FE são os que correspondem ao front end da aplicação que comunicam através de HTTP com uma *Application Programming Interface* (API) *Representational State Transfer* (REST) definida pelos serviços de BE, estes serviços de BE podem ser múltiplos e em cada um deles existe a sua própria DB para armazenar dados e o seu próprio Redis para tratar a cache. O formato do ambiente de uma aplicação é importante pois o motor de testes de sistema terá de operar sobre um ambiente deste género totalmente independente e vazio. Isto implica que terá de ser feito um ambiente adicional aos existentes cujo propósito será nomeadamente para correr os testes de sistema e ainda ser utilizado para a escrita e design dos testes.

Outro requisito definido pela empresa para esta solução é que a solução deve ser segura e não deve comprometer em nenhum momento a segurança já implementada nos seus

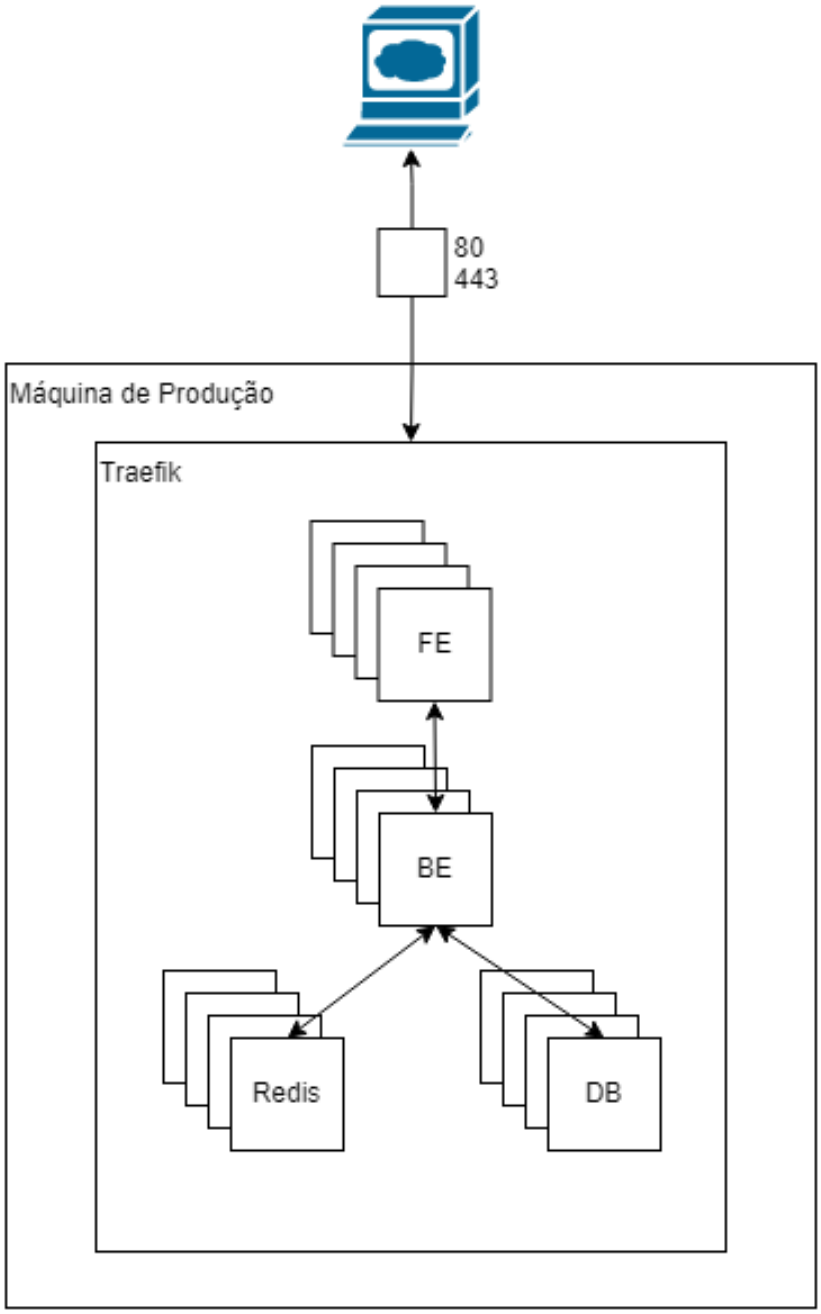

Figura 5.8: Exemplo de um ambiente de uma aplicação

processos da pipeline. Isto condicionou o desenvolvimento da solução de maneira a pôr a segurança em primeiro lugar, só depois a funcionalidade. Condicionou também com que as escolhas das ferramentas garantissem certos parâmetros de segurança.

5.2 Desenvolvimento concreto da solução

O desenvolvimento da solução foi dividido em duas iterações, uma para a integração da etapa de testes na pipeline da empresa, através do uso da ferramenta SonarQube, e a outra iteração é o desenvolvimento e integração do motor de testes com base em Selenium.

