Faculdade de Ciências e Tecnologia

Departamento de Engenharia Informática

Distributed System Tests Framework

Emanuel Pedro Guerra Correia

Dissertação *Distributed System Tests Framework* no âmbito do Mestrado em Engenharia Informática, especialização em Engenharia de Software orientada pelo Professor Doutor Nuno Antunes e apresentada à
Faculdade de Ciências e Tecnologia / Departamento de Engenharia Informática.

Julho 2019

UNIVERSIDADE D
COIMBRA

This page is intentionally left blank.

Resumo

A criação de uma plataforma distribuída enfrenta grandes desafios, nomeadamente ao nível da qualidade, pelo que é necessário um rápido desenvolvimento de testes que serão posteriormente executados nesses ambientes complexos. O projeto *System Test Framework 2.0* (STF2.0) aparece nesse âmbito conjuntamente com o facto da Feedzai querer ganhar o seu lugar no mundo das *Open-Source* e, simultaneamente, criar o maior entrosamento entre as equipas, criando uma *framework* única, simples e escalável.

A *System Test Framework 1.0* (STF1.0) criada pela equipa do Pulse, usa o Docker para configurar programaticamente o ambiente desejado e, em seguida, permite a qualquer engenheiro escrever testes de sistema usando o *JUnit*. A existência de quatro *frameworks* criadas por cada equipa com objetivos semelhantes, conduz à falta de coesão e ligação entre essas, pois nenhuma plataforma é tida como referência. Outro facto a ser contabilizado é a necessidade de escalar de forma horizontal e automaticamente as componentes criadas, de forma ao esforço ser dividido pelas máquinas.

No âmbito do estágio, o projeto STF2.0 visa desenvolver uma plataforma distribuída com o objetivo de testar os componentes internos e externos da *stack* tecnológica da *Feedzai*, realizando paralelização horizontal. Neste caso optou-se por usar ficheiros de *docker-compose* e de configuração de Kubernetes (K8S) e usar o sistema de gestão de *containers*, o K8S.

Concluindo, com a correta integração dos vários componentes internos e externos surgirão benefícios ao nível da escalabilidade e portabilidade. Todos estes problemas foram acautelados e resolvidos, sendo que a criação da STF2.0 foi um sucesso, dado que consegue de forma automática lançar um ambiente num *cluster* de K8S e executar os testes para os mesmo.

Palavras-Chave

"Testar","Imagens de Docker","K8S","Escalabilidade"

This page is intentionally left blank.

Abstract

The creation of a distributed platform faces major challenges, particularly at the level of quality, so it is necessary to rapidly develop tests that will later be executed in these complex environments. The stf project appears in this context together with the fact that Feedzai wants to gain its place in the Open Source world and at the same time to create the greatest inter-teaming, creating a single, simple and scalable framework.

The STF1.0 created by the Pulse team uses Docker to programmatically configure the desired environment and then allows any engineer to write system tests using JUnit. The existence of four frameworks created by each team with similar objectives, leads to the lack of cohesion and connection among these, since no platform is taken as a reference. Another fact to be counted is the need to scale horizontally and automatically the components created, so that the effort is divided by the machines.

In the scope of the stage, the STF2.0 project aims to develop a distributed platform with the objective of testing the internal and external components of the technology stack of Feedzai, performing horizontal parallelization. In this case we chose to use docker-compose files and K8S configuration and use the container management system, K8S.

In conclusion, with the correct integration of the various internal and external components there will be benefits in terms of scalability and portability. All of these problems were taken care of and solved, and the creation of STF2.0 was a success since it can automatically launch an environment in a cluster of K8S and execute the tests for them.

Keywords

"Testing","Docker Images","K8S","Scalability"

This page is intentionally left blank.

Agradecimentos

Primeiramente gostaria de agradecer à Feedzai, na pessoa do meu orientador Eng. Ricardo Lopes e ao meu orientador Professor Nuno Antunes por todo o aconselhamento e ajuda que precisei ao longo deste ano. Foi uma experiência muito interessante na qual ganhei muitos conhecimentos.

Outro agradecimento vai para a minha família e amigos por todo o apoio que me deram e pelo orgulho que mostraram por ter atingido esta fase académica. Será impossível algum dia retribuir tudo mas vou fazer tudo ao meu alcance para que seja uma viagem sempre em crescente e cheia de sucessos.

This page is intentionally left blank.

Para o meu irmão Miguel Zé.

Contents

This page is intentionally left blank.

Acrónimos

CD Continuous Deployment. x, xv, 5, 7, 10, 11, 32

CI Continuous Integration. x, xv, 5, 7, 10, 11, 32

CIDR Classless Inter-Domain Routing. 23

CPU Unidade central de processamento. 2, 4, 11, 16, 66

DEI Departamento de Engenharia Informática. 2

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

IDE Ambiente de desenvolvimento integrado. 33

K8S Kubernetes. iii, v, x, xv, 2, 3, 13, 14, 16, 17, 19–24, 26, 33, 35, 36, 38–40, 52, 56–58, 60, 61, 67, 71

MEI Mestrado de Engenharia Informática. 1

RAM Memória de acesso aleatório. 2, 4, 11

SO Sistema Operativo. 2, 11, 22, 35

SoA State-of-the-Art. 1

STF1.0 *System Test Framework 1.0*. iii, v, 12

STF2.0 *System Test Framework 2.0*. iii, v, xv, xvi, 1, 3, 7, 10, 12, 13, 31–33, 35–39, 42, 50, 52, 54, 56, 57, 59–64, 66, 67, 97

UC Universidade de Coimbra. 2, 46

VM Virtual Machine. x, xv, 1, 2, 7, 11

This page is intentionally left blank.

Lista de Figuras

This page is intentionally left blank.

Lista de Tabelas

This page is intentionally left blank.

Capítulo 1

Introdução

Este documento está organizado em nove secções e visa apresentar com detalhe o processo de análise, planeamento e desenvolvimento do projeto. Este projeto está inserido no âmbito do estágio curricular do Mestrado de Engenharia Informática (MEI), da Faculdade de Ciências e Tecnologias da Universidade de Coimbra (FCTUC).

A primeira secção explica o contexto do problema, onde se enquadra no curso, os objetivos a atingir e a motivação. A segunda secção engloba todo o *Background*, ou seja toda a informação prévia de interesse para contexto do projeto, desde os vários tipos de testes, a conceitos de desenvolvimento de software e uma breve comparação entre VMs e *containers* .

A terceira secção engloba as alternativas que se pode fazer uso para a STF2.0 e o State-of-the-Art (SoA), onde é avaliada a solução atual e as opções possíveis a utilizar. A quarta secção apresenta os requisitos tanto funcionais como não funcionais. A quinta secção apresenta apresenta os vários níveis da arquitetura. A sexta secção demonstra o planeamento do trabalho, a metodologia de desenvolvimento, que riscos existem e os desvios que aconteceram.

A sétima apresenta o processo de desenvolvimento da solução e os problemas inerentes ao mesmo. A oitava secção apresenta o processo de validação e uma avaliação crítica dos resultados. Por último é apresentada a conclusão na secção 9.

1.1 Contextualização do Problema

Com o escalar do número de testes a executar e um aumento de dados a analisar foi necessário pensar numa solução escalável e portável. Sendo que na Feedzai são executados testes a muitos elementos da *stack* interna, como o produto principal Pulse, que por sua vez pode ser integrado com diversos componentes externos (e.g.Bases de Dados de *Postgrés* ou *Cassandra*, etc.).

De forma a criar uma solução, que agradasse a todas as equipas, foi necessário criar um documento com todos os requisitos necessários a usar por cada equipa, e assim nasceu a ideia para a STF2.0.

Esta solução tem então de ser útil para as várias equipas, conseguir aceitar qualquer tipo de elemento da *stack* interna ou qualquer outro, desde que seja possível colocar num *container*, e que fosse automática, simples e escalável.

1.1.1 Contexto Organizacional

O projeto foi realizado na empresa *Feedzai* enquadrado no mestrado de Engenharia Informática do Departamento de Engenharia Informática (DEI) da Universidade de Coimbra (UC).

A Feedzai usa *real-time event processing* com técnicas de *machine learning* e *Big Data* para identificar transações fraudulentas de pagamento e minimizar riscos no setor financeiro.

O Pulse, principal produto da empresa, permite que as empresas analisem informações para manter os dados e transações de seus clientes seguros. O software da Feedzai deteta fraudes por meio de análises históricas e comportamentais profundas dos dados da organização. De acordo com a Forbes, em fevereiro de 2016, a Feedzai processou 1 bilião de dólares por dia no volume total de pagamentos. [5]

1.1.2 Contexto Tecnológico

Quando os *containers* foram introduzidos, as VM eram a opção de última geração para otimizar os recursos físicos de um *data center*. Esta solução foi eficaz mas tinha algumas falhas: as VM utilizavam muitos recursos porque exigiam um Sistema Operativo (SO) completo.

O objetivo dos *containers* é maximizar a utilização de recursos para alcançar um desempenho *bare-metal* semelhante com as vantagens das VM. Um ambiente *bare-metal* é um sistema de computador ou rede no qual as VM são instaladas diretamente no hardware, e não no SO. De forma a alcançar este objetivo, é partilhado um *kernel* com todas as aplicações em que a escolha dos recursos é ajustada a cada situação.

Cada *container* tem acesso exclusivo a recursos como Unidade central de processamento (CPU), Memória de acesso aleatório (RAM), disco e rede, e pode ser gerido por um *user*.

Sistemas de gestão de *containers*, como o K8S, oferecem uma solução que garante alta disponibilidade, recuperação de falhas e escalabilidade. Os sistemas são responsáveis por manipular um ou mais *clusters* de máquinas e detetar a disponibilidade de cada imagem em execução. Se uma máquina tiver um erro ou falha, a ferramenta deve ser capaz de reagir de forma consistente.

Este tipo de gestores lidam com várias máquinas que podem estar no mesmo *data center* ou em locais físicos diferentes de variados fornecedores, criando uma abstração em relação a como os recursos de computação da empresa são distribuídos.

Uma das grandes vantagens deste tipo de ferramentas é ter a capacidade de funcionar como um sistema operacional *multicloud* onde as empresas somente têm de designar quais recursos é que têm de estar em execução.

Docker

O *Docker* é uma tecnologia baseada em *containers*. No nível abstrato, um *container* é apenas um conjunto de processos isolados do restante do sistema, a partir de uma imagem distinta que fornece todos os recursos necessários para suportar os processos. No *Docker*, os *containers* em execução compartilham o *kernel* do SO do host.

O *Docker* é um projeto de código aberto baseado em *containers* de Linux, onde usa os recursos do *Kernel* do Linux, como *namespaces* e grupos de controlo.

A ascensão do K8S

O K8S é uma ferramenta *Open Source* de código aberto que pode dimensionar, distribuir e gerir automaticamente falhas em *containers*. Originalmente criado pelo Google e doado para a *Cloud Native Computing Foundation*, o K8S é amplamente usado em ambientes de produção para manipular *containers* de Docker e outros.

Como um produto de código aberto, está disponível em várias plataformas e sistemas. O Google Cloud, o Microsoft Azure e o Amazon AWS oferecem suporte oficial ao K8S.

A popularidade do K8S aumentou constantemente, com mais de quatro grandes lançamentos em 2017. O K8S também foi o projeto mais discutido no GitHub em 2017, e foi o projeto com o segundo maior número de *reviews*.

Linguagem de Programação

Para desenvolvimento da STF2.0 optou-se por usar a linguagem de programação Java, sendo a grande razão desta decisão o fato de ser a linguagem mais familiar ao autor e porque existem bibliotecas que permitem usar a API do K8S.

1.2 Identificação dos Problemas

Tendo em conta o contexto apresentado, foram identificados 5 problemas, que que serão abordados na tese:

- **Problema 1** - Há uma preocupação em que a *framework* seja *open-source* e simultaneamente que esteja integrada nos processos da Feedzai.

  Neste momento, os processos usam somente componentes internos. Simultaneamente, é a melhor forma de garantir que a plataforma é independente de qualquer outro projeto.

  Os sistemas internos a ter em atenção serão todos os processos enunciados no capítulo 3.2.1.

- **Problema 2** - Diferentes *frameworks* dificultam o *onboarding* entre diferentes equipas

  A questão passa pelo facto de puderem vir a existir dificuldades devido a uma possível repetição ou inconsistência, por os vários departamentos terem diferentes objetivos e formas de realizar as mesmas ações. Igualmente, torna as equipas mais coesas tendo em conta que todas trabalham na mesma *framework*.

- **Problema 3** - Documentação dispersa e incompleta

  Uma documentação detalhada é imperativa para que a compreensão seja simples e o mais célere possível. O ficheiro deve conter toda a configuração necessária e explicar todas as funcionalidades da aplicação. Será sempre uma possibilidade a criação de um fórum para ser usado por qualquer *user*.

- **Problema 4** - Não existem formas para realizar *debug* nos testes falhados ou com *bugs*

  Em qualquer projeto existem falhas de escrita no código e é necessário arranjar formas de facilitar o *debug*. Neste caso uma das soluções pode passar por indicar e abrir os portos das imagens que falharam.

- **Problema 5** - Necessidade de dividir as cargas por várias máquinas

  A escalabilidade horizontal é feita com o particionamento dos dados, ou seja, cada nó tem somente frações da informação. Com o dimensionamento horizontal, muitas vezes é mais simples dimensionar dinamicamente adicionando mais máquinas à *pool* existente.

  Desta forma, as máquinas são usadas uniformemente não criando disparidades no uso de RAM ou CPU, técnica que é usada por exemplo, no *Apache Casandra*.

1.3 Motivação

Toda a plataforma da *Feedzai* é um sistema distribuído, que contém diferentes componentes, tanto internos(*Pulse, Case Manager, Tokenizer, Genome*, ...) e externos (*Zookeeper, RabbitMQ, Cassandra, Spark, Hadoop / YARN, ...*).

Um dos maiores desafios para garantir um processo de qualidade durante o desenvolvimento de uma plataforma distribuída é apoiar o rápido desenvolvimento de testes que possam ser executados novamente nesses ambientes complexos. A primeira versão do *System Test Framework* criada pela equipa do *Pulse* usa o *Docker* para configurar programaticamente o ambiente desejado e, em seguida, permite escrever testes de sistema usando o *JUnit*, de forma simples, por qualquer engenheiro que esteja a desenvolver a plataforma.

No entanto, foram identificadas algumas lacunas na mesma, sendo que esta dependente do repositório do *Pulse*, não sendo possível usar fora da Feedzai e ao mesmo ao tempo não é possível escalar os componentes dos testes. Devido a isso, novas equipas que foram criadas juntamente com o crescimento da *Feedzai* criaram outras estruturas com objetivos semelhantes, e assim acabamos com 4 *frameworks* distintas:

- *Pulse (System Test Framework)*
- *Case Manager(Test Containers)*
- *Merchants and Banking (Cosy Test Framework)*
- *Insights (Test Containers)*

Estas *frameworks* são mais aprofundadas na **secção 3.2.1**.

Neste momento, concordamos que uma estrutura de testes única deve ser usada por todas as equipas do *Feedzai*. Esta decisão baseia-se em que:

- Estruturas diferentes levam a um grande número de adaptações de cada equipa;
- Estruturas diferentes dificultam a integração de pessoas em diferentes equipas;

- É melhor ter uma estrutura que possa ser compartilhada e mantida por todos, em vez de ter uma estrutura que seja mantida apenas por uma equipa. [23]

A nível pessoal, é bastante estimulante deixar uma marca numa empresa, do nível da *Feedzai*, ou mesmo na comunidade tecnológica global, visto que vai ser uma plataforma *Open Source*. Ao mesmo tempo, é um desafio bastante encorajador, tendo em conta que estou a ter oportunidade de trabalhar com novas tecnologias, como *docker* ou *kubernetes*.

Da mesma forma, ter a oportunidade de trabalhar com uma equipa experiente e com uma estrutura muito maior do que se está habituada no curso é desafiante, com várias máquinas e componentes externos. Finalmente, a área de testes sempre foi interessante para mim e na qual quero ganhar experiência.

1.4 Objetivos

Nesta tese são propostos um certo conjunto de objetivos, que se interligam com os requisitos posteriormente especificados.

- **Projetar a *framework* como um projeto *Open Source*** - Sendo que a Feedzai se quer afirmar no mundo do *Open-Source* e quer ter uma *framework* de testes completamente independente dos componentes internos.
- **Permitir a integração de componentes internos e externos que pertencem à *stack* da Feedzai** - Apesar da independência da ferramenta esta tem de conseguir incorporar todos os componentes já existentes na empresa de forma a que se consiga correr os testes internos.
- **Permitir conetar/integrar módulos** - A *framework* deve ser capaz de integrar módulos novos importando ou habilitando. Estes módulos podiam ser por exemplo, bibliotecas de teste das diversas equipas da empresa.
- **Permitir de integrar a solução com CI e CD**, como Jenkins ou GitLab CI;
- **Suportar a paralelização horizontal de testes 'out-of-the-box'**, usando soluções como *Kubernetes* ou *Docker Swarm*;

This page is intentionally left blank.

Capítulo 2

Background

Este capítulo está organizado em 4 secções.

A primeira inclui o tipo de testes que têm relação com a *framework*, isto é, os diferentes níveis e tipos de testes que permite criar. A segunda secção apresenta o que é Continuous Integration (CI) e a terceira o que é Continuous Deployment (CD). Para finalizar a última secção faz uma comparação entre Virtual Machine (VM)s e *containers*.

2.1 Tipos de teste

Nesta secção são apresentados os tipos de teste que têm interesse para o desenvolvimento da *System Test Framework 2.0* (STF2.0).

2.1.1 *Functional testing*

O *Functional testing* é um método em que todos os componentes são testados em conta os requisitos funcionais. As funções recebem um *input* e o *output* é analisado.

Esta experiência garante que os requisitos sejam satisfeitos adequadamente pelo programa, sem ter em conta o processamento mas sim os resultados obtidos. Durante o teste funcional, é utilizada a técnica *Black Box Testing*, na qual a lógica interna do sistema que está a ser testado não é conhecida pelo *tester*. Outros tipos também compõem o teste funcional como apresenta a figura 2.2

Neste tipo de testes são tidos em conta os atributos qualitativos e é calculado o tempo de execução dos processos. Nestes tipos de testes, podemos verificar quais são os componentes que demoram mais tempo a ser testados.

Metodologia Este tipo de teste segue um conjunto de passos apresentados na figura seguinte.

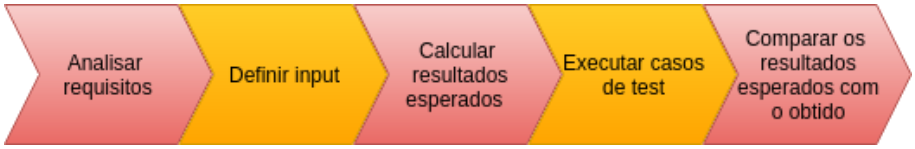

Figure 2.1: Metodologia do *Functional Testing*

Figure 2.2: Diagrama do *Functional Testing* (de [19])

2.1.1.1 *System Testing*

O teste do sistema não é um processo de testar as funções do sistema ou programa completo, porque isso seria redundante com o processo de teste de função. O mesmo tem uma finalidade específica: comparar o sistema ou programa com seus objetivos originais. Diante desse propósito, duas implicações são cruciais:

- Não está limitado aos sistemas. Se o produto é um programa, o teste do sistema é o processo de tentar demonstrar como o programa, como um todo, não atinge seus objetivos;
- é impossível se não houver um conjunto de objetivos mensuráveis e escritos para o produto;

Como representado na Figura 2.8, os passos deste tipo de testes são:

1 - Projetar o teste do sistema, analisando os objetivos;

2 - Formular casos de teste, tendo em conta a documentação do *user*; [24]

2.1.1.2 *Regression Testing*

Regression Testing é definido como um tipo de teste de software para confirmar que um programa não foi afetado por alterações no código. Este tipo de teste é uma escolha completa ou parcial de casos de testes executados anteriormente que são executados de novo. [16]

Existem 3 tipos de formas de realizar este teste.

- *Retest All* - todos os testes no conjunto de testes devem ser executados novamente tornando-se caro, pois requer muito tempo e recursos.

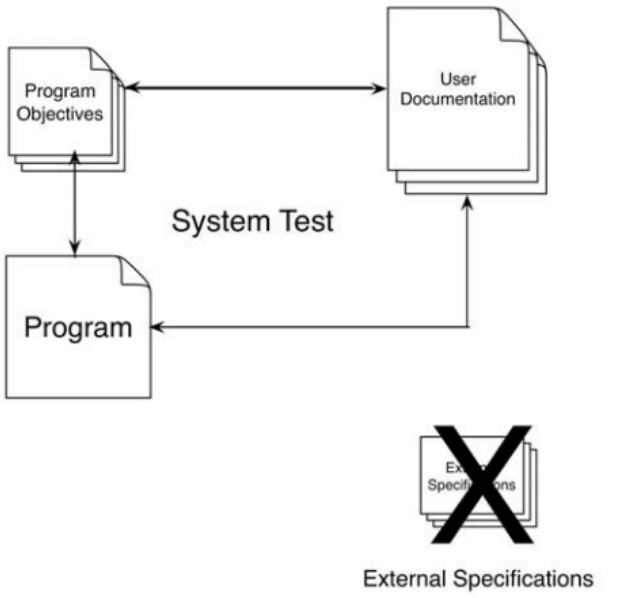

Figure 2.3: Diagrama do *System Test* (de [24])

- *Regression Test Selection* - Em vez de executar novamente todo o conjunto de testes, é melhor selecionar parte do conjunto de testes a ser executado. Estes casos podem ser divididos em duas categorias:
  - Casos de teste reutilizáveis, que são usados em ciclos de regressão subsequentes;
  - Casos de teste obsoletos que não podem ser não podem ser usados em ciclos sucessivos
- *Prioritization of Test Cases* - neste caso ordena-se os testes por prioridades, dependendo do impacto nos negócios, das funcionalidades críticas. A seleção de casos de teste com base na prioridade reduzirá bastante o conjunto de testes.

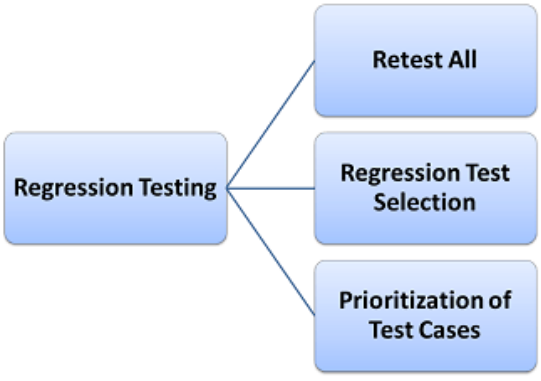

Figure 2.4: Diagrama do *Regression Testing* (de [16])

2.1.2 *Black-Box Testing*

Black-box testing ou *input/output-driven testing*, é um tipo de teste em que não existe preocupação sobre o comportamento interno ou estrutura do programa. Em vez disso,

disso concentra-se em encontrar circunstâncias em que o programa não se comporta como devia, segundo certas especificações.

Ou seja, dado um conjunto de *inputs* é esperado que produza um certo conjunto de *outputs* pré-determinados. [24]

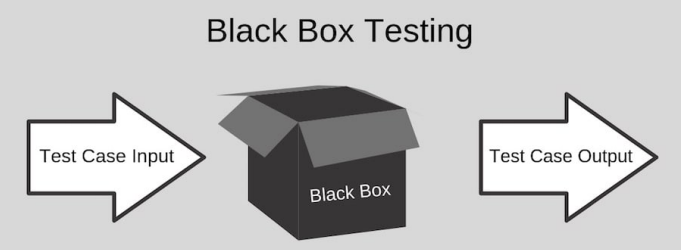

Figure 2.5: Diagrama do *Black-Box Testing* (de [1])

2.1.3 Ilações finais

Para contextualizar estes tipos de testes no documento, vou indicar como estes testes vão ser usados no estágio.

Já a STF2.0 vai ser utilizada para executar testes de *functional, regression, system e black-box testing.*

2.2 CI

CI é uma prática de desenvolvimento de software em que os membros de uma equipa integram o seu trabalho com frequência, pelo menos diariamente ou até pode levar a várias integrações por dia. Cada integração é verificada criando uma *build* e é revista por um método automático para detetar os índices de integração o mais rápido possível. Este método reduz erros de integração de código perto do lançamento. [21]

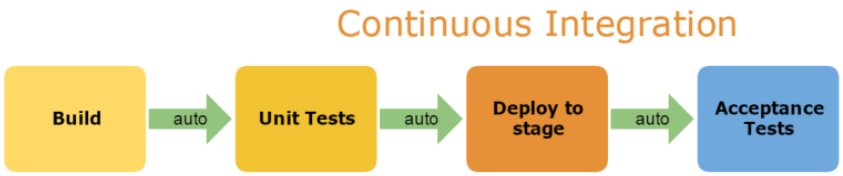

Figure 2.6: Gráfico de CI (de [3])

2.3 CD

CD é um método de entregar software em que qualquer *commit* que execute a fase de teste e passe com sucesso é automaticamente lançado no ambiente de produção. Esta estratégia é realizada em ciclos curtos e elimina o risco de erro humano, sendo que só é feito o *deployment* do produto caso não exista nenhum erro. [26]

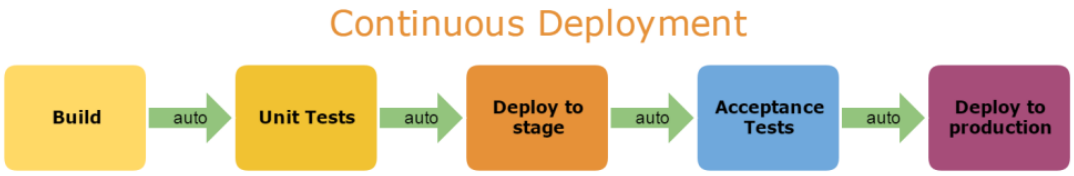

Figure 2.7: Gráfico de CD(de[3])

Observando **as figuras 2.6 e 2.7** denota-se que todos passos são automáticos como explicado nesta secção e na anterior. Sendo que o CI não lança em produção ao contrário do CD, sendo que com as figuras representam de forma mais intuítiva todo o processo.

