INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Testes em Continuous Delivery

RUI FILIPE VIEIRA MONTEIRO
Outubro de 2017

POLITÉCNICO
DO PORTO

Instituto Superior de Engenharia do Porto

Testes em *Continuous Delivery*

Rui Filipe Vieira Monteiro

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

Orientadora: Isabel Azevedo

Supervisor: Cristiano Cunha

Porto, Outubro 2017

Resumo

Com a evolução do desenvolvimento de *software* nos últimos anos, novos fatores foram equacionados pelos intervenientes desta vertente. A exigência do cliente final e a urgência, muitas vezes, de solucionar problemas nos ambientes de produção requer que uma nova versão da aplicação seja testada e disponibilizada rapidamente. A utilização de recursos para atingir este requisito deve também ser eficiente o que requer que os métodos utilizados devam ser convenientemente monitorizados de forma a otimizar a sua utilização. Com isto, o *Continuous Delivery* e abordagens a este conceito têm ganho uma grande notoriedade nas grandes empresas, sendo já um fator chave de destaque no mercado.

Nesse sentido, foi realizado um estudo aprofundado e avaliados conceitos relacionados com o *Continuous Delivery*, mais concretamente na descoberta de oportunidades de experiencias e aprendizagem. Após alguma investigação a monitorização dos testes ganham relevância e apresentou uma hipótese de colmatar lacunas naquele que era o processo de *Continuous Delivery* da empresa. Assim, métricas e outras abordagens a aplicar ao *Continuous Delivery* foram estudadas e utilizadas para melhorar a qualidade dos processos existentes na empresa. É então definido um guia a aplicar num determinado projeto, que poderá ser reutilizado ou adaptado em outros projetos.

Os resultados apresentados por este trabalho e a avaliação da solução permitem afirmar que o resultado deste projeto teve um contributo cientifico e técnico relevante não só para a empresa, mas como também para a área deste trabalho de mestrado.

Palavras-chave: *Countinuous Delivery,* Testes, Monitorização, Qualidade de *Software*, Desenvolvimento de *Software*

Abstract

With the evolution of software development in recent years, new factors were addressed by the stakeholders of this area. The end customer's requirements and the urgency often to troubleshoot production environments requires that a new version of the application be tested and made available quickly. The use of resources to achieve this requirement must also be efficient which requires that the methods used should be conveniently monitored in order to optimize their use. With this, the Continuous Delivery and approaches to this concept have gained a great notoriety in the big companies, being already a key factor of prominence in the market.

In this sense, an in-depth study was carried out and concepts related to Continuous Delivery were evaluated, more concretely in the discovery of opportunities of experiences and learning. After some investigation the monitoring of the tests gain relevance and presented a hypothesis to fill gaps in what was the process of Continuous Delivery of the company. Thus, metrics and other approaches to Continuous Delivery have been studied and used to improve the quality of existing processes in the company. It is then defined a guide to be applied in a particular project, which can be reused or adapted in other projects.

The results presented by this work and the evaluation of the solution allow to affirm that the result of this project had a relevant scientific and technical contribution not only for the company but also for the area of this master's work.

Keywords: *Countinuous Delivery, Testing, Monitoring, Software Quality, Software Development*

Agradecimentos

Este trabalho é o culminar de um ciclo de 6 anos de aprendizagem no ensino superior. Ao longo destes anos muitas foram as pessoas que tiveram um contributo importante, direta ou indiretamente, que me permitiu alcançar os objetivos e a realizar mais uma etapa da minha formação académica. Deixo aqui o agradecimento com breves palavras a algumas destas pessoas.

À minha namorada, Susana Daniela Pinto Santos, por toda a motivação e companhia prestada e essencialmente pelo enorme isentivo dado para que tudo isto fosse possível.

Um agradecimento especial à minha família, principalmente aos meus pais e irmã por toda a dedicação que sempre tiveram comigo e por todo o esforço aplicado para que este curso fosse uma realidade.

À minha orientadora, Isabel de Fátima Silva Azevedo, por todo o apoio prestado ao longo deste ano letivo, por todos os conselhos e opiniões que contribuíram de forma significativa para melhorar a qualidade deste documento.

Um agradecimento também ao ISEP e a todos os seus docentes que desde cedo mostraram a sua competência e sempre me forneceram todos os meios necessários para que conseguisse alcançar os objetivos.

Por fim e não menos importante, à Farfetch e à minha supervisão composta por Cristiano Cunha e Fernando Lima, o meu muito obrigado pela oportunidade e pela partilha de todo o conhecimento e experiência essencial à realização deste trabalho.

Índice

Lista de Figuras

Lista de Tabelas

Acrónimos e Siglas

Lista de Acrónimos e Siglas

BVT	*Build verification tests*
CD	*Continuous Delivery*
CI	*Continuous Integration*
CPU	*Central Processing Unit*
GUI	*Graphical User Interface*
MTTD	*Mean Time to Detect*
MTTR	*Mean Time to Repair*
MVP	*Minimun Viable Product*
TFS	*Team Foundation Server*
TTA	*Time to approve*
TTL	*Time to live*
UI	*User Interface*

1 Introdução

Neste primeiro capítulo descreve-se, de forma sucinta, o problema principal que se pretende analisar e resolver e, de que forma é que a sua resolução poderá acrescentar valor à empresa. Também é contextualizada a empresa para a melhor perceção da importância e impacto da solução e dos resultados esperados.

1.1 Contexto

A Farfetch é uma empresa tecnológica, fundada em 2008, por um empreendedor português e conta já com escritórios em nove países de todo o mundo e com cerca de 500 pessoas na área de desenvolvimento tecnológico. Com o objetivo de desenvolver o melhor sítio para vender moda, a empresa cresceu fornecendo aos seus clientes a possibilidade de comprar roupas de marcas de luxo em *boutiques* de todo o mundo, como por exemplo, Tóquio, Toronto, Milão e Miami. Desta forma, existe à disposição no *site* uma variedade enorme de peças de marcas de luxo estabelecidas para os mais interessantes novos *Designers*. A empresa conta já com vendas dos seus parceiros para mais de 190 países demonstrando assim a sua dimensão neste mercado.

Num único portal aparecem agregadas várias *boutiques* (lojas que vendem os seus produtos no *site* da Farfetch) e marcas de luxo para que desta forma seja possível proporcionar aos seus clientes uma experiencia única no que toca à compra de artigos de moda e desta forma permitir o crescimento deste mercado que irá ajudar a impulsionar esta marca Farfetch que começa já a ocupar o seu espaço no mercado.

Sendo um ponto em que as *boutiques* colocam os seus produtos à venda, é importante saber a relevância e o impacto que cada contrato celebrado pela empresa tem sobre a sua forma de atuar. A Farfetch conta já com mais de 500 *boutiques* a vender no seu *site* o que demonstra a confiança e a capacidade que a empresa tem de reter clientes que é um passo fundamental para o sucesso do seu negócio.

Sendo a Farfetch especialista no desenvolvimento de *software*, procura utilizar as melhores técnicas e métodos de desenvolvimento de forma e implementar processos que sejam capazes de responder às necessidades que vão surgindo. Isto leva a que o propósito deste estudo seja propor um processo capaz de responder a problemas detetados no processo de *Continuous Delivery* da empresa, balanceando o esforço aplicado na sua aplicação e tendo em conta aspetos como métricas, tipos e níveis de testes que sejam capazes de permitir, no menor tempo possível, validar aplicações e desta forma reduzir o seu tempo de aprovação. A automação de processos com *Continuous Delivery* é prática na empresa e consensual entre todos os colaboradores o que leva a que o esforço seja coeso para que exista um processo equilibrado e com valor para todos os setores tecnológicos da empresa.

Uma empresa tem 600 funcionários dedicados ao desenvolvimento de software. A constante mudança e o crescimento rápido com vários colaboradores novos num curto período de tempo, colocam barreiras à implementação de um sistema de entrega de *software* com alto nível de automação comum a todos sem que existam problemas a resolver.

A empresa procura, através de diferentes plataformas, tecnologias e métodos de trabalho, implementar um processo contínuo e automático de integração de *software* com o intuito de validar a qualidade do *software* desenvolvido. A deteção atempada de problemas com imediata correção dos mesmos permite reduzir o tempo de aprovação de uma *build* aplicacional e com maior confiança por parte dos responsáveis.

1.2 Problema

Uma empresa, ao implementar e desenvolver um processo automático de integração e entrega de *software,* tem a expectativa que este seja um processo rápido e eficaz, o que nem sempre acontece.

Na empresa existe a ambição de validar e aprovar as *builds* aplicacionais desenvolvidas num período inferior a quatro horas. No entanto, este valor não tem sido atingido.

Foi efetuado um esforço desequilibrado no desenvolvimento de testes automáticos que criou instabilidade no processo de validação de *software* que precisa de ser revisto e balanceado para que os tempos propostos sejam alcançados. A falta de métricas para acompanhar a progressão e a melhoria dos processos implementados também é evidente, o que dificulta a análise dos especialistas e atrasa a implementação de uma solução que permita entregar rapidamente *software* sem comprometimento da qualidade.

1.3 Objetivos

O principal objetivo deste trabalho é propor um modelo para a monitorização em *Continuous Delivery* para entrega de software de qualidade num espaço temporal curto. Este objetivo deverá ser concretizado para um projeto da empresa e a qualidade deverá ser aferida pela utilização de diversos testes e métricas. Pretende-se num espaço de tempo relativamente curto, aplicar um processo que poderá ser então consolidado, depois de analisados os seus resultados.

Assim, pretende-se definir uma proposta para a consideração de testes em determinadas fases de desenvolvimento juntamente com métricas com balanceamento da execução de diferentes tipos de testes (teste unitário, teste de componente, teste de Integração, teste de sistema, teste de aceitação) de forma complementar. Isto permite avaliar alternativas para a garantia de qualidade das versões disponibilizadas.

Outros dois objetivos que suportam o objetivo principal enunciado anteriormente são:

- Estudo de diversos testes de aplicações e a as suas características;
- Identificação e caracterização alargada de diversas métricas que permitam verificar o processo de desenvolvimento de software num ambiente de *Continuous Delivery.*

Este trabalho, apesar de se debruçar sobre problemas que existirão em várias empresas, está especialmente direcionado para as problemáticas existentes numa determinada empresa, devido ao facto de um processo de *Continuous Delivery* não seguir uma abordagem única, mas sim ser desenvolvido para empresas com requisitos e tipos de problemas que podem ser considerados muito particulares. No entanto, pretende-se que este trabalho possa contribuir para a implementação de processos semelhantes noutras organizações.

Existem alguns riscos conhecidos no trabalho que se propõe realizar. Assumindo que a solução proposta é adequada, poderá ser necessário que os colaboradores da empresa detenham algumas competências. Eventuais deficiências poderão requerer formação em determinadas áreas.

A aplicação escolhida para implementação da solução proposta e verificação dos resultados obtidos é a Portal *Slice PDP*, que consiste na separação do Portal da Farfetch em vários módulos independentes de forma a permitir *releases* mais simples controlando o impacto causado pelas mudanças efetuadas.

1.4 Abordagem preconizada

A realização do trabalho foi composta por três grandes fases, nomeadamente, o estudo do problema apresentado, conceitos e tecnologias envolventes no tema e que são importantes para a realização do mesmo, desenvolvimento de uma solução e a sua avaliação.

Na fase de estudo do problema, o maior foco passou por levantar e definir todos os conceitos e tecnologias essenciais à correta interpretação do trabalho. Para que seja possível o entendimento de algumas medidas tomadas no decorrer do trabalho esta fase estuda também o contexto, o valor e as oportunidades que poderão ocorrer na fase de desenvolvimento.

Este estudo permitiu então que fosse possível desenvolver uma solução que respondesse ao problema descrito anteriormente recorrendo às melhores práticas definidas no estudo efetuado do problema.

Por fim, a avaliação da solução implementada foi efetuada de modo a entender se o seu valor foi de encontro com as expectativas inicias e se responde corretamente às necessidades que envolviam o problema.

1.5 Estrutura

Este documento está organizado em 7 capítulos:

- Capítulo 1, o presente capítulo, apresenta, para além da abordagem preconizada ao problema, o contexto do problema em causa, os objetivos e resultados esperados no final do trabalho desenvolvido.
- Capítulo 2, Estado de arte, aborda o âmbito do trabalho analisando conceitos importantes e abordagens existentes que poderão ser úteis e incluídas na solução a apresentar.
- Capítulo 3, Análise de valor, efetua a análise de valor da proposta de solução para a empresa. Através do método *The new concept model* é possível identificar as várias fases de geração de valor.
- Capítulo 4, Estado atual, tem o objetivo de apresentar o estado atual do processo da empresa para que seja possível identificar de forma mais clara os pontos de melhoria e quais as reais necessidades da empresa.
- Capítulo 5, Definição da solução, apresenta as condições e abordagens a tomar na implementação da solução, como o desenho de uma proposta de *pipeline* que servirá de base posteriormente para a monitorização de dados com a aplicação de métricas adequadas a cada uma fase de testes.
- Capítulo 6, Avaliação de resultados, apresenta quais métodos a aplicar para verificação e análise de resultados referindo os estudos estatísticos a aplicar na solução para posterior análise.
- Capítulo 7, Conclusões, para além de apresentar as conclusões finais do trabalho elaborado refere o trabalho futuro a ter em conta neste trabalho e uma apreciação final e global do trabalho elaborado.

Este documento tem ainda os seguintes anexos:

- Anexo A, métricas recolhidas em Maio, apresenta os dados recolhidos com as métricas no mês de Maio relativos ao tempo de execução dos testes de integração, ao tempo de execução dos testes unitários, à quantidade de testes de integração, à quantidade de testes unitários e à percentagem de cobertura de código.
- Anexo B, métricas recolhidas em Julho, apresenta os dados recolhidos em Junho relativos às mesmas métricas do mês de Maio.
- Anexo C, métricas de *time to approve e time to live*, apresentam os dados recolhidos com as métricas *TTA* e *TTL* bem como a sua variação e informação de cada uma das suas fases.

2 Estado de arte

Neste capítulo caracteriza-se a área do trabalho a desenvolver com a descrição de vários conceitos, abordagens seguidas em várias empresas com o propósito de garantirem a entrega de produtos de qualidade num espaço o mais curto possível.

2.1 Conceitos

Neste capítulo serão descritos conceitos importantes relacionados com o problema a resolver e a abordagem utilizada na empresa para desenvolvimento de software.

2.1.1 *Continuous Delivery*

Várias abordagens têm sido propostas para garantir continuidade em alguma fase do desenvolvimento de software ou mesmo em várias. A expressão "engenharia contínua de software" tem sido utilizada para englobar várias dessas abordagens. *"O Continuous Integration requer um vínculo entre desenvolvimento e operações e, portanto, é muito relevante para o fenômeno DevOps. Dentro do Continuous Integration, podem ser identificados vários outros modos de atividades contínuas, nomeadamente Continuous Deployment e Continuous Delivery."*, referido por Brian Fitzgerald e Klaas-Jan Stol [1]. Sendo *Continuos Delivery* a mais relevante no âmbito do trabalho descrito neste documento.

Continuous Delivery (CD) é a prática de desenvolvimento de *software* que permite que a qualquer momento uma determinada versão de uma aplicação possa ser instalada nos ambientes de produção [2]. Apresenta ainda as seguintes características [3]:

- O *software* desenvolvido é instalado de forma contínua ao longo de toda a fase de desenvolvimento;
- As equipas dão prioridade em manter o *software* desenvolvido nas condições necessárias para ser instalado a qualquer momento;
- Qualquer pessoa pode obter *feedback* rápido e automatizado sobre o estado do ambiente de produção sempre que alguém efetua uma alteração;
- As instalações de versões desenvolvidas são possíveis através de um simples *click* num botão que inicia um processo automático de instalação.

Assim, *Continuous Delivery* é alcançado quando as equipas de desenvolvimento integram o *software* desenvolvido de forma contínua correndo testes automáticos capazes de validar o código produzido por estas equipas. Através de uma *pipeline* efetuar uma validação e instalação automáticas em vários ambientes até ao de produção evitando assim problemas quando for a vez de a aplicação ser instalada em produção.

A principal vantagem é um responsável de negócio pedir os desenvolvimentos em que serão instalados a qualquer altura nos ambientes de produção, com *Continuous Delivery* isso é possível e não levanta problemas nas equipas de desenvolvimento [3].

Posto isto, e segundo Nikhil Pathania [4], as práticas de *Continous Delivery* acentam sobre estas práticas:

- Uma boa estratégia de ramos;
- Um processo de *Countinuous Integration*;
- *Builds* distribuídas;
- Testes automáticos;
- Testes paralelos ou distribuídos;
- Provisionamento rápido e automático de ambientes;
- Promoção de código automática.

Para que seja percetível cada um destas práticas os próximos tópicos descrevem cada um deles.

2.1.1.1 *Estratégia de ramos*

Para o *Continuous Integration "trabalhar sobre um único ramo de master é a melhor abordagem. No entanto, a utilização de multiplos ramos para o desenvolvimento de software é mais produtivo do que fazer tudo no mesmo ramo."*, referido por Nikhil Pathania [4]. Na Figura 1, está ilustrada uma abordagem que representa uma abordagem de estratégia de ramos com multiplos ramos *feature.*

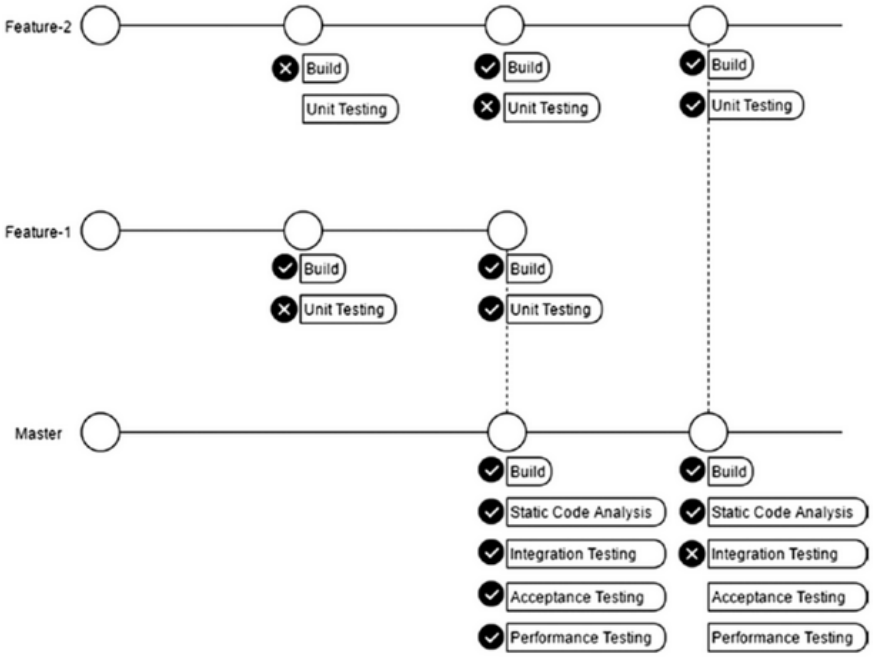


Figura 1 – Estratégia de ramos com ramos *feature*
Fonte: Pro Continuous Delivery: With Jenkins 2.0 [4]

No *workflow* representado na Figura 1 os *developers* fazem os seus desenvolvimentos nos ramos de *feature* que representam a linha de desenvolvimento de uma nova funcionalidade. Cada ramo de *feature* é criado a partir da última versão instalada em produção disponível no ramo de *master.*

Apesar de serem ramos separados, os ramos *feature* recebem todos os princípios de *Continuous Integration* que recebe o ramo principal. Posto isto, só será efetuado um *merge* para *master* caso todos os testes sejam executados com sucesso, apesar de na imagem estarem apenas ilustrados os testes unitários.

Após o *merge* caso algum teste falhe no ramo principal, é muito importante que o próximo commit seja com o intuito de corrigir este problema. Caso o problema não seja detetado rapidamente o novo código deverá ser retirado de *master* e os desenvolvimentos deverão prosseguir.

2.1.1.2 *Processo de Continuous Integration*

O processo de *Continuous Deployment & Integration* é o processo capaz de demonstrar que um *software* funciona. Sem este processo, a aplicação fica parada e partida até que alguém comprove que funciona. A ideia inicial do processo de *Continuous Integration (CI)* é que se a integração regular do código é boa, porque não faze-lo sempre? [5] Desta forma, com uma integração tão frequente das aplicações existe uma facilidade maior em detetar possíveis falhas no *software* o que torna o desenvolvimento mais rápido, com maior qualidade, com um custo menor e permite às equipas tornarem-se mais reativas ao estado da aplicação [6]. *"Continuous Integration doesn't get rid of bugs, but it does make them dramatically easier to find and remove." – Marin Fowler, Chief Scientist, ThoughtWorks.*

Posto isto, a maior vantagem de *CI* é o facto de este processo reduzir consideravelmente o custo do desenvolvimento de *software*. No entanto, existem outras vantagens na utilização de *CI* nas empresas tais como [7]:

- Integrações rápidas de *software*;
- Aumento da visibilidade do estado de uma aplicação;
- Deteção rápida de problemas na aplicação em causa;
- Menos tempo investido na análise de problemas;
- Maior tempo para desenvolvimento de novos requisitos;
- Redução dos problemas de integração visto ser possível entregar *software* mais rapidamente.

Apesar de representar uma série de vantagens para as empresas, *CI* requer um conjunto de boas práticas que utilizadas permitem tirar o maior partido destes processos e desta forma aumentar a sua produtividade. Deverá ser utilizado um único repositório de código que será compilado e instalado de forma automática em ambientes em tudo semelhantes aos de

produção. Este repositório deverá conter todo o projeto como: código, testes, *scripts* de base de dados, *build* e *deployment*, e tudo o resto que permita instalar, executar e testar a aplicação. Capacidade de executar uma build de forma automática é importante para que o processo seja capaz de facilmente compilar um projeto para o instalar num ambiente de testes e desta forma correr os testes contra este mesmo ambiente. Para que *CI* funcione deve ainda existir a coordenação de uma equipa que veja este processo como uma prática e não uma ferramenta [5]. Todas a alterações no controlo de versões devem ser pequenas e especificas o suficiente para que os objetivos de *CI* sejam atingidos, caso contrário com um número de elevado de alterações num determinado *commit* será insuportável a sua análise e identificação de problemas em caso de evidências das mesmas.

CI pode ainda ser utilizado com outra prática em que em conjunto com *CI* conseguem aumentar ainda mais as vantagens e colmatar algumas lacunas de *CI*. O *Continuous Deployment (CD)* está diretamente relacionado com *CI* e refere-se à entrega de software, validado através de testes automáticos, para os ambientes de produção. *"it is the practice of releasing every good build to users,"*- Jez Humble, *Continuous Delivery*. Utilizando os dois processos em simultâneo não só será possível reduzir os riscos e detetar as falhas de uma aplicação rapidamente como é possível entregar valor nos ambientes de produção de forma contínua. Com entrega de *software* contínua com baixos riscos é possível adaptar rapidamente a empresa às necessidades de negócio e dos utilizadores [7].

2.1.1.3 *Testes automáticos*

Um dos princípios do *Continuous Delivery* é *"Automate Almost Everything"* [3]. É natural que existam processos que não são possíveis de automatizar. Exemplos disso são os testes exploratórios que requerem experiencia de *testers* e a aprovação de determinadas tarefas são tipicamente feitas por humanos. No entanto, a lista de processos automatizáveis é bastante grande pode trazer muito valor ao processo aplicado.

Os testes são uma das componentes possíveis de automatizar e são claramente um requisito do *Countinuous Delivery.* Testes automáticos permitem a sua execução de forma mais rápida e frequente o que permite detetar falhas mais rapidamente. Com a automatização de testes é possível implementar uma *pipeline* contínua com *feedback* automático e desta forma automatizar a validação de *software.*

Posto isto, a automatização de testes apresenta vantagens para *Continuous Delivery*, como:

- Garantia do que funcionava continua a funcionar com a reutilização de testes;
- Verificação e validação do impacto das mudanças feitas no *software*;
- Testes efetivos em cada versão (regressão);
- Execução de testes com maior frequência;
- Maior cobertura dos testes funcionais em menos tempo.

Desta forma a equipa consegue focar a sua atenção no desenvolvimento de novas funcionalidades porque os testes automáticos irão encarregar-se de validar os impactos dessas alterações na aplicação.

2.1.1.4 *Testes paralelos ou distribuídos*

Nas práticas de *Continuous Delivery* a economização de tempo é algo essencial para uma rápida e eficaz validação de *software*. Tipicamente as equipas querem validar o seu código em vários cenários e o tempo perdido, por vezes, pode reduzir as aspirações de o conseguirem. Quando o isolamento dos testes é bom, uma possibilidade de acelerar o processo e economizar recursos passa por executar testes em paralelo. Testar em paralelo *"é o processo de execução de vários casos de teste em múltiplas combinações de sistemas operacionais e navegadores ao mesmo tempo"*, segundo Lubos Parobek, VP de produto na empresa SauceLabs [8].

Analisar e implementar testes paralelos num projeto é útil e segundo Lubos Parobek [8] estas são as principais vantagens de investir neste conceito:

- Testar uma maior compatibilidade da aplicação;
- Reduzir de forma significativa o tempo gasto em testes;
- Testes paralelos otimizam os conceitos de *Continuous Integration* e *Continuous Delivery;*
- O custo de cada um dos testes é menor;
- Suporta ciência de testes;
- Os testes sequenciais não desaparecem, podem continuar a ser executados;
- Os testes em paralelo podem ser executados gradualmente em caso de mudança.

Posto isto, quanto mais rápido um teste executar e quantos mais testes forem executados mais frequentes vão ser os lançamentos de novas versões e mais defeitos serão encontrados. Para além dos tempos de os testes serem mais curtos, o consumo da infraestrutura será menor o permitirá economizar o seu custo. Para além disto, o custo de longos ciclos de testes, os custos de lançamentos de novas versões atrasadas e o custo de clientes insatisfeitos devido à má qualidade do *software* irá também reduzir.

2.1.1.5 *Provisionamento rápido e automático de ambientes*

Numa *pipeline* automática, o provisionamento automático de ambientes é também importante para que seja possível instalar a aplicação com perfis distintos em cada um deles ou simplesmente executar testes de forma paralela contra ambientes distintos.

O provisionamento de ambientes trás vantagens para o *Continuous Delivery*, tais como:

- Velocidade e agilidade – Definições de serviços modelares para definir e modificar facilmente o serviço;
- Flexibilidade – Criação, atualização ou remoção da infraestrutura ou serviço;
- Maior controlo – Visibilidade das operações, controlo e *rol-back* de alterações;
- Processamentos totalmente automatizado – Alterações aos ambientes automatizados.

Desta forma e com a criação automática de ambientes para as várias fases de teste é possível reduzir o tempo *end-to-end* desde o requisito de iniciação da *pipeline* até que é instalada nos ambientes de produção.

2.1.2 *Pipeline*

Uma *pipeline* é um conjunto de passos automáticos que, em conjunto, são capazes de transformar o código de um repositório de versionamento num produto possível de entregar aos utilizadores ou clientes. O principal objetivo de uma *pipeline* é validar, através de um processo complexo, todas as alterações num *software* antes de este ser entregue. Este processo engloba a compilação do projeto em vários passos precedido de uma ou múltiplas etapas de testes e instalações [5].

