UNIVERSIDADE D
COIMBRA

Maria Manuela Boto Abrantes

QUALITY ASSURANCE AUTOMATION

Dissertação no âmbito do Mestrado em Engenharia Informática, especialização em Comunicações, Serviços e Infraestruturas, orientada pelo Professor Doutor Paulo de Carvalho e pelo Engenheiro Bernardo Marques, apresentada à Faculdade de Ciências e Tecnologia / Departamento de Engenharia Informática.

setembro de 2020

Esta página foi propositadamente deixada em branco.

Para ti, Avô.

Esta página foi propositadamente deixada em branco.

Agradecimentos

Aos meus pais, agradeço por todo o apoio e por possibilitarem a minha entrada na universidade.

Ao meu orientador, Professor Doutor Paulo de Carvalho, pelo apoio, disponibilidade e pela tranquilidade transmitidas.

Ao meu orientador da empresa, Engenheiro Bernardo Marques, pelo o conhecimento transmitido, incansável cooperação e orientação.

A Micas, independentemente da distância sempre presente, por todo o incentivo e motivação e pelas palavras certas no momento certo.

Ao meu parceiro de redes, pela calma infindável que sempre teve nas horas de trabalho, pelo apoio e por todos os momentos de diversão.

Ao meu primo, por todos os momentos de descontração e por me confiar o papel de madrinha.

A Cláudia, pelas incansáveis correções e sugestões de melhorias neste documento, pela paciência e companhia.

A Piedade, por todos os ensinamentos, compreensão e desabafos.

A todos os que tiveram presentes ao longo destes anos, o meu sincero agradecimento.

Esta página foi propositadamente deixada em branco.

Abstract

Quality assurance is a set of activities that ensures that software engineering processes are monitored and meet defined standards, which, ultimately, generates more confidence in product quality. With the growing need to produce software in a systematic and consistent manner, and to promote constant customer satisfaction and confidence in the product, ensuring its quality is essential. Ensuring the quality of the software is not an option, but rather the key factor for the success of a software and the company.

The quality assurance automation allows bugs to be detected in advance, thus speeding up the delivery process, which allows for a faster product launch. Automating quality assurance allows you to minimize costs and optimize time, turning tasks that were once manual and repetitive into automated tasks.

Thus, the main objective of this project is to create mechanisms for testing automation and software validation, ensuring that there are no setbacks in it. In a first phase, an investigation was carried out in software validation and mechanisms that allowed the integration of these same validations in the development cycle. Tools that best fit the work in question were analyzed. Subsequently, two battery of tests were planned and automated: one of performance to an event streaming platform with service-based architecture and the other of end-to-end tests to a user interface available through the use of a web browser. After implementation, the test batteries were performed and validated in order to verify their feasibility and usability.

The work developed made it possible to provide feedback to the development team regarding the impact that changes made to the code can have on the performance of the application programming interface and on the functional level of the user interface.

Keywords

Testing, Automation, Quality Assurance, Performance Testing, End-to-end Testing

Esta página foi propositadamente deixada em branco.

Resumo

Garantia de qualidade é um conjunto de atividades que garante que os processos de engenharia de software sejam monitorizados e atendam aos padrões definidos, o que, em última instância, gera mais confiança na qualidade do produto. Com a crescente necessidade de produzir software de forma sistemática e consistente, e de promover a constante satisfação dos clientes e confiança no produto, garantir a qualidade do mesmo é essencial. Garantir a qualidade do software não é uma opção, mas sim o fator chave para o sucesso de um software e da empresa.

A automação de garantia de qualidade permite que os bugs sejam detetados antecipadamente, agilizando, assim, o processo de entrega, o que permite um lançamento mais rápido do produto. A automação da garantia de qualidade permite minimizar custos e otimizar o tempo, tornando tarefas outrora manuais e repetitivas em tarefas automatizadas.

Desta forma, o principal objetivo deste projeto é a criação de mecanismos de automação de testes e validação de software, assegurando a inexistência de retrocessos no mesmo. Numa primeira fase foi realizada uma investigação em validação de software e mecanismos que permitissem a integração dessas mesmas validações no ciclo de desenvolvimento. Foram analisadas ferramentas que melhor se adequassem ao caso em questão. Posteriormente foram planeadas e automatizadas duas bateria de testes: uma de desempenho a uma plataforma de *streaming* de eventos com arquitetura baseada em serviços e outra de testes *end-to-end* a uma interface de utilizador disponível através da utilização de um *web browser*. Após a implementação, as baterias de testes foram executadas e validadas de modo a verificar a viabilidade e usabilidade destas.

O trabalho desenvolvido tornou possível dar feedback à equipa de desenvolvimento no que diz respeito ao impacto que as alterações efetuadas no código podem ter no desempenho da interface de programação da aplicação e ao nível funcional da interface de utilizador.

Palavras-Chave

Testes, Automação, Garantia de Qualidade, Testes de Desempenho, Testes *end-to-end*

Esta página foi propositadamente deixada em branco.

Índice

Esta página foi propositadamente deixada em branco.

Acrónimos

API *Application Programming Interface*. xix, xxi, 2, 4, 5, 15, 25, 29, 33, 35, 38, 39, 49, 52–54, 56, 59–66, 70, 71, 73, 80, 81, 83–86, 90, 111

ATF *Automated Tests Framework*. 25, 26, 64, 70, 72, 84

AWS *Amazon Web Services*. 5, 63, 84–86

BVA *Boundary Value Analysis*. 9

CD *Continuous Delivery*. 20, 45–47, 49, 68, 89

CD* *Continuous Deployment*. 20

CI *Continuous Integration*. 19, 20, 38, 45, 46, 49, 51, 55, 68, 89

CLI *Command Line Interface*. 50–52

CPU *Central Processing Unit*. 5, 13, 15, 35, 47, 48, 50, 62, 64, 71, 80, 83–85

CSV *Comma Separated Values*. 50

CVDS Ciclo de Vida de Desenvolvimento de Software. 2, 4, 6, 8, 11, 29, 35, 49, 62, 64, 81

DEI Departamento de Engenharia Informática. 1, 5

DSL *Domain-specific language*. 51

DTC *Data Trouble Codes*. 29, 61

FCTUC Faculdade de Ciências e Tecnologia da Universidade de Coimbra. 1

FIFO *First In First Out*. 38, 52, 53, 64

FTP *File Transfer Protocol*. 50

GNSS *Global Navigation Satellite System*. 61

GPS *Global Positioning System*. 60

GUI *Graphical User Interface*. 50, 51

HTML *Hypertext Markup Language*. 77

HTTP *HyperText Transfer Protocol*. xxi, 25, 26, 38, 51, 61, 65–67, 84–86, 114

HTTPS *HyperText Transfer Protocol Secure*. xxi, 38, 65, 84–86, 115

ID *IDentifier*. 24, 31, 33, 37–40, 61, 65, 66, 73, 77, 86

IMAP *Internet Message Access Protocol*. 50

IP *Internet Protocol*. 22, 25, 70, 71

ISTQB *International Software Testing Qualifications Board*. 12

IT *Information Technology*. 84, 85

JSON *JavaScript Object Notation*. 38, 51, 59–61, 86, 90

KiB Kibibyte. 68

KPIs *Key Performance Indicators*. 14, 15, 63, 64

LDAP *Lightweight Directory Access Protocol*. 50

MEI Mestrado em Engenharia Informática. 1

MI *Message Injector*. xxi, 4, 38, 49, 52

NFS *Network File System*. 29

OAT *Operational Acceptance Testing*. 12

PDCA *Plan Do Check Act*. 7

PIDs *Parameter IDs*. 29, 59, 61, 85

POP3 *Post Office Protocol 3*. 50

QA *Quality Assurance*. 1, 4, 6–8, 11, 28, 29

RAM *Random Access Memory*. 13, 15, 35, 41, 47, 62, 64, 71, 80, 83–85

Redis *REmote DIctionary Server*. 25

REST *REpresentational State Transfer*. 2, 25, 50, 55

SaaS *Software as a service*. 29

SCM *Source Code Management*. 68

SMS *Short Message Service*. 1

SMTP *Simple Mail Transfer Protocol*. 50

SOAP *Simple Object Access Protocol*. 50, 55

SOLID *Single Responsibility Principle, Open/Closed Principle, Liskov Substitution Principle, Interface Segregation Principle, Dependency Inversion Principle*. 37, 82

SSH *Secure Shell*. 68

TCP *Transmission Control Protocol*. 50

ToS *Threshold of Success*. 40

TTFB *Time To First Byte*. 15

TTLB *Time To Last Byte*. 15

UAT *User Acceptance Testing*. xix, 12, 30, 72, 73, 76, 81

UI *User Interface*. xix, 2, 4, 16, 28, 33, 38, 45–47, 50–54, 66, 72, 73, 76, 77, 89, 90

URL *Uniform Resource Locator*. 35, 68, 71, 78, 81

VM *Virtual Machine*. 48

Web *World Wide Web*. 2, 4, 15, 16, 25, 28, 50, 51, 54, 55, 65, 73

YAML *YAML Ain't Markup Language*. 47, 51, 67, 71

Esta página foi propositadamente deixada em branco.

Lista de Figuras

Lista de Tabelas

Esta página foi propositadamente deixada em branco.

Capítulo 1

Introdução

O presente documento apresenta o trabalho desenvolvido no âmbito da unidade curricular "Dissertação/Estágio" do Mestrado em Engenharia Informática (MEI) com especialização em Comunicações, Serviços e Infraestruturas.

O estágio decorreu na STRA, S.A., também denominada como Stratio Automotive ou Stratio. [1] O estágio foi supervisionado pelo Eng. Bernardo Marques (Stratio, Engenheiro de Garantia de Qualidade) e pelo Prof. Dr. Paulo de Carvalho (Departamento de Engenharia Informática (DEI), Faculdade de Ciências e Tecnologia da Universidade de Coimbra (FCTUC)).

1.1 Contexto

A Stratio foi fundada em 2012 em Coimbra, local onde é sedeada, e desenvolve tecnologias responsáveis pela gestão de frotas de veículos pesados. O seu principal propósito é automatizar a deteção de falhas em veículos. [1]

Inicialmente a Stratio *Databox* é colocada e configurada no veículo e, em tempo real, são recolhidos dados relativos ao veículo, desde dados da bateria, motor, sistema de travões, suspensão, caixa de velocidades entre outros. Após esta fase, os dados são analisados continuamente por algoritmos de *machine learning*. Posteriormente, o cliente pode acionar notificações via email e/ou *Short Message Service* (SMS) para a emissão de alertas e falhas. A prevenção de falhas permite reduzir os custos de manutenção e o tempo de reparação do veículo.

Este estágio integra-se na iniciativa da Stratio de produzir mecanismos de automação de testes de software. Os processos de *Quality Assurance* (QA) implementados pela Stratio focam-se particularmente na prevenção e no aprimoramento do processo de desenvolvimento por esta praticada.

1.2 Objetivos

O principal objetivo deste estágio é projetar e desenvolver soluções que permitam à Stratio o desenvolvimento de testes automáticos para a validação do software. Assim pretende-se munir a equipa de desenvolvimento de metodologias e ferramentas de modo a evitar a introdução de bugs durante o desenvolvimento do produto. É expectável expandir os

mecanismos de integração contínua já praticados pela equipa, melhorar as condições para a execução de testes automáticos, tornando a fase de validação o mais proativa e o menos invasiva possível para a equipa sem descartar nenhum nível de testes.

No âmbito deste projeto serão desenvolvidos testes automáticos para a validação de desempenho de uma API *REpresentational State Transfer* (REST) responsável pela receção de leituras de sensores distribuídos por vários veículos pesados e ligeiros. Ainda no âmbito deste projeto proceder-se-á ao desenvolvimento de testes automáticos focados na validação de uma UI disponível através da utilização de um *World Wide Web* (Web) *browser*, sendo que esta interface tem como propósito apresentar os dados processados relativos às leituras de sensores.

Para tal, é fundamental proceder ao levantamento do estado da arte no que toca à validação automática de software, nomeadamente, através da estratégia de teste *black-box*. Será também necessário recolher informação que permita que os ambientes alvo de teste sejam instanciados de forma a garantir propriedades necessárias aos testes como, por exemplo, serem repetíveis e com resultados semelhantes. Por último, dado que estes testes automáticos terão de ser inseridos num contexto de desenvolvimento de software em equipa, será ainda preciso investigar processos e metodologias que se adequem ao caso de uso da Stratio, bem como ferramentas de automação de processos que permitam integrar as validações automáticas desenvolvidas neste projeto no Ciclo de Vida de Desenvolvimento de Software (CVDS).

No final deste projeto deverá existir na Stratio um processo automático que providencie uma análise do desempenho da API REST, responsável pela receção de leituras de sensores dos veículos, sempre que nestas ou nas suas dependências sejam introduzidas alterações no código. Deverá ainda ser implementado um processo de validação automático que permita testar, pelo menos, um dos casos de uso da plataforma Web através da sua UI.

Resumindo, os objetivos deste estágio são:

- Desenvolvimento de testes automáticos para validação do desempenho de uma API que forneçam relatórios com diversas métricas, de modo a que se possa prever que quantidade de recursos é necessária para suportar determinada carga de mensagens;
- Desenvolvimento de testes automáticos para validação de uma UI;
- Integração dos testes automáticos com o CVDS.

1.3 Metodologia

O estágio teve lugar na Stratio, e como tal, será seguida a metodologia de desenvolvimento praticada pela equipa que a aluna integrou - Kanban. Kanban é um tipo de metodologia ágil que segue os princípios do manifesto ágil. [2]

Na fase inicial são planeadas as tarefas de *backlog*. De seguida, inicia-se o ciclo de desenvolvimento, neste caso de duas semanas. Durante este ciclo são realizadas reuniões diárias em que o objetivo é que cada elemento da equipa de software exponha as tarefas que tem realizado e que elenque as que planeia fazer de seguida. No final de cada *sprint* é realizado uma revisão, momento em que a equipa de software apresenta à restante empresa o que desenvolveu ao longo das duas semanas, e de identificar possíveis problemas bem como formas de os mitigar.

Durante o segundo semestre, o processo foi ligeiramente alterado de modo a adaptar-se às condições de trabalho remoto induzido pelas contingências da pandemia COVID-19. Cada semana passou a iniciar-se com uma reunião de "Status, Refinement & Planning" cujo objectivo era elencar e analisar os progressos alcançados em cada tarefa, identificar eventuais desvios e planear os respetivos ajustes, bem como definir as tarefas a serem desenvolvidas na semana seguinte.

Quinzenalmente é realizada um reunião entre a aluna e orientador do departamento com o propósito de definir certos aspetos relativamente ao estágio e verificar o progresso do mesmo.

Mensalmente passou a realizar-se uma reunião '*Retrospective*', entre a aluna e o orientador da empresa, cujo o objetivo era decidir sobre o *Start*, *Stop* e *Continue* de modo a indicar processos e métodos que podem melhorar o projeto, aqueles que devem parar de ser feitos, e aqueles que devem continuar a realizar-se, respectivamente. Sempre que era necessário, e quando a disponibilidade do orientador da empresa e da aluna coincidiam, era realizada uma breve reunião para esclarecimento de eventuais dúvidas.

Mensalmente é também realizada uma reunião com ambos os orientadores para esclarecimento de eventuais dúvidas, proporcionando a comunicação entre todos os intervenientes no estágio.

1.4 Planeamento

Recorreu-se à tecnologia GanttPRO [3] para a elaboração dos diagramas de Gantt para o primeiro e segundo semestre.

1.4.1 Planeamento primeiro semestre

No primeiro semestre, a fase inicial esteve focada na pesquisa de ferramentas e metodologias no âmbito do estágio, seguindo-se da definição do planeamento e clarificação dos objetivos propostos. Também nesta fase foi feito o levantamento do estado da arte e dos requisitos. Posteriormente, foi definida a arquitetura do projeto e os riscos associados ao estágio.

A escrita do relatório intermédio, apesar de estar indicada para a primeira semana de janeiro, foi efetuada durante toda a fase de documentação, uma vez que essas tarefas foram incluídas no relatório. Na calendarização foi tido em conta as entregas mensais de relatórios de progresso, bem como a submissão do relatório intermédio e a apresentação deste. O diagrama de *Gantt* planeado para o primeiro semestre pode ser observado no anexo A.

Existiram alguns desvios ao plano inicialmente definido para o primeiro semestre, tal como se pode observar no diagrama no anexo B.

A tarefa de pesquisa e a clarificação dos objetivos demoraram mais dois dias do que inicialmente previsto. A tarefa do estado da arte foi iniciada mais tarde, e a aluna optou por interromper esta tarefa e recomeçar posteriormente. Esta tarefa demorou mais uma semana do que o previsto e não foi possível finalizá-la na data inicialmente programada, causando atrasos significativos. As tarefas seguintes - requisitos, arquitetura e riscos - não foram finalizadas no dia e tempo previsto para as mesmas, que foi ligeiramente inferior ao realmente dedicado.

No diagrama é possível observar, a vermelho, duas tarefas, uma delas é o planeamento para o segundo semestre que não foi tida em conta anteriormente, e outra, como já foi referido, é a continuação da escrita do estado da arte. Claramente houve um grande desvio ao plano inicial, principalmente na tarefa de escrita do estado da arte, o risco R.1 (ver secção 3.7) não foi mitigado da melhor forma.

1.4.2 Planeamento segundo semestre

No segundo semestre, a começar na segunda semana de fevereiro, seria efetuada a automatização da bateria de testes de desempenho à Jam API com recurso a *scripts* de integração contínua. Com MI já desenvolvido pela Stratio, pretendia-se injetar mensagens num *endpoint* da Jam API para, posteriormente, serem recolhidas métricas - taxa de transferência e latência, outras métricas foram recolhidas após alguma investigação de quais se adequam melhor ao caso em questão. Idealmente, esta informação seria colocada numa base de dados e num relatório visual de desempenho. Posteriormente esta bateria de testes devia ser integrada no processo de validação do CVDS.

Na primeira semana de abril seria realizado o mesmo processo para os Serviços de Processamento de Dados.

As últimas duas semanas de maio seriam destinadas à validação da implementação efetuada.

As últimas quatro semanas – antes da entrega final - seriam destinadas à conceção e revisão do relatório, sendo este processo efetuado ao longo de todo o semestre.

O planeamento idealizado para o segundo semestre pode ser observado no diagrama de Gantt no anexo C.

1.4.3 Alterações ao planeamento do segundo semestre

Existiram bastantes alterações ao planeamento inicialmente definido para o segundo semestre que serão de seguida apresentadas.

Inicialmente previa-se automatizar testes de desempenho a Jam API e aos Serviços de Processamento de Dados. Após a defesa intermédia, surgiram dúvidas quanto à dimensão do projeto. Automatizar testes de desempenho às duas componentes, seria um trabalho repetitivo, ou seja, depois de automatizar os testes de desempenho a uma das componentes, a automatização dos testes para a outra componente seria semelhante. Alterações nas prioridades da empresa, também levaram a esta alteração.

Optou-se por substituir a tarefa de automatização de testes de desempenho aos Serviços de Processamento de Dados pela automatização de testes *end-to-end* a uma UI disponível através da utilização de um Web *browser*. Tal mudança implicou a utilização de uma nova tecnologia e linguagem. Deste modo, esta nova tarefa teve um valor acrescentado significativo para o estágio, permitindo a intervenção num contexto distinto e enriquecendo o estágio com novas aprendizagens e tecnologias. A automatização de testes *end-to-end* a uma UI é uma das várias tarefas planeadas no *roadmap* de QA da Stratio. Foi a tarefa escolhida para substituição tendo em conta a prioridade que a tarefa tinha no *roadmap*, o tempo do estágio e ainda o acrescento de conhecimento com novas tecnologias.

Outra mudança ocorrida ao inicialmente planeado foi a não utilização do MI desenvolvido pela Stratio para injeção de mensagens na componente Jam API. Esta ferramenta não foi

utilizada uma vez que existem outras no mercado que fornecem funcionalidades de interesse para este projeto, como a produção automática de relatórios com métricas de desempenho. Foi necessário fazer um levantamento do estado da arte para perceber qual a ferramenta que melhor se adequava ao caso de uso em questão, tarefa esta que não estava planeada.

No anexo D é possível observar o diagrama de Gantt real do segundo semestre.

As primeiras três semanas destinaram-se ao levantamento do estado da arte em ferramentas para injeção de mensagens.

Posteriormente seguiu-se a implementação para automatizar testes de desempenho.

A segunda tarefa de desenvolvimento - Validação do desempenho da API na *Amazon Web Services* (AWS) - era uma tarefa que não estava prevista. Foi realizada, dado que a empresa pretendia fazer a migração da Jam API do Azure para a AWS.

A terceira tarefa de desenvolvimento - alterações à fase de automatização de testes de desempenho - prendeu-se com o facto de a versão do Kubernetes utilizada na empresa apresentar limitações para a realização da mesma. Esta versão tem um problema já reportado ([4]) e resolvido em versões seguintes: não é possível limitar a utilização de *Central Processing Unit* (CPU) em *containers* Windows. Devido a restrições de *roadmap* da empresa, não seria atualizada a versão do Kubernetes durante o estágio e como tal foi necessário repensar a estratégia de execução dos testes e posteriormente proceder a mudanças na implementação.

De seguida realizou-se o levantamento do estado da arte em ferramentas de automatização de testes *end-to-end*.

Foi definido o plano de testes para a fase de automatização de testes *end-to-end* e de seguida foi desenvolvido todo o código necessário para a sua implementação.

As últimas semanas foram dedicadas à escrita deste documento e à preparação da apresentação final.

De forma mais detalhada são apresentas no anexo E as tarefas realizadas no segundo semestre. Estas tarefas foram definidas com recurso ao software Jira (ver subsecção 2.9.8) e como tal são classificadas pelos atributos sumário, prioridade e épico.

1.5 Controle de qualidade

Ao longo do estágio foram executadas algumas medidas de modo a monitorizar e assegurar o progresso da aluna. Todas as semanas era entregue um relatório semanal do estilo 5-15 [5] a ambos os orientadores onde são apresentadas as tarefas realizadas, as tarefas a realizar na semana seguinte e ainda as restrições que possam pôr/puseram em causa o planeamento. Quinzenalmente era realizada um reunião entre a aluna e orientador do departamento com objetivo de definir certos aspetos relativamente ao estágio e verificar o progresso da aluna. Mensalmente era realizada uma reunião com ambos os orientadores para esclarecer eventuais dúvidas e de modo a manter a comunicação entre todos os intervenientes no estágio. Por último, mensalmente a aluna era responsável por entregar o relatório de progresso na plataforma de estágios do DEI. [6]

1.6 Estrutura do documento

No primeiro capítulo procedeu-se à exposição do contexto e objetivos deste trabalho, bem como a metodologia utilizada no decorrer do estágio e o planeamento efetuado para este. O restante documento está organizado da seguinte forma:

Capítulo 2 pretende-se expor alguns conceitos fundamentais de QA, técnicas, níveis e tipos de teste de software e metodologia de desenvolvimento e operações. É elaborada uma análise de metodologias e processos para integrar validação no CVDS, incluindo o métodos de integração e entrega contínua. Apresenta-se também o *background* tecnológico, a arquitetura e processos da Stratio.

Capítulo 3 são expostos os requisitos relativos ao trabalho a desenvolver, os requisitos necessários para uma correta seleção das ferramentas, os riscos associados ao projeto, bem como um plano de mitigação para cada risco.

Capítulo 4 procede-se à comparação de ferramentas de integração contínua e contínuo *deployment*, ferramentas para injeção de mensagens e ferramentas para automatização de testes *end-to-end*.

Capítulo 5 apresenta-se em detalhe a arquitetura e a implementação realizada no projeto.

Capítulo 6 procede-se à exposição de processos para validação dos requisitos, e é feita uma avaliação do resultado final deste projeto, são ainda apresentados resultados dos testes realizados.

Capítulo 7 finalização do documento com uma visão geral do trabalho efetuado e do trabalho futuro.

Capítulo 2

Background

No presente capítulo proceder-se-á à exposição de conceitos utilizados neste documento de modo a contextualizar o leitor. Começa-se por introduzir o contexto sobre o que é a garantia de qualidade e qual é a sua importância. Posteriormente, são expostas as técnicas, níveis e tipos de teste dando ênfase às mais praticadas ao longo do projeto. A metodologia de desenvolvimento e operações, bem como as práticas a esta associadas, são também apresentadas, uma vez que são práticas implementadas na Stratio. Por último, é apresentado o *background* tecnológico, que menciona as tecnologias utilizadas ao longo deste projeto, e a arquitetura e processos empregues na Stratio.

2.1 *Quality Assurance*

A garantia de qualidade, do inglês *Quality Assurance* (QA), é essencial para assegurar a confiabilidade e a satisfação dos clientes num dado produto. O foco de QA é a constante melhoria de processos de modo a fornecer um produto de elevada qualidade ao cliente. Estes processos de melhoria devem ser eficientes e devem ser regidos por padrões de qualidade estabelecidos para produtos de software, admitindo que a qualidade de um produto de software é diretamente influenciada pelo ciclo que constitui esse mesmo software. [7]

QA possui um ciclo conhecido por ciclo *Plan Do Check Act* (PDCA) ou *Deming*. Este ciclo possui quatro fases:

1. Planear;
2. Fazer;
3. Verificar;
4. Agir.

Na primeira fase são definidos os objetivos e estabelecidos quais os processos requeridos para entregar um produto final de qualidade. Na fase seguinte é posto em prática o que foi definido na fase anterior, isto é, desenvolvimento dos processos. Na terceira fase é realizada uma análise dos processos desenvolvidos. É verificado se os processos se encontram em conformidade com o que foi definido na primeira fase e, caso não estejam de acordo, pretende-se identificar os problemas e definir o plano de mitigação. Na última fase, são implementadas alterações para refinar os processos e aperfeiçoar os resultados.

As quatro fases do ciclo de QA pretendem ajudar a alcançar e manter os padrões de qualidade estabelecidos para um determinado produto. [8]

Para garantir a qualidade que qualquer cliente procura, as organizações definem assim metodologias e processos de modo a garantir que as etapas, enunciadas anteriormente, sejam cumpridas. No entanto, um erro bastante comum prende-se com a ideia de que QA é conseguida num determinado momento do Ciclo de Vida de Desenvolvimento de Software (CVDS), quando na realidade QA deve ser garantida em todos os momentos do ciclo.

2.2 Técnicas de teste de software

O propósito desta secção é rever as técnicas de testes de modo a definir e apresentar o objetivo de cada técnica. As técnicas de testes são classificadas em *black-box*, *white-box* e técnicas baseadas na experiência.

2.2.1 *Black-box*

Black-box, tal como o nome sugere, é uma estratégia de teste em que não se conhece a estrutura interna do software a ser testado. Esta metodologia baseia-se apenas nos requisitos e especificações do software. [9] Também denominada como técnica comportamental ou baseada em comportamento, esta foca-se nos *inputs* e *outputs* do software, sem assumir qualquer conhecimento da implementação, isto é, não é considerado o comportamento interno deste. Esta técnica pode ser empregue em testes funcionais e não funcionais.

Existem diversas técnicas *black-box*. De seguida são expostas em detalhe as mais relevantes para este projeto.

Particionamento por equivalências

O particionamento por equivalências, tal como o nome indica, divide os dados/condições em partições, também denominadas como classes de equivalência, que devem exibir um comportamento semelhante, ou seja, o sistema deve tratá-los de igual forma. Assim, é necessário apenas testar um dado de cada partição, uma vez que numa partição todos os dados que a constituem são tratados da mesma maneira pelo software. [10]

Esta técnica é normalmente empregue para reduzir o número total de casos de teste a um conjunto finito de casos. Utiliza assim o menor número de casos de teste possível para cobrir o máximo de requisitos. [11]

Existem partições para valores válidos e inválidos. A partição de equivalência válida, como é denominada, contém os valores que devem ser aceites pelo sistema. Pelo contrário, a partição de equivalência inválida contém os valores que devem ser rejeitados pelo sistema. Cada valor só pode pertencer a uma e uma só partição.

A cobertura desta técnica, por norma expressa em percentagem, é calculada pelo número de partições de equivalência testadas por pelo menos um valor a dividir pelo número total de partições de equivalência definidas. Para atingir uma cobertura de 100% os casos de teste devem cobrir todas as partições definidas, utilizando pelo menos um valor de cada partição. [12]

Suponha-se o seguinte exemplo: testar um determinado campo num formulário que aceita no mínimo dez caracteres e no máximo vinte. Não é praticável testar os valores abaixo de dez, entre dez e vinte e acima de vinte. Aplicando a técnica de particionamento por

equivalências, divide-se o número possível de caracteres em três partições de equivalência. Neste caso foi dividido em três partições, uma válida e duas inválidas como é possível observar no esquema 2.1.

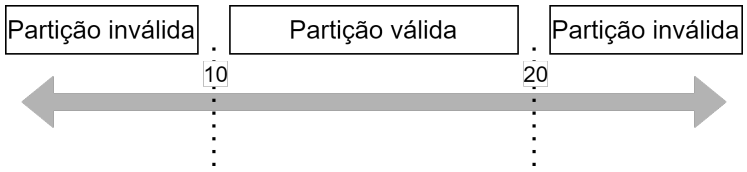

Figura 2.1: Exemplo particionamento por equivalências

Assim, na primeira partição abaixo de dez caracteres o sistema não deve aceitar; entre dez a vinte caracteres o sistema deve aceitar; e com mais de vinte caracteres o sistema não deve aceitar. Posteriormente é necessário escolher pelo menos um valor de cada partição para teste. Verifica-se assim que com esta técnica o número de casos de teste é significativamente reduzido, mantendo elevada a cobertura do teste.

Análise do valor fronteira

A análise do valor fronteira, do inglês *Boundary Value Analysis* (BVA), é utilizada para identificar problemas nos limites. Por norma os valores nas fronteiras do domínio são mais propensos a comportamentos incorrectos no sistema e esta técnica visa combater tal falha, testando os limites dos valores. [11]

Esta técnica é uma extensão da técnica de particionamento por equivalências. Nesta técnica os casos de teste são definidos nos limites das partições, ou seja, o mínimo e máximo de uma partição são os valores fronteira. [12]

Observando o esquema apresentado na figura 2.1, as fronteiras são nove e dez caracteres e vinte e vinte um caracteres. Algumas variações desta técnica identificam três valores fronteira - nove, dez, onze e dezanove, vinte e vinte um caracteres.

Esta técnica permite assim revelar defeitos que possam existir ao forçar o software a mostrar o seu comportamento nas fronteiras das partições. Comummente esta técnica é utilizada para testar os requisitos que exigem vários números como, por exemplo, datas e horas. [12]

2.2.2 *White-box*

As técnicas *white-box*, ao contrário das *black-box*, são baseados na estrutura interna do software. Esta técnica assume o conhecimento da lógica do item a ser testado. Neste caso o *tester* escolhe os *inputs* que permitem exercitar os caminhos através do código e determina quais serão os *outputs* adequados. [13] Existem várias técnicas *white-box*, entre as quais se destacam: *control flow*, *data flow* e teste de cobertura e decisão. [12]

2.2.3 Técnicas de teste baseadas na experiência

As técnicas de teste baseadas na experiência tiram proveito da experiência de programadores, *testers* e utilizadores para definir, implementar e executar testes. Estas técnicas são muitas vezes combinadas com as duas técnicas mencionadas anteriormente, *black-box* e *white-box*. [12] De seguida, expõem-se as duas principais técnicas baseadas na experiência.

Antecipação de erros

A antecipação de erros é uma técnica utilizada para prever erros e defeitos, com base no entendimento que o *tester* detém.

Esta técnica tira partido da habilidade e experiência anterior de um *tester* para identificar casos de teste especiais que, de outra forma, não seriam identificados.

Uma abordagem típica desta técnica consiste na identificação de defeitos comuns, e, com esses dados projetar casos de teste que irão exibir tais defeitos. [12]

Teste exploratório

O teste exploratório, também conhecido como teste informal ou não predefinido, é uma técnica em que os casos de teste não são definidos com antecedência, é sim realizada uma verificação do componente em tempo real. [14]

Este teste é realizado quando existem especificações inadequadas ou estas são inexistentes. Com este teste pretende-se, assim, aprender mais sobre o componente a testar e identificar as áreas que podem necessitar de mais testes. [12] O teste exploratório é visto como um exercício de "pensamento", ou seja, pretende-se tirar ideias para planear os casos de teste. [14]

2.3 Níveis de teste

Na figura 2.2 é apresentada a pirâmide de testes que define os níveis de teste. À medida que subimos o nível da pirâmide, aumenta a dificuldade de implementação, o tempo de execução e a integração dos sistemas, isto é, aumenta o número de componentes a testar simultaneamente. Este acréscimo de complexidade está associado à instabilidade nos testes, dificuldades de manutenção dos mesmos e longos tempos de execução. Como tal, a base da pirâmide deve cobrir o maior número de testes possíveis, de modo a que a fase de testes se torne num processo rápido.

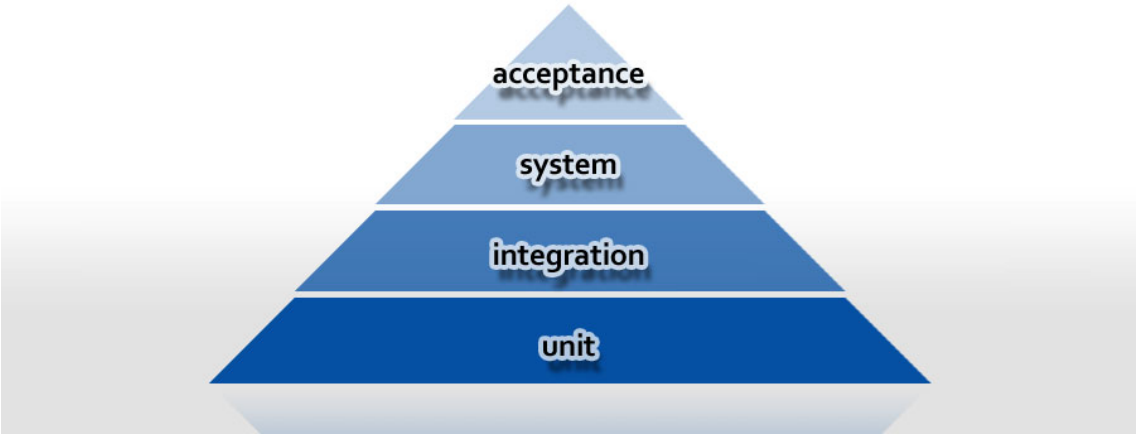

Figura 2.2: Pirâmide de níveis de teste (retirado de [15])

2.3.1 Testes de Unidade

Os testes de unidade, de componentes ou de módulos, têm como objetivo testar componentes do software de forma isolada. [12] São testes facilmente automatizados que executam rapidamente e é fácil de entender a causa caso o teste falhe. [16]

Por norma, quem realiza os testes a este nível é o programador que desenvolveu a componente em causa e, como tal, é utilizada a técnica *white-box*. Existem exceções em que é a equipa de QA que realiza estes testes.

Este nível é a base da pirâmide de testes, tal como podemos observar na figura 2.2. Quando bem implementados, os testes de unidade permitem descobrir bugs numa fase inicial do desenvolvimento, economizando tempo e e dinheiro ao CVDS.

2.3.2 Testes de Integração

Os testes de integração têm como propósito verificar o funcionamento da integração entre componentes do sistema. Podem ser divididos em dois níveis:

- Testes de integração de componentes - por norma efetuado pelos programadores após os testes de componentes, com a finalidade de testar a interação entre componentes (as menores unidades testáveis do produto);
- Testes de integração de sistema - executado pela equipa de QA, com a finalidade de testar a interação entre sistemas, *packages* ou até mesmo entre serviços externos à empresa. Este nível pode ser realizado após ou em paralelo com os testes de sistema.[12]

O foco destes testes consiste em verificar a comunicação entre componentes e não a funcionalidade de cada componente isoladamente como executado no nível de testes anterior.

Existem quatro tipos de abordagens neste nível: *bottom up*, *top down*, *hybrid* e *big-bang*.