2.4 VM's vs *Containers*

As máquinas virtuais fornecem a capacidade de emular um Sistema Operativo (SO) e, portanto, um computador separado, diretamente no host. Uma máquina virtual é composta *userspace* mais o espaço de *kernel* de um SO. As VMs são executadas em cima de uma máquina física *hipervisor* que é um programa que permite que vários SOs partilhem um único host de hardware.

Já os *containers* são uma forma de 'empacotar' software, principalmente todo o código, bibliotecas e dependências da aplicação. Eles fornecem um ambiente virtual leve que agrupa e isola um conjunto de processos e recursos, como Memória de acesso aleatório (RAM), Unidade central de processamento (CPU), disco do host, etc.. O isolamento garante que qualquer processo dentro do *container* não possa ver nenhum processo ou recurso fora do mesmo.

Os *containers* também fornecem a maioria dos recursos fornecidos por máquinas virtuais, como endereços IP, montagem em volume, gestão de recursos (CPU, RAM, disco), SSH (exec), imagens do SO.

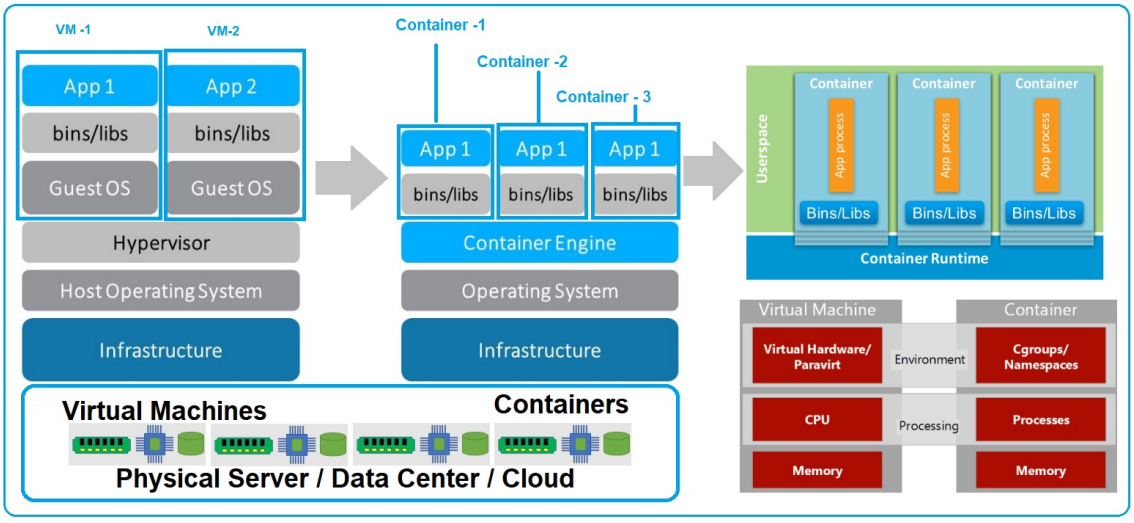

Figure 2.8: Diferenças entre as VM e os *containers* (de [4])

Capítulo 3

State-of-the-Art e Alternativas de Desenvolvimento

Este capítulo está organizado em duas secções.

A primeira secção as versões criadas fora do espectro da empresa Feedzai e na seguinte é apresentadas todas as soluções internas já implementadas. Para cada uma das sub-secções é explicado o contexto e as funcionalidades de cada uma das hipóteses.

A segunda secção apresenta as alternativas em que a STF2.0 pode ser desenvolvida.

3.1 State-of-the-Art

Neste secção, vão ser apresentadas as hipóteses internas e externas que consegue, totalmente ou em parte, realizar as mesmas operações que a STF2.0.

3.1.1 Soluções internas

Nesta secção vão ser apresentadas as atuais *frameworks* usadas pela empresa.

3.1.1.1 *System Test Framework 1.0* (STF1.0)(Pulse)

Esta é a versão inicial do sistema testes da componente principal da empresa, Pulse. A STF1.0 é uma biblioteca que permite construir um ambiente de teste com componentes isolados do *Feedzai Pulse* em uma única máquina, sem a necessidade de máquinas físicas ou virtuais separadas. Para obter o isolamento de componentes na mesma máquina, a STF1.0 tem toda a gestão do levantamento dos containers e a gestão do seu *lifecycle* é feito pela framework através da *Docker*API.

A principal motivação desta *framework* era permitir fazer testes de *fault-tolerance*, pelo que a mesma suporta uma configuração extensiva de como os *containers* são configurados ao nível do teste. [18]

3.1.1.2 Cosy-Test Framework

É uma estrutura simples que permite a integração com várias estruturas de teste. Com a *Cosy-Test* é possível usar somente ficheiros de *docker compose* e definir variáveis de ambiente para iniciar ambientes de *Docker*, executar testes e reduzir os ambientes sem dificuldades e restrições.

Principais Características

- Levanta ambientes antes dos testes e para-os depois;
- inicia os testes logo a seguir à validação dos *health checks*;
- Pode ser configurado para despejar *logs* de *containers* para fins de depuração;
- Os *containers* podem ser mantidos em funcionamento com falha, sucesso ou ambos;
- As portas expostas pelos *containers* são mapeadas para a máquina *host* e disponibilizadas para os testes.

Em que situações usar?

Esta ferramenta é necessária quando se tem um sistema complexo que requer ter muitos componentes a funcionar e simular um cenário real.

3.1.2 Soluções externas

Nesta secção, são apresentadas as soluções externas que tenham similaridades com o que se quer criar com a STF2.0.

3.1.2.1 K8s-Client da Fabric8io

A fabric8io criou uma versão alternativa do cliente nativo de Kubernetes (K8S). Esta solução apresenta funcionalidades muito interessantes para a proposta da tese, sendo estas:

- criar um cliente;
- configuração do cliente, para usar tanto em modo local ou usando um *cluster*;
- carregar recursos de fontes externas;
- todas as funções da APi de K8S;
- tem funções de monitorização;
- já contém testes unitários específicos de K8S;

Esta solução, serviu de base para ideias a usar na STF2.0, mas não foi usada porque não pode utilizada de forma automatizada e acaba por ser uma versão não oficial da API de K8S, podendo até criar dúvidas ao utilizadores. Especificamente, o sistema de criação de serviços e *deployments* foi baseado nesta versão. [9]

3.1.2.2 *Testcontainers*

O *Testcontainers* é uma livraria de Java que suporta testes JUnit, fornecendo instâncias leves e descartáveis de bases de dados comuns ou de qualquer outra instância possível de ser executada num *container* de Docker. Esta solução também é usada internamente.

Esta ferramenta facilita as seguintes ações:

- **Testes de integração para o *layer* dos dados**

  Usa instâncias de bases de dados de MySQL, PostgreSQL ou Oracle para testar o código de acesso da camada para que haja compatibilidade completa, mas sem que exista uma obrigatoriedade de configuração. Qualquer tipo de base de dados pode ser colocada em *containers*.

- **Testes de integração em aplicações**

  Para correr as aplicações num modo de teste *short-lived* com dependências, como bases de dados ou *message queues*.

- **Teste de Interface**

  Usando *containers* de *browsers*, compatíveis com Selenium, de forma a realizar testes automatizados na interface. Cada sessão, contém uma instância de um navegador web, e é gravada em vídeo ou somente se falhar.

Sendo que esta ferramenta só está adaptada para Docker não pode ser usada como solução efetiva. Inicialmente serviu para o estagiário ter noção de todo o ambiente e de como conseguiria instanciar *containers* e executar testes contra os mesmos.

Na realidade, a nossa ferramenta é um *testcontainers* adaptado a K8S mas em que tudo é feito de forma automatizada, desde a criação do ambiente, ao lançamento das instâncias *containers* e à execução de testes, sendo que é importante ter esta ferramenta por base, sendo que é bastante usada no meio de produção. Outras situações que rejeitam esta ferramenta é o facto de não conseguir escalar os testes no *cluster* e não existir suporte para K8S. [12]

3.2 Alternativas de Desenvolvimento

Este capítulo visa apresenta as várias alternativas de ferramentas a usar para gerir os *containers* e fazer uma comparação entre os mesmos e obter uma decisão ponderada e justificada.

3.2.1 Visão geral do *Kontena*

O ***Kontena***, à semelhança do *Docker Swarm*, foi criado com o intuito de combater o tempo entre a iniciação e a conclusão no processo de produção ou até mesmo a *steep learning curve* necessária para projetos de *Kubernetes*. O conceito *steep learning curve* advém da capacidade de aprendizagem vs o tempo despendido. Este é um sistema para gestão e monitorização de containers em diferentes hosts em qualquer infraestrutura de *cloud*.

Os destaques deste programa são o fato de ser escrito em *Ruby* e do requisito do servidor de autenticação ser separado. Além disso tem bons mecanismos de auditoria e de controlo

de acessos. O *Kontena* pode ser instalado em qualquer infraestrutura de*cloud* que execute Linux ou em qualquer tipo de máquina virtual.

Como o *Kubernetes*, o Kontena trabalha em um nível de abstração maior do que os *containers* onde os componentes de construção são chamados de serviços. Da mesma forma, também usa redes avançadas de sobreposição como *Weave* e *OpenVPN* para permitir comunicações entre serviços. [10]

3.2.2 Visão geral do *Apache Marathon/Mesos*

Mesos é um *kernel* distribuído que visa fornecer alocação dinâmica de recursos nos *datacenters* e é um gestor de *clusters*. Um *cluster* fornece isolamento eficiente de recursos e partilha de aplicações ou ou estruturas distribuídas.

O *Mesos* vem com várias *frameworks*, *stacks* de aplicações que usam as suas capacidades de partilha de recursos. Cada estrutura consiste em um organizador e um executor. Ao mesmo tempo este permite facilitar a implementação e gestão de aplicações em *clusters* de grande escala com mais eficiência.

O *Marathon* é uma *framework* que pode lançar aplicações e outras *frameworks*. Ao mesmo tempo, pode servir como uma plataforma de organização de *containers* que pode fornecer escalabilidade e auto-recuperação para cargas de trabalho nos mesmos.

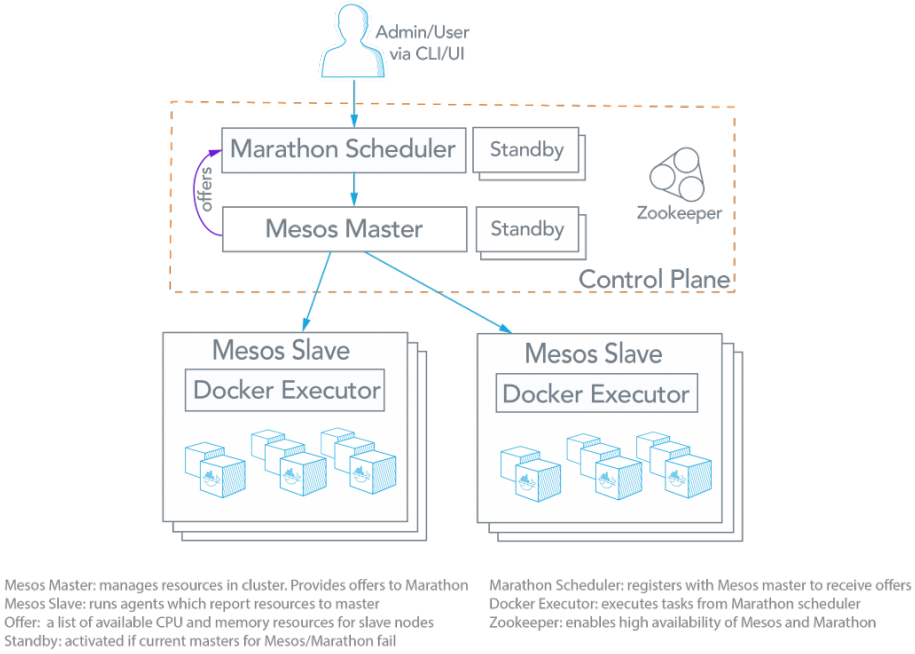

Figure 3.1: Arquitetura do Mesos (de [14])

Arquitetura A arquitetura como se pode verificar na figura 3.1 é composta por variados elementos. Como:

- ***Mesos Master*** - O *Master* é responsável por enviar tarefas para os *Slaves*. Ele mantém uma lista de recursos disponíveis e faz ligações deles para *frameworks*, e.g. *Hadoop*. O *Master* decide quantos recursos oferecer com base na estratégia de alocação e normalmente, haverá instâncias de reserva para assumir o controlo em caso

de falha. Ao mesmo tempo usa *Spark* para processamento de dados em larga escala e *Cassandra* para bancos de dados *NoSQL*;

- ***Mesos Slave*** - Responsável pela execução de tarefas. Todos os nós deste tipo enviam uma lista de seus recursos disponíveis para o *Master*;

- ***ZooKeeper*** - Usado para procurar endereço do *Master* atual. Várias instâncias do *ZooKeeper* são executadas para garantir a disponibilidade e reação a falhas;

- ***Framework*** - Regista com o *Mesos Master*, de modo a que este possa receber tarefas para executar nos nós *Slave*.

- ***Marathon Scheduler*** - Este componente recebe notificação do CPU disponível do *Mesos Master* e memória dos seus *Slaves*.

- ***Docker Executor*** - Esse componente recebe tarefas do *Marathon Scheduler* e lança os *containers* nos *Slave*.

3.2.3 Visão geral do *Kubernetes*

K8S é baseado em anos de experiência do Google na execução de cargas de trabalho em grande escala e em modo de produção. Por definição, é um sistema *open-source* para automatizar a implantação, o dimensionamento e a gestão de aplicações em *containers*. [15] Durante esta secção vão ser usadas seis terminologias específicas de K8S, aqui explicadas.

- **Pod** - É uma coleção de um ou mais *containers*. Atua como uma unidade central de gestão de K8S. Ao mesmo tempo, define o limite lógico para *containers* que partilham o mesmo contexto e recursos. [11]

- **Service** - É uma unidade que atua como um *load balancer*. Um serviço agrupa coleções lógicas de conjuntos que executam a mesma função para dar uma impressão de entidade única. [11]

- **Volume** - é essencialmente uma diretoria que pode ser acedida por todos os *containers* de um *pod*. Estes são preservados quando um *container* é reiniciado. [11]

- **Labels** - São *tags* arbitrárias que podem ser colocadas nas unidades de trabalho para defini-las como parte de um grupo. Eles podem ser selecionados para fins de gestão e segmentação por ação. [11]

- **Namespaces** - fornecem uma *scope* para os objetos do K8S. É como um *workspace* que partilham recursos como rede ou disco. [11] 1

- **Replication Controller** - Uma versão mais complexa do *pod* é conhecida como *Replicated Pod*. Estes são geridos por um *Replication Controller*, que garante que um número específico de réplicas de *pod* sejam executados a qualquer momento. [11]

Arquitetura Na figura 3.2 é apresentada a arquitetura desta ferramenta. Como é normal neste tipo de sistemas distribuídos existe um nó *master* e um ou vários nós workers.

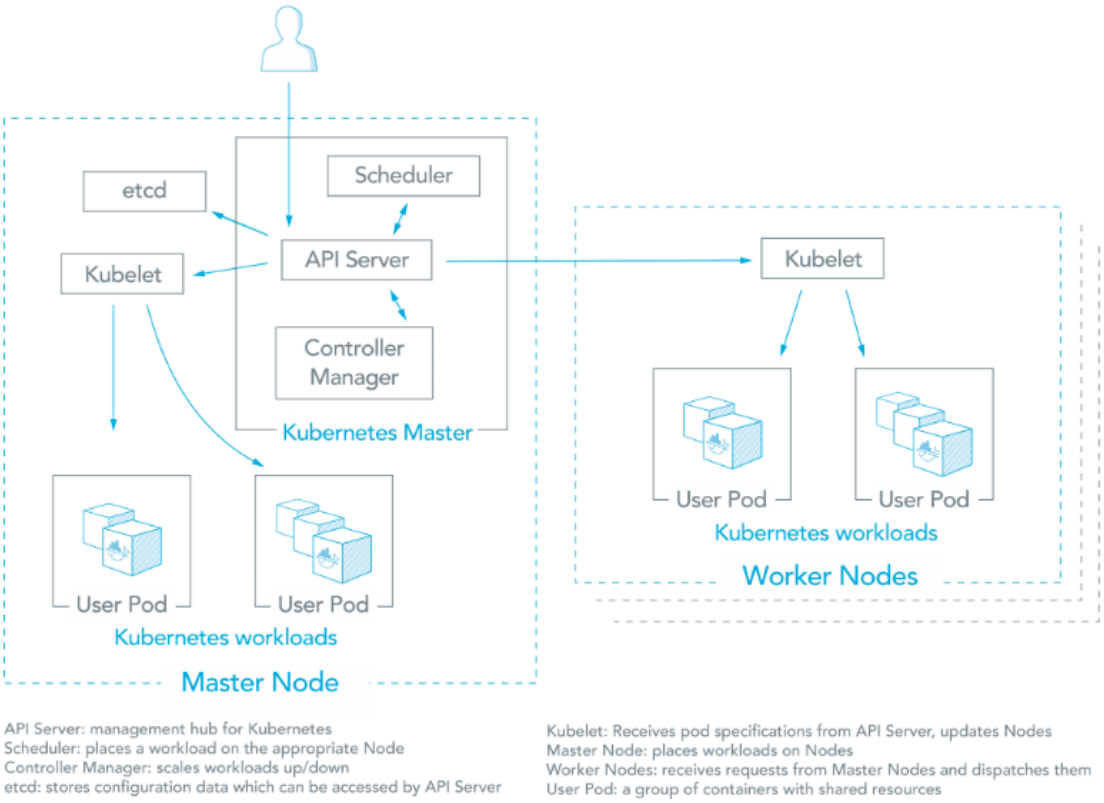

Figure 3.2: Arquitetura do K8S (de [14])

O ***Master Node*** é responsável pela gestão do *cluster*, sendo este o principal ponto de entrada de tarefas. Este mesmo nó, é o que gere os *workers* e onde os serviços estão a ser executados.

É composto por vários componentes:

- **Servidor da API** - É o principal ponto de gestão do *cluster*, pois permite que o *user* configure as cargas de trabalho e unidades organizacionais do *Kubernetes*. Ao mesmo tempo, é o ponto de entrada para todos os comandos REST usados para controlar o *cluster*.

- **etcd** - É um armazenamento, do tipo valor-chave e distribuído entre vários nós, que foi desenvolvido para ser usado principalmente para configuração partilhada de serviços. O K8S usa o etcd para armazenar dados de configuração que podem ser usados por cada um dos nós no *cluster*.

- ***Scheduler*** - Este componente configura os *pods* e serviços nos nós. Além disso, também é responsável pelo controlo da utilização de recursos em cada *host* para garantir que as cargas de trabalho não sejam definidas além dos recursos disponíveis.

- ***Controller-mananger*** - É um serviço geral que é responsável por controladores que regulam o estado do *cluster* e executam tarefas de rotina. O exemplo desse tipo de controlador é o *Replication Controller* e da forma como garante que o número de réplicas definidas para um serviço corresponda ao número atualmente implementado no *cluster*. Os detalhes dessas operações são gravados no *etcd*, em que o *Controller-mananger* procura alterações por meio do servidor da API.

Os ***worker nodes*** são os servidores que executam o trabalho no *Kubernetes*. Um nó pode ser uma máquina virtual ou uma máquina física, dependendo do *cluster*. Cada nó único tem os serviços necessários para executar *pods* e é gerido pelos componentes principais. Os serviços incluídos no nó são:

- **Docker** - É responsável por baixar as imagens e iniciar os *containers*, onde é executado. Cada unidade de trabalho é implementada como uma série de *containers*.
- **kubelet** - Obtém a configuração de um *pod* do servidor da API e garante que os *containers* descritos estejam em execução. Este é o serviço responsável pela comunicação com o *Master Node*. Ele é responsável por retransmitir informações de e para os serviços do plano de controlo, bem como interagir com o armazenamento do *etcd* para ler detalhes da configuração ou gravar novos valores.
- **kube-proxy** - é executado em cada nó para lidar com a sub-rede e garantir que os serviços estejam disponíveis para o exterior, como um *proxy* de rede e um *load balancer* para um serviço. Ao mesmo tempo gere o roteamento de rede para pacotes TCP e UDP.

3.2.4 Visão geral do *Docker Swarm*

Docker Swarm, como o nome indica, é o gestor de *containers* nativo do *Docker*, que consiste basicamente no *Docker Engine* implementado em vários nós. Neste sistema os *Manager Nodes* gerem a organização e gestão do *cluster*.

Um serviço consiste em tarefas que podem ser executadas num nó do *Swarm*, que podem ser replicados para serem executados em vários nós. No modelo de serviços replicados, o *load balancer* e o DNS interno podem ser usados para fornecer terminais de serviço altamente disponíveis.

Esta aplicação foi projetada para trabalhar em torno de quatro princípios chave:

- Simples, mas poderoso, sem grande preocupação de *user-friendliness*;
- Seguro por padrão com certificados gerados automaticamente;
- Compatibilidade retroativa com componentes existentes;
- Arquitetura resistente a *single-point-of-failures*;

Como pode ser visto na figura 3.3, a arquitetura do *Docker Swarm* consiste em *Managers* e *Workers*. O *user* pode especificar o estado desejado para os serviços no *cluster do Swarm* usando arquivos YAML. À semelhança do *Kubernetes*, apresentamos aqui os termos associados ao *Docker Swarm*:

- **Nó** - é uma instância de um Swarm. Os nós podem ser distribuídos no local ou em *clouds*;
- ***Swarm*** - é um *cluster* de nós ou de *Docker Engines*. No modo *Swarm*, o *user* organiza os serviços em vez de correr os *containers*.
- ***Manager Nodes*** - estes nós recebem definições de serviço do *user* e despacham o trabalho para os *Workers*. Caso necessário os *Manager Nodes* podem executar tarefas dos *Worker Nodes*;

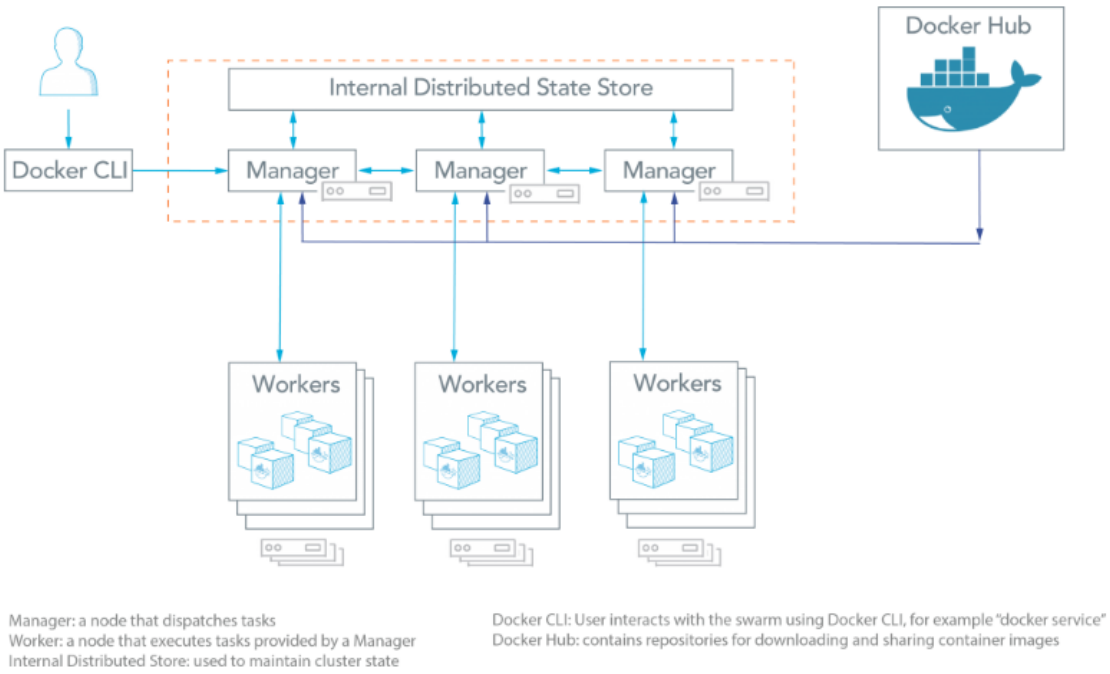

Figure 3.3: Visão geral do *Docker Swarm*(de [14])

- ***Worker Nodes*** - executam tarefas do *Manager Node*;
- **Serviço** - especifica a imagem do *containers* e o número de réplicas. Aqui está um exemplo de um comando de serviço que será criado em 2 nós disponíveis:

```
1        docker service create --replicas 2 --name mynginx nginx
2
```

- **Tarefa** - é uma unidade atómica de um serviço agendada num *Worker Node*. No exemplo acima, duas tarefas seriam agendadas por um *Master Node* em dois *Workers*. As duas tarefas serão executadas, de forma independente.

3.2.5 K8S vs *Mesos*

De forma a fazer uma análise extensa e concreta destas aplicações optei por dividir por atributos positivos e negativos e em que situações cada elemento se adequa. A área de gestão de *containers*, que tem tido um crescimento exponencial, dos elementos mais influentes têm sido *Kubernetes e Mesos*. As duas ferramentas são usadas para lançar *containers*. [25]

Em que situação se adequa?

Tendo em que, o K8S é uma comunidade de desenvolvimento altamente versátil , de código aberto e oferece um alto nível de portabilidade, já que é suportado pela Microsoft, IBM, VMWare etc., é ideal para quem quer iniciar-se na área de *clustering*.

Da perspetiva corporativa, esta plataforma é recomendada principalmente para empresas que estão prontas para a produção e precisam criar ambientes através de *containers*. É a melhor solução para sistemas altamente redundantes de média escala.

Já o *Mesos* é melhor para sistemas grandes e é projetado para redundância máxima, sendo recomendado para grandes cargas de trabalho como *Hadoop*. Com esta ferramenta têm-se uma estrutura que permite intercalar essas cargas entre si.

De forma curiosa, está a ser levada a adaptação de forma a adicionar conceitos e de forma a suportar a API de K8S, tornando-se num *gateway* para obter mais recursos para as aplicações do mesmo. Portanto, é a plataforma mais estável, mas excessivamente complexa para sistemas de pequena escala. É por estas razões que é normalmente recomendado e é melhor para empresas que usam *clusters* com várias nuvens e de várias regiões, como o *Twitter* ou a *Uber*.

Desempenho e Escalabilidade

O K8S é dimensionado para clusters de 5.000 nós. Vários clusters podem ser usados para escalar além desse limite. De acordo com a Digital Ocean, os clusters *Mesos* e *Marathon* foram dimensionados para 10.000 nós.