A *pipeline* tem a capacidade de facilitar o trabalho das equipas porque é bastante flexível e pode ser implementada com um índice de complexidade ajustável com o que poderá fazer sentido em cada uma das aplicações. É a *pipeline* que permite através dos seus processos automáticos transformar a ideia de negócio em algo que traga valor ao cliente. Através da possibilidade de configurar uma *pipeline* capaz de executar testes automáticos contra uma nova versão de *software* desenvolvida é possível obter rápido *feedback* sobre possíveis erros na aplicação reduzindo assim o custo do desenvolvimento da mesma aplicação. Através da Figura 2, é possível perceber a importância do *feedback* de uma *pipeline* para reduzir o custo do desenvolvimento de *software*. Quanto mais rápido as falhas forem detetadas, mais rápida será a sua correção. Isto poupa tempo de desenvolvimento e aumenta a confiança na aplicação.

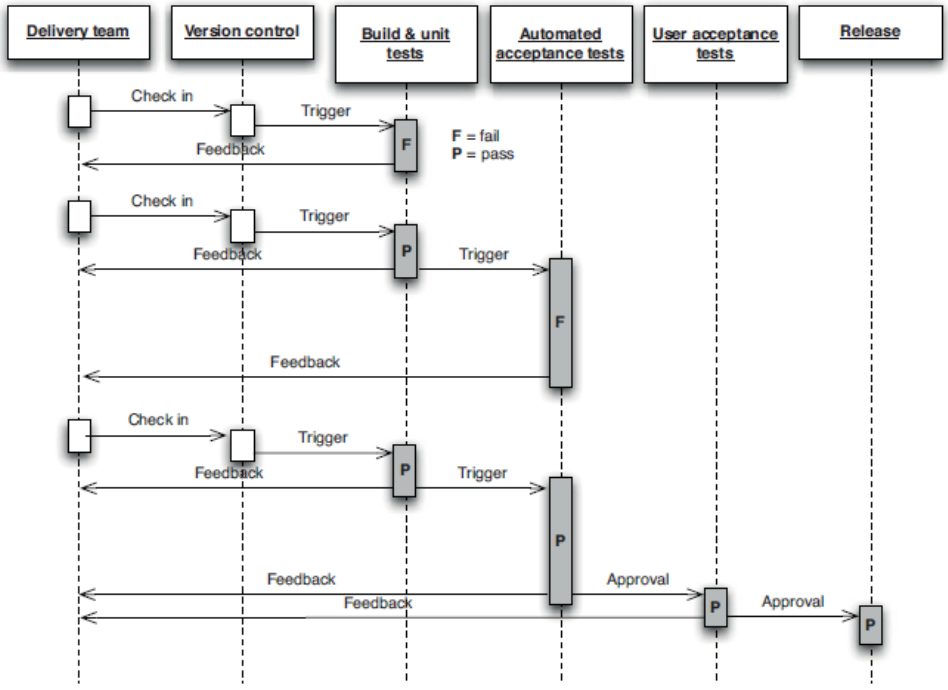


Figura 2 – Mudanças através de uma *pipeline* [5]

Os vários processos implementados numa *pipeline* devem ser rápidos, repetíveis e confiáveis. Isto é importante porque é assumido que uma *pipeline* irá ser executada durante toda a fase de desenvolvimento e posteriormente para correção de problemas, entretanto detetados e só desta forma será sustentável a sua análise para que seja possível analisa-la de forma eficiente.

2.1.3 *Testes e métricas*

2.1.3.1 *Pirâmide de testes*

A pirâmide de testes é um artefacto que ilustra que quanto mais na base da pirâmide os testes se situarem, mais rapidamente e com menos custos as falhas são detetadas e corrigidas [5].

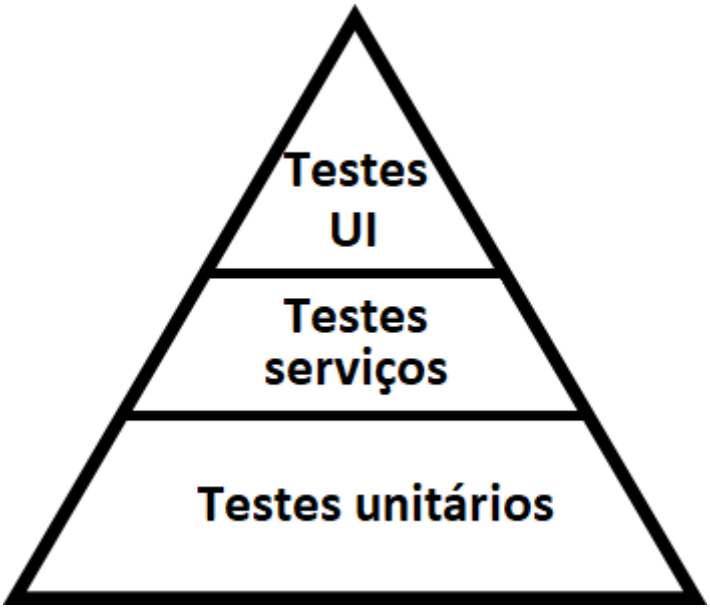


Figura 3 – Pirâmide de testes

A Figura 3 mostra uma pirâmide habitual em que testes de baixo nível são em maior número e realizados mais cedo. Com os testes unitários é possível atingir rapidamente uma elevada percentagem de cobertura do código desenvolvido com testes a pequenos blocos de código de forma isolada. Assim, determinadas falhas são identificadas numa fase precoce do processo antes de se efetuar a integração com outros serviços.

Mais perto do topo da pirâmide a complexidade dos testes aumenta. Normalmente são iniciados os testes de integração com outros serviços que são mais suscetíveis a falhas. Tipicamente na camada de serviços e de *UI* é possível dividir os testes efetuados em diferentes níveis de teste que permitem definir ainda mais os testes a desenvolver e a sua importância para o processo.

2.1.3.2 *Níveis de testes*

Os níveis de testes são importantes e bastante úteis para definir um processo continuo numa *pipeline* em que a intenção é garantir confiança na aplicação ao longo do tempo com a execução dos diferentes níveis de testes.

2.1.3.2.1 *Testes unitários*

Um teste unitário, escritos e executados por *developers*, é a menor parte testável de uma aplicação em que engloba funções, classes e interfaces. Este teste é o método pelo qual unidades individuais do código fonte são testadas para verificar se são aptas para serem utilizadas. Isto significa que o objetivo deste teste seja separar cada parte da aplicação e testa-las de forma independente e isolada. O que para qualquer função ou procedimento quando executadas quando determinados valores deverá retornar sempre valores apropriados.

As principais vantagens, segundo [9], na utilização de testes unitários são:

- Qualquer problema é encontrado numa fase inicial. Uma vez que os testes unitários são realizados por *developers* onde testam o seu código isoladamente antes de qualquer integração.
- Ajuda a manter e a alterar o código. Isto é possível tornando o código desenvolvido mais interdependente para que os testes possam ser executados. Isto faz com que as hipóteses de impacto nas mudanças introduzidas são reduzidas.
- O custo da correção de defeitos na aplicação é menor porque estes são detetados numa fase muito precoce do desenvolvimento da funcionalidade.
- O teste unitário ajuda no processo de *debug* da aplicação. Isto porque se um teste não tem sucesso apenas as últimas alterações feitas no código precisam de ser analisadas.

2.1.3.2.2 Testes de componente

Os testes ao componente têm como principal objetivo encontrar defeitos e comportamentos inesperados do funcionamento de módulos de *software,* programas, objetos, classes, etc., que são testáveis separadamente. Normalmente estes testes são executados em ambientes controlados e isolados de dependências alheias, para que tal seja possível normalmente removem-se essas dependências com sistemas *mock* que garantam a estabilidade do sistema a ser testado.

2.1.3.2.3 Testes de integração

Nos testes de integração o objetivo é mesmo testar a integração dos nossos componentes num ambiente controlado. Aqui tipicamente simula-se um fluxo de informação em que o maior foco é testar as interações entre diferentes partes do sistema.

A maior dificuldade dos testes de integração é, quando um problema ocorre, conseguir identificar exatamente qual o componente da integração é que possui efetivamente o problema e que medidas são necessárias para o corrigir. É também espectável de quando a integração aumenta o risco de falhas aumente também.

Neste nível devem ser validadas apenas as integrações dos vários componentes das aplicações. Visto que o comportamento isolado do mesmo foi já validado nos testes de componente e desta forma a confiança e qualidade da aplicação vai aumentando.

2.1.3.2.4 Testes de integração de componente

Estes testes executados depois dos testes de componente isolados tem o objetivo de testar as interações entre os componentes do *software* desenvolvido. Apesar dos componentes serem especificados em diferentes fazes devem ser integrados corretamente e funcionar em conjunto. É importante também abordar os casos negativos porque os componentes podem provocar perda de dados se não interagirem corretamente [10].

2.1.3.2.5 Testes de sistema

Depois de aplicação ser validada de forma isolada e integrada é tempo de elevar a fasquia e testar a mesma num ambiente igual ao de produção de forma a reduzir o risco de falhas por requisitos de infraestrutura não encontradas nos testes, é uma forma de validar essas questões.

Testes de sistema podem incluir testes a requisitos de alto nível, processos de negócio, casos de uso, interações com o sistema operativo ou ainda recursos do sistema como a performance.

2.1.3.2.6 Testes de integração do sistema

O teste de integração de sistema, normalmente efetuado após os testes de sistema, testa as interações entre diferentes sistemas. São verificadas as execuções adequadas entre componentes de *software* e uma interface adequada entre os componentes dentro da solução [11].

O teste de integração de sistema é também responsável por validar as dependências do projeto funcionam corretamente e que a integridade dos dados da aplicação é mantida entre módulos separados para toda a aplicação. Como a validação das dependências é um foco deste tipo de teste, normalmente estes testes são incluídos nas regressões para que exista uma camada capaz de validar o sistema.

2.1.3.2.7 Smoke tests

Os *Smoke tests* são testes que têm o propósito de validar apenas as funcionalidades básicas e importantes da aplicação para garantir que a aplicação está com as condições mínimas necessárias para avançar para outros tipos de teste. São testes que são executados, normalmente, após a instalação da aplicação para garantir que não existiu nenhum problema durante o processo. Caso sejam detetados problemas com *Smoke tests* a aplicação está perante sérios problemas de bloqueio que precisam ser corrigidos rapidamente e de forma prioritária.

Estes testes têm as seguintes características [12]:

- Rápidos a executar;

- Automatizados;
- Executados em todas as *builds*, sejam de testes ou não;
- Focados nas funções críticas das aplicações, por exemplo, o login;
- Número muito pequeno de testes;
- Executados em poucos minutos e não horas.

Os Build verification tests (*BVTs)* são uma subcategoria dos *Smoke tests* que cobrem uma maior área da aplicação. Enquanto que os *Smoke tests* validam apenas as funcionalidades criticas da aplicação os *BVTs* têm uma abrangência maior, no entanto, sempre focados com as funcionalidades criticas.

São testes que têm as seguintes características [12]:

- Testes de prioridade mais alta;
- Automatizados;
- Executados em cada *build*;
- Rápidos e executados em minutos.

2.1.3.2.8 Testes de aceitação

O principal objetivo dos testes de aceitação não passa por encontrar defeitos na aplicação. Nesta fase os testes podem até ser efetuados por clientes ou utilizadores do sistema, no entanto, no caso da empresa estes testes são normalmente efetuados por *testers* e qualquer outra pessoa que navegue nas aplicações. O objetivo é sim ganhar confiança na aplicação tendo sempre em atenção aspetos não funcionais definidos e que podem ter um foco maior nesta fase.

Estes testes são tipicamente executados de forma manual e são testes repetitivos que seguem um fluxo documentado que é rígido e que deve ser cumprido, no entanto, o *tester* pode enriquecer estes testes com novos casos de teste que vai identificando como sendo necessários e importantes. Apesar disto, e como são testes exigentes, exigem do *tester* capacidade de executar testes que o computador não é capaz com cenários complexos e com grandes fluxos de dados. [13]

Normalmente, são testes que não podem ser automatizados e que ajudam a entender o estado de maturidade da aplicação. Requer um *tester* experiente e conhecedor da aplicação com capacidades para efetuar os testes.

2.1.3.2.9 Testes Alpha

O teste *alpha* é um teste muito utilizado nas empresas e uma das estratégias que tem ganho maior notoriedade no ramo de testes. São testes executados quando os desenvolvimentos das funcionalidades estão próximos de terminar e são executados normalmente por um grupo independente da equipa do projeto, normalmente *testers* de outras equipas ou até qualquer um outro elemento que tenha interesse nestas mesmas funcionalidades.

O teste *alpha* é o teste final antes do lançamento do *software* para o cliente final e é composto por duas fases, segundo [14]:

- Numa primeira fase o teste *alpha* é executado pelos próprios *developers* em ambientes dedicados com o objetivo de encontrar defeitos o mais rápido possível;
- Numa segunda fase este teste é executado por uma equipa de *testers* para testes adicionais num ambiente semelhante ao de produção.

Portanto, este teste é um teste operacional executado por potenciais utilizadores ou equipas dedicadas para os realizar.

2.1.3.2.10 Testes Beta

Os testes beta são os testes executados nos ambientes de produção em condições reais. O objetivo do teste é instalar as funcionalidades nos ambientes de produção e analisar os resultados das interações dos utilizadores reais com a aplicação. Isto é importante para que seja possível perceber cenários de teste criados pelos próprios utilizadores e desta forma melhorar a qualidade do software.

As principais vantagens de utilizar testes beta, segundo [15], são:

- Permite a oportunidade de colocar a aplicação disponível para potenciais utilizadores antes de disponibilizar para todos;
- Os utilizadores podes testar a aplicação e enviar comentários durante o período de testes;
- Os utilizadores podem encontrar problemas que não foram detetados até esta fase;
- Ter a possibilidade de melhorar aspetos apontados por potenciais utilizadores aumenta o valor da aplicação;
- Os utilizadores que efetuam os testes beta poderão recomendar a aplicação a outros futuros utilizadores.

2.1.3.2.11 Testes exploratórios

Apesar de a automação de testes ser um caminho assumido na empresa, existem outro nível de testes a considerar neste estudo. Os testes exploratórios, têm uma importância significativa porque são testes que podem ser inseridos em qualquer versão da aplicação. Para que os testes exploratórios tenham importância e sejam credíveis é importante que o *tester* seja profundo conhecedor da aplicação e que seja capaz de perceber onde poderão estar os principais pontos de falha da mesma. Testes exploratórios são tipicamente *"um estilo de teste de software que estimula a liberdade pessoal e a responsabilidade do tester individual para otimizar de forma contínua o valor de seu trabalho, tratando a aprendizagem relacionada ao teste, o design do teste, à execução do teste e à interpretação dos resultados do teste como atividades de apoio mútuo que funcionam paralelamente ao longo do projeto."* [16] e têm ganho grande relevância nas metodologias Agile nos últimos anos.

Estes testes são executados manualmente pelo *tester* e segundo o livro *Test better by exploring: Harnessing human skills and knowledge* [13] os testes exploratórios são caracterizados por:

- São executados quando o *tester* não segue um guião e utiliza os seus conhecimentos para gerir estes testes.
- Os testes exploratórios podem ser estruturados e planeados sem restringir a liberdade do *tester* para escolher as melhores maneiras de testar dentro daqueles que são os limites da estrutura disponível.
- Exigem pessoas experientes com conhecimentos conceituados em testes.
- O *tester* deve aprender com os testes realizados de forma a identificar futuros pontos de falha.
- Possuem um *loop de feedback* imediato do resultado de um anterior que permite projetar o próximo teste. Isto permite guiar os testes em base dos resultados que vão sendo encontrados.

Posto isto, os testes exploratórios são uma componente muito importantes dos testes ao *software*. Fornecem benefícios em testes a aplicação com elevada interação de clientes, com um modelo de negocio complexo ou contextos sociais que exigem conhecimentos humanos para entender. São teste ótimos para colmatar áreas não cobertas por testes automáticos.

2.1.3.3 *Tipos de testes*

Os tipos de testes pretendem definir claramente o objetivo de um nível de teste. *"Um tipo de teste é focado num determinado objetivo de teste, que pode ser o teste da função a ser executada pelo componente ou sistema"* [17].

2.1.3.3.1 Testes funcionais

São testes às funções do componente ou subsistema que podem ser requisitos especificados, casos de uso ou uma especificação funcional. No entanto, podem existir testes a casos não documentados [5].

Os testes funcionais são desempenhados, normalmente, pelos *testers* conhecedores da aplicação, pois são testes efetuados a funções ou requisitos da aplicação conhecidos e definidos. São testes que têm como atenção a interoperabilidade das funções em sistemas específicos.

2.1.3.3.2 Testes não funcionais

São testes a requisitos que se inserem na aplicação relacionados com, por exemplo, testes de performance que permitem avaliar a capacidade de utilização da aplicação como utilizadores suportados e ponto de exaustão [5], testes de usabilidade que permitem verificar o comportamento da aplicação na presença de erros [5], testes de manutenção são importantes para perceber o desgaste das matérias utilizadas, testes de reutilização permitem perceber de que forma os componentes se comportam quando reutilizados por outros e testes de portabilidade que avaliam a potencialidade de um componente de ser utilizado em diferentes arquiteturas. São testes à forma como o sistema trabalha.

2.1.3.3.3 Testes estruturais

Os testes estruturais estão relacionados como código e a cobertura dos testes na aplicação. A cobertura dos testes na aplicação é um fator importante a ter em conta porque se a percentagem da cobertura dos testes não for 100% mais testes terão de ser desenhados pelas equipas para colmatar esse facto [5]. Apesar de poder ser aplicado em todos os níveis de testes, os testes estruturais são mais utilizados e fazem mais sentido nos testes de componente devido à maior facilidade e relevância de obter e utilizar estes dados.

2.1.3.3.4 Testes de regressão

Os testes de regressão são importantes para validar que uma alteração ou correção de um defeito efetuada na aplicação não colocou um problema novo numa parte já existente e testada da aplicação. Normalmente adequam-se a quantidade de testes da regressão consoante o risco da alteração a efetuar, no entanto, deverão ser sempre incluídos todos os testes existentes numa regressão para a garantia obtida ser numa percentagem superior.

2.1.3.4 *Métricas*

Num ambiente de *Continuous Delivery* existe uma grande pressão em validar o *software* de forma eficiente e eficaz de forma a garantir o mínimo de falhas para que seja possível aumentar a qualidade do *software* entregue ao cliente. Para tal, é necessário que exista um controlo deste processo através de indicadores que nos consigam dar *feedback* das várias vertentes num determinado momento. Desta forma, é possível agir em conformidade e perceber se o processo está a ser desenvolvido corretamente e se está a ser implementado de acordo com as espectativas. Em projetos de *software* não é possível terminar os mesmos com sucesso se não existir um controlo de métricas de qualidade, custo e eficácia dos processos em questão [18].

Deve estar presente que as métricas tornam-se irrelevantes se retiradas do contexto [19], não existem métricas que façam sentido em todos os casos de todas as empresas, cada caso é um caso e as métricas devem ser adotadas se fizerem sentido na realidade da empresa e se introduzirem valor no processo utilizado de modo a torna-lo mais robusto e monitorizado.

Normalmente as métricas têm o objetivo de:

- Permitir a obtenção de estimativas de custo e calendarização das ações a tomar;
- Suportar decisões com vista a melhorar o processo implementado, inclusive com alteração de tecnologias menos produtivas ou com balanceamento do esforço da equipa;
- Auxiliar a monitorizar a eficácia do processo implementado.

No entanto, os valores por si só não trazem valor ao projeto, porque *"Os testes vão gerar muitos números. Olhados de maneira crua e isolada, esses números serão apenas dados, que muitas vezes podem vir de mais do que uma fonte. Logo, é fundamental que seja feita uma análise para transformar esses números em informação útil que leve à melhor tomada de*

decisão. Isso só é possível com a leitura correta dos dados. Caso contrário, os responsáveis pela decisão ou vão decidir de forma errada ou simplesmente não vão decidir a partir das métricas geradas pelos testes." Referido por (R. Murillo, 2008)

Posto isto, existem métricas que são úteis no controlo do processo de *Continuous Delivery* que aumentam a capacidade de um *Lead* ou responsável técnico pela área ter uma perceção geral dos progressos efetuados no sentido de alcançar os objetivos definidos.

2.1.3.4.1 Código de teste e código aplicacional

O objetivo desta métrica é perceber a evolução do código produzido em testes automáticos com o código desenvolvido na aplicação. Esta é uma métrica que pode ser importante em algumas técnicas de desenvolvimento como o TDD que se espera um valor elevado nesta métrica que irá reduzir ao longo do tempo [20]. Apesar disto, esta métrica é mais relevante para as equipas com técnicas de desenvolvimento tradicional em que o código da aplicação é claramente superior ao código dos testes. Nestes casos, esta métrica mostra-se importante no sentido que ajudará estas equipas a balancear o seu esforço conseguindo assim produzir mais código de testes.

O uso desta métrica é recomendado e a sua não utilização deverá ser substituída por outra métrica que ajude a acompanhar a evolução do código de testes, é uma métrica importante nos processos de integração contínua porque mostra a capacidade dos testes face à aplicação testada. No entanto, a métrica terá de ser acompanhada por uma rigorosa análise aos testes existentes, porque código de testes sem validações, ou *asserts* não trás valor e é um erro inclui-lo nesta monitorização e pode inflacionar os resultados.

Pode ser calculado segundo a seguinte formula [20]:

$$\mathrm{T_i} = \frac{\mathrm{TLOT_i}}{\mathrm{TLOC_i}} \qquad (1)$$

Onde:
$TLOT_i$ = número total de linhas de código de testes na iteração i
$TLOC_i$ = número total de linhas de código aplicacional na iteração i

2.1.3.4.2 *Quantidade de casos de teste e Asserts*

Quando um programador está a desenvolver novo código é esperado que ele, ou um colega de equipa, faça acompanhar esse código com testes que validem essas funcionalidades. Testes esses que ajudarão a validar a build em que sairão estas funcionalidades e posteriormente nas regressões para validar as alterações efetuadas. Deste modo é esperado que os casos de teste e os respetivos *asserts* aumentem ao longo do tempo. Esta quantidade pode ser influência por outro tipo de métricas como a cobertura de código que influência o número de casos de teste desenvolvidos consoante a cobertura das funcionalidades desenvolvidas [20].

Pode ser calculado segundo a seguinte formula [20]:

$$\frac{\mathrm{TCT_i}}{\mathrm{TLOC_i}} \text{ e } \frac{\mathrm{TA_i}}{\mathrm{TLOC_i}} \qquad (2)$$

Onde:
TCT_i = número total de casos de teste na iteração i
TA_i = número total de *asserts* na iteração i
$TLOC_i$ = número total de linhas de código aplicacional na iteração i

2.1.3.4.3 Cobertura dos requisitos

Os requisitos devem ser claramente definidos numa fase inicial do desenvolvimento de *software*. Podem ser vistos como objetivos e as equipas devem focar o seu trabalho para os desenvolver. A qualidade de um componente de *software* pode muitas vezes ser definida pela sua capacidade de satisfazer os requisitos definidos. Devido à importância da clara identificação dos requisitos a implementar, é muito importante cobrir todos os requisitos com uma clara cobertura de testes de modo a garantir a qualidade da sua implementação. Para além disto, uma correta definição de um requisito ajuda posteriormente a tarefa de um *tester* ao definir o plano de testes a aplicar a cada um dos requisitos, desta forma é possível rapidamente analisar a cobertura dos requisitos e perceber se não existem testes perdidos que deveriam ser integrados nas validações dos requisitos.

$$\text{Requirements Coverage} = \left(\frac{\text{Number of requirements covered}}{\text{Total number of requirements}}\right) * 100 \qquad (3)$$

Como o próprio nome indica cobertura de requisitos apresenta a percentagem de requisitos que são cobertos por testes. Apesar da sua utilidade e importância na validação de *software*, esta percentagem por si só pode levantar dúvidas na totalidade na validação se não for bem aplicada. Num cenário ideal deveriam ser considerados como número de requisitos cobertos os requisitos que forem cobertos na sua totalidade por testes. No entanto, poderão existir requisitos mais complexos em que não é possível, devido ao custo associado ou limitações de tempo, cobrir as suas potencialidades a 100%, desta forma esta métrica pode tornar-se enganosa. Apesar disto, é um indicador importante para balancear o trabalho das equipas na especificação e realização de testes tendo em conta os requisitos cobertos ou não.

2.1.3.4.4 Distribuição de defeitos por áreas funcionais

No desenvolvimento de *software* é normal que existam requisitos com um grau de implementação tecnicamente mais exigente ou que pode levar a que existam mais falhas nos mesmos. É importante identificar essas áreas no *software* desenvolvido para que exista uma maior atenção nas mesmas. É esperado que com o passar do tempo, a taxa de defeitos encontrados seja inferior, no entanto, com o indicie de distribuição de defeitos pelo código desenvolvido é importante porque ajuda a identificar as áreas mais sensíveis do *software* e que devem ter uma especial atenção.

Normalmente, quando é identificada uma área com um nível de defeitos superior às restantes pode representar preocupações mais elevadas sobre a funcionalidade global do produto. Com a identificação destas áreas dá tempo às equipas responsáveis para que se preparem com a análise pertinente de como agir em caso de falha destas áreas.

$$\text{Defect Distribution} = \left(\frac{\text{Total number of defects}}{\text{Functional area(s)}}\right) * \text{Status} * \text{Phase} \qquad (4)$$

2.1.3.4.5 Quantidade de defeitos encontrados

Os testes não são executados para demonstrar que uma aplicação está livre de defeitos, mas sim para encontrar o máximo de defeitos possível. Encontrar defeitos o mais cedo possível é positivo porque para além de o *software* ser entregue com o mínimo de falhas possível torna o seu desenvolvimento mais sustentável porque a sua manutenção será menor. Um defeito ao ser encontrado deve ser registado bem como a sua causa e a sua resolução.

Numa fase imatura da equipa de desenvolvimento o número de defeitos encontrados poderá ser elevado, no entanto, tem a tendência de diminuir ao nível que a maturidade vai aumentando. Apesar disto, a complexidade das funcionalidades poderá também influenciar este número de forma negativa ou o tempo gasto pelas equipas em testes exploratórios [20].

Não existe uma regra concreta para a utilização desta métrica, deverá ser utilizada sempre que a equipa ou um supervisor desejar medir a quantidade de defeitos encontrados num *software* antes de o encontrar. No entanto, esta métrica pode ser facilmente inflacionada caso algum elemento da equipa encontre o defeito e o corrija sem o registar, desta forma a métrica poderá apresentar alguma margem de erro que pode ser considerada.

2.1.3.4.6 Percentagem de defeitos abertos e fechados

É possível caracterizar os defeitos em diferente status, Abertos e Fechados. Existem fases de desenvolvimento em que o número de defeitos é maior do que o normal o que pode provocar que sejam difíceis de controlar provocando a entrega de *software* com defeitos. Ao comparar a frequência de defeitos abertos com indicies de resolução dos mesmos dão indicadores sobre a capacidade dos *testers* trabalharem juntos dos *developers* para identificar e resolver problemas de *software* [19].

$$\text{Defect Open and Close Rate} = \frac{\text{Defects found before delivery}}{(\text{Defects found before delivery} + \text{Defects found after delivery})} * 100 \quad (5)$$

Desta forma é possível analisar de que forma a coordenação de trabalho nas equipas não está a afetar a deteção e remoção de defeitos que possam atrasar a entrega e prejudicar a qualidade do *software* alertando para possíveis problemas que podem ser resolvidos atempadamente para desta forma prevenir problemas maiores.