A abordagem *bottom up* começa por testar os componentes individualmente e vai sendo incrementado a integração entre eles. A abordagem *top down* é contrária ao *bottom up*, isto é, começa por ser testado o maior número de componentes integrados e posteriormente vai se reduzindo até testar os componentes individualmente. A abordagem *hybrid* combina as abordagens de *bottom up* e *top down*. A abordagem *big-bang* testa os componentes todos juntos de uma só vez. [17]

2.3.3 Testes de Sistema

Os testes de sistema têm como objetivo verificar o comportamento do sistema ou produto como um todo. Pretendem, assim, verificar se o sistema está em conformidade com os requisitos funcionais e não funcionais. Neste nível de testes são consideradas tarefas *end-to-end* do sistema, isto é, os testes realizados a este nível avaliam uma dada funcionalidade do início ao fim.

Como em todos os níveis anteriores, este nível de testes pretende evitar que escapem os bugs para o nível seguinte. Por norma, este nível é desempenhado pela equipa de QA e é utilizada a técnica *black-box*. Este nível gera informações que permitem decidir se o sistema ou produto podem ou não ser lançados.

2.3.4 Testes de Aceitação

Os testes de aceitação são a última fase da pirâmide de testes (figura 2.2). O seu foco não é encontrar bugs, mas sim avaliar a aptidão do sistema para ser entregue e utilizado pelo cliente. [12] Este nível de testes pretende verificar se o sistema está em conformidade com os requisitos inicialmente estabelecidos. Por norma a técnica usada é *black-box*. [18]

Segundo o *International Software Testing Qualifications Board* (ISTQB) existem quatro formas comuns de testes de aceitação:

- Testes de aceitação de utilizador;
- Testes de aceitação operacionais;
- Testes de aceitação de contrato e regulamento;
- Testes alfa e beta.

Os testes de aceitação de utilizador, do inglês *User Acceptance Testing* (UAT), focam-se em validar a capacidade para o uso do sistema. Tal como o nome indica, são executados por utilizadores e podem ser executados em ambientes reais ou de simulação. O objetivo é perceber se o sistema responde ou não às necessidades dos utilizadores.

Os testes de aceitação operacionais, do inglês *Operational Acceptance Testing* (OAT), focam-se em aspetos operacionais, tal como, *backup*, instalações, actualizações, desempenho. Estes testes, normalmente, são executados por administradores de sistema em ambientes de produção. Têm como propósito compreender se o sistema funciona corretamente sob circunstâncias excecional.

Os testes de aceitação de contrato e regulamento têm como alvo verificar se o contrato e regulamento são cumpridos. Este tipo de testes pode ser realizado por utilizadores ou por *testers*.

Os testes alfa e beta têm como propósito obter feedback de clientes ou futuros clientes. Por norma, os testes alfa são realizados primeiro que os testes beta e são executados por pessoas internas à empresa que desenvolveu o software, e não pela equipa de programadores. Os testes beta são realizados por clientes ou possíveis clientes. [12]

2.4 Tipos de teste

Existem dois grandes tipos de teste: funcionais e não funcionais.

Os testes funcionais avaliam se as funcionalidades que o software desempenha se encontram em conformidade com os requisitos funcionais. Focam-se no que é que o software faz. [12] Este tipo de testes permite verificar se o produto está apto a realizar as funcionalidades para o qual foi desenvolvido.

Os testes não funcionais tem como principal objetivo analisar as características do software, tal como eficiência de desempenho, compatibilidade, segurança, entre outras. [19] Este tipo de testes foca-se na "qualidade do comportamento" do sistema. Contrariamente aos testes funcionais, a preocupação destes testes é em como o software faz, e não o que deveria fazer. [12]

Existem muitos outros tipos de teste, sendo que, nas subsecções seguintes são expostos os mais relevantes para o entendimento deste documento.

2.4.1 Testes de Desempenho

Os testes de desempenho são um tipo de testes não funcional. Têm como objetivo verificar se o produto funciona como expectado sob determinada carga de trabalho. Este tipo de teste permite recolher informações de velocidade, escalabilidade e estabilidade do produto sob determinadas cargas de trabalho, para posteriormente ser verificado a sua concordância ou não com os requisitos definidos. [20]

Posto isto, o foco não é encontrar bugs, mas sim verificar se existem:

- problemas de velocidade - respostas lentas ou tempos de carregamento longos;
- *bottlenecks* - áreas congestionadas ou bloqueadas;
- problemas de configuração do software;
- recursos de hardware suficientes - *Central Processing Unit* (CPU) e *Random Access Memory* (RAM), por exemplo;
- escabilidade suficiente - se o produto de software tem ou não capacidade de lidar com um determinado número de utilizadores. [21]

Dentro dos testes de desempenho existem vários tipos de teste, tal como se pode observar no esquema da figura 2.3.

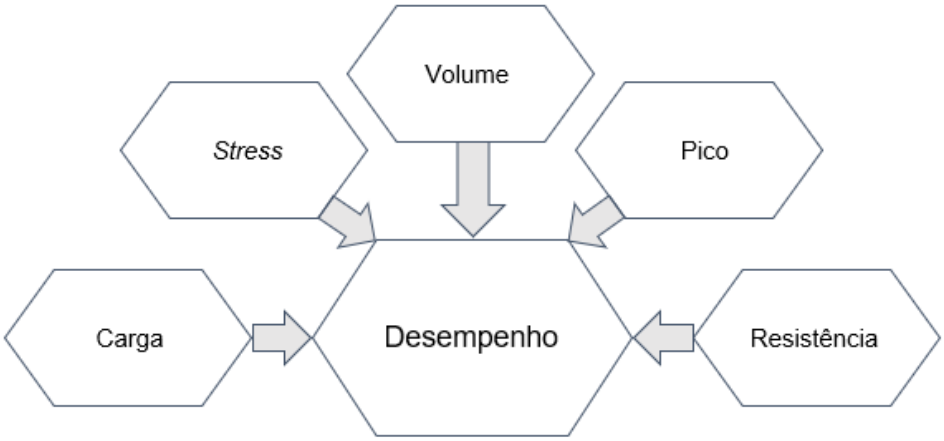

Figura 2.3: Tipos de testes de Desempenho

Os **testes de carga** pretendem medir o desempenho do sistema alvo de teste sob cargas de trabalho, que é expectável que este venha a observar quando em operação.

Os **testes de stress**, também conhecido como testes de fadiga, verificam o desempenho do sistema no limite das condições normais de trabalho, condições extremas de carga de trabalho. O objetivo deste teste é verificar se o sistema falha, em que momento essa falha ocorre e observar como o sistema reage a essa falha. [22]

Os **testes de volume**, também conhecidos como *flood tests*, uma vez que "inundam"o sistema. Estes testes analisam o desempenho do sistema quando este lida com um grande volume de dados, isto é, pretendem averiguar qual a eficiência do sistema com diferentes volumes de dados.

Os **testes de pico**, do inglês, *spike tests*, tencionam avaliar o sistema quando acontecem picos da carga de trabalho, isto é, quando a carga de trabalho aumenta substancial, rápida e repetidamente num período reduzido de tempo.

Os **teste de resistência**, também denominados testes de imersão, do inglês *soak test*, visam observar o desempenho do sistema com cargas de trabalho normais durante um longo período de tempo. Estes testes identificam a existência de falhas de memória no sistema, uma vez que, caso estas existam podem influenciar significativamente o desempenho do sistema ou até causar o lapso do sistema. [23]

Processo de Testes de Desempenho

O processo seguido, por norma, para a definição e execução de testes de desempenho é apresentado no esquema da figura 2.4.

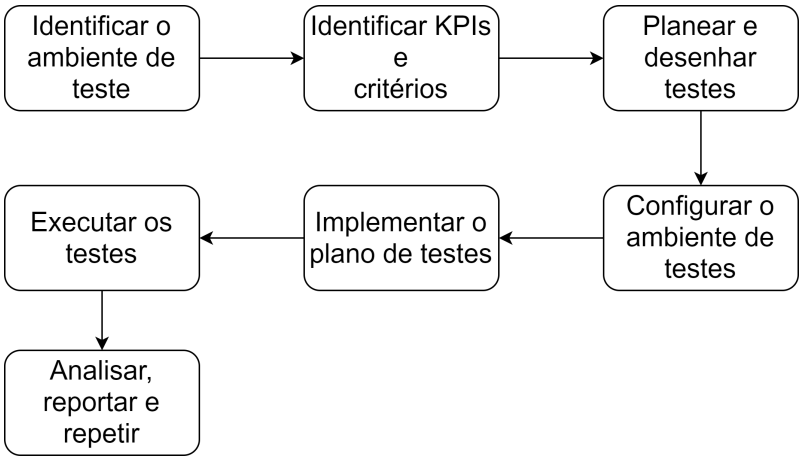

Figura 2.4: Processo para a execução de testes de desempenho

Passo 1: **Identificar o ambiente de teste**

Identificar o ambiente alvo de teste, desde as configurações de hardware às ferramentas disponíveis para, posteriormente, ser definido o plano de testes. Na maioria das situações, no contexto de testes de desempenho, o ambiente a testar não deve ser o ambiente de produção real. Envolve o esforço e coordenação de várias equipas, e o teste só deve ser executado em momentos de baixa utilização do produto para não comprometer a utilização deste pelos clientes reais. [24]

Deve ser sim, um ambiente o mais semelhante possível ao ambiente de produção ou um subconjunto deste ambiente com menos servidores ou base de dados, por exemplo, sendo que nem sempre é conseguido devido a restrições de recursos.

Passo 2: **Identificar os *Key Performance Indicators* (KPIs) de desempenho e os critérios**

Definir que KPIs serão calculados e identificar os critérios de aceitação e suspensão.

Passo 3: **Planear e projetar os testes de desempenho**

Identificar os cenários de teste, tendo em consideração a variação de parâmetros de entrada.

Passo 4: **Configurar o ambiente de teste**

Preparar o ambiente e todas as ferramentas necessárias para execução e monitorização do teste.

Passo 5: **Implementar o plano de teste**

Desenvolver toda a estrutura e código necessário para a execução dos testes.

Passo 6: **Executar os testes**

Execução e monitorização dos testes e guardar os resultados destes.

Passo 7: **Analisar e repetir**

Analisar os dados obtidos e executar novamente com os mesmo parâmetros de modo a obter resultados mais significativos. [21]

Métricas associadas aos Testes de Desempenho

Os KPIs, no contexto de testes de desempenho, são métricas que permitem avaliar a eficiência e eficácia de um produto de software. [25]

Existem vários KPIs de desempenho que devem ser identificados aquando a definição do plano de teste. De seguida, serão apresentados os mais comuns:

- Utilizadores concorrentes - quantidade de utilizadores virtuais que estão ativos num determinado momento;
- Taxa de transferência média - número de transições ou pedidos processados, em média, por unidade temporal (segundo, minuto, hora);
- Tempo de resposta - *Time To Last Byte* (TTLB) é o tempo do envio do primeiro byte até ao último recebido pelo utilizador, isto é, tempo decorrente entre o envio do pedido até à resposta ser recebida na totalidade;
- Taxa de erros - percentagem de pedidos que resultam em erros em comparação com todos os pedidos;
- Latência - *Time To First Byte* (TTFB) é tempo desde o envio do pedido, processamento do lado do servidor até ao momento que o cliente recebe o primeiro byte, ou seja, tempo decorrente entre o envio do pedido até começo da receção da resposta; [26]
- Utilização de CPU - quanto CPU é necessário para processar o pedido;
- Utilização de RAM - quanta RAM é necessária para processar o pedido.

2.4.2 Testes *End-to-end*

Os testes *end-to-end* são um método que testa um software do início ao fim. [27] O principal objetivo do teste é testar a experiência do utilizador final, ou seja, são reproduzidas as ações e comportamentos do utilizador e simulam-se em testes. [28]

O software testado pode ser uma aplicação *World Wide Web* (Web), uma aplicação de computador, uma *Application Programming Interface* (API), entre outras possibilidades. Independentemente do software, a ideia é testar o aplicativo com todas as suas dependências e simular as ações de um utilizador real.

Os testes *end-to-end* estão, por norma, associados ao topo da pirâmide de testes (ver imagem 2.2). São, assim, testes mais lentos de executar e mais difíceis de manter, em comparação com a base da pirâmide. [29]

Processo de Testes *End-to-end*

Os testes *end-to-end* executados neste projeto foram realizados a uma *User Interface* (UI) disponível através de um Web *browser* e, como tal, o processo seguidamente apresentado é adaptado a este caso. O esquema da figura 2.5 resume os passos deste processo.

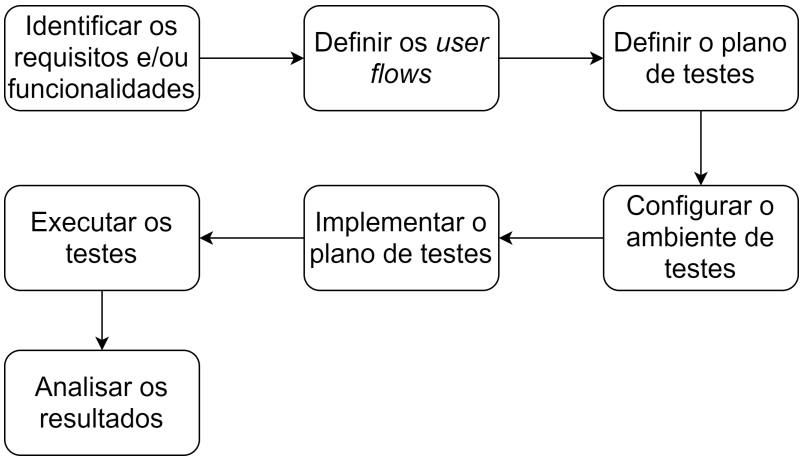

Figura 2.5: Processo para a execução de *end-to-end* à uma UI

Passo 1: **Identificar as funcionalidades**

Identificar e listar as funcionalidade da componente a ser testada. Listar os dados de entrada de uma dada ação e os resultados que são esperados obter. Neste primeiro passo deve ficar claro como a componente funciona em todos os aspetos, devem ser claramente identificadas as UI's, funcionalidades e recursos a testar, bem como é que estas devem responder.

Passo 2: **Definir *user flows***

Os *user flows* são um conjunto de etapas executadas por um utilizador, por meio de uma aplicação, para atingir um determinado objetivo. [30] Os *user flows* definidos devem ser os mais comuns, isto é, os caminhos que os utilizadores reais realizam mais frequentemente ao longo da utilização da aplicação.

Passo 3: **Definir o plano de testes**

Definir o plano de testes. Começando por definir que ambiente será testado bem como o *scope* de testes, as ferramentas e/ou linguagens que vão ser utilizadas para a implementação e os critérios de aceitação e suspensão.

Passo 4: **Configurar o ambiente de testes**

Preparar o ambiente alvo de teste, com todas as configurações necessárias para a execução dos testes.

Passo 5: **Implementar o plano de testes**

Desenvolver todo o código e/ou integração com outras ferramentas para executar o plano de testes.

Passo 6: **Executar os testes**

Execução dos testes e guardar os resultados destes.

Passo 7: **Analisar os resultados**

Analisar os resultados obtidos, verificar se o teste passou com sucesso ou insucesso, e neste último caso, identificar onde ocorreu a falha. [27]

2.4.3 Testes de Regressão

O teste de regressão consiste em testar um produto já testado após modificações a este. O principal objetivo é garantir que não tenham sido introduzidos bugs em áreas inalteradas do software após as alterações realizadas. Assim, pretende-se que um software funcione como esperado na sua totalidade após adição de novas funcionalidades ou atualizações. Este tipo de teste só é realizado quando são feitas modificações no software ou ambiente. [12]

2.4.4 Testes de mutação

Teste de mutação é um tipo de teste de software em que são efetuadas alterações no código-fonte para se verificar que os casos de teste são capazes de encontrar tais defeitos.

Este tipo de teste é *white-box* e é tipicamente realizado na base da pirâmide de testes (ver figura 2.2) - testes de unidade. O teste de mutação é um teste *fault-based*, isto é, baseado em falha, uma vez que envolve a criação de uma falha no programa. [31]

O principal objetivo deste teste é emular falhas no software, para criar um conjunto de programas com defeitos, denominados mutantes, para verificar a qualidade dos casos de teste. Após a execução destes mutantes, se o resultado for diferente do resultado da execução do programa original, significa que a falha foi detetada. [32]

Um indicador para medir a qualidade dos casos de teste é a pontuação de mutação. A pontuação de mutação é calculada pelo quociente entre o número de falhas detetadas e o número total de falhas injetadas. [33]

Observe-se o exemplo:

```
int checkSmaller(int x, int y){
    if (x < y){
        return x;
    }
    else{
        return y;
    }
}
```

Para criar o código mutante, basta fazer ligeiras alterações ao código original, como trocar os operadores, como por exemplo:

```
int checkSmaller(int x, int y){
    if (x > y){
        return x;
    }
    else{
        return y;
```

```
    }
}
```

2.5 Metodologia de Desenvolvimento e Operações

A metodologia Desenvolvimento e Operações, popularmente conhecido como *DevOps*, é descrito por Emily Freeman como: "(...) an engineering culture of collaboration, ownership, and learning with the purpose of accelerating the software development life cycle from ideation to production". [34]

Esta metodologia ou filosofia, como muitas vezes é considerada, surgiu da necessidade de unir as equipas de desenvolvimento e operações. Estas equipas, tradicionalmente em modelos *waterfall* ou ágeis, eram separadas, trabalhando assim independentemente e com processos diferentes. A separação destas equipas causava diversos problemas, sendo a falta de comunicação o mais problemático, o que levava a atrasos na produção do produto. A metodologia *DevOps* foi criada para colmatar estas falhas, com o propósito de melhorar a comunicação entre as equipas de desenvolvimento e operações e alinhar a estratégia de ambas, de forma a tornar as entregas do produto mais rápidas e frequentes. [35]

Na figura 2.6 é possível observar o ciclo DevOps, tipicamente, este é divido em oito fases.

A primeira tarefa do ciclo é o planeamento. É necessário planificar o produto que se pretende desenvolver. O código desenvolvido é escrito e integrado nas tarefas de código e *build*. A tarefa de teste é onde é feita a verificação se o produto desenvolvido está em conformidade com os requisitos inicialmente estabelecidos. No caso de a etapa de testes ter tido sucesso, é feita um *release*, isto é, uma nova versão do produto fica disponível em produção. Posteriormente no *deploy*, é feita a instalação e configuração da nova versão do produto num servidor. A tarefa de operação consiste na configuração do produto para o cliente, caso seja necessário. A última tarefa, monitorização, consiste em controlar o produto e registar possíveis falhas. [36]

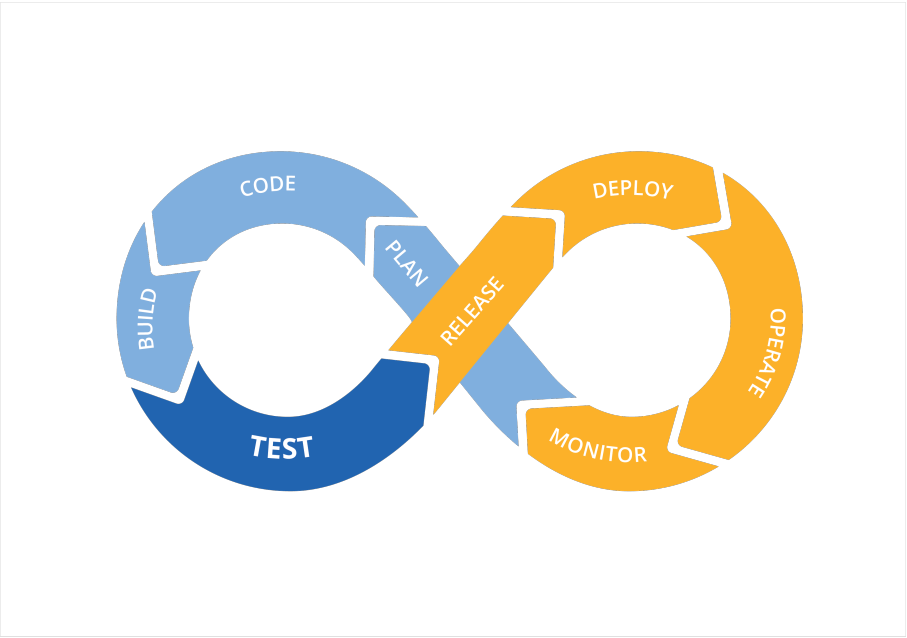

Figura 2.6: Ciclo de vida da metodologia DevOps (retirado de [37])

A metodologia DevOps envolve várias práticas ou fases que devem ser empregues: desenvolvimento contínuo, teste contínuo, integração contínua, entrega contínua, *deployment* contínuo e monitorização contínua. Na figura 2.7 relaciona-se estas práticas com as tarefas anteriormente mencionadas (figura 2.6). Algumas destas práticas vão ser detalhadas nas secções seguintes.

É importante referir que, quando a metodologia DevOps é adotada num caso real, nem sempre são implementas todas as práticas referidas ou, quando implementadas, nem sempre são automatizadas. Isto não significa que a metodologia DevOps não esteja a ser praticada, apenas indica que os processos requerem desenvolvimento e maturação.

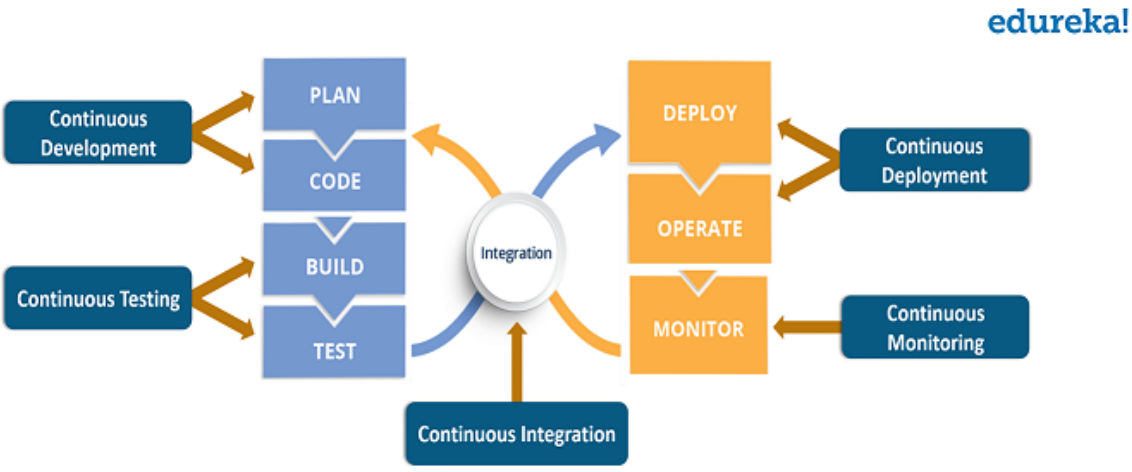

Figura 2.7: Ciclo de desenvolvimento da metodologia DevOps (adaptada de [37])

Esta metodologia pretende assim acelerar o processo de *deployment* e torná-lo mais frequente. Para além dos benefícios já mencionados, esta metodologia, combinada com testes automatizados, integração, entrega e monitorização contínua, permite que os problemas sejam encontrados com antecedência, contribuindo assim para um produto com mais qualidade.

2.6 Integração Contínua

A integração contínua ou *Continuous Integration* (CI) é um método em que os programadores integram continuamente as alterações efetuadas ao seu trabalho num repositório partilhado.

Cada submissão (*commit*) é verificada por uma bateria de testes e posteriormente é dado feedback ao programador sobre o sucesso ou insucesso da mesma, sendo que todo este processo é automatizado. A verificação a cada submissão e o respetivo feedback ao programador, permite que, de forma ágil, este seja informado de um eventual bug e possa corrigi-lo o mais rapidamente possível. Segundo Martin Fowler "Continuous Integrations doesn't get rid of bugs, but it does make them dramatically easier to find and remove." [38] Este método, ao permitir a localização de bugs no código atempadamente, reduz consideravelmente o tempo de correção destes e consequentemente a promoção da melhoria contínua do produto.

Um ponto principal deste processo é o controle de versões do código, uma vez que cada versão do software (*commit*), se encontra etiquetada com um resultado da bateria de

testes, toda a equipa sabe quais as versões com defeitos, por quem e onde é que estes foram introduzidos. [39]

2.7 Entrega Contínua

Entrega contínua, do inglês *Continuous Delivery* (CD), é um método comummente associado a CI. Este possibilita que as novas alterações no código sejam rapidamente disponibilizadas. As *releases* para produção podem ser mais frequentes (diariamente, semanalmente, etc) conforme o que for mais adequado para os requisitos definidos. [40] Os programadores unem as alterações efetuadas ao código num único artefacto. Posteriormente, é feita a *release* para produção, onde aguarda validação para mais tarde ser *deployed* (secção 2.8). [41] A equipa de desenvolvimento garante uma versão do produto concluída após cada *sprint* e a equipa de negócio é que decide quando é realizada a *release*. Esta etapa requer etapas manuais antes da *release* para produção, isto é, intervenção humana no processo. [42]

Quando é referida a velocidade como principal benefício, é de notar que é importante que a entrega contínua esteja associada a testes contínuos ao produto. [42] Esta prática aumenta a produtividade da equipa de desenvolvimento, uma vez que evita que esta desperdice tempo com a logística e aplique mais foco ao desenvolvimento do produto.

2.8 *Deployment* Contínuo

Continuous Deployment (CD*) é o método em que o processo de *release* de um dado produto é totalmente automatizado sem a necessidade de qualquer intervenção humana. Se todas as alterações efetuadas no código passarem nos testes automatizados, o produto é automaticamente *deployed* para produção. Caso contrário, o produto é rejeitado. [41] Este método permite assim que menores versões sejam *release* frequentemente, o que faz com que o feedback dos clientes seja mais rápido e, consequentemente, torna a resolução de problemas mais ágil para a equipa de desenvolvimento. [43]

Sem testes automatizados não é possível adoptar CD*. Os testes automatizados são referidos como a dependência mais crítica para a correta adoção do CD* e são a base deste método. [41] Um dos receios deste método é precisamente os testes automatizados, uma vez que é necessário ter um ambiente de testes confiável e rigoroso. O facto de não haver intervenção humana, se não for considerado algum teste ou o teste esteja mal definido, o produto pode passar na validação e ir para produção com bugs, o que pode causar custos substanciais à empresa. [43] Outra desvantagem apontada é o custo inicial de adoção de CD* e contínua manutenção deste. [41]

Muitas vezes CD* é confundido com CD (2.7), uma vez que tem a mesma sigla e possuem responsabilidades muito semelhantes. A principal diferença entre estas duas práticas, é que na entrega contínua (CD) há um passo manual antes da *release* para produção, enquanto que no contínuo *deployment* (CD*) o processo é totalmente automatizado. [40]

2.9 *Background* Tecnológico

Nas próximas subsecções são apresentadas tecnologias e ferramentas utilizadas no decorrer do estágio. Apesar de o contacto não ter sido direto com algumas das tecnologias a seguir

mencionadas, é de interesse a sua referência para melhor compreensão deste documento.

2.9.1 Jenkins

O Jenkins é uma ferramenta que foi concebida para facilitar a automação de tarefas e é utilizado na indústria para gerir e executar vários tipos de ficheiros - baterias de testes, *deployment*, gestão de ambientes, entre outros. Estas tarefas podem ser relativas à criação, teste, entrega ou *deploy* de software. Assim, o Jenkins integra vários processos da metodologia DevOps. [44] Com o Jenkins é possível acelerar todo o processo de desenvolvimento de software, uma vez que tarefas outrora manuais, são agora automatizadas com o Jenkins.

O Jenkins pode ser constituído por diversos nós ou *workers*. O nó mestre é responsável por gerir a configuração do sistema e por escalonar execuções de *builds* por nós onde essas são executadas. No caso da Stratio, o Jenkins é constituído por 3 nós.

De seguida são expostos alguns termos que serão mencionados ao longo deste documento, nomeadamente na secção 5.1.5:

- Uma *pipeline* do Jenkins é um conjunto de *plugins* que oferece suporte à implementação e integração de *pipelines* de entrega contínua no Jenkins. [45]
- Um item ou trabalho é uma entidade na interface do utilizador. [45]
- O *Step* é uma tarefa única e tem a responsabilidade de dizer ao Jenkins o que deve fazer dentro de um projeto. [46]
- A *stage* é um *step* para definir um subconjunto distinto de todo o Pipeline, que é usado por muitos *plugins* para visualizar ou apresentar o Jenkins. Um *stage* executa uma tarefa específica.
- O *workspace* é onde Jenkins cria seu projeto, contém o código-fonte e todos os arquivos gerados pela própria execução. O *workspace* é reutilizado para cada compilação, isto é, existe apenas um *workspace* por projeto. [47]
- Uma *build* é cada uma da execuções do item ou trabalho.

2.9.2 Docker

O Docker é uma ferramenta *open-source* que tem como objetivo simplificar os processos de criação, implantação e execução de aplicações baseadas em *containers*. É possível criar *containers* sem o Docker, mas este torna o processo mais simples, uma vez que o processo de criação de *container* é automatizado.[48]

Um *container* é uma tecnologia que aglomera código, bibliotecas e todas as suas dependências para a execução da aplicação em qualquer ambiente. [49]

O Docker é um virtualizador de *containers* que permite uma construção deste computacionalmente leve e rápida. Ao criar um *container* para cada processo, estes processos rapidamente podem ser partilhados com novas aplicações. Como o sistema operativo não precisa de ser inicializado para adicionar ou mover *containers*, o tempo para fazer *deployment* é consideravelmente mais curto. Esta funcionalidade também é útil quando se pretende fazer uma atualização numa parte da aplicação. Esta pode continuar a executar enquanto essa parte é retirada para atualização. [50]

O Docker tem controle de versões para as imagens docker. As imagens docker contêm o executável da aplicação e todas as dependências necessárias para ser executado como um *container*. [48] Cada vez que são feitas alterações à imagem, é criada nova camada e esta substitui a camada anterior como a nova versão da imagem. Uma vez que as camadas anteriores são guardadas é possível reverter para versões anteriores (*rollback*). Estas imagens servem muitas vezes como base para construção de novos *containers* o que acelera o processo de construção. [48]

O Docker apresenta assim funcionalidades que beneficiam equipas numa metodologia de desenvolvimento e operações (DevOps). [51] Como refere John Willis: "The capabilities of the Docker platform with containerized compute, storage and networking for distributed applications provide promising results when applied to the Three Ways of DevOps principles."[52]

2.9.3 Kubernetes

O Kubernetes, também denominado por kube ou k8s, é uma plataforma *open-source* de orquestração de *containers*. Esta plataforma automatiza muitos dos processos envolvidos no *deploying*, gestão e escalonamento de aplicações em *container*. [53] A figura 2.8 apresenta a arquitetura do Kubernetes.

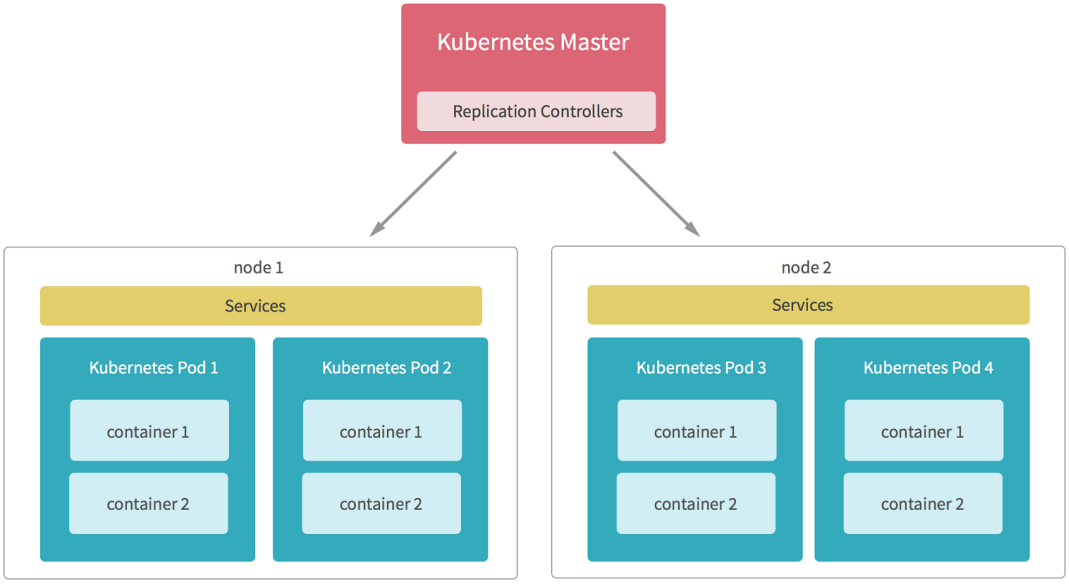

Figura 2.8: Conceitos Kubernetes (retirado de [54])

A menor unidade do Kubernetes é o *pod* que executa uma única instância de uma dada aplicação. [49] O *pod* pode conter um ou mais *containers* que partilham a interface de rede e recursos de armazenamento. A cada *pod* é atribuído um endereço *Internet Protocol* (IP) único permitindo que este comunique com outros *pods* e nós do *cluster*. [55] Para executar o *container* nos *pods* o Kubernetes usa um *container runtime*, que no caso é o Docker.

Baseado numa arquitetura *Master-Node*, o Kubernetes tem um nó *master* que é responsável por gerir os nós *worker* e gere automaticamente o agendamento dos *pods* pelos nós *worker* tendo em conta os recursos disponíveis de cada nó. [49] O nó *worker* é uma máquina virtual ou física que contém os serviços necessários para executar os *pods*. [49] O nó *master*

pode também executar os *pods* desde que seja configurado para tal. O nó *master* e um ou mais nós *worker* compõem assim o *cluster* do Kubernetes.[56]

Como já foi referido, esta plataforma permite escalar horizontalmente aplicações. O responsável por gerir tal escalonamento é o *replicaSet*. Após a criação de um *pod*, é possível que as suas instâncias sejam escaladas horizontalmente, aumentando assim os recursos da instância. [57] O *replicaSet* garante assim, em qualquer momento, que um determinado conjunto de réplicas de *pods* estejam em execução. [49]

O *deployment* é um objeto do Kubernetes que permite gerir um conjunto de *pods* idênticos ou *replicaSets*. O *deployment* tem como funcionalidade criar *pods*, garantir que estejam atualizados e que haja um número suficiente em execução. [58]

O *daemonSet* é um objecto que garante que todos ou um subconjunto de nós do *cluster* executam uma cópia de um determinado *pod*. [49]

Existem diversos objetos relacionados com esta plataforma, tendo sido só mencionados os mais relevantes para melhor perceção deste documento.

2.9.4 Apache Kafka

Apache Kafka [59] é uma plataforma de transmissão de mensagens baseada numa arquitetura *publish-subscribe*. Um produtor publica mensagens e os consumidores recebem essas mensagens. [60] É uma plataforma de transmissão distribuída, *open-source* e escrita em Scala e Java. [61]

O esquema apresentado na figura 2.9 pretende ilustrar a arquitetura do ecossistema do Apache Kafka.