Load Balancing

No K8S, os *Pods* são expostos através de um serviço, que pode ser um *load balancer*. No *Mesos*, a aplicação pode ser acedida através do Mesos-DNS, que pode atuar como um gestor de carga básico.

Atualizações e Reversões

O K8S suporta *"rolling-update"* e *"recreate"*. Já no *Mesos* se uma atualização tem falhas pode ser reparada usando um *deployment* atualizado que reverte as alterações.

Health Checks

Para o K8S são dois para verificações: *liveness* e *readiness*, que são responsivos. O *Mesos* faz *health-checks* contra as tarefas das aplicações.

Alta disponibilidade

Em K8S, os *Pods* são distribuídos entre os *Worker Nodes*, tolerando assim falhas de infraestrutura ou de aplicações. Os serviços com *load balancer* detetam *pods* com erros e removem-nos. Os vários nós podem ser balanceados para solicitações do *kubectl* ou de clientes. O *etcd* pode ser armazenado num *cluster* e os servidores da API podem ser replicados.

Considerando que no *Mesos*, as aplicações são distribuídas entre os *Slave Nodes*. A alta disponibilidade para *Mesos e Marathon* é suportada usando o *Zookeeper*, que mantém o estado de *cluster*.

Armazenamento

No K8S, existem duas APIs de armazenamento: a primeira que fornece abstrações para *back-ends* de armazenamento individuais (por exemplo, NFS, AWS EBS, ceph, flocker). O segundo fornece uma abstração para uma solicitação do recurso de armazenamento que pode ser preenchida com *back-ends* de armazenamento de diferentes origens.

No *Mesos*, um *container* do *Marathon* pode usar volumes persistentes, como MySQL, mas esses volumes são locais para o nó em que são criados, portanto, o mesmo deve ser executado no nó onde foi criado. A integração do *floker* pode ser uma solução pois suporta volumes persistentes que não são locais para um nó.

Rede

O modelo de rede do K8S é plano e permite que qualquer *pod* comunique com outros *pods* ou serviços. O modelo requer duas redes, uma para os *pods* e outra para os *serviços*. No entanto, nenhuma das redes precisa estar acessível fora do *cluster*, podendo implantar-se uma rede de sobreposição.

No *Mesos*, a rede pode ser configurada no *host mode* ou no *brigde mode*. No *host mode*, as portas são usadas por *containers*, o que pode levar a conflitos de porta. No *brigde mode*, as portas do *containers* são conectadas a portas do *host* usando o mapeamento de portas.

Vantagens do *Kubernete*s em relação ao *Mesos*

- tem a maior comunidade com mais de 50.000 *commits* e 1.200 colaboradores;
- tem uma ampla variedade de opções de armazenamento, incluindo nuvens públicas;
- é suportado por várias plataformas como *Red Hat OpenShift e o Microsoft Azure*;
- o armazenamento externo no *Mesos*, incluindo o Amazon EBS, está em versão beta;
- a curva de aprendizagem é íngreme e complexa para o *Mesos*;

Desvantagens do K8S em relação ao *Mesos*

- foi construído somente para gestão de *containers*;
- para usar mais de 5.000 nós é necessário lançar vários *clusters*;

Conclusão da Análise

Ambas as tecnologias têm algo a ver com *containers* do *Docker* e fornecem um sistema de gestão para os mesmos para portabilidade e escalabilidade das aplicações. Tanto o K8S quanto o *Mesos* visam facilitar o *deployment* e o gestão de aplicações dentro de *containers* no *datacenter* ou na nuvem. No final, trata-se de encontrar a solução certa de gestão de *cluster* que se adapta ao *user* da empresa, dependendo de suas necessidades atuais e futuras.

3.2.6 *K8S vs Docker Swarm*

Igualmente como no capítulo anterior, realizou-se análise extensa e concreta destas aplicações mas realizando as secções por atributos, mas comparando o *Docker Swarm e o K8S*.

Definição de aplicação

Ao usar K8S, uma aplicação pode ser lançada usando uma combinação de *pods*, *deployments* e serviços. Ao mesmo tempo o *Kompose*, que é o oficial do K8S, também suporta *docker compose* de forma parcial. Já com o *Swarm* só se pode lançar serviços no *cluster*, apesar que os ficheiro YAML podem especificar *containers* e o *Docker Compose* pode lançar a aplicação.

Instalação e configuração

No K8S todo o processo de instalação é manual, é preciso planear bem para colocar a correr e difere de SO para SO.

Enquanto que o *Docker Swarm* é simples de instalar em comparação com o K8S. Com o *Docker*, apenas é necessário um conjunto de ferramentas para aprender a desenvolver o ambiente e a configuração. O *Docker Swarm* também fornece flexibilidade, permitindo que qualquer novo nó entre num cluster existente como um *Managers* ou um *Worker*.

Trabalhando em dois sistemas

Sendo o *Docker Swarm* é uma ferramenta do *Docker*, a mesma linguagem é usada para navegar dentro da estrutura. Esta situação fornece variabilidade e velocidade para a ferramenta e dá ao *Docker* uma vantagem significativa em termos de usabilidade.

Logging **e monitorização**

O K8S suporta múltiplas versões de *logging* e monitorização:

- ***Elasticsearch / Kibana***, para registar os *logs* dentro do *container*;
- ***Heapster / Grafana / Influx*** para a monitorização do *container*;
- ***Sysdig cloud integration***

Com o *Docker Swarm* é suportado apenas para monitorização com os aplicações de terceiros, normalmente é recomendado usar *Reimann*.

Escalabilidade

O K8S é mais uma *framework all-in-one* para sistemas distribuídos. É um sistema complexo, pois oferece um conjunto unificado de APIs e fortes garantias sobre o estado do *cluster*, o que atrasa o *deployment* e a escalabilidade de *container*.

Em comparação com o K8S, o *Docker Swarm* pode lançar *container* mais rapidamente e isso permite tempos de reação rápidos para escalar. [14]

Alta disponibilidade

No K8S, todos os *pods* são distribuídos entre os nós e os serviços de balanceamento de carga detetam *pods* com erros e removem-nos, fornecendo alta disponibilidade.

O *Docker Swarm* também fornece alta disponibilidade, pois os serviços podem ser replicados pelos nós.

Rede

A rede do K8S é plana, pois permite que todos os *pods* comuniquem entre si, no entanto o modelo requer dois Classless Inter-Domain Routing (CIDR). O primeiro requer *pods* para obter um endereço IP, o outro é para serviços.

Num *Docker Swarm*, um nó que se junta a um *cluster* cria uma rede de sobreposição de serviços que abrange todos os *hosts* no *Swarm* e cria só um *host* para aceder aos *containers*. Os *users* têm sempre a opção de criptografar o tráfego de dados do *container* ao criar a rede de sobreposição.

Conclusão da Análise

Apesar de serem plataformas para o mesmo efeito acaba por não ser fácil fazer um comparação. O *Docker Swarm* é uma solução simples e fácil de trabalhar, enquanto o K8S é voltado para aqueles que precisam de suporte total com maior complexidade.

O *Docker Swarm* é preferido em ambientes onde a simplicidade e o rápido desenvolvimento são favorecidos. Considerando que o K8S é adequado para ambientes em que *clusters* de médio a grande porte estão executando aplicações complexas.

3.2.7 Conclusões finais entre *Kubernetes*, *Mesos* e *Swarm*

Definição da Aplicação

- As aplicações podem ser lançadas usando uma combinação de *pods*, *deployments* e serviços.	- As aplicações podem ser lançadas como serviços num *cluster Swarm*.	- Do ponto de vista do *user*, uma app é executado como tarefas agendadas pelo *Marathon* nos nós.
- Um *pod* é um grupo de *containers* e é a unidade atómica de um *deployment*, que pode ter réplicas em vários nós .	- As aplicações *multicontainer* podem ser especificadas usando arquivos YAML.	- Para o *Mesos*, uma aplicação é uma *framework*, que pode ser *Marathon*, *Cassandra*, *Spark* e outros.
- Um serviço é a "face externa" das cargas de trabalho do *container* que se integração DNS usando *round-robin* para chegada de pedidos.	- O *Docker Compose* pode lançar a *app*.	- O *Marathon* executa os *containers* como tarefas que são executadas em *Slave Nodes*.
- O balanceamento de carga é suportado.	- As tarefas, uma instância de um serviço em execução num nó, podem ser distribuídas entre os *datacenters* usando *labels*.	- O *Marathon* 1.4 introduz o conceito de *pods*, como o K8S, mas isso não faz parte do núcleo da *Marathon*.

Construções em escalabilidade

- Cada camada da aplicação é definida como um *pod* e pode ser dimensionada quando gerida por um *deployment*, através de um ficheiro YAML. - O escalonamento pode ser manual ou automático. - Os *pods* são mais úteis para executar aplicações auxiliares, como agentes de backup e proxies. - Também podem ser usados para executar *stacks* de aplicacções verticalmente integradas, como *LAMP (Apache, MySQL, PHP) ou ELK /|Elastic (Elastic-search, Logstash, Kibana).*	- Os serviços podem ser escalados usando modelos YAML do *Docker Compose.* - Os serviços podem ser globais ou replicados. - Os serviços globais são executados em todos os nós, os serviços replicados executam tarefas dos serviços nos nós. Por exemplo, um serviço MySQL com 3 réplicas será executado no máximo em 3 nós. - As tarefas podem ser ampliadas ou reduzidas e lançadas em paralelo ou em sequência.	- *Mesos CLI* ou *UI* podem ser usados. - Os *containers* de *Docker* podem ser iniciados usando as definições JSON que especificam o repositório, os recursos, o número de instâncias e o comando a ser executado. - O aumento de escala pode ser feito usando a *Marathon* UI, e o gestor do *Marathon* distribui esses *containers* em *Slave Nodes* com base em critérios específicos. - O escalonamento automático é suportado e as aplicações multi-camada podem ser lançadas usando grupos de aplicações.

Alta disponibilidade

- Os *deployments* permitem que os *pods* sejam distribuídos entre nós para fornecer alta escalabilidade, tolerando assim falhas de infraestrutura ou de aplicações. - Serviços com load balancer detetam *pods unhealthy* e removem-nos. - A alta disponibilidade do K8S é suportada. - Vários *Masters Nodes* e *Worker Nodes* podem ser balanceados por carga para solicitações de *kubectl* e clientes. - O etcd pode ser armazenado em *cluster* e os servidores da API podem ser replicados.	- Os serviços podem ser replicados entre os nós do *Swarm*. - Os *Swarm* Managers são responsáveis por todo o *cluster* e gerem os recursos dos *Worker Nodes*. - Os *Managers* usam *load-balancers* para expor os serviços externamente. - Os *Swarm Managers* usam o algoritmo *Raft Consensus* para garantir que eles tenham informações de estado consistentes.	- Os *containers* podem ser programados sem restrições no posicionamento do nó ou em cada *container* num nó exclusivo (o número de *Slave Nodes* deve ser pelo menos igual ao número de *containers*). - Alta disponibilidade para *Mesos* e *Marathon* é suportada usando o *Zookeeper*. - *Zookeeper* mantém o estado de *cluster*.

Load Balancing

- Os *pods* são expostos por meio de um serviço, que pode ser usado como um *load balancer* no *cluster*.	- O modo *Swarm* possui um componente DNS que pode ser usado para distribuir pedidos recebidos para um serviço. - Os serviços podem ser executados em portas especificadas pelo *user* ou podem ser atribuídas automaticamente.	- As portas do *host* podem ser mapeadas para várias portas dos *containers*, servindo como um *front-end* para outras aplicações.

Escalonamento automático

- O escalonamento automático usando um número simples de *pods* é definido declarativamente usando *deployments*.	- Não está disponível diretamente.	- O *Marathon* monitoriza continuamente o número de instâncias de um *container* do *Docker* em execução.
- O escalonamento automático usando métricas de recursos também é suportado.	- Para cada serviço, pode-se declarar o número de tarefas que se deseja executar.	- Se um dos *containers* falhar, o *Marathon* reprograma-o noutro *Slave Nodes*.
- As métricas de recursos variam de utilização de *CPU* e memória a pedidos ou pacotes por segundo e até métricas personalizadas.	- Quando se aumenta ou diminui manualmente, o gestor do *Swarm* adapta-se automaticamente adicionando ou removendo tarefas.	- O escalonamento automático usando métricas de recursos está disponível somente através de componentes suportados pela comunidade.

Health Checks

- Os *health checks* são de dois tipos: ->**liveness**, a aplicação ser responsiva; ->**readinesss**, é responsivo à aplicação mas está ocupado	- Os *health checks* do *Docker Swarm* são limitados aos serviços. - Se um *container* que suporta o serviço não aparecer (estado de execução), um novo *container* é iniciado. - Os *users* podem incorporar a funcionalidade de *health checks* em imagens do *Docker*	- Os *health checks* podem ser especificados para serem executar contra as tarefas. - Os pedidos de *health checks* estão disponíveis em vários protocolos, incluindo *HTTP, TCP* e outros.

Networking

- O modelo de rede é *flat*, permitindo que todos os *pods* comuniquem uns com os outros. - Políticas de rede especificam como os *pods* comunicam entre si. - O modelo requer dois CIDRs: um dos quais os *pods* obtêm um endereço IP e o outro para os serviços.	- O nó que une um *cluster* do *Docker Swarm* cria uma rede de sobreposição para serviços e uma *bridge network* para *containers*. - Por padrão, os nós no *cluster* do *Swarm* criptografam o controlo de sobreposição e o tráfego de gestão entre eles. - Os *user*s podem optar por criptografar o tráfego de dados do *container* ao criar uma rede de sobreposição por conta própria.	- A rede pode ser configurada no modo *host* ou no modo *bridge*, isto pode levar a conflitos de porta em qualquer *host*. - As portas do *host* podem ser atribuídas dinamicamente no momento do *deployment*. No modo *host*, as portas são usadas por *containers*, no modo *brige* as portas do *container* são conectadas a portas do *host* usando mapeamento.

Desempenho e Escalabilidade

- Kubernetes escala para *cluster*s de 5.000 nós. - Vários *clusters* podem ser usados para escalar além desse limite.	- O *Docker Swarm* foi escalado e testado até 30.000 *containers* e 1.000 nós com um *Swarm Manager*.	- A arquitetura de 2 camadas do *Mesos* (com *Marathon*) é muito escalável. - De acordo com a Digital Ocean, os *clusters Mesos* e *Marathon* foram escalados para 10.000 nós.

Vantagens

- Baseado na vasta experiência da Google e ampla variedade de opções de armazenamento, incluindo nuvens públicas.	- *Deployment* simples com o modo *Swarm* e está incluído no *Docker Engine*.	- Com só um único fornecedor pode permitir melhores com correções de erros e melhor coordenação com o desenvolvimento.
- *Deployed* em escala com mais frequência. Apoiado pela *Google(GKE) e RedHat (OpenShift)*;	- Integra-se com *Docker Compose* e *Docker CLI*.	- A arquitetura de 2 camadas permite implantar outras estruturas como *Spark* ou Redis.
- Pode superar restrições do *Docker* e *Docker*API.	- Muitos dos comandos CLI do *Docker* funcionam com o *Swarm* (Curva de aprendizagem mais fácil).	- Pode superar as restrições da API *Docker* e *Docker*.
- Maior comunidade com mais de 50.000 *commits* e 1200 contribuidores.	- Vários *plugins* de opções de armazenamento incluem o *Azure*, o *Google Cloud Platform*, etc.	- Organizações lançaram o *Mesos* numa escala maciça superior a 10.000 nós.

Desvantagens

- A instalação pode ser complexa.	- Falta de experiência com implementações de produção em escala.	- Armazenamento externo no *Mesos + Marathon*, incluindo a *Amazon EBS* ainda é uma versão beta.
- Utiliza um conjunto separado de ferramentas para gestão, incluindo o *kubectl CLI*.	- Limitado aos recursos da API do *Docker*.	- A instalação pode ser complexa devido para a arquitetura de 2 camadas com *Zookeeper* para gestão do *cluster*, e *HA Proxy* como *load balancer*.
- A falta de controlo sobre a comunidade.	- Os serviços devem ser escalonados manualmente.	- Múltiplo conjunto de ferramentas para gestão: *Mesos CLI*, *Mesos* UI, *Marathon CLI*, *Marathon UI*.

3.2.8 Decisão final

A industria está rapidamente a reconhecer o *Kubernetes* como o principal serviço. As suas alternativas fornecem um mecanismo de gestão muito mais simples de usar, daí por vezes as grandes empresas optarem por usar *Swarm*, por ser mais simples do que fazer a transição para a *cloud* com o *Kubernetes*, especialmente se houver muita gente envolvida na migração. Para equipas mais pequenas é preferível usar soluções mais leves, como o Nomad, porque é muito mais simples e integra-se bem aos serviços existentes do *Consul* e do *Vault*.

Como o objetivo é implementar o serviço exclusivamente de gestão de *containers* do *Docker* para uma equipa de *devOps* e a integrar a nossa solução nas infraestruturas já existentes ou *clouds* é o que se adapta melhor à nossa situação. Toda esta análise e tabela foi baseada em [14].

Capítulo 4

Análise de Requisitos

Para a STF2.0 foram acordados os requisitos, em conjunto com as várias equipas da Feedzai. Alguns dos atributos foram impostos pela empresa e outros deles auto-propostos. [23]

Nesta secção são listados todos os requisitos, divididos em 2 categorias:

- **Requisitos Funcionais** : Estas são declarações de serviços que o sistema deve fornecer, como o sistema deve reagir a entradas específicas e como o sistema deve se comportar em situações específicas. Em alguns casos, os requisitos funcionais também podem indicar explicitamente o que o sistema não deve fazer. [22]

- **Requisitos Não Funcionais** : Essas são restrições nos serviços ou funções oferecidas pelo sistema. Eles incluem restrições de tempo, restrições no processo de desenvolvimento e restrições impostas pelos padrões. Requisitos não funcionais geralmente se aplicam ao sistema como um todo, em vez de recursos ou serviços individuais do sistema. [22]

Para priorizar os requisitos foi usado o método MoSCoW que é uma técnica de priorização usada em gestão de projetos e em desenvolvimento de software para alcançar um entendimento comum com os interessados sobre a importância que eles atribuem à entrega de cada requisito. Assim sendo, segundo o modelo, as funcionalidades serão divididas por 3 Prioridades: [13]

- **Must** - funcionalidades que são estritamente obrigatórias e que têm influência na atividade da STF2.0.;

- **Should** - funcionalidades que não são essenciais mas que adicionam qualidade à STF2.0;

- **Could** - funcionalidades básicas que resolvem situações de menor prioridade, ou seja, são desejáveis mas não necessárias. Estas servem para melhorar a interação com o *user* ou para satisfazer e são somente incluídos se o tempo e os recursos o permitirem;

4.1 Requisitos Funcionais

RF1. Suportar a integração da solução com CI e CD - A STF2.0 tem de **suportar mecanismos de CI e CD**. A STF2.0 deve permitir que todas as funções sejam executadas de forma automática e que se guardem os dados.
Prioridade: Must

RF2. Suportar a integração de componentes externos - A STF2.0 tem de **suportar vários componentes externos**, a serem usados para testes, desde bases de dados relacionais e não relacionais a bibliotecas. Alguns destes componentes são: Zookeeper, RabbitMQ ,Postgres, Oracle,Cassandra, Spark ou Hadoop/YARN.
Prioridade: Must

RF3. Suportar a integração de componentes internos - A STF2.0 tem de **suportar todos os componentes internos**, de forma a que a *framework* seja global dentro a Feedzai. A questão da globalização é crucial para este projeto. Os componentes em causa são: Pulse, Case Manager, Insights, Genome, Tokenizer.
Prioridade: Must

RF4. Suportar integração com os *containers* do Chrome e Firefox - A STF2.0 tem de suportar a **integração com estes *browsers*** de forma a permitir testes de GUI.
Prioridade: Must

RF5. Suportar o *deployment* de ambientes com diferentes configurações - A STF2.0 tem de **suportar várias configurações** dependendo do tipo de testes. Cada equipa deve escolher os componentes e ambientes que quer testar. Estes teste podem ser funcionais, de tolerância a falhas, de desempenho, etc.
Prioridade: Must

RF6. Suportar a interação e gestão dos *containers* em execução - A STF2.0 tem de **aceder e gerir os componentes** que são usados nos testes, de forma a aceder a *logs* e a dados dos *containers*.
Prioridade: Must

RF7. Suportar ferramentas de *testing* - A STF2.0 tem de **suportar frameworks de tSeste** como *JUnit* e *Scala Rest* são usados em vários departamentos da Feedzai.
Prioridade: Must

RF8. Suportar conexão de bibliotecas externas usadas em *System Tests* - A STF2.0 tem de suportar **bibliotecas externas a ser usadas para gerir as API's de produto** como RMI, HTTPClient, etc. . Além dessa situação, estas configurações devem ser costumizáveis por cada equipa.
Prioridade: Must

RF9. Ter suporte para *Logs* - A STF2.0 tem de **suportar mecanismos de recolha de logs** pois para a realização de *debug* é extremamente necessário;
A cada instrução realizada ou erro, durante os testes, deve existir uma entrada no ficheiro de *log*, previamente criado para o conjunto de testes definidos. Este ficheiro de *output* deve aglomerar todos os testes iniciados pelo *tester* e organizar por módulos.
Prioridade: Must

RF10. Ter um modo *debug* - A STF2.0 tem de **suportar mecanismos de debug**, de forma a conseguir detetar mais facilmente erros ou *bugs* no sistema.
Prioridade: Must

RF11. Suportar mecanismos de recuperação de falhas - a STF2.0 deve de **parar em caso de falha, e criar um "estado estável"**. Este estado vai servir como ponto de partida para o desenvolvedor, que posteriormente, irá depurar o problema.
Prioridade: Should

RF12. Suportar *docker-compose* e ficheiros de configuração de K8S - a STF2.0 deve suportar **configurações para os testes usando *docker-compose* e ficheiros de configuração de K8S**, pois é uma forma simples de configurar os ambientes de testes e ao mesmo tempo já está a ser usado pelas equipas de *Solutions (Banking e Merchants)* e *Costumer Success*.
Prioridade: Must

RF13. Suportar *clusters* locais de K8S - a STF2.0 tem de **conseguir lançar os ambientes em *cluster* de K8S alojado na própria máquina**, como *minikube*.
Prioridade: Must

RF14. Suportar *clusters* externos de K8S - a STF2.0 tem de **conseguir lançar os ambientes em *cluster* de K8S que não está alojado na máquina local**, como o *cluster da Feedzai*.
Prioridade: Must

RF15. Suportar apresentação serviços ou instâncias - A STF2.0 poderá **apresentar serviços ou instâncias** previamente criadas, testadas ou *default* **numa lista**.
Prioridade: Could

RF16. Suportar filtro de procura de serviços ou instâncias segundo parâmetro - A STF2.0 poderá, através de um filtro, **mostrar uma lista segundo um parâmetro definido pelo utilizador**. Estes parâmetros poderão ser nome, categoria ou estado.
Prioridade: Could

RF17. Suportar filtro de serviços ou instâncias por categoria - A STF2.0 poderá **filtrar os serviços ou instâncias por categoria, por exemplo retornar só os testes com *PostGres***. Ao percorrer a lista só os da categoria procurada devem ser apresentados.
Prioridade: Could

RF18. Suportar filtro de serviços ou instâncias por nome - A STF2.0 poderá **filtrar os serviços ou instâncias por um nome específico**. Ao percorrer a lista só os da categoria procurada devem ser apresentados.
Prioridade: Could

RF19. Suportar filtro de serviços ou instâncias por estado - A STF2.0 poderá **filtrar os serviços ou instâncias por estado, por exemplo, procurar só pelos testes com falhas**. Ao percorrer a lista só os da categoria procurada devem ser apresentados.
Prioridade: Could

RF20. Suportar exportação de logs - A STF2.0 poderá **exportar os ficheiros de *log* em formato .log**, para que haja hipótese de ver o ficheiro em qualquer Ambiente de desenvolvimento integrado (IDE) de texto.
Prioridade: Could

A figura 4.1 demonstra o diagrama de casos de uso tendo em conta os requisitos funcionais.