2.1.3.4.7 Mean Time to Detect e Mean Time to Repair

Em determinadas alturas do desenvolvimento de *software*, especialmente no final da validação e entrega do produto, efetuam-se análises para perceber de que forma é possível melhorar o processo e que ações provocaram os tempos de aprovação. Existem métricas que ajudam a perceber alguns fatores que normalmente podem atrasar ou acelerar uma validação de um produto. O MTTD e MTTR dão a possibilidade de analisar a eficácia de uma equipa de desenvolvimento em determinadas situações.

$$\text{MTTD} = \frac{\text{Number of issues detected}}{\text{Total execution time}} \quad (6)$$

$$\text{MTTR} = \frac{\text{Number of issues fixed}}{\text{Total coding time}} \quad (7)$$

O MTTD permite perceber o tempo que foi demorado até que um defeito foi encontrado, enquanto que o MTTR se refere ao tempo que foi demorado a corrigir este mesmo defeito. Estes indicadores são muito importantes porque caso de apresentem elevados poderá estar em evidencia uma lacuna na equipa de desenvolvimento quer seja em competências técnicas como em défice de mão de obra.

2.1.3.4.8 Eficiência na resolução de defeitos

As métricas relacionadas com defeitos acrescentam sempre valor à análise efetuada sobre a eficácia de uma determinada equipa relacionada com a validação de software. A eficiência da resolução de defeitos apresenta a taxa na qual os elementos das equipas foram capazes de tratar e corrigir as falhas identificadas no produto. Um controlador do trabalho desempenhado pode utilizar esta métrica para perceber se o esforço aplicado na deteção e resolução de defeitos está a ser suficiente ou se será preciso agir em conformidade para melhorar este indicador.

$$\text{Defect Removal Efficiency} = \left(\frac{\text{Number of Pre Release Defects}}{\text{Number of Total Defects}}\right) * 100 \qquad (8)$$

2.1.3.4.9 Eficácia dos casos de teste

Só o facto de serem executados vários casos de teste não garante que o *software* está a ser corretamente testado. As especificações dos casos de teste devem ser claramente definidas em conformidade com os requisitos caso contrário existem fatores que irão contribuir negativamente para a eficácia de um caso de teste. Normalmente o que provoca a ineficácia de um caso de teste é a conceção de casos de teste com especificações funcionais incompletas, *design* de teste deficiente e interpretação errada das especificações de teste por *testers* [21].

$$\text{Test Case Effectiveness} = \left(\frac{\text{Faults found by test cases}}{\text{Total number of faults reported}}\right) * 100 \qquad (9)$$

Normalmente, é definido um valor que se considera adequado para que um caso de teste seja considerado eficaz. Caso um caso de teste esteja abaixo desse valor é uma indicação clara que se devem tomar ações para melhorar o modo como o caso de teste está a ser abordado. Uma abordagem que pode ser tida em conta é a análise casual de falhas detetadas pelos clientes em produção que não foram detetados no processo de *Continuous Delivery & Integration* [21].

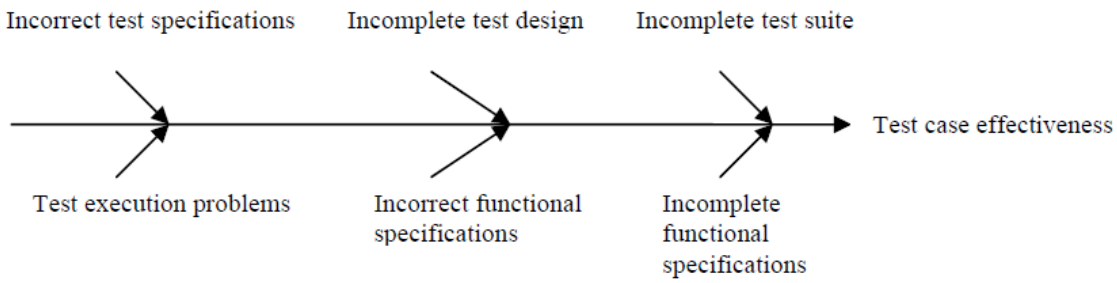


Figura 4 – Fatores que afetam a eficiência dos casos de teste [21]

Independentemente das abordagens sugeridas é necessário perceber quais os fatores que têm influência direta na eficácia de um caso de teste tais como especificação incorreta do teste que provoca uma execução de testes com problemas, uma incorreta especificação funcional que irá provocar problemas no design do teste e por fim uma especificação funcional incompleta irá provocar diretamente uma *test suite* incompleta, visíveis na Figura 4.

Posto isto, algumas boas práticas a ter em conta para derivar bons casos de teste de requisitos são [21]:

- Os detalhes e as complexidades da aplicação devem ser compreendidos e analisados;
- O comportamento do sistema, *workflow* e dependências devem ser entendidos;
- Deve ser analisa em que medida uma alteração numa parte da aplicação tem impacto nas outras partes da mesma;
- Os procedimentos de teste não devem ser concebidos apenas a partir de especificações de requisitos;
- Os testes não se devem sobrepor levando a que dois *testers* testem a mesma funcionalidade. Os testes devem ser escritos para serem executados de modo a serem reutilizados e combinados;
- O resultado esperado de um caso de teste não deve invalidar os resultados de um outro caso de teste;
- As condições prévias necessárias à execução de um teste devem ser determinadas analisando o fluxo do teste;
- Os casos de teste com maior prioridade devem ser realizados numa fase mais precoce da validação do *software*;
- Os casos de teste identificados devem conter uma prioridade com base nos padrões de maior risco da funcionalidade.

2.1.3.4.10 Tempo de execução dos casos de teste

Para um processo de *Continuous Delivery* esta métrica é muito importante para medir o tempo de execução de testes ao longo de uma *release*. É uma métrica simples que insere muito valor na análise aos testes efetuados. A tendência desta métrica é que aumente ao nível que as funcionalidades e a sua complexidade aumentam. No entanto, as equipas de desenvolvimento devem sempre ter a tendência de manter este valor o mais curto possível para que as *builds* sejam aprovadas rapidamente.

2.1.3.4.11 Cobertura de código

A cobertura de código é uma métrica utilizada em métodos de teste *White-Box*, ou seja, é necessário o acesso ao código para perceber a percentagem de código que está efetivamente

coberta pelos testes efetuados. *White-box testing* é baseado numa estrutura de *software* ou sistema bem definida que pode ser vista com o seguinte [22]:

- A estrutura de um componente de *software* tais como decisões, *branches* e caminhos distintos;
- A estrutura de integração do sistema deve contar um arvore de chamadas para documentar todas as integrações entre módulos;
- A estrutura do sistema deve conter um menu estruturado, um processo de negócio ou uma página *web* estruturada.

Está métrica é uma das mais importantes nas fases de teste pois revela indicadores da abrangência dos testes e de que forma as equipas se podem organizar para aumentar a percentagem de cobertura da sua aplicação através de testes.

Cobertura de código pode ser dividida em sub-métricas que ajudam a identificar diferentes abordagens de cobertura tais como [22]:

2.1.3.4.12 Statement coverage

Statement coverage é a métrica mais simples de entender e prática de analisar a cobertura do código da aplicação. Um pedaço de código ou instrução será coberta pelos casos de teste se for executada seja de forma correta ou não. Esta métrica é tipicamente utilizada para perceber de que forma os testes executados estão a cobrir as funcionalidades de um produto aumentando assim o grau de confiança nos testes realizados. No entanto, este tipo de cobertura por si só pode ser enganoso tendo em conta que mesmo com 100% de *statement coverage* poderão surgir problemas na aplicação. O ideal será combinar esta métrica com as *Decision e Branch coverage* e aí sim a sua confiança aumenta consideravelmente.

$$\text{Statement coverage} = \left(\frac{\text{Executable statements covered}}{\text{All Executable statements}}\right) * 100 \qquad (10)$$

Apesar de ser uma métrica simples de obter, poderá ser difícil de alcançar os 100% de *statement coverage* devido ao facto de existir código nas aplicações que normalmente são executados em casos de erro crítico e que não devem ocorrer. Nesse caso, deverá ser percebida a percentagem desse código para que seja possível prever uma margem de erro e caso o *statement coverage* não seja 100% será possível perceber o porquê.

2.1.3.4.13 Decision coverage

A cobertura de decisão obtida pela cobertura dos casos de teste de condições possíveis no código da aplicação, por exemplo, as opções de *True* e *False* numa condição *if*. A técnica de testes às decisões deriva de testes que executam determinadas condições especificas de que são originadas por pontos de decisão no condigo que transferem o fluxo do código para outra secção [22].

$$\text{Decision coverage} = \left(\frac{\text{Decision outcomes covered}}{\text{All possible decision outcomes}}\right) * 100 \qquad (11)$$

Com isto, é importante perceber a importância de combinar esta métrica com *Statement Coverage* devido ao facto de que se for conseguido 100% de *Decision Coverage* é garantido 100% de *Statement Coverage* nessa condição. O inverso não é garantido.

2.1.3.4.14 Path coverage

Como o *Decision coverage* é correspondente e está diretamente ligado com uma condição atómica e isolada de todo o fluxo do restante programa, o *Path coverage*, garante que todos os caminhos possíveis são testados na aplicação. Em qualquer programa de dimensões consideráveis terá um enorme número de caminhos que deverão ser testados e isso não é fácil de atingir. O exemplo seguinte ajuda a entender a logica de *Decision e Path coverage* e de que forma se complementam.

```
If($a){
       Print("$a é verdade!");
}
If($b){
       Print("$b é verdade!");
}
```

Neste exemplo, é possível obter um *Decision coverage* a 100% com ($a, $b) a serem inseridos com o valor *(True, True)*, porque ambas as condições se vão verificar e irá ser possível obter a percentagem máxima de decisão. No entanto, se for efetuado um teste com *(True, False*) não se irá obter um *Path coverage* de 100%. Neste exemplo, existem quatro caminhos possíveis que todos deverão ser testados. Irá ser obtido 100% de *Path coverage* e 100% de *Decision coverage*, apesar de uma condição bastar para obter 100% de *Decision Coverage*.

2.2 Abordagens em empresas reputadas

Nesta secção iram ser descritas as abordagens de outras empresas em relação à forma como testam e monitorizam os seus testes das suas aplicações com as práticas de automação e CD. Com isto, será possível perceber as vantagens e desvantagens de cada abordagem e de que forma é possível adotar algumas abordagens, adaptando-as ao contexto da Farfetch.

2.2.1 *Microsoft*

A Microsoft é uma empresa muito conceituada na área de desenvolvimento de *software* com provas dadas neste ramo. Apesar de existir uma vertente óbvia de desenvolvimento, a equipa com cerca de 9000 *testers* e a forma como validam o código e o controlam é também um ponto que tem a atenção da empresa fazendo com que esta tente implementar um processo capaz de responder às suas necessidades e de manter um indicie de qualidade de *software* o mais elevado possível.

Num processo de desenvolvimento os testes são algo importante e fundamental para que seja possível melhorar e aumentar o valor entregue aos clientes. A Microsoft tem isso claro para os seus colaboradores e procura utilizar boas práticas de *Design* de *software* e testes. Para a Microsoft um *software* bem desenhado antecipa muitos problemas que possam acontecer, e o *design* é um passo essencial para criar *software* que funcione bem para o cliente [23].

Como o tempo de validação de *software* é importante, a Microsoft procura através desta métrica, apesar de ser um ponto difícil de implementar, estimar o tempo de duração de um teste. Só um período alargado de estudo e planeamento poderá dar uma resposta correta a esta questão. Efetuar uma correta estimativa da duração da fase de testes é tão importante como o ato de escrever *features* de *software*. Desta forma, é possível balancear o esforço das equipas na sua implementação.

A Microsoft possui fatores que ajudam na estimativa da duração dos testes conforme indicado na Tabela 1, em que é possível entender cada um desses fatores.

Tabela 1 – Fatores na estimativa de tempos de teste da Microsoft [23]

Atributo	**Como considerar**
Histórico de aplicações	É possível estimar um design de testes com base no histórico de projetos anteriores.
Complexidade	Complexidade está diretamente com testabilidade. Algumas aplicações podem ser testadas mais rapidamente do que algumas mais complexas.
Objetivos de negócio	A aplicação é um protótipo ou uma demonstração? Ou é um *software* responsável pelo controlo de voo de uma nave espacial? Os objetivos de negócio influenciam a amplitude e profundidade do esforço de teste.
Conformidade	Se a aplicação tem de estar em conformidade com uma regra, estes requisitos devem ser considerados quando é estimado o tempo da tarefa.

Na Microsoft existe uma elevada preocupação com a automação dos processos e um claro investimento na vertente de *design* de *software*. Isto engloba o *design* de testes e aqui existe a preocupação de validar o software de uma forma bastante completa e assertiva. Existem uma serie de passos com o objetivo de testar uma parte diferente da aplicação, que em conjunto e com o tempo garantem uma validação completa do *software* desenvolvido. Para além do desenvolvimento de testes efetuam a monitorização e análise de resultados com a recolha de métricas dentro do contexto da aplicação.

Os *items* seguintes são aqueles que normalmente são discutidos numa especificação e *design* de testes [23]:

- Visão geral / objetivos;
- Estratégia;
- Testes funcionais;
- Testes de componente;
- Testes de integração / sistema;
- Testes de interoperabilidade;
- Testes de conformidade;
- Internacionalização e globalização;
- Testes de performance;
- Testes de segurança;
- Testes de configuração;
- Dependências;
- Métricas.

2.2.2 *Google*

A empresa Google também tem investido nas *releases* rápidas e frequentes com uma camada considerável de automação de testes. Apesar de terem a consciência de que uma *release* com qualidade é muito importante a Google têm o foco na rapidez das suas *releases*. Devido à enorme quantidade de tarefas criticas e minuciosas que tinham constantemente de encarar como migrações e instalações de novas bases de dados a Google implementou uma cultura na empresa com o lema *"Automate Yourself Out of a Job: Automate ALL the Things!"* [24].

2.2.2.1 *Continuous Integration*

A Google acredita que um sistema automático pode ser visto como uma plataforma que é facilmente estendida e aplicado mais sistemas. A automação não fornece apenas consistência [24]. Como normalmente um sistema informático requer de uma manutenção muito minuciosa, as tarefas humanas realizadas são de extrema importância e facilmente podem conter falhas e por vezes levar um tempo excessivo. Uma plataforma pode substituir essas tarefas realizando-as de forma mais frequente, mais rapidamente e com menos falhas. É possível ainda extrair mais facilmente métricas como performance.

No dia a dia e com o conceito de automação e plataforma bem introduzidos na realidade da empresa existem fatores que fazem com que a equipa continue a investir nestas abordagens que são:

- Reparações rápidas: Uma das métricas que esta empresa procura manter e melhorar de forma contínua é o MTTR. Com um sistema automático a ser executado regularmente e corretamente resulta na redução desta métrica. Desta forma é

possível focar a atenção em novas tarefas enquanto que o sistema de avalia de forma autónoma e fornece *feedback* continuo das alterações efetuadas.

- Ações rápidas: Google tem muitas tarefas automatizadas porque acredita que se não fosse o caso muitos dos seus serviços não seriam capazes de suportar [24]. A automação permite encadear uma série de tarefas de forma automática e desta forma permite suportar tempos limite de interações essenciais ao funcionamento dos serviços. A reação de uma máquina é normalmente mais rápida do que a reação de um humano e desta forma permite que haja uma intervenção cirúrgica e mais rápida do que se for um humano a realizar.
- Poupar tempo: os dois pontos anteriores acabam por culminar no objetivo da empresa de efetuar as suas tarefas rapidamente e desta forma poupar tempo e produzir mais e melhor. Para além da produção de mais *software,* os lideres da empresa acreditam que a automação melhora o ambiente de trabalho da empresa. "Se estamos a construir processo e soluções que não são automatizáveis, continuamos a ter pessoal dedicado a manter o sistema. Se tivermos o pessoal para fazer o trabalho estamos a alimentar as máquinas com suor e lagrimas de seres humanos. Basta pensar no *The Matrix* mas com menos efeitos especiais e mais Administradores de Sistema chateados.", Referido por Joseph Bironas, um SRE que liderou o Google Datacenter, representa a importância da automação para a empresa.

2.2.2.2 *Métricas*

A Google apesar de acreditar que existem sempre problemas inesperados nos sistemas que as melhores métricas não podem prever procura monitorizar os seus serviços mantendo num repositório central da empresa todas as métricas efetuadas a código, testes e *release*.

Posto isto, as métricas mais utilizadas a sua fase de testes são as seguintes:

- **MTTR** [24]: ajuda a empresa a avaliar o tempo demorado pelas suas equipas a reparar defeitos e a perceber de que forma um sistema automático contribui para a sua otimização. Acreditam que um sistema automático consegue rapidamente validar a correção inserida melhorando assim o MTTR.
- **TTL** [24]: esta métrica permite avaliar e analisar ao longo do tempo a duração total das *releases* realizadas na empresa. Desta forma é possível analisar possíveis problemas de TTL demasiado altos.

- **Número de casos de teste** [25]: utilizam esta métrica para perceber o investimento de existe na equipa de testes. Percebem o foco nos testes e na automação ao longo do tempo
- **Número de casos de teste executados** [25]: conseguem desta forma perceber quais os casos de teste executados e tentam perceber que razões levaram a que um certo número, se for o caso não seja executado. Isto é importante porque permite perceber a existência de material *legacy* na empresa que não é utilizado e causa entupia.
- **Número total de defeitos encontrados numa execução de testes** [25]: dependendo da maturidade da aplicação esta métrica permite analisar o estado de maturidade do software antes de ser instalado em produção.

2.2.2.3 *Tipos de teste*

A Google tem uma abordagem própria dos seus tipos de teste a executar. Contorna um pouco a normas tradicionais e renomeou os tipos de teste para que sejam mais claros dentro da comunidade de desenvolvimento da empresa. Desta forma o dialogo é mais claro e rapidamente de identificam que tipos de teste estão a ser abordados. Estes testes são então identificados da seguinte forma:

- *Small tests* [25]: equivalentes aos testes unitários testam uma unidade singular no código.
- *Medium tests* [25]: equivalentes aos testes de componente, testam a interação dos vários módulos de um componente de forma isolada num ambiente controlado ou num ambiente real.
- *Large tests* [25]: equivalentes aos testes de sistema efetuam testes com interação dos diferentes componentes da aplicação, tipicamente *end to end* num ambiente real equivalente ao de produção.

Apesar da convenção dos nomes atribuídos aos testes ser demasiado própria a sua utilização é bastante próxima ao comum e verificado na comunidade de *testers* existentes nas restantes empresas.

2.2.3 *Firefox*

A empresa Firefox possui uma metodologia de trabalho bastante própria e diferente das restantes. Não possuem um investimento muito grande numa equipa de testes própria, mas recorrem à comunidade dos seus utilizadores para que estes efetuem os testes para a empresa. O número estimado de contribuidores para efetuar os vários testes das versões são respetivamente 100,000 para *builds* noturnas, 1 milhão para versões *alpha*, 10 milhões para

versões *beta* e mais de 100 milhões para as versões *major* do *browser* Firefox [26]. À exceção no componente *Mozmill* que possui uma infraestrutura de testes automáticos, os restantes testes efetuados são maioritariamente manuais.

Para controlar todas estas regressões da comunidade, a empresa possui uma infraestrutura de *system-level regression testing* que consiste numa base de dados com o objetivo de armazenar e documentar os vários casos de teste funcionais com as respetivas execuções e resultados. Desta forma a comunidade consegue identificar um caso de teste não executado, executa-lo e efetuar a inserção do resultado e respetiva análise.

Devido à sua forma de atuar pouco comum o controlo destes testes efetuados necessita de um controlo mais expressivo por parte dos responsáveis pela qualidade do *software* desenvolvido. As métricas nesta empresa são algo muito importante e são das poucas formas existentes de monitorização e avaliação do estado de qualidade de uma versão.

Para tal a empresa utiliza as seguintes métricas [26]:

- Número de testes executados por dia;
- Número de testes únicos executados por dia;
- Número de *testers* por dia;
- Número de *builds* testadas por dia;
- Número de *commits* por dia;
- Número de idiomas testados por dia;
- Número de sistemas operativos testados por dia;

Depois de recolhidos os dados, a empresa aplica métodos estatísticos que têm o objetivo de avaliar a qualidade do *software* desenvolvido e a evolução do trabalho efetuado na empresa. É utilizado a análise estatística *R* para efetuar todos os cálculos. O método estatístico *Shapiro-Wilk* mostra que a informação analisada não possui uma distribuição distribuída normalmente [27]. São utilizadas análises estatísticas não paramétricas para comparação de alguns valores. Comparações de dois grupos distintos a empresa utiliza testes de *Wilcoxon rank-sum* [28] e para estudar o tamanho do efeito é utilizado *Cliff's Delta* [29].

2.2.4 *Spotify*

A Spotify é possivelmente a empresa a nível de organização de pessoas e coordenação de trabalho mais parecido com a Farfetch. Nesta empresa as equipas são encorajadas a aplicar princípios de *Lean Startup* tais como *MVP (Minimun Viable Product)* e aprendizagem validada. *MVP* significa efetuar *releases* o mais cedo e frequentes possível com a respetiva validação através de métricas e testes A/B para descobrir o que realmente funciona ou não funciona.

Devido ao objetivo de se tornarem mais rápidos nas *releases* de *software* com ciclos mais curtos e com maior facilidade de efetuar testes nestes ciclos, a empresa efetuou um forte investimento em automação de vários processos e testes, no entanto, a automação por si só é um desafio e uma tarefa difícil. Para os testes foi implementado o conceito de *Model-based testing* que permite separar o *design* dos testes da sua implementação atribuindo responsabilidades diferentes a cada um deles em que:

- *Models* são a camada de abstração na automação de testes;
- *Testers* usam os *models* para desenhar casos de teste;
- *Developers* implementam o código de testes.

Desta forma, conseguem investir e coordenar o desenho, codificação e execução de diferentes tipos de teste, tais como:

- Testes unitários*;*
- *Testes de Componente;*
- *Testes de Integração;*
- *Testes de Sistema;*
- *Testes à API;*
- *GUI tests;*
- *Testes de Performance;*

Desta forma conseguem dividir responsabilidades e com os vários tipos de testes efetuam uma validação do *software* de forma progressiva validando as várias áreas da aplicação sempre com o intuito de detetar os problemas o mais rapidamente possível para que o desenvolvimento seja mais rápido e com menos erros.

Outro método adotado pela empresa para aumentar o número de *releases* e desta forma tornar o processo de *Continuous Delivery* mais rápido e organizado, foi a criação de Tribos. Uma Tribo é considerada uma pequena startup dentro da empresa. É um conjunto de equipas autosustentaveis que possuem todas as competencias para efetuar o desenvolvimento de software e respetiva validação e instalação em produção.

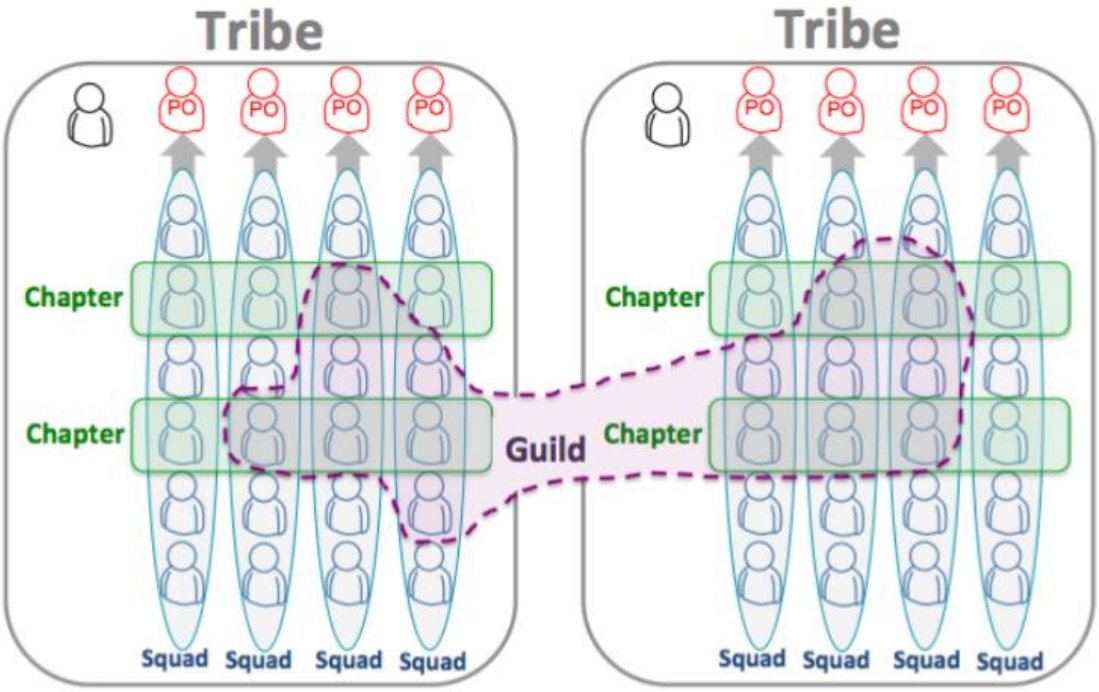


Figura 5 – Divisão de equipas na empresa spotify.
Fonte: https://ucvox.files.wordpress.com/2012/11/113617905-scaling-agile-spotify-11.pdf

Na Figura 5, está representado a forma como a empresa se organiza e divide os diferentes colaboradores. Cada Tribo é composta por diferentes equipas *SCRUM* que são a unidade básica de desenvolvimento de software. Cada equipa trabalha junta fisicamente e possuem todas as ferramentas necessárias para desenhar os requisitos, desenvolver, testar e enviar para produção cada uma das *releases*.

O propósito da criação de cada equipa é distribuir as várias áreas funcionais do *site* pelas equipas e desta forma cada equipa torna-se independente e focada numa área especifica o que é possível que exista uma especialização em cada uma das áreas para que desta forma se evolua de forma uniforme em toda a plataforma aplicando as tecnologias e metodologias mais atuais e inovadoras aumentando assim a qualidade da plataforma.

No entanto, existem sempre dependências entre áreas funcionais que criam dependências entre equipas. Isto não é necessariamente mau, as equipas trabalham juntas para desenvolver algo inovador e de qualidade. Apesar do objetivo passar por tornar as equipas as mais autónomas possíveis e tentar reduzir ao máximo as dependências existentes [30].

Apesar do objetivo ser a independência e a autonomia, a empresa tem consciência de que isso acarreta problemas que podem ter um custo elevado como por exemplo, o *tester* da equipa A estar a tentar resolver um problema há semanas que o *tester* da equipa B já resolveu. Para colmatar este problema e incentivar à partilha de experiencias entre equipas são criadas várias comunidades de trabalho. Comunidades essas que têm um responsável por dinamizar o grupo em que consiste incentivar a partilha de experiencias e problemas.

2.3 Portal Slice

A aplicação escolhida tem como objetivo iniciar o seu desenvolvimento seguindo as boas normas da programação. Isto leva a que existam *releases* automáticas e contínuas, com métricas dos testes desenvolvidos, com entregas de *software* rápidas e frequentes e com uma validação gradual com vários tipos de testes garantindo assim que as falhas são encontradas o mais depressa possível. Desta forma é possível que sejam detetadas rapidamente para que possam ser corrigidas atempadamente e desta forma prevenir entregas de *software* com um número de falhas elevado.

O Portal *Slice* consiste num novo componente *web* que irá substituir uma secção do Portal, que representa a principal aplicação da *Farfetch*. Com o inicio do desenvolvimento em março de 2017, este projeto foi desenvolvido por uma equipa de 9 elementos entre eles *developers, testers* e *UI developers*. Devido à complexidade do Portal, em que possui enumeras dependências o que provoca uma gestão da aplicação difícil e com *releases* demoradas, os Portal *Slices* têm o objetivo de dividir o Portal em vários pequenos módulos. Cada modulo, com complexidade mais inferior, tem os fundamentos de um micro-serviço [31] em que deverá conseguir ser gerido de forma independente. Desta forma, com a divisão do Portal em vários pequenos módulos, cada modulo terá testes específicos o que fará com que cada modulo seja menos suscetível a falhas aumentando assim a sua qualidade.

