Instituto Politécnico de Viseu
Escola Superior de Tecnologia e Gestão de Viseu

Bruno Miguel Tavares de Almeida

Implantação de Micro-Serviços em Docker e a sua orquestração com Kubernetes no Azure (Sonae - Fashion)

Novembro de 2019

Instituto Politécnico de Viseu

Escola Superior de Tecnologia e Gestão de Viseu

Bruno Miguel Tavares de Almeida

Implantação de Micro-Serviços em Docker e a sua orquestração com Kubernetes no Azure (Sonae - Fashion)

Tese de Mestrado

Sistemas e Tecnologias de Informação para as Organizações

Prof. Doutor Paulo Rogério Perfeito Tomé

Novembro de 2019

Esta página foi propositadamente deixada em branco.

Abstract

Nowadays, more and more companies are using agile software development to build their products. Monolithic applications are increasingly divided into micro-services that can be implemented and managed individually by agile teams.

Modern container technologies such as OpenVZ or Mesos Containerizer have helped this process and Docker has had the most prominent among container technologies.

The adoption of cloud based containers is motivated by many aspects from technical and organizational to economic ones.

With Docker, there was also the need to orchestrate environments that usually have multiple containers with the most used technology being Kubernetes.

The main objective of this internship was the analysis of new paradigms of container technologies. Docker was the main study, focusing on his orchestration, in order to understand the differentiating points of the various container orchestration platforms on the market and why Kubernetes is the most popular.

The result of this study aims to implement various applications with containers. The deployment process will have to be automatic, using the creation and configuration of CI/CD pipelines.

Since the cloud service used in the company is Microsoft Azure, we will be using Azure Kubernetes Service (AKS) which is a fully managed secure and highly available kubernetes service.

Finally we intend that the various processes carried out during the internship will be documented.

Keywords

Docker

Kebernetes

Orchestration

Pipelines

Azure

Continuous integration

Continuous delivery

Esta página foi propositadamente deixada em branco.

Resumo

Hoje em dia, cada vez mais as empresas utilizam o desenvolvimento ágil de software para construir os seus produtos. Aplicações monolíticas são cada vez mais divididas em micro-serviços, que podem ser implantados e geridos individualmente por equipas ágeis.

Tecnologias modernas de *containers* como o *OpenVZ* ou o *Mesos Containerizer* vieram ajudar este processo, com o *Docker* a ter o maior destaque entre as tecnologias de *containers*. Sendo que a adopção de *containers* na *cloud* é motivada por muitos aspectos, desde técnicos e organizacionais a económicos.

Com o *Docker*, surgiu também a necessidade de orquestrar os ambientes que normalmente têm vários *containers*, sendo a tecnologia mais utilizada o *Kubernetes*.

Este estágio teve como primeiro objectivo a análise de novos paradigmas de tecnologias de *container*. *Docker* foi o principal estudado, com foco na sua orquestração, de forma a perceber os pontos diferenciadores das várias plataformas de orquestração de *containers* existentes no mercado e o porquê do *Kubernetes* ser o mais popular.

O resultado deste estudo visa a implantação das várias aplicações em *containers*. Sendo que o processo de *deployment* terá que ser automático, recorrendo para isso à criação e configuração *pipelines* de *CI/CD*.

Uma vez que o serviço de *cloud* utilizado na empresa é o *Microsoft Azure*, será utilizado o *Azure Kubernetes Service (AKS)* que é um serviço do *Kubernetes* totalmente gerido, seguro e de elevada disponibilidade.

Finalmente pretende-se que os vários processos levados a cabo ao longo do estágio sejam documentados.

Palavras-chave

Docker

Kebernetes

Orquestração

Pipelines

Azure

Integração contínua

Entrega contínua

Esta página foi propositadamente deixada em branco.

Agradecimentos

Ao Professor Paulo Tomé, pela constante disponibilidade, apoio e acompanhamento em todos os períodos do meu estágio.

À equipa de IT da Sonae Fashion, pela integração que me proporcionaram desde o primeiro momento. Pelo companheirismo e o bom ambiente constante.

Ao meu orientador Manuel Soares, pelo interesse demonstrado no sucesso do meu estágio, pelo acompanhamento e disponibilidade.

Aos meus pais por terem estado sempre comigo.

À Cristiana, por nunca me ter deixado desistir.

Esta página foi propositadamente deixada em branco.

Conteúdo

1 Introdução **1**

1.1 Enquadramento 1

1.2 Organização 2

1.3 Motivação 2

1.4 Estrutura do documento 3

2 Apresentação do estágio **5**

2.1 Digital Catalog Platform 5

2.2 Objectivos 6

2.3 Riscos 7

2.4 Equipa 8

2.5 Planeamento inicial 8

2.6 Metodologia utilizada 9

3 Estado da Arte **15**

3.1 Conceitos 15

3.1.1 Micro-serviços 16

3.1.2 Containers 18

3.1.3 Orquestração 23

3.1.4 Integração e Entrega Contínua CI/CD 24

3.1.5 Auto Scaling 26

3.2 Tecnologias utilizadas 27

3.2.1 Docker 27

3.2.2 Kubernetes . . . 30
3.2.3 Microsoft Windows Azure . . . 41
3.2.4 Azure Kubernetes Service . . . 42
3.2.5 Container Registry . . . 43
3.2.6 Azure Devops . . . 44
3.2.7 Terraform . . . 45
3.3 Comparação de soluções . . . 46
3.3.1 Docker Swarm . . . 47
3.3.2 Nomad . . . 47
3.3.3 DC/OS . . . 48
3.3.4 Comparação de sistemas de orquestração . . . 48

4 Implementação **51**
4.1 Fluxos de implantação . . . 51
4.2 Conteinerização dos Micro-serviços . . . 53
4.3 Container registry . . . 59
4.4 Nuget Server . . . 61
4.5 Azure Kubernetes Service . . . 63
4.6 Kubernetes . . . 66
4.7 Configuração dos pipelines CI/CD . . . 76