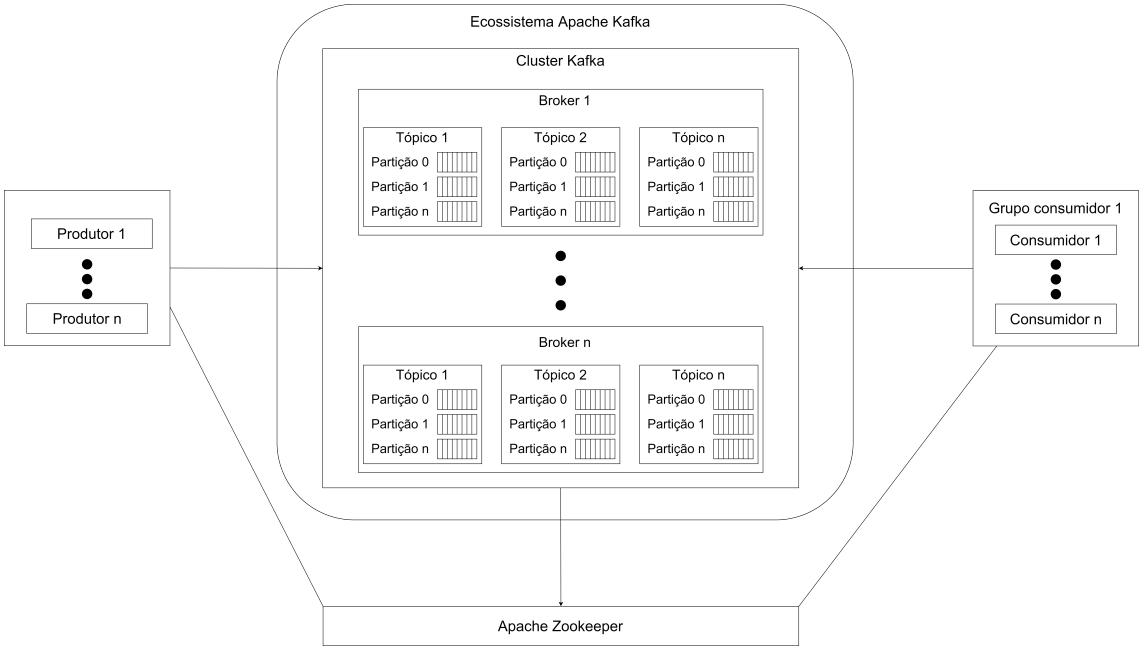

Figura 2.9: Arquitetura do Apache Kafka

O *cluster* Kafka, por norma, é constituído por diversos *brokers*. Um *broker* é o servidor Kafka e é o elo de ligação entre os produtores e consumidores, uma vez que estes não se comunicam diretamente. Um *broker* não tem estado, o responsável por controlar o seu

estado é o Zookeeper (ver 2.9.4). Um *broker* é capaz de lidar com terabytes de mensagens sem qualquer influência no seu desempenho.

De notar que um *cluster* Kafka pode ser constituído só por um *broker*, mas nesse caso não se iria tirar proveito dos diversos benefícios que um *cluster* Kafka pode oferecer, nomeadamente equilíbrio da carga e replicação de dados. [61]

Um tópico é um canal de comunicação, um produtor publica os dados para um determinado tópico e o consumidor lê de um dado tópico. Um tópico é identificado pelo seu nome e este deve ser único. Pode ser visto como uma categoria onde um dado tipo de mensagem é armazenado, isto é, um tópico recebe um certo tipo de mensagens. Não existe limitação para o número de tópicos presentes num *cluster*. Um tópico pode ser dividido em diversas partições. [62]

Uma partição é uma sequência imutável de mensagens. A cada mensagem na partição é atribuído um valor de *offset*. [63] O *offset* é um número de identificação sequencial e exclusivo de cada mensagem na partição. [59] Esta divisão dos tópicos em diversas partições, torna-o *multi-subscriber*, isto é, pode estar mais que um consumidor a ler de um tópico em paralelo. [59] O número de partições no tópico define o grau máximo de paralelismo ao ler desse tópico.

Para garantir o sequenciamento de mensagens por partição é possível adicionar uma chave à mensagem. Assim, sempre que um produtor publicar uma mensagem com uma determinada chave, esta é adicionada sempre à mesma partição. Caso não seja adicionada uma chave, as mensagens são gravadas em partições aleatoriamente. [62]

Cada partição possui um *broker* que atua como líder e pode ter um ou mais que atuam como seguidores. O líder é responsável pelos pedidos de leitura e escrita da partição, enquanto que o *broker* seguidor é responsável por replicar o líder. Quando um líder falha automaticamente um dos seguidores torna-se no novo líder. [59] Para uma equilibrada distribuição da carga, cada *broker* deve ser líder da mesma quantidade de partições. [61]

Cada partição pode ser replicada nos *brokers*, o número de vezes definido - fator de replicação. A réplica, como é denominada, é assim uma cópia de uma dada partição, um *backup*. A replicação ao nível das partições garante assim a tolerância a falhas e impede a perda de dados. [61]

O produtor é responsável por enviar as mensagens para o *broker*. Quando um novo *broker* é iniciado os produtores começam automaticamente a enviar mensagens para ele. [60]

Um vez que os *brokers* não guardam o seu estado, o consumidor deve guardar o *offset* da partição para saber quantas mensagens já consumiu. O valor do *offset* do consumidor é notificado pelo Zookeeper. O consumidor pode retroceder ou ir para qualquer ponto de uma partição, fornecendo um valor do *offset*. [60] O grupo de consumidores possui várias instâncias do consumidor em execução e é identificado por um *IDentifier* (ID) único. Como mencionado anteriormente, pode haver muitos produtores a escreverem para um dado tópico, é necessário então, que um grupo de consumidores seja capaz de lidar com um grande volume de informação mantendo a eficiência. [64] Uma instância de cada grupo apenas pode ler numa única partição. Assim o número de consumidores deve ser igual ao número partições, caso seja superior iram existir instâncias de consumidores inativos. [65]

O Zookeeper é responsável por coordenar os *brokers* no *cluster* Kafka. O Zookeeper notifica o produtor e consumidor quando um *broker* é inciado ou quando este deixa de funcionar. Pode também ter a responsabilidade de eleger o *broker* líder, isto é, em caso de falha, o Zookeeper elege o *broker* líder e o seguidor. [60]

Deste modo, o Kafka é uma ferramenta bastante complexa e poderosa capaz de lidar com grandes cargas de trabalho. É extremamente adequado ao caso de uso da Stratio, uma vez que os dados de cada peça precisam de ser processados em ordem, mas os dados de diferentes peças podem ser processados em paralelo.

Apache Zookeeper

O Apache Zookeeper [66] é uma ferramenta *open-source* que permite coordenar e gerir um serviço num ambiente distribuído. É utilizado para manter os dados de nomeação e configuração e fornece sincronização flexível e robusta nos sistemas distribuídos. No caso da sua utilização com o Apache Kafka, o Zookeeper controla o estado dos nós do *cluster* Kafka, dos tópicos, partições, etc. [67]

2.9.5 Redis

O *REmote DIctionary Server* (Redis) [68] é um armazenamento de estrutura de dados chave-valor em memória. É *open-source*, criado por Salvatore Sanfilippo e desenvolvido em ANSI C. [69]

A chave pode suportar diferentes estruturas de dados, como *strings*, *hashes*, índices geo-espaciais entre outros. [68] Permite também que uma chave expire após um determinado período de tempo definido, útil, por exemplo, para gerir sessões de utilizador. O Redis é muito rápido, tanto para leitura como para escrita de dados, uma vez que armazena os seus dados em memória. Também é capaz de manter os dados em disco, caso se pretenda, garantindo assim a persistência de dados. [70]

2.9.6 *Automated Tests Framework*

A *Automated Tests Framework* (ATF) é uma ferramenta projetada para facilitar a criação de *deployments* parciais ou completos dos diversos serviços utilizados na Stratio. A ATF foi desenvolvida pela Stratio e escrita na linguagem Python.

Esta ferramenta permite a integração de vários serviços, permitindo que todos os serviços que compõem a plataforma da empresa sejam instanciados de forma rápida e escalável. Assim o processo de *deployment*, outrora demorado, é agora automatizado, permitindo em qualquer momento criar e remover serviços.

A ATF é capaz de instanciar serviços como o Kafka, Zookeeper, Redis, MsSql entre outros, num *cluster* Kubernetes. É responsável por iniciar e configurar um *container* com uma dada imagem Docker e faz o seu lançamento num *cluster* Kubernetes.

A ATF permite a gestão de dependências, por exemplo, o Kafka depende do Zookeeper, como tal, o Zookeeper tem de ser instanciado primeiro e de seguida o Kafka será lançado com as configurações (endereço IP, porto, etc) previamente guardadas do serviço Zookeeper. Faz também a gestão de dados, isto é, permite instanciar uma base de dados com dados importados de uma outra base de dados.

A ATF possui uma *REpresentational State Transfer* (REST) API. O REST é um padrão de design para API. O REST é uma maneira de dois sistemas se comunicarem via *HyperText Transfer Protocol* (HTTP) de maneira semelhante aos *browsers* e servidores Web. [71]

A REST API da ATF tem diversos *endpoints*. Estes permitem a listagem, criação, atuali-

zação e destruição de ambientes, basta, para tal, fazer um pedido HTTP para o *endpoint* pretendido.

A ATF permite, assim, emular o funcionamento da plataforma em uso no ambiente real e ter um ambiente isolado e seguro para executar testes, diminuindo assim o risco de defeitos no produto.

2.9.7 Git

O Git é um sistema *open-source* de controle de versão distribuído. É utilizado para guardar conteúdo, normalmente código. O Git mantém um histórico das mudanças que ocorreram no código do projeto e facilmente é possível reverter para uma versão específica. Facilita a colaboração entre vários utilizadores uma vez que junta automaticamente as alterações (*merge*) quando o mesmo ficheiro é editado por mais que um utilizador. [72] O GitLab [73] foi a plataforma utilizada para gestão de repositórios Git.

2.9.8 Jira Software

O Jira Software [74] é um software para gestão de projetos. Desenvolvido pela Atlassian, tem como objetivo ajudar as equipas a gerir o trabalho. Este software é compatível com Scrum, Kanban e metodologias mistas. [74]

Na imagem 2.10 é possível observar o quadro Kanban no Jira, relativo ao projeto *Sw Foresight* em específico da equipa *Snickers* - equipa da qual a estagiária integrou.

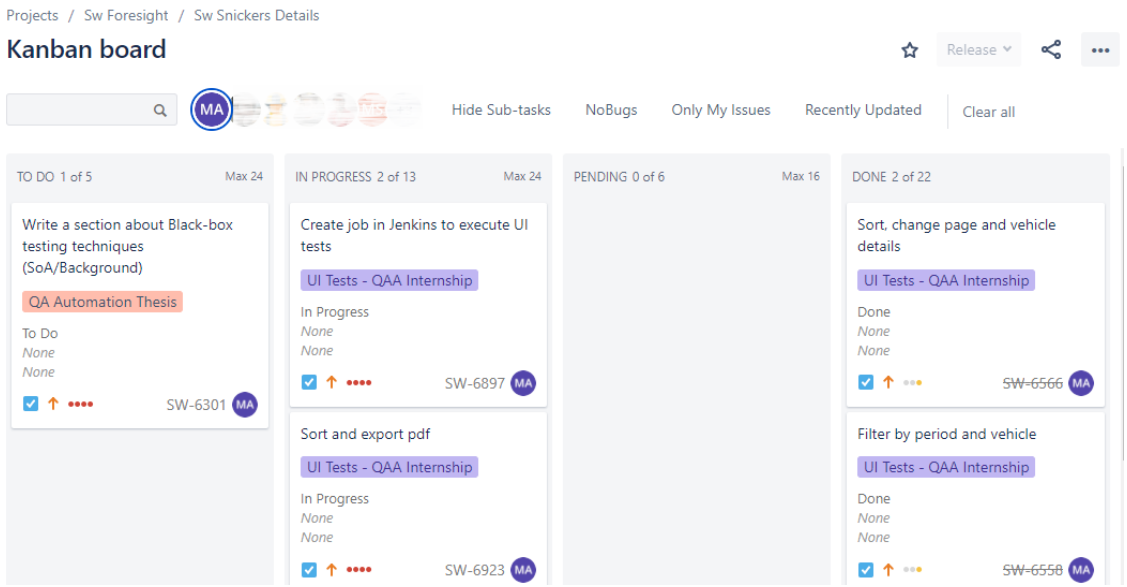

Figura 2.10: Quadro Kanban

Para a criação de um *issue* é necessário preencher diversas informações, tal como se pode observar na imagem 2.11.

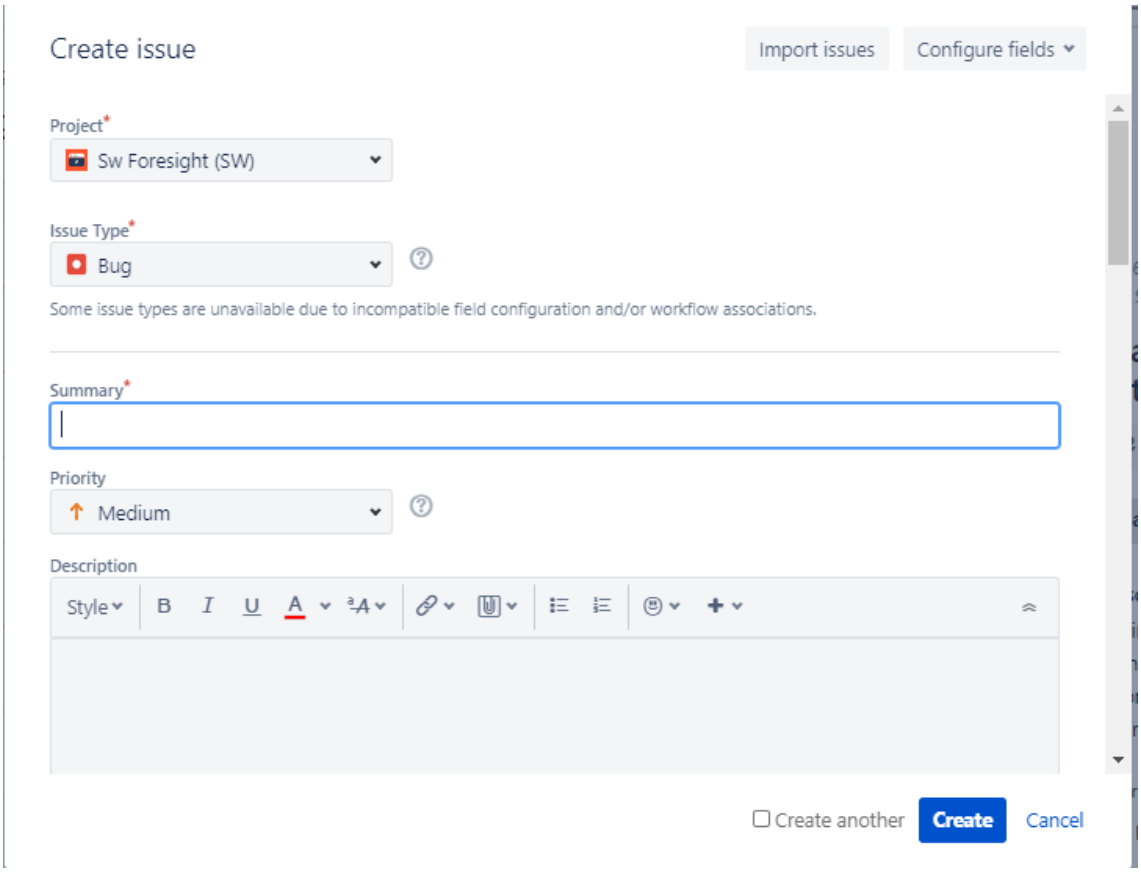

Figura 2.11: Criação de um *issue*

As informações obrigatórias e/ou mais relevantes a preencher, no caso desta equipa, são as seguintes:

- Tipo de *issue* - pode assumir cinco tipos:
  - Bug - problema que prejudica ou impede as funções do produto;
  - *Epic* - grande *user story* que necessita de ser detalhado;
  - *Spike* - tarefa para as fases de descoberta/pesquisa;
  - *Story* - *user story*;
  - *Task* - tarefa que precisa de ser realizada.
- Sumário - um título do *issue*;
- Prioridade - indica a importância do *issue*. A prioridade pode ser:
  - Crítica - A mais prioritária, por vezes uma tarefa inesperada que tem prioridade sobre as já definidas;
  - Alta - Importante e pode bloquear o progresso para outras tarefas;
  - Média - Pode ter o potencial de afetar o progresso para outras tarefas;
  - Baixa - Tarefa menor e facilmente contornada.
- Descrição - explicação detalhada do *issue*;
- Equipa - associação do *issue* a equipa pretendida;

- *Assignee* - responsável por realizar/resolver o *issue*;
- *Epic Link* - associação a um épico já criado;
- Aprovadores - utilizadores responsáveis por aprovar o *issue*.

Cada *issue* pode passar por quatro estados principais, como se pode observar pelas quatro colunas apresentadas na figura 2.10:

- *TO DO* - *issue* que aguarda a sua execução;
- *IN PROGRESS* - *issue* que está a ser executado pelo responsável a quem foi atribuído;
- *PENDING* - *issue* que está pendente de algo ou alguém antes de ir para o próximo estado. O *issue* neste estado pode dividir-se em cinco estados menores:
  - *For approval* - *issue* à espera de aprovação;
  - *Blocked* - *issue* impedido de ser concluído;
  - *Feedback needed* - *issue* à espera de parecer;
  - *In Test* - *issue* em fase de teste;
  - *Standby* - *issue* em espera.
- *DONE* - *issue* concluído.

Existem mais três estados que não é possível observar no quadro kanban: *backlog* - *issue* que será executado num futuro, *deployed* - *issue* que foi introduzido em produção e *cancelled* - *issue* interrompido e considerado concluído.

Os *issues* criados no âmbito deste estágio são apresentados no anexo E, divididos em três épicos, *Jam.API Performance Validation*, referente à automatização do testes de desempenho; o *End-to-end Tests - QA Automation*, referente à automatização dos testes de UI; e o *QA Automation Internship Report* referente às tarefas de documentação.

2.9.9 Confluence

O Confluence [75] é uma ferramenta de colaboração de conteúdo. Desenvolvida pela Atlassian, tem como objectivo ajudar as equipas a partilhar conhecimento. Os utilizadores podem criar, editar e comentar de forma colaborativa na interface do Confluence. Esta ferramenta permite assim que toda a documentação fique organizada e concentrada num único local. O Confluence pode ser integrado com o Jira Software (subsecção 2.9.8), permitindo assim que na interface do Confluence seja possível visualizar, interagir e referir os *issues* presentes no Jira. [76]

2.9.10 Clockify Time Tracker

O Clockify Time Tracker [77] é uma ferramenta que permite registar o tempo gasto em cada tarefa. O Clockify apresenta uma versão Web que permite ver em detalhe as tarefas registadas, o tempo dedicado a cada uma e ainda relatórios resumo que podem ser filtrados por diversas categorias.

Possui também uma extensão para o Chrome e adiciona o cronómetro a diversos serviços, inclusive o Jira Software, o que permite registar o tempo dedicado a cada *issue*. A utilização desta ferramenta revelou-se adequada para controle pessoal do tempo dedicado a cada *issue*.

2.10 Arquitetura e Processos Stratio

De seguida é apresentada a arquitetura geral da solução da plataforma de software da empresa. Posteriormente, é exposto o CVDS e o fluxo de trabalho do Git postos em prática na empresa. Tal contexto é essencial para perceber onde é que os testes se vão integrar no CVDS. A maneira como se garante a qualidade do software está fortemente dependente dos processos de desenvolvimento utilizadas num projeto de software. Como tal, é necessário perceber tais processos, uma vez que os próprios processos definidos para QA estão associados aos processos de desenvolvimento, podendo acelerar e melhorar ou travar e piorar o desenvolvimento consoante a boa ou má definição destes processos.

2.10.1 Arquitetura da Stratio

A solução da plataforma de software da empresa é *Software as a service* (SaaS), isto é, o *deployment* é gerido pela empresa na *cloud*.

O software é *rolling release*, software que se encontra em constante desenvolvimento e há apenas uma versão que é suportada que é a versão mais recente.

De maneira a contextualizar o leitor, a arquitetura geral da solução da plataforma de software da empresa é exposta no anexo confidencial F.

Uma API é um conjunto de rotinas, protocolos e ferramentas para criar aplicações de software. [78] Uma API permite que o serviço comunique com outros serviços e produtos, sem ser necessário perceber como estes foram desenvolvidos. Portanto, facilita o processo de desenvolvimento de aplicações, reduzindo a quantidade de código e economizando tempo aos programadores. [79] A Jam API (ver subsecção 5.1.1) foi desenvolvida pela Stratio e é responsável por receber e enviar mensagens de/e para as peças. Após a receção de mensagens, a Jam API escreve as mensagens para o tópico Kafka (ver 2.9.4) "api-all-requests". As mensagens com leituras de *Parameter IDs* (PIDs) e *Data Trouble Codes* (DTC) são também mantidas em filas na API, onde, após algum tempo, serão escritas para o tópico Kafka "api-messages".

Posteriormente as mensagens são processadas pelos serviços de Processamento de Dados. A natureza e complexidade dos dados exigiam que o processamento fosse dividido em vários Serviços de Processamento de Dados: *Alerts*, DTC, *Vehicle Info*, *Vehicle Events*, *Notifications*, *Vehicle State* entre outros. Assim cada serviço processa os dados de uma determinada forma consoante a sua funcionalidade. Esta separação resultou num aumento do desempenho uma vez que o processamento de diferentes informações é feito em paralelo.

Por último, os dados são guardados em quatro tipo de *data stores* (MS SQL Server, Elasticsearch, Redis e *Network File System* (NFS)) e utilizados por outros serviços que necessitem desses dados.

2.10.2 Ciclo de Vida de Desenvolvimento de Software da Stratio

A equipa de desenvolvimento pratica a metodologia Scrum. Scrum é uma metodologia iterativa e incremental, é um tipo de metodologia ágil e como tal segue os princípios do manifesto ágil. [2] Na figura 2.12 pode observar-se um diagrama da metodologia Scrum. Na fase de *sprint planning* são planeadas as tarefas de *backlog*, tarefas criadas pelo *product owner*. O *product owner* é o elo de ligação entre a equipa de desenvolvimento e os clientes. É ele que transmite à equipa de desenvolvimento os novos recursos a serem implementados

e os prioritiza. É responsável por esclarecer qualquer detalhe para que a equipa possa desenvolver um novo recurso conforme o desejado pelo cliente. É também o responsável pela aprovação das entregas finais.

A equipa faz iterações Scrum com duração de duas semanas. No final da *sprint* é criada uma *release* que é *deployed* para um ambiente de UAT e após aprovação do *product owner* é feito o *deployment* para produção.

Durante este ciclo são realizadas reuniões diárias em que o objetivo é que cada elemento da equipa de software exponha as tarefas que tem realizado e quais planeia fazer de seguida. No final de cada *sprint* é realizado um *review*, momento em que a equipa de software apresenta à restante empresa o que desenvolveu ao longo das duas semanas e identificar possíveis problemas e formas de o mitigar.

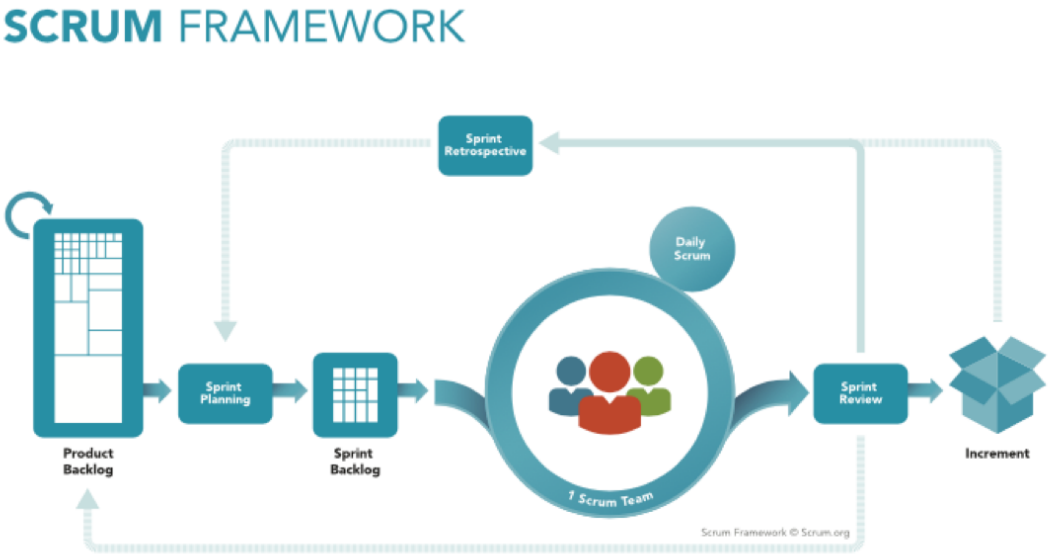

Figura 2.12: Diagrama da metodologia Scrum (retirado de [80])

2.10.3 Fluxo de trabalho do Git

O fluxo de trabalho do Git seguido na empresa é baseado no Gitflow. O Gitflow é um design de fluxo de trabalho publicado por Vincent Driessen. Este fluxo de trabalho define um modelo de ramificação, atribui funções muito específicas a cada ramo, estabelecendo como e quando é que eles interagem. [81] Um ramo, do inglês *branch*, é um ponteiro para as alterações realizadas nos ficheiros do projeto. Quando se pretende fazer uma alteração, seja corrigir um bug ou adicionar uma nova funcionalidade, é conveniente que as alterações sejam feitas num novo *branch* e depois integradas no *branch* principal. Deste modo, o código instável a ser escrito fica isolado sem afetar outros *branches*. [82]

O modelo utilizado na empresa é baseado no Gitflow, sendo que a empresa só emprega alguns dos *branches* definidos deste fluxo.

Como se pode observar na imagem 2.13 existem três *branches*: o *master*, o *develop* e *feature*. O *master* é o *branch* ativo do projeto e é contém o código que esta em produção. No caso Stratio, o *master* é equivalente ao *branch release*.

O *develop* é o *branch* de desenvolvimento, é o mesmo que o *master* mas é utilizado para o desenvolvimento de novas *features*.

A *feature* serve para novos recursos a implementar no projeto, primeiramente o *branch* de desenvolvimento é clonado e a implementação é feita neste. Quando a implementação da *feature* acabar o código é integrado (*merged*) para o *branch* de desenvolvimento. O nome da *feature* deve ser identificado pelo ID da tarefa no Jira e por uma breve descrição.

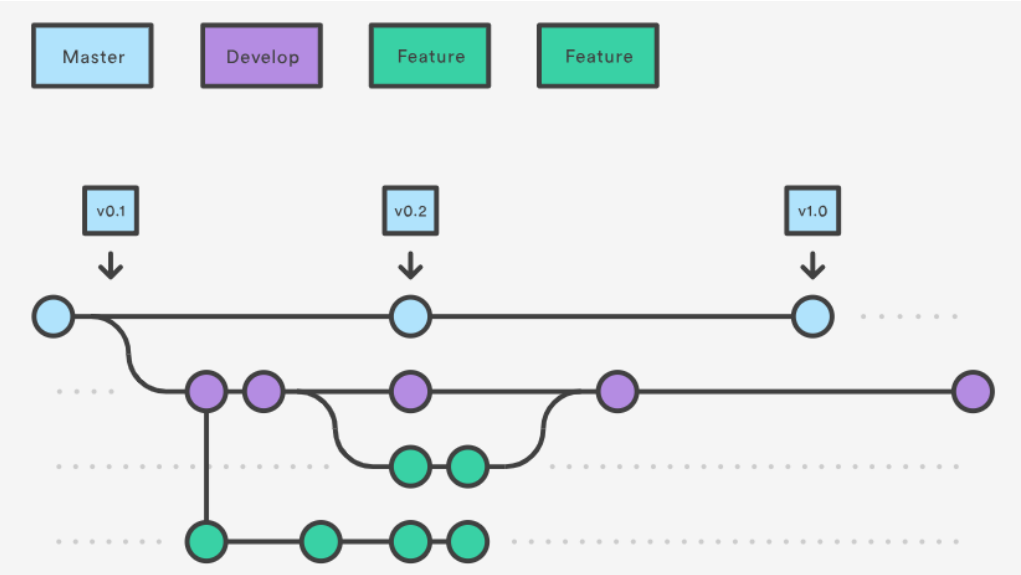

Figura 2.13: Esquema representativo do fluxo de trabalho Git (retirado de [81])

Por vezes, é utilizado outro *branch*, o *hotfix*. Quando é preciso resolver algum problema crítico no código de produção, é criado um *hotfix* a partir do *master*. Assim que a correção estiver concluída e aprovada por um *product owner* é feito *deployment* para produção. Este *branch* é exclusivamente dedicado a correções de bugs e permite que a equipa resolva o problema sem interromper o restante fluxo de trabalho. Este é único *branch* que deriva diretamente do *master*, mas é necessário um cuidado extra ao interagir diretamente com este, uma vez que é o código que esta em produção.

Esta página foi propositadamente deixada em branco.

Capítulo 3

Requisitos e Análise de Riscos

Este capítulo inicia-se pela listagem dos requisitos funcionais e não funcionais do sistema a implementar no decorrente estágio e os requisitos técnicos.

De seguida, são elencados os requisitos necessários para as ferramentas a utilizar para a injeção de mensagens e automatização de testes *end-to-end* a uma *User Interface* (UI). Ainda não tinha sido tomada qualquer decisão de qual ferramenta utilizar, e a definição de requisitos permitiu um levantamento do estado da arte com foco específico nas funcionalidades que as ferramentas deviam apresentar.

Posteriormente são expostos os requisitos de uma das componentes alvo de testes - Jam *Application Programming Interface* (API).

Por último, são identificados os riscos associados ao projeto, bem como um plano de mitigação.

3.1 Requisitos Funcionais

Os requisitos funcionais descrevem quais as funções que o produto a desenvolver, durante o estágio, deverá exercer. Nesta secção são listados os requisitos funcionais, estes são identificados por um *IDentifier* (ID), classificados por ator e a respectiva prioridade. Estes requisitos estão ainda divididos consoante o seu épico, API Perf Test e E2E Tests, referente a automatização dos testes de desempenho da componente Jam API e a automatização dos testes *end-to-end* a uma UI, respectivamente.

3.1.1 Atores

São considerados três atores - programador de software, operador de software e *product owner* -, divididos pelos dois épicos a realizar, uma vez que a sua função difere consoante os épicos.

Em relação ao épico API Perf Test:

- O programador de software tenciona saber se as alterações que produziu na componente Jam API permanecem em conformidade com os requisitos da componente;
- O operador de software deseja entender quais são os recursos necessários alocar para a componente de modo a respeitar os requisitos definidos para esta;

- O *product owner* pretende perceber se há ou não necessidade de intervir e onde. Isto é, quais são as alterações necessárias de modo a escalar o número de clientes.

Em relação ao épico E2E Tests:

- O programador de software tenciona saber se o código que produziu está em conformidade com a bateria de teste definida, e caso contrário, perceber onde está a falha;

- O *product owner* pretende perceber se existem condições para uma nova *release* consoante o sucesso ou insucesso da bateria de testes.

3.1.2 Prioridade

Para classificar os requisitos funcionais, quanto à sua prioridade, recorreu-se ao método de MoSCoW [83]:

- ***Must have*** - requisitos cruciais para o sucesso do projeto;

- ***Should have*** - requisitos que são importantes para o projeto mas que não são essenciais;

- ***Could have*** - requisitos que são desejáveis de ter no projeto mas não são necessários;

- ***Won't have*** - requisitos que foram pensados mas não serão implementados neste estágio, podendo no entanto ser tidos em conta num trabalho futuro.

Os requisitos funcionais são apresentados na tabela 3.1.

ID	Descrição	Ator	Prioridade
	API Perf Test		
RF.1	A bateria de testes deve ser capaz de gerar um relatório de desempenho para cada execução, que inclua informações do mínimo de utilização de *Central Processing Unit* (CPU) e *Random Access Memory* (RAM), taxa de transferência e tempo de resposta obtidos	Todos	*Must have*
RF.2	A bateria de testes deve ser capaz de validar os requisitos da componente Jam API descritos na tabela 3.5	Todos	*Must have*
RF.3	A bateria de testes deve ser incluída no Ciclo de Vida de Desenvolvimento de Software (CVDS), no sentido de dar feedback aquando alterações no código base da API	Programador de software	*Could have*
RF.4	A bateria de testes deve ser passível de ser executada contra qualquer *branch* da Jam API através de uma *pipeline* do Jenkins	Programador de software	*Should have*
	E2E Tests		
RF.5	A bateria de testes deve ser capaz de validar de forma automática os *user flow* definidos na subsecção 5.2.1	Programador de software e *Product owner*	*Must have*
RF.6	A bateria de testes deve ser capaz de reportar o sucesso ou não do teste e indicar onde ocorreu a falha	Programador de software e *Product owner*	*Must have*
RF.7	A bateria de testes deve ser passível de ser executada contra qualquer *Uniform Resource Locator* (URL) numa *pipeline* do Jenkins	Programador de software e *Product owner*	*Should have*
RF.8	A bateria de testes deve ser incluída no CVDS, no sentido de dar feedback de forma contínua	Programador de software e *Product owner*	*Could have*

Tabela 3.1: Requisitos Funcionais

3.2 Requisitos Não Funcionais

Os requisitos não funcionais definem os atributos de qualidade de um produto, isto é, quais as características que o produto deve ter. Os atributos de qualidade vão ser classificados por prioridade no seguinte formato (impacto na arquitetura, valor de negócio), com a escala:

- A - alta;
- M - média;
- B - baixa.

Na tabela 3.2 são enumerados os atributos de qualidade e, de seguida, o diagrama da figura 3.1 apresenta a árvore de utilidade de modo a organizar e priorizar os atributos de qualidade.

Característica	Descrição	Prioridade
Funcionalidade	Capacidade que o sistema tem de responder às necessidades a que foi proposto fazer	A,A
Confiabilidade	Capacidade que o sistema tem de manter o seu desempenho sob determinadas condições por um determinado período de tempo	M,A
Usabilidade	Capacidade que o sistema tem de ser compreendido pelos utilizadores e facilmente utilizável	M,A
Eficiência	Capacidade que o sistema tem de executar determinadas funcionalidades sob determinadas circunstâncias num espaço de tempo	M,A
Manutenção	Capacidade que um sistema tem de estar sujeito a mudanças, isto é, o esforço necessário para fazer modificações	A,M
Portabilidade	Capacidade que um sistema tem de executar num determinado ambiente	A,M
Auditabilidade	Capacidade que o sistema tem de ser monitorizado	B,A
Escabilidade	Capacidade do sistema de lidar com o aumento exponencial da sua atividade mantendo o seu desempenho	A,A

Tabela 3.2: Requisitos não funcionais

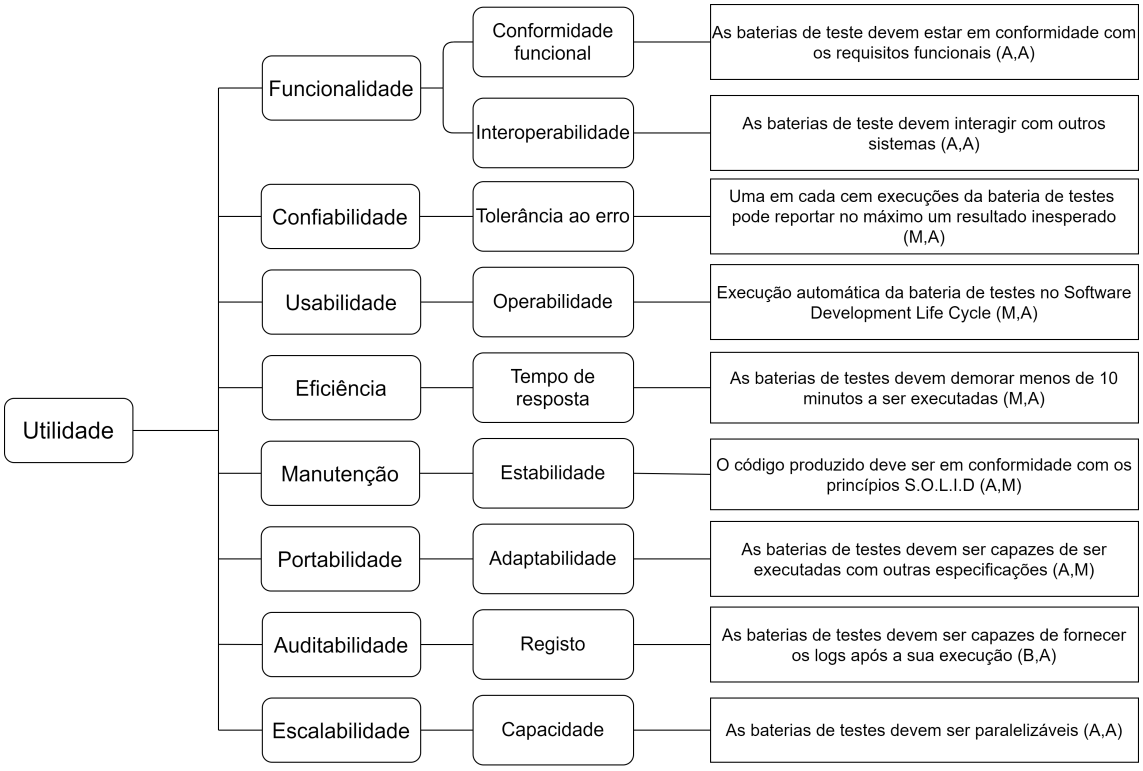

Figura 3.1: Árvore de utilidade

3.3 Requisitos Técnicos

Os requisitos técnicos referem-se ao *background* técnico necessário para o desenvolvimento deste projeto, tal como, a linguagem de programação, servidores, o sistema operativo e ainda os padrões que o código deve seguir.