Este novo módulo introduziu ainda métricas na vertente de testes em *Continuous Delivery*. O objetivo é através destes indicadores, perceber quais os fatores que contribuíram para que uma nova *release* demorasse mais ou menos tempo e porquê. Este é um plano útil para balancear o esforço do trabalho praticado pelas equipas que irá permitir alcançar novas metas.

2.4 Ferramentas e tecnologias

Nesta secção serão descritas as tecnologias e ferramentas utilizadas para o desenvolvimento e execução de testes ao Portal *Slice.*

2.4.1 *React*

O *React* é uma libraria Javascript para construir interfaces de utilizador. O *React* com a sua abordagem declarativa torna a criação de *UIs* interativas menos custosa. Desenha na aplicação vistas simples para cada estado e o *React* vai de forma eficiente atualizar a renderização apenas dos componentes que contenham alterações. Para além disto, vistas declarativas tornam o código desenvolvido mais previsível e fácil de analisar.

Utiliza uma base de componentes com a criação de componentes encapsulados que gerem o seu próprio estado. Visto que a lógica dos componentes está escrita em *Javascript* em vez de modelos, é possível facilmente passar dados complexos através do seu aplicativo e manter o estado da aplicação fora do DOM.

2.4.2 *Jest*

O *Jest* é usado por grandes empresas como o Facebook para testar código *JavaScript*, incluindo aplicações utilizando *React*. Por ser uma ferramenta que não requer configuração, tem a tendência de ser útil pois quando os engenheiros recebem ferramentas prontas a usar, acabam por escrever e desenvolver mais testes o que por sua vez resulta em bases de código mais estáveis e saudáveis [32].

Algumas particularidades do *Jest* é a sua configuração muito fácil, visto que a simples criação de um projeto com esta tecnologia é configurado automaticamente. Para além disto, tem um modo feedback instantâneo em que a ferramenta apenas executa os ficheiros de teste relativos a funcionalidades alteradas e está otimizado para dar resultados rapidamente.

O *Jest* possui ainda o *Snapshot testing* que representa a captura de arvores *React* ou outros valores serializáveis para simplificar o teste e para analisar como o estado muda ao longo do tempo. Possui paralelização através de vários processos para maximizar a sua performance e ainda a possibilidade de incluir relatórios de cobertura de código relativo a estes testes.

2.4.3 *Selenium*

Selenium é a ferramenta responsável pela automatização de *browsers.* Esta ferramenta é utilizada para automatizar uma aplicação *web* para fins de teste, no entanto, poderá ser utilizada para automatizar processos *web* repetitivos, como tarefas administrativas efetuadas em aplicações *web*.

Para a interação com o *browser* é utilizado o *Selenium WebDriver* e em conjunto com o *Selenium* permiter criar testes com as duas seguintes características:

- Testes automáticos robustos e baseados no *browser*;
- Dividir e distribuir scripts em vários ambientes.

O *Selenium* possui ainda a possibilidade de paralelização e automação de múltiplos *browsers* em simultâneo com a *SeleniumGrid* que permite desta forma diminuir os tempos de regressão desta ferramenta.

2.4.4 *Apache JMeter*

O *JMeter* é responsável por medir a performance de uma aplicação. A ferramenta foi projetada para testar aplicações web, no entanto, tem sido expandida para outro tipo de aplicações.

Esta ferramenta pode ser utilizada para testar o desempenho de recursos estáticos ou dinâmicos. Tem a possibilidade de simular uma carga elevada num servidor ou grupo de

servidores para que seja possível testar o seu funcionamento em stress e qual a sua capacidade e ainda analisar o seu comportamento sob diferentes tipos de carga.

Apesar de ser desenvolvido a pensar nas aplicações web, o *JMeter* não é um *browser*. Trabalha sim no nível de protocolo, ou seja, trabalha como um *browser* mas de forma limitada. O *JMeter* apenas mede o tempo de resposta de uma pagina e não executa todas as ações suportadas por um *browser* como a execução de *javascript* encontrado nas páginas. Também não faz as páginas HTML como um *browser*, para além de ser possível visualizar o HTML na resposta estas temporizações não estão incluídas em nenhuma amostra.

3 Análise de valor

A empresa com a mentalidade de acompanhar a evolução tecnológica, procura estar na vanguarda das tecnologias e processos informáticos que lhe consigam proporcionar as melhores soluções do mercado que correspondam diretamente às suas necessidades.

As decisões internas a nível tecnológico têm sempre por base o objetivo de mudança para algo que acrescente valor ao que já existe e é produzido na empresa. É com esse intuito que este trabalho será efetuado e que a solução atingida corresponda a um aumento na qualidade dos processos já existentes.

Para entregar valor à empresa este trabalho tem de conjugar para além de competência na sua função uma vertente inovadora e criativa capaz de marcar a diferença para que seja mais fácil a sua inclusão e aceitação nos atuais intervenientes da empresa.

Entende-se por competente uma solução inovadora que marque a diferença através da inserção de novas ideias que permitam melhorar a forma como introduz os tipos e os níveis de teste no seu processo de *Continuous Delivery*. Introdução essa que permita ganhar confiança de uma *release* ao longo do tempo com o intuito de ganhar confiança na mesma e o facto de detetar as falhas o mais cedo possível.

A solução além de criativa e inovadora deverá ser responsiva o que fará que esteja preparada para dar resposta aos vários pedidos que irá encarar. Flexível para o facto de a mesma solução poder ser reutilizada para as diferentes realidades de todas as equipas tecnológicas da empresa. Deverá ainda ser evoluída tecnologicamente de forma a fornecer as melhores soluções e abordagens disponíveis e que se adequam à sua realidade.

Para uma melhor perceção do modelo de negócio envolvente na solução a ser apresentada, o seguinte modelo *canvas,* apresentado na Figura 6, procura descrever os principais fatores desse modelo.

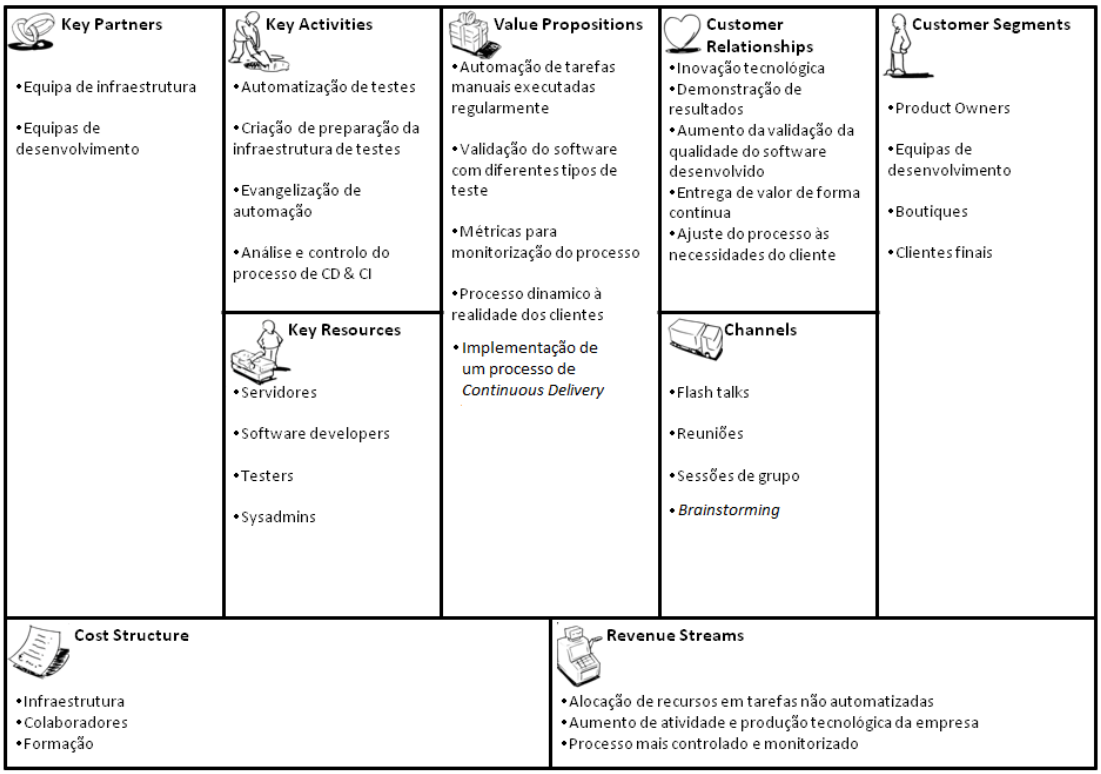


Figura 6 – Modelo canvas

No contexto empresarial tecnológico, as tecnologias utilizadas representam um fator diferenciador daquilo que representa a empresa no mundo da moda, mercado de negócio da empresa, a evolução tecnológica deve acompanhar as constantes e repentinas mudanças do setor com critério e competência. Para que tal seja possível a empresa deverá estar preparada com metodologias modernas e inovadoras que consigam garantir que rapidamente é validado e entregue para produção um componente desejado pelo cliente.

A empresa oferece aos seus clientes uma plataforma *e-commerce* desenvolvida com as tecnologias mais recentes do mercado e tem em preocupação de garantir qualidade do *software* entregue com processos estudados e implementados para que rapidamente estas alterações estejam disponíveis. Os principais interessados por estas abordagens são os nossos *Product Owners* e restantes *Stakeholders* como as *boutiques* que são as principais interessadas em ver as suas intenções reproduzidas em funcionalidades capazes de marcar a diferença junto da concorrência e desta forma aumentar a taxa de retenção de clientes.

Conhecendo estas expectativas dos clientes, a empresa procura desenvolver o melhor processo utilizando tecnologias inovadoras para que desta forma os clientes se mantenham satisfeitos na entrega atempada de novos desenvolvimentos de *software*. Desta forma é possível que exista em a empresa e os seus *Stakeholders* uma cumplicidade que respeitada e assumida por ambas as partes permite à empresa desenvolver as espectativas dos *Stakeholders* utilizando os meios essenciais permitindo assim aos *Stakeholders* aumentar o

leque de clientes e desta forma aumentar as suas vendas que será o principal objetivo dos mesmos.

Para além das soluções de baixos custos implementadas pela empresa e da qualidade da solução, estes processos devem ser flexíveis o suficiente de forma a se adaptarem as necessidades de cada um dos *Stakeholders* para que com um único processo, se consiga colmatar as necessidades de vários clientes. Desta forma, com um processo flexível as mudanças de negócio não são tão problemáticas e a empresa rapidamente adapta os seus produtos às particularidades que vão sendo alteradas.

A competência técnica é também uma mais valia para os clientes. A empresa procura de várias formas continuar o seu investimento nas pessoas garantindo sempre a formação adequada dos seus programadores. A preocupação da evolução tecnológica dá maiores garantias de sucesso no futuro, ou uma maior facilidade ao lidar com a mudança e isto fortalece a confiança que existe entre a empresa e os seus clientes.

3.1 The new concept model (NCD)

3.1.1 *Identificação da oportunidade*

Ao longo do tempo, com o crescimento aplicacional da empresa, começaram a surgir evidências de carência de processos capazes de dar resposta às espectativas de automação quer por parte dos responsáveis da empresa quer por parte dos *Product Owners* e restantes *Stakeholders* associados à empresa. Desde que se definia um novo requisito a ser desenvolvido, até que este fosse entregue para produção, seria aplicado um processo excessivamente manual e suscetível a erros que atrasava muito o desenrolar da entrega de valor ao cliente.

Com o evoluir da vertente tecnológica do mercado e em especial da área de automação de processos nos diversos setores, a empresa identificou com a automação de processos um fator diferenciador da concorrência e como sendo uma realidade capaz de agilizar o processo delegando uma maior responsabilidade para os computadores o que reduz o número de erros manuais e desta forma obter um processo mais eficiente e eficaz e com um custo inferior.

Devido ao elevado número de tarefas desempenhadas manualmente exatamente da mesma forma, por uma ou mais pessoas, ao longo do tempo, foram identificados padrões que seriam possíveis automatizar. Isto, permitiria alocar as pessoas noutras tarefas o que iria proporcionar um crescimento maior do ramo tecnológico da empresa tendo em conta que as tarefas que até ao momento seriam desempenhadas por pessoas seriam responsabilidades das máquinas que para além de as executar de forma mais rápida, faziam-no de forma continuada e numa frequência mais elevada o que aumentaria a produção.

3.1.2 *Análise da oportunidade*

Após a identificação da oportunidade de investir na vertente de automação da empresa seria importante perceber de que forma é que esta intenção seria encarada pelos intervenientes da empresa. Outra questão importante será perceber até que nível seria algo possível de implementar em todas as áreas de desenvolvimento de *software* da empresa com um processo único e aceite por todos apenas com algumas mudanças específicas do contexto de trabalho de cada uma das equipas.

Com o crescimento bastante acentuado da empresa a um nível elevado, o desenvolvimento e evolução tecnológica da empresa tem de acompanhar este crescimento de forma a dar resposta à necessidades e espectativas dos clientes. Com o aumento do número de clientes a cada dia, as exigências são cada vez mais e especificas em que apenas uma empresa estruturada e capaz de dar resposta rapidamente a esses pedidos consegue reter os clientes e desta forma aumentar a sua probabilidade de sucesso. Apesar disto, os clientes querem agilidade na disponibilização dos seus pedidos nos ambientes de produção de forma a poderem usufruir de uma melhor experiencia de utilização dos serviços da empresa.

Tendo em mente o futuro da empresa, uma empresa global com milhões de utilizadores, o número de colaboradores está em crescimento o que aumenta a probabilidade de falhas nas passagens de conhecimento de tarefas desempenhadas manualmente ou uma passagem mais lenta dessa informação. Com a automação de processos esta informação partilhada é cada vez menos e assim será possível reduzir a probabilidade de erro.

Com a mentalidade de automação percebida e encarada por todos, será então possível desenvolver e implementar mecanismos 100% automáticos de validação e entrega de *software* aos clientes que permitirá às equipas aumentar a sua produção de novas funcionalidades e uma correção mais atempada de problemas detetados quer na fase de desenvolvimento quer já numa fase posterior do desenvolvimento de *software* que são as falhas detetadas pelos utilizadores finais.

Sendo processos automáticos, terão de ser desenvolvidos e mantidos pelos responsáveis pelos mesmos o que será possível serem controlados e continuamente analisados para que possam ser melhorados quer a nível de performance como a nível de eficiência. Com o passar do tempo a tendência deverá ser o aproximar processos capazes de aumentar o valor da empresa e não de o atrasar de forma a que exista a necessidade de intervenção humana para colmatar as suas falhas.

Para existir uma harmonia na implementação e articulação dos vários mecanismos automáticos, estes devem ser específicos a cada tarefa, até aqui manual ou uma nova tarefa identificada, devem ser atómicos e reutilizáveis noutras vertentes diferentes, devem ser claras o suficiente de forma a permitir a um novo membro perceber o seu objetivo e que permita controlar a sua pertinência para o processo através de análise profunda ao histórico da sua utilização.

Comparando o estado da empresa com a concorrência, foram identificados casos de sucesso como é o caso da *Spotify* em que com automatização de processos de desenvolvimento conseguem rapidamente disponibilizar novas funcionalidades aos seus clientes e isso é um fator de diferenciação devido à boa taxa de retenções de clientes que é essencial para o crescimento da empresa.

3.1.3 *Geração de ideias*

A geração de ideias a aplicar numa solução partiu do diálogo e discussão das partes interessadas que em conformidade declararam intenções que gostariam de ver desenvolvidas e perceberam de que forma poderiam ser atingidas. A geração de ideias é um processo que existe até ao fim de um determinado desenvolvimento com ajustes constantes até se obter uma solução considerada ótima. Gerar ideias consiste em fazer o levantamento das mesmas que podem, entretanto, ser desenvolvidas, esquecidas, alteradas, enriquecidas ou repensadas. É algo que faz parte do desenvolvimento tecnológico.

Quando alguém na empresa acredita que possuiu uma ideia diferenciadora que é capaz de inserir algo novo no que já é produzido na empresa contribuindo de forma positiva para o seu rendimento, partilha com os restantes responsáveis de modo a perceber a viabilidade de se aplicar a mesma. O objetivo desta partilha passa por tentar enriquecer a mesma através de pessoas mais experientes que consigam dar o seu contributo positivo.

Apesar disto, é importante perceber que existe uma cultura organizacional para se testarem novas ideias com o intuito de perceber as vantagens da sua execução. A empresa dá muito valor a novas ideias e incentiva os seus colaboradores em contribuir com as mesmas com alguns incentivos aos mesmos. Uma ideia inovadora pode ser sinal de grande valor para o seu futuro. Um exemplo disso é a realização de eventos internos (*Hakaton*) com o objetivo de serem desenvolvidas e testadas novas ideias no âmbito do negócio praticado ou fora dele.

3.1.4 *Seleção de ideias*

Quando são identificadas várias ideias que podem ser desenvolvidas, o importante na seleção das pretendidas é o facto de que uma má decisão pode comprometer a estabilidade da empresa no futuro, seja esse impacto significativo ou não. No meio de todas as ideias geradas, terão de ser selecionadas as que têm maior garantia de aumento de valor à empresa. Estas decisões podem ser tomadas em grupo ou de forma individual comparando todas as ideias e optando por algumas. Entretanto, na empresa surgiram várias ideias entre elas desenvolver automatização de processos em que é credível que irá aumentar a produtividade de todas as equipas e na especificação de métricas que permitam monitorizar, controlar a avaliar o esforço das equipas em determinadas tarefas que podem passar despercebidas e desta forma é possível atuar de forma mais eficiente para as resolver. É sobre estas duas ideias que o trabalho se vai desenvolver.

3.1.5 *Definição do conceito*

O desenvolvimento de um processo capaz de automatizar e agilizar o processo de entrega de *software* com critério, qualidade e com o controlo do processo com métricas apropriadas para o efeito é um investimento da empresa a pensar no futuro e nos resultados a longo prazo. Este conceito é algo inovador no mercado que tem ganho força nos últimos tempos e é essencial investir neste conceito que ajudará à empresa entregar mais valor ao cliente, em maior qualidade e de forma mais recorrente.

Com a constante pressão dos *Stakeholders* de verem as suas intenções convertidas em funcionalidades, estas abordagens irão oferecer-lhes garantias de entrega de *software* continua e frequente com a possibilidade de verificar a qualidade do *software* desenvolvido com vários períodos de testes que irão garantir uma maior qualidade. Com isto, o conceito em questão representa uma oportunidade quer na aproximação dos clientes como a nível financeiro que permite a que a empresa consiga balancear o seu esforço de acordo com as espectativas.

4 Estado inicial

Nesta secção irão ser analisados as metodologias de trabalho na empresa, tecnologias e outros conceitos de *testing* capazes de definir o processo atual para que seja possível descrever uma hipótese de solução capaz de ser integrada no processo atual, mas que consiga colmatar as falhas descritas. Para o desenvolvimento de uma nova solução capaz de responder ao problema descrito, inicialmente irão ser analisados os problemas já ocorridos na empresa e tentar aplicar as soluções utilizadas, caso faço sentido.

4.1 Release

É no processo de *release* e *Continuous Deployment* que será focado este estudo, porque é aqui que estão os problemas identificados e que serão procuradas soluções para os corrigir. Inicialmente será efetuada uma descrição da organização do trabalho da generalidade das aplicações, organização essa que tem como objetivo ser alterada para um melhor entendimento entre as equipas e para que o processo de *release* seja mais simples e rápido que é o que se pretende.

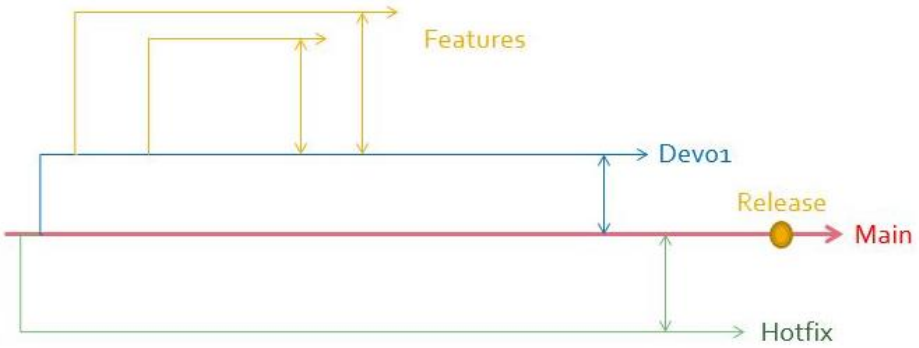


Figura 7 – Estratégia de *Branchs*

Na Figura 7 é possível ver o processo atual de desenvolvimento de *software* na empresa. Existe um repositório de código comum a todas as equipas denominado *Team Foundation Service*, pertencente à Microsoft que armazena a maioria do código aplicacional desenvolvido na empresa. É neste repositório de controlo de versões que é efetuada uma estratégia de ramos que como é possível visualizar é complexa e requer muita atenção para evitar problemas maiores no momento de efetuar os *merges*.

Atualmente, existe um ramo *Main* que contém a versão da aplicação equivalente à versão que no momento está no ambiente de produção. Durante a fase de desenvolvimento de uma nova versão, são criados ramos de desenvolvimento (DEV01) separados do ramo de *Main*. Para uma nova *feature* são criados novos ramos para o seu desenvolvimento.

No final do desenvolvimento, será preciso efetuar todos os *merges* necessários para o ramo de *Main* para que seja lançada uma nova versão para o ambiente de produção. No entanto, a qualquer momento, podem surgir bugs em produção que têm de ser prontamente corrigidos e têm prioridade sobre os desenvolvimentos de uma nova versão. Nesse caso, existem os que se chamam de ramos *HotFix* que têm como objetivo lançarem uma correção rápida que é o mais breve possível incluída no ramo de *Main* e dessa forma é permitido lançar uma versão corretiva para o(s) problema(s) identificado(s).

Posto isto, existe uma série de problemas que ocorrem com frequência tais como:

- Número elevado de pessoas a trabalhar sobre o mesmo ramo.
- Quantidade elevada de *merges* de resolução complexa e morosa.
- Regressões de testes demoradas devido à complexidade de *merges*.

Para que estes problemas sejam reduzidos o trabalho deverá ser realizado maioritariamente sob o ramo de *Main* para que seja possível reduzir cada vez mais a quantidade de ramos de *Hotfix* e desta forma ser mais fácil e rápido instalar novas versões em produção.

4.2 Pipeline

Para o processo de *Continuous Delivery* ser possível é utilizada uma plataforma que oferece condições para construir o que tem pelo nome *pipeline* que é uma combinação de ações, normalmente automáticas, ao longo do tempo em que juntas permitem validar e instalar uma versão de software em vários ambientes de teste e em produção. Essa plataforma, denominada *Jenkins*, com uma serie de *plugins* associados, dão-nos a possibilidade de uma forma muito visual e trivial perceber o estado atual de uma *build* ao longo do tempo, como pode ser visualizado na Figura 8.

Figura 8 – *Pipeline* no *Jenkins*

A ideia é implementar um certo número de estágios que permitam, em cada um deles, ganhar confiança na *build* ao longo do tempo. Com isto, é pretendido responder ao principal objetivo

de detetar falhas na aplicação o mais cedo possível aplicando diferentes tipos de teste em cada um dos estágios.

Na Figura 8, está presente um exemplo típico de uma *pipeline* na empresa. Existem dois estágios iniciais, normalmente, *StageMock* e *Stage1*, que servem para executar testes de componente, num ambiente em que a aplicação é instalada com um perfil Mock, e Testes de Integração com serviços reais, respetivamente. O *StageQA* é utilizado para regressões, ou testes de aceitação manuais, que permitem a *build* avançar para o *StageLive* que fará a entrega do *software* para produção.

Os dois estágios iniciais possuem uma característica própria que os distingue. A nível de infraestrutura, estes dispõem de um grupo de máquinas virtuais preparadas para receber a nova *build* para ser testada de forma totalmente automática. Quando uma nova *build* chega ao *Jenkins*, é adicionada a uma fila de *builds*. Sendo a sua vez de ser executada, o *Jenkins* consulta o grupo de máquinas virtuais para verificar se existe alguma disponível para ser utilizada por este serviço. Existindo esta é bloqueada e associada ao estágio em questão até que o mesmo a liberte.

Tendo uma máquina atribuída à nova *build* lançada para este serviço, o primeiro passo será efetuar a instalação (*deploy*) da aplicação na máquina atribuída. Tendo sucesso, automaticamente será lançado um *job* de testes automáticos que irá ser executado contra a máquina reservada, que contem a nova *build*. Este job de testes, ao terminar a sua execução, disponibiliza um relatório detalhado com o *stacktrace* de possíveis testes que possam falhar.

No final, quer os testes sejam executados com sucesso, quer não o passo seguinte é efetuar um restauro total à maquina utilizada, para garantir que todas as alterações são retiradas e a maquina fica estável para receber uma nova *build* de outro projeto. Desta forma é possível garantir a integridade da infraestrutura e assim retirar interferências alheias que possam retirar confiança e credibilidade dos resultados dos testes executados.

O objetivo do *StageQA* é mais no sentido de testar e garantir a retrocompatibilidade com novas versões de outros serviços. É um ambiente de integração continua em que se uma aplicação inserir lá erros, pode ter impacto com regressões de outras equipas. Este é o ponto onde os *testers* efetuam os seus testes de aceitação e aprovam a *build* para ser instalada no ambiente de produção caso não detetem nada em contrario.

O *StageLive* é o estágio mais recentemente implementado e permite às equipas a autonomia de instalarem as suas alterações nos ambientes de produção com as respetivas monitorizações e acompanhamento durante os primeiros minutos. Esta ação é apenas por precaução, tendo em conta que a *build* passou por varias fases de aprovação e deverá apresentar um grau de confiança e qualidade elevado.

4.3 Pirâmide de testes

Os princípios fundamentais de testes respeitam regras específicas que determinam padrões e formas de atuar para que a função dos *testers* seja efetuada com o maior rendimento possível. Existem boas práticas a respeitar que nem sempre são possíveis de atingir, seja por falta de experiência ou porque é seguido outro caminho que se vem a verificar errado.

Um dos princípios de teste passa pela conhecida Pirâmide de Testes, explicada no subcapítulo Pirâmide de testes, que permite perceber de que forma é possível balancear os testes de forma a garantir uma maior confiança na qualidade da *build* no menor tempo possível.

No entanto, com o desenrolar do desenvolvimento das aplicações na empresa, sendo maioritariamente aplicações Web, o foco direcionou-se muito para os testes *UI* às aplicações sendo acompanhado com testes unitários aos métodos das aplicações.

Isto levou a que em vez de uma pirâmide de testes, fosse alcançado um défice significativo para os testes aos serviços e desta forma de certa forma a uma ampulheta de testes representada na Figura 9.

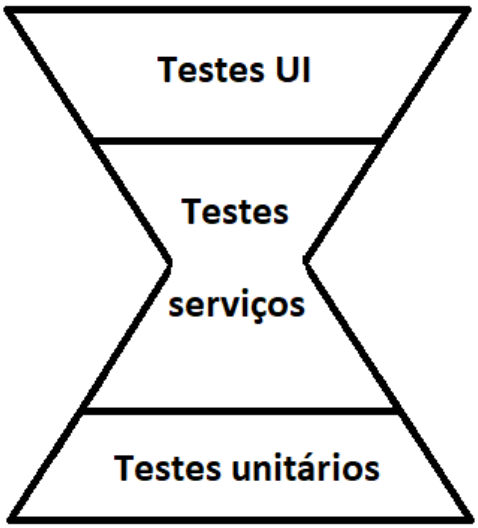


Figura 9 - Distribuição de testes na empresa

Desta forma, o sentido e orientação do trabalho deverá concentrar esforços no desenvolvimento de mais testes aos serviços até alcançar um ponto de superioridade em relação ao *UI tests*.