5 Conclusões **83**
5.1 Estágio . . . 83
5.2 Resultados . . . 84
5.3 Trabalho futuro . . . 85

Lista de Figuras

2-1 Aplicação *Digital Catalog Platform* 6

2-2 Metodologia *Agile* [2] 10

2-3 Jira - Atlassian 12

3-1 Abordagem de monolíto e de micro-serviços 16

3-2 Exemplo de uma máquina dedicada 19

3-3 Exemplo de uma máquina virtual 20

3-4 *Containers* 21

3-5 *OS Containers vs App Containers* 22

3-6 Orquestração 23

3-7 *Integração contínua (adaptado de* [8]) 24

3-8 *Entrega contínua (adaptado de* [7]) 25

3-9 *Auto Scaling* 26

3-10 *Drivers* e os recursos do *kernel* utilizados pelo *Docker* (adaptado de [36]) 28

3-11 Arquitectura do *Docker* [10] 29

3-12 Arquitectura do kubernetes 31

3-13 Arquitectura do kubernetes - mestre/nós 32

3-14 *Pod* 34

3-15 Fluxo de um *pod* (adaptado de [15]) 35

3-16 Rede em um *pod* 36

3-17 Armazenamento em um *pod* 37

3-18 Armazenamento persistente [16] 38

3-19 Exemplo da criação de um serviço 39

3-20 *RollingUpdate* 40
3-21 *ReplicaSet e Deployment* 40
3-22 *Namespaces* 41
3-23 Serviços do *Microsoft Azure* (adaptado de [33]) 42
3-24 *AKS* com *Azure Active Directory* 43
3-25 Exemplo de ambiente com *Container Registry* 44
3-26 Arquitectura *CLI* do *Terraform* [25] 46

4-1 Desenho dos fluxos de implantação 52
4-2 *Dockerfile* primeira abordagem 54
4-3 Árvore da solução 55
4-4 *Dockerfile* segunda abordagem 55
4-5 *Dockerfile* serviço de *Front-end* 56
4-6 Exemplo de serviço no *Docker Compose* 57
4-7 Executar serviço no *Docker Compose* 58
4-8 *Docker images* 58
4-9 *Docker Containers* 58
4-10 *Container Registry - Digital Catalog Platform* 61
4-11 *Artifact - Nuget Server* 62
4-12 Pacotes *nuget* no VS 63
4-13 *Povider do Terraform* 63
4-14 *Terraform - AKS* 64
4-15 *Dashboard - AKS* 65
4-16 *Ficheiro Kubernetes* 66
4-17 Converter texto para *base64* em sistemas *Linux* 68
4-18 *ClusterIP Kubernetes* 70
4-19 *ClusterIP* para a *WebAPI* do DCP 70
4-20 *Balanceador com Kubernetes* 71
4-21 *Ingress controllers* 73
4-22 *Ingress controllers* DCP 74

4-23 Configuração de recursos para um *pod* 74
4-24 Ficheiro de configuração das *pipelines* 77
4-25 *Pepilines* do DCP 78
4-26 *Release* do DCP 79
4-27 Estágio de um *pipeline* 80

Lista de Tabelas

2.1 Constituição da equipa de trabalho de IT comércio electrónico da Sonae Fashion 8
2.2 Planeamento inicial de trabalhos 9

3.1 Comparação de sistemas de orquestração de *containers* (X é um recurso suportado) 49

Esta página foi propositadamente deixada em branco.

Glossário

Back-end Parte da aplicação alusiva ao servidor onde se encontra a camada lógica do negócio invocada normalmente pelo front-end.

Branch Persistência de eventuais alterações efetuadas no repositório.

Commit Persistência de alterações efetuadas num dado repositório.

Deploy Disponibilização de um sistema para utilização, seja num ambiente de produção, testes, desenvolvimento, entre outros.

Front-end Parte gráfica da aplicação que é direcionada para o utilizador final com a qual este interage directamente.

Merge Integração de várias alterações num mesmo ficheiro.

Acrónimos

API Application Programming Interface.

CPU CPU Central Processing Unit.

DNS Domain Name System.

ERP Enterprise resource planning.

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

IP Internet Protocol.

IT Information technology.

OS Operating System.

VM Virtual Machine.

YAML Yet Another Markup Language.

Esta página foi propositadamente deixada em branco.

Capítulo 1

Introdução

O presente estágio enquadra-se no âmbito da disciplina de Dissertação/Projecto/Estágio do Mestrado em Sistemas e Tecnologias de Informação para as Organizações da Escola Superior de Tecnologia e Gestão de Viseu do Instituto Politécnico de Viseu. Será inserido em contexto empresarial na Fashion que faz parte do grupo Sonae.

O capítulo introdutório está dividido em quatro secções. A primeira secção faz um enquadramento do estágio a ser desenvolvido. A segunda secção apresenta a organização Sonae Fashion que faz parte do grupo Sonae e que irá acolher o estagiário durante o período de quatro meses. A terceira aborda a motivação para este estágio. Por último, e não menos importante, é apresentada a estrutura do documento.

1.1 Enquadramento

Durante o estágio estive integrado na equipa de *IT* da Sonae Fashion. Esta equipa foi criada com o intuito de desenvolver o projecto *Digital Catalog Platform* que será apresentado mais à frente.

Este projecto será desenvolvido inicialmente para dar resposta às lojas online da Mo e Zippy, nacionais e internacionais. Mas um dos objectivos é ser modular para servir outros negócios. Por exemplo, estar preparado para lidar com produtos que hoje serão

calças ou camisolas, mas amanhã os produtos podem ser frigoríficos ou bicicletas, e a peça de software ser a mesma. Além disso terá que ser escalável, por exemplo em época de saldos em vez de apenas um *pod* a tratar de transformar dados, ser possível aumentar o número de *pods* rapidamente. Lidar bem com questões de disponibilidade, por exemplo se uma máquina for abaixo, estar imediatamente outra a ocupar esse lugar. Ser confiável, não falhar ou no caso de falha os dados permanecerem intactos. E por fim a interoperabilidade, as aplicações conseguirem ser executadas em diferentes *clouds* ou plataformas.