5.2.1 Primeira iteração

A primeira iteração vai se focar na integração da etapa de testes na pipeline, há que relembrar a Figura 3.4 para perceber qual é o ideal de uma pipeline de DevOps. A atual pipeline da empresa não tem nela integrada a etapa de testes como exemplificado na Figura 5.9.

Figura 5.9: Pipeline incial

A etapa de testes vai ser integrada entre a fase de Build e a fase de Release, pois idealmente o software deve ser testado antes de ser entregue para os ambientes de desenvolvimento, como mostrado na Figura 5.10.

Figura 5.10: Pipeline atual

Depois de o software ser desenvolvido (Develop) e colocado no repositório de código é acionado um processo de Build no Jenkins que vai começar por puxar a mais recente versão do código e correr um script definido num Jenkinsfile específico a cada projeto.

Os serviços de BE que foram o foco desta implementação, são desenvolvidos em Java e utilizam o Maven para compilar o seu projeto. Portanto o script do Jenkins vai começar por compilar o código com o Maven, este processo de compilação é encarregue também de correr os testes unitários e testes de integração feitos em JUnit caso existentes no serviço de BE. Este processo de compilação do Maven trata também, em conjunto com o JUnit e o JaCoCo, da criação de um relatório acerca dos testes unitários e testes de integração calculando assim a taxa de sucesso dos testes e o cálculo da cobertura dos testes sobre o código. Caso algum dos testes falhe geralmente a compilação é dada como falhada, e a pipeline não avança. Portanto dizer exatamente que a fase de testes da pipeline

é inexistente é incorreto. No entanto existem projetos em que o comando de compilação pode ter a opção de não serem executados os testes unitários e testes de integração e, portanto, isto pode variar de serviço para serviço.
O script do Jenkins continua com a criação do pacote de java gerado pela compilação e é atualizada a imagem do serviço no repositório de imagens dos serviços (Release). É então acionado após esta etapa o processo de implementação e atualização da imagem nos ambientes de desenvolvimento (Deploy) para o serviço ficar operacional no ambiente (Operate). Mediante o resultado do script, sucesso ou falha, é enviado um email a certas entidades da empresa com o resultado, para que estes possam gerir e notificar os desenvolvedores certos sobre o resultado.

É importante perceber que atualmente este script apenas pode falhar com erros de compilação e falha nos testes unitários ou testes de integração. A introdução do SonarQube neste script vai possibilitar que sejam criadas razões de falha, ao nível do próprio código que foi desenvolvido. Isto porque o SonarQube tem a capacidade de analisar o código todo do projeto e avaliar possíveis bugs e erros no código que possam não ter sido detetados no processo de compilação.
Juntamente com a capacidade de analisar o código do serviço, este tem a capacidade de analisar os relatórios produzidos pelo JaCoCo e criar o seu próprio relatório que mede o código ao nível de, por exemplo, linhas de código cobertas por testes unitários tal como calcular a percentagem de código que é coberta por testes. Isto tudo para dizer, como já explicado neste documento, que o SonarQube permite definir um padrão de qualidade código para os serviços através de um quality gate.

Portanto foi implementada e integrada uma imagem Docker com a versão mais recente do SonarQube numa máquina de desenvolvimento. A documentação sobre esta imagem da ferramenta foi bastante útil na configuração da mesma, possibilitou que facilmente fosse feita uma ligação a uma da base de dados PostgreSQL de desenvolvimento da empresa. Para além das ligações à base de dados e o lançamento da própria ferramenta numa máquina de produção, foi criado um volume Docker para persistir os dados e configurações feitas dentro da ferramenta. Este volume Docker não é mais que uma pasta partilhada entre a máquina que corre o container e o container. No âmbito de DevOps é muito importante ter em conta a persistência dos dados, sem este volume caso algo inoportuno acontecesse e o container fosse desligado, todas as configurações feitas na ferramenta seriam perdidas permanentemente. Assim caso aconteça algo ao container ou seja preciso desliga-lo por alguma razão, nada é perdido e basta apenas iniciá-lo, e ele é inicializado no estado em que se encontrava. A configuração deste volume também é importante para o caso de ser preciso fazer uma migração da ferramenta para uma outra máquina no futuro.

Após lançado em produção, o SonarQube teve de passar uma configuração inicial básica, que implicava a autenticação e a criação de contas para as entidades da empresa. Com esta inicialização da ferramenta feita, o próximo passo é interligar o SonarQube ao Jenkins, para tal é utilizada uma extensão do Jenkins em que definimos o nosso servidor de SonarQube. Tal como no Jenkins, no SonarQube também é necessário definir um webhook que permite aceitar pedidos do Jenkins. Todas estas configurações que permitem a intercomunicação do SonarQube com o Jenkins são protegidas com secrets e chaves para garantir a segurança na comunicação entre as duas ferramentas.