4.4 Métricas atuais

A empresa ao longo dos últimos tempos tem vindo a utilizar duas métricas que servem para estipular metas a ser alcançadas. São métricas bastante abrangentes que conseguem monitorizar um conjunto alargado de tarefas quantificando-as num valor. É com base nestas

métricas que as equipas efetuam a organização do seu trabalho para que consigam atingir os objetivos a que se propõe. No entanto, este número de métricas é redutor o que leva a que não se consiga identificar com certeza o que leva à variação destes valores. Isto levou à necessidade de adoção de novas métricas que serão identificadas no decorrer deste trabalho.

4.4.1 *Time to approve*

Quando existe a identificação de uma nova funcionalidade, é iniciado o período de desenvolvimento desse mesmo requisito. Como referido anteriormente, durante toda a fase de desenvolvimento o *software* é sujeito a uma série de validações que num momento complicado podem ser detetadas inúmeras falhas nesta aplicação. Após, a correta validação das funcionalidades e a constatação de que a aplicação está num estado de maturidade considerado adequado, a aplicação é considerada aprovada para instalação em produção.

O *Time to approve (TTA)* mede precisamente o tempo que levou desde que se iniciou o desenvolvimento deste requisito até que o mesmo foi aprovado para ser entregue em produção. Esta métrica tem a vantagem de ajudar a perceber quanto tempo um requisito demora a ser desenvolvido e aprovado. No entanto, ignora todas as ações que condicionam estes tempos como tempo de duração dos testes, tempo de correção de defeitos, tempo de desenvolvimento de *software* ou tempo de desenvolvimento de testes.

Apesar disto, é uma métrica mais abrangente que permite entender a progressão do desenvolvimento de *software* na empresa podendo ser agregada com métricas auxiliares aumento assim o seu valor.

4.4.2 *Time to live*

O *Time to live (TTL)*, por sua vez, mede o tempo desde que foi iniciado o desenvolvimento de um novo requisito até que o mesmo é instalado nos ambientes de produção. Esta métrica é bastante útil pois permite perceber a duração do trabalho efetuado após a aprovação de uma build. Existem também inúmeros fatores que influenciam negativamente esta métrica como sobrecarga da equipa responsável pelas instalações nos ambientes de produção, instabilidade dos ambientes de produção e estabilização destes mesmos ambientes.

Esta métrica permite perceber se o apoio que existe às equipas na entrega de valor aos clientes é capaz de dar resposta rapidamente considerando o número de aplicações aprovadas para serem instaladas no ambiente de produção. Desta forma é possível agir em conformidade com os tempos apresentados nesta métrica permitindo que exista uma análise dos problemas que provoquem um valor elevado nesta métrica.

5 Definição da solução

A definição da solução analisa e descreve a solução implementada neste trabalho descrevendo as abordagens utilizadas. Para além da solução apresentada são descritas também as métricas utilizadas e de que forma contribuíram para a concretização da solução.

5.1 Abordagem

Para que o processo de *Continuous Delivery* seja implementado corretamente existem práticas de desenvolvimento de *software* que deverão ser aplicadas em toda a sua fase. Para CD e CI é importante que as alterações em cada uma das *builds* validadas sejam as mais pequenas e especificas possível de forma a permitir rapidamente identificar o problema caso este surja nas regressões à aplicação. Quanto mais complexa for a alteração ou maior o número de alterações mais tempo irá ser investido para a análise dos problemas e desta forma maior o custo será do desenvolvimento.

CD tem uma relação direta com versões e controlo de versões, portanto, um repositório bem organizado e simples permitirá a que o processo seja aplicado mais facilmente removendo complexidade posteriormente na análise e monitorização do processo. Um repositório sempre bastante próximo da versão de produção é uma vantagem, pois permite às equipas agir rapidamente em caso de problemas neste ambiente, lançando versões corretivas que serão validadas num processo adequado e que será entregue em produção rapidamente.

Desta forma, as equipas devem trabalhar partindo sempre de um ramo *Main* removendo a complexidade de gestão de versões dos ramos de *Release*. Centralização do código deverá ser um passo importante a implementar de modo a permitir que o processo de CD seja implementado e que seja tirado partido de todas as suas potencialidades.

5.2 Desenho

A solução foca-se essencialmente na implementação e monitorização da *pipeline* de instalação e validação do *software* desenvolvido pelas equipas. No entanto, para perceber se existem fatores externos que condicionam o tempo de execução de uma *pipeline* como tempo de desenvolvimento de um requisito ou o tempo de correção de um defeito encontrado na *pipeline*, devem ser efetuadas análises a estes fatores para identificar possíveis problemas alheios que atrasem a regularidade da execução da *pipeline*. Apesar disto, deverá ser efetuada também uma análise adequada à própria *pipeline* de modo a identificar problemas internos ao processo para agir em conformidade com os mesmos.

5.2.1 *Pipeline*

Um processo de *Continuous Delivery* é normalmente constituído por uma *pipeline* que consiste numa sequência automática de tarefas, realizadas pelo computador, que permitem validar e analisar o estado de maturidade de uma aplicação ou pedaço de código. Consoante a *pipeline* avança, a sua complexidade deve acompanhar este avanço e aumentar de forma ponderada e ajustável. Não faz sentido avançar numa validação a partir do momento em que um problema é detetado. Todas as restantes validações vão ser inconclusivas enquanto não existir uma análise concreta dos impactos deste problema e a sua correção não for efetuada.

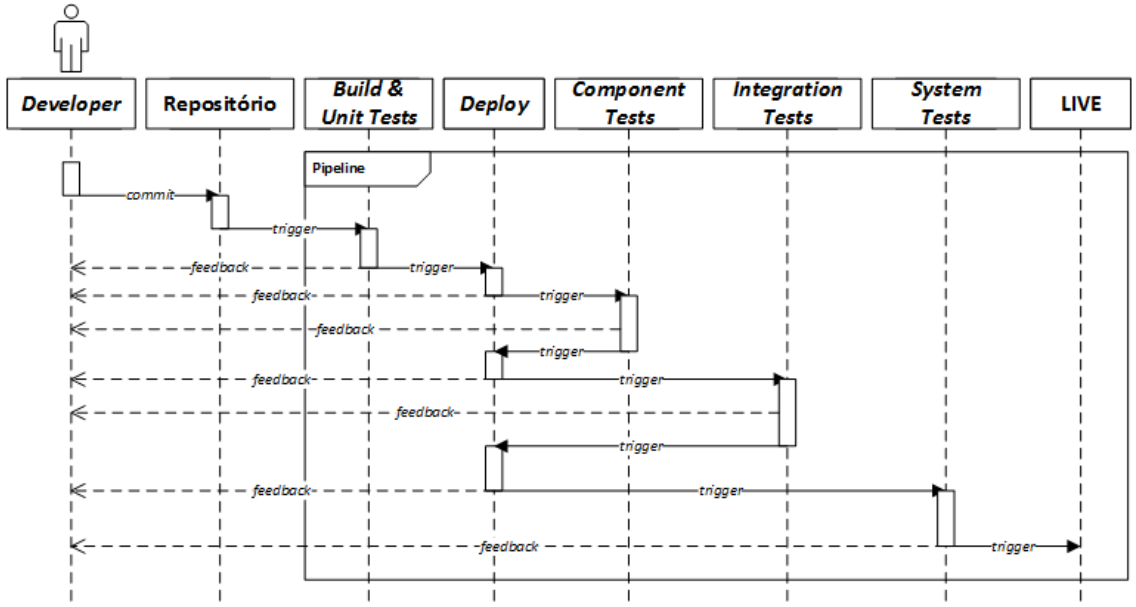


Figura 10 – Processo de *Continuous Delivery* a implementar

Na Figura 10, consegue-se analisar a solução implementada na empresa com a inclusão de vários tipos de teste que juntos e de forma contínua irão validar o *software* com *feedback* recorrente que permitirá uma ação rápida e desta forma diminuir o tempo de desenvolvimento. A *pipeline* consiste em executar os testes com menor complexidade primeiro devido ao facto de serem testes tipicamente mais pequenos, com menos variáveis que possam afetar o resultado e atribuídos a uma parte especifica da aplicação. O objetivo desta abordagem é detetar o máximo de problemas possível nas primeiras fases de testes de forma a reduzir o esforço e o tempo da sua resolução e para aumentar o grau de confiança das fazes de teste seguintes.

Uma *pipeline* tipicamente inicia com a compilação do projeto e a execução de testes unitários, conforme ilustrado já anteriormente em Figura 2. Nesta solução, devido à infraestrutura da empresa e à utilização dos controladores de versões *TFS,* a parte de testes unitários será efetuada antes de a aplicação ser enviada para a *pipeline* devido ao facto de ser mais eficiente e dar um *feedback* mais rápido ao *developer* responsável por lançar uma nova versão da aplicação.

5.2.2 *Testes*

A execução dos vários tipos de teste da *pipeline* deverá ser efetuada em ambientes de teste com a arquitetura o mais próximo dos ambientes de produção possível. Isto requer uma manutenção diária destes ambientes com a atualização contínua das novas versões instaladas em produção das várias aplicações fundamentais ao bom funcionamento da aplicação a testar. Isto é importante para garantir uma execução de testes independente num ambiente estável com garantias de que se existirem problemas, estes serão das alterações introduzidas pela nova versão da aplicação.

Os testes adotados na implementação da *pipeline* são testes que têm por base a pirâmide de testes descrita anteriormente em Pirâmide de testes. Isto garante a correta aplicação de conceitos de teste comuns a todos e não causa problemas nas nomenclaturas aplicadas. O princípio do aumento da complexidade dos testes ao longo do tempo está também bastante vincada na solução e é desta forma que a abordagem a aplicar será elaborada.

Todos os novos requisitos desenvolvidos devem ser considerados para teste quando existirem testes capazes de os validar. Este esforço tem de ser incluído na fase de desenvolvimento de forma a permitir um equilíbrio de testes e funcionalidades de forma a que as regressões de testes consigam validar a totalidade da aplicação permitindo desta forma aumentar o grau de confiança ao instalar a aplicação nos ambientes de produção.

Os tipos e níveis de teste a ter em conta na solução deverão ser utilizados na aplicação de forma ponderada de forma a garantir a correta validação da aplicação consoante as boas práticas de testes. Estes tipos e níveis de teste a considerar estão descritos nas secções Níveis de testes e Tipos de testes.

5.3 Gestão de ramos aplicada

Com o intuito de aproximar as equipas de um processo automático com *releases* contínuas e com rápido *feedback* foram introduzidas melhorias no processo de gestão de ramos no desenvolvimento das aplicações. A abordagem aplicada, para organizar e simplificar o desenvolvimento de *software* e de modo a que o método de trabalho seja cada vez mais próximo do *Continuous Delivery,* foi introduzida uma nova forma de gestão de ramos em que o principal objetivo visa reduzir o tempo investido em *merges requests* e desta forma as equipas sentem-se mais seguras e organizadas na sua forma de trabalhar.

Figura 11 – Estrutura de ramos aplicada

Esta nova abordagem é possível devido às *pipelines* existentes em que é possível rapidamente testar um novo desenvolvimento com um *feedback* importante e desta forma é possível atuar no imediato sobre eventuais problemas que existam. No entanto, o caminho a seguir será sempre investir nos métodos de trabalho e procurar atingir o esperado que será trabalhar apenas sobre o ramo de *Main* em que a gestão do número de pessoas a trabalhar sobre o mesmo ramo envolve maiores desafios como a gestão do próprio código, mas que é um passo importante para cada vez mais se melhorarem os processos de *Continuous Delivery.*

Para alcançar esta nova forma de trabalho foi necessário perceber de que forma a equipa se conseguiria organizar para que fosse possível trabalhar de forma correta e sem perda de rendimento de todos. A comunicação é essencial e a atuação rápida sobre problemas encontrados é muito importante para que o trabalho de uns elementos da equipa não seja impactado por todos e por fim a coordenação é o crucial para que tudo seja possível. Para tal a seleção do projeto e a atribuição deste projeto a uma só equipa proporcionaram com que esta abordagem fosse possível. Quanto maior for o número de pessoas a trabalhar no mesmo repositório mais difícil se torna atingir estas abordagens. De salientar também que estas abordagens são possíveis devido à mentalidade e estratégias implementadas pela equipa de tentar chegar à meta em que cada *merge* para *Main* irá ser instalada nos ambientes de produção em questão de minutos.

5.4 Esforço aplicado em testes

Durante o processo de desenvolvimento, a pirâmide de testes é respeitada e o processo de validação de *builds* é validado quase na totalidade de forma automática. O maior investimento do desenvolvimento de testes unitários proporcionou uma cobertura de código aplicacional em média na ordem dos 85%. Desta forma a maioria dos cenários estão cobertos desde o início do desenvolvimento. Posteriormente, existiu um grande investimento no desenvolvimento de testes automáticos em metodologias *Mock*. O *Mock* permite testar a aplicação de forma isolada removendo a instabilidade recorrente das suas dependências, desta forma é possível testar a aplicação com estabilidade e rapidamente antes de serem efetuados uns conjuntos de teste menores que irão validar a integração da aplicação com as suas dependências. Nesta fase, o risco é menor porque o funcionamento da aplicação foi já validado antes desta fase. No entanto, é importante serem efetuados alguns testes exploratórios para perceber se não existem problemas com a integração dos componentes validando requisitos não funcionais como a performance.

5.5 Infraestrutura

Devido à abordagem de grupo de máquinas virtuais (VMs) utilizadas por todas as aplicações em que o conceito consiste na atribuição de uma máquina disponível a uma *pipeline* de uma aplicação para que esta possa ser testada isto provoca uma fila de *pipelines*. Uma particularidade destas máquinas é que possuem todas as aplicações da Farfetch instaladas o

que permite que uma determinada aplicação possua as suas dependências e possa ser rapidamente testada. No entanto, uma aplicação que não necessite das suas dependências instaladas para ser validada numa primeira fase, vai ter um período, por vezes longo, à espera de uma *VM* livre para que possa utilizar.

Isto levou a uma alteração a nível de infraestrutura que proporcionou a algumas aplicações particulares, com princípios de micro serviços [31] terem máquina sempre disponíveis e não precisarem de esperar numa fila de *pipelines*. Esta mudança consistiu na criação de novas *VMs*, mais limitadas a nível de especificações, mas que proporciona às aplicações maioritariamente *Mock* um ambiente ideal para que possam ser validadas rapidamente. Por não precisarem das suas dependências, as aplicações *Mock*, não utilizam Base de dados o que faz com não introduzam alterações significativas ao estado da *VM* e que esta não necessite de ser restaurada no final de cada validação. Desta forma, foi possível retirar o processo de restauração das VMS da *pipeline* e desta forma, retirar certa de 15 minutos de cada restauro.

Ou seja, com a criação de máquinas dedicadas para a validação de aplicações *Mock*, foi possível incentivar as equipas a investirem neste tipo de testes que os fará respeitar a pirâmide, reduzir o tempo de espera de uma *VM* para que fosse possível validar a aplicação e para além do tempo de espera, reduzir em cerca de 15 minutos o tempo de restauro da *VM* utilizada para que esta fique num estado consistente para ser utilizada por outras aplicações.

5.6 Monitorização

Durante todo o período de implementação da solução o processo deverá ser acompanhado de métricas relacionadas com testes que permitam avaliar a eficiência do processo aplicado. O objetivo destas métricas é de monitorizar os testes executados em cada *pipeline* para que seja possível efetuar ajustes no processo e se a distribuição dos testes e do trabalho das equipas está a ser feito corretamente. Para além das métricas já existentes na empresa, nomeadamente, *TTL* e *TTA* devem ser consideradas novas métricas para tornar a monitorização e controlo das fases de testes mais minucioso e com um critério até agora não conseguido na empresa.

5.6.1 *Métricas utilizadas*

As métricas recolhidas e apresentadas neste documento são que mais se apropriaram à realidade do projeto e das metodologias de trabalho adotadas pela equipa. Devido ao facto de a equipa agir rapidamente aos defeitos encontrados na aplicação, pela *pipeline,* estes são prontamente corrigidos no *commit* seguinte. Desta forma, ações são partilhadas pela equipa para que estes sejam corrigidos. O que faz com que não existam registos suficientes para uma análise de métricas relacionadas com defeitos. Posto isto, o trabalho focou-se na monitorização do esforço no desenvolvimento de testes e de infraestrutura.

Todas as métricas apresentadas foram obtidas no mesmo contexto e com as mesmas condições, apesar do elevado perfil *Mock* da aplicação, existiu este cuidado para evitar que fatores externos afetassem o estudo e desta forma a equipa caminhasse no sentido errado.

5.6.1.1 *Quantidade de casos de teste*

A quantidade de *casos de teste* implementados é importante porque permite perceber a complexidade em validar todas as funcionalidades da aplicação e o esforço aplicado pela equipa em validar estas funcionalidades. Isto dá visibilidade à equipa, juntamente com a duração dos mesmos, calcular o esforço necessário para validar a aplicação. Para além disto, estas métricas apresentam uma linha no tempo interessante em que se consegue facilmente perceber que o desenvolvimento da aplicação numa fase inicial apresenta valores muito mais instáveis com vários *outliers,* normais da consolidação de processos.

Para além disto, a pirâmide de testes esteve sempre presente na hora de definir casos de teste e é percetível que a pirâmide foi respeitada o que permite que seja implementado um processo de validação progressiva da aplicação com *feedback* continua utilizando uma *pipeline* proposta anteriormente.

5.6.1.2 *Testes unitários*

Os testes unitários são testes de caixa branca que têm como objetivo validar unidades pequenas de código. São testes muito próximos do código e que são tipicamente mais simples, atómicos e rápidos na sua execução. São os testes executados em tempo de compilação e que rapidamente fornecem um *feedback* do potencial estado da aplicação. São os testes em maior quantidade devido à sua facilidade de execução e *feedback* pertinente.

Estes testes são consensuais na equipa e são obrigatórios na fase de desenvolvimento da aplicação. A sua qualidade e estabilidade é importante, porque nos dias que correm existem editores de código que executam os testes unitários sempre que se efetuar uma alteração da aplicação o que permite rapidamente perceber se as novas alterações tiverem algum impacto nas funcionalidades já existentes na aplicação. Representam uma primeira fase de validação do *software* desenvolvido.

Tabela 2 – Quantidade de testes unitários em Maio

Quantidade	Versão
287	1.0.17128.60
283	1.0.17128.61
290	1.0.17128.62
296	1.0.17132.73
331	1.0.17132.74
311	1.0.17132.77
311	1.0.17135.78
329	1.0.17135.79
329	1.0.17135.80
358	1.0.17137.88
358	1.0.17137.89
391	1.0.17138.93
391	1.0.17138.94
412	1.0.17143.110
412	1.0.17145.113

Tabela 3 – Quantidade de testes unitários em Julho

Quantidade	Versão
729	1.0.17187.241
729	1.0.17187.242
732	1.0.17187.243
750	1.0.17187.244
750	1.0.17187.245
750	1.0.17188.246
748	1.0.17188.248
748	1.0.17193.261
748	1.0.17194.266
751	1.0.17194.267
751	1.0.17194.268
751	1.0.17195.271
751	1.0.17195.272
773	1.0.17195.276
773	1.0.17195.277

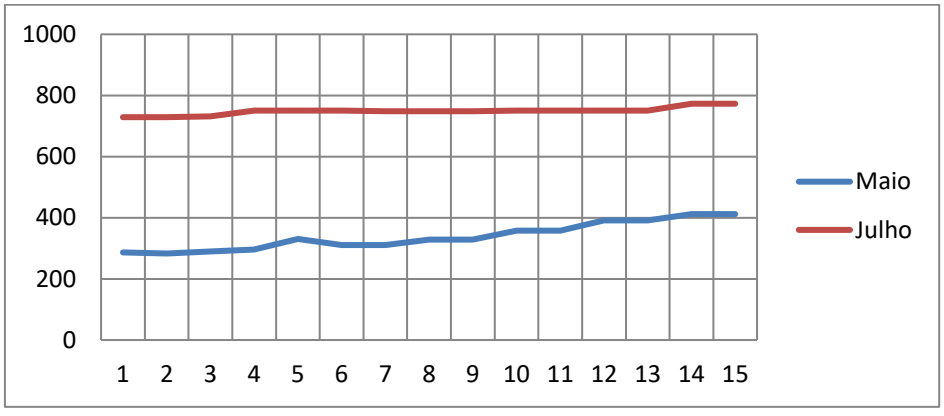


Figura 12 – Tendência da quantidade de testes unitários

Na Tabela 2, na Tabela 3 e na Figura 12 é possível perceber o constante aumento da sua quantidade devido à constante evolução da aplicação. A tendência com o evoluir da aplicação é que este número estabilize e que não existem grandes variações. Deste modo a pirâmide de testes é respeitada algo que é muito importante atingir os objetivos propostos.

5.6.1.3 *Testes de Componente*

Estes testes têm o objetivo de validarem a aplicação numa perspetiva do utilizador com testes de componente, a um perfil *Mock* da aplicação, à UI da aplicação em que são testados os métodos de renderização, navegação e sessões. Devido ao perfil *Mock* da aplicação, em que todas as dependências são substituídas por ferramentas de resposta pré-definida e

automática, estes testes, utilizando tecnologia *Selenium*, apresentam tempos de execução bastante interessantes que podem ser analisados em Tempo de execução dos de teste. Como referido anteriormente estes testes representam uma maior complexidade e tempo de execução o que façam que sejam inseridos num patamar superior da pirâmide de testes e deste modo sejam em menor quantidade.

Tabela 4 – Quantidade de testes de componente em Maio

Quantidade	Versão
98	1.0.17128.60
98	1.0.17128.61
100	1.0.17128.62
106	1.0.17132.73
110	1.0.17132.74
110	1.0.17132.77
112	1.0.17135.78
110	1.0.17135.79
110	1.0.17135.80
116	1.0.17137.88
64	1.0.17137.89
124	1.0.17138.93
124	1.0.17138.94
137	1.0.17143.110
143	1.0.17145.113

Tabela 5 – Quantidade de testes de componente em Julho

Quantidade	Versão
207	1.0.17187.241
207	1.0.17187.242
207	1.0.17187.243
209	1.0.17187.244
209	1.0.17187.245
209	1.0.17188.246
209	1.0.17188.248
209	1.0.17193.261
209	1.0.17194.266
209	1.0.17194.267
207	1.0.17194.268
207	1.0.17195.271
207	1.0.17195.272
207	1.0.17195.276
208	1.0.17195.277

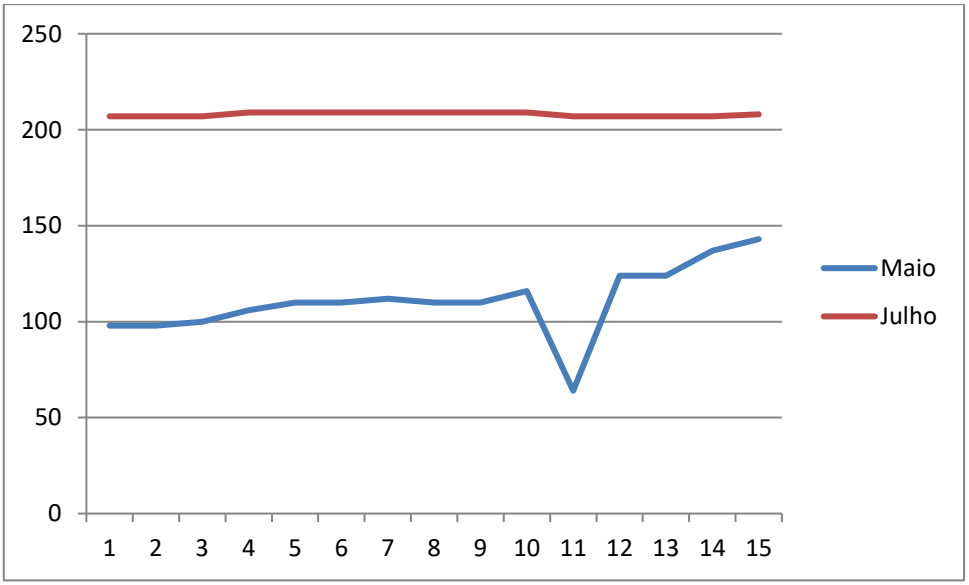


Figura 13 – Tendência da quantidade de testes de componente

Através da Tabela 4, da Tabela 5 e da Figura 13 é percetível o investimento da equipa nos testes de UI validando assim uma componente muito importante da aplicação. O crescimento destes testes ao longo do tempo é clara e é possível detetar uma quebra no número de destes executados, pontual e que serviu para validar um componente especifico da aplicação.

O número destes testes duplicou ao longo do tempo de foram aplicadas algumas técnicas para permitir manter o tempo de execução. Técnicas como paralelização de testes e reutilização de *WebDrivers* que permitem assim executarem vários testes em simultâneo aumentando a produtividade destes testes.

5.6.1.4 *Tempo de execução dos casos de teste*

O tempo de execução dos casos de teste é uma métrica que requer sempre especial atenção porque é uma métrica que pode variar facilmente por fatores externos à aplicação, como a instabilidade de infraestrutura. No entanto, são valores muito importantes e que são preponderantes para que os objetivos sejam atingidos.

Na Tabela 6 e na Tabela 7, estão apresentados os tempos em minutos da duração dos testes de componente desenvolvidos em *Selenium,* responsáveis por testar a navegação na aplicação e que por si só são já mais demorados na sua execução.

Tabela 6 – Tempo de execução dos testes de componente em Maio

Tempo/minutos	Versão
3,36	1.0.17128.60
2,16	1.0.17128.61
4,01	1.0.17128.62
4,49	1.0.17132.73
6,44	1.0.17132.74
6,29	1.0.17132.77
4,04	1.0.17135.78
4,29	1.0.17135.79
4,1	1.0.17135.80
4,66	1.0.17137.88
3,58	1.0.17137.89
4,94	1.0.17138.93
4,45	1.0.17138.94
5,75	1.0.17143.110
5,31	1.0.17145.113

Tabela 7 – Tempo de execução dos testes de componente em Julho

Tempo/minutos	Versão
4	1.0.17187.241
4,02	1.0.17187.242
4,03	1.0.17187.243
3,88	1.0.17187.244
3,97	1.0.17187.245
3,95	1.0.17188.246
4,24	1.0.17188.248
4,01	1.0.17193.261
4,28	1.0.17194.266
4,53	1.0.17194.267
4,09	1.0.17194.268
5,25	1.0.17195.271
4,9	1.0.17195.272
5,72	1.0.17195.276
6,17	1.0.17195.277

Associando estes tempos à quantidade de testes de componente, apresentados na Tabela 4 e na Tabela 5, consegue-se perceber que desde a primeira recolha destas métricas a quantidade de testes executados duplicou, mas, no entanto, os tempos mantiveram-se praticamente

inalterados. Para isto, medidas tiveram de ser tomadas e foi introduzida a metodologia de paralelização de testes. Isto permite em que vários testes possam ser executados em paralelo de modo a que os tempos totais da execução de testes possam baixar de forma significativa.

A Figura 14 é útil para rapidamente se perceber a tendência dos tempos dos testes de componente em que claramente é visível uma instabilidade típica de uma fase inicial do desenvolvimento da aplicação e a tendência de um período de desenvolvimento mais avançado da mesma.