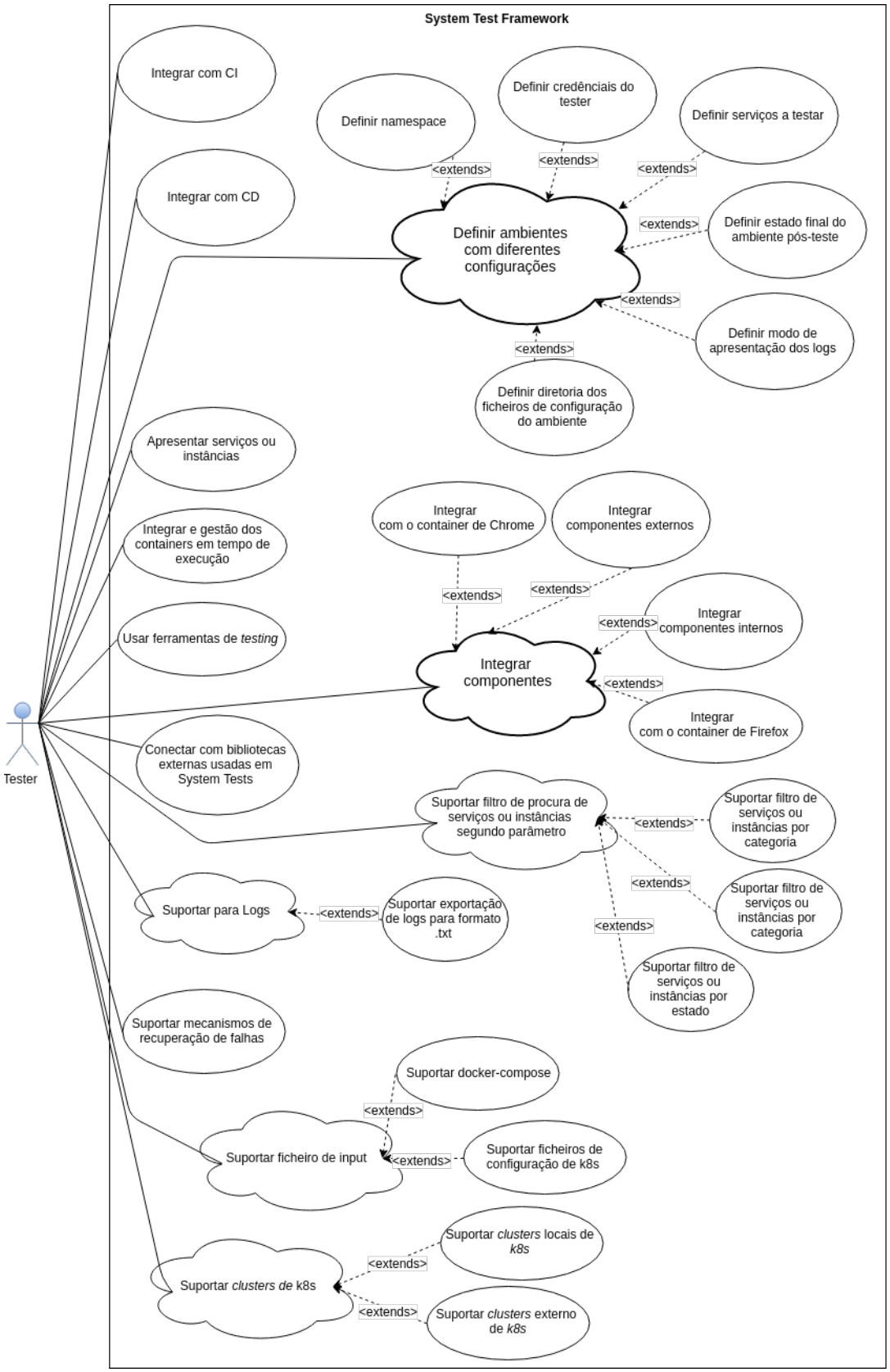

Figure 4.1: Diagrama de Casos de Uso

4.2 Requisitos Não Funcionais

Este tipo de requisitos estão divididos em cinco categorias. [23]

4.2.1 Escalabilidade

Escalabilidade é a capacidade atender à crescente demanda de pedidos ao sistema. Os critérios de escalabilidade do sistema podem incluir a capacidade de acomodar um número crescente de:

- *users*
- número de testes a correr em paralelo

Neste momento, a dimensão de dados estão a crescer exponencialmente, as expectativas de desempenho do sistema seguem o mesmo sentido. Desta forma, é necessário garantir que a STF2.0 consiga aguentar o número de testes que são pedidos em paralelo.

E1. Suportar testes em paralelo e conseguir escalar horizontalmente - A STF2.0 tem de integrar *Kubernetes* para suportar o escalonamento horizontal com base na capacidade dos *clusters* das várias máquinas usadas nos testes.
Prioridade: Must

Influência na arquitetura: Para suportar este requisito foi necessário criar uma relação de 1 para N entre a API de K8S e o *Test Environment*. E ao mesmo tempo levou à implementação da *hash*.

4.2.2 Portabilidade

A portabilidade em software é a capacidade deste puder ser usado em um SO diferente daquele em que foi criado, migrações profundas. Neste caso específico o facto de este programa ter de estar adaptado a trabalhar em *Ubuntu* e *MacOS*, que são os sistemas usados na Feedzai, nas equipas de engenharia.

P1. Suportar execução para sistemas MacOs e Ubuntu - A STF2.0 tem de **suportar os sistemas usados nos dispositivos da Feedzai**, que são MacOS e Ubuntu.
Prioridade: Must

Influência na arquitetura: Para suportar este requisito foram necessários adaptar a arquitetura para que ambos os sistemas sejam incluídos.

4.2.3 Desenvolvimento

Um processo de desenvolvimento de software é um conjunto de atividades, parcialmente ordenadas, com a finalidade de obter um produto de software.

D1. Desenvolvida em Java - A STF2.0 deve ser desenvolvida tendo em conta que é a linguagem mais utilizada nos projetos da Feedzai e aquela que é mais confortável para os seus *developers* de forma a facilitar a manutenção da mesma.
Prioridade: Should

Influência na arquitetura: Para suportar este requisito toda a STF2.0 foi desenvolvida em Java.

4.2.4 Usabilidade

Usabilidade é um termo usado para definir a facilidade com que as pessoas podem empregar uma ferramenta ou objeto a fim de realizar uma tarefa específica e importante. A usabilidade pode também se referir aos métodos de mensuração da usabilidade e ao estudo dos princípios por trás da eficiência percebida de um objeto. [20]

U1. Apresentar os erros de forma destacada - A STF2.0 deve **apresentar os erros que decorrem ao longo do processo de forma destacada**, contornado a encarnado e indicando a razão para o mesmo erro.
Prioridade: Should

Influência na arquitetura: Para suportar este requisito foi necessário criar um cliente, que acede por terminal.

U2. Descrever os inputs - A STF2.0 deve fornecer uma **breve descrição de cada input pedido** e indicar as funcionalidades disponíveis, de forma a facilitar a adaptação.
Prioridade: Should

Influência na arquitetura: Para suportar este requisito foi necessário adicionar um ficheiro de texto com a descrição completa dos inputs da STF2.0, do tipo README, para que o cliente possa aceder.

U3. Fornecer documentação - A STF2.0 deve **ter um ficheiro README** de de maneira a explicar de forma breve como usar a ferramenta.
Prioridade: Should

Influência na arquitetura: Para suportar este requisito foi necessário adicionar um ficheiro de texto com notas para facilitar a execução, do tipo README, para que o cliente possa aceder.

U4. Limitar hipóteses de input - A STF2.0 poderá **limitar certos parâmetros** de forma a não exceder os recursos existentes.
Prioridade: Could

Influência na arquitetura: Para suportar este requisito foram colocados os dois *inputs* possíveis, ficheiro de configuração de K8S, no formato .yaml, e o ficheiro de docker-compose, no formato, .yml.

U5. Minimizar ambiguidade - A STF2.0 poderá **conter frases que sejam entendidas pelos utilizadores**. É muito importante que os users percebam de forma clara o que é escrito nos comentários sobre o funcionamento da plataforma.
Prioridade: Could

Influência na arquitetura: Para suportar este requisito foi necessário adicionar um ficheiro de texto com notas para facilitar a compreensão, do tipo README, para que o cliente possa aceder. Ao mesmo tempo, a arquitetura é o mais clara e concisa possível.

4.2.5 Suportabilidade

Suportabilidade é um dos aspetos da qualidade de software que se refere à habilidade de um suporte técnico de instalar, configurar e monitorizar produtos computacionais, iden-

tificar exceções ou falhas, depurar ou isolando problemas até a fonte. Ao mesmo tempo prover manutenção de software ou hardware a fim de solucionar um problema ou restaurar o produto a um estado funcional. [17]

S1. Ser testável - A STF2.0 tem, a partir do momento em que são propostos os requisitos para lançar os componentes, **estes devem ser validados**.
Prioridade: Must

Influência na arquitetura: Para suportar este requisito foi necessário o teste de Junit, criado para executar contra os componentes. Ao mesmo tempo, foram adicionados os endpoints, em HTTP, onde os testes vão aceder.

S2. Documentar todos os requisitos funcionais - A STF2.0 deve, de forma a conseguir validar e ter boa manutenção dos requisitos, **ter todos os requisitos funcionas documentados de forma clara e inequívoca**. À descrição pode ser adicionado Casos de Uso ou *Mock-Ups*.
Prioridade: Should

Influência na arquitetura: Para suportar este requisito foi necessário que todos os componentes que são necessários para a implementação da STF2.0 estejam inseridos na arquitetura.

Capítulo 5

Arquitetura de solução

Este capítulo está dividido em 2 secções. Na primeira é apresentada uma arquitetura abstrata. A segunda apresenta um diagrama de componentes e uma explicação correspondente.

5.1 Primeiro Nível

Neste nível a arquitetura é apresentada de forma abstrata, na **figura 5.1**, para dar uma ideia geral de como a STF2.0 vai operar com o cliente, *Docker* e K8S. Inicialmente o cliente define o ficheiro que quer usar para as configurações e o teste a realizar. A STF2.0 recebe os dados do cliente, executa os comandos da *API* de K8S que vai criar e executar todas as operações sobre os *containers*, alojadas no *cluster*. De ter em conta, que o papel do K8S é criar os ambientes de forma distribuída onde vão ser executados os testes. O ambiente, onde vai ser executado todo o processo, depende da escolha do cluster e da máquina ou máquinas que o utilizador usar.

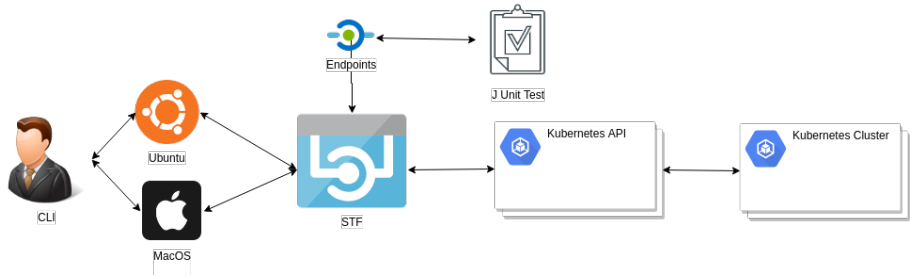

Figure 5.1: Arquitetura de Primeiro nível

5.2 Segundo Nível

Neste capítulo, assumimos que o componente é um artefacto de software. Tendo em conta a **figura 5.2**, que se encontra em maior plano no **apêndice C**, podemos concluir que o cliente é que executa a STF2.0, através da linha de comandos, após definir o ficheiro de configuração que pode ser de dois tipos:

- ficheiro de *docker-compose*, que tem de sofrer uma conversão para o ser lido em K8S;

- ficheiro de configuração nativo de K8S;

O passo seguinte é a execução do *Parser* que vai receber o ou os ficheiros e converter para um modelo de serviços e *deployments*. Assim que este passo é realizado o *Handler* vai começar a fazer chamadas à API de K8S de forma a criar o *namespace* no *cluster* especificado. Posteriormente, é criada uma *hash* para cada um dos testes a realizar, que vai ficar associada a todos os componentes desse teste.

A partir deste ponto, o *Handler* faz chamadas à API para criar todos os componentes, *pods*, *deployments* e serviços. Estes últimos vão ter sempre um *load balancer* associado de forma a existir uma forma de acesso a cada serviço fora do *cluster*. Assim que o ambiente se encontra lançado, é executado o teste especificado pelo cliente e após a execução deste, com sucesso ou não, são guardados os *logs* num ficheiro .log numa diretoria especificada.

De denotar que dentro do mesmo *cluster* podem ser criados vários *namespaces* baseados do tipos de componentes que se executa. E ao mesmo denotar que existem várias relações de *1 to Many*:

- para cada execução podem existir vários ficheiros de .yml ou .yaml;
- para cada execução podem existir vários *hashed environements*, dependendo dos ficheiros de configuração dos utilizadores;
- Cada *hasehd environment* pode ter vários componentes, de qualquer tipo;

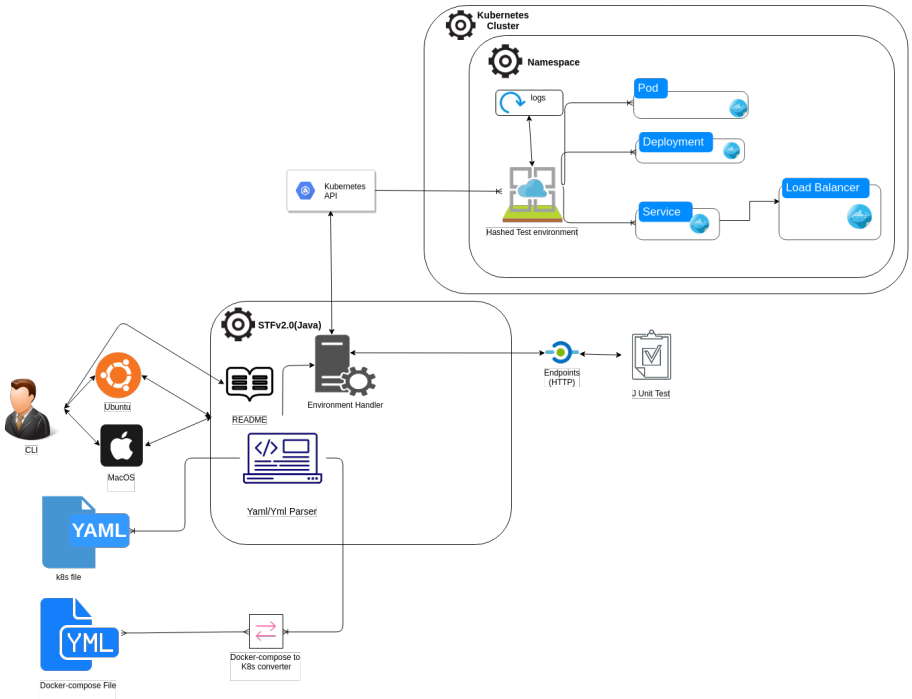

Figure 5.2: Arquitetura de segundo nível da STF2.0

5.2.1 Componentes entre o k8s

Todos os componentes enunciados foram explicados na secção 3.1.3 . Neste diagrama já se tem uma noção mais profunda do que compõe os nós que são lançados pelo K8S.

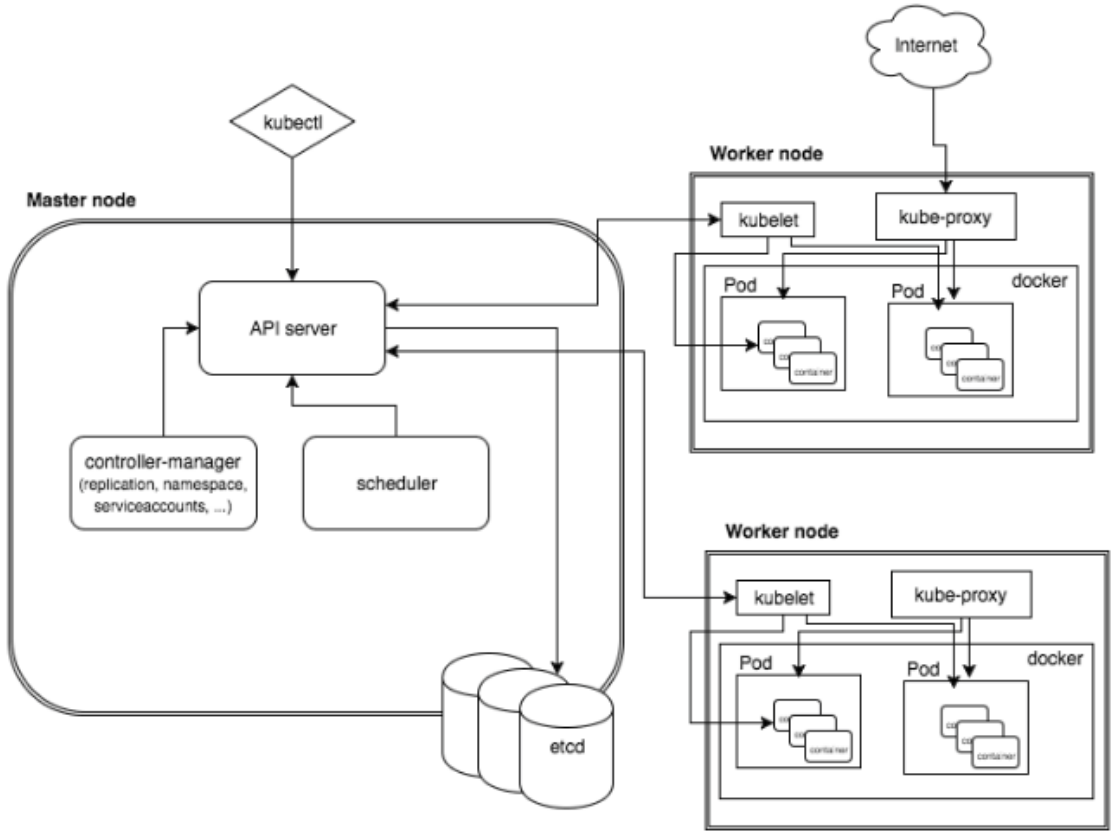

Figure 5.3: Arquitetura dos componentes do K8S (de [8])

No master podemos verificar que a API recebe os comandos do *kubectl*, do *controller-manager* recebe o número de replicas de cada *pod*, os *endpoints* e gere as mudanças. O *scheduler* interage com a API de forma a organizar todos os *deployments*. O *etcd* serve para guardar ou ler os dados do *cluster*.

Já nos *Worker Nodes* temos um *kublet* que chama ou é chamado pela API, sendo que trata das especificações recebidas e controla o estado dos *pods* e reporta ao *Master Node*. Cada um destes nós terá um *proxy* através do qual podem ser acedidos do exterior.

5.2.2 Componentes do *POD*

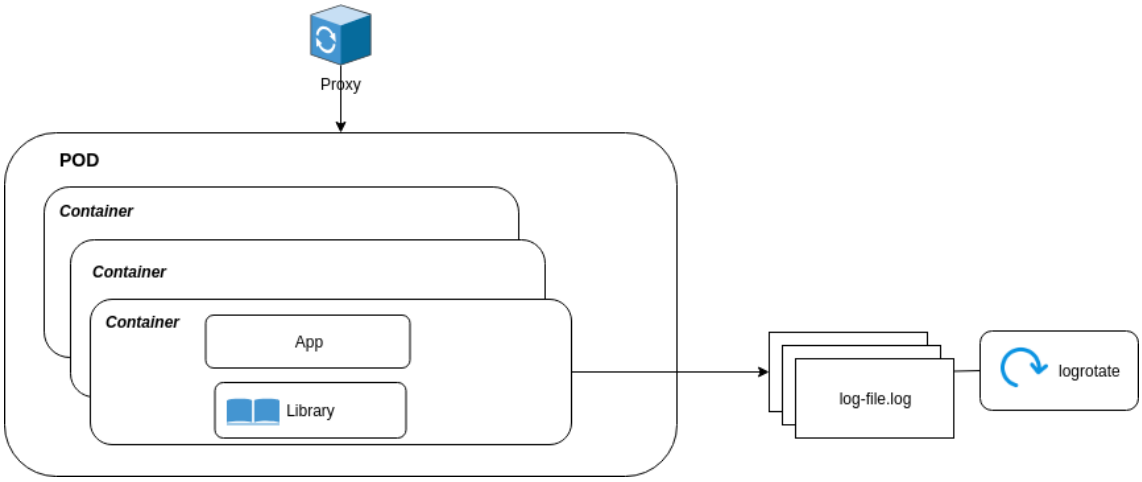

Figure 5.4: Arquitetura dos componentes do POD

Os *pods* como demonstrado no diagrama anterior são imagens de *Docker*. Estas imagens contêm *containers* que possuem a aplicação que vai correr e todas as livrarias necessárias à mesma. Cada POD por ter um ou vários *containers*, dependendo do número de réplicas lançado.

Capítulo 6

Planeamento

Como qualquer projeto, de desenvolvimento de sistemas ou de código, é preciso fazer uma análise exaustiva do planeamento. Nesta secção vamos definir a metodologia de trabalho, as razões que levaram a esta decisão e os riscos do projeto. Na terceira e última secção é apresentado o escalonamento previsto e real.

6.1 Metodologia de Desenvolvimento

A STF2.0 é um projeto inovador, sendo que é uma nova *framework* é necessário ter em conta que se tem de integrar alguns testes e componentes já previamente criados na empresa. Sendo que o prazo não é propriamente longo e o ser uma situação praticamente nova ficou patente que a melhor forma era usar um método ágil.

Tendo em conta que não existia conhecimento deste tipo de *clustering*, das ferramentas e dos componentes da empresa, teve de se optar inicialmente por investir tempo em fazer *on-boarding*, levantamento de requisitos da aplicação a desenvolver. Após a conclusão deste último passo, seguiu-se para a fase de desenho da arquitetura e da especificação dos diagramas de classes e de entidade e relacionamento.

No segundo semestre, e realizando todos os passos anteriores, passou-se à implementação da solução proposta. Foi usada a metodologia *Personal Kanban* [7] e as razões prendem-se com o facto deste tipo de metodologia permitir uma gestão flexível do projeto, baseado em iterações de muito curta duração, de forma a ter um controlo próximo sobre trabalho desenvolvido e de forma que não existam grandes desvios do plano inicial. Ao mesmo tempo tem-se flexibilidade para adaptação a possíveis alterações do projetado.

Ao mesmo tempo tem as vantagens de:

- **Visualizar o trabalho de perto:** Permite em qualquer altura ter noção da carga de trabalho a ser executada. Facilita a visão da tarefa a realizar a seguir, tendo em conta prioridades.
- **Limitar o trabalho em progresso:** Permite limitar o número de tarefas a realizar ao mesmo tempo, facilitando o *tracking* do projeto e evita os perigos do multitasking, que pode levar ao *burnout*.

Sendo que este método, deve seguir uma tabela específica exemplificada, na **figura 6.1**. De denotar que não foi usado um quadro físico mas sim digital.

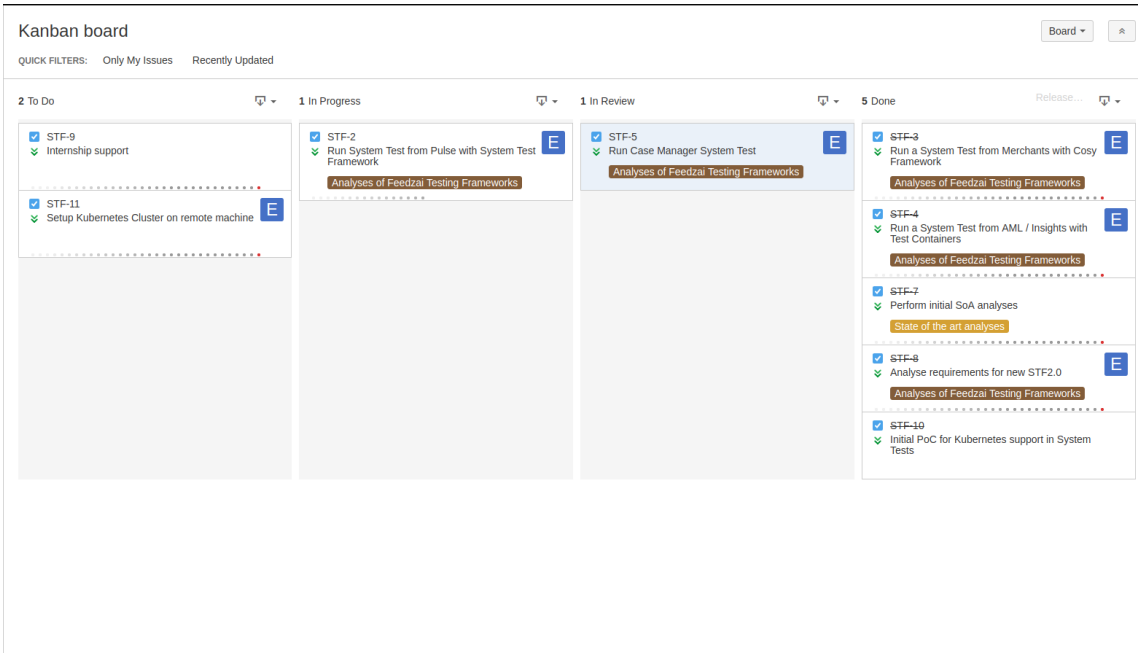

Figure 6.1: Exemplo do quadro do método *Personal Kaban*

6.2 Riscos

Um risco é qualquer eventualidade que possa, potencialmente, ter um impacto positivo ou negativo no projeto a decorrer. Um risco identificado pode não vir a ter qualquer consequência no objetivo do projeto, no entanto para prevenir problemas inesperados é necessário que estes se identifiquem à priori e que se faça uma gestão dos mesmo de forma a que possam ser controlados. Para além disso, para um risco identificado deverá ser estudada uma forma de minimizar os custos e problemas que podem vir a causar no projeto.

A gestão de riscos é um processo de identificação, avaliação, mitigação, monitorização e estudo de resposta a cada risco. Esta gestão é realizada em todos os ciclos do projeto, de forma a que o projeto possa permanecer no caminho certo, cumprindo todos os objetivos e a data limite de entrega. Este processo é realizado durante todos os ciclos de vida do projeto, visto que os riscos são mutáveis e a sua classificação também.

6.2.1 Limiar de Sucesso(ToS)

- Entregar todos os requisitos de prioridade "Must" até ao fim do estágio;

6.2.2 Descrição Riscos

R-1	**Falta de experiência com a tecnologia *Docker***
Descrição	O aluno em causa não tinha experiência com a tecnologia *Docker*
Consequências	- Dificuldade em estimar tempo das tarefas - Dificuldade em definir uma arquitetura no processo inicial - Trabalho pode perder qualidade - Atrasos nas entregas
Intervalo de Tempo	Extenso
Impacto	Crítico
Probabilidade	Muito elevada

Table 6.1: Tabela do R-1

R-2	**Falta de experiência com a tecnologia *k8s***
Descrição	O aluno em causa não tinha experiência com a tecnologia *k8s*
Consequências	- Dificuldade em estimar tempo das tarefas - Dificuldade em definir uma arquitetura no processo inicial - Trabalho pode perder qualidade - Atrasos nas entregas
Intervalo de Tempo	Extenso
Impacto	Crítico
Probabilidade	Muito elevada

Table 6.2: Tabela do R-2

R-3	**Alterações aos requisitos ao longo do projeto**
Descrição	Alterações aos requisitos ao longo do projeto
Consequências	- Alterações ao planeamento do projeto - Perdas de tempo indesejadas
Intervalo de Tempo	Médio
Impacto	Crítico
Probabilidade	Reduzida

Table 6.3: Tabela do R-3

R-4	**Indisponibilidade de um dos orientadores**
Descrição	Um dos orientadores estar indisponível devido a fatores externos como doença ou trabalho
Consequências	- Atraso na entrega - Falta de conselhos que podem levar a erros no projeto - Falta de controlo
Intervalo de Tempo	Curto
Impacto	Elevado
Probabilidade	Reduzida

Table 6.4: Tabela do R-4

R-5	**Falta de conhecimento das *frameworks* da empresa**
Descrição	O aluno não tem conhecimentos das várias *frameworks* da empresa, enunciadas no documento
Consequências	- Atraso na adaptação e desenvolvimento - Erros de integração - Atraso na entrega
Intervalo de Tempo	Longo
Impacto	Médio
Probabilidade	Elevada

Table 6.5: Tabela do R-5

Legenda

Todos os riscos serão classificados em três categorias: intervalo de tempo, impacto e probabilidade de ocorrência. Estas três categorias dividem-se em 4 patamares descritos de seguida:

Impacto

- **Crítico** - Limiar de sucesso não obtido.
- **Médio** - Limiar de sucesso obtido com um esforço adicional pouco significativo.
- **Baixo** - Limiar de sucesso obtido com um esforço adicional não significativo.