Na tabela 3.3 são enumerados os requisitos técnicos. Mais uma vez, estes são identificados por um ID, por uma categoria e por uma breve descrição desta.

ID	Categoria	Descrição
RT.1	Linguagem de Programação	O código deve ser desenvolvido em Groovy, Python e JavaScript
RT.2	Sistema Operativo	Os sistemas operativos a utilizar são Windows e/ou Linux
RT.3	Ferramentas *open-souce*	As ferramentas a utilizar são: Jenkins, Kubernetes e Cypress
RT.4	Sistema de controle de versões	GitLab [73]
RT.5	Princípio	O princípio a seguir é *Single Responsibility Principle, Open/Closed Principle, Liskov Substitution Principle, Interface Segregation Principle, Dependency Inversion Principle* (SOLID) [84]

Tabela 3.3: Requisitos Técnicos

Alguns dos requisitos anteriormente apresentados, podem ser vistos como restrições. Por exemplo, a escrita de código em Groovy é restringida uma vez que é a linguagem que o Jenkins utiliza.

O requisito RT.5, relativo ao princípio SOLID, contém cinco importante princípios de seguida apresentados:

- **S** - princípio da responsabilidade única: uma classe deve ser especializada num único assunto e possuir apenas uma responsabilidade dentro do software;
- **O** - princípio aberto-fechado: os objetos ou entidades devem estar abertos para extensão, mas fechados para modificação, isto é, quando é necessário adicionar novos comportamentos ou recursos ao software, devemos estender e não alterar o código original;
- **L** - princípio da substituição de Liskov: uma classe derivada deve ser substituível pela sua classe base;
- **I** - princípio da segregação da interface: uma classe não deve ser forçada a implementar interfaces e métodos que não irá utilizar;
- **D** - princípio da inversão da dependência: depender de abstrações e não de implementações, os módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender da abstração. [84]

3.4 Requisitos da ferramenta para injeção de mensagens

A ferramenta para injeção de mensagens, denominada *Message Injector* (MI) irá ser a ferramenta responsável pela injeção de mensagens na Jam API no âmbito da automatização dos testes de desempenho (épico *API Perf Tests*). Como tal, primeiramente definiu-se quais as funcionalidades imprescindíveis que o MI devia apresentar, mais uma vez classificados por um ID e por prioridade. Estes requisitos podem ser observados na tabela 3.4.

ID	Descrição	Prioridade
RMI.1	O MI deve permitir injetar no mínimo uma carga de 500 mensagens por segundo	*Must have*
RMI.2	O MI deve permitir a personalização do *dataset*, isto é as mensagens devem ser no formato *JavaScript Object Notation* (JSON)	*Must have*
RMI.3	O MI deve garantir que não injeta mais do que uma mensagem ao mesmo tempo por peça (Stratio *Data-box*) e que as mensagens seguem a ordem *First In First Out* (FIFO) por peça, isto é, a ordem das mensagens por peça deve ser crescente no parâmetro temporal da mensagem	*Must have*
RMI.4	O MI deve injetar pedidos *HyperText Transfer Protocol* (HTTP)	*Must have*
RMI.5	O MI deve injetar pedidos *HyperText Transfer Protocol Secure* (HTTPS)	*Should have*
RMI.6	O MI deve conseguir ser integrado com uma ferramenta de *Continuous Integration* (CI)	*Must have*
RMI.7	O MI não deve ter qualquer custo associado	*Must have*

Tabela 3.4: Requisitos do MI

3.5 Requisitos da ferramenta para automatização de testes *end-to-end* a uma *user interface*

O único requisito que a ferramenta para automatização de testes *end-to-end* a uma UI deve apresentar, é a capacidade de testar uma UI em Angular. Este requisito tem prioridade *must have*.

3.6 Requisitos da componente Jam *Application Programming Interface*

Os requisitos de seguida enunciados, dizem respeito à componente da arquitetura contra à qual a bateria de testes é executada - Jam API.

Os requisitos foram divididos em dois grupos, os requisitos funcionais e os requisitos de desempenho (não funcional). Cada requisito é identificado por um ID e classificado por prioridade, seguindo, mais uma vez, o método de MoSCoW (ver subsecção 3.1.2). Os requisitos podem ser observados na tabela 3.5.

ID	Descrição	Prioridade
	Requisitos Funcionais	
RF.1	A ordem de chegada das mensagens à Jam API tem de ser com data e hora da mensagem (dt) crescente por peça	*Must have*
RF.2	Não existe perda de mensagens	*Must have*
	Requisitos de Desempenho	
RD.1	Submetida a uma carga de 50 pedidos por segundo, a Jam API dever ser capaz de responder a estes pedidos com um tempo de resposta igual ou inferior a 100 milissegundos para pelo menos 99.9% dos pedidos recebidos	*Must have*
RD.2	Submetida a uma carga de 50 pedidos por segundo, a API dever ser capaz de responder a estes pedidos com um tempo de resposta igual ou inferior a 1000 milissegundos para pelo menos 99.99% dos pedidos recebidos	*Must have*
RD.3	Submetida a uma carga de 250 pedidos por segundo, a API dever ser capaz de responder a estes pedidos com um tempo de resposta igual ou inferior a 100 milissegundos para pelo menos 99.9% dos pedidos recebidos	*Must have*
RD.4	Submetida a uma carga de 250 pedidos por segundo, a API dever ser capaz de responder a estes pedidos com um tempo de resposta igual ou inferior a 1000 milissegundos para pelo menos 99.99% dos pedidos recebidos	*Must have*
RD.5	Submetida a uma carga de 500 pedidos por segundo, a API dever ser capaz de responder a estes pedidos com um tempo de resposta igual ou inferior a 250 milissegundos para pelo menos 99.9% dos pedidos recebidos	*Must have*
RD.6	Submetida a uma carga de 500 pedidos por segundo, a API dever ser capaz de responder a estes pedidos com um tempo de resposta igual ou inferior a 10000 milissegundos para pelo menos 99.99% dos pedidos recebidos	*Must have*

Tabela 3.5: Requisitos da componente Jam API

3.7 Riscos

Esta secção visa pormenorizar o plano de gestão de riscos e o limiar de sucesso no âmbito do estágio.

O limiar de sucesso (*Threshold of Success* (ToS)) de um projeto é uma condição ou conjunto de condições que devem ser atingidas para que o projeto possa ser considerado bem sucedido.

A gestão de riscos permite compreender quais são os problemas que podem levar ao insucesso do projeto bem como definir um plano de mitigação para cada risco. As tarefas de gestão de riscos devem ser regulares e, portanto, sujeitas a alterações frequentes. Por este motivo, nas subsecções seguintes, serão apresentadas as duas iterações efetuadas no decorrer do estágio, garantindo, deste modo que as tarefas de gestão de riscos sejam contínuas. Para cada iteração será construída a matriz de exposição de risco. Consideram-se quatro níveis de exposição: baixo (verde), médio (amarelo), elevado (laranja) e crítico (vermelho). Por uma questão de simplicidade, a cada risco será associado um ID. Para cada risco irá ser exposto um plano de mitigação, caso seja exequível.

3.7.1 Limiar de Sucesso

A condição definida para atingir o limiar de sucesso, é a entrega de todos os requisitos com a prioridade *must have*.

3.7.2 Primeira iteração

A primeira iteração situou-se entre o início do estágio e a submissão intermédia do relatório (setembro - janeiro). Na figura 3.2 apresenta-se a matriz de exposição de risco para a primeira iteração.

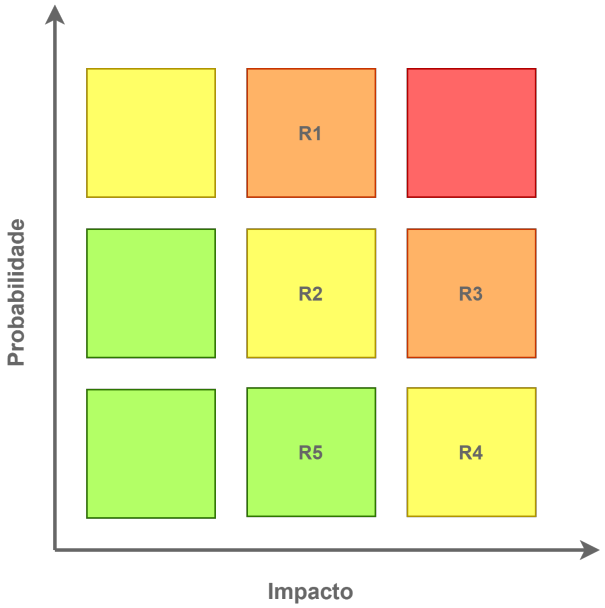

Figura 3.2: Matriz de exposição de risco - primeira iteração

R.1 - Mau planeamento

O planeamento pode não ter em conta todas as tarefas necessárias para o cumprimento dos requisitos. As estimativas calculadas para cada tarefa podem não ser as indicadas, certas tarefas podem demorar mais tempo a realizar do que o calculado nas estimativas. No primeiro semestre, o planeamento definido pode não ter em conta o trabalho de outras unidades curriculares frequentadas pela aluna.

Plano de contingência: Alocar mais esforço a unidade curricular "Dissertação/Estágio". Conciliar o esforço dedicado as outras unidades curriculares frequentadas pela aluna de modo a não prejudicar nenhuma destas.

R.2 - Alterações aos requisitos

Requisitos mal definidos e/ou não compreendidos, e possíveis alterações significativas aos requisitos podem causar sérias preocupações ao longo do estágio.

Plano de contingência: Efetuar diversas iterações de revisão dos requisitos, de modo a clarificar a compreensão destes e apurar se se encontra algum em falta.

R.3 - Conhecimento anterior

A aluna não tem experiência anterior com as tecnologias a utilizar durante o estágio e desconhece a lógica de negócio dos serviços, as suas arquiteturas e dependências.

Plano de contingência: Recolha de informação acerca das tecnologias.

R.4 - Recursos computacionais

Durante o horário de expediente, isto é, quando a equipa estiver a usar os recursos disponíveis, pode não ser possível executar a bateria de testes.

Plano de contingência: Executar a bateria de testes fora do horário de expediente e/ou alocar mais RAM

R.5 - Empresa

Possíveis mudanças na empresa e alterações de orientador da empresa, podem causar instabilidade no decorrer do estágio.

3.7.3 Segunda iteração

A segunda iteração situou-se entre a submissão intermédia e a submissão final do relatório (fevereiro - setembro). Na figura 3.3 apresenta-se a matriz de exposição de risco para a segunda iteração.

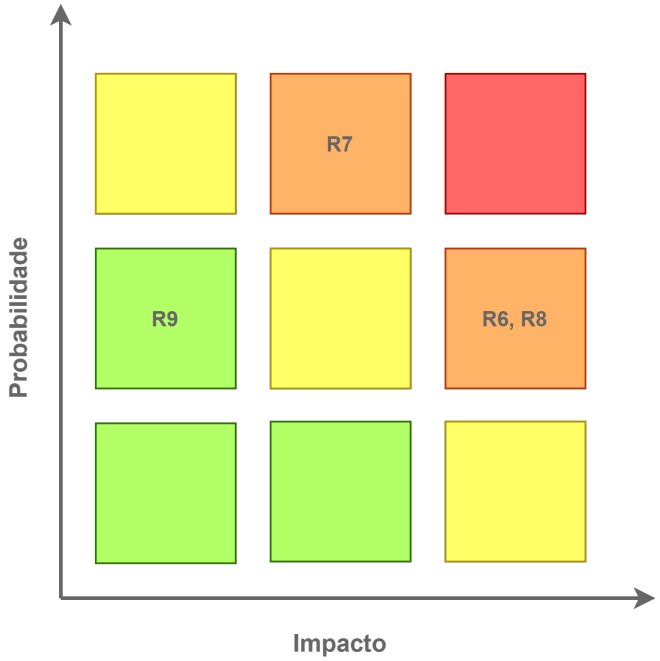

Figura 3.3: Matriz de exposição de risco - segunda iteração

R.6 - Alterações ao *scope* do trabalho

O *scope* do trabalho poderá sofrer alterações que levará ao mau planeamento das novas tarefas e consequentes atrasos.

Plano de contingência: Planear e considerar as novas tarefas minuciosamente.

R.7 - Falta de conhecimento/experiência com as tecnologias da empresa

O elevado número de tecnologias já utilizadas pela empresa (Docker, Jenkins, Kubernetes, Kafka, Git), com as quais a aluna não tem experiência e terá de ter interação, poderá levar a uma forma incorreta ou menos eficiente de interagir e integrar estas tecnologias. Consequentemente, esta dificuldade, poderá gerar atrasos muito significativos na entrega.

Plano de contingência: Leitura detalhada de conceitos relativos a estas tecnologias.

R.8 - Tecnologias escolhidas

As tecnologias escolhidas para a realização deste estágio, para além de a aluna não ter experiência/conhecimento, podem não ser as mais adequadas. Consequentemente, a qualidade do trabalho poderá ser diminuta face ao uso de outras possíveis tecnologias

Plano de contingência: Levantamento do estado da arte rigoroso, de modo, a que as tecnologias escolhidas sejam as mais adequadas.

R.9 - Incumprimento da prática da metodologia

A falta de experiência a trabalhar com a metodologia praticada pela equipa que a aluna integrou (Kanban), poderá levar ao incumprimento de certos processos.

Plano de contingência: A atribuição do orientador da empresa para contextualizar sobre os processos utilizados na empresa.

3.7.4 Riscos verificados

Dos riscos apresentados anteriormente, materializaram-se os riscos: R.1, R.2, R.3 e R.5 no primeiro semestre, e no segundo semestre o R.6 e R.7.

O R.1 verificou-se e mostrou ter um grande impacto no projeto, a pobre clarificação dos objetivos da proposta e consequentemente o mau planeamento das tarefas, levou a mudanças e atrasos significativos deste projeto.

O R.2 referente a mudança de requisitos, verificou-se após o primeiro semestre, devido as mudanças no *scope* do projeto.

O R.3 relativo a falta de conhecimento da arquitetura e processos da Stratio verificou-se e teve bastante impacto no projeto, tendo só sido mitigado no segundo semestre.

O R.5 ocorreu devido as mudanças de orientador.

O R.6 verificou-se após a defesa intermédia, contudo o planeamento inicialmente definido para as tarefas não diferenciou significativamente do planeamento realizado.

O R.7 referente ao elevado número de tecnologias a utilizar, verificou-se mas foi mitigado com leitura detalhada da documentação dessas tecnologias.

Esta página foi propositadamente deixada em branco.

Capítulo 4

Estado da arte

Neste capítulo proceder-se-á à exposição do levantamento do estado da arte. O primeiro refere-se a tecnologias de *Continuous Integration* (CI)/*Continuous Delivery* (CD), que apesar de a ferramenta já estar escolhida pela empresa, foi útil perceber a evolução neste mercado e que funcionalidades outras ferramentas poderiam oferecer.

O segundo refere-se a virtualização, comparando máquinas virtuais a *containers*, de modo a perceber, qual fará mais sentido utilizar no âmbito de virtualização de ambientes para teste.

O terceiro estado da arte é referente a escolha da ferramenta para injeção de mensagens tendo em conta os requisitos definidos anteriormente.

Por último, foi feito um estado na arte em tecnologias que permitam a automatização de testes *end-to-end* a uma *User Interface* (UI) e a respectiva escolha da mais adequada.

4.1 Tecnologia de Integração contínua e contínuo *deployment*

Existem diversas ferramentas de CI/CD. De seguida proceder-se-á à exposição de algumas das ferramentas, será feita uma comparação entre estas por forma a decidir qual é a que mais se adequa às necessidades da empresa.

As ferramentas escolhidas para esta análise foram o Jenkins, o Jenkins X e o Travis CI. [85] O Jenkins e o Travis CI são a primeira e segunda, respetivamente, da lista de melhores ferramentas CI, e são as que detém mais utilizadores. Embora o Jenkins X não conste do topo da lista apresentada em [85], considerou-se que é uma ferramenta com potencial interesse neste contexto, uma vez que é uma solução muito recente (introduzida em 2018) o que dá algumas garantias sobre a sua manutenção durante os anos vindouros e facilita o processo de gestão no Kubernetes - a plataforma utilizada pela empresa para orquestração de ambientes.

4.1.1 Jenkins

O Jenkins [86] é uma ferramenta *open-source* de CI/CD. Atualmente é uma das ferramentas mais popular com uma vasta comunidade (mais de um milhão de utilizadores [87]). O Jenkins é gratuito e possui uma grande quantidade de *plugins*, o que o torna uma ferramenta bastante flexível, sendo ainda possível criar um *plugin* e partilhá-lo com a restante

comunidade. Graças aos seus *plugins*, é compatível com Docker, Kubernetes e muitas outras plataformas. Esta ferramenta permite criar e testar projetos de software, é desenvolvida em Java, e facilita atividades de teste em tempo real. Esta ferramenta destaca-se por fornecer suporte a *pipelines* desenvolvidas na linguagem Groovy ([88]) e permite definir vários estados de uma versão compilada do software ou partes deste. Maioritariamente é instalado em máquinas locais (*on-premises*), mas também pode ser executado em servidores na *cloud*. Um ponto crucial que o Jenkins oferece é o facto de se puder configurar várias máquinas (*workers*). Isto permite ter várias máquinas com diferentes sistemas operativos (Windows, Linux e macOS). Para cada tarefa pode ser definido em que máquina esta vai ser executada, distribuindo assim o trabalho pelas várias máquinas.

Uma das principais fraquezas apontadas a esta ferramenta é a documentação insuficiente, principalmente informação acerca da criação de *pipelines*, o que leva a uma curva de aprendizagem acentuada. Outra fraqueza mencionada é em relação a UI, pois é desatualizada e confusa. [89]

4.1.2 Jenkins X

O Jenkins X [90] é uma ferramenta *open-source* baseada em Jenkins e Kubernetes [91].

Uma das principais vantagens desta ferramenta é o facto de simplificar problemas causados pela complexidade do Kubernetes. Por exemplo, se necessitarmos de um *cluster* Kubernetes, o Jenkins X configura e instala-o, sem necessidade de compreender a complexidade deste processo.

Esta ferramenta define aspetos do ciclo de vida de desenvolvimento de software, toma decisões, e é considerada uma ferramenta de opinião. Contudo, ao mesmo tempo é flexível de modo a permitir aos utilizadores ajustar a ferramenta às suas necessidades.

O Jenkins X utiliza as capacidades identificadas por Nicole Forsgren, Jez Humble e Gene Kim, no livro "Accelerate":

- Utiliza controle de versões para todos os artefactos;
- Automatiza o processo de *deployment*;
- Implementa CI e CD;
- Utiliza uma arquitetura fracamente acoplada. [92]

De notar que esta ferramenta não tem uma interface gráfica, a gestão é feita exclusivamente pela linha de comandos.

4.1.3 Travis CI

Travis CI [93] é uma ferramenta *open-source* que sincroniza com o GitHub [94]. Esta solução é gratuita para projetos *open-source* no GitHub e para as primeiras 100 compilações; a partir daí o preço pode variar entre 65 € - 410 € por mês. Esta ferramenta suporta várias linguagens, mas apenas suporta dois sistemas operativos, o Linux e o macOS. Contudo, a documentação alerta que poderá haver indisponibilidade de certas funcionalidades quando os dois sistemas operativos estão ativos em simultâneo. [89] Tal como o Jenkins, esta ferramenta pode ser instalada *on-premisses* ou na *cloud*, e é fácil de instalar e de configurar.

A UI é responsiva e é apontada como ponto forte desta ferramenta. As suas *pipelines* são desenvolvidos em *YAML Ain't Markup Language* (YAML).

Os principais pontos fracos mencionados é o facto de não suportar CD e só sincronizar com o GitHub.

4.1.4 Notas Finais

A tabela 4.1 pretende comparar as três ferramentas apresentadas anteriormente.

Funcionalidade/Ferramenta	Jenkins	Jenkins X	Travis CI
Custo	Gratuito	Gratuito	65€ - 410€/mês
Sistemas Operativos	Windows, Linux e macOS	Windows, Linux e macOS	Linux e macOS
Hosting	*On-premise/Cloud*	*On-premise/Cloud*	*On-premise/Cloud*
Interface Gráfica	Sim	Não	Sim
Linguagem *pipelines*	Groovy	YAML	YAML
Empresas que usam	Facebook, Netflix	Letgo	Lyft, MIT

Tabela 4.1: Análise comparativa das ferramentas

Como é possível observar o Travis CI é uma ferramenta com custos associados, e como tal, a menos interessante para empresa. Outro dos motivos para esta ferramenta não ser a adequada para o caso da Stratio, é o facto de esta só sincronizar com o GitHub, sendo que a empresa utiliza o GitLab.

Dito isto, as ferramentas mais adequadas são o Jenkins ou Jenkins X. O Jenkins X tem uma vantagem que o Jenkins não tem: facilita o processo de instalar e configurar um *cluster* Kubernetes. Por outro lado, o Jenkins X não tem interface gráfica, o que pode fazer com que a curva de aprendizagem seja acentuada para equipas não familiarizadas com a linha de comandos. Segundo Adam Bertram [95]: "Jenkins can perform all the same tasks as Jenkins X, including in Kubernetes, but might require the user to create extensive configurations, as well as spend considerable time to learn (...)."

Depois de uma exaustiva análise entre o Jenkins e o Jenkins X, considera-se que a ferramenta que melhor se adequa às necessidades da empresa é o Jenkins X. Contudo, a ferramenta que será utilizada é o Jenkins, uma vez que esta decisão já tinha sido tomada anteriormente ao começo do estágio.

4.2 Virtualização

Esta secção visa expor a virtualização no contexto do teste de software. Na realização destes testes é necessário ter um ambiente onde seja possível testar o software com diversas configurações. Utilizar um ambiente com hardware real aumenta os custos e o esforço da equipa.

A virtualização é o processo no qual um recurso único do sistema - *Central Processing Unit* (CPU), *Random Access Memory* (RAM) ou disco - é virtualizado e apresentado como vários recursos. A virtualização permite que a capacidade total de uma máquina física seja distribuída entre ambientes, ou seja, fornece um ambiente onde é possível testar o software num único sistema de hardware. Desta forma, o sistema de hardware real fica protegido de possíveis bugs, ou seja, se ocorrer algum problema no sistema virtual o sistema real não será afetado. [96]

De seguida, apresentam-se duas formas de virtualização, máquinas virtuais e *containers*, e é feita uma análise comparativa entre ambas.

4.2.1 Máquinas Virtuais

A máquina virtual, *Virtual Machine* (VM), é um programa de software que fornece emulação de um hardware físico. Uma VM é um ambiente virtual que funciona como um sistema de computador com o seu próprio CPU, memória e interface de rede. [97]

O Hypervisor é o software que replica as funcionalidades dos recursos físicos de hardware subjacentes. Posteriormente os recursos de hardware virtualizados são disponibilizados para as aplicações em execução na VM. [98]

Uma VM funciona, assim, como um computador isolado do restante sistema. É possível ter várias VM's num único computador, ou seja, ter um único hardware físico a operar várias VM's independentes, permitindo assim, por exemplo, ter vários sistemas operativos em simultâneo num único computador. [97]

As operações das VM's consomem muitos recursos, sendo esta uma das desvantagens apontadas. Outra desvantagem prende-se com a migração, por exemplo, caso se pretenda migrar uma aplicação entre máquinas virtuais diferentes, todo o sistema operativo precisa de ser migrado junto com a aplicação. [98]

Embora a virtualização tenha sido criada para otimizar a distribuição de recursos de hardware, um dos grandes inconvenientes das VM's é o desperdício de recursos. Os recursos inicialmente disponibilizados para uma VM dificilmente são consumidos na totalidade. Os recursos não utilizados podem não ser incluídos na distribuição por outras VM's. [98]

4.2.2 *Containers*

A virtualização via *container* ou *containerization* permite que aplicações de software sejam executadas em sistemas operativos virtuais. [99]

Um *container* é um pacote de software que contém todas as dependências necessárias para executar uma aplicação de software.

Um *container* cria uma abstração ao nível do sistema operativo, ou seja, operam dinamicamente e partilham os mesmos recursos físicos - hardware. [100]

Desta forma, os *containers* criam vários ambientes de sistemas operativos isolados no mesmo *kernel* do sistema. Uma das grandes vantagens prende-se com o facto de apenas as bibliotecas e outros componentes serem executados separadamente em cada *container*, isto torna-os mais eficientes em termos de recursos.

Os *containers* podem operar em ambientes de execução isolados na mesma máquina, daí a sua utilidade no desenvolvimento e teste de aplicações. Desta forma, um componente da aplicação pode ser executado na sua totalidade num ambiente, sem pôr em causa outros componentes, reduzindo assim possíveis riscos.

A maior desvantagem apontada aos *containers* é a falta de segurança.

4.2.3 Notas Finais

A principal diferença entre máquina virtuais e *containers* é a nível arquitetónico. Na figura 4.1 observa-se as diferenças na arquitetura.

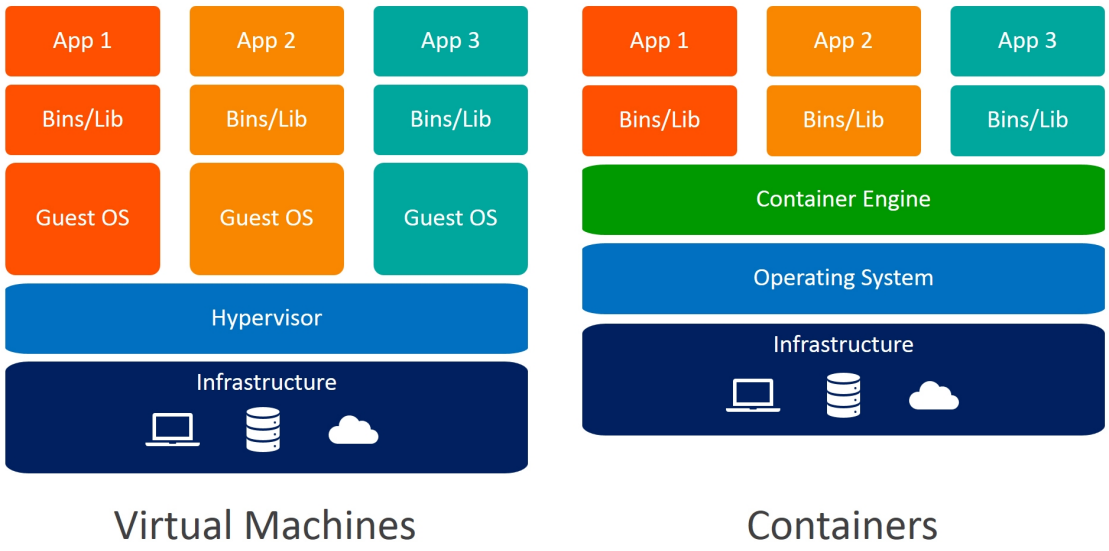

Figura 4.1: Diferenças na arquitetura entre máquinas virtuais e *containers* (retirado de [101])

Como se pode observar, a principal diferença é que as máquinas virtuais virtualizam o hardware e os *containers* virtualizam os sistemas operativos. Ou seja, as máquinas virtuais consomem toda a memória que lhes é alocada, enquanto que os *containers* alocam a memória de forma dinâmica. Os *containers* maximizam a utilização de recursos nos ambientes virtualizados, contrariamente às máquinas virtuais. [102]

As máquinas virtuais são consideradas mais seguras face aos *containers*, uma vez que estas são totalmente isoladas.

Em relação ao tempo de inicialização, um *container* é mais rápido face as máquinas virtuais. [103]

A decisão final recaiu em virtualização via *container*, uma vez que estes facilitam os processos em *pipelines* de CI/CD e são ideais para metodologias e práticas DevOps. Os programadores não necessitam de realizar tarefas de configuração árduas para cada *sprint* do Ciclo de Vida de Desenvolvimento de Software (CVDS). [98]

Outro dos aspetos tido em conta é a velocidade de inicialização de um *container*.

4.3 Ferramenta para injeção de mensagens

O *Message Injector* (MI) irá ser a ferramenta responsável pela injeção de mensagens na Jam *Application Programming Interface* (API). Conforme os requisitos definidos na secção 3.4, foi realizado todo o processo de pesquisa de modo a encontrar qual a ferramenta que melhor se adequa ao requerido.

Durante a pesquisa de ferramentas para injeção de mensagens, foram encontradas diver-

sas que, na teoria, se pareciam adequar. Porém, quando realizada uma experiência com a ferramenta para verificação de tais funcionalidades, observou-se que nem sempre esta correspondia ao que era mencionado.

Verificou-se também que, algumas das ferramentas mais populares, tinham custos associados à medida que a quantidade de mensagens injetadas aumentava, nomeadamente o WebLOAD [104] e o LoadNinja [105].

Após esta pesquisa, foram selecionadas as ferramentas que melhor se adequavam ao caso de uso em questão e com maior adesão da comunidade. A informação relativa à popularidade e a adesão à ferramenta foi retirada de [106] e [107].

Seguidamente, são apresentadas detalhadamente as ferramentas selecionadas, nomeadamente Apache JMeter, Taurus, Gatling e Predator.

4.3.1 Apache JMeter

O Apache JMeter [108] é um software *open-source*, desenvolvido em Java, que permite executar testes de desempenho e funcionais. Este software é capaz de simular carga num servidor ou grupo de servidores, com o objetivo de analisar o desempenho com diferentes níveis de carga. [109]

Inicialmente o JMeter foi planeado apenas para testar aplicações *World Wide Web* (Web). Posteriormente foi estendido para suportar testes a vários tipos diferentes de aplicações e protocolos tais como: serviços Web *Simple Object Access Protocol* (SOAP) e *REpresentational State Transfer* (REST), base de dados, protocolos *Transmission Control Protocol* (TCP), *File Transfer Protocol* (FTP) e *Lightweight Directory Access Protocol* (LDAP) e ainda protocolos de correio electrónico - *Simple Mail Transfer Protocol* (SMTP), *Internet Message Access Protocol* (IMAP) e *Post Office Protocol 3* (POP3).

O JMeter apresenta uma estrutura *multithreading* permitindo, assim, amostras concorrentes e simultâneas, ou seja, o JMeter é capaz de enviar pedidos, para um determinado servidor, de diversos utilizadores em simultâneo. [110] Desta forma é possível simular comportamentos reais e expectáveis de um determinado servidor, ou seja, cargas a que o servidor é e poderá vir a ser submetido.

O JMeter oferece dois modos, o modo *Graphical User Interface* (GUI) e o modo *Command Line Interface* (CLI). O primeiro modo possui uma UI enquanto que o outro modo é em linha de comandos.

A UI do JMeter permite a criação do plano de teste, o que possibilita a adição e edição de vários elementos deste. O JMeter oferece uma grande diversidade de elementos num plano de teste, desde elementos de configuração para definir variáveis que são processadas no início de cada execução do teste, por exemplo, ler um ficheiro *Comma Separated Values* (CSV) para posteriormente atribuir a informação a uma dada variável ou para armazenar e posteriormente enviar *cookies* para um Web *browser*; controladores do tipo *samplers* ou lógicos, para enviar pedidos para um servidor e para controlar a lógica para decidir quando enviar os pedidos, respectivamente; e ainda os *listeners* que fornecem diversas formas para visualizar os resultados do teste. [109]

Já o modo GUI não é aconselhado para execução de testes de desempenho uma vez que, quando executado um teste com muitos utilizadores/*threads*, o JMeter bloqueia ou, quando são utilizados *listeners* no plano de teste estes aumentam o consumo de CPU e de memória, afetando o desempenho da aplicação. Como tal, o modo GUI deve ser utilizado para a

criação do plano de teste e para fins de *debugging*. No modo CLI devem ser executados os ficheiros criados no modo GUI com os elementos *listeners* desativados, melhorando assim o desempenho do JMeter. [111]

O JMeter apresenta uma documentação detalhada e concisa bem como uma grande quantidade de utilizadores, o que facilita o processo de aprendizagem. No entanto, esta ferramenta é caracterizada como tendo uma curva de aprendizagem acentuada. [112]

Uma das limitações do JMeter é a complexidade do ficheiro jmx gerado após a criação do plano de testes, um ficheiro extenso e que requer algum conhecimento dos elementos do JMeter para a sua compreensão. [113]

Os relatórios gerados pela UI do JMeter são de difícil compreensão e a obtenção e personalização deste dados é considerado um trabalho árduo. [114]

4.3.2 Taurus

O Taurus [115] é uma ferramenta de automação de teste, *open-source* criada pela BlazeMeter [116]. Taurus é acrónimo de *Test AUtomation RUnning Smoothly*. [117]

Esta ferramenta possibilita a execução de testes desempenho e de testes funcionais. [118] O Taurus providencia um *Domain-specific language* (DSL) para definir cenários de teste de desempenho, pode ser escrito em YAML ou *JavaScript Object Notation* (JSON). [117] É uma das grandes vantagens desta ferramenta, visto que são linguagens legíveis e o ficheiro de teste fica assim descrito de uma forma perceptível simplificando assim as revisões de código. [107]

O Taurus disponibiliza uma camada de abstração sobre diferentes ferramentas de testes desempenho e/ou funcionais, isto é, estende e abstrai outras ferramentas, como o JMeter, Gatling, Robot, The Grinder, JUnit, Selenium entre outras. Algumas destas são limitadas a um protocolo, outras não oferecem a possibilidade de parametrização, são estas e outras lacunas que o Taurus pretende colmatar. [119] O Taurus acrescenta um DSL ausente em algumas destas ferramentas, fornecendo ficheiros de configuração de fácil leitura e interpretação, mesmo sem experiência com as ferramentas suportadas. O Taurus tem ainda a capacidade de unir vários ficheiros de teste existente num único cenário. [117]

A integração do Taurus com serviços de CI, como o Jenkins, é simples, basta utilizar o modo CLI do Taurus. [118]

Outra funcionalidade do Taurus é a possibilidade de integração com o BlazeMeter [116]. Deste modo, os resultados do teste podem ser enviados para o site do BlazeMeter. Este possui uma versão gratuita e oferece relatórios muito detalhados e em tempo real. [119]

4.3.3 Gatling

O Gatling [120] é uma ferramenta *open-source* focada na automação e análise do desempenho de aplicações Web. [121] Esta ferramenta suporta os protocolos *HyperText Transfer Protocol* (HTTP), WebSockets e eventos enviados pelo servidor e pode ser integrada com ferramentas de integração contínua, como o Jenkins. [120]

O Gatling fornece uma GUI que funciona como um intermediário entre o *browser* e o servidor HTTP. O modo GUI gera uma simples simulação que reproduz a navegação gravada. [120] Posteriormente é gerado um ficheiro Scala e a sua execução só pode ser efetuada no

modo CLI. [122]

A documentação do Gatling é considerada clara e concisa. [123]

Esta ferramenta gera um relatório no final da execução do teste. Durante a execução do teste não é dado qualquer tipo de informação, sendo considerada esta uma das grandes desvantagens. [123] A monitorização durante a execução de um teste de desempenho é considerada um dos aspetos mais relevantes, muitas das vezes não faz sentido esperar pelos resultados finais do teste, pois este pode falhar numa fase inicial. O Gatling FrontLine, a versão empresarial com custos associados, já oferece relatórios mais abrangentes e em tempo real. [120]

4.3.4 Predator

Predator [124] é uma plataforma distribuída *open-source* de testes de desempenho para API's. Esta plataforma não tem qualquer custo associado e é baseada no Artillery [125] como mecanismo para execução de carga.