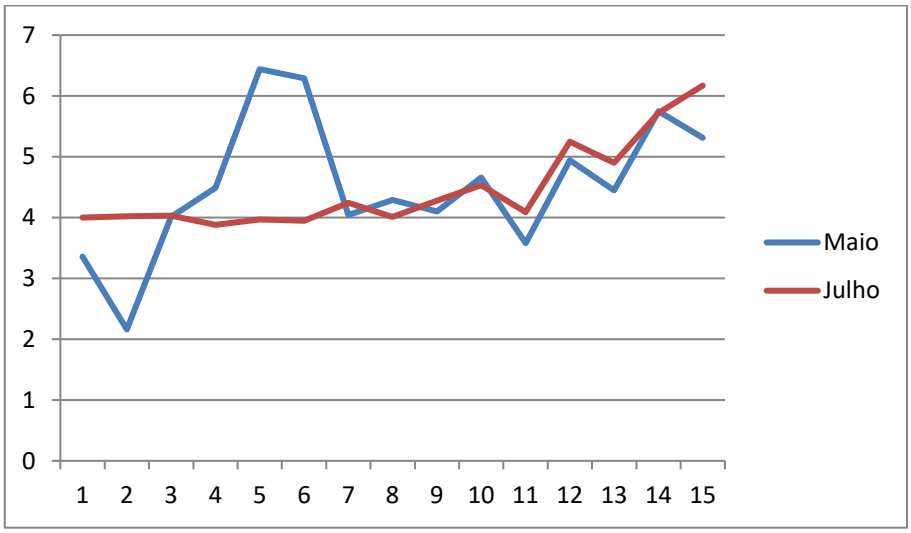


Figura 14 – Tendência do tempo de execução dos casos de teste

É também percetível, para alem da estabilidade dos tempos de execução, que os tempos se mantêm constantes em níveis quantitativos ao longo do tempo devido ao paralelismo. O facto destes testes possuírem um perfil da aplicação 100% *Mock* permite obter tempos consideravelmente baixos e desta forma validar a aplicação alvo num ambiente complementarmente controlado e sem erros alheios à mesma. Desta forma, o ambiente dedicado a este tipo de testes foi também um impulsionador para que estes valores fossem alcançados e que permite à equipa em questão de cerca de 4 minutos conseguir validar a sua aplicação com este tipo de teste.

5.6.1.5 *Cobertura de Requisitos*

Um dos fatores obrigatórios para que um requisito seja aprovado é que o mesmo tem de ser validado de testado com testes automáticos. Ou seja, só seria possível entregar uma nova funcionalidade e determina-la como concluída quando esta possuísse testes automáticos capazes de a validar. É intrínseco do cenário de desenvolvimento adotado pela equipa que os testes automáticos são requisito obrigatório para a entrega do *software* ao cliente.

Com esta abordagem foi possível caminhar no sentido de alcançar um processo de *Continuous Delivery* próximo do desejado com desenvolvimentos e instalações em produção totalmente automatizados. Desta forma a equipa consegue obter uma cobertura da aplicação bastante elevada e tem confiança no *software* desenvolvido.

Com isto, a importância da *pipeline*, e o seu estado, é fundamental para a equipa. É muito importante manter a *pipeline* estável e agir rapidamente em caso de falha e manter o processo preparado e pronto para a qualquer momento se efetuarem intervenções em produção rapidamente. Resumidamente, a cobertura dos requisitos com testes automáticos foi de **100%**.

5.6.1.6 *Cobertura de código*

As métricas de cobertura de código são todas obtidas no âmbito de testes unitários. Devido a mudanças de infraestrutura não foi possível recolher a percentagem de cobertura de código dos outros tipos de teste. Apesar disto, é possível claramente perceber a importância que a equipa entregou à metodologia de testes e o nível e rigor que alcançaram na validação da sua aplicação, passo importante para alcançar o *Commit to Live* [33] rapidamente. A ferramenta utilizada para recolher esta métrica, *OpenCover*, permite retirar três tipos de cobertura, nomeadamente, *Class Coverage, Method Coverage* e *Statement Coverage.*

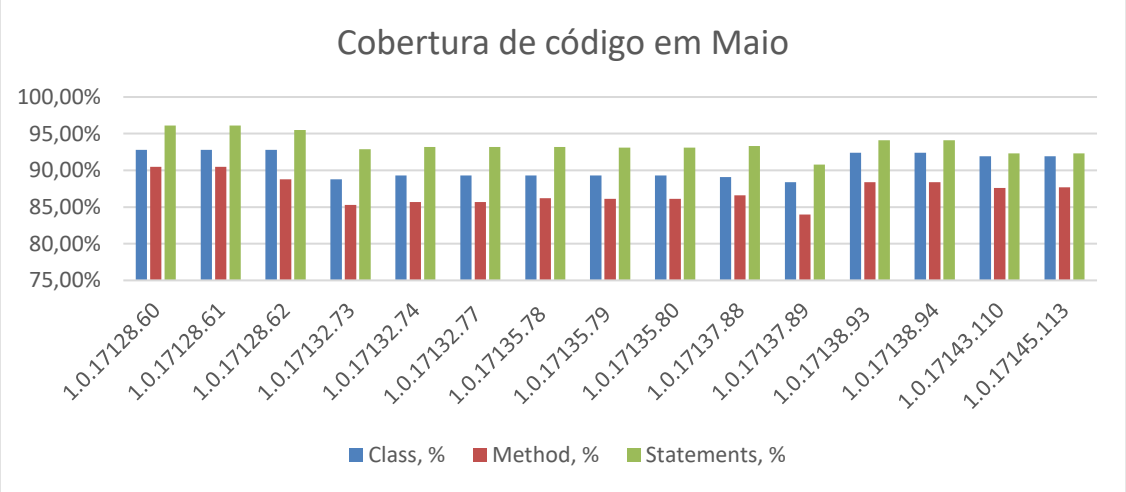


Figura 15 – Cobertura de código recolhida em Maio

Na Figura 15, referente a uma fase precoce do desenvolvimento da aplicação entende-se que apesar de existir alguma inconsistência nos seus valores, a percentagem de cobertura da aplicação foi desde o início muito alta e permitiu desde cedo à equipa ganhar confiança nos seus desenvolvimentos e elevar o nível de confiança nos seus testes. Para manter estes níveis de cobertura uma abordagem implementada foi a introdução de regras e requisitos da aplicação. Neste caso foi introduzida a regra que uma determinada versão não conseguiria avançar na *pipeline* com uma percentagem de cobertura de 85%. Este valor, considerado pelo *Statement Coverage,* foi definido numa fase bastante inicial do projeto e foi considerada ambiciosa, no entanto, foi atingida rapidamente e nunca foi impedimento para aprovação de uma *release*.

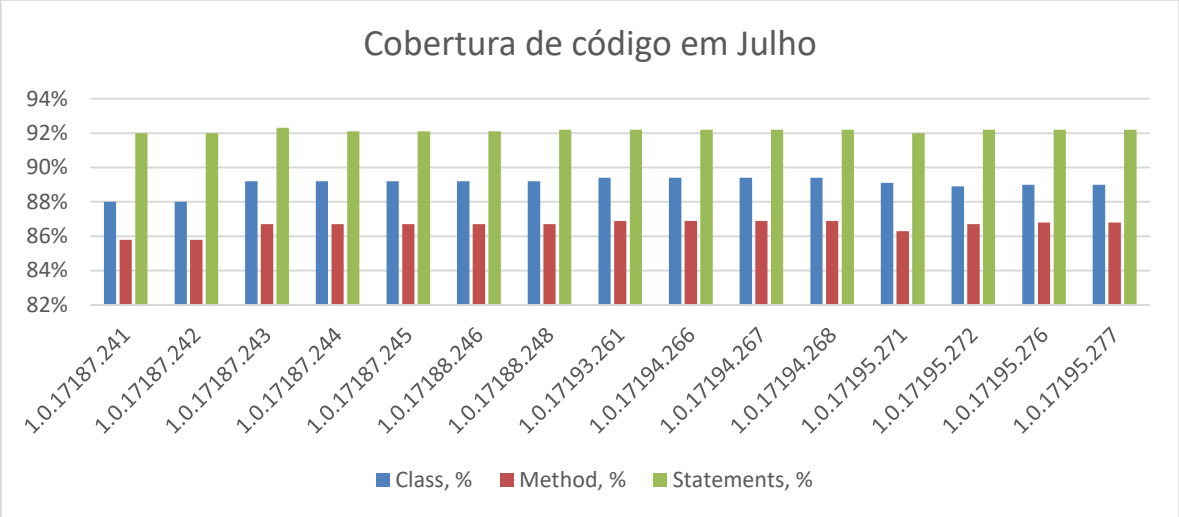


Figura 16 – Cobertura de código recolhida em Julho

Numa fase adiantada do desenvolvimento da aplicação interessa perceber que as mudanças ao serem menores, praticamente não existem variação de cobertura de código. A principal vantagem é que os novos desenvolvimentos introduzidos serão acompanhados com uma elevada percentagem de cobertura o que permitirá perceber rapidamente o impacto destas alterações na restante aplicação. Importante voltar a salientar que esta percentagem é referente apenas aos testes unitários da aplicação.

Para explicar melhor cada uma das coberturas aplicadas, de seguida irá ser efetuada uma análise a cada uma delas, salientando que as três se influenciam e que analisar uma de forma independente pode não trazer o valor pretendido para o projeto.

5.6.1.6.1 Class coverage

Esta métrica é utilizada pela equipa para rapidamente perceber quais as áreas da aplicação que são cobertas por testes automáticos e desta forma delinear trabalho futuro para colmatar determinadas áreas não cobertas. É uma métrica útil porque com uma análise superficial é possível a um responsável de qualidade da equipa identificar pontos de risco da aplicação. No entanto, esta métrica por si só não dá segurança à equipa em termos de defeitos ou falhas na mesma, portanto, esta métrica só faz sentido com as outras duas métricas descritas em Method coverage e Statement Coverage.

Tabela 8 – *Class Coverage* recolhido em Maio

BuildVersion	***Class, %***
1.0.17128.60	92,80%
1.0.17128.61	92,80%
1.0.17128.62	92,80%
1.0.17132.73	88,80%
1.0.17132.74	89,30%
1.0.17132.77	89,30%
1.0.17135.78	89,30%
1.0.17135.79	89,30%
1.0.17135.80	89,30%
1.0.17137.88	89,10%
1.0.17137.89	88,40%
1.0.17138.93	92,40%
1.0.17138.94	92,40%
1.0.17143.110	91,90%
1.0.17145.113	91,90%

Tabela 9 – *Class Coverage* recolhido em Julho

BuildVersion	***Class, %***
1.0.17187.241	88%
1.0.17187.242	88%
1.0.17187.243	89,20%
1.0.17187.244	89,20%
1.0.17187.245	89,20%
1.0.17188.246	89,20%
1.0.17188.248	89,20%
1.0.17193.261	89,40%
1.0.17194.266	89,40%
1.0.17194.267	89,40%
1.0.17194.268	89,40%
1.0.17195.271	89,10%
1.0.17195.272	88,90%
1.0.17195.276	89%
1.0.17195.277	89%

Analisando a Tabela 8 e Tabela 9 percebe-se a cobertura de áreas funcionais é alta, acima dos objetivos internamente definidos em que a cobertura deveria ser superior a 85%. Esta métrica permitiu à equipa organizar o seu trabalho e tornar os planeamentos mais simples devido à rápida perceção de carência de testes em algumas áreas funcionais.

5.6.1.6.2 Method coverage

O *Method coverage* é uma das métricas que aumentaria o seu valor com a contabilização dos testes UI que não foi possível. Os testes unitários, apesar de mostrarem uma grande cobertura da aplicação não são capazes de validar os métodos de renderização da página de forma fiável e são tipicamente deixados para os testes mais adiantados da aplicação como os testes de componente ou os testes de integração.

No entanto, para um *Developer,* perceber quais os métodos que estão a ser cobertos pelos testes automáticos é ótimo porque rapidamente consegue entender o fluxo dos testes e perceber quais os cenários que podem ser idealizados como teste para colmatar a baixa percentagem de *Method Coverage.*

É uma métrica consensual no processo de *Continuous Delivery* e adotada em todos os novos projetos adotados na empresa.

Tabela 10 – *Method Coverage* recolhido em Maio

BuildVersion	***Method*, %**
1.0.17128.60	90,50%
1.0.17128.61	90,50%
1.0.17128.62	88,80%
1.0.17132.73	85,30%
1.0.17132.74	85,70%
1.0.17132.77	85,70%
1.0.17135.78	86,20%
1.0.17135.79	86,10%
1.0.17135.80	86,10%
1.0.17137.88	86,60%
1.0.17137.89	84,00%
1.0.17138.93	88,40%
1.0.17138.94	88,40%
1.0.17143.110	87,60%
1.0.17145.113	87,70%

Tabela 11 – *Method Coverage* recolhido em Julho

BuildVersion	***Method*, %**
1.0.17187.241	85,80%
1.0.17187.242	85,80%
1.0.17187.243	86,70%
1.0.17187.244	86,70%
1.0.17187.245	86,70%
1.0.17188.246	86,70%
1.0.17188.248	86,70%
1.0.17193.261	86,90%
1.0.17194.266	86,90%
1.0.17194.267	86,90%
1.0.17194.268	86,90%
1.0.17195.271	86,30%
1.0.17195.272	86,70%
1.0.17195.276	86,80%
1.0.17195.277	86,80%

Na primeira recolha desta métrica é clara uma queda de 5% desta métrica que foi posteriormente colmatada com novos testes automáticos que provocou uma nova subida. No entanto, não voltando a alcançar os 90% iniciais, provavelmente devido à implementação de novas funcionalidades de renderização de página que provocou a descida. Apesar disto, a segunda recolha revela apenas variações mínimas típicas da correção de problemas, testes e novas experiencias necessárias à aplicação.

Resumindo, na ótica de *Developer,* esta é uma métrica bastante útil e que consegue ser rapidamente corrigida devido ao facto de ser a mais próxima da mentalidade do desenvolvimento de *software* na ideologia de um programador.

5.6.1.6.3 Statement Coverage

O *Statement Coverage* das três métricas de cobertura é a mais influente no projeto porque é a que possui requisito mínimo para que a aplicação seja considerada para produção. No entanto, é também a cobertura mais difícil de melhorar devido à sua complexidade.

O *Statement Coverage* é referente à totalidade de linhas de código cobertas pelos testes em questão. O que faz com que seja impossível existir >0% de *Class Coverage* e 0% de *Statement Coverage* ou >0% de *Method Coverage* e 0% *Statement Coverage*. No entanto, >0% de *Statement Coverage* implica obrigatoriamente >0% de *Method* e *Class Coverage.*

Tabela 12 – *Statement Coverage* recolhido em Maio

BuildVersion	Statements, %
1.0.17128.60	96,10%
1.0.17128.61	96,10%
1.0.17128.62	95,50%
1.0.17132.73	93%
1.0.17132.74	93,20%
1.0.17132.77	93,20%
1.0.17135.78	93,20%
1.0.17135.79	93,10%
1.0.17135.80	93,10%
1.0.17137.88	93,30%
1.0.17137.89	90,80%
1.0.17138.93	94,10%
1.0.17138.94	94,10%
1.0.17143.110	92%
1.0.17145.113	92%

Tabela 13 – *Statement Coverage* recolhido em Maio

BuildVersion	Statements, %
1.0.17187.241	92%
1.0.17187.242	92%
1.0.17187.243	92,30%
1.0.17187.244	92,10%
1.0.17187.245	92,10%
1.0.17188.246	92,10%
1.0.17188.248	92,20%
1.0.17193.261	92,20%
1.0.17194.266	92,20%
1.0.17194.267	92,20%
1.0.17194.268	92,20%
1.0.17195.271	92%
1.0.17195.272	92,20%
1.0.17195.276	92,20%
1.0.17195.277	92,20%

Analisando a Tabela 12 e Tabela 13, percebe-se e de acordo com todas as métricas aplicadas que na fase inicial do projeto a dificuldade em manter o nível de cobertura elevado é mais complicado devido às constantes alterações da aplicação. O desenvolvimento de funcionalidades e respetivos testes pode ser desfasado no tempo o que provoca em que determinados momentos a cobertura seja inferior, no entanto, a equipa sempre procurou acompanhar nas novas versões as novas funcionalidades com os respetivos testes automáticos e desta forma os dados recolhidos são fidedignos e úteis para a perceção da cobertura dos testes desenvolvidos face ao desenvolvimento da aplicação.

6 Avaliação e resultados

Esta secção terá o objetivo de efetuar a análise e avaliação da solução proposta. Quais as abordagens a utilizar durante a fase de análise como grandezas para avaliar a solução, hipótese(s) e testes estatísticos de modo a suportar os resultados do trabalho desenvolvido e metodologias de avaliação.

6.1 Abordagem

Tendo em conta que o trabalho proposto visa no principal objetivo de implementar um processo de *Continuous Delivery* com vários tipos e níveis de teste e métricas, que garantem uma *release* de *software* no menor tempo possível, capazes de os monitorizar o trabalho terá por base uma serie de experimentações. Para além de uma comparação das métricas obtidas inicialmente com as métricas obtidas no final do trabalho terão de existir medições periódicas que ajudem efetivamente a perceber a verdadeira evolução do processo.

Para analisar a evolução do trabalho, a análise das experimentações teve de se focar em avaliar fatores como os tempos, tendência, resultados e quantidade de testes executados ao longo de um processo de *release*. Estas grandezas identificadas são importantes pois permitam que exista uma análise concreta da implementação da solução com dados que são possíveis analisar através dos métodos estatísticos apropriados. Para efetuar esta análise irão ser utilizados nos estudos estatísticos as próprias métricas implementadas na solução do problema.

6.2 Monitorização

O desenvolvimento do projeto conteve uma forte componente de experimentação que visa a monitorização de várias métricas de forma a tomar ações para que estas possam ser melhoradas. Tendo em conta que as métricas a ser utilizadas para o estudo estatístico são as que são sugeridas na solução, estas serão também consideradas como sendo as nossas hipóteses para o estudo estatístico.

Para a recolha destas métricas é importante que os vários testes que irão permitir avaliar a solução final sejam executados nas mesmas condições de infraestrutura para eliminar a existência de fatores externos que possam condicionar a correta execução de testes, fatores como problemas de memórias, *CPUs* e arquiteturas *Windows*. Para além das condições de infraestrutura, os testes considerarão as mesmas aplicações comparando os resultados de cada uma de forma individual para posteriormente serem comparadas para que exista uma perceção se a solução estará a atingir os objetivos.

6.3 Avaliação

Para avaliação das medidas aplicadas e métricas obtidas é fundamental perceber a variação das métricas *TTA* e *TTL* e perceber se apresentam valores que fornecem confiança e credibilidade para que a equipa consiga analisar os mesmos e gerir os seus desenvolvimentos de acordo com as variações destas métricas.

Com isto, foram recolhidas as métricas de *TTA* e *TTL* das versões instaladas em produção, bem como os testes de cada fase da *pipeline* que influenciem diretamente estas métricas, e aplicado o estudo estatístico que demonstrará se a distribuição destes valores é ou não normal.

6.3.1 *Time to Live e Time to Approve*

Estas métricas como referido neste documento têm o objetivo de medir o tempo de aprovação e *Go Live* de uma versão da aplicação. Estas métricas representam a soma dos tempos de cada fase da *pipeline* que no final permitem à equipa identificar os pontos mais demorados e perceber de que forma poderão atuar e tornar os processos e métodos mais eficazes e deste modo reduzir estas métricas que é o principal objetivo.

Tipicamente, com o evoluir da aplicação e com o crescimento das baterias de teste, estes valores tendem em aumentar de forma significativa. Isto revela um desafio porque manter ou reduzir estes tempos requer um estudo dos métodos aplicados e recorrendo às métricas anteriores apresentadas neste documento, é possível perceber de que forma a equipa deve atuar para não penalizar o seu *TTL*.

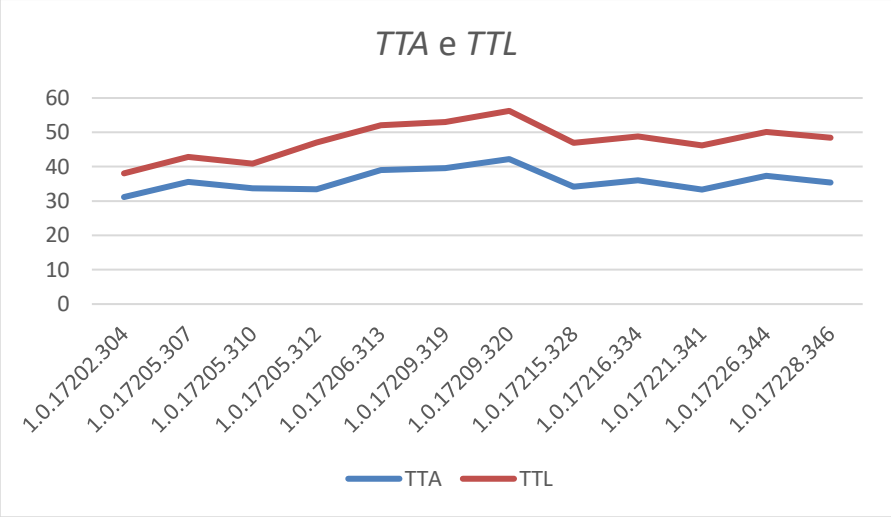


Figura 17 – *TTA* e *TTL* medidos ao longo do projeto

Apesar da tendência destes valores ser para aumentar, na Figura 17 percebe-se que existiu uma redução clara dessa tendência. Com o tempo máximo de 42,21 minutos e tempo médio de 35,91 minutos para *TTA* e tempo máximo de 56,25 minutos e tempo médio de 47,55

minutos para *TTL,* o principal fator que permitiu a redução destas métricas ao longo do tempo e sua estabilização foi a mudança da quantidade de testes executados em cada fase da *pipeline* e do aumento da paralelização dos testes executados.

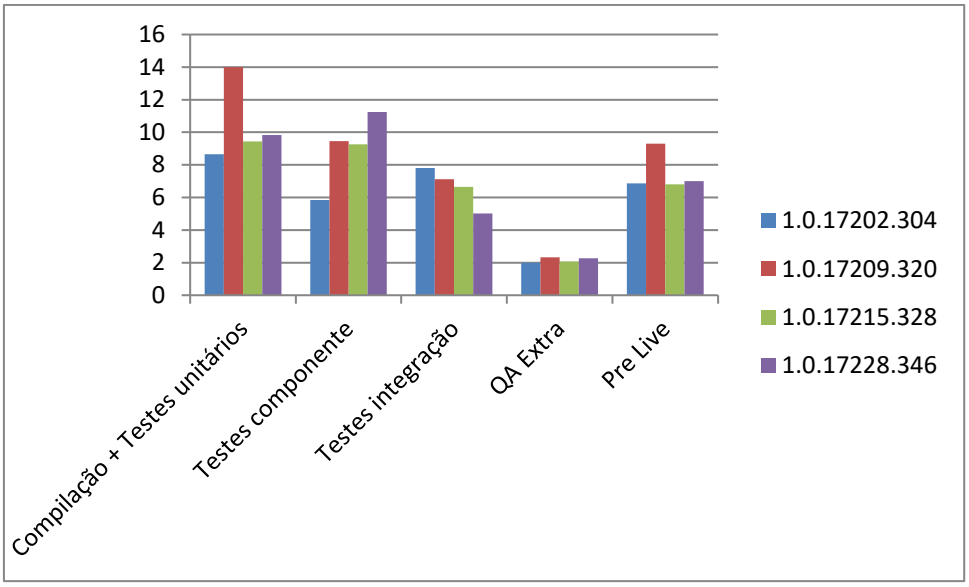


Figura 18 – Tempos de cada fase do *TTA*

Para entender melhor a variação do *TTA* a Figura 18 apresenta informação relevante sobre esse propósito. Analisando e tirando os *outliers* da primeira e última fase, a principal variação apresentada é entre os testes de componente e os testes de integração. Isto deve-se ao facto de inicialmente a equipa produzir código ignorando a integração com as suas dependências e aplicando um perfil 100% *Mock.* Isto fazia com que estes testes fossem divididos pelas duas fazes por ordem de relevância, ou seja, os testes mais importantes seriam executados num primeiro estagio e os restantes no seguinte.

No entanto, com o decorrer do tempo e com a instalação da aplicação nos ambientes de produção, a integração ganhou uma maior importância e todos os testes de componente passaram a ser executados na fase respetiva. Isto levou a um aumento significativo desta fase. Apesar disto e com a introdução dos testes de integração em menor número e respeitando a pirâmide, esta fase reduziu o seu tempo de execução o que permitiu equilibrar os tempos de execução e manter a regularidade desta métrica sem *outliers*.

Tabela 14 – Estudo estatístico *Shapiro-Wilk Test* para as métricas *TTA* e *TTL*

Shapiro-Wilk Test		
	TTA	***TTL***
W	0,95822	0,98172
p-value	0,758148	0,989583

alpha	0,05	0,05
normal	*yes*	*yes*

Para verificar a normalidade dos dados foi aplicado o estudo estatístico *Shapiro-Wilk Test* às amostras recolhidas de *TTA* e *TTL*, estudo esse que está apresentado na Tabela 14, permite retirar conclusões positivas quanto à normalidade dos dados e desta forma perceber se a evolução do desenvolvimento da aplicação é também normal.

7 Conclusões

Neste capitulo é apresentado de forma resumida o trabalho efetuado destacando os objetivos alcançados. São destacados os principais contributos para o que o trabalho fosse possível de realizar. É também nesta secção que será efetuado um balanço final de todo o trabalho realizado e a descrição das principais limitações e futuro trabalho a realizar de modo a melhorar ainda mais a monitorização de testes em *Continuous Delivery* na empresa.

7.1 Objetivos alcançados

No final deste trabalho é possível dizer que os principais objetivos foram alcançados. Existe um guia de *Continuous Delivery* proposto neste documento com os principais conceitos a aplicar nos projetos da empresa. O modelo proposto envolve níveis e tipos de testes a aplicar numa *pipeline* sugerida, capaz de automatizar todo o processo de geração e validação da qualidade de uma nova versão até a sua disponibilização para o utilizador ou cliente final.

Para complementar este guia, foram também introduzidas métricas que ajudam a equipa a monitorizar e a controlar as suas ações e desta forma melhorar os processos aplicados. Outro fator importante destas métricas é fundamentar as suas decisões e explicar o porquê de certas decisões. Desta forma a utilização de novas métricas na empresa foi também um objetivo importante que foi alcançado com este trabalho.

Por fim, outro objetivo importante alcançado foi o valor da métrica *Time to live* desta aplicação em que é possível, para além de gerar e testar de forma totalmente automática, disponibilizar uma nova versão nos ambientes de produção em menos de uma hora. Desta forma o desenvolvimento de *software* é mais rápido e barato e os problemas em produção são rapidamente corrigidos.

7.2 Limitações e Trabalho Futuro

A principal limitação do trabalho que condicionou o número de métricas introduzidas no processo foi o facto de não serem registados todos os pequenos defeitos encontrados, mas sim optar por os corrigir de imediato.

Para a consolidação da solução proposta e verificação dos resultados obtidos uma análise de novos projetos para implementar a solução descrita no documento. Com isto, no futuro o objetivo será reutilizar o processo descrito neste documento ou adapta-lo a novas realidades de outros projetos da empresa e inserir a monitorização em *Continuous Delivery* a todos os projetos.

Relativamente a melhorias ao trabalho elaborado, para além de incentivar o uso de novas métricas por parte das equipas, deverá também existir um trabalho maior junto das equipas

para que se sintam motivadas a utilizar também os melhores conceitos e abordagens de *Continuous Delivery*.