Probabilidade

- **Muito elevada** - Probabilidade de ocorrência superior a 90%.
- **Elevada** - Probabilidade de ocorrência entre 70 a 90%.
- **Média** - Probabilidade de ocorrência entre 40 a 70%.
- **Reduzida** - Probabilidade de ocorrência inferior a 40%.

Intervalo de Tempo

- **Extenso** - Duração total do projeto.
- **Longo** - Dois meses.
- **Médio** - Um mês.
- **Curto** - Duração inferior a um mês.

6.3 Escalonamento de Tarefas

Para controlar o desenvolvimento foi criado um escalonamento. Este planeamento foi feito de forma muito genérica, mas no primeiro semestre era previsto que o estagiário tivesse o primeiro contacto com a empresa e a metodologia de trabalho da mesma. Ao mesmo tempo, especificar os requisitos e realizar o estado da arte e conseguir ter contacto com as tecnologias a utilizar no segundo semestre.

Este escalonamento cobre o período de 2 de outubro, dia de início do estágio, a dia 9 de julho, dia da apresentação formal da tese ao júri. Existiu um ligeiro atraso do estágio devido a alguns problemas logísticos entre a Universidade de Coimbra (UC) e a empresa Feedzai.

Neste momento, já é possível ter, concretamente, todas as tarefas realizadas em ambos os semestres e o tempo despendido nas mesmas. Da mesma forma, é possível saber as variações entre o tempo esperado para a realização do projeto e do que foi gasto na realidade.

No primeiro semestre todas as tarefas foram realizadas, apesar que em relação à tese escrita foram feitas alterações à entrega intermédia, no 2º semestre. Todo o conteúdo necessário para a realização da tarefa foi obtido no tempo devido, sendo realizado um Estudo da Arte aprofundado.

Todas as tarefas propostas, para o 2º semestre, tendo em conta os requisitos pedidos pela entidade patronal, foram realizados com sucesso. Após duas apresentações internas, para conseguir ter *feedback* de elementos fora do projeto e com experiência de desenvolvimento, foram adicionas e alteradas tarefas do plano inicial.

Nas seguintes secções serão demonstradas as tabela do método de gantt, já os gráficos encontram-se no **Apêndice B Planeamento**. Da mesma forma serão apresentados os desvios, entre a versão prevista e a real, para ambos os semestres.

6.3.1 Versão planeada

Esta secção contém o projeção, realizada no início do mestrado, do tempo a ser despendido na realização de todos os componentes do projeto.

6.3.1.1 Primeiro Semestre

	Task Name	Duration	Start	End	Predecessors
1	**1º Semestre**	**85 days**	**02/10/2018**	**28/01/2019**	
2	**1. Onboarding na Feedzai**	**32 days**	**02/10/2018**	**14/11/2018**	
3	**1.a Familiarização com o projeto e empresa**	**32 days**	**02/10/2018**	**14/11/2018**	
4	1.a.i Entender os prinicípios e conhecer a equipa	2 days	02/10/2018	03/10/2018	
5	1.a.ii Configuração da máquina e das credências	2 days	04/10/2018	07/10/2018	
6	1.a.iii Conhecer os produtos existentes, ex. Pulse	28 days	08/10/2018	14/11/2018	
7	**2. Análise de Requisitos**	**6 days**	**15/11/2018**	**22/11/2018**	**2**
8	2.a Recolha	2 days	15/11/2018	17/11/2018	
9	2.b Análise	4 days	18/11/2018	22/11/2018	
10	**3. Estado da Arte**	**44 days**	**22/11/2018**	**22/01/2019**	
11	3.a Investigação	7 days	22/11/2018	30/11/2018	
12	3.b Análise	7 days	30/11/2018	10/12/2018	
13	3.c Testes complementares	3 days	18/01/2019	22/01/2019	
14	**4. Escrita do Relatório Intermédio**	**29 days**	**10/12/2018**	**17/01/2019**	
15	4.a Capitulo 1	3 days	10/12/2018	12/12/2018	
16	4.b Capitulo 2	5 days	13/12/2018	19/12/2018	
17	4.c Capitulo 3	7 days	19/12/2018	27/12/2018	
18	4.d Capitulo 4	5 days	27/12/2018	02/01/2019	
19	4.e Capitulo 5	3 days	02/01/2019	04/01/2019	
20	4.f Capitulo 6	4 days	04/01/2019	09/01/2019	
21	4.g Capitulo 7	3 days	09/01/2019	11/01/2019	
22	4.h Capitulo 8	0 days	11/01/2019	11/01/2019	
23	4.i Correções	5 days	11/01/2019	17/01/2019	
24	5. Plano de trabalho para o 2º Semestre	1 day	18/01/2019	18/01/2019	
25	6. Entrega da Meta Intermédia	1 day	23/01/2019	23/01/2019	
26	**7. Preparação da Defesa Intermédia**	**4 days**	**23/01/2019**	**28/01/2019**	
27	7.a Recolha da informação a apresentar	1 day	23/01/2019	23/01/2019	
28	7.b Criação da Apresentação	3 days	23/01/2019	27/01/2019	
29	7.c Estudo da Apresentação	0 days	28/01/2019	28/01/2019	

Figure 6.2: Tabela do Escalonamento prevista do primeiro semestre

6.3.1.2 Segundo Semestre

	Task Name	Duration	Start	End	Predecessors
1	**2º Semestre**	**152 days**	**06/02/2019**	**09/07/2019**	
2	**1. Desenvolvimento da Framework em Java**	**112 days**	**06/02/2019**	**30/05/2019**	
3	1.a Criar uma load balancer para um container	1 day	06/02/2019	06/02/2019	
4	1.b Obter endereço de acesso para um container	1 day	07/02/2019	07/02/2019	
5	1.c Lançar um teste local de pequena complexidade	1 day	07/02/2019	07/02/2019	
6	1.d Criar função para obter todos os componentes	2 days	08/02/2019	11/02/2019	
7	1.e Criar função para obter logs de cada container	2 days	11/02/2019	12/02/2019	
8	1.f Lançar testes locais em paralelo	3 days	12/02/2019	14/02/2019	
9	1.g Automação dos processos criados nos pontos 3-8	7 days	15/02/2019	25/02/2019	
10	1.h Realizar escalonamento horizontal	7 days	25/02/2019	05/03/2019	
11	1.i Lançar a STF no cluster de k8s da Feedzai	5 days	06/03/2019	12/03/2019	
12	1.j Criar conversor de ficheiros de docker para kubernetes	3 days	12/03/2019	14/03/2019	
13	1.k Criar função para lançar serviços e deployments automáticamente	7 days	14/03/2019	22/03/2019	
14	**1.l Entrega da STF como prova de conceito**	**3 days**	**22/03/2019**	**24/03/2019**	
15	1.l.i Correcção baseda na review do orientador	3 days	22/03/2019	24/03/2019	
16	**1.m Apresentação da PoC à equipa de qualidade**	**1 day**	**28/03/2019**	**28/03/2019**	
17	1.m.i Revisão das features a criar	1 day	28/03/2019	28/03/2019	
18	1.n Criar método de lançamento dos componentes com hash	15 days	28/03/2019	17/04/2019	
19	1.o Automação do lançamento dos processos no cluster local e externo	7 days	17/04/2019	25/04/2019	
20	1.p Intergração dos componentes internos da Feedzai	20 days	25/04/2019	22/05/2019	
21	1.q Testes à framework	3 days	22/05/2019	24/05/2019	
22	1.r Escrita do manual	3 days	28/05/2019	30/05/2019	
23	2. Apresentação final à empresa da SFT2.0	1 day	30/05/2019	30/05/2019	
24	**3. Escrita contínua da Tese**	**20 days**	**30/05/2019**	**26/06/2019**	
25	3.a Correção pós defesa intermédia	7 days	30/05/2019	07/06/2019	
26	3.b Correção Final	17 days	07/06/2019	23/06/2019	
27	4. Entrega final da tese	1 day	01/07/2019	01/07/2019	
28	5. Criar Apresentação final	7 days	01/07/2019	09/07/2019	

Figure 6.3: Tabela do Escalonamento prevista do segundo semestre

6.3.2 Versão Real

Esta versão é a que realmente aconteceu ao longo do projeto. Estas datas foram recolhidas usando a ferramenta *clockify* [2], de forma a manter *tracking* do tempo despendido na realização tarefas do projeto.

6.3.2.1 Primeiro Semestre

	Task Name	Duration	Start	End	Predecessors
1	**1º Semestre**	**84 days**	**02/10/2018**	**28/01/2019**	
2	**1. Onboarding na Feedzai**	**14 days**	**02/10/2018**	**21/10/2018**	
3	**1.a Familiarização com o projeto e empresa**	**14 days**	**02/10/2018**	**21/10/2018**	
4	1.a.i Entender os prinicípios e conhecer a equipa	2 days	02/10/2018	03/10/2018	
5	1.a.ii Configuração da máquina e das credências	2 days	04/10/2018	07/10/2018	
6	1.a.iii Conhecer o Pulse	10 days	08/10/2018	21/10/2018	
7	**2. Análise de Requisitos**	**20 days**	**22/10/2018**	**16/11/2018**	**2**
8	2.a Recolha	10 days	22/10/2018	02/11/2018	
9	2.b Análise	11 days	02/11/2018	16/11/2018	
10	**3. Estado da Arte**	**31 days**	**16/11/2018**	**28/12/2018**	
11	3.a Investigação	13 days	16/11/2018	04/12/2018	
12	3.b Análise	5 days	04/12/2018	10/12/2018	
13	3.c Testes complementares	3 days	26/12/2018	28/12/2018	
14	**4. Escrita do Relatório Intermédio**	**31 days**	**11/12/2018**	**22/01/2019**	
15	4.a Capitulo 1	3 days	11/12/2018	13/12/2018	
16	4.b Capitulo 2	3 days	13/12/2018	17/12/2018	
17	4.c Capitulo 3	10 days	17/12/2018	28/12/2018	
18	4.d Capitulo 4	4 days	28/12/2018	02/01/2019	
19	4.e Capitulo 5	6 days	02/01/2019	09/01/2019	
20	4.f Capitulo 6	2 days	09/01/2019	10/01/2019	
21	4.g Capitulo 7	4 days	10/01/2019	15/01/2019	
22	4.h Capitulo 8	0 days	15/01/2019	15/01/2019	
23	4.i Correções	6 days	15/01/2019	22/01/2019	
24	5. Entrega da Meta Intermédia	1 day	23/01/2019	23/01/2019	
25	**6. Preparação da Defesa Intermédia**	**2 days**	**24/01/2019**	**27/01/2019**	
26	6.a Recolha da informação a apresentar	1 day	24/01/2019	24/01/2019	
27	6.b Criação da Apresentação	2 days	24/01/2019	25/01/2019	
28	6.c Estudo da Apresentação	0 days	27/01/2019	27/01/2019	
29	7. Plano de trabalho para o 2º Semestre	0 days	28/01/2019	28/01/2019	

Figure 6.4: Tabela do escalonamento real do primeiro semestre

6.3.2.2 Segundo Semestre

	Task Name	Duration	Start	End	Predecessors
1	**2º Semestre**	**157 days**	**30/01/2019**	**09/07/2019**	
2	**1. Desenvolvimento da Framework em Java**	**129 days**	**30/01/2019**	**11/06/2019**	
3	1.a Criar uma load balancer para um container	6 days	30/01/2019	06/02/2019	
4	1.b Obter endereço de acesso para um container	2 days	06/02/2019	07/02/2019	
5	1.c Lançar um teste local de pequena complexidade	5 days	07/02/2019	13/02/2019	
6	1.d Criar função para obter todos os componentes	3 days	13/02/2019	15/02/2019	
7	1.e Criar função para obter logs de cada container	5 days	15/02/2019	19/02/2019	
8	1.f Lançar testes locais em paralelo	2 days	19/02/2019	20/02/2019	
9	1.g Automação dos processos criados nos pontos 3-8	10 days	20/02/2019	01/03/2019	
10	1.h Realizar escalonamento horizontal	1 day	02/03/2019	02/03/2019	
11	1.i Lançar a STF no cluster de k8s da Feedzai	5 days	05/03/2019	09/03/2019	
12	1.j Criar conversor de ficheiros de docker para kubernetes	7 days	12/03/2019	18/03/2019	
13	1.k Criar função para lançar serviços e deployments automáticamente	4 days	19/03/2019	22/03/2019	
14	**1.l Entrega da STF como prova de conceito**	**7 days**	**22/03/2019**	**28/03/2019**	
15	1.l.i Correcção baseda na review do orientador	7 days	22/03/2019	28/03/2019	
16	**1.m Apresentação da PoC à equipa de qualidade**	**1 day**	**29/03/2019**	**29/03/2019**	
17	1.m.i Revisão das features a criar	2 days	29/03/2019	30/03/2019	
18	1.n Criar método de lançamento dos componentes com hash	21 days	30/03/2019	19/04/2019	
19	1.o Automação do lançamento dos processos no cluster local e externo	15 days	19/04/2019	03/05/2019	
20	1.p Intergração dos componentes internos da Feedzai	19 days	03/05/2019	21/05/2019	
21	1.q Escrita do manual	9 days	21/05/2019	29/05/2019	
22	1.r Testes à stf	14 days	29/05/2019	11/06/2019	
23	**2. Escrita contínua da Tese**	**19 days**	**11/06/2019**	**29/06/2019**	
24	2.a Correção pós defesa intermédia	4 days	11/06/2019	14/06/2019	
25	2.b Correção Final	17 days	11/06/2019	27/06/2019	
26	3. Apresentação final à empresa da SFT2.0	1 day	27/06/2019	27/06/2019	
27	4. Entrega final da tese	1 day	01/07/2019	01/07/2019	
28	5. Criar Apresentação final	7 days	01/07/2019	09/07/2019	

Figure 6.5: Tabela do escalonamento real do segundo semestre

6.3.3 Análise

Como era previsto, existiu uma discrepância entre os dois planos definidos. Esta situação deve-se a que ao longo do desenvolvimento foram dadas novas funcionalidades e também com maior conhecimento dos métodos são feitas adaptações. A correção de bugs, ao longo do desenvolvimento da STF2.0, levou a alguma perda de tempo. Mas o facto que criou maior atraso, foi a adaptação das imagens de docker, que é aprofundado na **secção 7.2.1**, de forma a realizar os testes em paralelo com a *hash* específica.

6.3.3.1 Desvios

ID da Tabela	Tarefa Realizada	Tempo de desvio em relação ao previsto
1.a	Familiarização com o projeto e a empresa	+18
2	Análise de Requisitos	+14
3	Estado da Arte	-13
4	Escrita do Relatório Intermédio	a : 0
		b : -2
		c : +3
		d : +1
		e : +3
		f : -2
		g : +1
		h : 0
		i : +1
6.b	Criação da Apresentação	+1

Table 6.6: Tabela de desvios do primeiro semestre

ID da Tabela	Tarefa Realizada	Tempo de desvio em relação ao previsto
1	Desenvolvimento da Framework em Java	+16
2	Escrita contínua da Tese	-4

Table 6.7: Tabela de desvios do segundo semestre semestre

Como é possível observar grande parte dos desvios devem-se a tarefas que demoraram mais do que o esperado. Esta situação é normal por duas razões, a primeira deve-se ao facto do estudante não estar habituado a programar tarefas num plano de longa duração e a outra é o facto durante um projeto existirem alterações ao longo do desenvolvimento.

Especificamente, no primeiro semestre, pode-se observar que os maiores desvios foram no *onboarding* e na análise de requisitos, que aconteceu porque existiam vários conceitos a apreender da empresa e porque a análise de requisitos deve ser feita de forma consistente pois influência todo o projeto, desde planeamento a desenvolvimento.

Capítulo 7

Desenvolvimento da Solução

Este capítulo está divido em duas secções, a primeira descreve o desenvolvimento do produto desenvolvido e a segundo é resumir os principais desafios ao longo da implementação.

7.1 Trabalho Desenvolvido

O trabalho realizado ao longo deste estágio académico resultou na produção e documentação da STF2.0, a versão 2.0, sendo que é uma tentativa da anteriormente implementada na Feedzai. Nesta versão foram definidas e implementadas funcionalidades de gestão, lançamento de ambientes em K8S a partir de qualquer ficheiro de *docker-compose* ou ficheiros de configuração de K8S e execução de testes em paralelo e de forma escalável.

Toda a *framework* foi desenhada no sentido se ser leve, independente de qualquer modulo interno de software da empresa (*Open-Source*) e de forma a correr através de um terminal sem uso de uma UI específica. Assim que o teste é executado são apresentados os links para aceder ao componente lançado e ao mesmo tempo pode-se aceder à interface do *cluster* para aceder a *live logs*.

No final existe a possibilidade de manter todo o ambiente, desde serviços a *pods*, ou limpar todos os recursos.

7.1.1 Escolha dos elementos a usar

Tendo em conta a linguagem de programação a escolha recaiu sobre a escrever todo o código da STF2.0 em Java, pois é linguagem de eleição e na qual o autor tem maior experiência e ao mesmo tempo é a linguagem mais usada pela empresa.

O uso de K8S foi justificada na secção 3.1. Por outro lado, inicialmente o input da *framework* seria um ficheiro de K8S mas de forma a ser mais fácil a adaptação dos utilizadores foi adicionada a opção para *docker-compose*.

De forma a ter uma melhor perspetiva, vou apresentar imagens das interfaces dos *clusters* e excertos da aplicação, usando o ficheiro petclinic e os seus outputs. O petclinic é uma imagem de *docker* que contém uma aplicação simples de uma clínica veterinária com uma dependência de uma base de dados MySql. No apêndice A encontram-se os exemplos dos ficheiros de *docker-compose e kompose* usados na STF2.0.

Na figura seguinte é possível observar que o processo foi iniciado e que o ambiente está a

ser criado. Sendo que este teste está a ser executado no *cluster* externo do l*abs da Feedzai* é mudado o contexto para que se possa aceder ao mesmo.

Para aceder a qualquer *cluster* privado é necessário criar um ficheiro *secret* com todas as configurações e *tokens* do mesmo. Neste caso temos um *warning* pois a chave já tinha sido previamente adicionada. De reter também que os dois serviços e *deployments* são lançados e que é apresentada a informação que os serviços estão pendentes até que a situação se altere e os *pods* estejam a ser executados na sua plenitude. Posteriormente, são apresentados os links de acesso a cada serviço, que podem ser acedidos nos testes.

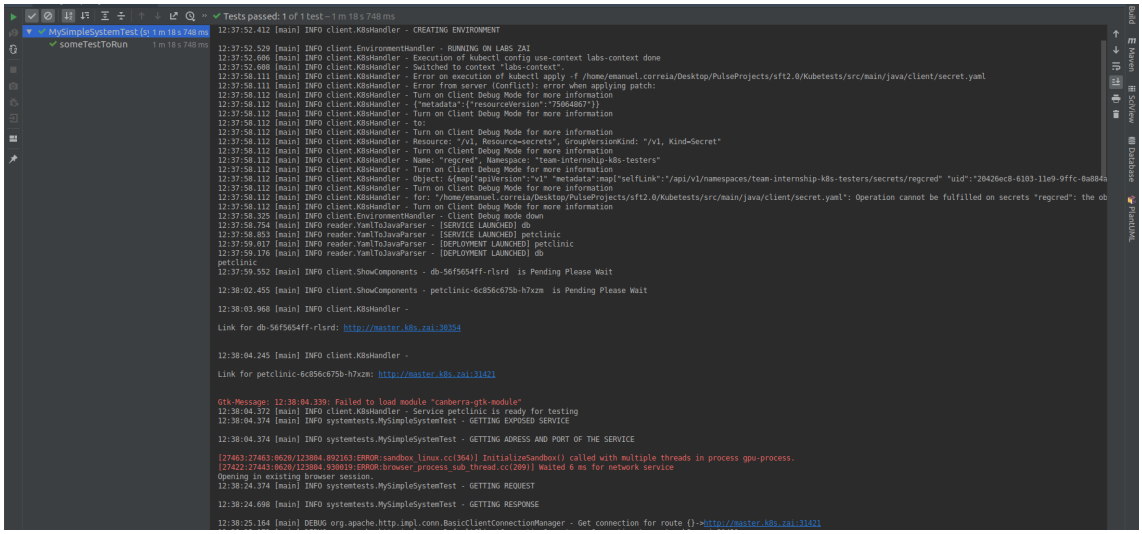

Figure 7.1: Excerto do output do teste ao *container* do petclinic

Nesta figura, é apresentado o resultado do teste que é demonstrar a lista dos veterinários. Ao mesmo tempo é demonstrado o link no qual o teste foi executado e os passos da limpeza do ambiente.

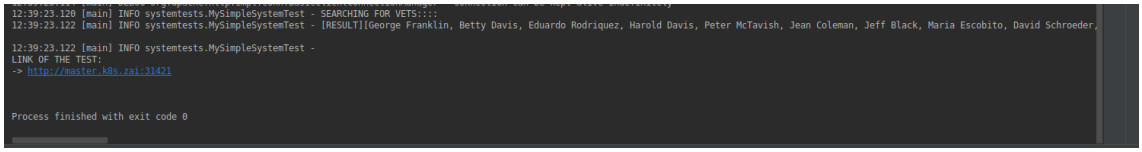

Figure 7.2: Excerto do output do teste ao *container* do petclinic

Após o ambiente ser lançado é possível aceder, caso não se elimine os componentes do teste como no exemplo de output anterior, e conseguir observar a interface do *pod*, neste caso do petclinic. Na primeira figura é apresentado a interface quando se abre o link e na segunda a interface com a listagem de alguns veterinários que são usados na pesquisa.

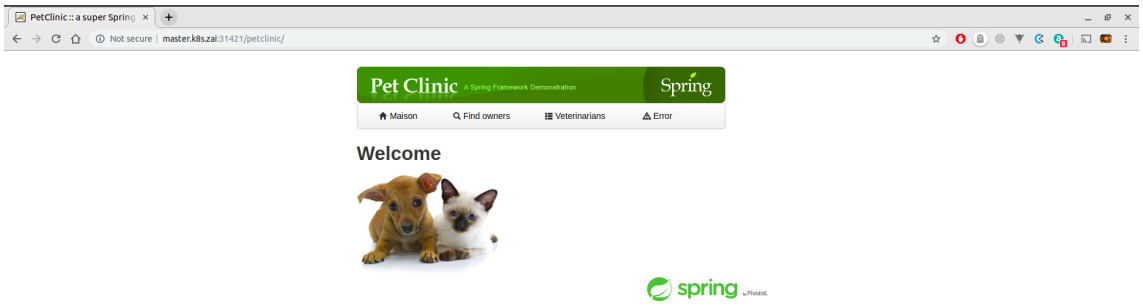

Figure 7.3: Exemplo da interface do *home* do *container* do petclinic

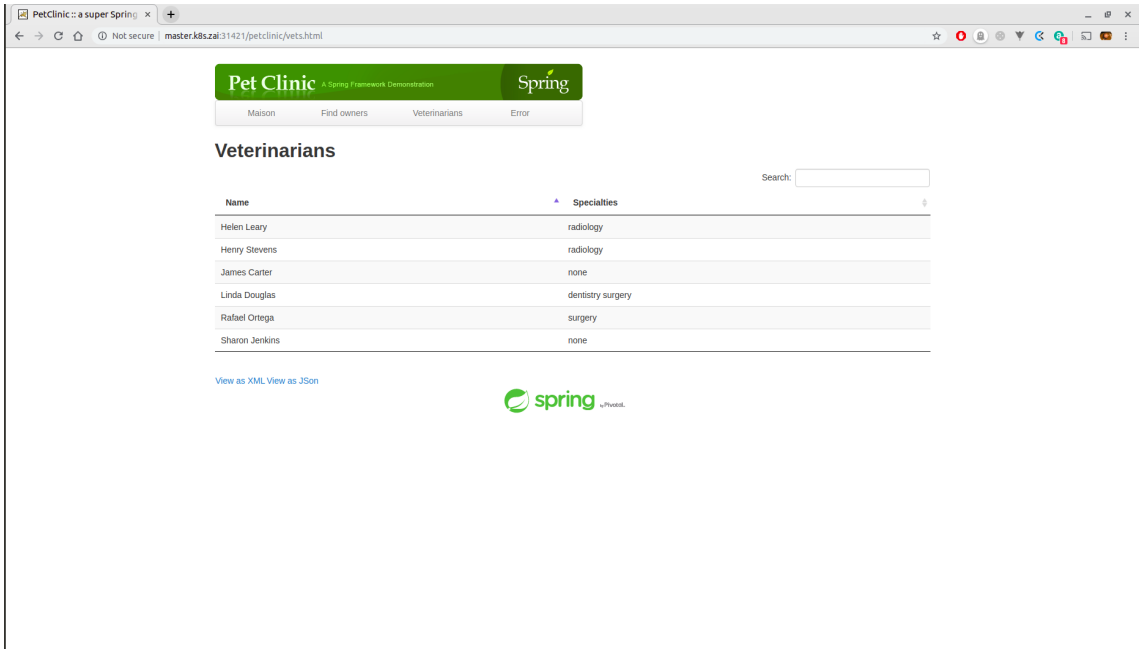

Figure 7.4: Exemplo da interface da lista de veterinários do *container* do petclinic

Para finalizar esta secção, nas figuras seguintes será apresentada a interface do *labs.feedzai.com*. De referir que também pode ser usado um cluster local, como minikube, com uma interface bastante semelhante à do *cluster* externo, segundo escolha do utilizador ao criar o teste.

Esta parte serve mais para comprovação de que realmente o resultado esperado foi obtido.

Na **figura 7.5** e **7.6** é possível comprovar que tanto os serviços como os *pods* foram lançados com sucesso e que não sofrer erros que os levaria a ficarem indisponíveis. Como se pode observar que é criado um serviço para cada *pod* e ao mesmo tempo a STF2.0 cria,

igualmente, um *load balancer* para cada um dos mesmos serviços. Estes *load balancers* servem para criar os links de acesso aos *pods* demonstrados nos output da **figura 7.1**.