1.2 Organização

A Sonae é uma organização multinacional que gere um portefólio diversificado de negócios nas áreas de retalho, serviços financeiros, tecnologia, centros comerciais e telecomunicações [22].

A Sonae Fashion é uma empresa responsável pela área de retalho especializado da Sonae na área de vestuário, através das marcas Deeply (equipamento e vestuário desportivo), MO (vestuário, calçado e acessórios de homem, senhora e criança), Zippy (vestuário, calçado e acessórios de bebé e criança), Losan (especializada no negócio grossista de vestuário de criança, com uma forte presença internacional) e Salsa (jeans, vestuário e acessórios) [23].

1.3 Motivação

A escolha do estágio curricular e da instituição em questão teve em conta alguns factores como, a importância no mundo empresarial da instituição em causa, e do desafio do projecto que me foi apresentado, que denotava desde o início um desafio bastante interessante.

As tecnologias utilizadas também são uma motivação extra. Devido ao crescente interesse das empresas, e da taxa de crescimento das várias tecnologias utilizadas,

nomeadamente dos micro-serviços, do *Docker* e da sua orquestração com o *Kubernetes* nas *clouds*, estes tópicos oferecem possibilidades de pesquisa e de desafio acrescido.

As *clouds* no mundo dos sistemas de informação estão em constante crescimento, que alavanca a flexibilidade da entrega de aplicações. Os *containers* permitem executar aplicações de forma flexível em ambientes virtuais como as *clouds* com menor necessidade de desempenho de *hardware* que as máquinas virtuais (*VMs*).

Com o aumento da adopção de arquitecturas micro-serviço também levou ao aumento de força dos *containers* que conseguem isolar a aplicação do sistema operativo do hospedeiro, no entanto os *containers* ainda necessitam de serem orquestrados e geridos, o que abriu um novo nicho para estruturas de gestão de *containers*, sendo que em 2015 o Google lançou um candidato a esse nicho, o *Kubernetes*.

1.4 Estrutura do documento

Este relatório é composto, por mais quatro capítulos, ao longo dos quais são apresentadas as actividades desenvolvidas no Estágio Curricular.

No capítulo 2, (Apresentação do estágio) contém a apresentação do projecto *Digital Catalog Platform*, dos objectivos do estágio, riscos inerentes ao estágio, equipa de trabalho, planeamento inicial e metodologia.

No capítulo 3, (Estado da Arte) apresentam-se conceitos, tecnologias utilizadas e comparação de soluções.

No capítulo 4, (Implementação) apresentam-se os fluxos de implantação, conteinerização dos micro-serviços, *Container Registry*, *Nuget Server*, *Azure Kubernetes Service*, *Kubernetes* e configuração dos *pipelines* de *CI/CD*.

No capítulo 5, (Conclusões) apresenta-se a reflexão final ao trabalho desenvolvido, tendo como base o trabalho realizado, resultados alcançados e trabalho futuro.

Esta página foi propositadamente deixada em branco.

Capítulo 2

Apresentação do estágio

Neste capítulo é apresentado o projecto *Digital Catalog Platform*, os objectivos do estágio, a análise de riscos do mesmo, a constituição da equipa de trabalho na instituição, o planeamento inicial do estágio e as metodologias utilizadas ao longo do mesmo.

2.1 Digital Catalog Platform

Com a decisão de uma nova peça de comércio electrónico, foi necessária uma camada de integração para enviar os dados de vendas dos sistemas *ERP* (*Retek* e *SAP*) para a nova peça de comércio electrónico (produtos, preços, categorias, imagens, ...).

O sistema anterior já era antigo, muito difícil de manter e lento para as necessidades. Era necessária uma nova plataforma, em vez de alterar o sistema anterior, daí nasceu o *Digital Catalog Platform*.

Digital Catalog Platform é uma colecção de micro-serviços, que, juntos, utilizando um *event-driven pattern*, integra dados de fornecedores, e transforma os dados para uma estrutura com formato de comércio electrónico e exporta os dados para vários clientes.

Algumas das suas funcionalidades mais relevantes são as seguintes:

- Integra as actualizações de mensagens do *ERP*.
- Integra imagens dos estúdios para utilização em comércio electrónico.
- Guarda dados integrados em um catálogo de várias lojas de comércio electrónico de estrutura mais flexível e genérico.
- API RESTfull para gerir os catálogos.
- Exportar actualizações de catálogo para vários pontos de extremidade configuráveis.
- Enriquecer os dados do catálogo, para os vários clientes de exportação (em tempo de execução).
- Lista os dados do catálogo, calculados em tempo de execução, sem a necessidade de índices e invalidações periódicas.

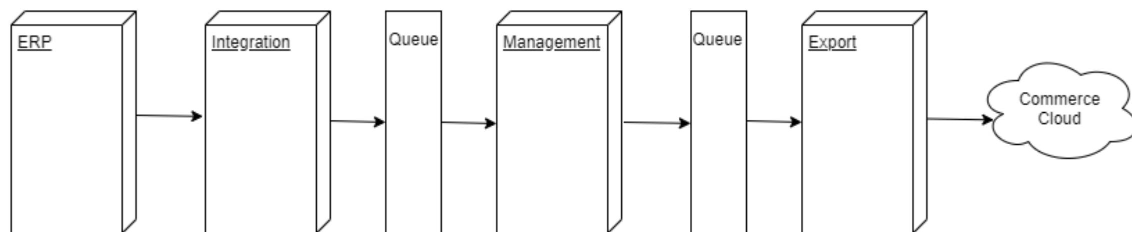

Figura 2-1: Aplicação *Digital Catalog Platform*

Os módulos de *Integration*, *Management* e de *Export* (ver figura 2-1), são constituídos por vários micro-serviços, sendo na sua maioria serviços de *Back-end*, o módulo de *Management* contém também serviços de *Front-End*.

2.2 Objectivos

O principal objectivo do estágio foi o de implantar os micro-serviços em *containers Docker* e a sua orquestração com recurso ao *Kubernetes* em ambientes *cloud*, sendo neste caso a *cloud* utilizada o Microsoft Azure.