Ao termos o SonarQube e o Jenkins conectados e a comunicar o próximo passo foi definir um quality gate no SonarQube para definir os padrões de qualidade de código. Optou-se inicialmente por um quality gate com estandartes não muito elevados apenas em código novo, para que seja avaliado o comportamento da nova tecnologia, e para que este possa ser aperfeiçoado com tempo.

Com o quality gate definido só falta adicionar ao script do Jenkins os novos passos para que após a compilação (Build) e antes de ser feita a atualização da imagem do serviço (Release) os resultados da compilação e o código sejam enviados para o SonarQube para que este possa fazer a sua análise. Os resultados da análise do SonarQube são então avaliados consoante o seu quality gate e é então emitido o resultado do quality gate como sucesso ou falha. O resultado do quality gate é então enviado de volta para o Jenkins e mediante o resultado, o script do Jenkins avança para a próxima fase ou é interrompido.

Adicionalmente, no script do Jenkins, foi adicionado um recipiente novo no processo de enviar emails. Achou-se que era também importante que o desenvolvedor responsável pela entrega do código seja notificado com o resultado do script do Jenkins.

Inicialmente, por opção das entidades superiores da empresa, foi desativada a possibilidade do SonarQube interromper a pipeline para que sejam analisados os resultados dos quality gates. Estes resultados dos quality gates, tal como toda a análise do código feita pelo SonarQube é logo disponibilizada automaticamente na própria página do SonarQube.

Em síntese após uma entrega de código, o Jenkins vai começar por obter a última versão do código, vai compilá-lo (Build), correr os testes unitários e testes de integração, vai enviar os relatórios dos testes e o código para o SonarQube, este vai analisar esta informação e calcular o resultado do quality gate, procede a enviar o resultado de volta ao Jenkins, este mediante o resultado vai avançar com o script ou interrompê-lo (Test). Caso avance, procede para atualizar a imagem do serviço (Release) e assim é então acionado o

processo de atualização da imagem no ambiente de desenvolvimento (Deploy), para que as alterações feitas possam ser operáveis (Operate). Mediante o resultado do script seja este falha ou sucesso é enviado um email para as entidades responsáveis, e agora também será enviado para o desenvolvedor responsável pela entrega.

5.2.2 Segunda iteração

Esta iteração vai agora focar-se no desenvolvimento e implementação do motor de testes utilizando a tecnologia do Selenium, tal como a sua automatização e inserção na pipeline.

Antes de avançar com a descrição da iteração, é importante relembrar o foco e a importância deste motor de testes, que é nomeadamente tentar criar uma maneira de correr testes de sistema totalmente automática. Outro ponto que deve ser tomado em conta, é tentar criar um sistema que seja capaz de ser utilizado por entidades da empresa, cuja função não é desenvolver código, ou seja, tentar criar algo que uma pessoa sem grande experiência de programação seja capaz de poder desenvolver novos testes e aumentar a pilha de testes de sistema deste motor sem grandes dificuldades.

Para a automatização do processo, podemos sempre contar com o Jenkins para o fazer através de um script que siga os passos necessários para concretizar o dito processo. Posto isto esta iteração vai ser descrita em duas partes principais que são a criação do motor de testes e o script do Jenkins para a integração deste motor na pipeline.

O pensamento de como o motor de testes deveria ser implementado, começou com a experimentação do Selenium IDE. O Selenium IDE é uma extensão de browser que permite a criação de um teste de sistema através de record e playback. Simplesmente abrimos a página web que queremos testar, começamos o record do Selenium IDE, fazemos o nosso teste manualmente na página, o Selenium IDE regista todos os passos e interações com a página web num script, e depois disso podemos apenas correr o script em modo de playback, e automaticamente o Selenium IDE corre o script na página. Assim é feita a criação dos testes de sistema, no entanto o processo de correr os testes de sistema é feito manualmente através do uso do Selenium IDE.
Felizmente o Selenium IDE permite a exportação dos testes de sistema para outras linguagens de programação, uma dessas opções de exportação era a exportação dos testes no modo de testes unitários de JUnit. Apesar de não serem testes unitários, para um compilador com o Maven, eles comportavam-se dessa forma o que significava que podiam ser corridos num projeto Maven.

Assim criou-se um projeto Maven com todas as dependências necessárias para correr estes testes que resultaram da exportação. Estas dependências eram nomeadamente as

bibliotecas do JUnit, as bibliotecas dos comandos Selenium para o JUnit, e finalmente uma dependência para as conexões com o ChromeDriver. Ao analisar os testes que eram exportados, notou-se que existia para cada teste a configuração do ChromeDriver, onde se definiam algumas configurações, como por exemplo a definição de esperas máximas antes de timeout e a definição de ser headless, que significa apenas que o browser não necessita de ser mostrado no ecrã e que os testes são executados no browser num modo de background, esta opção é especialmente útil para se conseguir correr o projeto facilmente sem ter que ser mostrado os ecrãs daquilo que está a ser testado e no fim da execução apenas é mostrado o resultado do teste. No fim de cada teste este ChromeDriver é descartado. Como para cada teste esta configuração é igual, de maneira a facilitar a importação de novos testes no projeto, foi criada uma superclasse que continha já as configurações, a inicialização e a eliminação da instância do ChromeDriver. Agora para importar testes bastava apenas o bloco de teste sem as configurações e adicionar apenas a anotação que este estende a superclasse que trata do setup do ChromeDriver.