O Predator permite agendar a execução de cada teste e fornece relatórios bastante completos com diversas métricas de desempenho de cada execução. Esta plataforma permite guardar informação dos vários relatórios gerados, possibilitando assim que sejam feitas comparações com versões anteriores, sendo possível identificar a existência ou não de regressões no desempenho de uma API. [124] Esta plataforma oferece uma série de características relevantes, tais como: relatórios em tempo real, instalação rápida, permite agendar os testes e possui uma UI intuitiva.

O Predator foi desenvolvido para uso próprio da empresa ZOOZ. Inicialmente, esta empresa utilizava o BlazeMeter para executar os ficheiros do JMeter, para o elaboração de testes de desempenho. Porém, a dificuldade em escrever testes que replicassem fluxos dos seus clientes, levou esta empresa a criar uma nova plataforma. O Predator foi desenvolvido de acordo com requisitos muito específicos que suprisse as necessidades da empresa, o que nem sempre se adequa a outros casos de uso. [126]

Aquando da realização desta pesquisa, eram adicionadas e corrigidas funcionalidades com alguma frequência na plataforma, indiciando lançamentos constantes de novas versões. Outra das desvantagens do Predator é a fraca documentação e a escassa comunidade a fazer uso desta plataforma.

4.3.5 Notas Finais

Numa fase inicial do projeto, estava previsto usar o MI desenvolvido pela Stratio. Uma vez que este assegura todas as funcionalidades requeridas, embora, não garanta a ordem *First In First Out* (FIFO) por peça. Apesar disto, não foi a ferramenta utilizada, pois existem outras ferramentas no mercado que fornecem funcionalidades de interesse para o desenvolvimento deste projeto, como a produção automática de relatórios visuais com métricas de desempenho. Como tal, foi realizado um estado da arte neste âmbito, tendo em conta as ferramentas consideradas mais comuns, que possuem uma comunidade abastada de utilizadores e que assegurem alguns dos requisitos anteriormente definidos.

Na tabela 4.2 enuncia-se a relação entre as ferramentas selecionadas e os requisitos definidos na secção 3.4.

	Apache JMeter	Taurus	Gatling	Predator
RMI.1	✓	✓	✓	✓
RMI.2	✓	✓	✓	*
RMI.3	✓	X	✓	*
RMI.4	✓	✓	✓	✓
RMI.5	✓	✓	✓	✓
RMI.6	✓	✓	✓	✓
RMI.7	✓	✓	✓	✓

Tabela 4.2: Relação entre as ferramentas selecionadas e os requisitos requeridos
* - Sem informação disponível

Como se pode observar na tabela 4.2, em relação a ferramenta Predator, não foi possível perceber se detinha ou não as funcionalidades requeridas. Posto isto, foi necessário verificar, se com o recurso ao Predator era possível personalizar o *dataset* e garantir apenas uma mensagem em simultâneo por peça e que estas seguissem a ordem FIFO. Como tal, foi montado um cenário experimental e realizados alguns testes para verificar se a ferramenta era, ou não, adequada a este caso de uso.

O Predator foi instalado localmente num *container* Docker. Depois deste ter sido lançado, foi possível aceder a UI do Predator e configurar um teste básico, apenas com uma mensagem a ser injetada numa API lançada num ambiente de teste no Kubernetes.

Na imagem 4.2 é possível observar os resultados de um teste através da interface do Predator.

Test Name	Start Time	End Time	Duration	Status	Arrival Rate	Ramp To	Success Rate	RPS	Parallelism	Notes
APIIII	Feb 14, 2020 10:25 AM	Feb 14, 2020 10:26 AM	1 minute	Finished	100	N/A	0%	100	1	
APIIII	Feb 14, 2020 10:21 AM	Feb 14, 2020 10:22 AM	1 minute	Finished	100	N/A	0%	100	1	
APIIII	Feb 14, 2020 10:01 AM	Feb 14, 2020 10:02 AM	1 minute	Finished	100	N/A	0%	100	1	
APIIII	Feb 14, 2020 9:36 AM	Feb 14, 2020 9:37 AM	1 minute	Finished	100	N/A	0%	100	1	
APIIII	Feb 14, 2020 9:33 AM	Feb 14, 2020 9:34 AM	1 minute	Finished	100	N/A	0%	99	1	

Figura 4.2: Cenário experimental - Interface do Predator

Após a realização deste cenário experimental foi possível observar que, por vezes, com uma determinada carga e/ou com um *dataset* maior - uma mensagem com mais campos - não era possível realizar o teste, uma vez que a ferramenta deixava de responder. Apontando assim que a versão utilizada ainda não estaria estável.

Concluindo, após a realização do cenário experimental com o Predator foi excluída imediatamente a sua utilização. A escassa documentação e o pouco suporte a esta ferramenta, foram também dois fatores impactantes para a sua exclusão.

Relativamente ao Taurus, este, quando utilizado isoladamente, não detém todos os requisitos do caso de uso, sendo que a sua utilização como camada de abstração de outra ferramenta, por exemplo com o JMeter ou Gatling, já se adequa a este caso de uso.

Tanto o JMeter como o Gatling verificam todas as funcionalidades requeridas. O JMeter foi lançado em 1998 e o Gatling em 2012. A longevidade do JMeter pode ser vista de duas

formas: por um lado, é uma ferramenta que em certo ponto poderá estar obsoleta, mas que, por outro lado, detém muitos anos de suporte e manutenção estando, portanto, mais estável. [127]

Para edição do ficheiro de teste no Gatling é necessário ter conhecimentos da linguagem Scala, enquanto que na ferramenta JMeter basta utilizar a UI deste. Dado que a UI do JMeter é bastante intuitiva e de fácil interação, o JMeter é considerado de mais rápida aprendizagem face ao Gatling [128]

Por todos os motivos mencionados, a escolha final recai sobre o JMeter.

A utilização do Taurus como camada de abstração do JMeter não foi excluída e a decisão sobre a sua utilização só será tomada na altura da implementação.

4.4 Ferramenta para automatização de testes *end-to-end* a uma *User Interface*

Os testes *end-to-end* são testes lentos, complicados e dispendiosos. No entanto, a importância destes testes é significativa. Avaliar se uma determinada aplicação Web funciona como é esperado para o utilizador ou confirmar que uma nova funcionalidade não danificou funcionalidades anteriores são alguns dos benefícios dos testes *end-to-end*. Existem diversas ferramentas que permitem a automatização de testes *end-to-end*, sendo que o Selenium WebDriver e o Cypress foram as ferramentas escolhidas para analisar. O Selenium WebDriver por ser uma ferramenta bastante utilizada na comunidade e pela sua longevidade (2007). O Cypress pela sua rápida adoção pela comunidade, mesmo só tendo sido lançado em 2014, e por ser considerado por muitos o futuro neste tipo de ferramentas.

4.4.1 Selenium WebDriver

O Selenium WebDriver [129] é uma coleção de API's *open-source* utilizada para automação de testes de aplicações Web. [130] Também comummente conhecida como Selenium 2, é uma biblioteca que permite interagir com um *browser*, da mesma forma que um utilizador real faria e verificar se a aplicação funciona como o esperado. [131]

O Selenium WebDriver suporta diferentes *browsers*, nomeadamente, o Chrome, o Firefox, o Safari, o Microsoft Internet Explorer e o Edge, sendo inclusive possível testar uma dada aplicação nos diferentes *browsers* em simultâneo. [132]

Um dos principais benefícios desta biblioteca recai na possibilidade de escolher qual a linguagem de programação para escrever um teste automatizado. O Selenium suporta Ruby, C#, Java, JavaScript, Python, PHP, Perl, entre outras. [131]

O Selenium WebDriver é independente da plataforma, uma vez que o mesmo código pode ser executado por diferentes sistemas operativos. [130]

A comunicação entre a biblioteca e o *browser* é direta e é apontado como um dos benefícios desta ferramenta. Antes do Selenium WebDriver, o Selenium Remote Control agia como uma barreira ente o utilizador e o *browser*, sendo necessário iniciar uma aplicação antes de iniciar o teste. O Selenium WebDriver veio colmatar esta lacuna, tornando o processo de execução de teste mais simples.

Contundo são apontadas várias desvantagens a esta ferramenta. O processo de instalação

é considerado complexo e demorado. [133] Esta apenas suporta aplicações Web, e não é possível executar testes de automação em aplicações Web como SOAP ou REST. [134]

O Selenium tem uma curva de aprendizagem acentuada. É necessário um profundo conhecimento de uma das linguagens de programação suportadas pela ferramenta para automatizar testes com maior eficiência. Apesar da sua enorme comunidade (de acordo com [135] a 20-05-2020 mais de 38 mil empresas), não existe qualquer suporte técnico especializado. O único suporte para resolução de problemas é o suporte em fóruns da comunidade e tutoriais, o que pode levar a uma solução não convencional de um dado problema, uma vez que não são os criadores do produto a responder. O Selenium apenas disponibiliza uma lista de outras empresas de consultoria para obtenção de suporte técnico. [136]

Outra desvantagem remete para a escassez e para a confusa documentação oficial do Selenium WebDriver.

O Selenium WebDriver não oferece qualquer tipo de relatório de teste, o que representa uma das suas grandes desvantagens, uma vez que é essencial obter os resultados dos teste no final da sua execução. Para a obtenção destes relatórios, é possível integrar o Selenium WebDriver com outras ferramentas, como o JUnit, TestNG e Allure. [136]

4.4.2 Cypress

O Cypress [137] é uma ferramenta *open-source* para testar aplicações num *browser*, independentemente da linguagem e estrutura da aplicação. O site do Cypress descreve a ferramenta como "Fast, easy and reliable testing for anything that runs in a browser". [137]

Os testes são escritos em JavaScript e são, por isso, de fácil leitura e compreensão. Esta ferramenta permite executar testes *end-to-end*, de integração e unitários.

Esta ferramenta disponibiliza o Cypress Test Runner e Cypress Dashboard. O primeiro é gratuito e *open-source* e a sua integração com serviços de CI como, o Jenkins, Travis CI e Circle CI, é facilmente conseguida. O segundo é um serviço opcional e pago, a partir de um determinado número de execuções em paralelo, que oferece um *dashboard* que acompanha o Cypress Test Runner. O Cypress Dashboard permite paralelismo automático e o balanceamento de carga, aumentando a velocidade do teste. Este modo é facilmente integrado com serviços de CI, permitindo total visibilidade dos testes em execução através do serviço de CI. Permite ainda receber alertas via Slack ou GitHub quando o teste falha.

O Cypress tem a capacidade de executar testes em diferentes *browsers*, nomeadamente no Chrome, Firefox, Edge, Electron e Brave, sendo, ainda capaz de testar responsividade de uma aplicação, testando diferentes janelas de visualização. [137]

O Cypress proporciona diversas funcionalidades que são de interesse mencionar:

- fornece uma interface onde é possível, em tempo real, ver as interações com a aplicação; [138]
- tira *snapshots*, desta forma basta passar o rato sobre o comando para ver exatamente o que aconteceu naquele momento - *time travel*;
- guarda *screenshots* em caso de falha do teste e, se o utilizador assim o desejar, guarda vídeos completos da execução dos testes;

Funcionalidade\Ferramenta	Selenium WebDriver	Cypress
Linguagens para desenvolvimento	Ruby, C#, Java, JavaScript, Python, PHP, Perl	JavaScript
Instalação	Complexa	Rápida e fácil
Browsers	Chrome, Firefox, Safari, Internet Explorer, Edge	Chrome, Firefox, Edge, Electron, Brave
Snapshots, *Screenshots*, Vídeo	-	++
Velocidade de execução	Reduzida	Elevada
Multi-tab	Sim	Não
Execução em paralelo	Sim	Sim (pago)
Documentação	+	++

Tabela 4.3: Análise comparativa entre o Selenium WebDriver e Cypress

- no final da execução dos testes dá o parecer do sucesso ou insucesso de cada teste e quanto tempo demorou a sua execução. [137]

A documentação oficial do Cypress é muito clara e detalhada. São disponibilizados muitos exemplos onde são explicados detalhadamente todos os passos e são, ainda, sugeridos modos de otimização e alertas para o que não deve ser feito.

Uma das desvantagens apontadas ao Cypress, é o facto de o Cypress Test Runner não permitir paralelizar a execução dos testes, sendo esta lacuna solucionada com o Cypress Dashboard. Outra desvantagem é que o Cypress não suporta *multi-tab*, isto é, várias guias num *browser*.

4.4.3 Notas finais

A tabela 4.3 pretende comparar diversos aspetos chave do Selenium WebDriver e Cypress.

Com o Selenium WebDriver, os testes podem ser escritos em várias linguagens, enquanto que no Cypress a linguagem possível para desenvolvimento é só uma - JavaScript.

Ao nível da instalação, o Cypress é de rápida e fácil instalação. Não é necessária qualquer configuração, basta instalar um executável e todas as dependências são automaticamente instaladas e configuradas. [139] Enquanto que no Selenium WebDriver é necessário configurar os *drivers* e instalar a ligação da linguagem - o mapeamento de uma API ou biblioteca de software para outra linguagem de modo a ser utilizada em dois ambientes de desenvolvimento diferentes. [140]

Em relação aos *browsers* contra aos quais é possível executar os testes, o Selenium WebDriver suporta mais *browsers* do que o Cypress.

Com o Cypress é possível observar o teste a correr em tempo real, guardar *screenshots* de quando o teste falha, ou até mesmo vídeos completos da execução do teste, enquanto que com Selenium WebDriver tal não é possível. Com o Cypress é também disponibilizada informação final acerca do sucesso ou insucesso do teste, enquanto que com o Selenium WebDriver tal só é possível quando este é integrado com outras ferramentas

A execução de testes no Selenium WebDriver são, geralmente, lentos. Pelo contrário, com o Cypress a execução de testes é bastante rápida. Dado que, com o Cypress a execução do teste é no *browser*, o teste está a ser executado ao mesmo tempo que a aplicação. [141]

O Selenium WebDriver é executado fora do *browser*, o que permite a execução de comandos

remotos, mas nunca é possível entender as reações dos eventos que estão a ser executados no *browser*. Por sua vez, o Cypress é exatamente o oposto, ele é executado dentro do *browser*. O que permite o acesso nativo a todos os elementos, mas acarreta uma desvantagem, o de não suportar *multi-tab*. [139]

O Selenium WebDriver permite a execução de testes em paralelo, enquanto que com o Cypress, essa funcionalidade é paga.

Em relação a documentação, esta é muito ampla e profunda no Cypress, e cobre praticamente todos os assuntos, enquanto que a documentação do Selenium WebDriver não é tão completa. [141]

Por todos os motivos referidos anteriormente, e apesar da juvenilidade do Cypress fase ao Selenium WebDriver, a decisão final recaiu sobre a utilização do Cypress. O Cypress adequa-se às necessidades para o desenvolvimento deste projeto. Não existia qualquer tipo de restrição em relação a linguagem, nem ao *browser*. O Cypress tem emergido na comunidade muito rapidamente, e a velocidade de execução de teste e a sua documentação são duas das grandes vantagens face ao Selenium WebDriver.

Esta página foi propositadamente deixada em branco.

Capítulo 5

Arquitetura e Desenvolvimento

Este capítulo visa expor a arquitetura e implementação realizados ao longo do estágio. Vai ser divido em duas partes, uma referente à automatização dos testes de desempenho (épico: API Perf Tests) e outra referente à automatização dos testes *end-to-end* (épico: E2E Tests). É, assim, exposto em detalhe a arquitetura e todo o desenvolvimento efetuado para ambos os épicos.

5.1 Automatização dos testes de desempenho

De seguida é apresentado o plano de teste definido para a execução de testes de desempenho à componente Jam *Application Programming Interface* (API) e é explicado em detalhe toda a implementação efetuada para a execução do mesmo.

5.1.1 Plano de teste à Jam *Application Programming Interface*

Esta secção visa pormenorizar o plano de testes à componente Jam API. O plano de testes é um documento que descreve em detalhe as atividades de teste. [12] O conteúdo deste documento varia, mas o objetivo de qualquer um é servir de guia para a execução das atividades de teste. [142]

Jam *Application Programming Interface*

Como já foi referido, a Jam API foi desenvolvida pela Stratio e é responsável por receber e enviar mensagens de/e para as peças. O formato das mensagens pode ser *JavaScript Object Notation* (JSON), JSON comprimido com *heatshrink*, binário ou binário comprimido com *heatshrink*.

A Jam API tem seis *endpoints*:

- GET/actions - para as peças transferirem ações que tenham pendentes (*updates* de *firmware*, *reboots*, etc);
- GET/configurations - para as peças obterem a sua própria configuração;
- GET/systempids - para as peças saberem que *Parameter IDs* (PIDs) (leituras feitas aos sensores dos veículos) têm de ler;

- POST/measurements - receção de leituras de sensores dos veículos;
- POST/realtime - receção de leituras de sensores e *Global Positioning System* (GPS) sem *batching*;
- GET/status - para as peças fazerem *update* do seu estado atual.

O *endpoint* a ser testado na execução deste plano de teste é o *measurements*, a peça envia uma mensagem com diversos parâmetros e a Jam API responde com uma mensagem de sucesso ou um código de erro. O *endpoint* escolhido para alvo de teste é o principal responsável pela carga na Jam API, recebe cerca de dois pedidos por minuto por peça, e é considerado bastante crítico no caso de ocorrer alguma falha.

Dataset

O *dataset* representa a mensagem a ser injetada no *endpoint* measurements da Jam API. Como já foi mencionado, a Jam API pode receber mensagens em diversos formatos, mas na execução deste plano de teste, o formato da mensagem a ser injetada será em JSON.

No *listing* 5.1 é possível observar um exemplo de uma mensagem que a peça envia para o *endpoint* measurements da Jam API.

```
{"clk": "ASASsadw34wsdAS",
  "dvc": "SQA7ABRQM0hWMzIg",
  "dt": 1826739822,
  "seq": 255,
  "ltc": 1232341244,
  "net": 1,
  "vrs": "020103/020402",
  "gps": [
    { "lt": 42.000231, "ln": -8.231879, "dt": 1826739823}
  ],
  "dtc": [
    { "sid": 23, "cd": "UDA0MkE=", "dt": 1826739824},
    { "sid": 4,  "cd": "QzA0MkE=", "dt": 1826739825}
  ],
  "pid": [
    { "id": 313, "vl": 12.3, "dt": 1826739825, "st": 2},
    { "id": 1453, "vl": 60,  "dt": 1826739828}
  ],
  "acc": [
    { "x": 0.25, "y": 1.07, "z": 0.02, "m": 1.23, "dt": 182673
       9825}
  ],
  "ign": [
    { "is": 1, "dt": 1826739825}
  ]
}
```

Listing 5.1: Exemplo de uma mensagem recebida pela Jam API

Como é possível observar a mensagem contém vários campos, de notar o significado de cada um:

- clk - chave do cliente;
- dvc - *IDentifier* (ID) da peça;
- dt - *timestamp* da mensagem;
- seq - número de sequência da mensagem;
- ltc - *timestamp* do último PIDs recebido;
- net - tipo de rede;
- vrs - versões de *firmware*;
- gps - lista de locais *Global Navigation Satellite System* (GNSS) medidos;
- dtc - lista de *Data Trouble Codes* (DTC) medidos;
- pid - lista de PIDs - leituras feitas aos sensores do veículo;
- acc - lista de componentes de aceleração medidos;
- ign - lista de estados de ignição medidos.

Para a execução desta bateria de teste foi definida a mensagem no formato JSON com 70 PIDs que é o equivalente a uma mensagem comprimida. A mensagem a injetar será sempre a mesma, sendo alterado o cliente e o veículo de onde é proveniente a mensagem e o dt e seq é incrementado em cada mensagem injetada.

Setup

O esquema da figura 5.1 representa o *setup* para execução da bateria de teste à Jam API. Os ficheiros de configuração do JMeter são executados através do Taurus e é introduzida carga, isto é, uma determinada quantidade de mensagens por segundo no *endpoint* measurements da Jam API via *HyperText Transfer Protocol* (HTTP). Posteriormente estas são escritas para o tópico Kafka api-all-requests.

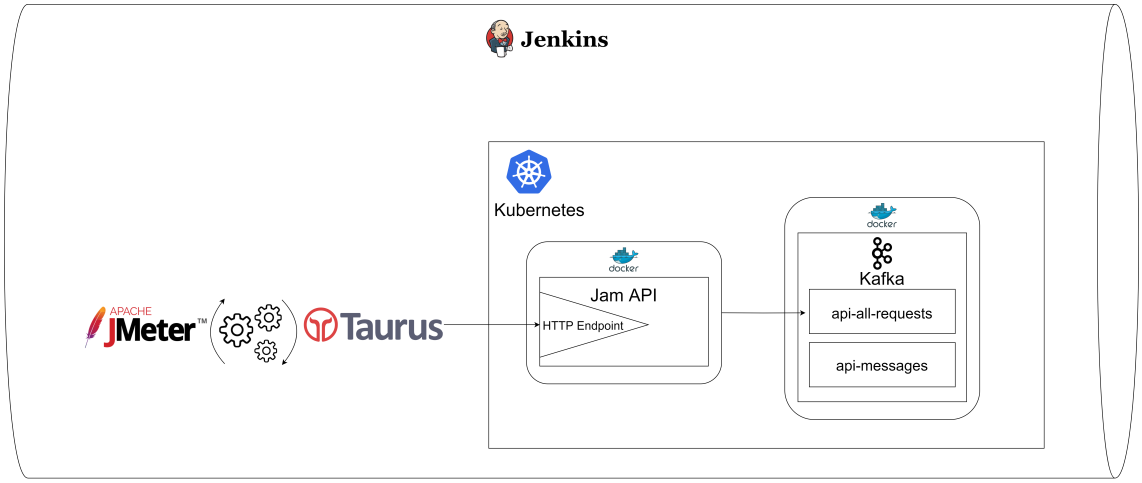

Figura 5.1: *Setup* para a execução da bateria de testes a Jam API

Objetivo

O principal objetivo da execução deste plano de testes é validar os requisitos funcionais e não funcionais (ver tabela 3.5 na secção 3.6) da funcionalidade relativa ao processamento de mensagens de leituras de sensores que são recebidas pela Jam API, isto é, exercer o *endpoint* measuments. Posteriormente incluir tais resultados no Ciclo de Vida de Desenvolvimento de Software (CVDS), dando assim a garantia que para futuras versões da Jam API os requisitos relativos ao processamento de mensagens de leituras de sensores foram devidamente validados.

Outro dos objetivos é encontrar a quantidade mínima de recursos de *Central Processing Unit* (CPU) e *Random Access Memory* (RAM) para os quais a Jam API cumpra os requisitos não funcionais de desempenho (ver tabela 3.5 na secção 3.6). Para tal serão analisados o mínimo, a média e o máximo de utilização de CPU e RAM do *container* que executa a Jam API durante a execução da bateria de testes. Subsequentemente, será definido um valor aceitável de utilização de CPU e RAM expectáveis quando uma dada carga é injetada.

Especificações da máquina do Kubernetes

O ambiente alvo de testes será lançado no Kubernetes. Como tal, é importante perceber que diferenças existem relativas ao ambiente real. O *cluster* Kubernetes está configurado fisicamente numa máquina na Stratio com quatro CPU's físicos do modelo Intel(R) Xeon(R) Gold 6138.

A nível da rede, o facto de todos os *containers* estarem na mesma mesma máquina física, a latência entre serviços será mínima ou nenhuma. Tal não acontece no ambiente real, uma vez que, em alguns casos, os serviços estão em edifícios diferentes. Para mitigar este problema foi lançado um *container* Pumba [143] que permite a introdução de latência entre os *containers* de modo a ser possível simular os atrasos na rede que acontecem no ambiente de produção.

Outro dos defeitos, impossível de controlar, é o facto de que no nó de Kubernetes para além dos *containers* do ambiente alvo de teste são também alocados outros não relacionados com o ambiente a testar. Tal facto vai gerar carga de CPU, RAM, disco e rede no nó de Kubernetes, podendo alterar os resultados da bateria de testes.

Ambiente alvo de teste

O ambiente alvo de teste é o ambiente contra o qual a carga será injetada. Este ambiente deve ser o mais semelhante possível ao ambiente de produção, de tal modo, a que as métricas de desempenho recolhidas sejam realísticas e tenham impacto em futuras decisões quanto aos recursos necessários sob determinadas cargas.

O ambiente é constituído pelos seguintes *containers*, como se pode observar no esquema da figura 5.2.

Como foi referido, o ambiente alvo de testes deve ser o mais semelhante possível ao ambiente de produção. Contudo não foi possível criar um ambiente de testes com todos os clientes, por diversas razões de seguida enumeradas:

- Tempo de execução do teste - no início de cada teste o ambiente vai ser lançado no Kubernetes, um ambiente com todos os clientes, ou seja, com as bases de dados de

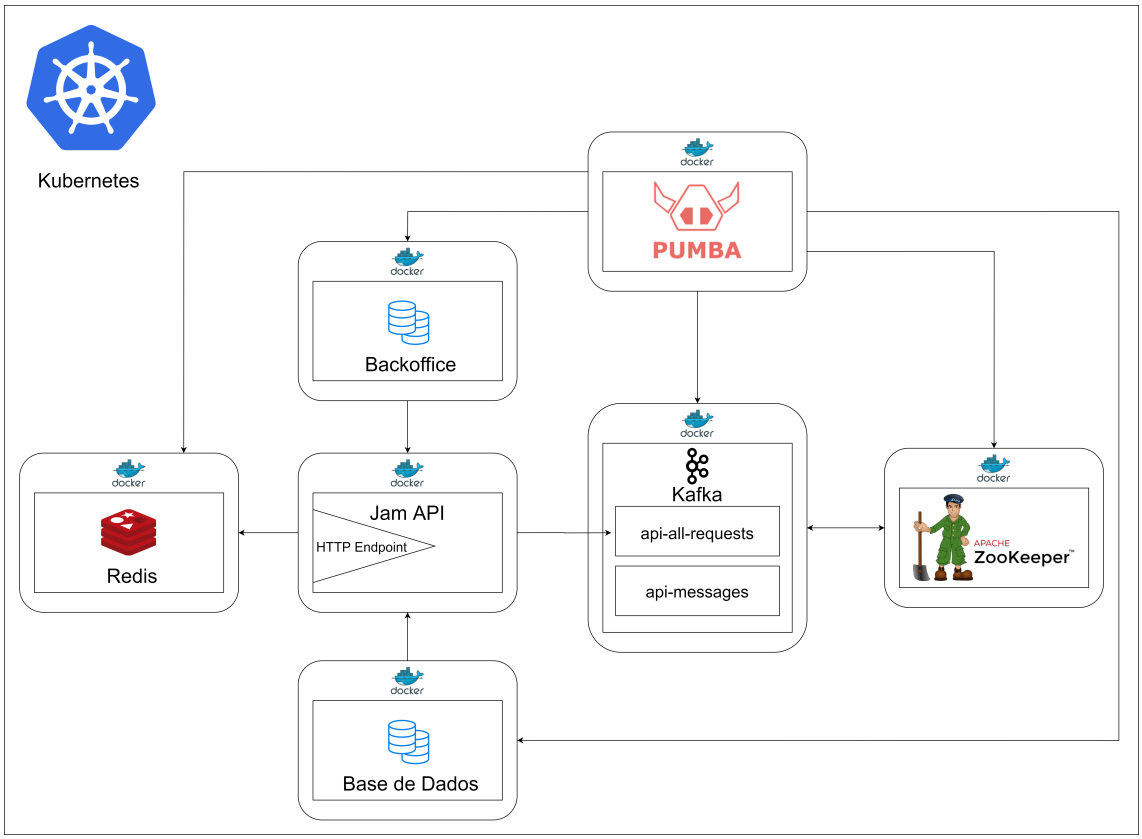

Figura 5.2: Constituição do ambiente alvo de teste

todos os clientes, iria demorar muito mais tempo a ser criado, e como tal, o tempo de execução do teste iria consequentemente aumentar;

- Quantidade de recursos - quanto mais bases de dados de clientes forem lançadas mais recursos são necessários;

- *Overhead* de sincronização - este problema foi encontrado após os testes de validação à Jam API na *Amazon Web Services* (AWS) (ver secção 6.2). Foi possível observar problemas de *overhead* de sincronização entre *threads* - peças - por parte do JMeter.

Posto isto, o ambiente criado contém dois clientes. São dois clientes representativos, o primeiro cliente tem cerca de 100 veículos e o segundo 120 veículos, um número de veículos suficientemente grande para injeção de uma carga considerável. São clientes com uma ampla diversidade do tipo de veículos: camiões, autocarros, veículos eléctricos e de combustão.

Os *containers* do Redis, Backoffice e Kafka são dependências da Jam API, isto é, precisam de ser lançados para que o ambiente funcione corretamente. O *container* Zookeper é dependência do Kafka.

Métricas

As métricas de desempenho ou *Key Performance Indicators* (KPIs) a retirar na execução desta bateria de testes são enumeradas de seguida, sendo possível observar outras métricas.

- Taxa de transferência - número de mensagens que a Jam API foi capaz de processar por segundo;
- Tempo de resposta no percentil 99.9 e 99.99 - é o tempo que uma peça demora a obter a resposta da Jam API nos percentis 99.9 e 99.99;
- A percentagem mínima, média e máxima de utilização de CPU do *container* que executa a Jam API no Kubernetes durante a execução do teste;
- A percentagem mínima, média e máxima de utilização de RAM do *container* que executa a Jam API no Kubernetes durante a execução do teste.

(ver subsecção 2.4.1 para mais informações acerca KPIs de desempenho)

Critérios de aceitação

Os critérios de aceitação são as condições necessárias para a conclusão bem sucedida de uma bateria de testes. Os critérios de aceitação definidos são:

- Pedido para criação do ambiente através da *Automated Tests Framework* (ATF) retorna o código 200 - resposta positiva de criação de ambiente;
- Execução com sucesso do JMeter e Taurus;
- As mensagens injetadas seguem a ordem *First In First Out* (FIFO) por peça;
- O número de mensagens injetadas na Jam API é igual ao número de mensagens escritas para o tópico api-messages do Kafka;
- Os requisitos não funcionais de desempenho da componente têm de ser cumpridos (ver tabela 3.5 na secção 3.6);
- Todos os recursos criados (ambiente no Kubernetes e ficheiros gerados) devem ser eliminados no final da execução do teste ou se alguma *stage* do *pipeline* falhar.

Critério de suspensão

O critério de suspensão diz respeito à condição ou condições para suspensão da execução da bateria de testes, isto é, respostas que não são aceitáveis para a continuação da execução da bateria.

Caso algum dos critérios de aceitação, definidos anteriormente, seja violado, o teste é abortado com erro imediatamente, sendo este o critério de suspensão definido para este plano de testes.

5.1.2 Integração com o Ciclo de Vida de Desenvolvimento de Software

Para entender a integração desta bateria de testes com o CVDS é necessário entender o processo de fluxo de trabalho do Git aplicado na Stratio, exposto na subsecção 2.10.3.

O esquema da imagem 5.3 apresenta a integração da bateria de testes com o CVDS. Este fluxo é utilizado para outra baterias de testes, não só para os de desempenho, e representa a forma como a empresa implementou o processo de integração contínua.

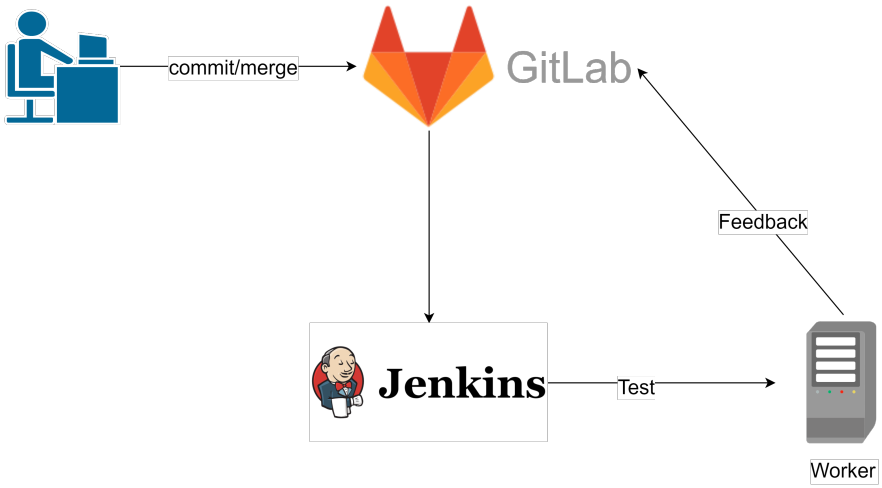

Figura 5.3: Processo de integração contínua

Quando um programador faz um *commit*/*merge* e submete no GitLab, este envia uma notificação para o Jenkins. De seguida, o Jenkins clona o repositório com as alterações efetuadas para um dos seus *workers* disponíveis, e este realiza os testes definidos na respetiva *pipeline*. Posteriormente esta *pipeline* gera um relatório com o resultado e notifica o GitLab com a indicação se o código submetido passou ou não no teste. Desta forma, o código é continuamente alvo de teste.

A bateria de testes de desempenho vai correr automaticamente sempre que for feito um *merge* para o *branch* de desenvolvimento no repositório relativo a Jam API. A bateria de testes de desempenho pode ser executada manualmente contra uma *feature*, sempre que o programador assim o entender. Para tal, basta indicar o ID do *commit* e executar a bateria de testes.

Idealmente a bateria de testes de desempenho podia ser executada automaticamente contra todas as *features*. Desta forma seria mais fácil e rápido de descobrir onde estaria o problema caso o teste falhe. Esta não foi a abordagem escolhida, dado que a bateria de testes consome recursos (criação do ambiente, utilização de um *worker* Jenkins para a execução da bateria, etc) e tempo na sua execução. Posto isto, a abordagem escolhida - executar a bateria de testes de desempenho automaticamente quando é feito um *merge* para desenvolvimento - é uma abordagem mais equilibrada, não impactando o trabalho dos programadores.

5.1.3 Desenvolvimento - Fase Inicial

Como já mencionado, foi utilizado o Apache JMeter como ferramenta para injeção de carga. O ficheiro para injeção de carga foi configurado através da interface do JMeter, que posteriormente gera um ficheiro do tipo jmx. Na imagem 5.4 é possível observar a interface do JMeter configurado para este caso.

Uma *Thread Group* é o ponto inicial do plano de teste e é responsável por controlar o número de *threads* que o JMeter vai utilizar para a execução do teste. [109]

Posteriormente foi adicionada um *Controller* do tipo *Samplers*. Os *Samplers* têm a responsabilidade de enviar pedidos para um servidor. O *Sampler* adicionado foi o HTTP *Request*, que permite enviar pedidos HTTP ou *HyperText Transfer Protocol Secure* (HTTPS) para um servidor *World Wide Web* (Web). [109]

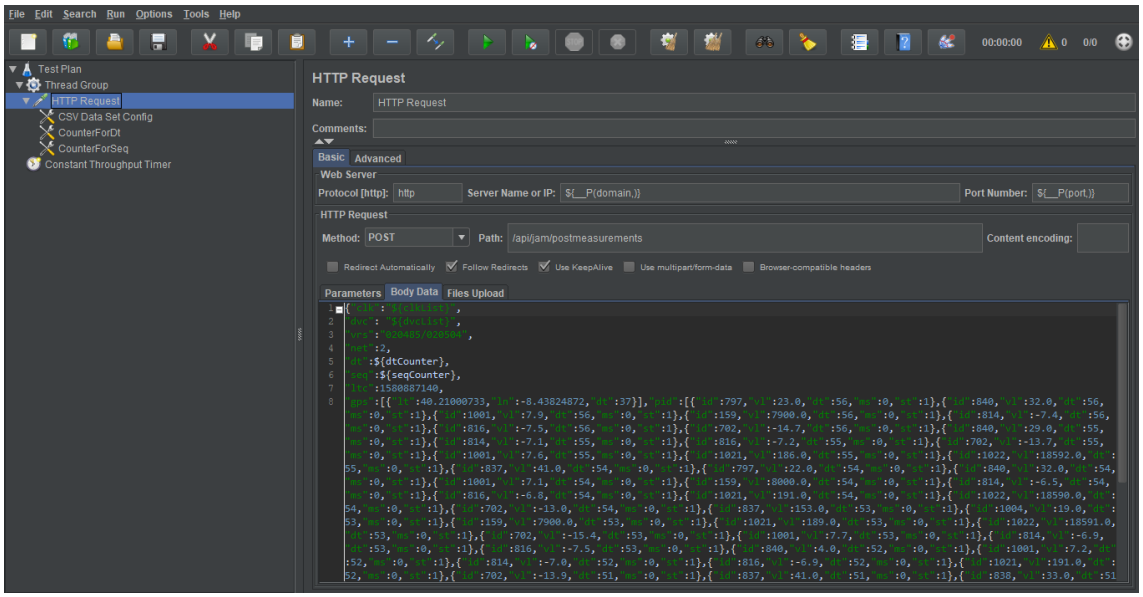

Figura 5.4: Interface do JMeter

O número de *threads* (*concurrency*), o número de execuções do teste (*iterations*) , o domínio (*domain*) e o porto (*port*) onde se pretende injetar carga, são passados por parâmetro quando o ficheiro é executado, uma vez que, são valores de interesse alterar de maneira rápida e fácil. Todos estes parâmetros podem ser definidos na *User Interface* (UI) do JMeter caso se pretenda efetuar o teste através desta - unicamente para fins de *debug* (ver subsecção 4.3.1).