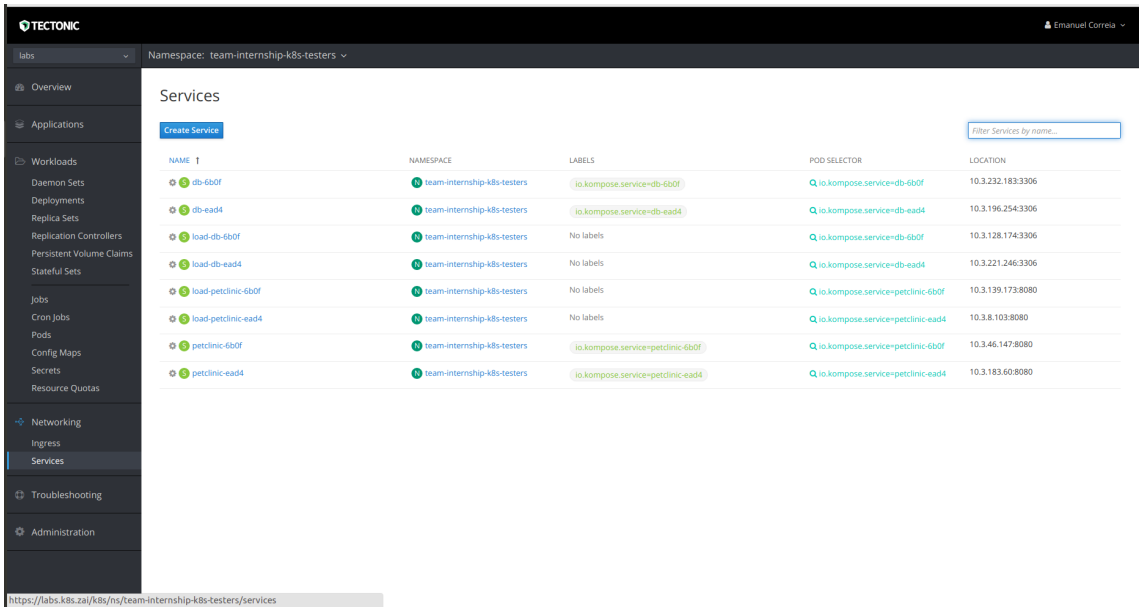

Figure 7.5: Lista de serviços no *cluster* externo

Na **figura 7.6** é demonstrado que os *pods* são lançados com uma *hash*, esta solução irá ser aprofundada na **secção 7.2**. E na imagem **7.3** é demonstrada a página de cada *pod*.

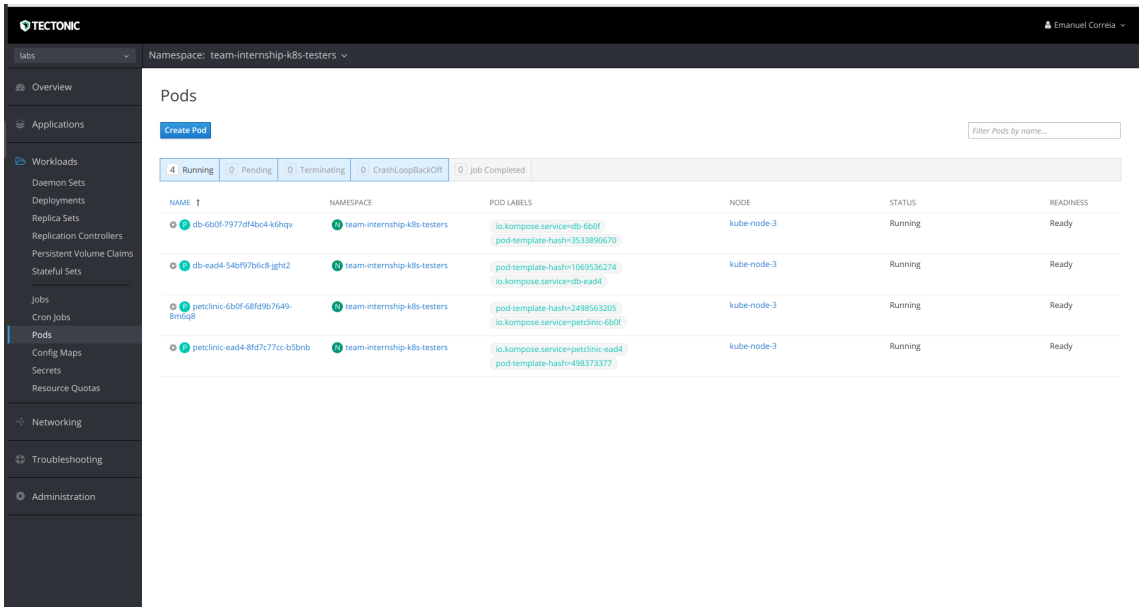

Figure 7.6: Lista de *pods* no *cluster* externo

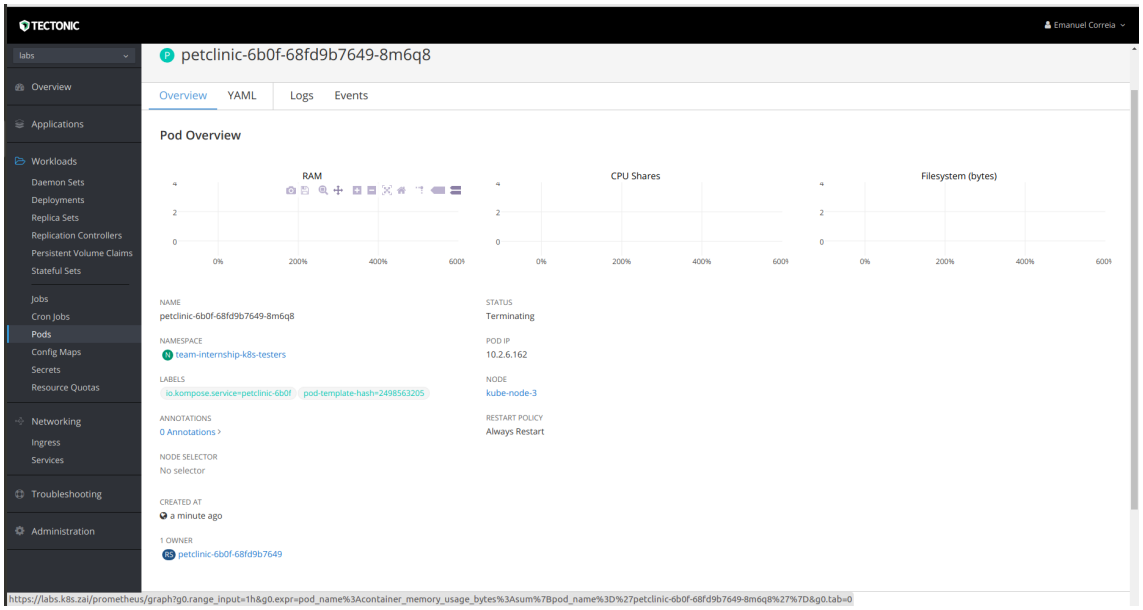

Figure 7.7: Página exemplo para um *pod*

Como é possível verificar na **figura 7.8**, existe um *secret* com o nome "regcred", que foi exclusivamente criado para que seja permitido o acesso ao utilizador com estas credenciais.

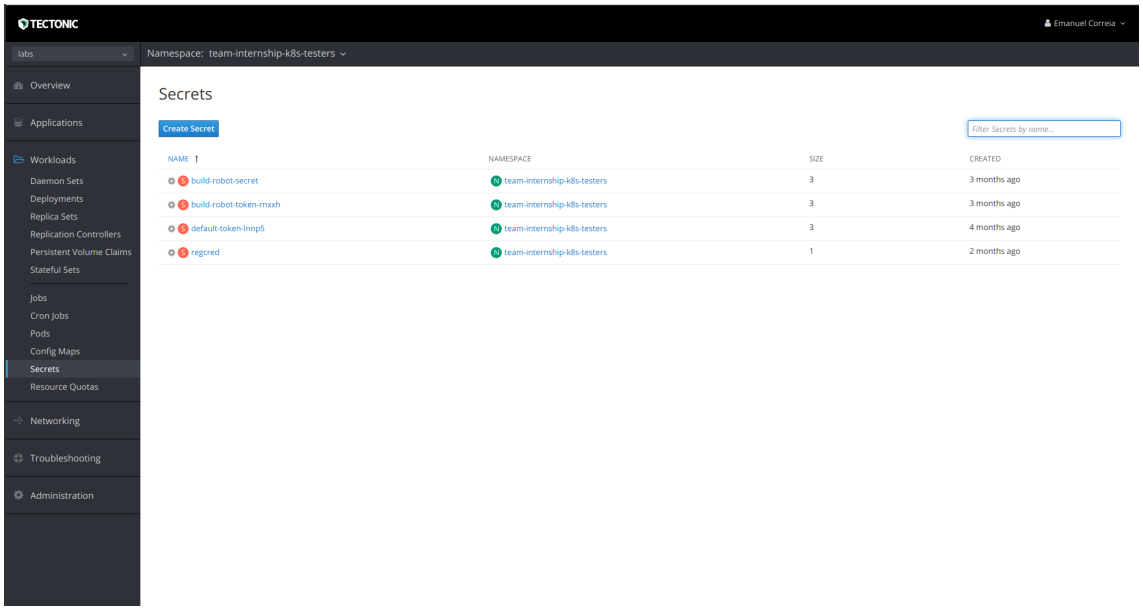

Figure 7.8: Lista de *namespaces* no *cluster* externo

Assim sendo concluímos que a STF2.0 consegue lançar *containers* em paralelo, a partir de um ficheiro de *docker-compose* ou de K8S, e que permite o acesso, recolha de *logs* e gestão de todos os componentes.

De denotar que se o utilizador não quiser ler aos ficheiros de *logs* criados no fim da execução pode usar a interface do *cluster* para os conseguir analisar em direto.

7.2 Desafios Encontrados

Ao longo do estágio foram aparecendo certos desafios e os quais obrigaram a tomada de decisões. Nesta secção é ilustrado cada problema, a solução e o porquê da tomada de decisão.

7.2.1 Testes em Paralelo

De forma isolada, este foi claramente o maior problema ao longo da implementação. Inicialmente a solução foi criada usando somente um *pod* de cada serviço como um exemplo simples, para prova de conceito. Posteriormente é que existiu a tentativa de executar em paralelo.

Ao executar a STF2.0 verificamos que existia sempre um erro pois os *pods* e os serviços tinham nomes repetidos. A solução encontrada foi criar uma ***hash*** específica para cada teste, que trouxe outro desafio, as imagens não estão preparadas para serem executadas em paralelo. Neste caso, as imagens foram alteradas de forma a receber uma variável de ambiente que contivesse a *hash* e altera-se os ficheiros de configuração necessários.

Para executar os testes desta forma é preciso que sejam feitas alterações à imagem. Todas as imagens do teste precisam de contactar entre si, obrigando assim a alterar os *endpoints* de contacto, e não podem ter o mesmo nome pois o K8S não aceita. Por exemplo é necessário mudar as *connection.strings* das bases de dados de forma a conter a *hash*, esta situação deve ser aplicada pelo próprio *container* quando está a ser executado.

7.2.2 Estado dos pods

Sendo que por vezes os *pods* levam um certo tempo a estar em conformidade com o seu *health-check*, condição que confirma que o *pod* está a funcionar devidamente, só se deve criar o *load balancer* após essa confirmação.

No caso dos pods, foi bastante simples, mas já no caso dos serviços foi bastante diferente. No caso dos serviços, verificou-se que ao obter o estado o tipo nem sempre era o mesmo e criava um erro. Foi demoroso até encontrar o que criava esta situação era um erro da API de K8S [6], mas assim que se chegou ao problema a solução foi simples. O problema acontecia quando se tentava apagar os *pods* após realizar o teste, sendo que inicialmente retornava um objeto em vez de retornar um estado, que confirmava que a instrução tinha tido sucesso. Como solução foi necessário tratar este tipo de exceção, como um caso especial, onde foi necessário esperar até que o retorno fosse o esperado.

7.2.3 Secrets

Ao tentar aceder ao *cluster* externo, através da STF2.0, foi encontrado um erro de acesso por falta de credenciais, que faz todo o sentido. Para esta situação foi necessário pesquisar a solução e essa foi criar um ficheiro *secret* e enviá-lo para o *cluster* antes de qualquer operação. Se este for aceite a *framework* é executada caso contrário para e apresenta o erro.

7.2.4 Quota

Outra e última situação que precise de referência foi a questão da quota de recursos para cada *cluster*. Cada um destes, tem um quota que os componentes podem usar e que não podem de forma nenhuma ultrapassar ou não são lançados. Neste caso, temos a solução que passa por aumentar esta quota podendo ter mais recursos ou senão no ficheiro de K8S ou *docker-compose*, é possível definir os recursos a usar por cada serviço tendo sempre em conta os limites do *cluster*.

Capítulo 8

Validação

8.1 Plano de validação

De forma a verificar se o sistema está a funcionar da maneira correta, é necessário testar todos os seus recursos. Portanto, tendo em mente os requisitos previamente identificados é apresentado um plano de teste. **Os ficheiros usados para os testes** encontram-se no **Apêndice A.1**. De forma a ser mais percetível os **relatórios extensos dos testes** encontram-se no **Apêndice A.2**.

8.1.1 Testes aos Requisitos Funcionais

Para cada requisito funcional definido, identificamos um conjunto de testes a serem realizados para verificar a validade do sistema.

Sendo que o ideal seria apresentar um caso de teste para cada requisito, na totalidade de todos os seus componentes, mas de forma a não ser repetitivo, não são testadas todas as combinações possíveis. Assim sendo os **TF2 e TF3** terão exemplos mais generalizados.

Muito dos testes são baseados em alterações ao ficheiro de docker, que é a base para o ambiente. Os testes **TF17, TF18, TF19 e TF20** não são executados pois as *features* não foram implementadas diretamente na *framework* mas que estão disponíveis nos *clusters*.

Do **TU1 a TS2** não foram documentados pois são situações que estão incluídas em testes anteriores.

Dada esta exposição, listamos os testes considerados por nós:

[**TF1.**] Teste para o requisito funcional "Suportar a integração da solução com CI e CD"

- Verificar que a STF2.0 corre e lança o ambiente, com sucesso, através da linha de comandos e sem que seja necessário mais nada;
- Verificar que a STF2.0 é capaz de se integrar com o *Git*;requisito

[**TF2.**] Teste para o requisito funcional "Suportar a integração de componentes externos"

- Verificar que a STF2.0 executa o teste de sistema, com sucesso, através ficheiros de Docker, um serviço de Cassandra;

- Verificar que a STF2.0 executa o teste de sistema, com sucesso, através ficheiros de Docker, um serviço de Postgres;

[**TF3.**] Teste para o requisito funcional "Suportar a integração de componentes internos".

- Verificar que a STF2.0 executa o teste de sistema, com sucesso, através ficheiros de Docker, um serviço de Pulse;

[**TF4.**] Teste para o requisito funcional "Suportar integração com os *containers* do Chrome e Firefox"

- Verificar que a STF2.0 executa, com sucesso, através ficheiros de Docker, um *container* de Chrome e de Firefox;
- Ver que é possível aceder a cada um dos *containers*, através do endereço dado pela STF2.0;

[**TF5.**] Teste para o requisito funcional "Suportar o *deployment* de ambientes com diferentes configurações"

- Verificar que a STF2.0 executa o teste de sistema, com sucesso, através ficheiros de Docker com mais que uma imagem;
- Verificar que a STF2.0 executa o teste de sistema, com sucesso, através ficheiros de Docker com uma rede interna definida;
- Verificar que a STF2.0 executa o teste de sistema, com sucesso, através ficheiros de Docker com definição de variáveis de ambiente;
- Verificar que a STF2.0 executa o teste de sistema, com sucesso, em que é definido um *namespace* específico;
- Verificar que a STF2.0 executa o teste de sistema, com sucesso, em que é definido um ficheiro de configuração do cliente específico;
- Verificar que a STF2.0 executa o teste de sistema, com sucesso, em que é definida uma diretoria específica de ficheiros de docker ou K8S;
- Verificar que a STF2.0 executa o teste de sistema, com sucesso, em que é definido o url de um *cluster* específico.
- Verificar que a STF2.0 executa o teste de sistema, com sucesso, em que é definido o nome do serviço a testar.

[**TF6.**] Teste para o requisito funcional "Suportar a interação e gestão dos *containers* em tempo de execução"

- Verificar que após o ambiente ser lançado pela STF2.0, é possível aceder ao *container* pelo endereço especificado pela mesma.

[**TF7.**] Teste para o requisito funcional "Suportar framework de testes como *JUnit*"

- Verificar que a STF2.0 é executada através de um teste de sistema, usando *JUnit*.

[**TF8.**] Teste para o requisito funcional "Suportar conexão de bibliotecas externas usadas em *System Tests*"

- Verificar que é possível aceder a um serviço de HTTP;

[**TF9.**] Teste para o requisito funcional "Ter suporte para *Logs*"

- Definir o valor do **PrintLogs** a **true**;
- Verificar que após lançar o ambiente, que é possível ver os *logs* dos *containers*, no terminal, durante a execução do teste;

[**TF10.**] Teste para o requisito funcional "Ter um modo *debug*"

- Definir o valor da **DebugFlag** a **true**;
- Verificar que após lançar o ambiente, que é possível ver todos os passos a ser realizados na execução dos testes;
- Verificar que após lançar o ambiente, é possível aceder aos *containers* através dos endereços dados pela STF2.0.

[**TF11.**] Teste para o requisito funcional "Suportar mecanismos de recuperação de falhas"

- Verificar que após lançar o ambiente e de correr um teste de sistema com erros, que é possível aceder aos *containers* através dos endereços dados pela STF2.0 e que é possível aceder aos *logs*.

[**TF12.**] Teste para o requisito funcional "Suportar *docker-compose*"

- Verificar que se consegue lançar, com sucesso, a STF2.0 usando um ficheiro de *docker-compose*.

[**TF13.**] Teste para o requisito funcional "Suportar ficheiros de configuração de K8S"

- Verificar que se consegue lançar, com sucesso, a STF2.0 usando um ficheiros de configuração de K8S.

[**TF14.**] Teste para o requisito funcional "Suportar *clusters* locais de K8S"

- Verificar que se consegue lançar, com sucesso, a STF2.0, neste caso, no ambiente de *minikube*

[**TF15.**] Teste para o requisito funcional "Suportar *clusters* externos de K8S"

- Verificar que se consegue lançar, com sucesso, a STF2.0, neste caso, no ambiente *labs da Feedzai*.

[**TF16.**] Teste para o requisito funcional "Suportar apresentação serviços ou instâncias"

- Verificar que após lançar o ambiente, com sucesso, a STF2.0 consegue mostrar todos os serviços e instâncias lançadas no terminal.

[**TF17.**] Teste para o requisito funcional "Suportar filtro de procura de serviços ou instâncias segundo parâmetro"

- Verificar que após lançar o ambiente, com sucesso, a STF2.0 consegue mostrar todos os serviços e instâncias lançadas no terminal, segundo um nome, estado, tipo ou hash.

[**TF18.**] Teste para o requisito funcional "Suportar filtro de serviços ou instâncias por categoria"

- Verificar que após lançar o ambiente, com sucesso, a STF2.0 consegue mostrar todos os serviços lançados no terminal;
- Verificar que após lançar o ambiente, com sucesso, a STF2.0 consegue mostrar todos os *deployments* lançadas no terminal;
- Verificar que após lançar o ambiente, com sucesso, a STF2.0 consegue mostrar todos os *pods* lançados no terminal;

[**TF19.**] Teste para o requisito funcional "Suportar filtro de serviços ou instâncias por nome"

- Verificar que após lançar o ambiente, com sucesso, a STF2.0 consegue mostrar todos os componentes, que contêm um nome ou uma hash, lançados no terminal;

[**TF20.**] Teste para o requisito funcional "Suportar filtro de serviços ou instâncias por estado"

- Verificar que após lançar o ambiente, com sucesso, a STF2.0 consegue mostrar todos os componentes, que se encontram num estado específico, lançados no terminal;

[**TF21.**] Teste para o requisito funcional "Suportar exportação os ficheiros de .log"

- Verificar que ao lançar o ambiente, todos os *logs* dos componentes, durante a execução do teste, são guardados num ficheiro de .log.

8.1.2 Testes aos requisitos Não Funcionais

[**TE1.**] Teste para o requisito não funcional "Suportar testes em paralelo"

- Verificar que ao lançar o ambiente em paralelo, que todos os *containers* se encontram no estado "Running", de forma estável,ou seja, sem recorrer a relançamentos.

[**TP1.**] Teste para o requisito não funcional "Suportar execução para sistemas *MacOs e Unix*"

- Verificar que ao lançar o ambiente, garantir que a ferramenta funciona de igual forma em *MacOS ou Unix*, e é possível correr testes utilizando-a em ambos.

[**TD1.**] Teste para o requisito não funcional "Desenvolvida em Java"

- Verificar que a STF2.0, foi programada em JAVA.

[**TU1.**] Teste para o requisito não funcional "Apresentar os erros de forma destacada"

- Verificar que após lançar o ambiente e o teste ao dar erro, apresenta de forma clara e destacada que existiu um erro na execução.

[**TU2.**] Teste para o requisito não funcional "Fornecer uma forma de ajuda"

- Verificar que o README contempla a explicação dos inputs;
- Verificar que as mensagens de erro contemplam soluções;

[**TU3.**] Teste para o requisito não funcional "Limitar hipóteses de input"

- Verificar que existem pré-condições para os valores de inputs;

[**TU4.**] Teste para o requisito não funcional "Diminuir ambiguidade"

- Perguntar e ter *feedback* das pessoas se entendem os comentários do sistema e mensagens do mesmo e comparar com o pretendido.

[**TS2.**] Teste para o requisito não funcional "Documentar todos os requisitos funcionais"

- Verificar que todos os requisitos funcionais estão bem documentados, apresentando a um elemento que não tenha conhecimento do projeto e vendo se percebe tudo de forma clara e objetiva.

8.2 Avaliação

Os testes anteriormente descritos foram executados por um utilizador, de forma a saber se a STF2.0 realiza as funcionalidades para o qual foi concebida.

Os Resultados são apresentados na tabela 8.1

Table 8.1: Tabela de testes

Teste	Prioridade	Resultado
TF1	Must	V
TF2	Must	V
TF3	Must	V
TF4	Must	V
TF5	Must	V
TF6	Must	V
TF7	Must	V
TF8	Must	V
TF9	Must	V
TF10	Must	V
TF11	Should	V
TF12	Must	V
TF13	Must	V
TF14	Must	V
TF15	Must	V
TF16	Could	V
TF17	Could	X
TF18	Could	X
TF19	Could	X
TF20	Could	X
TF21	Could	V

Table 8.2: Tabela de testes

Teste	Prioridade	Resultado
TE1	Must	V
TP1	Must	V
TD1	Should	V
TU1	Should	V
TU2	Should	V
TU3	Should	V
TU4	Should	V
TS2	Could	V

Table 8.3: Tabela de testes

V - Passou
X - Falhou

Como é possível verificar todos os requisitos de prioridade **Must** foram cumpridos sendo que podemos concluir que o principal objetivo do estágio, definido no primeiro semestre, foi concluído. De observar que alguns **Could** não foram implementados por necessidade de reduzir a *scope* tendo em conta o tempo disponível, com o acordo da Feedzai. Mesmo assim tendo em conta os teste apresenta um saldo bastante positivo, de 93,75% dos requisitos pedidos foram realizados, sendo que os de prioridade:

- **Must** têm um peso de 3 pontos;
- **Should** têm um peso de 2;
- **Could** têm um peso de 1;

Capítulo 9

Conclusão e Trabalho Futuro

Este capítulo está dividido em duas secções, a primeira apresenta o futuro do projeto e a segunda apresenta a reflexão e análise do trabalho realizado no estágio.

9.1 Trabalho Futuro

A STF2.0, do ponto de vista da Feedzai e do estudante, é tida como uma solução para continuar de desenvolver e com valor real.

Após a apresentação interna final, foram definidos objetivos delineadores de novas *features* para o projeto, que passam por:

- Um dos problemas, pós-implementação, é ter de definir manualmente o CPU e memória em cada componente, no *docker-compose*. Sendo que não existem certezas que o valor é o mais eficiente, é necessário encontrar uma solução para que a definição destes valores passe a ser automática;

- Preparar uma release oficial, da Feedzai, para uma plataforma Open-Source, de forma a distribuir a solução;

- Garantir que a STF2.0 permite ligações por *RMI* aos endpoints dos testes, tendo em conta que a empresa depende muito de testes de *Messaging* , processos assíncronos e grande parte da *codebase* de testes usa chamadas *RMI*.

- Conseguir passar ficheiros a usar nos testes para dentro do *cluster*, havendo a possibilidade de usar volumes;

- Neste momento, se não existirem recursos livres no cluster para os componentes, o teste não é lançado. Assim sendo é necessário criar uma forma de escalonamento de tarefas e conseguir criar uma forma de os testes que não são executados por falta de recursos fiquem em espera numa *queue*;

- Conseguir enviar os *logs* dos componentes para uma plataforma de *logging*, por exemplo *Elastic Search*;

9.2 Conclusão

Findado o período definido para a meta final do estágio académico, e é a fase de fazer uma revisão do trabalho realizado. De referir que na fase de *Onboarding* existiu uma grande preocupação que esta fosse simples e rápida por parte de todos os elementos da empresa.

Inicialmente, é de referir que esta é a primeira experiência na área de desenvolvimento de software ao nível empresarial. Como era de esperar, foi possível ver as diferenças em relação ao ambiente académico, desde a forma de estruturar os projetos à dimensão dos mesmos, e que é necessário um maior nível de responsabilidade e compromisso.

Como esperado no primeiro semestre, o tempo investido passou maioritariamente com a familiarização das ferramentas a usar, com a especificação dos requisitos necessários para o produto e a análise das ferramentas que existem no mercado.

No segundo semestre, foi possível desenvolver a STF2.0, realizando 94% dos requisitos, sendo que todos os de prioridade *Must* foram realizados. Ao mesmo tempo compreendi que não é fácil planear um projeto a longo prazo e que existem variadas previsões que não são tão lineares como pensado inicialmente.

O balanço global para mim foi bastante positivo sendo que foi um projeto com algum grau de exigência e com elementos com que nunca tinha trabalhado, K8S e docker. Além desta situação, acho que o ambiente de trabalho e as condições foram excelentes para contribuir para o resultado final.