O objectivo do estágio passou ainda por criar e configurar *pipelines* de *CI/CD* com o objectivo de automatizar todo o processo de *deploy* dos vários serviços.

Para o período do estágio foram traçados alguns objectivos principais. Dos objectivos principais é de salientar:

1. Investigar e desenho das soluções.
2. Implantar dos serviços em *Docker*.
3. Armazenar das imagens *docker* em uma solução de *Container Registry*.
4. Criar e configuração do AKS *(Azure Kubernetes Service)*.
5. Implementar da orquestração dos *containers* com *kubernetes*.
6. *Autoscaling* com base em métricas.
7. Criar de *pipelines* para automatizar os processos de *deploy*.
8. Documentar.

2.3 Riscos

Num projecto estão inerentes os riscos, que, quando não são identificados a tempo, podem levar o projecto a falhar.

Alguns dos riscos presentes são:

- A falta de especificações ou indefinição de âmbito, quando as especificações estão por definir ou mal definidas pode levar ao incumprimento de prazos. Inicialmente no estágio haviam objectivos em que as específicações não estavam totalmente definidas, o que levou a um acréscimo do risco de falha de cumprimento de prazos.
- A falta de priorização de objectivos. Os projectos de tecnologias de informação necessitam de uma gestão ágil, sendo que com isso as equipas trabalham com prazos mínimos e alterações frequentes nos propósitos, que exigem acções imediatas. Com essa situação a dificuldade para priorizar tarefas, impedindo uma melhor performance da equipa. Sendo que no estágio nem todas as tarefas estavam priorizadas e as que estavam algumas delas tiveram a sua priorização

alterada.

- A curva de aprendizagem também é um risco, principalmente em projectos com um curto período de tempo. Pelo facto de no inicio do estágio não ter contacto com algumas das tecnologias utilizadas, levou a um acréscimo do risco do projecto falhar.

2.4 Equipa

O estágio inseriu-se na equipa de trabalho da Sonae Fashion na área de comércio electrónico. Esta equipa é responsável pela parte de IT das lojas online da MO e Zippy. A constituição da equipa, pode ser visualizada na seguinte tabela (os nomes dos restantes elementos das equipas foram omitidos).

Cargo	**Colaborador**
Product Owner	
Team Leader	
Scrum Master	
Developers	Bruno Almeida
DevOps	Bruno Almeida
Quality Assurance	

Tabela 2.1: Constituição da equipa de trabalho de IT comércio electrónico da Sonae Fashion

2.5 Planeamento inicial

O planeamento inicial do estágio, após a análise dos vários objectivos propostos foi o seguinte (ver tabela 2.2).

Tarefa	**Estimativa**

Investigação e desenho das soluções.	Uma semana
Implantação dos serviços em *Docker*.	Quatro semanas
Armazenamento das imagens *Docker* em uma solução de *Container Registry*.	Dois dias
Criação e configuração do AKS *(Azure Kubernetes Service)*.	Uma semana e meia
Implementação da orquestração dos *containers* com *kubernetes*.	Quatro semanas
Criação de *pipelines* de *CI/CD* para automatizar os processos de *deploy*.	Quatro semanas
Documentação.	Uma semana

Tabela 2.2: Planeamento inicial de trabalhos

2.6 Metodologia utilizada

Apesar de na empresa não existir uma metodologia de trabalho imposta como obrigatória, o que faz todo o sentido, uma vez que as metodologias de desenvolvimento devem ser escolhidas tendo em conta vários factores, como o projecto em si, o cliente

e a própria equipa de desenvolvimento.

Para este projecto foi escolhido o *Scrum*, que faz parte do movimento *Agile* (figura 2-2). Como uma grande parte dos projectos que adoptam *Scrum*, este também fez algumas adaptações ao manifesto (documento que articula um conjunto de valores e princípios para orientar decisões sobre como desenvolver *software* de alta qualidade mais rapidamente) do *Srum* [34].

Figura 2-2: Metodologia *Agile* [2]

Agile é um conjunto de valores e princípios que descrevem as interacções e actividades diárias de um grupo. O próprio *Agile* não é prescritivo ou específico [35].

A metodologia Scrum segue os valores e princípios do *Agile*, mas inclui outras definições e especificações, especialmente em relação a certas práticas de desenvolvimento de *software* [35].

Algumas vantagens do *Scrum* e do *Agile*:

- Produtos de melhor qualidade. A capacidade de surgir e evoluir os requisitos e a capacidade de adotar mudanças ajudam a garantir que a equipa construa o produto certo, que ofereça o valor esperado ao cliente ou utilizador [27].
- Entrega mais rápida dos produtos. Comparado às metodologias tradicionais, o *Scrum* é capaz de concluir e entregar os projectos cerca de 40% mais rápido aos clientes [26].
- Maior satisfação das partes interessadas. O envolvimento activo de um dono do produto, a alta transparência do produto o progresso, e a flexibilidade de mudar quando é necessário, criam muito melhor envolvimento e satisfação do

cliente [27].

- Funcionários mais felizes. O envolvimento activo, cooperação e colaboração em equipas de *Scrum* bem sucedidas criam um local de trabalho mais agradável [27].

A metodologia *Scrum* é definida por funções de equipa, eventos (cerimónias), artefactos e regras.

Durante o estágio foram feitas as várias cerimónias do *Scrum*, como a *sprint* que é um período de tempo determinado durante o qual um trabalho específico é concluído e preparado para revisão. As *sprints* no *Scrum* têm a duração geralmente entre duas a quatro semanas [28]. No caso do estágio as *sprints* tiveram a duração de duas semanas. O uso do curto intervalo de tempo teve como base a necessidade de *input* constante do cliente no que toca a aprovação de desenvolvimentos e planeamento futuro. Revelou-se adequado por mostrar disponibilidade e adaptabilidade da equipa perante o cliente, e permitiu à equipa estar preparada para mudanças sem desenvolvimentos desnecessários e/ou redundantes.

O *stand-up* diário é uma curta reunião de comunicação (em geral no máximo de 15 minutos de duração), em que cada membro da equipa fala de forma breve e clara sobre o progresso do seu trabalho desde a última *stand-up*, o que planeia fazer até à *stand-up* seguinte, e se existe algum bloqueio que possa impactar o seu trabalho [28].