O método é POST e a injeção de mensagens é feita no *endpoint* da Jam API measurements.

O *Body Data* representa a mensagem a ser injetada. O parâmetro clk e dvc da mensagem, chave do cliente e o ID da peça, respectivamente, são lidos de um ficheiro. Este ficheiro que contém o clk e dvc, será gerado depois do ambiente ser lançado no Kuberntes, após uma *query* as base de dados lançadas.

Os parâmetros dt e seq, *timestamp* da mensagem e o número de sequência da mensagem, respectivamente, são incrementados no CounterForDt e CounterForSeq.

O *counter* é um elemento de configuração do JMeter, que permite configurar um ponto de partida, um máximo e um incremento. Este contador é iniciado aquando o teste, e é incrementado na execução de cada *thread*, isto é, em cada pedido HTTP efetuado. Desta forma, os campos dt e seq são incrementados a cada pedido realizado a Jam API.

A *Thread Group* foi adicionado um elemento - *Constant Throughput Timer* - este permite manter a taxa de transferência constante. Deste modo é possível controlar a frequência de pedidos ao servidor por minuto, sendo este valor passado por parâmetro (*throughputSamplesMinute*). Cada *thread* irá tentar manter a taxa de transferência definida, sendo que nem sempre é possível. O valor definido da taxa de transferência pode não ser atingido porque o servidor pode não ser capaz de lidar com tantas mensagens simultaneamente ou pode ser pela própria definição do plano de teste. Isto é, podem existir outros elementos do teste, por exemplo, os contadores que podem fazer com que a taxa de transferência seja menor do que a definida.

5.1.4 Decisão de Implementação - Apache JMeter & Taurus & Blazemeter

Inicialmente foi criada uma *pipeline* no Jenkins que executava o ficheiro jmx mencionado anteriormente. A complexidade deste, os fracos relatórios finais gerados e a dificuldade em analisar os dados, levaram à pesquisa de diversas especificações para melhorar tal problema.

A integração do Apache JMeter com o Taurus e, por vezes, com o BlazeMeter, era referida muitas vezes para a resolução deste problema. [144] O Taurus funciona como uma camada de abstração sobre o JMeter, com o objetivo de simplificar os processos de integração do JMeter com o Jenkins e a extração dos resultados do teste. [117]

Posto isto foi realizado um cenário com o JMeter, Taurus e BlazeMeter para verificação da possível usabilidade para este projeto.

Para tal, o Taurus foi instalado localmente e o ficheiro jmx, anteriormente gerado pelo JMeter, foi executado pelo Taurus, recorrendo ao comando bzt e com os respetivos parâmetros anteriormente definidos.

O Taurus gera diversos ficheiros e resultados visuais na própria consola. Um dos ficheiros é um *YAML Ain't Markup Language* (YAML). Este ficheiro gerado pelo Taurus é simples e curto, ao contrário daquele gerado pelo JMeter, que é extenso e confuso.

A este ficheiro YAML gerado pelo Taurus foram adicionadas duas funcionalidades. A primeira funcionalidade prende-se com o percentil, foi adicionado um módulo para calcular o tempo de resposta no percentil 99.99, uma vez que tal não foi possível com o JMeter.

A outra funcionalidade prende-se com a geração de relatórios. Esta funcionalidade permite agregar os resultados do teste num único ficheiro. Desta forma, os resultados ficam agregados num único local facilitando a sua interpretação.

Quando é adicionada a opção -report ao comando bzt, os resultados do teste são enviados ao serviço de relatórios do BlazeMeter. Quando o comando é executado com esta opção é gerado um *link* que abre automaticamente no *browser* padrão. O relatório gerado por este serviço, contém informação muito abrangente de métricas de desempenho, códigos de resposta do pedido HTTP efetuado, média de largura de banda entre outras. Na imagem 5.5 observa-se a primeira página deste relatório.

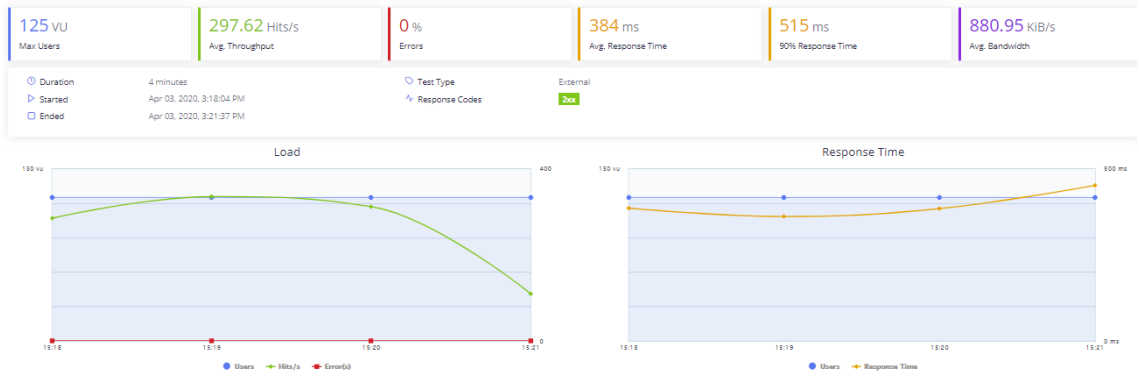

Figura 5.5: Primeira página do relatório gerado pelo BlazeMeter

Como é possível observar pela imagem, a primeira página corresponde ao sumário do relatório e inclui:

- Máximo de utilizadores - número máximo de *threads*, neste caso as peças colocadas nos veiculo, em simultâneo num determinado momento. De importante notar que não corresponde ao total de peças este valor só é obtido multiplicando o número de peças pelo número de iterações do teste;
- Taxa de transferência média - o número médio de pedidos por segundo por peça durante o teste;
- Erros - a percentagem de erros no total dos pedidos;
- Tempo médio de resposta - quanto tempo, em milissegundos, uma peça em média levou para receber uma resposta para o seu pedido;
- Tempo de resposta de 90% - tempo de resposta mais lento, em milissegundos, recebido pelo percentil 90;
- Largura de banda média - consumo médio de largura de banda em Kibibyte (KiB) por segundo. Esta métrica pode ser util para entender a carga típica da rede nos servidores durante casos de alto tráfego.

Ainda na mesma página, é mostrada a duração do teste, os horários de início e fim do teste e o códigos de resposta. Os gráficos, são relativos a carga e tempo de resposta. No primeiro é possível observar o número máximo de usuários, taxa de transferência por segundo e os erros. No segundo relaciona-se o número máximo de peças em função do tempo de resposta, evidenciando como o tamanho da carga afeta os tempos de resposta. [145]

5.1.5 Criação do projeto no Jenkins

O Jenkins, como já mencionado anteriormente, é a ferramenta de *Continuous Integration* (CI)/*Continuous Delivery* (CD) que a empresa utiliza. E foi a ferramenta utilizada para automatizar os testes de desempenho. Os conceitos de seguida mencionados foram definidos na subsecção 2.9.1.

Para iniciar é necessário adicionar um novo item ou trabalho. Um item pode ser de vários tipos, como é possível observar na figura 5.6.

Para o desenvolvimento em questão, foi escolhida a opção *Freestyle project*. Esta opção permite a criação de um item de uso geral e que fornece o máximo de flexibilidade. [47] Após esta escolha é necessário configurar o projeto. Informações como: o nome do projeto, descrição do projeto e parâmetros do projeto devem ser preenchidas. Neste caso foi definido um parâmetro, o *branch* contra qual é pretendido executar a *pipeline*.

Posteriormente é necessário configurar a parte da *pipeline*. Como é possível observar na imagem 5.7, ao definir a *Pipeline* foi escolhida a opção "*Pipeline Script from SCM*"(*Source Code Management* (SCM)). É possível escrever o código da *pipeline* na área de texto da página de configuração desta, mas torna-se difícil de manter e escrever. A opção escolhida vem assim facilitar isso, uma vez que o código da *pipeline* fica num ficheiro Groovy, ficheiro esse que é versionado no repositório Git.

De seguida, é preenchido o *Uniform Resource Locator* (URL) do repositório onde se encontra o ficheiro e as respetivas credencias para o Jenkins aceder ao repositório, uma vez que é um projeto privado (chave *Secure Shell* (SSH)). Ao executar o projeto, o Jenkins irá clonar o código e executar o ficheiro indicado relativo à *pipeline*.

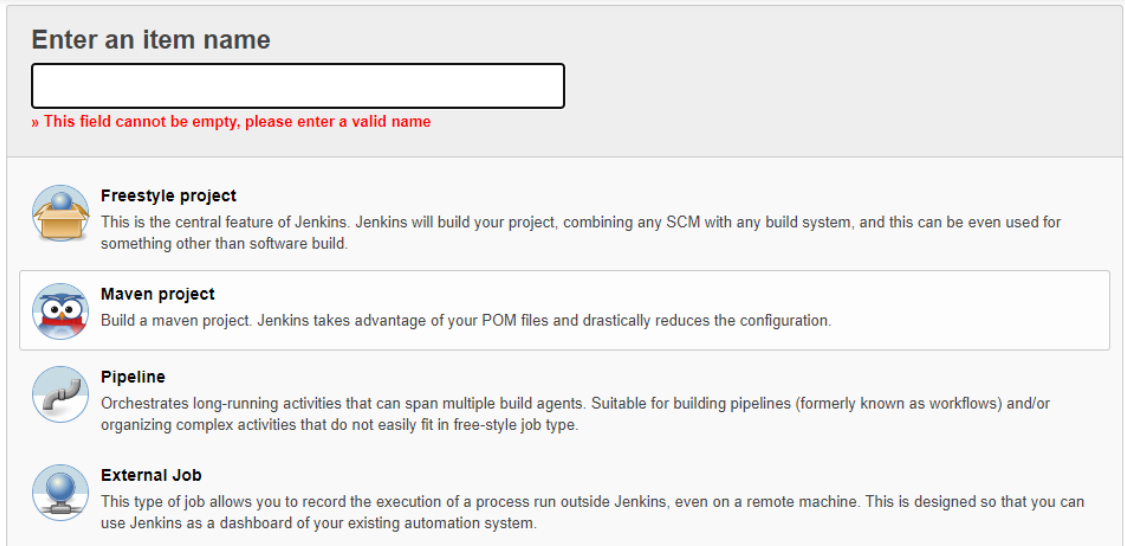

Figura 5.6: Criação de um novo item

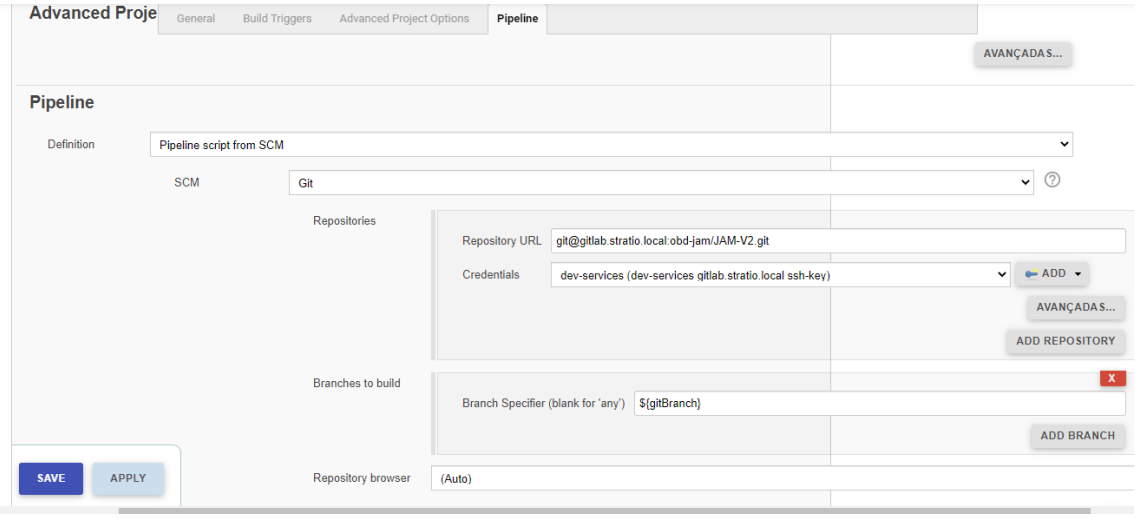

Figura 5.7: Configuração do projeto no Jenkins

5.1.6 *Pipeline* desenvolvida

Antes de explicar a *pipeline* desenvolvida no Jenkins, é de notar alguns termos do Jenkins referidos na subsecção 2.9.1.

O ficheiro com o código em Groovy relativo à *pipeline* desenvolvida, começa com alguns *imports*, declarações de variáveis, incluído o número da *build* atual.

Recorrendo ao *plugin Timestamper* [146] - para adicionar data e hora aos *logs* da consola - o restante código é dentro da chamada da função *timestamps*.

De seguida, a função *node*, permite alocar uma máquina e um *workspace* no ambiente do Jenkins, e é aqui que o trabalho será executado.

Dentro deste nó, são criadas três *stages* - *Env*, *Load* e *Clean* - com determinadas tarefas,

tal como é possível observar na imagem 5.8. Se ocorrer algum erro em alguma tarefa na primeira ou segunda *stage*, imediatamente a terceira *stage* é executada.

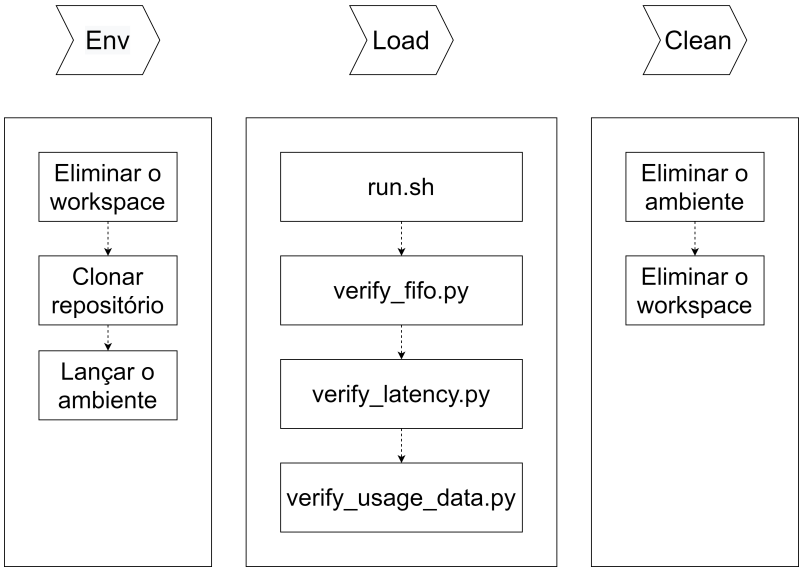

Figura 5.8: *Stages* da *pipeline*

Primeira *Stage* - Env

Na primeira *stage*, inicialmente o *workspace* é eliminado antes da *build* começar a execução, para que não sejam mantidos ficheiros de *builds* anteriores e poder causar conflito.

De seguida, o repositório é clonado, este contém ficheiros que serão executados na *stage* seguinte e também o código relativo a Jam API. É gerada a imagem Docker da Jam API para, posteriormente, ser esta a ser lançada no ambiente.

Posteriormente é feita uma chamada à função *atf* para lançar o ambiente no Kubernetes. Esta função tem como parâmetros: o *endpoint* da API da ATF, neste caso env/create, uma vez que se pretende criar um novo ambiente; a configuração do ambiente; e o método, que por *default* é *POST*, que é o caso.

A configuração do ambiente é lida de um ficheiro onde estão indicados todos os serviços que se pretendem lançar e a identificação do *namespace* - nome do ambiente. O *namespace* é acompanhado do número da *build* atualmente em execução.

São guardadas em variáveis informações do endereço *Internet Protocol* (IP) e porto do *container* lançado da Jam API e do Kafka para posteriormente serem utilizadas.

Por último, nesta *stage* é feita uma *query* as bases de dados no ambiente lançado no Kubernetes para obter o clk e dvc de peças ativas.

Segunda Stage - Load

Primeiramente é executado o ficheiro run.sh. Para tal recorre-se ao *step* sh que é utilizado para executar um comando shell. [147]

Para executar tal ficheiro é necessário recorrer ao *plugin* do Jenkins *Credentials Binding* que permite vincular credenciais a variáveis. É feita uma chamada a função *withCredentials* e é vinculado o ficheiro secreto que também contém as credencias à variável kubeconfig.

O ficheiro run.sh é um ficheiro *bash* com oito parâmetros:

- kubeconfig - parâmetro associado às credenciais para o Kubernetes;
- file - ficheiro YAML extraído após a execução do ficheiro JMeter através do Taurus;
- concurrency - número de *threads*, isto é, número de peças contidas no ficheiro devices.txt;
- iterations - número de execuções de teste;
- port - porto do *container* da Jam API lançada no Kubernetes;
- domain - nome do servidor ou endereço IP;
- throughputSamplesMinute - número de amostras a injetar por minuto;
- report - se estiver definido gera um relatório no BlazeMeter.

Inicialmente este ficheiro começa por recolher métricas de CPU e RAM relativas ao *container* da Jam API lançado no Kubernetes na *stage* anterior - daí a necessidade das credenciais do Kubernetes.

De seguida é executado o comando bzt com os devidos parâmetros para proceder à injeção de mensagens na Jam API.

Depois da execução do ficheiro run.sh são executadas as diversas verificações.

Primeiramente, é executado o ficheiro verify_fifo.py, mais uma vez recorrendo ao *step* sh. Para tal, é preciso instalar o pipenv - ferramenta para gerir dependências nos projetos Python. Este ficheiro recebe como argumentos o endereço IP e o porto do *container* que executa o Kafka no ambiente lançado no Kubernetes e o número total de mensagens injetadas na Jam API.

Com recurso à biblioteca KafkaConsumer do Python, é estabelecida a conecção com o *broker* Kafka no tópico 'api-messages' e o conteúdo de cada mensagem é guardado num dicionário. O conteúdo desse dicionário vai ser percorrido, e para o mesmo dvc, caso exista um dt menor que na mensagem anterior, isto é, para a mesma peça, se a mensagem seguinte tiver um dt inferior à mensagem anterior, é gerado um erro. Se o número de mensagens lidas do tópico for superior ao número de mensagens injetadas na Jam API é gerado um erro. Se a última mensagem recebida foi há mais de 15 segundos e o número de mensagens recebidas é inferior ao número de mensagens injetadas na Jam API é gerado um erro.

De seguida é executado o verify_latency.py, recorrendo mais uma vez ao *step* sh para tal. Começa por ler o ficheiro gerado após a execução do comando bzt que contém informação do valor do tempo de resposta por percentil. De seguida tais valores serão comparados com os valores aceitáveis de tempo de resposta para um dado percentil já definidos.

Por último, é executado o ficheiro verify_usage_data.py. Este ficheiro vai ler as informações de utilização de CPU e RAM do *conatiner* que executa a Jam API. Posteriormente é encontrado o mínimo, a média e o máximo de utilização CPU e RAM, para ser feita a comparação com os valores aceitáveis definidos.

Ao longo desta *stage* a verificação da ordem das mensagens, do tempo de resposta e de CPU e RAM são imprimidas na consola do Jenkins, caso a mensagem seja de insucesso automaticamente a terceira *stage* é realizada. São ainda mostrados nos *logs* do Jenkins, os valores do tempo de resposta nos percentis 99.9 e 99.99, bem como o URL para aceder ao relatório no BlazeMeter caso essa opção tenha sido escolhida.

Terceira Stage - Clean

Na última *stage*, o ambiente inicialmente criado no Kubernetes é eliminado. Para tal, é feita uma chamada da função *atf*, com o *endpoint* da ATF env/teardown; o *namespace* do ambiente; e o método DELETE. Por ultimo o *workspace* é eliminado.

5.1.7 Fluxo de comunicação

O seguinte fluxo de comunicação pretende resumir a comunicação entre os diversos componentes que compõem a arquitectura definida para a automatização de testes de desempenho. O fluxo de comunicação entre as diversas componentes na arquitetura de teste pode observar-se na imagem 5.9.

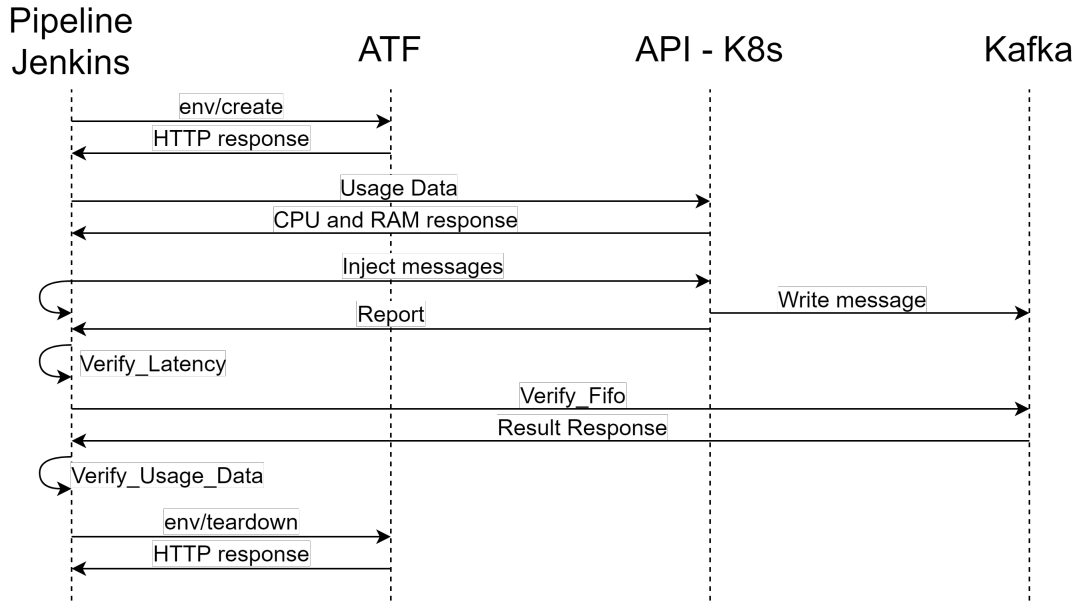

Figura 5.9: Fluxo de comunicação

5.2 Implementação - Automatização dos testes *End-to-end*

5.2.1 Plano de Teste à *User Interface* do Stratio-Play

Esta secção visa pormenorizar o plano de testes definido para a automatização de testes *end-to-end* à uma UI.

Objetivo

Atualmente na Stratio, antes de um *deployment* para produção, é lançado um ambiente no Kubernetes, o *User Acceptance Testing* (UAT). Neste ambiente são realizados testes manuais a UI somente as novas funcionalidades implementadas.

O objetivo deste plano de testes é colmatar tal falha. Antes de uma nova versão do produto ser *deployment* para produção, executar esta bateria de testes irá permitir confirmar de

forma automática que novas e antigas funcionalidades funcionem conforme o esperado (teste de regressão, ver subsecção 2.4.3).

Ambiente alvo de testes

O ambiente alvo de testes será o UAT que está no *cluster* Kubernetes. O UAT é um ambiente muito semelhante ao de produção e contém diversos serviços, Jam API, Redis, Backoffice, Kafka, Zookeper, bases de dados de clientes, Stratio-play, entre outros.

O Stratio-play é a plataforma Web direcionada para o cliente. Esta aplicação Web é onde os clientes monitorizam e gerem as suas frotas. A UI desta aplicação é o alvo de teste.

O lançamento do UAT no Kubernetes é da responsabilidade de um elemento da equipa e acontece sempre que novas funcionalidades necessitam de ser verificadas, por norma, antes de um *deployment* para produção, como já foi explicado anteriormente (ver subsecção 2.10.2)

A constituição do ambiente de UAT pode ser observada na imagem 5.10

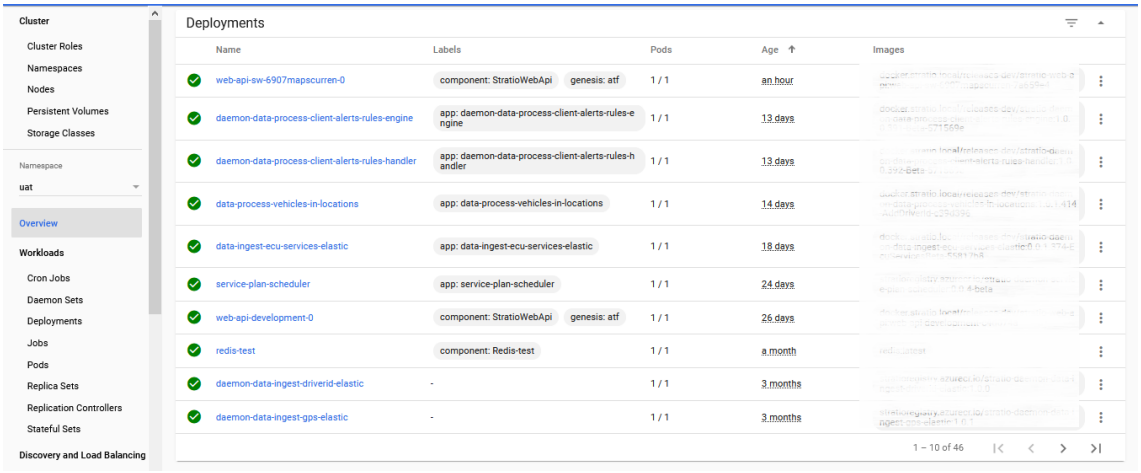

Figura 5.10: *Deployment* do UAT no Kubernetes

Scope de testes

A UI do Stratio-play é constituída por diversas páginas. Este plano de testes irá cobrir a página de início de sessão e a página de ocorrências. Estas duas páginas são consideradas as mais críticas no caso da presença de um bug, isto é, o cliente irá imediatamente aperceber-se caso exista algum problema numa destas páginas. Daí a necessidade de automatização de testes a estas páginas.

Em relação à página de início de sessão foram definidos quatro casos de teste, identificados por um ID.

- L.1 - Verificar que é necessário inserir email ou nome de utilizador para iniciar sessão;
- L.2 - Verificar que é necessário a *password* para iniciar sessão;
- L.3 - Verificar que é necessário um email ou nome de utilizador e *password* válidos para iniciar sessão;

- L.4 - Iniciar sessão com sucesso.

Em relação à página de ocorrências, inicialmente foi necessário fazer um levantamento das funcionalidades desta página de modo a criar *user flows* que cobrissem o maior número de funcionalidades. Na imagem 5.11 observa-se a página de ocorrências.

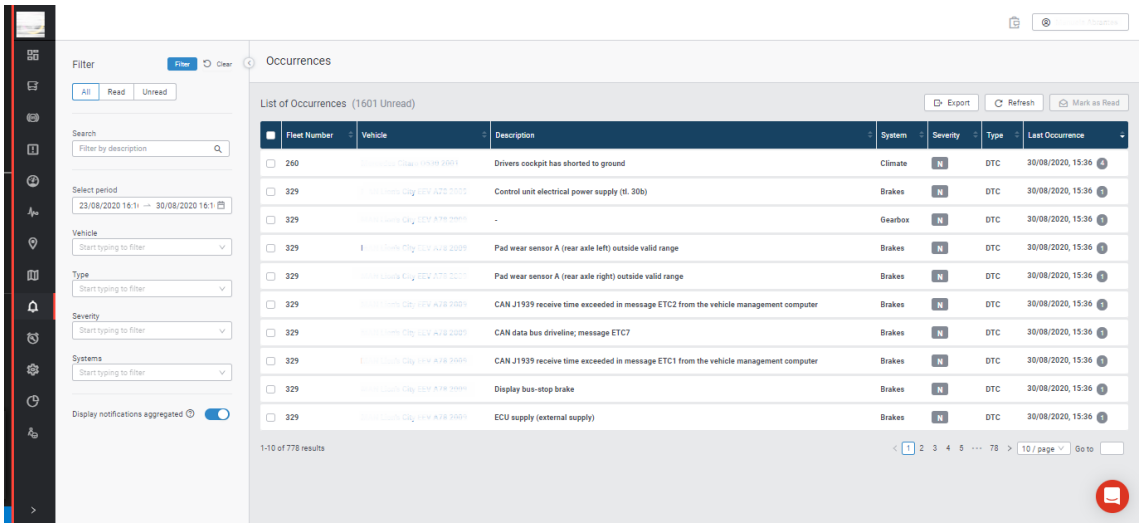

Figura 5.11: Página de ocorrências

Foram definidos três *user flow*, de seguida apresentados:

***User flow* - 1**

1. Iniciar sessão;
2. Clicar na página de ocorrências;
3. Filtrar as ocorrências por veículo;
   (a) Escrever caracteres inválidos;
   (b) Selecionar um veículo válido;
4. Filtrar por período;
   (a) Com recurso ao *popup* do calendário;
   (b) Inserir a data manualmente;
5. Limpar filtros;
6. Selecionar a próxima página com recurso à seta;
7. Filtrar por ocorrências não lidas;
8. Selecionar uma ocorrência e marcar como lida.

***User flow* - 2**

1. Iniciar sessão;
2. Clicar na página de ocorrências;

3. Ordenar as ocorrências por ordem crescente de número de frota;
4. Ordenar as ocorrências por ordem alfabética consoante o tipo da ocorrência;
5. Exportar para pdf;

***User flow* - 3**

1. Iniciar sessão;
2. Clicar na página de ocorrências;
3. Ordenar as ocorrências por ordem decrescente de gravidade da ocorrência;
4. Selecionar todas as ocorrências;
5. Aumentar o número de ocorrências por página;
6. Mudar de página utilizando o *go to*, recorrendo à técnica de particionamento por equivalências para definir para que números mudar a página;
7. Ordenar por ordem alfabética consoante o sistema da ocorrência;1
8. Selecionar os detalhes do veículo.

A matriz de *traceability* exposta na tabela 5.1 relaciona, assim, as funcionalidades da página de ocorrências com os *user flows* definidos, ajudando a compreender que funcionalidades são cobertas pelos *user flows*.

	User flow - 1	*User flow* - 2	*User flow* - 3
Ver ocorrência			
Selecionar detalhes de um veículo			X
Selecionar todas as ocorrências			X
Selecionar uma ocorrência	X		
Avançar/Retroceder para a página seguinte/anterior	X		
Saltar para uma determinada página			X
Mudar o número de ocorrências mostradas por página			X
Marcar ocorrência como lida	X		
Exportar		X	
Ordenar por número da frota		X	
Ordenar por identificação do veículo			
Ordenar por descrição da ocorrência			
Ordenar por sistema da ocorrência			X
Ordenar por severidade da ocorrência			X
Ordenar por tipo da ocorrência		X	
Ordenar por data da última ocorrência			
Filtrar por todas as ocorrências	X	X	X
Filtrar por ocorrências lidas			
Filtrar por ocorrências não lidas	X		
Filtrar por descrição da ocorrência			
Filtrar por período da ocorrência	X		
Filtrar por identificação do veículo	X		
Filtrar por tipo de ocorrência			
Filtrar por severidade da ocorrência			
Filtrar por sistema da ocorrência			
Limpar filtros	X		

Tabela 5.1: Matriz de *traceability* - página de ocorrências

Critério de aceitação

O teste não deve falhar qualquer verificação efetuada. Verificações como a confirmação de que as ocorrências foram ordenadas por um certo campo; quando é ativado algum filtro só são mostradas as respetivas ocorrências; verificações de mudança de página e de escrita de algum valor.

Critério de suspensão

O teste é imediatamente suspenso caso falhe algum tipo de verificação.

5.2.2 Integração com o Ciclo de Vida de Desenvolvimento de Software

O esquema da figura 5.12 mostra quando é feito o deployment para o UAT.

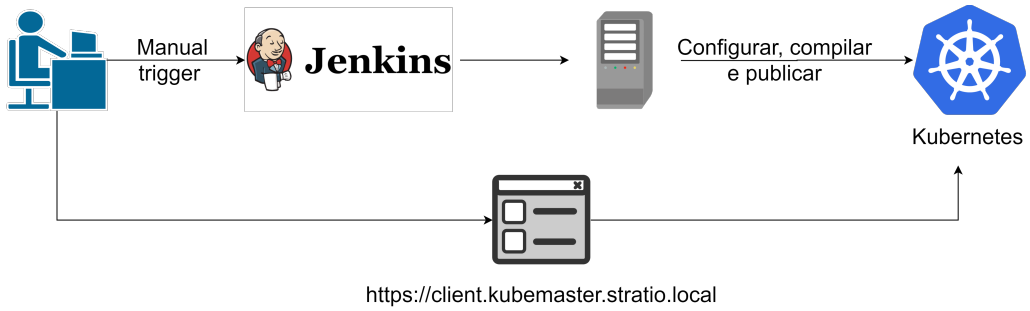

Figura 5.12: Contínuo *deployment* para UAT

O processo de *deploy* do UAT é iniciado manualmente, ou seja, o programador é que determina quando é que o tenciona fazer. Para tal, basta indicar o *branch* e ambiente para qual pretende fazer o *deploy*. Após essa indicação, o Jenkins inicia um *job* num dos *workers* disponíveis que clona o repositório e configura, compila e publica o ambiente no *cluster* Kubernetes.

Por norma, o programador pretende fazer um *deploy* para o UAT quando são adicionadas novas funcionalidades que precisam de ser testadas antes de serem lançadas em produção.

O plano de testes a UI deve ser então executado automaticamente após um *deploy* para o UAT, sendo também capaz de ser executado manualmente sempre que o programador assim o entenda.

5.2.3 Desenvolvimento - Fase inicial

Inicialmente, o Cypress foi instalado localmente como dependência de desenvolvimento do projeto Stratio-play.

Foi criado um *branch* para o épico E2E Tests, dentro deste foram criados *branchs* em conformidade com as tarefas definidas no Jira. Após o desenvolvimento e aprovação dos *merge requests*, este era *merge* para o *branch* do épico que, quando finalizado, foi *merge* para o *branch* de desenvolvimento.

Inicialmente foi criado o ficheiro login.js onde contém os casos de teste definidos para esta página e as respetivas verificações.

Foi criado um comando para o início de sessão válido para evitar a duplicação de código, uma vez que para os seguintes testes é necessário iniciar sessão antes.

Em occurrences.js é implementado todo o código necessário para realizar os *user flows* definidos.

Quando é necessário clicar num elemento, recorre-se ao comando cy.get() que tem como argumento o seletor pretendido e ao cy.click(). Seletores são utilizados para encontrar elementos *Hypertext Markup Language* (HTML) com base no seu nome, ID, classe, tipo, atributos e valores dos atributos.

Posteriormente a uma ação é feita a respetiva verificação de que a ação aconteceu. Por exemplo, quando as ocorrências são ordenadas pelo número de frota do veículo associado, é verificado que as ocorrências são ordenadas por ordem crescente ou decrescente do número de frota.