Como podemos observar na Figura 5.13 o motor de testes começa pela execução do comando Maven para iniciar a compilação e correr os testes unitários, são então compiladas as dependências e é feita a ligação entre o Maven e o ChromeDriver que está presente na máquina que corre os testes, de seguida é feito o setup do ChromeDriver na superclasse que definimos, de seguida este é inicializado, são corridos os testes com os comandos Selenium no driver, o ChromeDriver é descartado, após todos os testes serem corridos é gerado o ficheiro reports.xml com os resultados dos testes.

Os testes de sistema têm a capacidade de testar a aplicação em todos os seus níveis através do ecrã da aplicação, o que significa que para testes mais eficientes correr estes testes num ambiente vazio (com as bases de dados vazias) é ideal, porque permite a criação de testes mais complexos. Outra vantagem de correr os testes num ambiente vazio é que as bases de dados de produção não estão a ser cheias com dados irrelevantes destinados puramente aos testes. Para tal foi criado um ambiente como o referenciado na Figura 5.8 individual com a sua própria rede de Traefik, para este ambiente não interferir com a rede Traefik já implementada, que é feito com as imagens Docker mais recentes da aplicação a ser testada. No fundo este ambiente é um ambiente igual ao ambiente de desenvolvimento da aplicação, mas que em vez de serem bases de dados de produção as bases de dados são containers Docker com uma imagem do PostgreSQL. Os serviços de BE utilizam o liquibase para gerir os esquemas das bases de dados, portanto ao ser inicializado um desses serviços, que agora está conectado a uma base de dados vazia, o liquibase vai se responsabilizar por criar o esquema da base de dados para aquele serviço e manter consistência com o ambiente de desenvolvimento.

Outra vantagem deste ambiente individual com a sua própria rede é, por questões de segurança, termos a opção de limitar quem acede a este ambiente, ou seja, este ambiente é criado e apenas poderá ser acedido pelo próprio motor de testes e mais ninguém. Para tal foi necessário criar uma imagem Docker com o projeto Maven do motor de testes que vai ser corrido dentro da nova rede do Treaefik. Em termos de configurações desta imagem foi apenas necessário instalar dentro da imagem o ChromeDriver, adicionar os certificados CA da aplicação web e criar um volume Docker para obter o ficheiro dos relatórios de teste.

Foi também criado um ambiente igual ao mencionado anteriormente, mas que se conecta à rede Traefik local. O único propósito deste ambiente é para a criação dos testes, pois estes necessitam ser criados num ambiente vazio, de maneira a conseguir consistência.

O próximo passo é então a criação de um script Jenkins para levantar este ambiente e correr o motor de testes nele. Para além da criação do script é necessário configurar este processo para este ser executado automaticamente, sem ser necessário acionar os testes manualmente. Inicialmente foi pensado em acionar o motor de testes a cada alteração no código, no entanto isto não foi possível devido à computação necessária para o fazer, visto que existem projetos que são compostos por imensos serviços que utilizam muito poder de computação. Em alternativa o script do Jenkins vai ser acionado uma vez por dia, todos os dias, de madrugada, onde a máquina de produção não está a utilizar tanta computação.

Para colocar isto em perspetiva de uma pipeline do DevOps, considerando a Figura 5.10 como o processo de uma entrega de software o motor de testes irá cobrir, em vez de uma entrega só, vai cobrir uma coleção de entregas que foram feitas num certo intervalo de tempo, que no nosso caso é nas últimas vinte e quatro horas. Para criar uma perspetiva gráfica, a Figura 5.11 mostra a cobertura do motor de testes. Voltando agora para o script

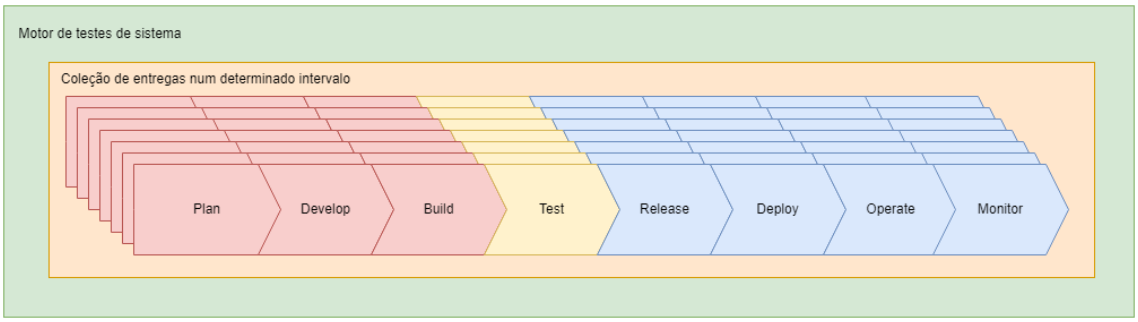

Figura 5.11: Cobertura do motor de testes em perspetiva das entregas da pipeline

do Jenkins, este, resumidamente, terá de criar o ambiente a ser testado, testar o ambiente