O planeamento da *sprint* são as reuniões de planeamento, que definem os itens da lista de dependências do produto que irão entrar na próxima *sprint* e como serão desenvolvidos esses itens [28]. No estágio o planeamento da *sprint* era feito a cada duas semanas antes do início de cada *sprint*.

Revisão da *sprint* é o evento para a equipa apresentar o trabalho realizado durante a *sprint*. O dono do produto verifica se o trabalho realizado cumpre os critérios de aceitação definidos, aceitando ou rejeitando o mesmo. Os clientes ou partes interessadas fornecem o seu *feedback*, com o objectivo de garantir o incremento de valor do trabalho desenvolvido [28]. No estágio a revisão da *sprint* estava planeada para ser

efectuada a cada duas semanas, mas nem sempre aconteceu, fruto da indisponibilidade do *product owner* e/ou c do cliente (equipa de negócio).

A retrospectiva é a reunião da equipa de final do *sprint*, com o objectivo de determinarem o que correu bem, o que não correu bem e o que é possível fazerem para melhorarem na próximo *sprint* [28]. No estágio a retrospectiva era feito a cada duas semanas no final de cada *sprint*.

Os artefactos que fazem parte do *Scrum* é o *Product Backlog*, o *Sprint Backlog* e o Incremento [21].

No estágio para a gestão dos artefactos foi utilizado o *Jira Software* da *Atlassian* que oferece quadros de *Scrum* prontos a serem utilizados (figura 2-3).

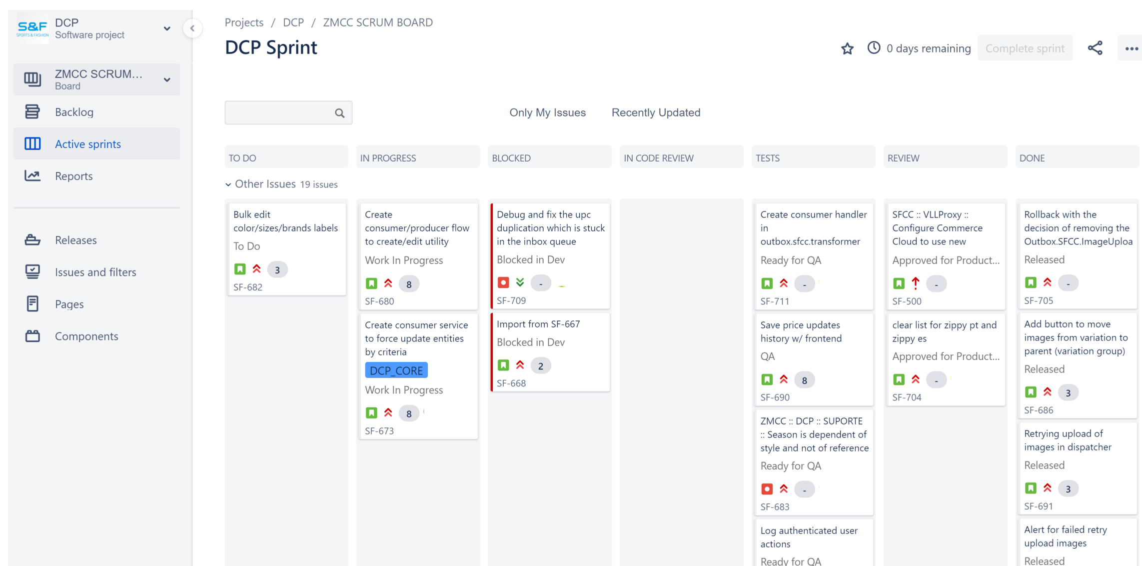

Figura 2-3: Jira - Atlassian

Os painéis são centrados na gestão de tarefas, onde elas são mapeadas para os fluxos de trabalho personalizáveis. Os painéis do *Jira Software* oferecem transparência em todo o trabalho da equipa e visibilidade sobre o estado de cada tarefa de trabalho. Os recursos de monitorização de trabalho e os relatórios de desempenho em tempo real (gráficos de *burn-up/down*, relatórios das *sprints*, e gráficos de performance) permitem a monitorização da produtividade das equipas ao longo do tempo [12].

Esta página foi propositadamente deixada em branco.

Capítulo 3

Estado da Arte

A análise dos conceitos teóricos e das tecnologias utilizadas em um projecto são tarefas de extrema importância, para com isso existir uma percepção da viabilidade do projecto, com o objectivo de descobrir se as tecnologias conseguem responder às necessidades do projecto e efectuar uma comparação de tecnologias/soluções existentes, comparando os pontos fortes e fracos.

O presente capítulo aborda numa primeira secção os conceitos teóricos necessários para a compreenção das tecnologias utilizadas, em que é efectuada uma análise às mesmas na segunda secção. Por último na terceira secção é efectuada uma comparação de soluções existentes de orquestração de *containers*.

3.1 Conceitos

Existem alguns conceitos básicos que são necessários para explicar com mais detalhes, a fim de entender o trabalho realizado durante o estágio. Esses conceitos serão apresentados ao longo da presente secção.

3.1.1 Micro-serviços

A abordagem tradicional para a criação de aplicações tem como foco as construções monolíticas. Nelas, todas as partes implantáveis ficam contidas na própria aplicação [29].

Os micro-serviços são uma arquitectura e uma abordagem para escrever *software*. Com eles, as aplicações são desmembradas em componentes mínimos e independentes. Diferente da abordagem tradicional monolítica em que toda a aplicação é criada como um único bloco, os micro-serviços são componentes separados que trabalham juntos para realizar as mesmas tarefas (figura 3-1). Cada um dos componentes ou processos é um micro-serviço. Essa abordagem de desenvolvimento de *software* valoriza a granularidade, a leveza e a capacidade de compartilhar processos semelhantes entre as várias aplicações. Trata-se de um componente indispensável para a optimização do desenvolvimento de aplicações para um modelo nativo em nuvem [29].