No passo 6 do *user flow* 3 é recolhida a informação de quantas páginas existem no total e, recorrendo à técnica de particionamento por equivalências (ver a subsecção 2.2.1), são definidas as páginas que devem ser digitadas para posteriormente se observar o comportamento.

Na figura 5.13 observa-se a UI do Cypress a ser executado no *browser* padrão instalado na máquina que no caso é o Google Chrome.

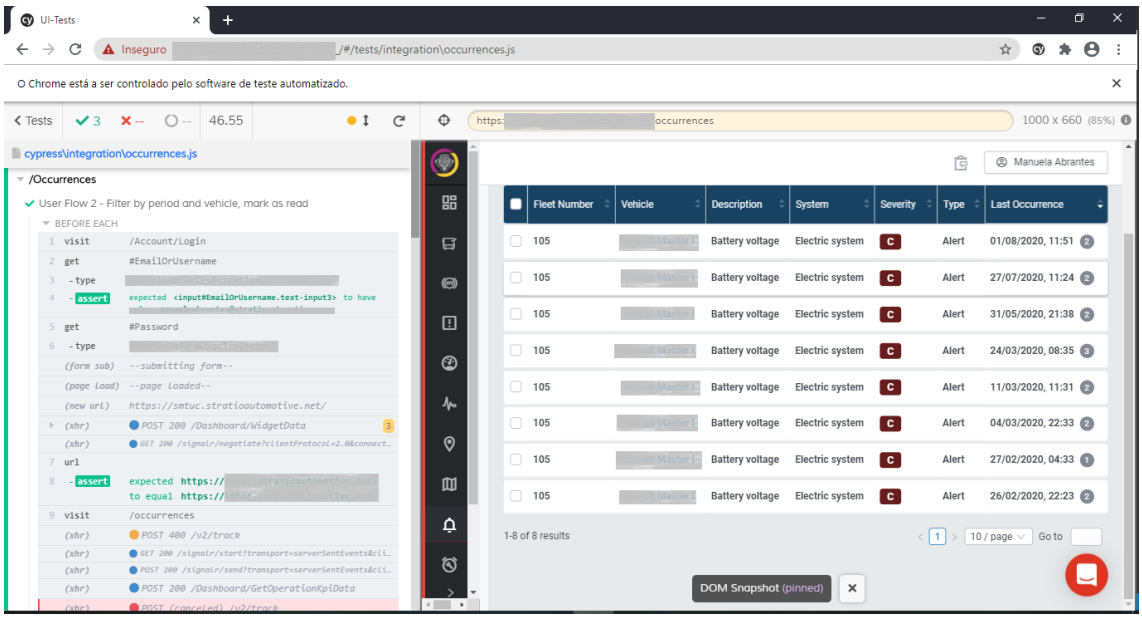

Figura 5.13: Interface do Cypress durante a execução dos testes

5.2.4 Criação de um projeto no Jenkins

De maneira semelhante à mencionada na subsecção 5.1.5 foi criado um item do tipo *Freestyle project*.

Foi necessário adicionar o *plugin* NodeJs e instalar o NodeJs, bem como algumas depen-

dências necessárias nas máquinas Linux do Jenkins, uma vez que é nestas que o projeto será executado.

O repositório é clonado, repositório esse que contém o código desenvolvido para a automatização dos testes *end-to-end*. A opção de parametrização do projeto é adicionada com dois parâmetros: o URL contra qual os testes serão executados e o *browser* onde serão executados os testes. Na secção de ambiente de construção é configurado o ambiente NodeJs para executar o Cypress. Posteriormente, na secção *build* é escolhida a opção '*execute shell*', onde será adicionado o comando cypress, com os dois parâmetros definidos anteriormente, para executar os ficheiros de teste. No final da execução é possível observar que testes passaram e falharam e quanto tempo demorou a execução. É ainda gerado um vídeo com a gravação da execução do teste no *browser*.

Capítulo 6

Validação e Resultados

No presente capítulo proceder-se-á à exposição de métodos de validação e é realizada uma avaliação do estado do projeto.

Por último são mostrados os resultados obtidos para ambos os épicos.

6.1 Validação

Esta secção visa apresentar os processos utilizados para validação do projeto, bem como os casos de testes definidos para cada requisito. É ainda feita uma avaliação dos casos de teste definidos, de modo a perceber que requisitos e tarefas foram implementadas.

6.1.1 Revisão de código

Cada tarefa de código desenvolvida, como já foi referido, era submetida a um *merge request*. Caso este não fosse aprovado, a tarefa voltava à fase de desenvolvimento e só quando aprovado é que poderia ser *merge* para o *branch* principal do épico.

Após a abertura de um *merge request*, o responsável designado, neste caso o orientador da empresa, revia e analisava o código em detalhe. Desta forma, podiam ser encontrados problemas não detetados anteriormente e até sugestões de melhorias, melhorando assim a qualidade do código.

Na figura 6.1 visualiza-se um exemplo de *merge request*.

6.1.2 Testes aos Requisitos Funcionais

Para cada requisito funcional definido foi identificado um ou um conjunto de casos de testes para permitir a sua validação.

No caso do épico API Perf Tests, alguns destes casos de teste foram realizados recorrendo ao tipo de teste de mutação (ver subsecção 2.4.4). Outros deles foram verificados com recurso a outras ferramentas, como o Conduktor [148] que permite controlar as mensagens recebidas no tópico Kafka bem como o conteúdo da mensagem.

No caso do épico E2E Tests, os casos de teste foram maioritariamente confirmados com recurso às funcionalidades de *snapshot*, *screenshot* e de vídeo que o Cypress oferece.

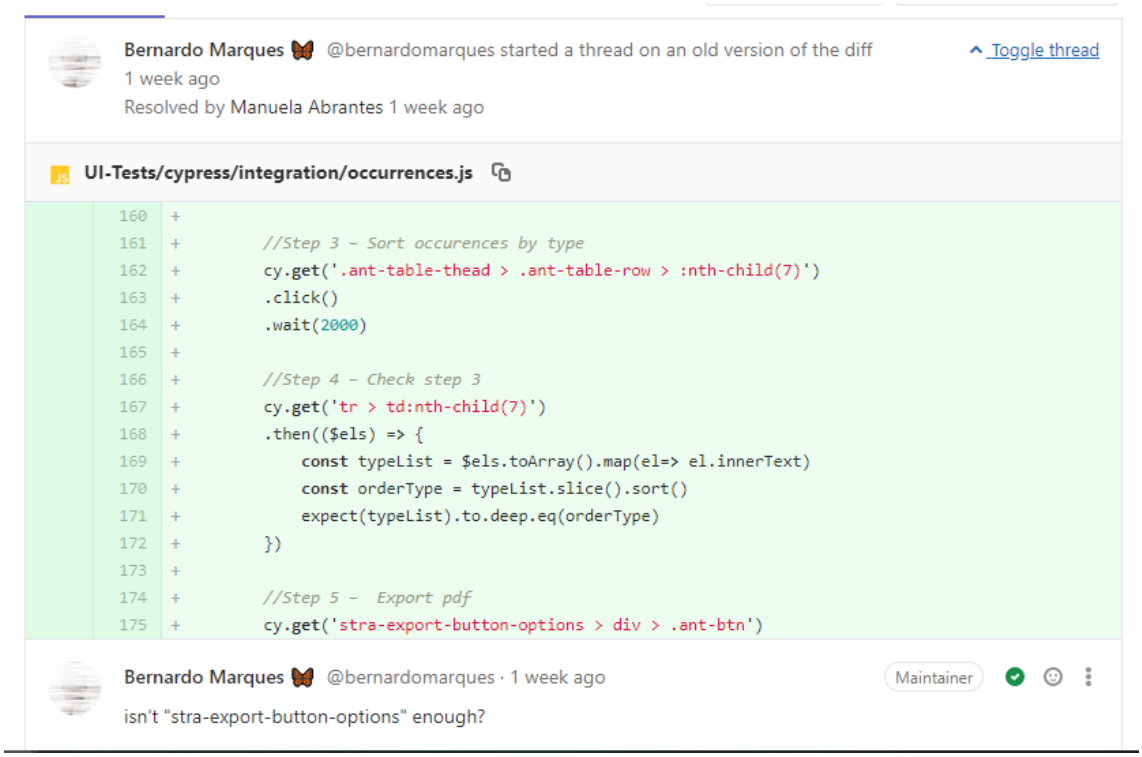

Figura 6.1: Exemplo de um *merge request*

De seguida apresentam-se os casos de teste definidos para cada requisito:

TF.1 Teste ao requisito funcional RF.1 - A bateria de testes deve ser capaz de gerar um relatório de desempenho para cada execução que inclua informações do mínimo de utilização de *Central Processing Unit* (CPU) e *Random Access Memory* (RAM), taxa de transferência e latência obtidos

- Verificar que a bateria de testes é capaz de reportar o mínimo de utilização de CPU;
- Verificar que a bateria de testes é capaz de reportar o mínimo de utilização de RAM;
- Verificar que a bateria de testes é capaz de reportar a taxa de transferência;
- Verificar que a bateria de testes é capaz de reportar o tempo de resposta.

TF.2 Teste ao requisito funcional RF.2 - A bateria de testes deve ser capaz de validar os requisitos da componente Jam *Application Programming Interface* (API) descritos na tabela 3.5

- Verificar que a ordem de chegada das mensagens à Jam API tem a data e hora da mensagem (dt) crescente por peça;
- Verificar que não existe perda de mensagens;
- Verificar que é apresentado o tempo de resposta para o percentil 99.9%;
- Verificar que é apresentado o tempo de resposta para o percentil 99.99%.

TF.3 Teste ao requisito funcional RF.3 - A bateria de testes deve ser incluída no Ciclo de Vida de Desenvolvimento de Software (CVDS), no sentido de dar feedback aquando alterações no código base da API

- Verificar que quando é feito um *merge* para o *branch* de desenvolvimento no repositório relativo à Jam API a bateria de testes de desempenho é executada automaticamente.

TF.4 Teste ao requisito funcional RF.4 - A bateria de testes deve ser passível de ser executada contra qualquer *branch* da Jam API através de uma *pipeline* do Jenkins

- Verificar que é possível executar a bateria de testes de desempenho através de uma *pipeline* do Jenkins;
- Verificar que é possível correr a bateria de testes de desempenho contra a um dado *branch*.

TF.5 Teste ao requisito funcional RF.5 - A bateria de testes deve ser capaz de validar de forma automática os *user flows* definidos na subsecção 5.2.1

- Verificar que todos os passos definidos para os *user flows*, incluindo as verificações, são exercidos.

TF.6 Teste ao requisito funcional RF.6 - A bateria de testes deve ser capaz de reportar o sucesso ou não, do teste e indicar onde ocorreu a falha

- Verificar que a bateria de testes *end-to-end* reporta o sucesso ou não dos testes definidos;
- Verificar que a bateria de testes *end-to-end* reporta onde ocorre a falha, no caso do insucesso do teste.

TF.7 Teste ao requisito funcional RF.7 - A bateria de testes deve ser passível de ser executada contra qualquer *Uniform Resource Locator* (URL) numa *pipeline* do Jenkins

- Verificar que é possível executar a bateria de testes *end-to-end* através de uma *pipeline* do Jenkins;
- Verificar que é possível correr a bateria de testes *end-to-end* contra um dado URL.

TF.8 Teste ao requisito funcional RF.8 - A bateria de testes deve ser incluída no CVDS, no sentido de dar feedback de forma contínua

- Verificar que quando é feito um *deployment* de um ambiente *User Acceptance Testing* (UAT) a bateria de *end-to-end* é executada automaticamente.

6.1.3 Testes aos requisitos não funcionais

Para cada requisito não funcional definido foi identificado um ou um conjunto de casos de testes para permitir a validação do requisito.

Alguns dos casos de teste identificados, foram verificados no *merge request*, como por exemplo, o requisito não funcional de usabilidade TNF.3 que só foi possível verificar pelo parecer do orientador da empresa. Outros foram verificados pela execução da bateria de testes em sim, como o caso do requisito não funcional de funcionalidade TNF.1.

O teste do requisito a confiabilidade TNF.2 só permite obter resposta durante a execução das baterias de testes durante um período tempo longo e ,como tal, ainda não é possível confirmar.

De seguida enumeram-se os casos de teste definidos para cada requisito:

TNF.1 Teste ao requisito não funcional de funcionalidade

- Verificar que as baterias de teste estão em conformidade com objetivos definidos para cada uma delas.

TNF.2 Teste ao requisito não funcional de confiabilidade

- Verificar que as baterias de teste conseguem manter o seu desempenho sob determinadas condições por um determinado tempo.

TNF.3 Teste ao requisito não funcional de Usabilidade

- Verificar que os ficheiros README associados a cada bateria de testes, expliquem de forma perceptível como devem ser executadas, no caso de se pretender correr localmente;
- Verificar a compreensão dos planos de testes definidos, por parte de outros utilizadores.

TNF.4 Teste ao requisito não funcional de eficiência

- Verificar que o tempo de execução das baterias de teste é inferior a 10 minutos.

TNF.5 Teste ao requisito não funcional de manutenção

- Verificar que o código segue os princípios *Single Responsibility Principle, Open/Closed Principle, Liskov Substitution Principle, Interface Segregation Principle, Dependency Inversion Principle* (SOLID).

TNF.6 Teste ao requisito não funcional de portabilidade

- Verificar que as baterias de teste são passíveis de ser executadas com outras especificações, nomeadamente, noutros sistemas operativos e com outras parameterizações do teste.

TNF.7 Teste ao requisito não funcional de auditabilidade

- Verificar que as baterias de testes são capazes de reportar informação acerca da execução destas.

TNF.8 Teste ao requisito não funcional de escabilidade

- Verificar que tanto a bateria de testes de desempenho, como a bateria de testes *end-to-end* são possíveis de correr em paralelo.

6.1.4 Avaliação

A tabela 6.1 apresenta os resultados dos teste aos requisitos funcionais e a tabela 6.2 aos requisitos não funcionais.

Teste	TF.1	TF.2	TF.3	TF.4	TF.5	TF.6	TF.7	TF.8
Resultado	✓	✓	X	✓	✓	✓	✓	✓

Tabela 6.1: Resultados dos testes aos requisitos funcionais

Teste	TNF.1	TNF.2	TNF.3	TNF.4	TNF.5	TNF.6	TNF.7	TNF.8
Resultado	✓	*	✓	✓	✓	✓	✓	✓

Tabela 6.2: Resultados dos testes aos requisitos não funcionais
✓- Passou
X - Falhou
* - Sem informação disponível

Todos os requisitos com prioridade *must have* foram implementados e validados com sucesso, observando-se situação semelhante para os dois requisitos com prioridade *should have*. Em relação aos dois requisitos com prioridade *could have*, um deles não foi implementado - RF.3.

Em relação aos requisitos não funcionais, um deles ainda não pode ser verificado (TNF.2), observando-se que os restantes foram validados e aprovados.

Em relação às tarefas criadas no Jira (ver anexo E) para implementação dos requisitos, é importante fazer uma análise destas.

Todas as tarefas com prioridade crítica foram realizadas.

A tarefa SW-5218 e SW-6897, duas tarefas com prioridade baixa, vistas como melhorias, serão idealmente implementadas num trabalho futuro.

A tarefa SW-6180 diz respeito ao requisito funcional RF.3, requisito esse que não foi implementado.

A tarefa SW-5150 teve bastantes alterações devido à versão do Kubernetes que a empresa utiliza. Inicialmente, o plano de testes foi definido da seguinte forma: limitar a utilização do CPU e RAM do *container* da Jam API no ambiente lançado no Kubernetes e ir baixando estes valores de modo a relacionar a capacidade de resposta da Jam API com os recursos desta.

A versão do Kubernetes tem um problema já reportado ([4]) e resolvido em versões seguintes: não é possível limitar a utilização de CPU em *containers* Windows. Devido a restrições de *roadmap* da empresa, a versão não foi atualizada e, como tal, foi necessário repensar a estratégia de execução dos testes. Foi então criada a tarefa SW-6053 em que à medida que a carga é injetada na Jam API são recolhidas métricas de utilização de CPU e RAM para, posteriormente, serem retiradas conclusões.

A tarefa SW-6048 foi cancelada, uma vez que foi detetado um problema por elementos na empresa noutro contexto quando era lançado um ambiente no Kubernetes. O ambiente lançado através da *Automated Tests Framework* (ATF) para a realização dos testes de desempenho à Jam API necessita, para além de lançar o seu *container*, de todas as suas dependências (tal como referido na subsecção 5.1.1). Uma das dependências é um *container* Redis, onde foi detetada uma falha na comunicação com este. Tal correção não foi resolvida durante o decorrer do estágio. Como tal, os valores retirados das baterias de teste de desempenho podem não ser os mais adequados. Visto isto, definir asserções com base em métricas que podem não estar totalmente corretas não fazia sentido, e como tal a tarefa não foi realizada. Os resultados das execuções da bateria de teste até então efetuadas não são apresentadas neste documento pelo motivo referido anteriormente.

A tarefa SW-5525 não estava inicialmente planeada mas foi efetuada, uma vez que a empresa pretende migrar a Jam API do Azure para *Amazon Web Services* (AWS). Foi então executada a bateria de testes de desempenho à Jam API na AWS, permitindo retirar informações de valor acerca da capacidade de resposta da API e informações de recursos e infraestrutura para o elemento de *Information Technology* (IT).

6.2 Resultados épico API Perf Tests

Esta secção pretende apresentar os resultados relativos à validação da Jam API na AWS, bem como retirar conclusões acerca destes. Para a realização dos testes é necessário ter contexto de determinadas métricas atualmente na empresa.

À data da realização das seguintes baterias de testes, a empresa possuía cerca de 3300 peças instaladas nos veículos e chegavam cerca de 80 mensagens por segundo. Prevê-se que, até ao final do corrente ano, tenham a totalidade de 8000 peças instaladas, o que equivale aproximadamente a 200 mensagens por segundo.

É necessário também perceber a relação entre alguns dos parâmetros de *input* para melhor percepção das conclusões face aos resultados obtidos.

$$Número\,total\,de\,amostras = Threads \times Iterations$$

$$Taxa\,de\,transferência\,expectável = Threads \times throughputSamplesMinute$$

Como já mencionado, a empresa pretende fazer a migração da Jam API do Azure para a AWS, como tal, a bateria de testes de desempenho foi executada neste ambiente. O objetivo da execução desta bateria de testes contra ao ambiente na AWS prende-se com a observação dos recursos de utilização de CPU e RAM quando a Jam API é submetida a uma determinada carga de trabalho. Uma vez que a Jam API lançada na AWS suporta tanto o protocolo *HyperText Transfer Protocol* (HTTP) como *HyperText Transfer Protocol Secure* (HTTPS) e o ficheiro de configuração jmx desenvolvido no JMeter só suportava HTTP, foi

feita uma ligeira alteração de modo a que fosse possível passar o protocolo por parâmetro. Assim, quando for necessário executar com um ou outro protocolo, basta adicionar o parâmetro *protocol* ao comando de execução bzt, além dos parâmetros já mencionados na secção 5.1.3 - *concurrency*, *iterations*, *domain* e *throughputSamplesMinute*.

Importa referir que a bateria de testes realizada nos dois cenários foi executada através da pipeline do Jenkins. Em ambos os cenários o *dataset* enviado é o mesmo, isto é, as mensagens enviadas em cada pedido para Jam API de cada teste de cada cenário são as mesmas, com cerca de 70 *Parameter IDs* (PIDs) - equivalente a uma mensagem comprimida com *heatshrink*.

Foram então definidos dois cenários: o cenário 1 em que os pedidos realizados à Jam API são via HTTP, e o cenário 2 em que os pedidos são HTTPS. Os resultados dos testes realizados no cenário 1 e 2 são expostos nas tabelas G.1 e G.2, respetivamente, do anexo G.

O *input* apresentado nas tabelas consiste nos seguintes parâmetros:

- Concurrency - Número de *threads*, isto é, número de peças;
- Iterations - Número de execuções de teste;
- Throughput - Número de amostras a injetar por segundo.

O *output* consiste em:

- Avg. Throughput - Média de taxa de transferência, mensagem/segundo, que efectivamente foi atingida;
- Máximo Throughput (msg/s) - Máximo de taxa de transferência, mensagem/segundo, que foi atingida;
- Avg. Resp. Time - Média de tempo de resposta, em segundos;
- Final Avg. Th. - Média de taxa de transferência, mensagem/segundo, das três execuções;
- Final Avg. Resp. Time - Média de tempo de resposta, em segundos, das três execuções.

Foram feitas três execuções para cada *input* para que, deste modo, os resultados tenham significância estatística. Aumentando o número de execuções de cada teste a aleatoriedade dos resultados terá menos impacto nas conclusões finais retiradas. Deste modo os resultados retirados são mais fidedignos e ponderados.

A bateria de testes permitiu assim que o elemento da equipa de IT pudesse controlar a utilização dos recursos na AWS, como CPU e RAM, à medida que os pedidos eram efetuados. Deste modo, o elemento de IT conseguiu retirar conclusões à cerca das limitações dos recursos e como estes eram influenciados pela quantidade de mensagens injetadas.

De notar, que em todos os testes efetuados, os valores de latência e tempo de resposta eram iguais.

Na tabela G.1 é possível observar que com 125 *threads* e com o valor de taxa de transferência definido de 5 mensagens por segundo era expectável que o resultado da taxa de transferência chegasse a 625 mensagens por segundo (125 x 5), como se pode observar em média chega às

302 mensagens por segundo. Como já mencionado anteriormente, o valor de *input* definido de taxa de transferência é o valor que cada *thread* irá tentar atingir, sendo que nem sempre é possível.

Quando o número de iterações aumenta é possível observar um aumento da média do tempo de resposta, uma vez que aumenta o número de amostras enviadas a Jam API. O aumento do número de iterações aumenta o tempo de execução do teste o que torna impraticável temporalmente executar 500 iterações com mais de 125 *threads*, uma vez que o teste começa a demorar cerca de 15 minutos a executar.

Após a execução destes testes, foi possível verificar que existe um *overhead* de sincronização entre *threads* por parte da configuração efetuada no JMeter. Isto deve-se ao facto de que cada *thread* tem de passar pelos seguintes passos:

1. Ler o ficheiro que contém as informações da chave do cliente e *IDentifier* (ID) da peça, clk e dvc respectivamente;
2. Definir o campo dt;
3. Definir o campo seq;
4. Substituir as variáveis na mensagem *JavaScript Object Notation* (JSON)
5. Enviar o pedido HTTP ou HTTPS, conforme o caso;
6. Esperar a resposta ao pedido efetuado.

Desta forma, concluiu-se que o número de *threads* tem um limite, isto é, a partir de um certo número de *threads* o JMeter deixa de responder ou atinge taxas de transferência muito reduzidas.

Como o principal objetivo deste teste era observar a utilização dos recursos da Jam API na AWS, as verificações não eram necessárias. Para atingir uma maior taxa de transferência, um teste foi executado simultaneamente nos três *workers* do Jenkins, desta forma consegue-se triplicar a carga de um teste. Com 125 threads, 300 iterações e 5 mensagens por segundo em cada *worker*, o que equivale a um total de 375 threads, 900 iterações e 15 mensagens por segundo. Com o protocolo HTTP o resultado obtido para média de taxa de transferência foi de 593 mensagens/segundo. Com o protocolo HTTPS o resultado obtido para média de taxa de transferência foi 547 mensagens/segundo.

6.3 Resultados épico E2E Tests

A bateria de testes referente ao épico E2E Tests foi executada através da pipeline do Jenkins.

Os resultados obtidos da página de início de sessão, são apresentados na tabela 6.3, sendo identificados pelo ID definido anteriormente na subsecção 5.2.1.

ID	L.1	L.2	L.3	L.4
Resultado	✓	✓	✓	✓

Tabela 6.3: Resultados dos testes à pagina de início de sessão
✓- Passou
X - Falhou

Os resultados obtidos da página de ocorrências, são apresentados na tabela 6.4, sendo identificados pelo *user flow* e passo definido anteriormente na subsecção 5.2.1.

User Flow	1										2					3							
Passo	1	2	3a	3b	4a	4b	5	6	7	8	1	2	3	4	5	1	2	3	4	5	6	7	8
Resultado	✓	✓	✓	✓	✓	X	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Tabela 6.4: Resultados dos testes à pagina de ocorrências
✓- Passou
X - Falhou

Como é possível verificar, no *user flow* 1 no passo 4b, o teste falhou. Verificou-se que existia um problema, quando a data é introduzida manualmente e posteriormente se clica no botão de filtrar, verifica-se que a data não atualiza e mantém data *default*. Posteriormente foi seguido o processo para abertura de bugs no Jira, para este ser corrigido pela equipa responsável.

Esta página foi propositadamente deixada em branco.

Capítulo 7

Conclusão

A primeira fase deste projeto, que decorreu no primeiro semestre, foi dedicada aos processos de gestão de um projeto.

Inicialmente foram definidos em concreto os objetivos deste estágio, bem como o planeamento do mesmo.

De seguida iniciou-se a fase de investigação em técnicas, níveis e tipos de teste e da metodologia de desenvolvimento e operações, bem como das suas práticas de *Continuous Integration* (CI)/*Continuous Delivery* (CD).

Subsequentemente seguiu-se o levantamento dos requisitos. Este processo exigiu constante comunicação entre a aluna e a empresa, de modo a que as diversas iterações dos requisitos permitissem a chegada à iteração final em conformidade com as necessidades da empresa.

Por último foi definida a arquitetura e foram estabelecidos os riscos associados ao projeto. Foi apresentada a respetiva matriz de exposição de riscos e estabelecidos planos de mitigação para cada um.

No começo do segundo semestre foram feitas diversas alterações para colmatar as vulnerabilidades apontadas ao projeto na defesa intermédia deste.

Precedeu-se à fase de implementação, a escolha da ferramenta mais adequada para a automatização dos testes de desempenho e outra ferramenta para a automatização de testes *end-to-end* a uma *User Interface* (UI).

A última fase de validação permitiu verificar o trabalho desenvolvido e garantir a qualidade deste.

Concluindo, foi um projeto que fomentou em todos os momentos a aprendizagem da aluna, com novos conceitos e tecnologias e adaptação a um ambiente empresarial.

7.1 Trabalho futuro

Existe muito que pode ser feito no âmbito da automação de garantia de qualidade na Stratio. Existe inclusive um plano de ação já definido, com diversas tarefas ordenadas por prioridade que devem ser realizadas pela equipa de controle de qualidade e de desenvolvimento, com a finalidade de melhorar a qualidade do software.

Em relação aos épicos realizados neste estágio, existem diversas melhorias que podem ser

feitas.

Em relação ao épico API Perf Tests:

- À bateria de testes adicionar a possibilidade de escolher o formato das mensagens a injetar na Jam *Application Programming Interface* (API) - *JavaScript Object Notation* (JSON), JSON comprimido *heatshrink*, binário ou binário comprimido com *heatshrink* (tarefa SW-5218 no anexo E);
- Testar os restantes *endpoints* da Jam API, tendo em conta que, cada um deles tem uma funcionalidade diferente e os campos da mensagem são também diferentes;

Em relação ao épico E2E Tests:

- Adicionar melhorias ao comando de *login* (tarefa SW-6720 no anexo E). Em vez de se repetirem os passos de início de sessão, seria feito um pedido para o mesmo *endpoint* e guardado o *token* do utilizador, deste modo o tempo de execução dos testes seria reduzido significativamente;
- Adicionar o caso de recuperação da palavra chave;
- Definir e implementar *user flows* para as restantes páginas da UI;
- Definir um acordo de nomenclatura dos seletores com a equipa de *frontend*, de modo a tornar mais simples de encontrar os elementos para quem automatiza os testes à UI.

Referências

[1] "Stratio Automotive." [Online]. Available: https://stratioautomotive.com/ (Acedido em 2020-01-05).

[2] "Principles behind the Agile Manifesto." [Online]. Available: https://agilemanifesto.org/principles.html (Acedido em 2019-11-05).

[3] "Online Gantt Chart Software for Project Planning | GanttPRO." [Online]. Available: https://ganttpro.com/# (Acedido em 2019-10-30).

[4] "CPU resource limit is not working with windows containers · Issue #75296 · kubernetes/kubernetes." [Online]. Available: https://github.com/kubernetes/kubernetes/issues/75296 (Acedido em 2020-08-06).

[5] "What Is a 5-15 Report? Spoiler — It's a Report for People Who Have No Time for Reports - CO2 Partners." [Online]. Available: https://www.co2partners.com/515-report-increasing-effectiveness-200-percent/ (Acedido em 2020-01-18).

[6] "Estágios @ DEI - FCTUC." [Online]. Available: http://estagios.dei.uc.pt/ (Acedido em 2019-12-26).

[7] "What is Quality Assurance(QA)? Process, Methods, Examples." [Online]. Available: https://www.guru99.com/all-about-quality-assurance.html (Acedido em 2020-01-20).

[8] "The Four Steps of a Quality Assurance Cycle." [Online]. Available: https://blog.jrecorp.com/blog/the-four-steps-of-a-quality-assurance-cycle (Acedido em 2020-01-20).

[9] "What is BLACK Box Testing? Techniques, Example & Types." [Online]. Available: https://www.guru99.com/black-box-testing.html (Acedido em 2019-11-13).

[10] "What is Equivalence partitioning in Software testing?" [Online]. Available: http://tryqa.com/what-is-equivalence-partitioning-in-software-testing/ (Acedido em 2020-08-29).

[11] "What is Boundary value analysis and Equivalence partitioning?" [Online]. Available: https://www.softwaretestinghelp.com/what-is-boundary-value-analysis-and-equivalence-partitioning/ (Acedido em 2020-08-29).

[12] "Certified Tester Foundation Level Syllabus." [Online]. Available: https://www.istqb.org/downloads/send/51-ctfl2018/208-ctfl-2018-syllabus.html (Acedido em 2019-11-13).

[13] "White Box Testing - Software Testing Fundamentals." [Online]. Available: http://softwaretestingfundamentals.com/white-box-testing/ (Acedido em 2019-11-13).

[14] "What is Exploratory Testing? Techniques with Examples." [Online]. Available: https://www.guru99.com/exploratory-testing.html (Acedido em 2020-08-05).

[15] "Scrum Academy :: International Agile Tester Foundation - Chapter 7 - Agile Testing Methods." [Online]. Available: https://www.scrum.as/academy.php?show=2{&}chapter=7 (Acedido em 2019-11-23).

[16] "Unit Testing Tutorial: What is, Types, Tools, EXAMPLE." [Online]. Available: https://www.guru99.com/unit-testing-guide.html (Acedido em 2019-11-23).

[17] "Integration Testing - Software Testing Fundamentals." [Online]. Available: http://softwaretestingfundamentals.com/integration-testing/ (Acedido em 2019-11-26).

[18] "Acceptance Testing - Software Testing Fundamentals." [Online]. Available: http://softwaretestingfundamentals.com/acceptance-testing/ (Acedido em 2019-11-27).

[19] "ISO 25010." [Online]. Available: https://iso25000.com/index.php/en/iso-25000-standards/iso-25010?limit=3{&}start=6 (Acedido em 2019-11-13).

[20] "Performance Testing Tutorial: What is, Types, Metrics & Example." [Online]. Available: https://www.guru99.com/performance-testing.html (Acedido em 2020-01-14).

[21] "Performance Testing Types, Steps, Best Practices, and Metrics." [Online]. Available: https://stackify.com/ultimate-guide-performance-testing-and-software-testing/ (Acedido em 2020-08-23).

[22] "Performance Testing Types, Steps, Best Practices, and Metrics." [Online]. Available: https://stackify.com/ultimate-guide-performance-testing-and-software-testing/ (Acedido em 2020-01-15).

[23] "Performance Testing - Tools, Steps, and Best Practices - KeyCDN." [Online]. Available: https://www.keycdn.com/blog/performance-testing (Acedido em 2020-01-15).

[24] "Don't do performance testing in production environments." [Online]. Available: https://techbeacon.com/app-dev-testing/dont-do-performance-testing-production-environments-only (Acedido em 2020-08-23).

[25] "Load Testing With KPIs Part I: What Are KPIs? - DZone Performance." [Online]. Available: https://dzone.com/articles/load-testing-with-kpis-part-1-what-are-kpis (Acedido em 2020-08-23).

[26] "Timeline Report – BlazeMeter." [Online]. Available: https://guide.blazemeter.com/hc/en-us/articles/206733919-Timeline-Report-Timeline-Report (Acedido em 2020-08-26).

[27] "End To End Testing: A Detailed Guide | BrowserStack." [Online]. Available: https://www.browserstack.com/guide/end-to-end-testing (Acedido em 2020-08-28).

[28] "End-to-End Testing with Cypress.io — A Lowdown." [Online]. Available: https://madebyextreme.com/insights/end-to-end-testing-with-cypress-io (Acedido em 2020-08-28).

[29] "What Is End to End Testing? A Helpful Introductory Guide." [Online]. Available: https://www.testim.io/blog/end-to-end-testing-guide/ (Acedido em 2020-08-28).

[30] "The Difference: User Flows vs User Journeys - Product School." [Online]. Available: https://productschool.com/blog/product-management-2/user-flows-vs-user-journeys/ (Acedido em 2020-08-28).

[31] A. F. Aristides Dasso, "Fault-Based Testing," Tech. Rep., 2006.

[32] L. Miguel Agante Gonçalo, R. André Brajczewski Barbosa, and R. Lúcia de, "Software Verification Aimed at Security Vulnerabilities," Universidade de Coimbra, Tech. Rep., nov 2019. [Online]. Available: http://hdl.handle.net/10316/88060

[33] "Mutation Testing in Software Testing: Mutant Score & Analysis Example." [Online]. Available: https://www.guru99.com/mutation-testing.html (Acedido em 2020-09-02).

[34] "DevOps For Dummies - Emily Freeman - Google Livros." [Online]. Available: https://books.google.pt/books/about/DevOps_For_Dummies.html?id=JNT8uQEACAAJ&redir_esc=y (Acedido em 2019-11-05).

[35] "Understanding the DevOps Process Flow | Lucidchart Blog." [Online]. Available: https://www.lucidchart.com/blog/devops-process-flow (Acedido em 2020-01-11).

[36] "DevOps Lifecycle | Detailed Concept of DevOps Lifecycle." [Online]. Available: https://www.educba.com/devops-lifecycle/ (Acedido em 2020-01-13).

[37] "DevOps Life cycle - Explore About Each Phase in DevOps Life cycle." [Online]. Available: https://medium.com/edureka/devops-lifecycle-8412a213a654 (Acedido em 2020-01-14).

[38] "Continuous Integration." [Online]. Available: https://martinfowler.com/articles/continuousIntegration.html (Acedido em 2020-01-04).

[39] "What Is Continuous Integration and Why Do You Need It? - Code Maze." [Online]. Available: https://code-maze.com/what-is-continuous-integration/ (Acedido em 2020-01-05).

[40] "Continuous integration vs. continuous delivery vs. continuous deployment." [Online]. Available: https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment (Acedido em 2020-01-15).

[41] "What Is Continuous Deployment? | Atlassian." [Online]. Available: https://www.atlassian.com/continuous-delivery/continuous-deployment (Acedido em 2020-01-15).

[42] "Continuous Delivery | Get started with CI/CD | Atlassian." [Online]. Available: https://www.atlassian.com/continuous-delivery (Acedido em 2020-01-15).

[43] "Understanding the DevOps Process Flow | Lucidchart Blog." [Online]. Available: https://www.lucidchart.com/blog/devops-process-flow (Acedido em 2020-01-15).

[44] "What is Jenkins? | Jenkins For Continuous Integration | Edureka." [Online]. Available: https://www.edureka.co/blog/what-is-jenkins/ (Acedido em 2020-06-22).

[45] J. U. Handbook, "Jenkins User Handbook," Tech. Rep.

[46] "Glossary." [Online]. Available: https://www.jenkins.io/doc/book/glossary/ (Acedido em 2020-06-12).

[47] J. Ferguson and S. Hudson, "The Definitive Guide," Tech. Rep., 2011.

[48] "What is Docker? | IBM." [Online]. Available: https://www.ibm.com/cloud/learn/docker (Acedido em 2020-03-24).

[49] "Containers - Kubernetes." [Online]. Available: https://kubernetes.io/docs/concepts (Acedido em 2020-04-29).