através do motor de testes e finalmente apagar o ambiente criado. Uma representação gráfica pode ser observada na Figura 5.12. Vamos agora utilizar a Figura 5.13 para nos

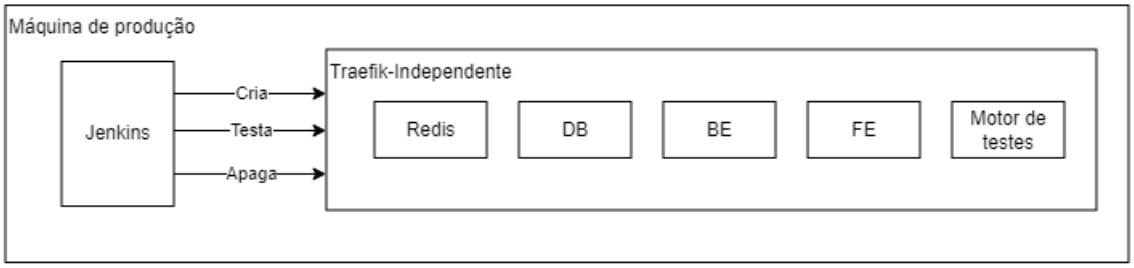

Figura 5.12: Esquema do script Jenkins

guiar no fluxo do script do Jenkins.
O script é então inicializado uma vez por dia a uma determinada hora, e começa por criar a rede do Traefik-Independente e correr o container Docker com a imagem do Traefik. Agora com a rede configurada procede a puxar as imagens Docker mais recentes dos containers que compõem o ambiente. Com as imagens puxadas, este começa a inicializar os containers por passos, começando pelos containers de Redis e DB pois os serviços de BE dependem destes containers para serem inicializados. Após garantir que estes containers estão iniciados e prontos a utilizar, inicializa o resto dos containers que contêm os serviços de BE e de FE e o script neste momento espera uns momentos para garantir que todos os containers são iniciados com sucesso.
Antes de avançar para executar o motor de testes, alguns destes containers de BE especialmente comunicam com outros containers de BE e para isso ser concretizado, visto que o ambiente não é público, é necessário adicionar nestes containers os certificados CA da aplicação web.
Com o ambiente totalmente configurado e inicializado, é então inicializado o container do motor de testes que vai correr os testes no ambiente criado. Após a execução do motor de testes é obtido o ficheiro com os resultados dos testes, neste momento o ambiente é terminado e apagado, e um email com os resultados dos testes é enviado para as entidades responsáveis pela gestão da aplicação.

Em síntese, automaticamente, a uma determinada hora do dia, é corrido o script descrito. Este script foca-se em correr o motor de testes de sistema num ambiente totalmente vazio, avaliar os resultados e notificar as entidades necessárias sobre os resultados destes testes. O que difere a execução deste script de um dia para o outro, é a coleção de entregas de código que ocorreram nesse período. Caso neste período não tenham sido feitas entregas de código é expectável que a execução do script seja a mesma do dia anterior.