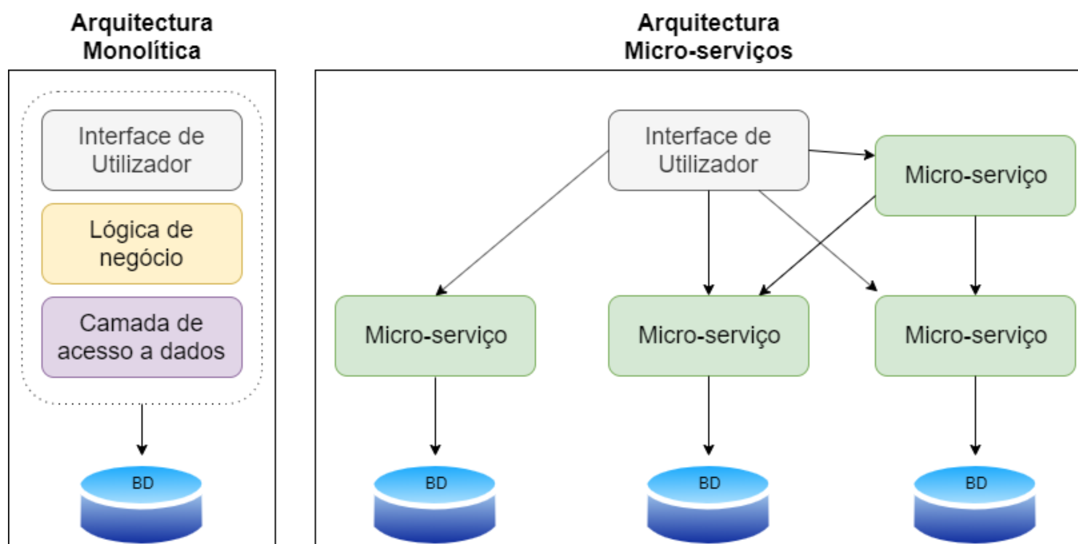

Figura 3-1: Abordagem de monolíto e de micro-serviços

Embora a arquitetura dos micro-serviços possa parecer mais complicada do que sua contraparte monolítica, as suas vantagens são múltiplas.

Uma das vantagens é a escalabilidade aprimorada, com os micro-serviços é possível

escalar independentemente cada um dos serviços, sendo a facilidade e o custo do dimensionamento drasticamente menor do que em um sistema monolítico [17].

Uma outra grande vantagem é a resiliência. Os serviços independentes, se construídos corretamente, não afetam uns aos outros. Isso significa que, se um micro-serviço falhar, o restante da aplicação permanece a funcionar, pelo contrário normalmente no modelo monolítico em que uma falha pode levar à inactividade de toda a aplicação [17].

Ao utilizar micro-serviços elimina-se a necessidade de criar peças ou funcionalidades padrão utilizadas na organização várias vezes, por exemplo, autenticação e gestão de utilizadores. Ao desenvolver esses serviços é possível dividir aplicações monolíticas em várias aplicações menores e mais sustentáveis [1].

Uma outra vantagem é o aumento da produtividade de desenvolvimento. Um novo programador normalmente começa a produzir mais rapidamente em micro-serviços, uma vez que é mais fácil de perceber uma pequena parte isolada do que uma aplicação monolítica inteira [17].

Equipas de desenvolvimento menores e mais ágeis. Nas organizações de *software* modernas, as equipas normalmente são organizadas pelos micro-serviços em que trabalham. Essas equipas envolvem menos pessoas e estão mais focadas na tarefa em questão [17].

Apesar das várias vantagens da utilização de micro-serviços esta arquitectura também traz alguns desafios.

Os testes sejam *end-to-end* ou de integração, podem ser mais difíceis que em sistema monolíticos. Uma falha em um serviço por exemplo no inicio do fluxo, pode provocar outras falhas mais adiante em outros serviços [38].

Os *logs* são das partes mais importantes de uma aplicação, principalmente em aplicações críticas e grandes. Com os micro-serviços, é necessário a existência de *logs* centralizados, permitindo assim unificar os *logs* dos vários serviços, senão é praticamente impossível gerir os logs de cada serviço separadamente.

Com os micro-serviços aumenta a complexidade operacional geral de um projeto e, como resultado, exige uma equipa de operações madura para gerir os vários serviços [1].

A implantação manual de micro-serviços é complexa demais. Assim sendo, é necessário investir bastante em automação, levando isso a um desafio da configuração inicial por exemplo de *pipelines* de *CI/CD* [1].

Um outro desafio dos micro-serviços é o controlo de versões. A actualização de versões é sempre um enorme desafio quando existem vários serviços, em que uma alteração em um serviço pode impactar vários outros. Para resolver esse problema existem normalmente duas opções, utilizar lógica condicional ou como alternativa, utilizar diferentes versões para diferentes clientes, sendo que a manutenção dessas diferentes versões pode tornar-se uma verdadeira dor de cabeça.

3.1.2 Containers

Ao início na revolução das tecnologias de informação, a maioria das aplicações era implantada directamente no *hardware* físico, no sistema operativo da máquina. Com esta abordagem o tempo de execução das aplicações era partilhado [36].

A implantação era estável, centrada no *hardware* e tinha um longo ciclo de manutenção. Era gerida principalmente por um departamento dedicado à administração de sistemas, em que o programadores tinham menos flexibilidade.

Nesses casos, os recursos de *hardware* eram regularmente subutilizados, em que muitas das aplicações tradicionais não tiravam recurso de processadores *multi-core*, e em outros casos uma máquina estava apenas dedicada para um tipo de aplicação.

A figura 3-2 mostra essa configuração.