7.3 Apreciação final

No final deste trabalho é possível dizer que a aplicação do modelo preconizado que integra boas práticas de *Continuous Delivery* permitiu alcançar bons resultados e é desde já um ótimo exemplo para todo o trabalho que irá ser desenvolvido na empresa.

A equipa responsável pelo desenvolvimento do projeto teve um contributo essencial que permitiu alcançar estes bons resultados, procurando seguir as indicações recebidas no âmbito deste trabalho.

No final deste trabalho pode-se afirmar que o mesmo teve um contributo cientifico e técnico relevante não só para a empresa, mas como também para a sua área.

8 Referências

[1] B. Fitzgerald e Klaas-JanStol, Continuous software engineering: A roadmap and agenda, Journal of Systems and Software 123, 2017.

[2] L. Chen, *Continuous Delivery: Huge Benefits, but Challenges Too,* IEEE Software , 2015.

[3] M. Fowler, "ContinuousDelivery," 30 Maio 2013. [Online]. Available: https://martinfowler.com/bliki/ContinuousDelivery.html. [Acedido em 04 Fevereiro 2017].

[4] N. Pathania, Pro Continuous Delivery: With Jenkins 2.0, India, 2017.

[5] J. Humble e D. Farley, Continuous Delivery, Boston: Addison-Wesley, 2011.

[6] Atlassian, "Continuous Delivery | Get started with CI/CD - Atlassian," Atlassian, [Online]. Available: https://www.atlassian.com/continuous-delivery.

[7] ThoughtWorks, "Continuous Integration," ThoughtWorks, [Online]. Available: https://www.thoughtworks.com/continuous-integration. [Acedido em 30 January 2017].

[8] L. Parobek, "7 Reasons to Move to Parallel Testing," devops.com, 18 Julho 2016. [Online]. Available: https://devops.com/7-key-reasons-make-move-sequential-parallel-testing/.

[9] I. E. Certification, "What is Unit testing?," [Online]. Available: http://istqbexamcertification.com/what-is-unit-testing/.

[10] I. E. Certification, "What is Component integration testing?," [Online]. Available: http://istqbexamcertification.com/what-is-component-integration-testing/.

[11] I. E. Certification, "What is System integration testing?," [Online]. Available: http://istqbexamcertification.com/what-is-system-integration-testing/.

[12] J. Waletzky, "Smoke Tests vs. BVTs - Crosslake Technologies," Crosslake, 26 April 2012. [Online]. Available: http://www.crosslaketech.com/smoke-vs-bvt/. [Acedido em 18 January 2017].

[13] J. Itkonen, M. V. Mäntylä e C. Lassenius, Test better by exploring: Harnessing human skills and knowledge, IEEE, 2015.

[14] I. E. Certification, “What is Alpha testing?,” [Online]. Available: http://istqbexamcertification.com/what-is-alpha-testing/.

[15] I. E. Certification, “What is Beta testing?,” [Online]. Available: http://istqbexamcertification.com/what-is-beta-testing/.

[16] C. G. U. S. A. a. K. P. S. M. A. Shah, Towards a hybrid testing process unifying exploratory testing andscripted testing, J. Softw. Evol. Process, 2013.

[17] istqbexamcertification, “What are Software Test Types?,” [Online]. Available: http://istqbexamcertification.com/what-are-software-test-types/.

[18] A. K., “Important Software Test Metrics and Measurements,” 14 Dezembro 2016. [Online]. Available: http://www.softwaretestinghelp.com/software-test-metrics-and-measurements/. [Acedido em 15 Janeiro 2017].

[19] Zephyr, “QA Metrics: The value of testing metrics within software development,” [Online]. Available: https://www.getzephyr.com/resources/whitepapers/qa-metrics-value-testing-metrics-within-software-development.

[20] A. A. Vicente, Definição e gerenciamento de métricas de teste no contexto de métodos ágeis, São Paulo: USP São Carlos, 2010.

[21] W. Afzal, Metrics in Software Test Planning and Test Design Processes, Ronneby, Sweden, 2007.

[22] I. S. T. Q. B. (ISTQB), Certified Tester Foundation Level Syllabus, 2011.

[23] A. Page, K. Johnston e B. Rollison, How we test software at Microsoft, Washington: Microsoft Press, 2009.

[24] B. Beyer, C. Jones, J. Petoff e N. R. Murphy, Site Reliability Engineering - How Google Runs Production Systems, USA: O'Reilly Media, Inc., 2016.

[25] J. Whittaker, J. Arbon e J. Carollo, How Google Tests Software, Westford: Addison-

Wesley, 2012.

[26] M. V. Mäntylä, B. Adams, F. Khomh, E. Engström e K. Petersen, On Rapid Releases and Software Testing: A Case Study and a Semi-Systematic Literature Review, 2014.

[27] Z. Hanusz, J. Tarasinska e W. Zielinski, "Shapiro - Wilk Test with known mean," *REVSTAT – Statistical Journal,* vol. 14, 2014.

[28] H.-I. Park, "A Generalization of Wilcoxon Rank Sum Test," *Applied Mathematical Sciences,,* vol. 9, 2015.

[29] G. Macbeth, E. Razumiejczyk e R. Ledesma, "Cliff's Delta Calculator: A non-parametric effect size program for two groups of observations*," *Univ. Psychol.,* vol. 10, 2011.

[30] H. Kniberg e A. Ivarsson, "Scaling Agile @ Spotify with Tribes, Squads, Chapters & Guilds," Spotify, Outubro 2012. [Online]. Available: https://ucvox.files.wordpress.com/2012/11/113617905-scaling-agile-spotify-11.pdf.

[31] R. M. M. A. Irakli Nadareishvili, *Microservice Architecture: Aligning Principles, Practices, and Culture,* Sebastopol: O'Reilly Media, 2016.

[32] Facebook, "Jest - Delightful JavaScript Testing," 2017. [Online]. Available: https://facebook.github.io/jest/.

[33] K. Morris, *Infrastructure as Code: Managing Servers in the Cloud,* Sebastopol: O'Reilly books, 2016.

[34] D. R. Paul, D. R. Niewoehner e D. L. Elder, Engineering Reasoning, 2006.

[35] I. E. Certification, "What are Software Testing Levels?," ISTQB, [Online]. Available: http://istqbexamcertification.com/what-are-software-testing-levels/.

9 Anexos

Anexo A – Métricas recolhidas em Maio

As tabelas 16 a 20 ilustram os dados recolhidos com as métricas no mês de Maio e o respetivo estudo estatístico *Shapiro-Wilk Test* para cada uma das métricas analisadas. Estas tabelas apresentam os dados relativos ao tempo de execução dos testes de integração, ao tempo de execução dos testes unitários, à quantidade de testes de integração, à quantidade de testes unitários e à percentagem de cobertura de código.

Tabela 15 - Tempo de execução dos Testes de integração em Maio

Time/minutes	**Build**
3,36	1.0.17128.60
2,16	1.0.17128.61
4,01	1.0.17128.62
4,49	1.0.17132.73
6,44	1.0.17132.74
6,29	1.0.17132.77
4,04	1.0.17135.78
4,29	1.0.17135.79
4,1	1.0.17135.80
4,66	1.0.17137.88
3,58	1.0.17137.89
4,94	1.0.17138.93
4,45	1.0.17138.94
5,75	1.0.17143.110
5,31	1.0.17145.113

	Time/minutes
Mean	4,524666667
Standard Error	0,29012619
Median	4,45
Mode	#N/D
Standard Deviation	1,123653904
Sample Variance	1,262598095
Kurtosis	0,359759977
Skewness	-0,06153279
Range	4,28
Maximum	6,44
Minimum	2,16
Sum	67,87
Count	15
Geometric Mean	4,381579902
Harmonic Mean	4,219679611
AAD	0,832266667
MAD	0,49
IQR	1,1

Box Plot

	Time/minutes
Min	3,36
Q1-Min	0,665
Med-Q1	0,425
Q3-Med	0,675
Max-Q3	1,315
Mean	4,524666667

Min	3,36
Q1	4,025

Shapiro-Wilk Test

	Time/minutes
W	0,968765577
p-value	0,839370018
alpha	0,05
normal	yes

Median	4,45
Q3	5,125
Max	6,44
Mean	4,524666667

Grand Min	0

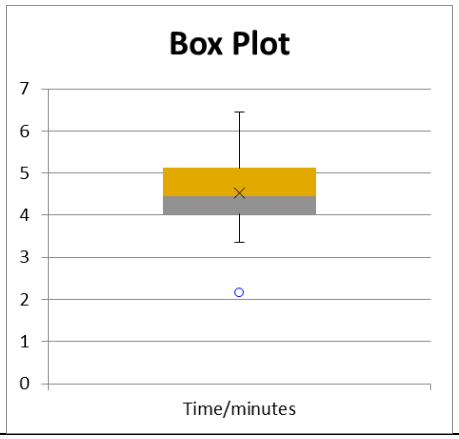


Tabela 16 - Tempo de execução dos testes unitários em Maio

Time/Seconds	Build
6	1.0.17128.60
5	1.0.17128.61
6	1.0.17128.62
6	1.0.17132.73
8	1.0.17132.74
7	1.0.17132.77
11	1.0.17135.78
7	1.0.17135.79
7	1.0.17135.80
13	1.0.17137.88
6	1.0.17137.89
9	1.0.17138.93
8	1.0.17138.94
8	1.0.17143.110
8	1.0.17145.113

	Time/Seconds
Mean	7,666667
Standard Error	0,540429
Median	7
Mode	6
Standard Deviation	2,093072
Sample Variance	4,380952
Kurtosis	2,058165
Skewness	1,374178
Range	8
Maximum	13
Minimum	5
Sum	115
Count	15
Geometric Mean	7,434536
Harmonic Mean	7,231769
AAD	1,511111
MAD	1
IQR	2

Box Plot

	Scores
Min	5
Q1-Min	1
Med-Q1	1
Q3-Med	1
Max-Q3	3
Mean	7,666666667

Min	5
Q1	6

Shapiro-Wilk Test

	Scores
W	0,868770297
p-value	0,032368373
alpha	0,05
normal	no

Median	7
Q3	8
Max	11
Mean	7,666666667
Grand Min	0

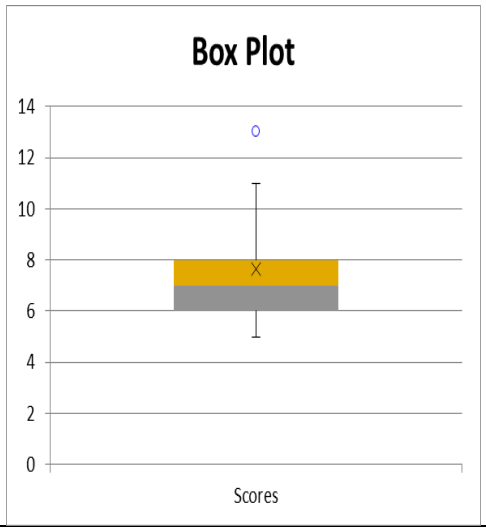


Tabela 17 - Quantidade de testes de integração em Maio

Quantidade	**Build**
98	1.0.17128.60
98	1.0.17128.61
100	1.0.17128.62
106	1.0.17132.73
110	1.0.17132.74
110	1.0.17132.77
112	1.0.17135.78
110	1.0.17135.79
110	1.0.17135.80
116	1.0.17137.88
64	1.0.17137.89
124	1.0.17138.93
124	1.0.17138.94
137	1.0.17143.110
143	1.0.17145.113

	Quantidade
Mean	110,8
Standard Error	4,761152
Median	110
Mode	110
Standard Deviation	18,43986
Sample Variance	340,0286
Kurtosis	2,382693
Skewness	-0,68295
Range	79
Maximum	143
Minimum	64
Sum	1662
Count	15
Geometric Mean	109,179
Harmonic Mean	107,2951
AAD	12,16
MAD	10
IQR	17

Box Plot

	Quantidade
Min	98
Q1-Min	5
Med-Q1	7
Q3-Med	10
Max-Q3	23

Shapiro-Wilk Test

	Quantidade
W	0,917864341
p-value	0,178698763
alpha	0,05
normal	yes

Mean	110,8
Min	98
Q1	103
Median	110
Q3	120
Max	143
Mean	110,8
Grand Min	0

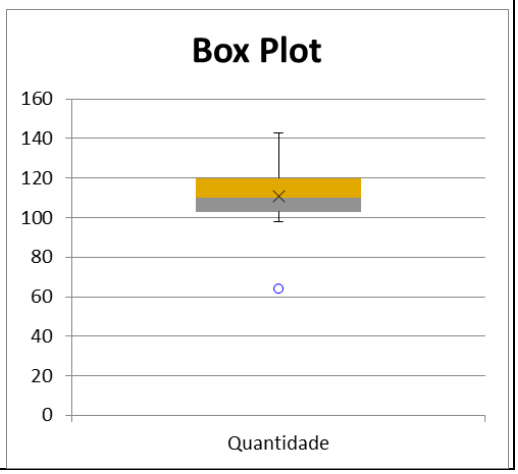


Tabela 18 - Quantidade de testes unitários em Maio

Quantidade	Build
287	1.0.17128.60
283	1.0.17128.61
290	1.0.17128.62
296	1.0.17132.73
331	1.0.17132.74
311	1.0.17132.77
311	1.0.17135.78
329	1.0.17135.79
329	1.0.17135.80
358	1.0.17137.88
358	1.0.17137.89
391	1.0.17138.93
391	1.0.17138.94
412	1.0.17143.110
412	1.0.17145.113

	Quantidade
Mean	339,2667
Standard Error	11,69227
Median	329
Mode	311
Standard Deviation	45,28397
Sample Variance	2050,638
Kurtosis	-1,18921
Skewness	0,435075
Range	129
Maximum	412
Minimum	283
Sum	5089
Count	15
Geometric Mean	336,5122
Harmonic Mean	333,8405
AAD	38,18667
MAD	33
IQR	71

Box Plot

	Quantidade
Min	283
Q1-Min	20,5
Med-Q1	25,5
Q3-Med	45,5
Max-Q3	37,5
Mean	339,2666667

Shapiro-Wilk Test

	Quantidade
W	0,907236871
p-value	0,122824166
alpha	0,05
normal	yes

Min	283
Q1	303,5
Median	329
Q3	374,5
Max	412
Mean	339,2666667
Grand Min	0

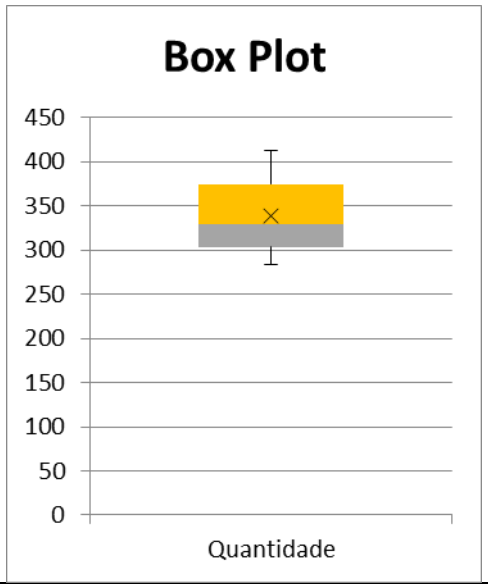
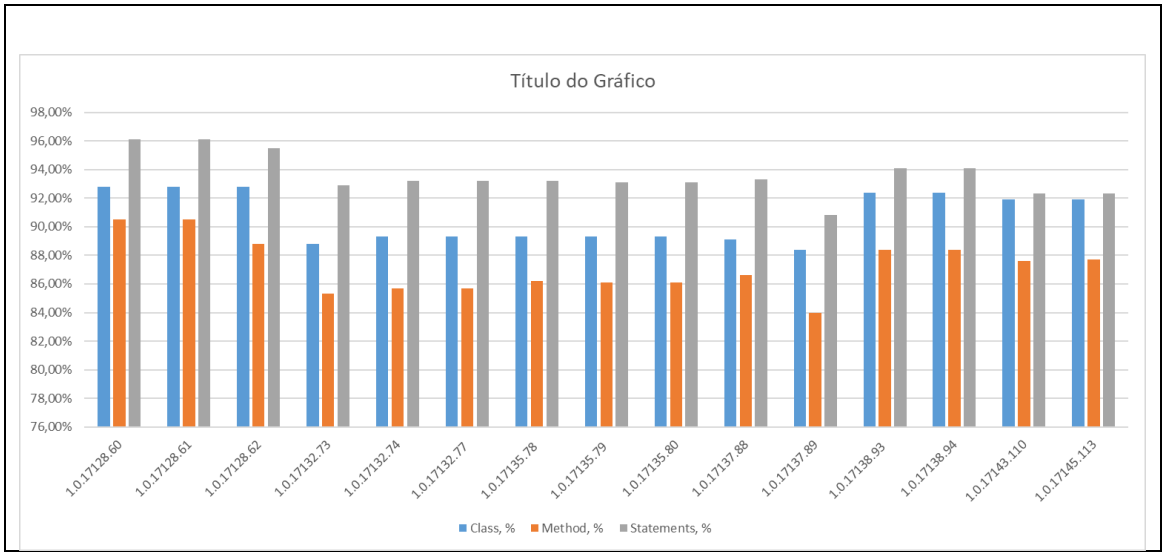


Tabela 19 - Cobertura de código em Maio

BuildVersion	Class, %	Method, %	Statements, %	Nº Unit Tests
1.0.17128.60	92,80%	90,50%	96,10%	287
1.0.17128.61	92,80%	90,50%	96,10%	283
1.0.17128.62	92,80%	88,80%	95,50%	290
1.0.17132.73	88,80%	85,30%	93%	296
1.0.17132.74	89,30%	85,70%	93,20%	331
1.0.17132.77	89,30%	85,70%	93,20%	311
1.0.17135.78	89,30%	86,20%	93,20%	311
1.0.17135.79	89,30%	86,10%	93,10%	329
1.0.17135.80	89,30%	86,10%	93,10%	329
1.0.17137.88	89,10%	86,60%	93,30%	358
1.0.17137.89	88,40%	84,00%	90,80%	358
1.0.17138.93	92,40%	88,40%	94,10%	391
1.0.17138.94	92,40%	88,40%	94,10%	391
1.0.17143.110	91,90%	87,60%	92%	412
1.0.17145.113	91,90%	87,70%	92%	412


Título do Gráfico
98,00%
96,00%
94,00%
92,00%
90,00%
88,00%
86,00%
84,00%
82,00%
80,00%
78,00%
76,00%
1.0.17128.60
1.0.17128.61
1.0.17128.62
1.0.17132.73
1.0.17132.74
1.0.17132.77
1.0.17135.78
1.0.17135.79
1.0.17135.80
1.0.17137.88
1.0.17137.89
1.0.17138.93
1.0.17138.94
1.0.17143.110
1.0.17145.113
Class, %
Method, %
Statements, %

Anexo B – Métricas recolhidas em Julho

Neste anexo B, as tabelas 21 a 25 apresentam as mesmas métricas referidas e descritas no anexo A, no entanto, com dados referentes ao mês de Julho. Esta recolha de informação foi importante para que uma comparação de dados ao longo do tempo fosse possível para uma posterior analise da progressão do trabalho.

Tabela 20 - Tempo de execução dos Testes de integração em Julho

Time/minutes	**Build**
4	1.0.17187.241
4,02	1.0.17187.242
4,03	1.0.17187.243
3,88	1.0.17187.244
3,97	1.0.17187.245
3,95	1.0.17188.246
4,24	1.0.17188.248
4,01	1.0.17193.261
4,28	1.0.17194.266
4,53	1.0.17194.267
4,09	1.0.17194.268
5,25	1.0.17195.271
4,9	1.0.17195.272
5,72	1.0.17195.276
6,17	1.0.17195.277

	Time/minutes
Mean	4,469333
Standard Error	0,184905
Median	4,09
Mode	#N/D
Standard Deviation	0,716135
Sample Variance	0,51285
Kurtosis	1,106195
Skewness	1,461425
Range	2,29
Maximum	6,17
Minimum	3,88
Sum	67,04
Count	15
Geometric Mean	4,421553
Harmonic Mean	4,379232
AAD	0,563111
MAD	0,15
IQR	0,71

Box Plot

	Time/minutes
Min	3,88
Q1-Min	0,125
Med-Q1	0,085
Q3-Med	0,625
Max-Q3	1,005
Mean	4,469333333
Min	3,88
Q1	4,005
Median	4,09
Q3	4,715
Max	5,72

Shapiro-Wilk Test

	Time/minutes
W	0,772612723
p-value	0,001671714
alpha	0,05
normal	no

Mean	4,469333333
Grand Min	0

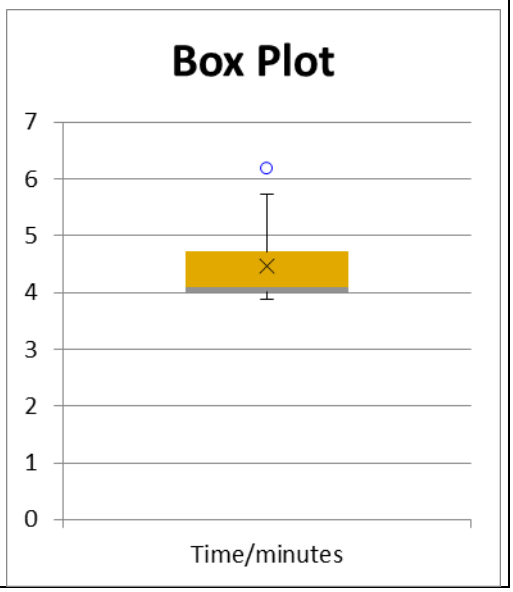


Tabela 21 - Tempo de execução dos testes unitários em Julho

Time/Seconds	Build
19	1.0.17187.241
19	1.0.17187.242
22	1.0.17187.243
17	1.0.17187.244
19	1.0.17187.245
17	1.0.17188.246
16	1.0.17188.248
18	1.0.17193.261
16	1.0.17194.266
17	1.0.17194.267
17	1.0.17194.268
17	1.0.17195.271
18	1.0.17195.272
22	1.0.17195.276
22	1.0.17195.277

	Time/Seconds
Mean	18,4
Standard Error	0,541603
Median	18
Mode	17
Standard Deviation	2,097618
Sample Variance	4,4
Kurtosis	-0,42094
Skewness	0,88512
Range	6
Maximum	22
Minimum	16
Sum	276
Count	15
Geometric Mean	18,2938
Harmonic Mean	18,19313
AAD	1,68
MAD	1
IQR	2

Box Plot

	Scores
Min	16
Q1-Min	1
Med-Q1	1
Q3-Med	1

Shapiro-Wilk Test

	Scores
W	0,838441454
p-value	0,011963921
alpha	0,05
normal	no

Max-Q3	3
Mean	18,4
Min	16
Q1	17
Median	18
Q3	19
Max	22
Mean	18,4
Grand Min	0

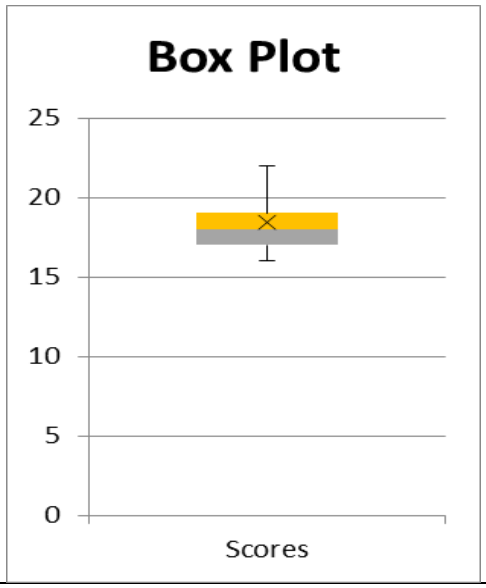


Tabela 22 - Quantidade de testes de integração em Julho

Quantidade	**Build**
207	1.0.17187.241
207	1.0.17187.242
207	1.0.17187.243
209	1.0.17187.244
209	1.0.17187.245
209	1.0.17188.246
209	1.0.17188.248
209	1.0.17193.261
209	1.0.17194.266
209	1.0.17194.267
207	1.0.17194.268
207	1.0.17195.271
207	1.0.17195.272
207	1.0.17195.276
208	1.0.17195.277

	Quantidade
Mean	208
Standard Error	0,258199
Median	208
Mode	207
Standard Deviation	1
Sample Variance	1
Kurtosis	-2,23077
Skewness	0
Range	2
Maximum	209
Minimum	207
Sum	3120
Count	15
Geometric Mean	207,9978
Harmonic Mean	207,9955
AAD	0,933333
MAD	1
IQR	2

Box Plot

	Quantidade
Min	207
Q1-Min	0
Med-Q1	1
Q3-Med	1

Shapiro-Wilk Test

	Quantidade
W	0,686301478
p-value	0,00017804
alpha	0,05
normal	no

Max-Q3	0
Mean	208
Min	207
Q1	207
Median	208
Q3	209
Max	209
Mean	208
Grand Min	0

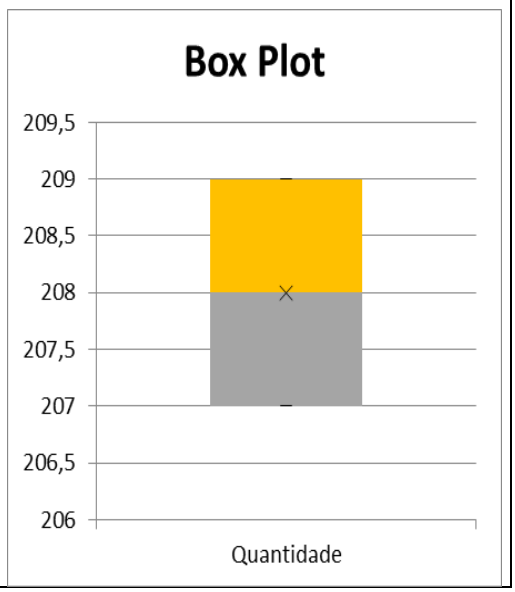


Tabela 23 - Quantidade de testes unitários em Julho

Quantidade	**Build**
729	1.0.17187.241
729	1.0.17187.242
732	1.0.17187.243
750	1.0.17187.244
750	1.0.17187.245
750	1.0.17188.246
748	1.0.17188.248
748	1.0.17193.261
748	1.0.17194.266
751	1.0.17194.267
751	1.0.17194.268
751	1.0.17195.271
751	1.0.17195.272
773	1.0.17195.276
773	1.0.17195.277

	Quantidade
Mean	748,9333
Standard Error	3,282662
Median	750
Mode	751
Standard Deviation	12,7137
Sample Variance	161,6381
Kurtosis	0,727036
Skewness	0,289546
Range	44
Maximum	773
Minimum	729
Sum	11234
Count	15
Geometric Mean	748,8329
Harmonic Mean	748,7326
AAD	7,946667
MAD	2
IQR	3

Box Plot

	Quantidade
Min	748
Q1-Min	0
Med-Q1	2
Q3-Med	1

Shapiro-Wilk Test

	Quantidade
W	0,824085378
p-value	0,007619358
alpha	0,05
normal	no

Max-Q3	0
Mean	748,9333333
Min	748
Q1	748
Median	750
Q3	751
Max	751
Mean	748,9333333
Grand Min	0