Referências

[1] Black box testing vs white box testing: Know the differences. https://phoenixnap.com/blog/white-box-vs-black-box-testing.

[2] Clockify - 100% free time tracking software. https://clockify.me/.

[3] Devops: Continuous integration vs continuous deployment. https://www.nastel.com/blog/.

[4] Docker containers and kubernetes: An architectural perspective. https://dzone.com/articles/docker-containers-and-kubernetes-an-architectural.

[5] Feedzai stops financial fraud with machine learning. https://www.forbes.com/sites/tomgroenfeldt/2016/02/16/feedzai-stops-financial-fraud-with-machine-learning/#309b37c6599d.

[6] Github: Getting exception from deletenamespacedstatefulset() call. https://github.com/kubernetes-client/java/issues/86, [Accessed 28 June 2019].

[7] How to use personal kanban to visualize your work. https://lifehacker.com/productivity-101-how-to-use-personal-kanban-to-visuali-1687948640.

[8] Introduction to kubernetes architecture. https://x-team.com/blog/introduction-kubernetes-architecture/.

[9] Java client for kubernetes openshift 3. https://github.com/fabric8io/kubernetes-client.

[10] Kubernetes alternatives: A look at swarm, marathon, nomad, kontena. http://techgenix.com/kubernetes-alternatives/.

[11] Kubernetes concepts. https://kubernetes.io/docs/concepts/.

[12] Kubernetes support revisited - issue 1135 - testcontainers/testcontainers-java - github. https://github.com/testcontainers/testcontainers-java/issues/1135, [Accessed 27 June 2019].

[13] Moscow method. https://en.wikipedia.org/wiki/MoSCoW_method.

[14] Platform9 kubernetes ebook. https://platform9.com/wp-content/uploads/2018/08/kubernetes-comparison-ebook.pdf.

[15] Production-grade container orchestration. https://kubernetes.io/.

[16] Regression testing. https://www.guru99.com/regression-testing.html.

[17] Suportabilidade. https://pt.wikipedia.org/wiki/Suportabilidade.

[18] Testcontainers. `https://www.testcontainers.org/`.

[19] Types of functional testing. `https://www.proprofs.com/quiz-school/story.php?title=test-design3`.

[20] Usabilidade. `https://pt.wikipedia.org/wiki/Usabilidade`.

[21] P. Duvall. Improving software quality and reducing risk. In *Continuous integration*, Upper Saddle River, NJ, USA, 2013. Addison-Wesley.

[22] S. Ian. Second edition. In *Software engineering*, Boston, USA, 2016. Pearson.

[23] Ricardo Lopes. System test framework 2.0. `https://docs.feedzai.com/display/~ricardo.lopes/System+Test+Framework+2.0`.

[24] M. Myers. Second edition. In *Art Of Software Testing*, Hoboken, NJ, 2004. Wiley, John and Sons, Incorporated.

[25] Hind Naser. Kubernetes vs mesos: A comparison of containerization platforms. `https://vexxhost.com/blog/kubernetes-mesos-comparison-containerization/`.

[26] P. Swartout. A quickstart guide. In *Continuous Delivery and DevOps*, Birmingham, UK, 2012. Packt Publishing.

This page is intentionally left blank.

Apêndice A

Ficheiros de configuração e Relatório de Testes

Este capítulo apresenta duas secções. A primeira contém os vários ficheiros de configuração usados para os testes. A segunda apresenta os relatórios pormenorizados dos testes.

A.1 Ficheiros de configuração

Nesta secção são apresentados os ficheiros de configuração de *docker-compose* e K8S. E os ficheiros que contêm os testes.

A.1.1 Ficheiro de docker-compose do projeto Pulse e *Postgres*

```
version: '3'

services:
  pulse:
    depends_on:
      - db
    image: docker.feedzai.com/zaickathon/kubertests:pulse_v1.0
    ports:
      - "18080:8080"
    deploy:
      resources:
        limits:
          cpus: '2'
          memory: 6G
        reservations:
          cpus: '1'
          memory: 4G
    networks:
      - kubetests
  db:
    image: postgres:9.4
    expose:
```

```
      - "5432"
    ports:
      - 5432:5432

    healthcheck:
      test: ["CMD-SHELL", "pg_isready","-U","postgres"]
      interval: 10s
      timeout: 5s
      retries: 5
    networks:
      - kubetests
    deploy:
      resources:
        limits:
          cpus: '1'
          memory: 2G
        reservations:
          cpus: '0.5'
          memory: 1G

networks:
  kubetests:
    driver:
      bridge
```

A.1.2 Ficheiro de docker-compose do projeto Pulse e *Cassandra*

```
version : "3"
services:
  pulse:
    image: docker.feedzai.com/zaickathon/kubertests:pulse_cassandra_v1.27
    ports:
      - 18080:8080
    depends_on:
      - cassandra
    links:
      - cassandra
    networks:
      - mynetwork1
    environment:
      TERM: "xterm"
    deploy:
      resources:
        limits:
          memory: '6G'
        reservations:
          memory: '5G'

  cassandra:
    image: cassandra:2.1.12
```

```
    expose:
      - "9042"
    restart: always
    ports:
      - "9042:9042"
    environment:
      CASSANDRA_LISTEN_ADDRESS: "localhost"
      CASSANDRA_ENDPOINT_SNITCH: "GossipingPropertyFileSnitch"
      CASSANDRA_DC: "dc1"
      MAX_HEAP_SIZE: "1G"
      HEAP_NEWSIZE: "256M"
    networks:
      - mynetwork1
networks:
  mynetwork1:
    driver: bridge
```

A.1.3 Ficheiro de teste para o Pulse

```
package systemtests;

import client.K8sHandler;
import client.TestConfig;
import org.junit.Test;

/**
 * Unit Test for simple App.
 *
 * @author Emanuel Pedro (emanuel.correia@feedzai.com)
 */
public class PulseT {

    /**
     * Class with the settings of the test.
     */
    private TestConfig testConfig = new TestConfig.Builder()
            .withNamespace("team-internship-k8s-testers")
            .withClientFile("/home/emanuel.correia/Desktop/PulseProjects/
            sft2.0/Kubetests/src/main/java/client/config.yaml")
            .isDockerComposeSetter(true)
            .withConfigFileDir("/home/emanuel.correia/Desktop/
            PulseProjects/sft2.0/Kubetests")
            .withK8Surl("https://master.k8s.zai:443")
            .isLocalFlagSetter(true)
            .withServiceToTest("chrome")
            .isCleanerFlagSetter(false)
            .isDebugFlagSetter(false)
            .build();


    /**
     * Handler to launch and acess to the tests.
     */
    private K8sHandler k8sHandler = new K8sHandler(testConfig);

    /**
     * Tester.
     */
```

```
40      @Test
41      public void someTestToRun() {
42          final K8sHandler.ExposedService exposed = new K8sHandler
43          .ExposedService().findService(k8sHandler.getHashedService());
44          final String pulseAddress = exposed.getIp();
45          final String pulsePort = exposed.getPort();
46          assert pulseAddress != null;
47          assert pulsePort != null;
48          k8sHandler.logging(k8sHandler.getHashedService(),
49          testConfig.getNamespace(), testConfig);
50          k8sHandler.namespaceCleaner(testConfig);
51      }
52  }
```

A.1.4 Ficheiro de docker-compose do projeto Petclinic

```
1   version : "3"
2   services:
3     petclinic:
4       image: anthonydahanne/spring-petclinic
5       ports:
6         - 8080:8080
7       networks:
8         - spring-petclinic
9       depends_on:
10        - db
11      deploy:
12        resources:
13          limits:
14            cpus: '1'
15            memory: '4G'
16          reservations:
17            cpus: '0.5'
18            memory: '3G'
19      healthcheck:
20        test: ["CMD", "sh", "-c", "curl -f http://localhost:8080/petclinic"]
21        interval: 30s
22        timeout: 10s
23        retries: 10
24    db:
25      image: mysql:5.7
26      ports:
27        - 3306:3306
28      environment:
29        MYSQL_ROOT_PASSWORD: petclinic
30        MYSQL_DATABASE: petclinic
31      healthcheck:
32        test: ["CMD", "mysqladmin" ,"ping", "-h", "localhost"]
33        timeout: 20s
34        retries: 10
35  networks:
36    spring-petclinic:
```

A.1.5 Ficheiro de teste do projeto Petclinic

```
package systemtests;

        import client.K8sHandler;
        import client.TestConfig;
        import com.google.common.collect.ImmutableList;
        import io.restassured.builder.RequestSpecBuilder;
        import io.restassured.specification.RequestSpecification;
        import org.junit.Assert;
        import org.junit.Test;
        import org.slf4j.Logger;
        import org.slf4j.LoggerFactory;

        import static io.restassured.RestAssured.given;

/**
 * Unit Test for simple App.
 *
 * @author Ricardo Lopes (ricardo.lopes@feedzai.com)
 * @author Emanuel Pedro (emanuel.correia@feedzai.com)
 */
public class MySimpleSystemTest {

    /**
     * Logger to print info on screen during tests.
     */
    private static final Logger logger = LoggerFactory.getLogger(
    MySimpleSystemTest.class);


    /**
     * Class with the settings of the test.
     */
    private TestConfig testConfig = new TestConfig.Builder()
            .withNamespace("team-internship-k8s-testers")
            .withClientFile("/home/emanuel.correia/Desktop/PulseProjects/
    sft2.0/Kubetests/src/main/java/client/config.yaml")
            //.isDockerComposeSetter(false)
            .isDockerComposeSetter(true)
            .withConfigFileDir("/home/emanuel.correia/Desktop/PulseProjects/
    sft2.0/Kubetests/")
            .withK8Surl("https://master.k8s.zai:443")
            //.withServiceToTest("pulse")
            .isLocalFlagSetter(false)
            .withServiceToTest("petclinic")
            .isCleanerFlagSetter(false)
            .isDebugFlagSetter(false)
            .build();


    /**
     * Handler to launch and acess to the tests.
     */
    private K8sHandler k8sHandler = new K8sHandler(testConfig);

    /**
     * Tester.
     */
    @Test
    public void someTestToRun() {
        logger.info("GETTING EXPOSED SERVICE\n");
        final K8sHandler.ExposedService exposed = new K8sHandler.
```

```
    ExposedService().findService(k8sHandler.getHashedService());
59          final String petClinicAddress = exposed.getIp();
60          final String petClinicPort = exposed.getPort();
61          logger.info("CHECKING ADRESS\n");
62          exposed.checkUrl(String.format("http://%s:%s", petClinicAddress,
    petClinicPort));
63          logger.info("ADRESS [OK]\n");
64
65          logger.info("GETTING REQUEST\n");
66
67          final RequestSpecification spec = new RequestSpecBuilder()
68                  .setBaseUri(String.format("http://%s", petClinicAddress))
69                  .setPort(Integer.parseInt(petClinicPort))
70                  .setBasePath("petclinic")
71                  .build();
72          logger.info("GETTING RESPONSE\n");
73          final String response = given()
74                  .spec(spec)
75                  .params("lastName", "")
76                  .when()
77                  .get("owners")
78                  .asString();
79
80          final ImmutableList<String> allVets = ImmutableList.of(
81                  "George Franklin",
82                  "Betty Davis",
83                  "Eduardo Rodriquez",
84                  "Harold Davis",
85                  "Peter McTavish",
86                  "Jean Coleman",
87                  "Jeff Black",
88                  "Maria Escobito",
89                  "David Schroeder",
90                  "Carlos Estaban"
91          );
92          logger.info("SEARCHING FOR VETS::::");
93          for (final String vet : allVets) {
94              Assert.assertTrue(response.contains(vet));
95          }
96          logger.info("[RESULT]{}\n", allVets);
97          final String link = "\nLINK OF THE TEST:\n-> http://" +
    petClinicAddress + ":" + petClinicPort + "\n\n";
98          logger.info(link);
99          k8sHandler.logging(k8sHandler.getHashedService(), testConfig.
    getNamespace(), testConfig);
100         k8sHandler.namespaceCleaner(testConfig);
101
102     }
103
104
105 }
```

A.1.6 Ficheiros de k8s do projeto Petclinic

A.1.6.1 Ficheiro petclinic-service

```
apiVersion: v1
kind: Service
metadata:
  annotations:
    kompose.cmd: kompose convert -f docker-compose.yml
    kompose.version: 1.17.0 (a74acad)
  creationTimestamp: null
  labels:
    io.kompose.service: petclinic
  name: petclinic
spec:
  ports:
  - name: "8080"
    port: 8080
    targetPort: 8080
  selector:
    io.kompose.service: petclinic
status:
  loadBalancer: {}
```

A.1.6.2 Ficheiro petclinic-deployment

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  annotations:
    kompose.cmd: kompose convert -f docker-compose.yml
    kompose.version: 1.17.0 (a74acad)
  creationTimestamp: null
  labels:
    io.kompose.service: petclinic
  name: petclinic
spec:
  replicas: 1
  strategy: {}
  template:
    metadata:
      creationTimestamp: null
      labels:
        io.kompose.service: petclinic
    spec:
      containers:
      - image: docker.feedzai.com/zaickathon/kubertests:petclinic
        livenessProbe:
          exec:
            command:
```

```
            - sh
            - -c
            - curl -f http://localhost:8080/petclinic
          failureThreshold: 10
          periodSeconds: 30
          timeoutSeconds: 10
        name: petclinic
        ports:
        - containerPort: 8080
        resources:
          limits:
            cpu: "1"
            memory: "4294967296"
          requests:
            cpu: 500m
            memory: "3221225472"
      restartPolicy: Always
status: {}
```

A.1.6.3 Ficheiro db-service

```
apiVersion: v1
kind: Service
metadata:
  annotations:
    kompose.cmd: kompose convert -f docker-compose.yml
    kompose.version: 1.17.0 (a74acad)
  creationTimestamp: null
  labels:
    io.kompose.service: db
  name: db
spec:
  ports:
  - name: "3306"
    port: 3306
    targetPort: 3306
  selector:
    io.kompose.service: db
status:
  loadBalancer: {}
```

A.1.6.4 Ficheiro db-deployment

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  annotations:
    kompose.cmd: kompose convert -f docker-compose.yml
    kompose.version: 1.17.0 (a74acad)
  creationTimestamp: null
  labels:
    io.kompose.service: db
  name: db
spec:
  replicas: 1
  strategy: {}
  template:
    metadata:
      creationTimestamp: null
      labels:
        io.kompose.service: db
    spec:
      containers:
      - env:
        - name: MYSQL_DATABASE
          value: petclinic
        - name: MYSQL_ROOT_PASSWORD
          value: petclinic
        image: mysql:5.7
        livenessProbe:
          exec:
            command:
            - mysqladmin
            - ping
            - -h
            - localhost
          failureThreshold: 10
          timeoutSeconds: 20
        name: db
        ports:
        - containerPort: 3306
        resources: {}
      restartPolicy: Always
status: {}
```

A.1.7 Ficheiro de docker-compose para os *containers* de Chrome e Firefox

```
version: "3"
services:
pulse:
  depends_on:
    - db
  image: docker.feedzai.com/zaickathon/kubertests:pulse_v1.0
  ports:
    - "18080:8080"
  deploy:
    resources:
      limits:
        cpus: '2'
        memory: 6G
      reservations:
        cpus: '1'
        memory: 4G
  networks:
    - kubetests
db:
  image: postgres:9.4
  expose:
    - "5432"
  ports:
    - 5432:5432

  healthcheck:
    test: ["CMD-SHELL", "pg_isready","-U","postgres"]
    interval: 10s
    timeout: 5s
    retries: 5
  networks:
    - kubetests
  deploy:
    resources:
      limits:
        cpus: '1'
        memory: 2G
      reservations:
        cpus: '0.5'
        memory: 1G
firefox:
  image: selenium/node-firefox:3.14.0-gallium
  depends_on:
    - hub
  environment:
    HUB_HOST: hub
  deploy:
    resources:
```

```
        limits:
          cpus: '0.5'
          memory: '1G'
        reservations:
          cpus: '0.5'
          memory: '1G'
  chrome:
    image: selenium/node-chrome:3.14.0-gallium
    depends_on:
      - hub
    environment:
      HUB_HOST: hub
    deploy:
      resources:
        limits:
          cpus: '0.5'
          memory: '1G'
        reservations:
          cpus: '0.5'
          memory: '1G'

  hub:
    image: selenium/hub:3.14.0-gallium
    ports:
      - "4444:4444"
    networks:
      spring-petclinic:
```

A.1.8 Ficheiro de teste para os *containers* de Chrome

```
package systemtests;
import client.K8sHandler;
import client.TestConfig;
import java.net.MalformedURLException;
import java.net.URL;
import javax.print.DocFlavor;
import org.junit.Test;
import org.openqa.selenium.Capabilities;
import org.openqa.selenium.remote.RemoteWebDriver;
import org.openqa.selenium.remote.DesiredCapabilities;


public class BrowserTest {
    static Capabilities chromeCapabilities =
    DesiredCapabilities.chrome();
    static Capabilities firefoxCapabilities =
    DesiredCapabilities.firefox();
    /**
     * Class with the settings of the test.
     */
    private TestConfig testConfig = new TestConfig.Builder()
            .withNamespace("team-internship-k8s-testers")
            .withClientFile("/home/emanuel.correia/Desktop/PulseProjects
            /sft2.0/Kubetests/src/main/java/client/config.yaml")
            .isDockerComposeSetter(true)
```

```
26              .withConfigFileDir("/home/emanuel.correia/Desktop/
27              PulseProjects/sft2.0/Kubetests")
28              .withK8Surl("https://master.k8s.zai:443")
29              .isLocalFlagSetter(false)
30              .withServiceToTest("chrome","pulse")
31              .isCleanerFlagSetter(false)
32              .isDebugFlagSetter(false)
33              .build();
34
35
36      /**
37       * Handler to launch and acess to the tests.
38       */
39      private K8sHandler k8sHandler = new K8sHandler(testConfig);
40
41      /**
42       * Tester.
43       */
44      @Test
45      public void main() throws MalformedURLException {
46          final K8sHandler.ExposedService chromeExposed = new K8sHandler.
47          ExposedService().findService(k8sHandler.getHashedService()[0]);
48
49          final K8sHandler.ExposedService pulseExposed = new K8sHandler.
50          ExposedService().findService(k8sHandler.getHashedService()[1]);
51
52          final String chromeAddress = chromeExposed.getIp();
53          final String chromePort = chromeExposed.getPort();
54
55          final String pulseAddress = pulseExposed.getIp();
56          final String pulsePort = pulseExposed.getPort();
57
58          RemoteWebDriver chrome = new RemoteWebDriver(
59          new URL("http://" + chromeAddress + ":" + chromePort +"/wd/hub"),
60          chromeCapabilities);
61
62          // run against chrome
63          chrome.get("http://" + pulseAddress + ":" + pulsePort);
64          logger.info(chrome.getTitle());
65
66          chrome.quit();
67      }
```

A.2 Relatórios dos testes

Neste capítulo são apresentados os relatórios passo a passo dos testes realizados para comprovar os requisitos.

Caso de Teste:	TF1	**Nome do Caso de Teste:**	Suportar a integração da solução com CI e CD
Sistema:	System Test Framework v2.0	**Data:**	29/05/2019
Desenhado por:	Emanuel Pedro	**Executado por:**	Emanuel Pedro; Ricardo Lopes
Descrição:	Suportar mecanismos de ci e cd. Esta é a definida para executar a stf para que todas as funções sejam executadas de forma automática e que se guardem os dados.		

Pre-Condições:
Criar ficheiros de docker-compose ou kubernetes que inclua o Pulse e Postgrés.

Passo	Acção	Resultado Esperado	Passou/Falhou
1	Colocar o projeto num repositorio externo, nexus	Projeto é alojado no repositório	
2	Importar classe num projeto novo	Acede as funcionalidade sem erros	
3	Criar um teste com as configurações pretendidas	O teste aceita todas as configurações sem apresentar erros	
4	Aceder ao terminal e executar "mvn test -D=PulseTest"	Teste lançado com sucesso e apresentar os links de acesso	

Pós-Condições: O teste realizado com sucesso

Figure A.1: Relatório do teste TF1

Caso de Teste:	TF2 e TF3	**Nome do Caso de Teste:**	Suportar a integração de componentes externos e internos
Sistema:	System Test Framework v2.0	**Data:**	29/05/2019
Desenhado por:	Emanuel Pedro	**Executado por:**	Emanuel Pedro
Descrição:	A framework deve suportar todos os componentes internos da Feedzai, devido a ser uma constante repetição para cada um dos componentes e por questões de tempo o exemplo vai ser só feito para o Pulse. Como o Pulse precisa de uma base de dados o teste também vai incluir um componente externo, neste caso Postgrés		

Pre-Condições:
Criar ficheiros de docker-compose ou kubernetes que inclua o Pulse e Postgrés.

Passo	Acção	Resultado Esperado	Passou/Falhou
1	Criar um teste com as configurações pretendidas	O teste aceita todas as configurações sem apresentar erros	
2	Aceder ao terminal e executar "mvn test -D=PulseTest"	Teste lançado com sucesso e apresentar os links de acesso	
3	Aceder ao endereço do Pulse dado pela stf após o teste	Usando um browser aceder e abrir a página de login do pulse	
4	Colocar credênciais e carregar no botão "login"	Ser redirecionado para página principal do componente	
5	Lançar uma instância do Pulse	Receber a mensagem "Instância criada com sucesso"	
6	**Verificar pós-condição 1**		

Pós-Condições:
Containers continuam a correr após a execução do teste

Figure A.2: Relatório do teste TF2 e TF3

Caso de Teste:	TF4	**Nome do Caso de Teste:**	Suportar integração com os containers de Chrome e Firefox
Sistema:	System Test Framework v2.0	**Data:**	29/05/2019
Desenhado por:	Emanuel Pedro	**Executado por:**	Emanuel Pedro
Descrição:	A stf deve conseguir lançar espeficamente os componentes de chrome e firefox		

Pre-Condições:
Criar ficheiros de docker-compose ou kubernetes que inclua o chrome e o firefox

Passo	Acção	Resultado Esperado	Passou/Falhou
1	Criar um teste com as configurações pretendidas	O teste aceita todas as configurações sem apresentar erros	
2	Aceder ao terminal e executar "mvn test -D=browsersTest"	Teste lançado com sucesso e imprime o titulo apresentado no browser	
3	**Verificar pós-condição 1**		

Pós-Condições:
Containers continuam a correr após a execução do teste

Figure A.3: Relatório do teste TF4

Caso de Teste:	TF5	**Nome do Caso de Teste:**	Suportar o deployment de ambientes com diferentes configurações
Sistema:	System Test Framework v2.0	**Data:**	30/05/2019
Desenhado por:	Emanuel Pedro	**Executado por:**	Emanuel Pedro e Ricardo Lopes
Descrição:	A stf item de suportar várias configurações dependendo do tipo de testes. Cada equipa deve escolher os componentes e ambientes que quer testar. Anteriormente já foram testadas com diferentes imagens, com namespace específico e com uma configuração própria do cliente. Neste caso vamos testar no ambiente local		

Pre-Condições:
Cria ficheiro de docker-compose, neste caso é usado o do petclinic.

Passo	Acção	Resultado Esperado	Passou/Falhou
1	Criar um teste com as configurações pretendidas e com a "isLocalFlagSetter" a "true"	O teste aceita todas as configurações sem apresentar erros	
2	Aceder ao terminal e executar "mvn test -D=MySystemTest"	Teste lançado com sucesso e a lista de veterinários da clínica	
3	Aceder ao terminal e executar "minikube dashboard"	Abre o browser com a página do minikube	
4	Escolher o namespace "team-internship-k8s-testers"	Ter acesso aos componentes do namespace	
5	Aceder a "Pods"	Observar que todos os container estão a correr	

Pós-Condições:
Containers continuam a correr após a execução do teste.

Figure A.4: Relatório do teste TF5

Caso de Teste:	TF6	**Nome do Caso de Teste:**	Suportar a interação e gestão dos containers em tempo de execução
Sistema:	System Test Framework v2.0	**Data:**	05/06/2019
Desenhado por:	Emanuel Pedro	**Executado por:**	Emanuel Pedro
Descrição:	A STF2.0 tem de aceder e gerir os componentes que são usados nos testes, de forma a aceder a logs e a dados dos containers		

Pre-Condições:
Cria ficheiro de docker-compose, neste caso é usado o do petclinic.

Passo	Acção	Resultado Esperado	Passou/Falhou
1	Criar um teste com as configurações pretendidas	O teste aceita todas as configurações sem apresentar erros	
2	Aceder ao terminal e executar "mvn test -D=MySystemTest"	Teste lançado com sucesso e a lista de veterinários da clínica	
3	Aceder ao endereço do petclinic.	Conseguir entrar no container lançado e aceder a todos os dados	
4	**Verificar pós-condição 1**		

Pós-Condições:
Containers continuam a correr após a execução do teste

Figure A.5: Relatório do teste TF6

Caso de Teste:	TF7	**Nome do Caso de Teste:**	Suportar framework de testes como JUnit
Sistema:	System Test Framework v2.0	**Data:**	05/06/2019
Desenhado por	Emanuel Pedro	**Executado por:**	Emanuel Pedro
Descrição:	A STF2.0 tem de suportar mecanismos de Unit Testing como JUnit e Scala Rest são usados em vários departamentos da Feedzai		

Nota: Esta situação já é testada e comprovada, por exemplo no TF6 pois o teste que é feito para o petclinic é usando JUnit

Figure A.6: Relatório do teste TF7

Caso de Teste:	TF8	**Nome do Caso de Teste:**	Suportar conexão de bibliotecas externas usadas em System Tests
Sistema:	System Test Framework v2.0	**Data:**	06/06/2019
Desenhado por:	Emanuel Pedro	**Executado por:**	Emanuel Pedro
Descrição:	A STF2.0 tem de suportar bibliotecas externas a ser usadas para gerir as API's de produto como RMI, HTTPClient, etc. . Além dessa situação, estas configurações devem ser costumizáveis por cada equipa		

Nota: Esta situação já é testada e comprovada, por exemplo no TF6 pois o teste ja comprova a situação do HTTP.

Figure A.7: Relatório do teste TF8

Caso de Teste:	TF9 e TF21	**Nome do Caso de Teste:**	Ter suporte para Logs
Sistema:	System Test Framework v2.0	**Data:**	06/06/2019
Desenhado por:	Emanuel Pedro	**Executado por:**	Emanuel Pedro e Ricardo Lopes
Descrição:	A STF2.0 tem de suportar mecanismos de recolha de logs pois para a realização de debug é extremamente necessário; A cada instrução realizada ou erro, durante os testes, deve existir uma entrada no ficheiro de log, previamente criado para o conjunto de testes definidos.		