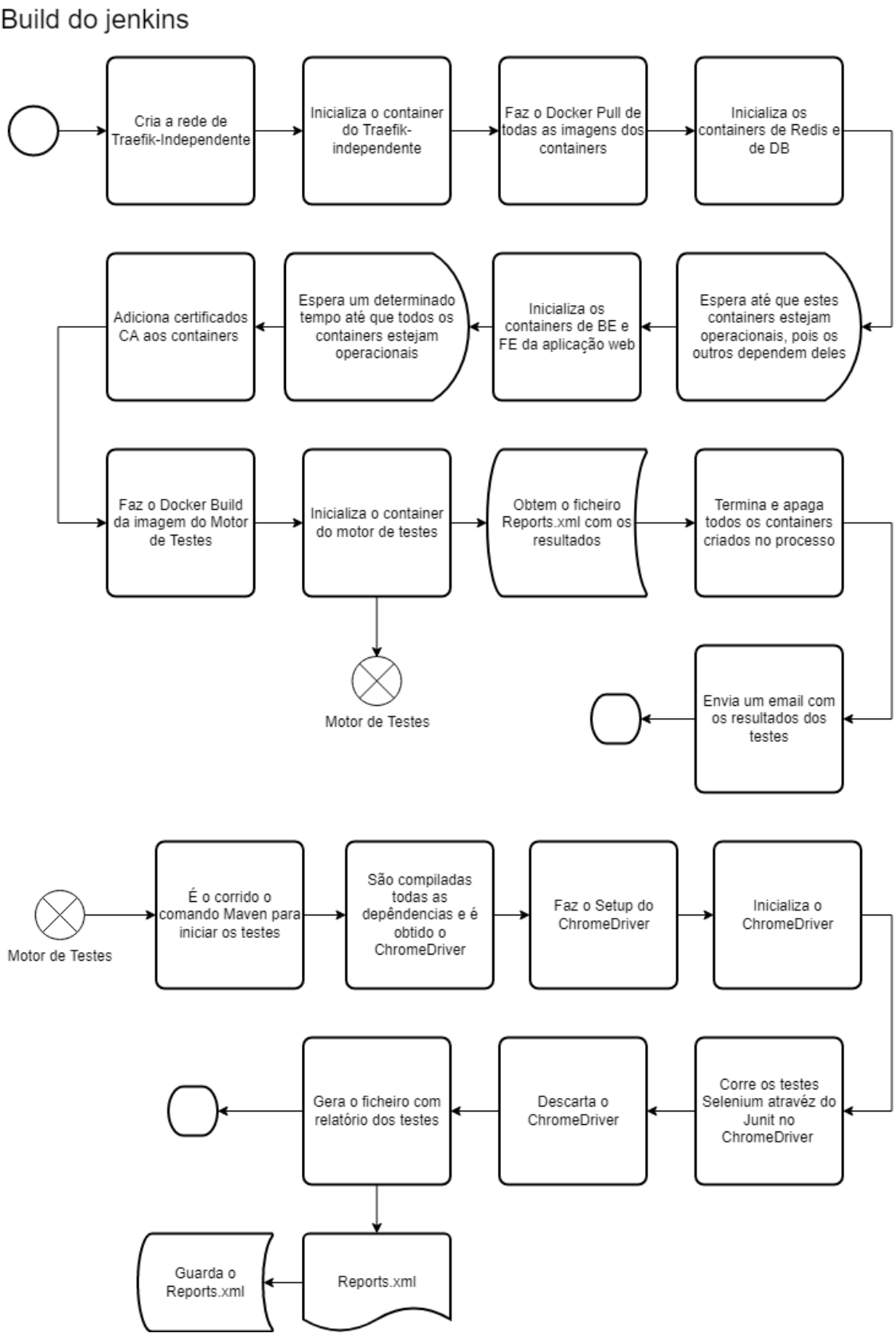

Figura 5.13: Fluxo do script Jenkins

5.3 Reflexão

Atualmente na empresa, estas duas iterações que foram desenvolvidas estão em produção e fazem parte da pipeline.
Por um lado, temos a integração da nova etapa de testes na pipeline, Integrada através do SonarQube que ativamente mede todo o código que é introduzido na pipeline.
Por outro lado, temos a introdução de uma nova etapa que é executada periodicamente, cujo foco é a execução de testes de sistema sobre um ambiente, alterado de período a período, mediante uma coleção de entregas de código que ocorreram durante esse tempo.
Para criar uma perspetiva gráfica sobre como se encontra a pipeline no momento a Figura 5.14, mostra uma possível representação e interpretação da mesma. Como podemos in-

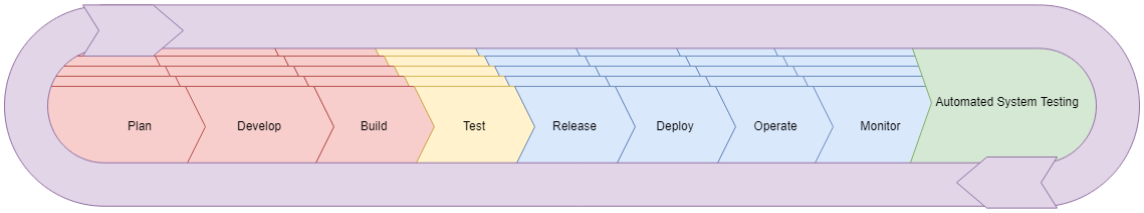

Figura 5.14: Interpretação gráfica das alteraçções à pipeline

terpretar na imagem, temos o ciclo da pipeline normal com a nova etapa de Test, e a nova etapa que corre os testes de sistema (Automated System Testing). Visto que os testes de sistema não são executados a cada entrega de código, é importante salientar que a etapa é aplicada sobre um grupo destas entregas que foram feitas num certo intervalo de tempo.

É também relevante mencionar que a pilha de testes de sistema que está atualmente em uso pela empresa tem sido desenvolvida por um colaborador sem grande experiência de programação, ao qual foi dada uma formação sobre o Selenium IDE e o processo de exportação e importação dos testes. Para além disso foram fornecidos documentos com instruções de como inicializar o ambiente para a criação dos testes.