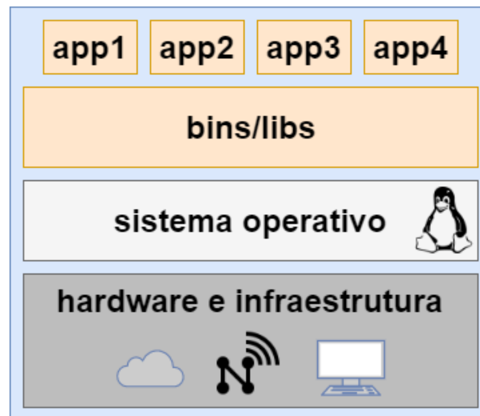

Figura 3-2: Exemplo de uma máquina dedicada

Para superar as limitações estabelecidas pela implantação tradicional, surgiu a virtualização. Aparecendo hipervisores como o *XEN* ou o *Hyper-V* entre várias outras. Com isto foi possível a emulação do *hardware* para máquinas virtuais (VMs) e implantar um sistema operativo, que pode mesmo ser diferente do sistema operativo do hospedeiro. Isso significa que somos responsáveis pela gestão de actualizações, segurança ou desempenho dessa máquina virtual [36].

Com a virtualização do tipo 2, as aplicações são isoladas ao nível da máquina virtual e os seus ciclos de vida estão dependentes das mesmas. Com tudo isto, o retorno de investimento é superior, existe mais flexibilidade, mas a complexidade também aumenta, assim como a redundância [36].

A figura 3-3 mostra essa configuração.

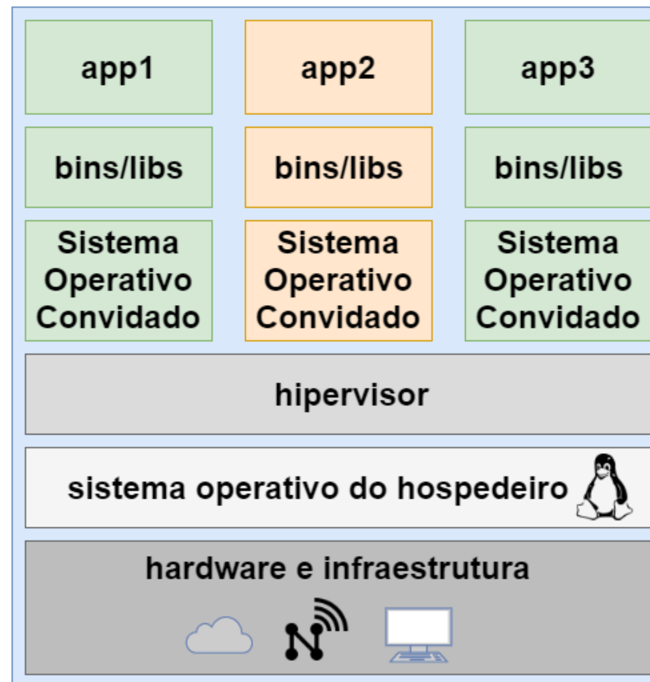

Figura 3-3: Exemplo de uma máquina virtual

Posteriormente à virtualização, a área de tecnologias de informação está cada vez mais centrada em aplicações.

A camada do hipervisor foi removida com o objectivo de reduzir a emulação e a complexidade do *hardware*. As aplicações são empacotadas com o ambiente de tempo de execução (*container runtime*), e são implementados em um *container* [36].

OpenVZ, Solaris Zones e LXC são alguns exemplos de tecnologia de *containers*.

Uma das vantagens dos *containers* em relação às máquinas virtuais é o melhor aproveitamento das capacidades de *hardware*, que fica menos comprometido com a execução de vários sistemas operativos em simultâneo [36].

Essa menor exigência de recursos, faz com que a tecnologia de *containers* seja mais portátil e flexível, sendo possível a sua transferência entre máquinas muito mais fácil.

Os *containers* podem ser criados muito mais rapidamente que as máquinas virtuais.

Sendo que isso contribui para ambientes mais ágeis e facilita o aparecimento de novas abordagens, como por exemplo os micro-serviços.

Um outro ponto interessante é o de que múltiplos *containers* em execução na mesma máquina, podem mais facilmente compartilharem as suas aplicações e recursos do que as máquinas virtuais.

Mas os *containers* são considerados menos seguros do que as máquinas virtuais, devido a que nos *containers* tudo é executado no sistema operativo hospedeiro (figura 3-4). Se um *container* for comprometido, talvez seja possível obter acesso ao sistema operativo da máquina hospedeira.

Os *containers* podem ser um pouco mais complexos de configurar e de gerir, talvez por estes motivos houve uma demora da adopção em massa dos *containers*, apesar de existir essa tecnologia há vários anos.

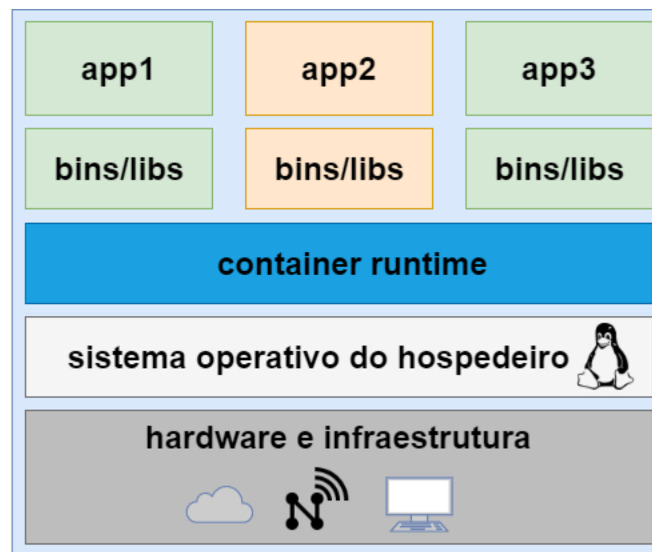

Figura 3-4: *Containers*

Resumidamente, uma máquina virtual emula um sistema operativo completo e carrega um *kernel* na sua própria região de memória, enquanto os *containers* executam um sistema operativo dentro do sistema operativo da máquina hospedeira, o que significa que o sistema operativo do *container* compartilha o mesmo *kernel* que o sistema

operativo do hospedeiro, levando a uma redução do desempenho e da sobrecarga.

A utilização dos *containers* depende das necessidades do utilizador, existem dois tipos de *container* como mostrado na figura 3-5, *OS containers* e *application container*.