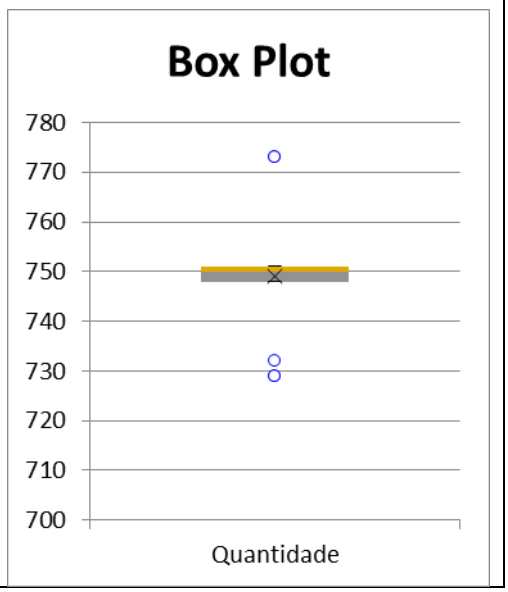
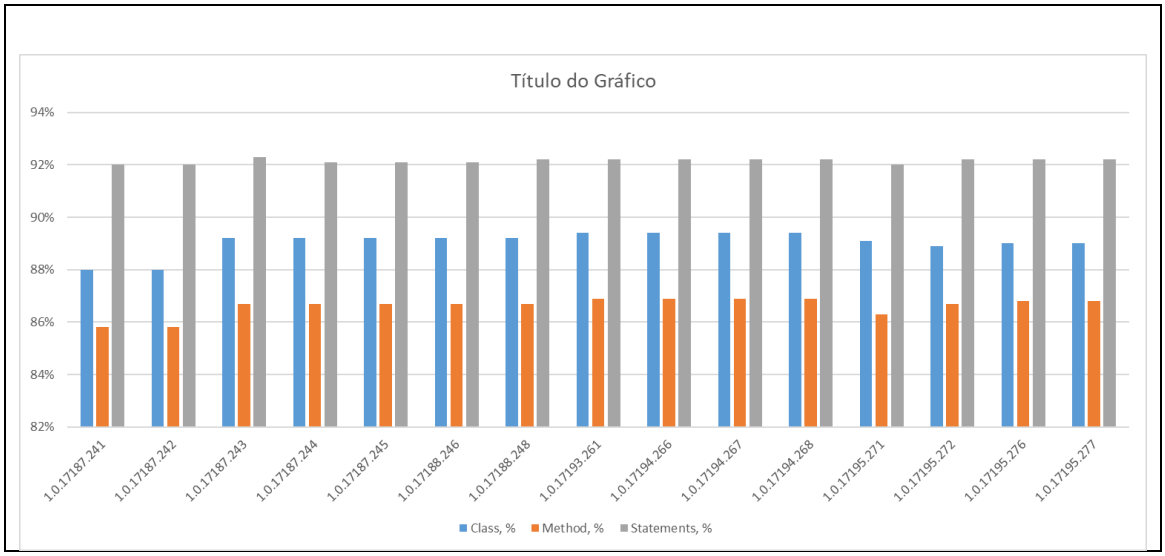


Tabela 24 - Cobertura de código em Julho

BuildVersion	Class, %	Method, %	Statements, %	Nº Unit Tests
1.0.17187.241	88%	85,80%	92%	729
1.0.17187.242	88%	85,80%	92%	729
1.0.17187.243	89,20%	86,70%	92,30%	732
1.0.17187.244	89,20%	86,70%	92,10%	750
1.0.17187.245	89,20%	86,70%	92,10%	750
1.0.17188.246	89,20%	86,70%	92,10%	750
1.0.17188.248	89,20%	86,70%	92,20%	748
1.0.17193.261	89,40%	86,90%	92,20%	748
1.0.17194.266	89,40%	86,90%	92,20%	748
1.0.17194.267	89,40%	86,90%	92,20%	751
1.0.17194.268	89,40%	86,90%	92,20%	751
1.0.17195.271	89,10%	86,30%	92%	751
1.0.17195.272	88,90%	86,70%	92,20%	751
1.0.17195.276	89%	86,80%	92,20%	773
1.0.17195.277	89%	86,80%	92,20%	773


Título do Gráfico
94%
92%
90%
88%
86%
84%
82%
1.0.17187.241
1.0.17187.242
1.0.17187.243
1.0.17187.244
1.0.17187.245
1.0.17188.246
1.0.17188.248
1.0.17193.261
1.0.17194.266
1.0.17194.267
1.0.17194.268
1.0.17195.271
1.0.17195.272
1.0.17195.276
1.0.17195.277
Class, %
Method, %
Statements, %

Anexo C – Valores das métricas *Time to approve e time to live*

As tabelas 26 a 28 apresentam, para além do estudo estatístico que verifica a normalidade dos dados recolhidos, as métricas *TTA* e *TTL* bem como as suas variações. Estes valores são relevantes para a análise e o impacto das ações tomadas ao longo do trabalho.

As tabelas 29 a 34 apresentam os dados recolhidos para cada um dos estágios da pipeline aplicada na solução.

Tabela 25 – Valores do *Time to approve*

BuildVersion	TTA
1.0.17202.304	31,16
1.0.17205.307	35,61
1.0.17205.310	33,7
1.0.17205.312	33,45
1.0.17206.313	39,06
1.0.17209.319	39,57
1.0.17209.320	42,21
1.0.17215.328	34,22
1.0.17216.334	36,03
1.0.17221.341	33,3
1.0.17226.344	37,32
1.0.17228.346	35,34

	TTA
Mean	35,91417
Standard Error	0,907008
Median	35,475
Mode	#N/D
Standard Deviation	3,141969
Sample Variance	9,871972
Kurtosis	-0,02599
Skewness	0,619383
Range	11,05
Maximum	42,21
Minimum	31,16
Sum	430,97
Count	12
Geometric Mean	35,79114
Harmonic Mean	35,67108
AAD	2,436528
MAD	1,935
IQR	4,1175

Box Plot

	TTA
Min	31,16
Q1-Min	2,4775
Med-Q1	1,8375
Q3-Med	2,28
Max-Q3	4,455
Mean	35,91417

Min	31,16

Shapiro-Wilk Test

	TTA
W	0,95822
p-value	0,758148
alpha	0,05
normal	yes

Q1	33,6375
Median	35,475
Q3	37,755
Max	42,21
Mean	35,91417

Grand Min	0

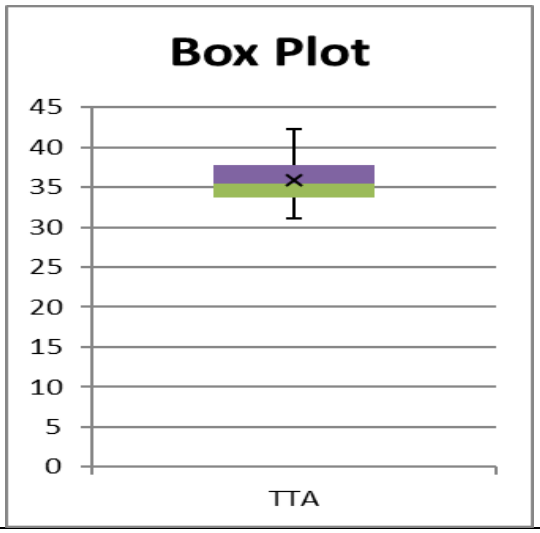


Tabela 26 – Valores to *Time to Live*

BuildVersion	TTL
1.0.17202.304	38,06
1.0.17205.307	42,81
1.0.17205.310	40,85
1.0.17205.312	47,01
1.0.17206.313	52,08
1.0.17209.319	52,99
1.0.17209.320	56,25
1.0.17215.328	46,96
1.0.17216.334	48,8
1.0.17221.341	46,17
1.0.17226.344	50,11
1.0.17228.346	48,46

	TTL
Mean	47,54583
Standard Error	1,494207
Median	47,735
Mode	#N/D
Standard Deviation	5,176084
Sample Variance	26,79184
Kurtosis	-0,14752
Skewness	-0,25905
Range	18,19
Maximum	56,25
Minimum	38,06
Sum	570,55
Count	12
Geometric Mean	47,28063
Harmonic Mean	47,0085
AAD	3,9025
MAD	3,36
IQR	5,2725

Box Plot

	TTL
Min	38,06
Q1-Min	7,27

Shapiro-Wilk Test

	TTL
W	0,98172
p-value	0,989583

Med-Q1	2,405
Q3-Med	2,8675
Max-Q3	5,6475
Mean	47,54583

Min	38,06
Q1	45,33
Median	47,735
Q3	50,6025
Max	56,25
Mean	47,54583

Grand Min	0

alpha	0,05
normal	yes

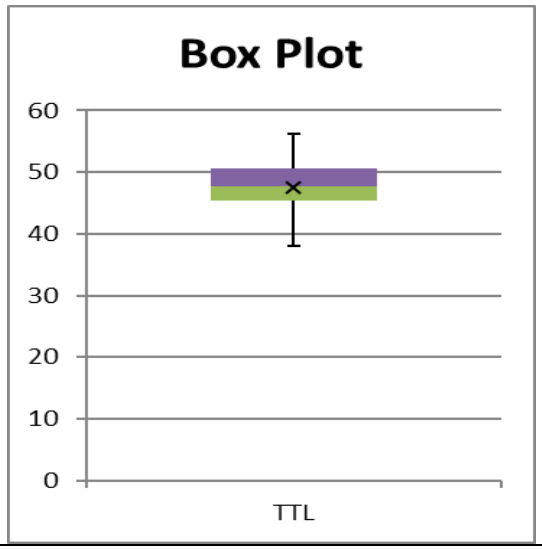
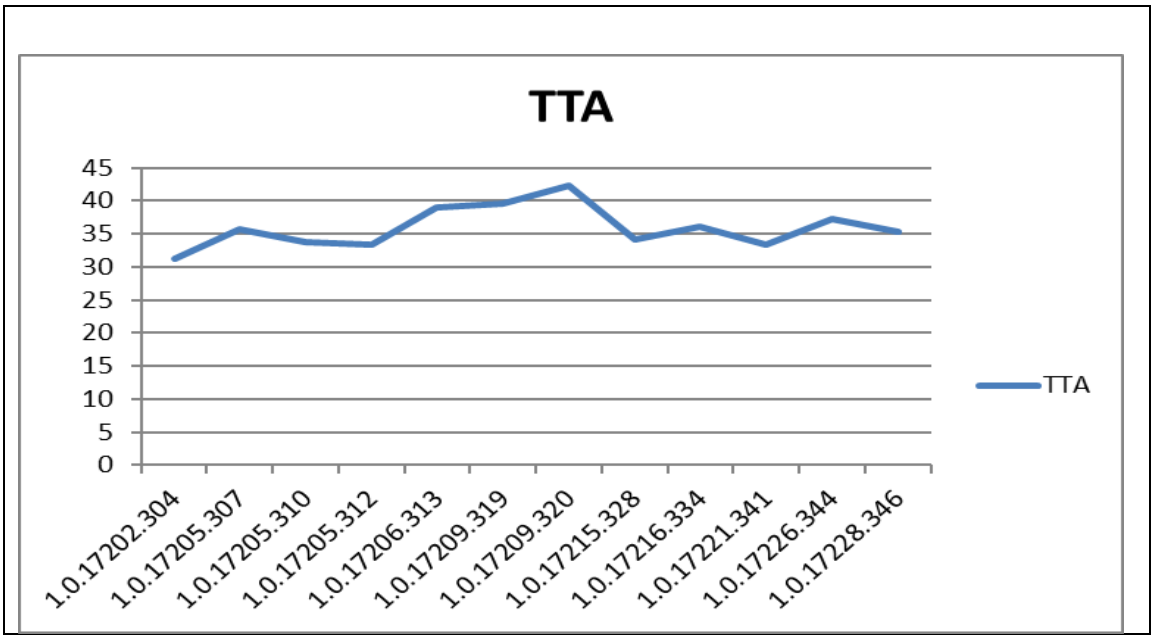


Tabela 27 - Variação do Time to aprove e Time to live



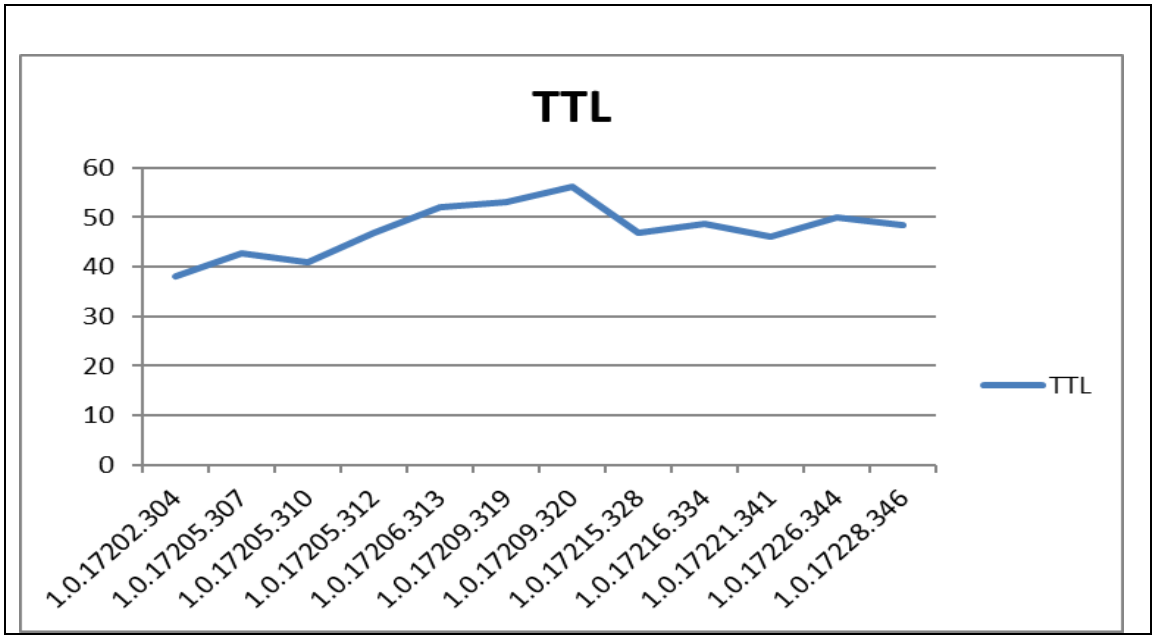

TTL
60
50
40
30
20
10
0
1.0.17202.304
1.0.17205.307
1.0.17205.310
1.0.17205.312
1.0.17206.313
1.0.17209.319
1.0.17209.320
1.0.17215.328
1.0.17216.334
1.0.17221.341
1.0.17226.344
1.0.17228.346
TTL

Tabela 28 - Tempo de estágio de compilação e testes unitários

Versão	Compilação + Testes unitários
1.0.17202.304	8,65
1.0.17205.307	10
1.0.17205.310	9,23
1.0.17205.312	9,15
1.0.17206.313	8,98
1.0.17209.319	10,32
1.0.17209.320	14
1.0.17215.328	9,43
1.0.17216.334	9,57
1.0.17221.341	9,7
1.0.17226.344	10
1.0.17228.346	9,83

Minutos

	Compilação + Testes unitários
Mean	9,905
Standard Error	0,396968627
Median	9,635
Mode	10
Standard Deviation	1,375139662
Sample Variance	1,891009091
Kurtosis	8,52722543
Skewness	2,730313206
Range	5,35
Maximum	14
Minimum	8,65
Sum	118,86
Count	12
Geometric Mean	9,831564994
Harmonic Mean	9,769717846
AAD	0,783333333
MAD	0,385
IQR	0,79

Box Plot

	Compilação + Testes unitários
Min	8,65
Q1-Min	0,56
Med-Q1	0,425
Q3-Med	0,365
Max-Q3	0,32
Mean	9,905

Min	8,65
Q1	9,21
Median	9,635
Q3	10
Max	10,32
Mean	9,905

Grand Min	0

Shapiro-Wilk Test

	Compilação + Testes unitários
W	0,671373557
p-value	0,000452815
alpha	0,05
normal	no

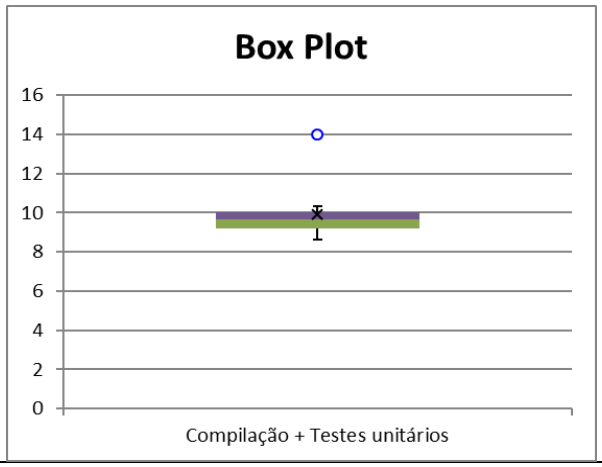


Tabela 29 - Tempo de estágio de testes ao componente

Versão	Testes componente
1.0.17202.304	5,85
1.0.17205.307	9,15
1.0.17205.310	9,15
1.0.17205.312	6,09
1.0.17206.313	10,22
1.0.17209.319	9,53
1.0.17209.320	9,46
1.0.17215.328	9,26
1.0.17216.334	9,11
1.0.17221.341	11,62
1.0.17226.344	14,05
1.0.17228.346	11,24

Minutos

	Testes componente
Mean	9,560833333
Standard Error	0,640231325
Median	9,36
Mode	9,15
Standard Deviation	2,217826366
Sample Variance	4,918753788
Kurtosis	0,962740704
Skewness	0,110509503
Range	8,2
Maximum	14,05
Minimum	5,85
Sum	114,73
Count	12
Geometric Mean	9,312558928
Harmonic Mean	9,048599148
AAD	1,481111111
MAD	0,555
IQR	1,335

Box Plot

	Testes componente
Min	9,11
Q1-Min	0,03
Med-Q1	0,22
Q3-Med	1,115
Max-Q3	1,145

Shapiro-Wilk Test

	Testes componente
W	0,915706692
p-value	0,252315178
alpha	0,05
normal	yes

Mean	9,560833333

Min	9,11
Q1	9,14
Median	9,36
Q3	10,475
Max	11,62
Mean	9,560833333

Grand Min	0

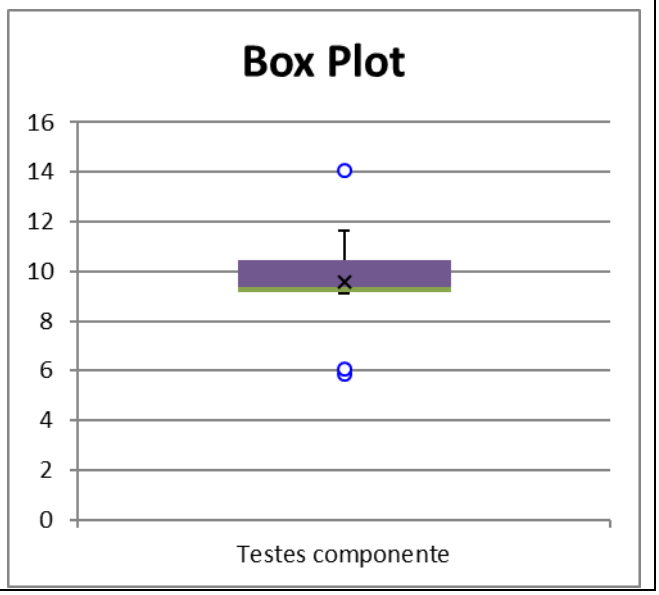


Tabela 30 - Tempo de estágio de testes de integração

Versão	Testes integração
1.0.17202.304	7,8
1.0.17205.307	7,7
1.0.17205.310	7,32
1.0.17205.312	7,52
1.0.17206.313	8,87
1.0.17209.319	8,39
1.0.17209.320	7,12
1.0.17215.328	6,65
1.0.17216.334	8,05
1.0.17221.341	4,95
1.0.17226.344	6,53
1.0.17228.346	5,01

Minutos

	Testes integração
Mean	7,159166667
Standard Error	0,3508722
Median	7,42
Mode	#N/D
Standard Deviation	1,215456954
Sample Variance	1,477335606
Kurtosis	0,092152557
Skewness	-0,792099245
Range	3,92
Maximum	8,87
Minimum	4,95
Sum	85,91
Count	12
Geometric Mean	7,054308602
Harmonic Mean	6,938804963
AAD	0,922638889
MAD	0,7
IQR	1,2425

Box Plot

	Testes integração
Min	4,95

Shapiro-Wilk Test

	Testes integração
W	0,921100506

Q1-Min	1,67
Med-Q1	0,8
Q3-Med	0,4425
Max-Q3	1,0075
Mean	7,159166667

Min	4,95
Q1	6,62
Median	7,42
Q3	7,8625
Max	8,87
Mean	7,159166667

Grand Min	0

p-value	0,29515878
alpha	0,05
normal	yes

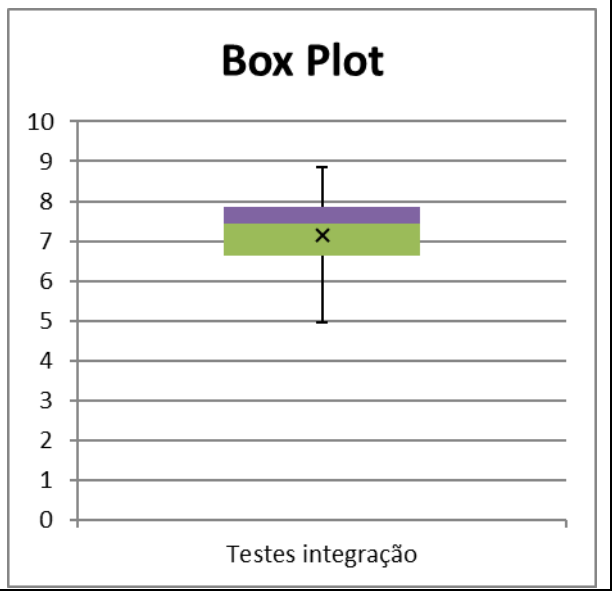


Tabela 31 - Tempo de estágio de QA extra

Versão	QA Extra
1.0.17202.304	1,99
1.0.17205.307	1,98
1.0.17205.310	2,24
1.0.17205.312	2,06
1.0.17206.313	2,12
1.0.17209.319	2,06
1.0.17209.320	2,33
1.0.17215.328	2,08
1.0.17216.334	2,08
1.0.17221.341	0
1.0.17226.344	0
1.0.17228.346	2,26

Minutos

	QA Extra
Mean	1,766666667
Standard Error	0,240196679
Median	2,07
Mode	2,06
Standard Deviation	0,832065703
Sample Variance	0,692333333
Kurtosis	2,458402331
	-
Skewness	1,978194164
Range	2,33
Maximum	2,33
Minimum	0
Sum	21,2
Count	12
Geometric Mean	#NÚM!
Harmonic Mean	#NÚM!
AAD	0,588888889
MAD	0,085
IQR	0,1625

Box Plot	Shapiro-Wilk Test

	QA Extra
Min	1,98
Q1-Min	0,0075
Med-Q1	0,0825
Q3-Med	0,08
Max-Q3	0,18
Mean	1,766666667

Min	1,98
Q1	1,9875
Median	2,07
Q3	2,15
Max	2,33
Mean	1,766666667

Grand Min	0

	QA Extra
W	0,58528195
p-value	8,10788E-05
alpha	0,05
normal	no

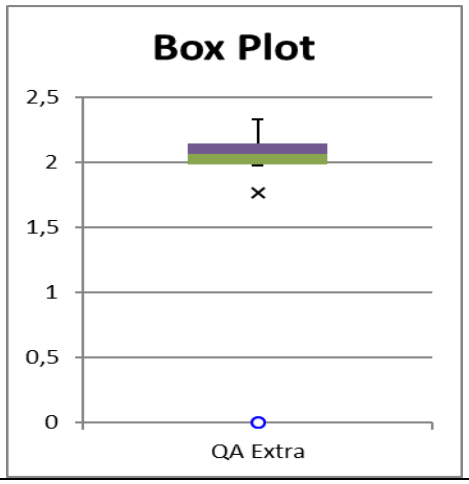


Tabela 32 - Tempo de estágio de Pré Live

Versão	Pre Live
1.0.17202.304	6,87
1.0.17205.307	6,78
1.0.17205.310	5,76
1.0.17205.312	8,63
1.0.17206.313	8,87
1.0.17209.319	9,27
1.0.17209.320	9,3
1.0.17215.328	6,8
1.0.17216.334	7,22
1.0.17221.341	7,03
1.0.17226.344	6,74
1.0.17228.346	7

Minutos

	Pre Live
Mean	7,5225
Standard Error	0,338002432
Median	7,015
Mode	#N/D
Standard Deviation	1,17087477
Sample Variance	1,370947727
Kurtosis	-1,096155003
Skewness	0,506262149
Range	3,54
Maximum	9,3
Minimum	5,76
Sum	90,27
Count	12
Geometric Mean	7,441413686
Harmonic Mean	7,363246553
AAD	0,996666667
MAD	0,255

		IQR	1,895

Box Plot

	Pre Live
Min	5,76
Q1-Min	1,035
Med-Q1	0,22
Q3-Med	1,675
Max-Q3	0,61
Mean	7,5225

Min	5,76
Q1	6,795
Median	7,015
Q3	8,69
Max	9,3
Mean	7,5225

Grand Min	0

Shapiro-Wilk Test

	Pre Live
W	0,856289605
p-value	0,043937746
alpha	0,05
normal	no

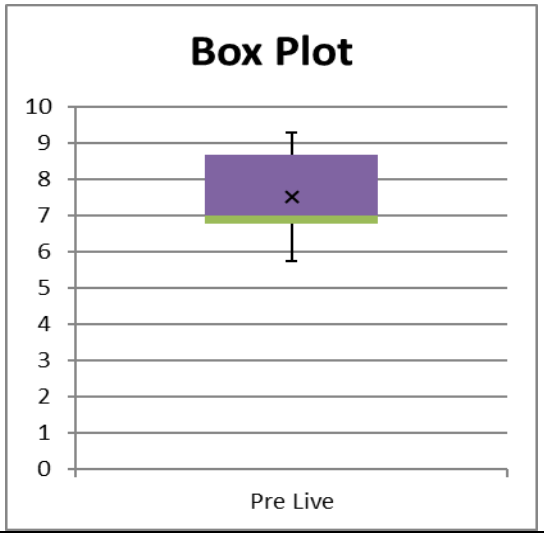


Tabela 33 - Tempo do estágio de Live

Versão	Live
1.0.17202.304	6,9
1.0.17205.307	7,2
1.0.17205.310	7,15
1.0.17205.312	13,56
1.0.17206.313	13,02
1.0.17209.319	13,42
1.0.17209.320	14,04
1.0.17215.328	12,74
1.0.17216.334	12,77
1.0.17221.341	12,87
1.0.17226.344	12,79
1.0.17228.346	13,12

Minutos

	Live
Mean	11,63166667
Standard Error	0,79949021
Median	12,83
Mode	#N/D
Standard Deviation	2,769515328
Sample Variance	7,670215152
Kurtosis	-0,379877742
Skewness	-1,252364158
Range	7,14
Maximum	14,04
Minimum	6,9
Sum	139,58
Count	12
Geometric Mean	11,25948253

		Harmonic Mean	10,82177242
		AAD	2,274166667
		MAD	0,44
		IQR	1,84
Box Plot		Shapiro-Wilk Test	
	Live		*Live*
Min	12,74	W	0,681931294
Q1-Min	-1,385	p-value	0,000567585
Med-Q1	1,475	alpha	0,05
Q3-Med	0,365	normal	no
Max-Q3	0,845		
Mean	11,63166667		
Min	12,74		
Q1	11,355		
Median	12,83		
Q3	13,195		
Max	14,04		
Mean	11,63166667		
Grand Min	0