Este último ponto é muito importante salientar, pois mostra a importância que o DevOps tem numa equipa de desenvolvimento, em que são desenvolvidas estruturas e automatizados processos para melhorar a qualidade do código e da aplicação, neste caso, sem que os outros desenvolvedores se preocupem com a estrutura e aquilo que está por detrás dos processos que facilitam e melhoram a entrega do software.

É complemente irrelevante, por exemplo, para a pessoa que cria os testes de sistema, como está estruturado o ambiente, como é o corrido o script e como é feita a análise dos resultados. A sua única preocupação é apenas a criação e a exportação dos testes para o motor, o resto é tudo um processo automático.

Capítulo 6

Conclusão

Este capítulo simboliza o fim deste projeto realizado na TrustSystems, começa por uma análise do trabalho realizado, posteriormente serão evidenciadas as maiores dificuldades encontradas e finalmente, serão apresentadas sugestões para um trabalho futuro com o intuito de melhorar aquilo que foi desenvolvido.

Neste estudo foi feita a integração e implementação de uma nova etapa de testes automáticos na estrutura da pipeline DevOps da TrustSystems. Esta etapa de testes automáticos foi, na sua implementação, como descrito no documento, dividida em duas partes, uma relativamente ao controlo de qualidade, ao nível de código, utilizando a ferramenta SonarQube e a outra relativamente a testes de sistema utilizando um motor de testes com base na tecnologia Selenium.

Foi analisada a estrutura DevOps da empresa, as tecnologias que nesta integram e a metodologia de como as entregas de software eram feitas. Foi incorporado, em primeiro lugar, o SonarQube num ambiente de produção, que foi conectado ao Jenkins e possibilitou assim, com que no processo de entrega de software se integrasse um novo passo que faz uma análise ao código, e utilizando um quality gate se definissem requisitos para a qualidade do mesmo. Assim, com sucesso, a estrutura DevOps da TrustSystems passou a interromper entregas de código que apresentam codigo instável ou pouco confiável.

Posteriormente foi desenvolvido um motor de testes de sistema com base na tecnologia do Selenium, utilizando uma coleção de outras tecnologias. Com isto foi alcançado um processo que executa testes de sistema num ambiente, em que este processo pôde também ser automatizado estrutura DevOps existente. Aqui foi alcançado, como pretendido, um bom balanço daquilo que é o desenvolvimento de software em DevOps, onde o produto desenvolvido pôde ser utilizado por outros desenvolvedores, sem estes terem que obrigatoriamente perceber como este funciona. Em que este mesmo produto melhorou o seu processo de desenvolvimento de código.

Outro exemplo deste tipo de desenvolvimento DevOps foi alcançado, com o desenvolvimento do motor de testes, onde colaboradores sem grande experiência, a nível de programação, conseguiram ser capazes de criar testes de sistema utilizando o Selenium IDE e fazendo a exportação para o motor de testes.

Finalmente, é importante salientar que todos os objetivos propostos foram cumpridos. E que no atual momento estes processos desenvolvidos encontram-se em produção, e fazem parte da estrutura DevOps de um dos projetos de software da empresa.

6.1 Dificuldades encontradas

Uma das principais dificuldades deste projeto foi aprender a utilizar um grande número de ferramentas e tecnologias. Especialmente o Jenkins na criação de scripts de raiz para o processo do motor de testes de sistema. Outra grande dificuldade, e foi onde precisei mais de ajuda, foi na construção de um ambiente local que simula a pipeline, onde tive imensas dificuldades com a criação dos ambientes utilizando o Docker com o Traefik, devido aos certificados para garantir o HTTPS.
Aprender todos os processos de entrega que a empresa utilizava tal como a perceção de como funcionavam os seus ambientes foi também muito difícil.
Finalmente, e mais geral, comecei este projeto a saber pouco sobre o DevOps, estudar os seus conceitos, perceber as suas filosofias e perceber como aplicá-las na prática foi algo que dificultou-me imenso todo este processo de desenvolvimento.

6.2 Trabalho futuro

Como mencionado na parte da implementação do motor de testes, relativamente à criação dos testes, estes são feitos num ambiente local com bases de dados vazias. Visto que existem aplicações que são compostas por vários componentes, estes ambientes podem tornar-se pesados demais para serem corridos numa máquina normal. O que pode tornar o processo de criação de testes automáticos lento e desagradável. Para tal uma possível melhoria seria criar um ambiente extra, para além dos quatro existentes, numa máquina de produção, com bases de dados vazias e descartáveis, cujo objetivo seria mesmo o processo de criação de testes. No entanto este ambiente pode também ser reutilizado de certa forma para os testes de ambiente serem corridos, no seu processo regular automático.
Esta melhoria tem três vertentes positivas, em primeiro lugar torna o processo de criação de testes num processo mais agradável, em segundo reduz ainda mais a quantidade de informação que um colaborador, que faça testes, precise de ter para criar os próprios testes e em último reutiliza-se o ambiente onde se correm os testes e onde se criam, poupando assim em níveis de computação.