Pre-Condições:
Cria ficheiro de docker-compose, neste caso é usado o do petclinic.

Passo	Acção	Resultado Esperado	Passou/Falhou
1	Criar um teste com as configurações pretendidas e com a flag **isPrintLogsFlagSetter** a true e **withLogDir** com a diretoria onde se quer guardar os logs.	O teste aceita todas as configurações sem apresentar erros	
2	Aceder ao terminal e executar "mvn test -D=MySystemTest"	Teste lançado com sucesso e a lista de veterinários da clínica; Logs impressos no ecrã;	
3	Aceder à pasta dos logs	Ter acesso a todos os ficheiros .log de cada container do teste	
4	**Verificar pós-condição 1**		

Pós-Condições:
Containers continuam a correr após a execução do teste

Figure A.8: Relatório do teste TF9 e TF21

Caso de Teste TF10 **Nome do Caso de Teste:** Ter um modo debug

Sistema: System Test Framework v2.0 **Data:** 07/06/2019

Desenhado po Emanuel Pedro **Executado por:** Emanuel Pedro e Ricardo Lopes

Descrição: A STF2.0 tem de suportar mecanismos de debug, de forma a conseguir detetar mais facilmente erros ou bugs no sistema

Pre-Condições:
Cria ficheiro de docker-compose, neste caso é usado o do petclinic.

Passo	Acção	Resultado Esperado	Passou/Falhou
1	Criar um teste com as configurações pretendidas e com a flag **DebugFlag** a true.	O teste aceita todas as configurações sem apresentar erros	
2	Aceder ao terminal e executar "mvn test -D=MySystemTest"	Teste lançado com sucesso e a lista de veterinários da clínica; Cada passo executado pelo cliente é impresso e demonstrado o output da acção.	
3	Aceder aos logs no directorio raiz	Ter acesso a todos os ficheiros .log de cada container do teste	
4	Aceder ao link dado pela stf	Obter toda a informação possível do container	
5	**Verificar pós-condição 1**		

Pós-Condições:
Containers continuam a correr após a execução do teste

Figure A.9: Relatório do teste TF10

Caso de Teste: TF11 **Nome do Caso de Teste:** Suportar mecanismos de recuperação de falhas

Sistema: System Test Framework v2.0 **Data:** 07/06/2019

Desenhado por Emanuel Pedro **Executado por:** Emanuel

Descrição: Verificar que após lançar o ambiente e de correr um teste unitário com erros, que épossível aceder aos containers através dos endereços dados pela STF2.0 de forma a receber debug

Pre-Condições:
Cria ficheiro de docker-compose, neste caso é usado o do petclinic.

Passo	Acção	Resultado Esperado	Passou/Falhou
1	Criar um teste com as configurações pretendidas e com a flag **withServiceToTest**("petclini"), introduzindo um erro procurando um test inexistente	O teste aceita todas as configurações sem apresentar erros	
2	Aceder ao terminal e executar "mvn test -D=MySystemTest"	Teste lançado com sucesso e pára com a mensagem "Unknown Service"	
4	Aceder ao link dado pela stf, manualmente	Obter toda a informação possível do container	
5	**Verificar pós-condição 1**		

Pós-Condições:
Containers continuam a correr após a execução do teste

Figure A.10: Relatório do teste TF11

Caso de Teste: TF12 e TF13 **Nome do Caso de Teste:** Suportar docker-compose e kompose

Sistema: System Test Framework v2.0 **Data:** 07/06/2019

Desenhado por: Emanuel Pedro **Executado por:** Emanuel

Descrição: a STF2.0 deve suportar configurações para os testes usando docker-compose e Kompose, pois é uma forma simples de configurar os ambientes de testes e ao mesmo tempo já está a ser usado pelas equipas de Solutions (Banking e Merchants) e Costumer Success.

Pre-Condições:
Cria ficheiro de docker-compose ou kompose
Nota: Esta situação já foi comprovada anteriormente

Figure A.11: Relatório do teste TF12 E TF13

Caso de Teste: TF14 e TF15 **Nome do Caso de Teste:** Suportar clusters locais e externos de k8s

Sistema: System Test Framework v2.0 **Data:** 09/06/2019

Desenhado por Emanuel Pedro **Executado por:** Emanuel Pedro

Descrição: a STF2.0 tem de conseguir lançar os ambientes em cluster de K8S alojado na própria máquina, como minikube, ou fora como , o cluster da Feedzai

Nota: Esta situação já são testadads anteriormente como no TF5 para local e TF6 para externo

Figure A.12: Relatório do teste TF14 e TF15

Caso de Teste: TF16 **Nome do Caso de Teste:** Suportar apresentação serviços ou instâncias

Sistema: System Test Framework v2.0 **Data:** 07/06/2019

Desenhado por Emanuel Pedro **Executado por:** Emanuel

Descrição: Verificar que após lançar o ambiente, com sucesso, a STF2.0 consegue mostrar todos os serviços e instâncias lançadas no terminal

Notas:
Esta situação para já acontece em todas as execuções da stf, quando é apresentado o link para cada container

Figure A.13: Relatório do teste TF16

Caso de Teste: TE1 **Nome do Caso de Teste:** Suportar testes em paralelo

Sistema: System Test Framework v2.0 **Data:** 07/06/2019

Desenhado por: Emanuel Pedro **Executado por:** Emanuel Pedro e Ricardo Lopes

Descrição: A STF2.0 tem de integrar Kubernetes para suportar o escalonamento horizontal com base na capacidade dos clusters das várias máquinas usadas nos testes.

Pre-Condições:
Cria dois ficheiros de docker-compose ou kubernetes, neste caso usamos o Petclinic

Passo	Acção	Resultado Esperado	Passou/Falhou
1	Criar um teste com as configurações pretendidas, podem ser repetidos mudando somente o nome	Os testes aceita m todas as configurações sem apresentar erros	
2	Aceder ao terminal e executar "mvn test"	Testes lançados com sucesso	
3	Aceder aos endereços dos dois serviços dados pela stf após o teste	Através dos links aceder à página principal do petclinic com sucesso através dos dois links	
6	**Verificar pós-condição 1**		

Pós-Condições:
Containers continuam a correr após a execução do teste

Figure A.14: Relatório do teste TE1

Caso de Teste:	TP1	**Nome do Caso de Teste:**	Suportar execução para sistemas MacOs e Unix
Sistema:	System Test Framework v2.0	**Data:**	07/06/2019
Desenhado por:	Emanuel Pedro	**Executado por:**	Emanuel Pedro e Ricardo Lopes
Descrição:	A STF2.0 tem de suportar os sistemas usados nos dispositivos da Feedzai, que são MacOS e Unix.		

Notas:
Esta situação já é revista anteriormente pois são usados ambos os sitemas em situações distintas

Figure A.15: Relatório do teste TP1

Caso de Teste:	TD1	**Nome do Caso de Teste:**	Desenvolvida em Java
Sistema:	System Test Framework v2.0	**Data:**	10/06/2019
Desenhado por:	Emanuel Pedro	**Executado por:**	Emanuel Pedro
Descrição:			

Nota:
Como se pode observar no Apêndice A o código é realizado em Java

Figure A.16: Relatório do teste TD1

This page is intentionally left blank.

Apêndice B

Planeamento

Neste apêndice são apresentadas as análises gráficas do método de Gannt, que devido às dimensões não exista possibilidade de enquadrar no Capítulo 6.

B.1 Gráficos do método de Gannt do escalonamento previsto

B.1.1 Primeiro Semestre

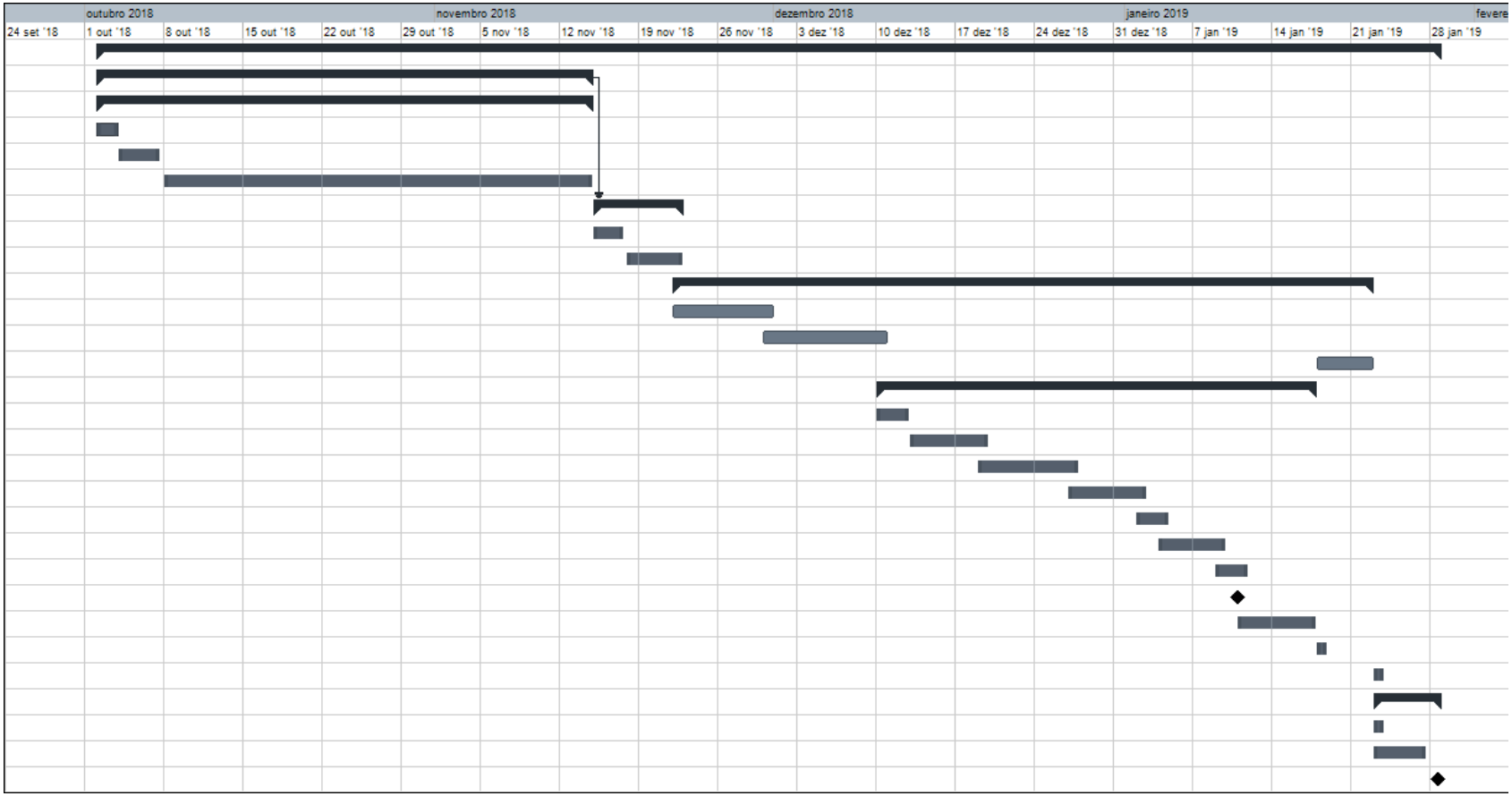

Figure B.1: Gannt do escalonamento previsto do primeiro Semestre

B.1.1.1 Segundo Semestre

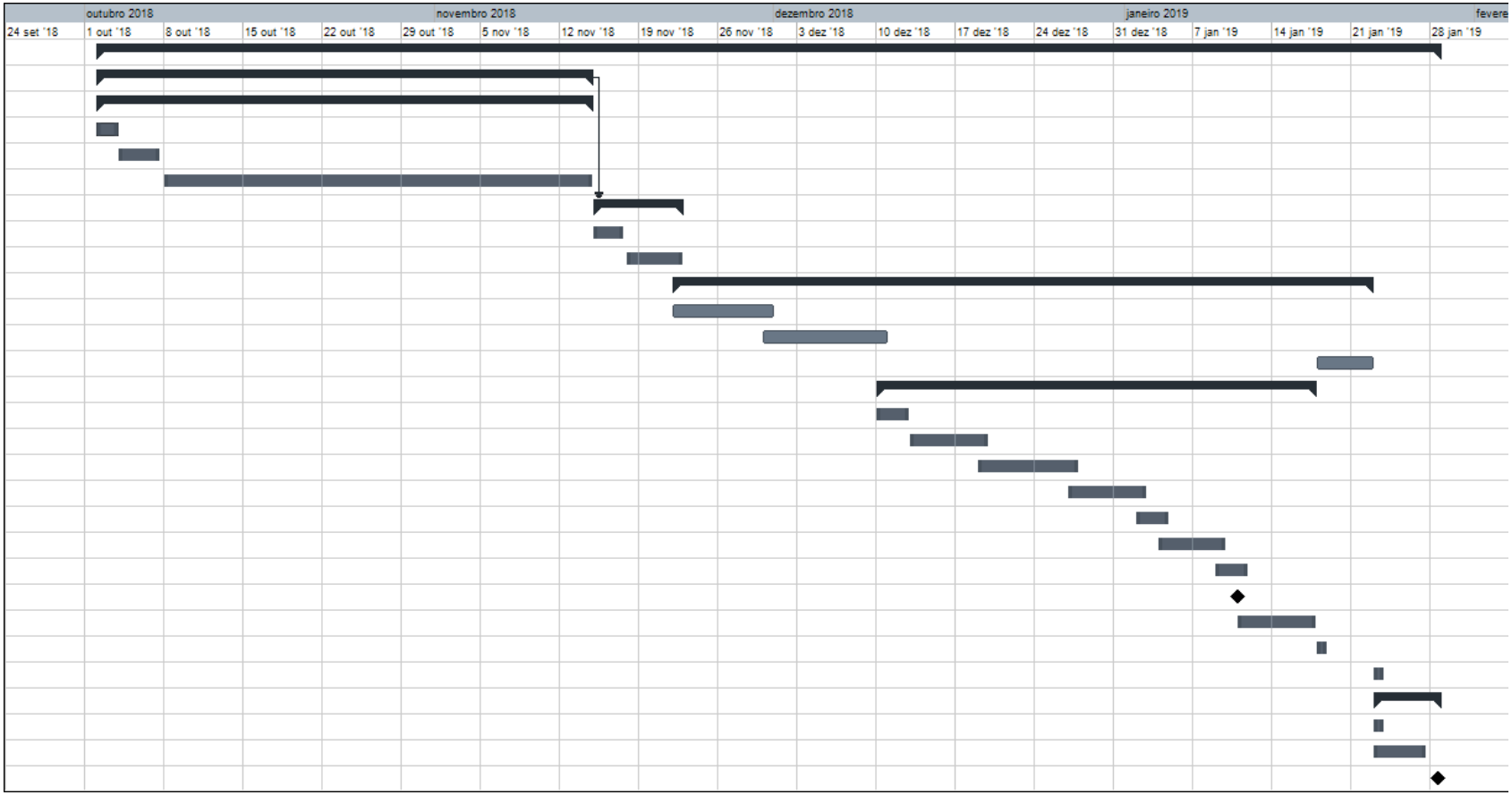

Figure B.2: Gannt do escalonamento previsto do segundo Semestre

B.1.2 Gráficos do método de Gannt do escalonamento real

B.1.2.1 Primeiro Semestre

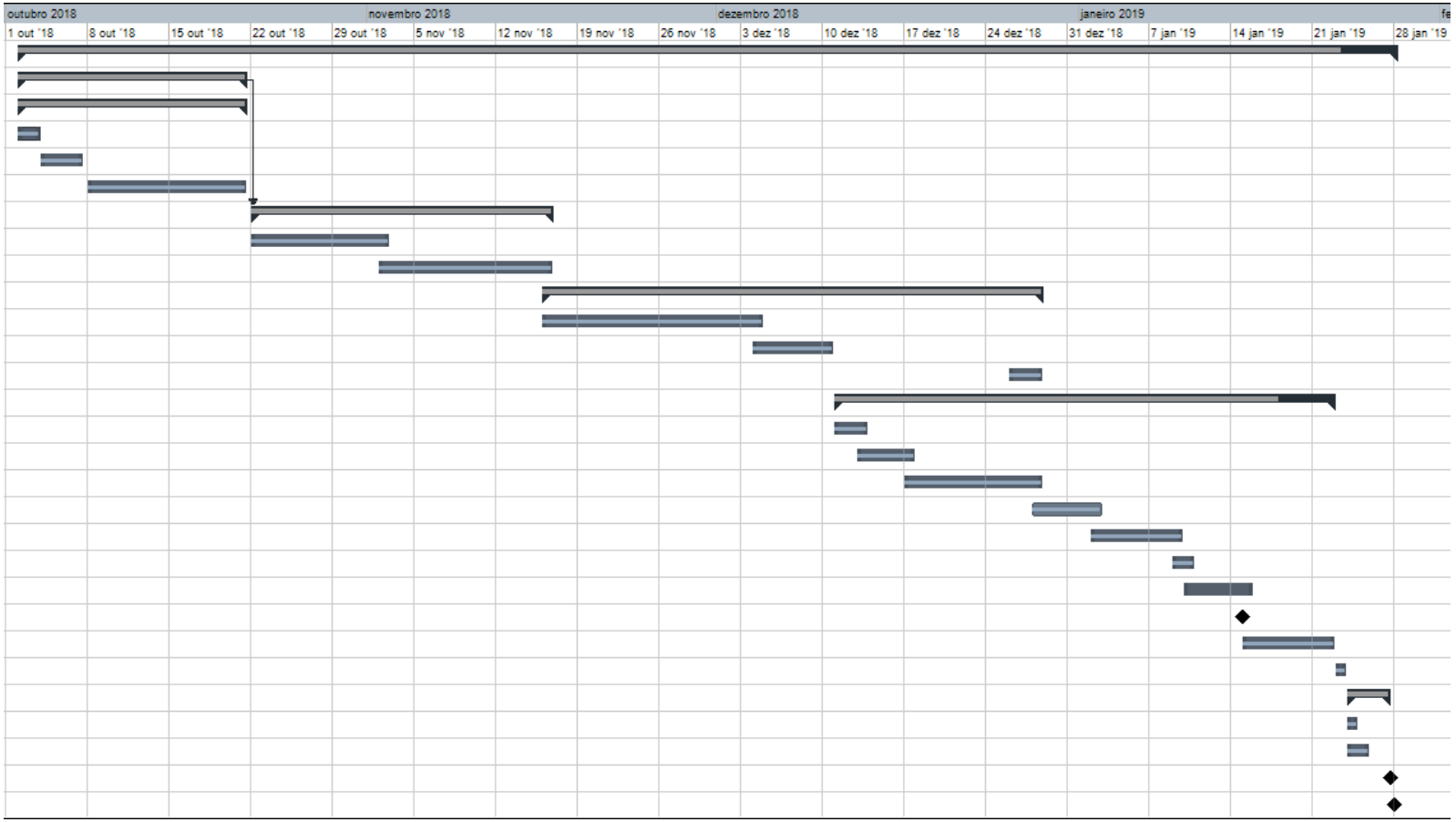

Figure B.3: Gannt do escalonamento real do primeiro semestre

B.1.2.2 Segundo Semestre

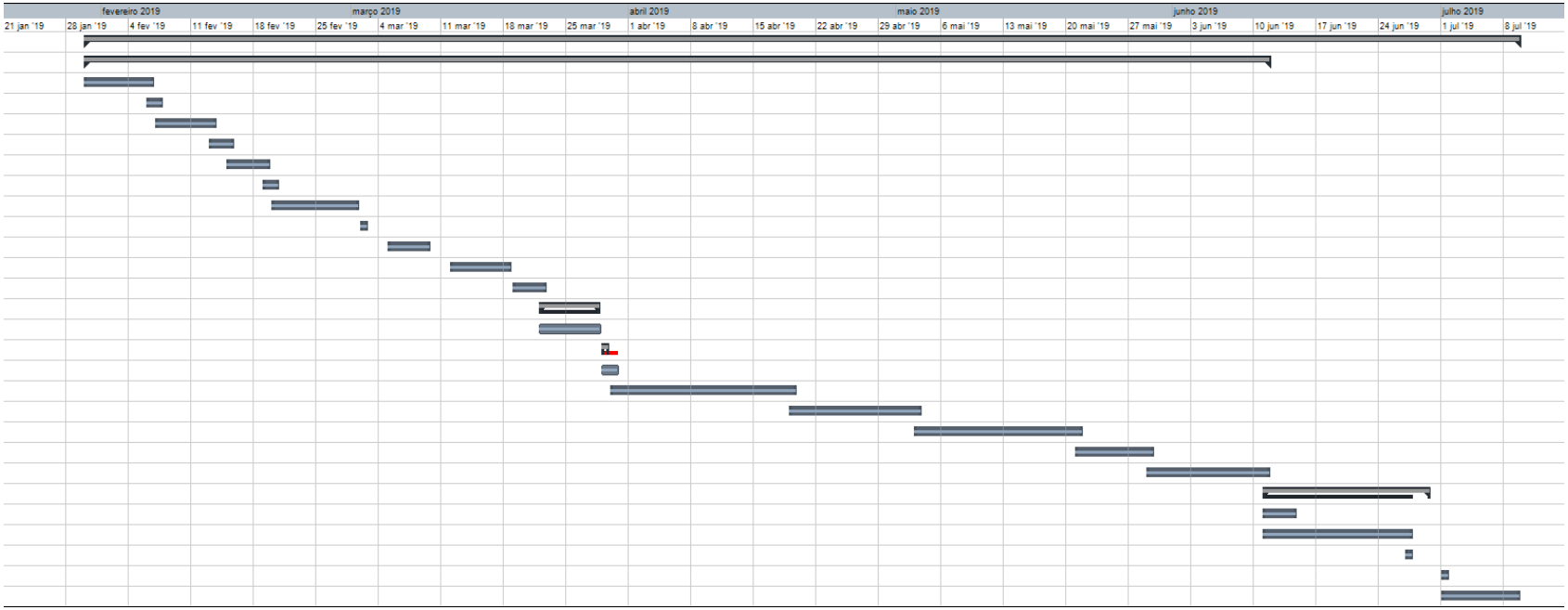

Figure B.4: Gannt do escalonamento previsto do real Semestre

This page is intentionally left blank.

Apêndice C

Arquitetura de Segundo Nível

Este apêndice vai servir para demonstrar a arquitetura de segundo nível de forma mais legível.

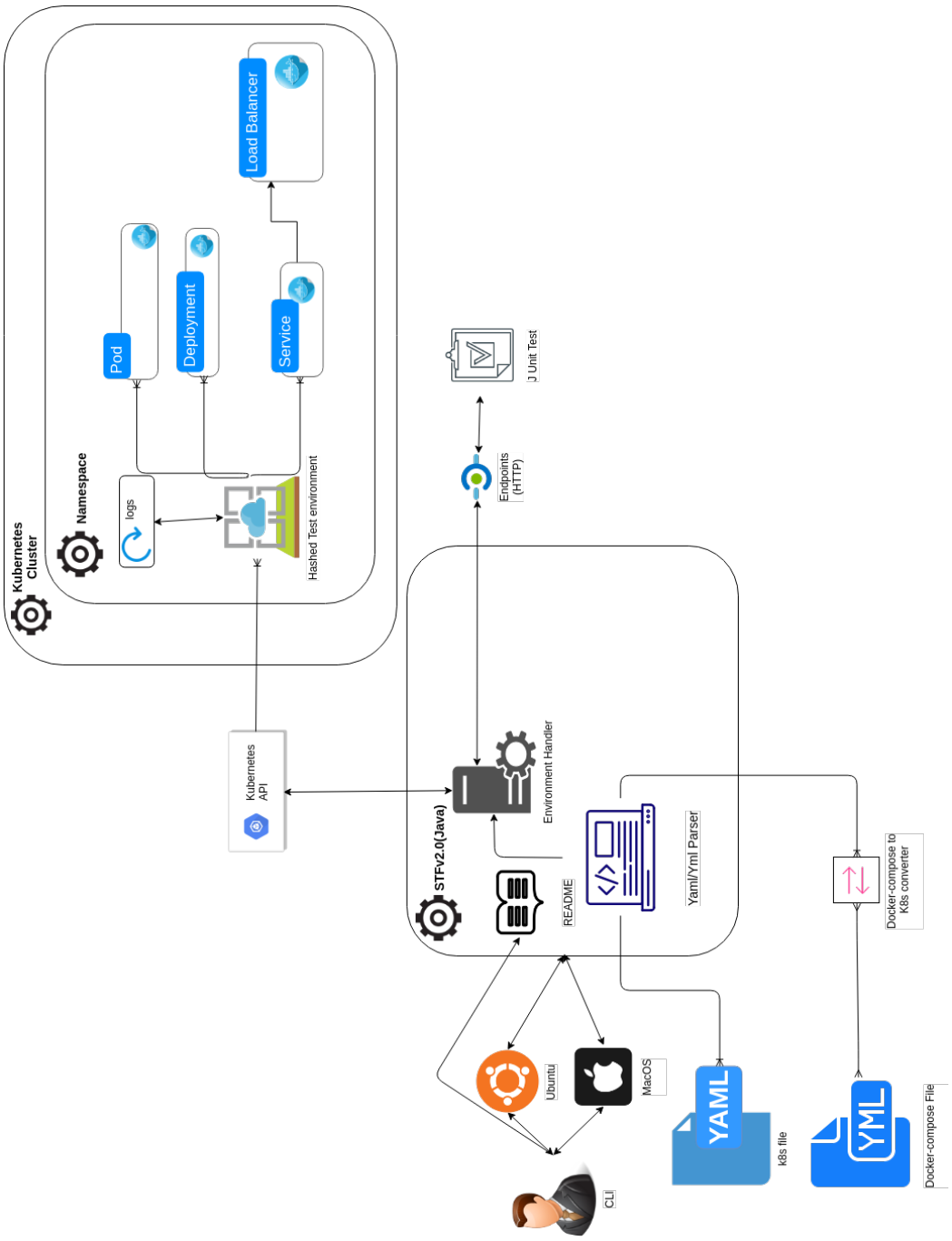

Figure C.1: Arquitetura de segundo nível da STF2.0