[50] "What is Docker?" [Online]. Available: https://www.redhat.com/en/topics/containers/what-is-docker (Acedido em 2020-03-24).

[51] "What is Docker? | Opensource.com." [Online]. Available: https://opensource.com/resources/what-docker (Acedido em 2020-03-26).

[52] J. Willis, "Docker and the three ways of devops," Tech. Rep., 2015. [Online]. Available: https://goto.docker.com/rs/929-FJL-178/images/20150731-wp_docker-3-ways-devops.pdf

[53] "What is Kubernetes?" [Online]. Available: https://www.redhat.com/en/topics/containers/what-is-kubernetes (Acedido em 2020-04-29).

[54] "Kubernetes - Bauman National Library." [Online]. Available: https://en.bmstu.wiki/index.php?title=Kubernetes&mobileaction=toggle_view_desktop (Acedido em 2020-06-29).

[55] R. Filipe Rama Silva, "Plataforma Escalável de Monitorização para Ambientes Cloud," Universidade de Coimbra, Tech. Rep., apr 2019. [Online]. Available: http://hdl.handle.net/10316/86356

[56] "Arquitetura do cluster | Documentação do Kubernetes Engine." [Online]. Available: https://cloud.google.com/kubernetes-engine/docs/concepts/cluster-architecture (Acedido em 2020-05-22).

[57] "Kubernetes For Developers Part 2 – Replica Sets and Deployments | nirmata." [Online]. Available: http://nirmata.com/2018/03/03/kubernetes-for-developers-part-2-replica-sets-and-deployments/ (Acedido em 2020-05-20).

[58] "Kubernetes Deployment Tutorial with YAML - Kubernetes Book." [Online]. Available: https://matthewpalmer.net/kubernetes-app-developer/articles/kubernetes-deployment-tutorial-example-yaml.html (Acedido em 2020-05-19).

[59] "Apache Kafka." [Online]. Available: https://kafka.apache.org/ (Acedido em 2019-12-03).

[60] "Kafka hands-on Guide to using publish-subscribe based messaging system (PART I)," [Online] https://medium.com/@vrushaliraut1234/kafka-hands-on-guide-to-using-publish-subscribe-based-messaging-system-part-i-cda9eada3b06. (Acedido em 2020-06-03).

[61] "Part 1: Apache Kafka for beginners - What is Apache Kafka? - CloudKarafka, Apache Kafka Message streaming as a Service." [Online]. Available: https://www.cloudkarafka.com/blog/2016-11-30-part1-kafka-for-beginners-what-is-apache-kafka.html (Acedido em 2020-06-03).

[62] "Top Apache Kafka Terminologies and Concepts - DataFlair." [Online]. Available: https://data-flair.training/blogs/kafka-terminologies/ (Acedido em 2020-06-07).

[63] "Apache Kafka Architecture - javatpoint." [Online]. Available: https://www.javatpoint.com/apache-kafka-architecture (Acedido em 2020-06-07).

[64] "Apache Kafka - Terminology - Ruwan Sri Wickaramarathna - Medium." [Online]. Available: https://medium.com/@ruwansriw/apache-kafka-terminology-f0b5350c26e4 (Acedido em 2020-06-17).

[65] "Kafka Architecture and Its Fundamental Concepts - DataFlair." [Online]. Available: https://data-flair.training/blogs/kafka-architecture/ (Acedido em 2020-06-08).

[66] "Apache ZooKeeper." [Online]. Available: https://zookeeper.apache.org/ (Acedido em 2020-03-05).

[67] "What is Zookeeper and why is it needed for Apache Kafka? - CloudKarafka, Apache Kafka Message streaming as a Service." [Online]. Available: https://www.cloudkarafka.com/blog/2018-07-04-cloudkarafka_what_is_zookeeper.html (Acedido em 2020-03-07).

[68] "Redis." [Online]. Available: https://redis.io/ (Acedido em 2020-03-05).

[69] "Redis Tutorial - w3resource." [Online]. Available: https://www.w3resource.com/redis/ (Acedido em 2020-06-01).

[70] "Redis – o que é e para que serve? – Desenvolvedor Ninja." [Online]. Available: http://desenvolvedor.ninja/redis-o-que-e-e-para-que-serve/ (Acedido em 2020-06-01).

[71] "What Is a REST API? — SitePoint." [Online]. Available: https://www.sitepoint.com/developers-rest-api/ (Acedido em 2020-05-04).

[72] "What is Git and Why Should You Use It? Free Intro to Git Guide." [Online]. Available: https://www.nobledesktop.com/learn/git/what-is-git (Acedido em 2020-03-30).

[73] "The first single application for the entire DevOps lifecycle - GitLab | GitLab." [Online]. Available: https://about.gitlab.com/ (Acedido em 2020-01-09).

[74] "Jira | Issue & Project Tracking Software | Atlassian." [Online]. Available: https://www.atlassian.com/software/jira (Acedido em 2020-01-11).

[75] "Confluence - Accomplish more together | Atlassian." [Online]. Available: https://www.atlassian.com/software/confluence (Acedido em 2020-03-30).

[76] "Confluence: What is Confluence? - Atlassian Documentation." [Online]. Available: https://confluence.atlassian.com/confeval/confluence-evaluator-resources/confluence-what-is-confluence (Acedido em 2020-03-30).

[77] "Clockify - 100% Free Time Tracking Software." [Online]. Available: https://clockify.me/ (Acedido em 2020-03-30).

[78] "What is API - Application Program Interface? Webopedia Definition." [Online]. Available: https://www.webopedia.com/TERM/A/API.html (Acedido em 2020-02-10).

[79] "What is an API?" [Online]. Available: https://www.redhat.com/en/topics/api/what-are-application-programming-interfaces (Acedido em 2020-02-10).

[80] "The Scrum Framework Poster | Scrum.org." [Online]. Available: https://www.scrum.org/resources/scrum-framework-poster (Acedido em 2020-01-05).

[81] "Gitflow Workflow | Atlassian Git Tutorial." [Online]. Available: https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow (Acedido em 2020-06-25).

[82] "About branches - GitHub Help." [Online]. Available: https://help.github.com/en/github/collaborating-with-issues-and-pull-requests/about-branches (Acedido em 2020-06-28).

[83] "What is MoSCoW Prioritization? | Overview of the MoSCoW Method." [Online]. Available: https://www.productplan.com/glossary/moscow-prioritization/ (Acedido em 2019-12-02).

[84] "SOLID Principles: Explanation and examples - ITNEXT." [Online]. Available: https://itnext.io/solid-principles-explanation-and-examples-715b975dcad4 (Acedido em 2019-11-19).

[85] "What are the best Continuous Integration Tools?" [Online]. Available: https://stackshare.io/continuous-integration (Acedido em 2020-01-18).

[86] "Jenkins." [Online]. Available: https://jenkins.io/

[87] "What Is Continuous Integration and Why Do You Need It? - Code Maze." [Online]. Available: https://code-maze.com/what-is-continuous-integration/ (Acedido em 2020-01-05).

[88] "The Apache Groovy programming language." [Online]. Available: http://groovy-lang.org/ (Acedido em 2020-01-05).

[89] "Comparison of Most Popular Continuous Integration Tools: Jenkins, TeamCity, Bamboo, Travis CI and more | AltexSoft," [Online] https://www.altexsoft.com/blog/engineering/comparison-of-most-popular-continuous-integration-tools-jenkins-teamcity-bamboo-travis-ci-and-more/. (Acedido em 2020-01-06).

[90] "Jenkins X." [Online]. Available: https://jenkins-x.io/ (Acedido em 2019-10-22).

[91] The Linux Foundation, "Production-Grade Container Orchestration - Kubernetes," 2019. [Online]. Available: https://kubernetes.io

[92] "Accelerate | Jenkins X." [Online]. Available: https://jenkins-x.io/about/overview/accelerate/ (Acedido em 2020-01-06).

[93] "Travis CI - Test and Deploy Your Code with Confidence." [Online]. Available: https://travis-ci.org/ (Acedido em 2020-01-06).

[94] "The world's leading software development platform · GitHub." [Online]. Available: https://github.com/ (Acedido em 2020-01-06).

[95] "When to use Jenkins vs. Jenkins X for pipeline automation." [Online]. Available: https://searchitoperations.techtarget.com/tip/When-to-use-Jenkins-vs-Jenkins-X-for-pipeline-automation (Acedido em 2020-01-12).

[96] "What Happens When You Use Virtualization in Software Testing? - DZone Performance." [Online]. Available: https://dzone.com/articles/what-happens-when-you-use-virtualization-in-softwa (Acedido em 2020-07-06).

[97] "What is a virtual machine (VM)?" [Online]. Available: https://www.redhat.com/en/topics/virtualization/what-is-a-virtual-machine (Acedido em 2020-07-06).

[98] "Virtual Machines (VMs) vs Containers: What's The Difference? – BMC Blogs." [Online]. Available: https://www.bmc.com/blogs/containers-vs-virtual-machines/ (Acedido em 2020-07-07).

[99] "Virtualization via Containers." [Online]. Available: https://insights.sei.cmu.edu/sei{_}blog/2017/09/virtualization-via-containers.html (Acedido em 2020-07-07).

[100] "Containers vs Virtual Machines | Atlassian." [Online]. Available: https://www.atlassian.com/continuous-delivery/microservices/containers-vs-vms (Acedido em 2020-07-07).

[101] "A Practical Guide to Choosing between Docker Containers and VMs." [Online]. Available: https://www.weave.works/blog/a-practical-guide-to-choosing-between-docker-containers-and-vms (Acedido em 2020-07-06).

[102] "What's the difference between VMs & Containers? | AKF Partners." [Online]. Available: https://akfpartners.com/growth-blog/vms-vs-containers (Acedido em 2020-07-07).

[103] Q. Zhang, L. Liu, C. Pu, Q. Dou, L. Wu, and W. Zhou, "A Comparative Study of Containers and Virtual Machines in Big Data Environment," Tech. Rep., 2018.

[104] "Load Testing and Website Performance Testing Tools | WebLOAD." [Online]. Available: https://www.radview.com/ (Acedido em 2020-02-27).

[105] "LoadNinja | Performance Testing and Load Testing Tool." [Online]. Available: https://loadninja.com/ (Acedido em 2020-02-27).

[106] "15 BEST Performance Testing Tools (Load Testing Tools) in 2020." [Online]. Available: https://www.softwaretestinghelp.com/performance-testing-tools-load-testing-tools/ (Acedido em 2020-02-22).

[107] "Top 14 Open Source Performance Testing Tools for Load & Stress Testing (2020)." [Online]. Available: https://testguild.com/open-source-performance-testing-tools/ (Acedido em 2020-02-22).

[108] "Apache JMeter - Apache JMeter^{TM}." [Online]. Available: https://jmeter.apache.org/ (Acedido em 2020-02-27).

[109] "Apache JMeter - User's Manual: Component Reference." [Online]. Available: https://jmeter.apache.org/usermanual/ (Acedido em 2020-06-16).

[110] "What is JMeter? Introduction & Uses." [Online]. Available: https://www.guru99.com/introduction-to-jmeter.html (Acedido em 2020-03-02).

[111] "JMeter – Why Non GUI mode is preferable for Load/Performance Testing? How to run JMeter in Non GUI mode? | Help you to test." [Online]. Available: https://helpyoutotest.wordpress.com/2018/03/22/ (Acedido em 2020-03-02).

[112] "How to Load Test: A developer's guide to performance testing | by Rafiullah Hamedy | Medium." [Online]. Available: https://medium.com/@rhamedy/how-to-load-test-a-developers-guide-to-performance-testing-5264faaf4e33 (Acedido em 2020-03-30).

[113] "What is JMeter? Advantages and Limitations of JMeter." [Online]. Available: https://artoftesting.com/what-is-jmeter (Acedido em 2020-03-27).

[114] "JMeter vs Selenium: What is preferred by Testers | BrowserStack." [Online]. Available: https://www.browserstack.com/guide/jmeter-vs-selenium (Acedido em 2020-03-30).

[115] "Taurus." [Online]. Available: https://gettaurus.org/ (Acedido em 2020-03-10).

[116] "BlazeMeter, an Enterprise-grade Continuous Testing Platform." [Online]. Available: https://www.blazemeter.com/ (Acedido em 2020-03-12).

[117] "Taurus: A New Star in the Test Automation Tools Constellation | BlazeMeter." [Online]. Available: https://www.blazemeter.com/blog/taurus-new-star-test-automation-tools-constellation/ (Acedido em 2020-03-27).

[118] "Load testing and functional testing with Taurus - Xray Cloud Documentation - Xpand IT Documentation Portal." [Online]. Available: https://confluence.xpand-it.com/display/XRAYCLOUD/Load+testing+and+functional+testing+with+Taurus (Acedido em 2020-03-10).

[119] "Getting Started with Taurus | Rise." [Online]. Available: https://gettaurus.org/learn/?utm{_}source=BM{&}utm{_}medium=kb{&}utm{_}campaign=creating-a-new-taurus-test (Acedido em 2020-03-12).

[120] "Gatling Open-Source Load Testing – For DevOps and CI/CD." [Online]. Available: https://gatling.io/ (Acedido em 2020-02-27).

[121] "Gatling Reviews: Pricing & Software Features 2020 - Financesonline.com." [Online]. Available: https://reviews.financesonline.com/p/gatling/ (Acedido em 2020-03-10).

[122] "Gatling Performance Testing Pros and Cons | BlazeMeter." [Online]. Available: https://www.blazemeter.com/blog/gatling-performance-testing-pros-and-cons (Acedido em 2020-03-10).

[123] "Gatling Performance Testing Pros and Cons - DZone Performance." [Online]. Available: https://dzone.com/articles/gatling-performance-testing-pros-and-cons (Acedido em 2020-03-10).

[124] "Predator - Distributed open-source performance testing platform for APIs." [Online]. Available: https://www.predator.dev/ (Acedido em 2020-02-27).

[125] "Artillery - a modern load testing toolkit." [Online]. Available: https://artillery.io/ (Acedido em 2020-02-27).

[126] "By: Niv Lipetz, Software Engineer @ ZOOZ - ZOOZ Engineering - Medium." [Online]. Available: https://medium.com/zooz-engineering/by-niv-lipetz-software-engineer-zooz-b5928da0b7a8 (Acedido em 2020-03-05).

[127] "JMeter vs Gatling Tool - Load Testing - OctoPerf." [Online]. Available: https://octoperf.com/blog/2015/06/08/jmeter-vs-gatling/ (Acedido em 2020-03-10).

[128] "When Should I use Gatling vs. JMeter? A Tale of Two Tools - Flood." [Online]. Available: https://www.flood.io/blog/when-should-i-use-gatling-vs-jmeter-a-tale-of-two-tools (Acedido em 2020-03-11).

[129] "SeleniumHQ Browser Automation." [Online]. Available: https://www.selenium.dev/ (Acedido em 2020-05-28).

[130] "What is Selenium Web Driver? Definition of Selenium Web Driver, Selenium Web Driver Meaning - The Economic Times." [Online]. Available: https://economictimes.indiatimes.com/definition/selenium-web-driver (Acedido em 2020-05-29).

[131] "Selenium WebDriver What is it?" [Online]. Available: https://testguild.com/selenium-webdriver/ (Acedido em 2020-06-12).

[132] "What is Selenium WebDriver? Difference with RC." [Online]. Available: https://www.guru99.com/introduction-webdriver-comparison-selenium-rc.html (Acedido em 2020-05-30).

[133] "What is Selenium Web Driver? | How IT Works | Architecture & Advantages." [Online]. Available: https://www.educba.com/what-is-selenium-web-driver/ (Acedido em 2020-06-1).

[134] "Selenium Limitations - javatpoint." [Online]. Available: https://www.javatpoint.com/selenium-limitations (Acedido em 2020-05-29).

[135] "Companies using Selenium and its marketshare." [Online]. Available: https://enlyft.com/tech/products/selenium (Acedido em 2020-05-28).

[136] "Pros and Cons of Selenium Testing Automation Software | AltexSoft." [Online]. Available: https://www.altexsoft.com/blog/engineering/the-good-and-the-bad-of-selenium-test-automation-tool/ (Acedido em 2020-06-15).

[137] "JavaScript End to End Testing Framework | cypress.io." [Online]. Available: https://www.cypress.io/ (Acedido em 2020-06-14).

[138] "Cypress.io: A hands-on overview. Introduction | by shaban.rahman | Medium." [Online]. Available: https://medium.com/@klmlfl/cypress-io-a-hands-on-overview-ad4d498944ee (Acedido em 2020-06-12).

[139] "Cypress.io Vs Selenium Test Automation." [Online]. Available: https://testguild.com/cypress-io-vs-selenium-test-automation/ (Acedido em 2020-05-30).

[140] "Which Language Bindings should be used with Selenium? | BlazeMeter." [Online]. Available: https://www.blazemeter.com/blog/which-language-bindings-should-be-used-with-selenium (Acedido em 2020-06-13).

[141] "Cypress vs Selenium WebDriver: Better, or just different? - Automated Visual Testing | Applitools." [Online]. Available: https://applitools.com/blog/cypress-vs-selenium-webdriver-better-or-just-different/ (Acedido em 2020-06-1).

[142] "Plano de Teste - Um Mapa Essencial para Teste de Software." [Online]. Available: https://www.devmedia.com.br/plano-de-teste-um-mapa-essencial-para-teste-de-software/13824 (Acedido em 2020-05-28).

[143] "GitHub - alexei-led/pumba: Chaos testing, network emulation and stress testing tool for containers." [Online]. Available: https://github.com/alexei-led/pumba (Acedido em 2020-03-05).

[144] "Taking JMeter Tests to the Next Level with Taurus - Intive Blog." [Online]. Available: https://blog.intive-fdv.com/taking-jmeter-tests-to-the-next-level-with-taurus/ (Acedido em 2020-03-27).

[145] "Summary Report – BlazeMeter." [Online]. Available: https://guide.blazemeter.com/hc/en-us/articles/206733909-Summary-Report-Summary-Report (Acedido em 2020-08-26).

[146] "Timestamper | Jenkins plugin." [Online]. Available: https://plugins.jenkins.io/timestamper/ (Acedido em 2020-06-15).

[147] "Running multiple steps." [Online]. Available: https://www.jenkins.io/doc/pipeline/tour/running-multiple-steps/ (Acedido em 2020-06-16).

[148] "Conduktor | Kafka Desktop Client - Beautiful UI." [Online]. Available: https://www.conduktor.io/ (Acedido em 2020-08-15).

Anexos

Esta página foi propositadamente deixada em branco.

Anexo A

Diagrama de *Gantt* planeado para o primeiro semestre

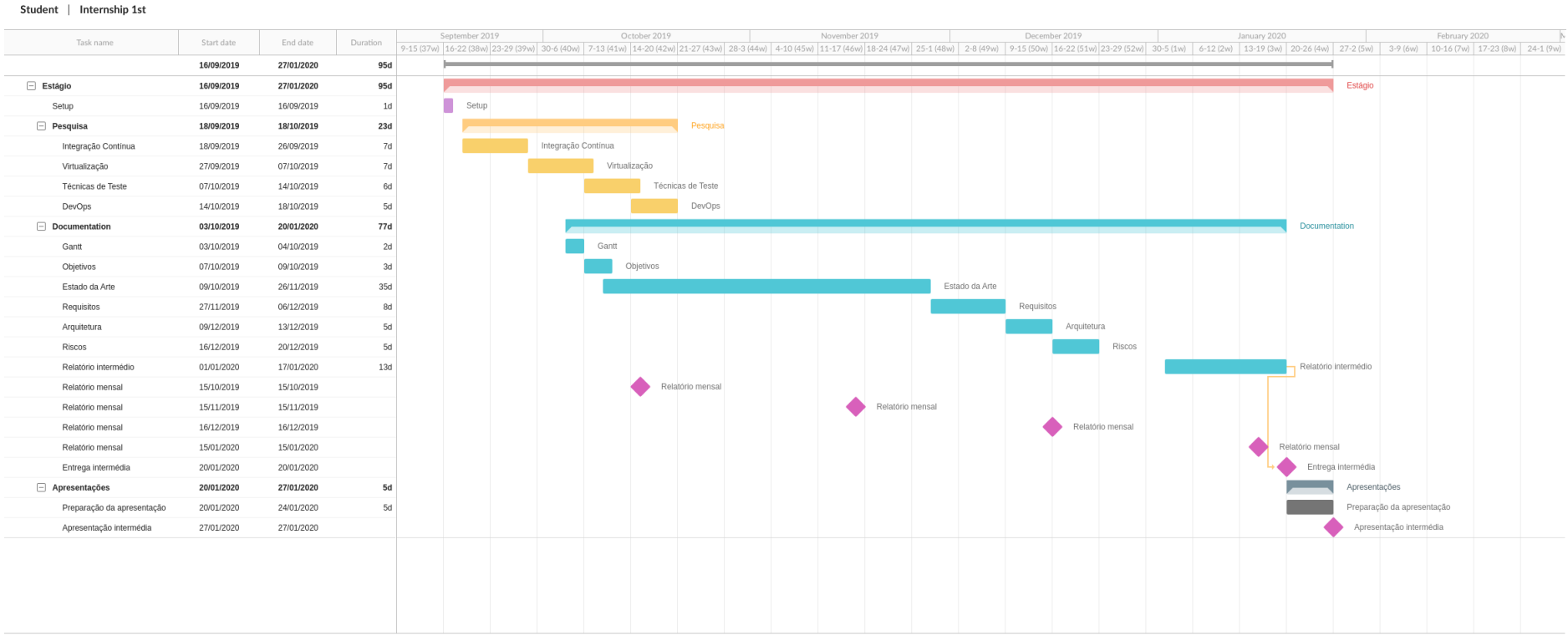

Figura A.1: Diagrama de *Gantt* planeado para o primeiro semestre

Anexo B

Diagrama de *Gantt* executado no primeiro semestre

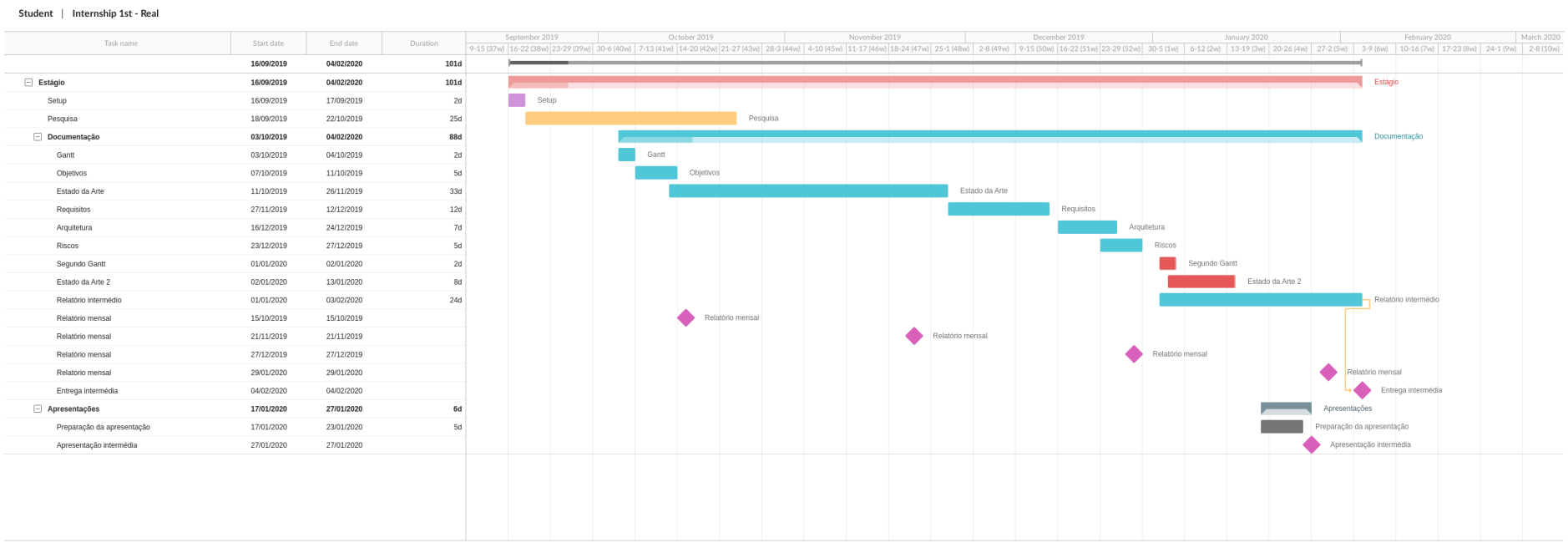

Figura B.1: Diagrama de *Gantt* executado no primeiro semestre

Anexo C

Diagrama de *Gantt* planeado para o segundo semestre

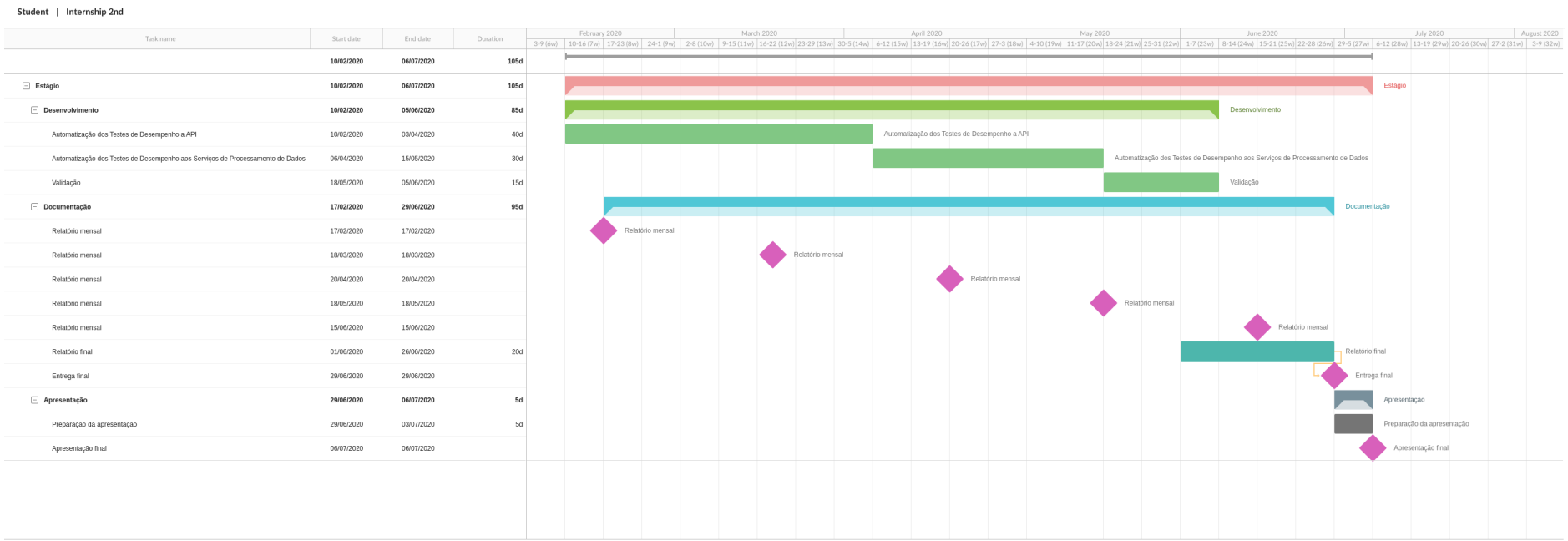

Figura C.1: Diagrama de *Gantt* planeado para o segundo semestre

Anexo D

Diagrama de *Gantt* executado no segundo semestre

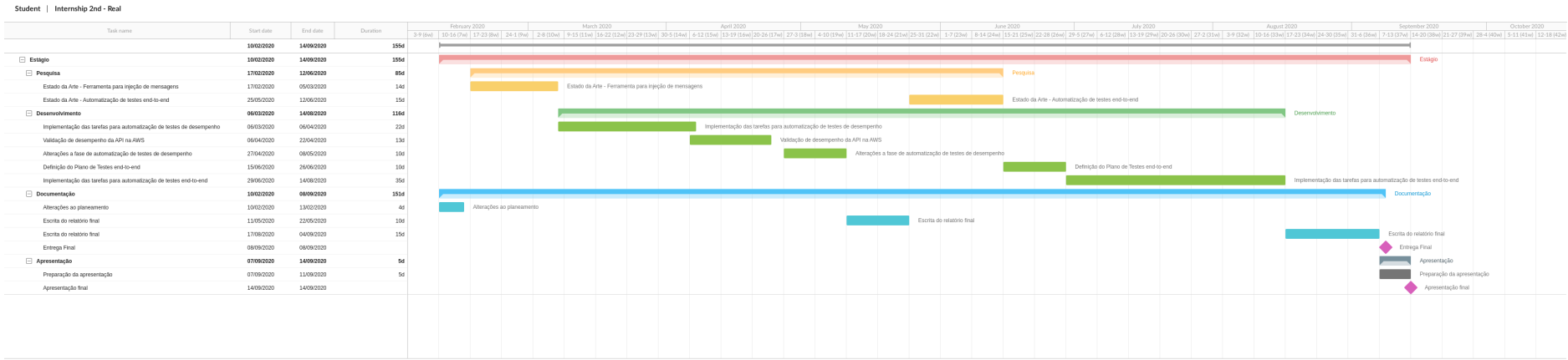

Figura D.1: Diagrama de *Gantt* executado no segundo semestre

Anexo E

Tarefas definidas

ID	Sumário	Prioridade	Estado
	API Perf Tests		
SW-4984	Escolher uma ferramenta para injeção de carga na API	Crítica	Feito
SW-5150	Definição de Plano de Teste	Crítica	Feito
SW-5214	Criar *pipeline* do Jenkins	Crítica	Feito
SW-5215	Executar o JMeter através da *pipeline* do Jenkins	Crítica	Feito
SW-5216	Verificar a ordem das mensagens	Alta	Feito
SW-5217	Validação de latência	Alta	Feito
SW-5218	Adicionar suporte para injeção de mensagens nos restantes formatos (binário e comprimida com *heatshrink*)	Baixa	Backlog
SW-5221	Lançamento do ambiente através da *pipeline* do Jenkins	Alta	Feito
SW-5222	Ferramenta para limitar rede no Kubernetes para simular o ambiente real	Baixa	Feito
SW-5223	Experimentar o Taurus com a configuração do JMeter	Baixa	Feito
SW-5224	Parameterizar o JMeter com o Taurus	Média	Feito
SW-5525	Validação de performance da *Application Programming Interface* (API) na AWS	Média	Feito
SW-6048	Definir asserções para os de testes de desempenho com base na latência e na utilização de CPU e RAM	Média	Cancelada
SW-6053	Recolha das métricas de utilização de CPU e RAM	Crítica	Feito
SW-6055	Criar um *dataset* para injetar na Jam API, baseado nos dados de dispositivo na BD	Baixa	Feito
SW-6070	Parameterizar a *pipeline* do Jenkins para correr os testes de desempenho da API contra qualquer *branch*	Alta	Feito
SW-6180	Execução automática da bateria de testes de desempenho	Baixa	Backlog
	E2E Tests		
SW-6053	Escolher uma ferramenta para automatização de testes *end-to-end*	Crítica	Feito
SW-6177	Definir o plano de testes *end-to-end*	Crítica	Feito
SW-6541	Criar um projeto Cypress no Stratio-play e automatizar o login	Crítica	Feito
SW-6556	Criar um comando de login	Média	Feito
SW-6557	Automatizar o login com credenciais inválidas	Alta	Feito
SW-6558	Implementar o *user flow* 1	Alta	Feito
SW-6564	Automatizar a seleção de todas as ocorrências	Média	Feito
SW-6566	Implementar o *user flow* 2	Alta	Feito
SW-6720	Melhorias ao comando de login	Baixa	Backlog
SW-6897	Criar um trabalho no Jenkins para executar a bateria de testes *end-to-end*	Média	Feito
SW-6923	Implementar o *user flow* 3	Alta	Feito
SW-6926	Execução automática da bateria de testes *end-to-end*	Baixa	Feito

Tabela E.1: *Issues* definidos através do Jira

Anexo F

Arquitetura geral da solução da plataforma de software

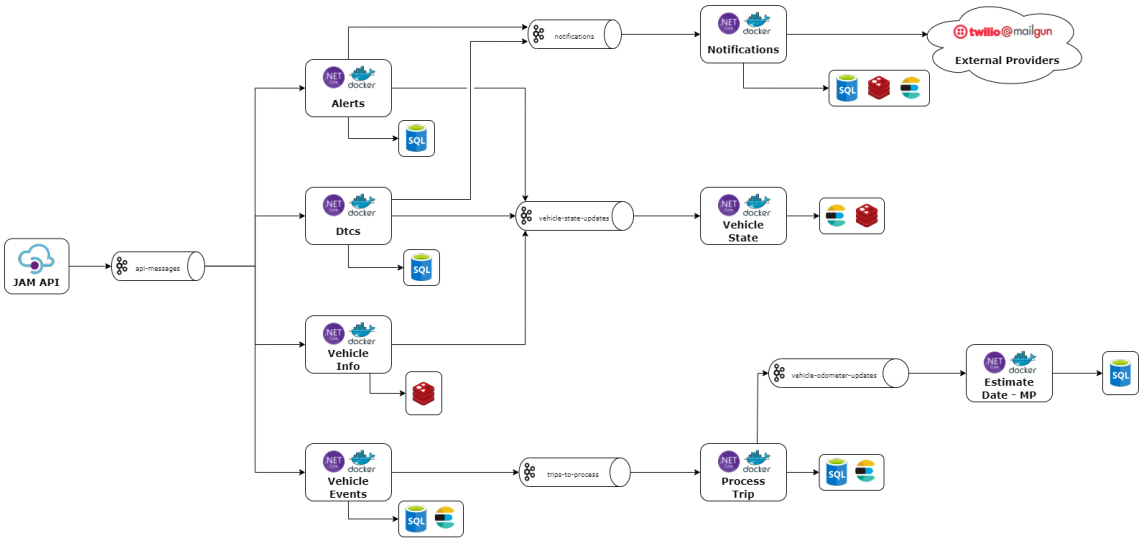

Figura F.1: Arquitetura geral da solução da plataforma de software

Anexo G

Validação ao ambiente na *Amazon Web Services*

Input			Output				
Concurrency	Iterations	Throughput (s)	Avg. Throughput (msg/s)	Máximo Throughput (msg/s)	Avg. Resp. Time (s)	Final Avg. Th	Final Avg. Resp. Time
125	300	5	350.47	387.8	0.347	302.05	0.366
			281.95	353.8	0.378		
			273.72	390.4	0.374		
200	300	5	361.45	547.8	0.396	350.44	0.438
			350.88	545.7	0.429		
			338.98	451.8	0.490		
275	300	5	307.84	476	0.679	342.27	0.633
			358.7	524.6	0.615		
			360.26	492.9	0.604		
125	500	5	263.71	354.3	0.408	268.92	0.408
			261.51	354.9	0.413		
			281.53	365	0.402		
125	300	7	195.31	335	0.541	207.37	0.463
			231.48	384.4	0.401		
			195.31	433.4	0.446		

Tabela G.1: Cenário 1 - Protocolo *HyperText Transfer Protocol* (HTTP)

Input			Output				
Concurrency	Iterations	Throughput (msg/s)	Avg. Throughput (msg/s)	Máximo Throughput (msg/s)	Avg. Resp. Time (s)	Final Avg. Th	Final Avg. Resp. Time
125	300	5	357.14	492.8	0.156	379.30	0.155
			364.08	547.6	0.142		
			416.67	536.5	0.166		
200	300	5	394.74	704.6	0.218	399.44	0.215
			375	724.2	0.221		
			428.57	742	0.206		
275	300	5	416.67	708.6	0.301	-	-
			371.62	634.1	0.413		
			N.R	N.R	N.R		
125	500	5	298.35	385.7	0.357	381.07	0.372
			282.81	381.5	0.374		
			250	376	0.385		
125	300	7	312.5	365.4	0.386	294.90	0.378
			251.68	396.9	0.375		
			320.51	383.9	0.373		

Tabela G.2: Cenário 2 - Protocolo *HyperText Transfer Protocol Secure* (HTTPS)