Bibliografia

[1] Code quality and code security, 2021.

[2] Docker frequently asked questions (faq), Dec 2021.

[3] *ISO/IEC/IEEE 24765:2010(E): Systems and software engineering – Vocabulary*. IEEE., 2021.

[4] Metric definitions, 2021.

[5] Minimum acceptable, 2021.

[6] Quality profiles, 2021.

[7] The selenium browser automation project, 2021.

[8] Free and open source code quality and security ide extension, 2022.

[9] Archiveddocs. Test early and often.

[10] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. *2013 35th International Conference on Software Engineering (ICSE)*, 2013.

[11] Baeldung. Code coverage with sonarqube and jacoco, Dec 2021.

[12] Baeldung. Intro to jacoco, Aug 2021.

[13] Anirban Basu. *Software quality assurance, testing and metrics*. PHI Learning Private Limited, 2015.

[14] Tobias Baum, Hendrik Leßmann, and Kurt Schneider. The choice of code review process: A survey on the state of the practice. *Product-Focused Software Process Improvement Lecture Notes in Computer Science*, page 111–127, 2017.

[15] Tobias Baum, Olga Liskin, Kai Niklas, and Kurt Schneider. Factors influencing code review processes in industry. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016.

[16] Larry Brader, Howard F. Hilliker, and Alan Cameron Wills. *Testing for continuous delivery with Visual Studio 2012*. Microsoft, 2012.

[17] Lionel Briand and Yvan Labiche. A uml-based approach to system testing. *Software and Systems Modeling*, 1(1):10–42, 2002.

[18] G. Ann Campbell. Cognitive complexity. *Proceedings of the 2018 International Conference on Technical Debt*, 2018.

[19] Andrej Dyck, Ralf Penners, and Horst Lichter. Towards definitions for release engineering and devops. *2015 IEEE/ACM 3rd International Workshop on Release Engineering*, 2015.

[20] By: IBM Cloud Education. What is java spring boot?

[21] Nicole Forsgren and Mik Kersten. Devops metrics. *Communications of the ACM*, 61(4):44–48, 2018.

[22] Pixel-KillerPixel-Killer 18111 gold badge11 silver badge88 bronze badges and G. Ann SonarSource TeamG. Ann SonarSource Team 21.1k44 gold badges3333 silver badges6060 bronze badges. Sonarqube: Qualify cognitive complexity, Aug 1965.

[23] Paul Hamill. *Unit test frameworks*. OReilly, 2004.

[24] Dorota Huizinga and Adam Kolawa. *Automated defect prevention: best practices in software management*. Wiley-Interscience, 2007.

[25] Marick. How to misuse code coverage. *1999 Reliable Software Technologies*, 1999.

[26] Varun Mittal and Shivam Aditya. Recent developments in the field of bug fixing. *Procedia Computer Science*, 48:288–297, 2015.

[27] Martyn A. Ould and Charles Unwin. *Testing in software development*. Published by Cambridge University Press on behalf of the British Computer Society, 1994.

[28] Roberto Pietrantuono, Antonia Bertolino, Guglielmo De Angelis, Breno Miranda, and Stefano Russo. Towards continuous software reliability testing in devops. *2019 IEEE/ACM 14th International Workshop on Automation of Software Test (AST)*, 2019.

[29] Gerald Schermann, Jurgen Cito, Philipp Leitner, and Harald C. Gall. Towards quality gates in continuous delivery and deployment. *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, 2016.

[30] Sein, Henfridsson, Purao, Rossi, and Lindgren. Action design research. *MIS Quarterly*, 35(1):37, 2011.

[31] Jens Smeds, Kristian Nybom, and Ivan Porres. Devops: A definition and perceived adoption impediments. *Lecture Notes in Business Information Processing Agile Processes in Software Engineering and Extreme Programming*, page 166–177, 2015.

[32] Sam Brannen Stefan Bechtold.

[33] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. When and why your code starts to smell bad. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, 2015.

[34] Filippos I. Vokolos and Elaine J. Weyuker. Performance testing of software systems. *Proceedings of the first international workshop on Software and performance - WOSP 98*, 1998.

[35] Mandi Walls. *Building a DevOps culture*. OReilly Media, 2013.

[36] E.j. Weyuker and F.i. Vokolos. Experience with performance testing of software systems: issues, an approach, and case study. *IEEE Transactions on Software Engineering*, 26(12):1147–1156, 2000.

[37] Tao Xie, Kunal Taneja, Shreyas Kale, and Darko Marinov. Towards a framework for differential unit testing of object-oriented programs. *Second International Workshop on Automation of Software Test (AST 07)*, 2007.

[38] Peter Zimmerer. Strategy for continuous testing in idevops. *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, page 532–533, 2018.