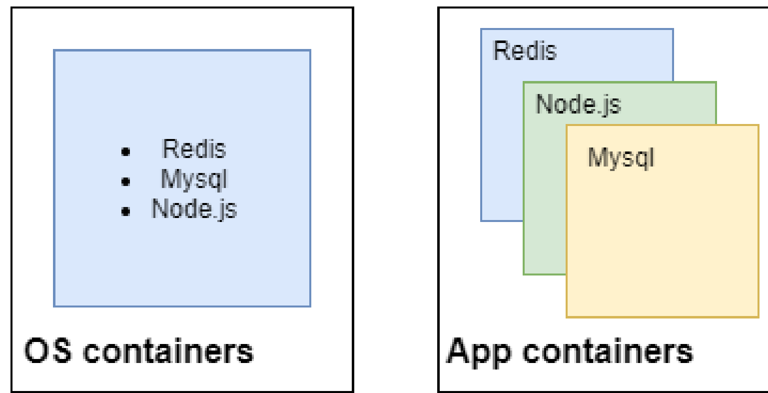

Figura 3-5: *OS Containers vs App Containers*

OS containers são ambientes virtuais, que combinam *cgroups* e *namespaces* para conseguirem fornecer um isolamento para as aplicações.

Este tipo de *container* é bastante útil para simular um sistema operativo dentro de um *container*, utilizando para isso tecnologias como *LXD*, *LXC* ou *OpenVZ*. Com isto é possível a criação de um ambiente bastante semelhante ao de uma máquina virtual, mas com a vantagem de ter uma inicialização bastante mais rápida.

Uma das desvantagens deste tipo de *container* é que não é possível para um sistema *Linux* hospedar por exemplo um *Windows OS container*.

Os *Application containers* são bastante úteis para isolarem e executarem um único serviço.

É possível a execução de vários *containers*, todos eles isolados do sistema operativo hospedeiro, estando um único serviço a ser executado em cada *container* sem a necessidade de ser carregado todo o sistema operativo, sendo apenas carregadas as bibliotecas necessárias para o funcionamento do serviço.

Os *Application containers* podem ser utilizados dentro de um *OS container*.

3.1.3 Orquestração

A orquestração de aplicações ou serviços é o processo de integrar duas ou mais aplicações/serviços para automatizar um processo ou sincronizar dados [30].

Na orquestração de *containers* está adicionalmente incluida a gestão, o estado, a reinicialização de *containers* não íntegros, escalabilidade de aplicações e permite que os vários *containers* comuniquem entre si através de descoberta de redes e serviços.

Existem ferramentas do *docker* para orquestração. O *docker compose* pode ser utilizado para orquestrar vários *containers* mas está limitado a que todos eles habitem no mesmo *host* [18]. Por outro lado, o *docker swarm* inclui gestão para vários *hosts* [24].

Os provedores de serviços de *cloud* estão a apostar cada vez mais em fornecer soluções para orquestração de *containers*, sendo a maioria dessas soluções com base no *kubernetes*.

Um exemplo de orquestração pode ser visto na figura 3-6, em que uma *Framework* de orquestração, consegue orquestrar tanto aplicações como máquinas em um *Datacenter*.

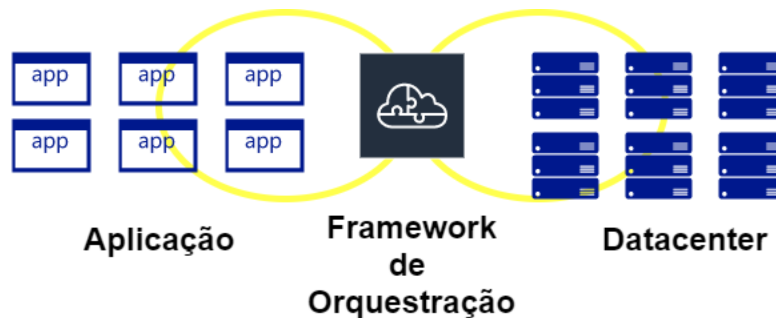

Figura 3-6: Orquestração

3.1.4 Integração e Entrega Contínua CI/CD

A integração contínua é uma prática em que o projecto deve ter um repositório central em que os programadores enviam o seu código para esse repositório o mais frequentemente quanto possível (normalmente no prazo de algumas horas). Normalmente existe um controlo de versões (Bitbucket, gitlab, etc), que cria e executa testes automaticamente a cada alteração de código no repositório, em que esse processo de detecção de erros é disparado a cada alteração de código no repositório central (figura 3-7). Dessa forma os problemas de implementações são encontrados de forma automática e de forma bastante rápida, uma vez que é apenas o tempo de envio do código para o repositório central e o processo automático executar e enviar o resultado.

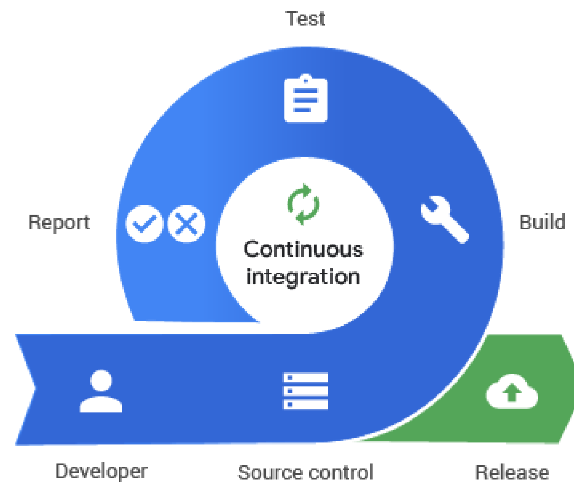

Figura 3-7: *Integração contínua (adaptado de* [8])

O objectivo principal da entrega contínua é fornecer o produto incrementalmente em iterações mais curtas. Em outras palavras, entrega contínua é a implementação de um ciclo curto em que o código é frequentemente desenvolvido, compilado, verificado, os testes são automatizados e as implantações são frequentes (figura 3-8). A entrega contínua oferece fluxos consistentes, que permitam a entrega de novos recursos para o cliente o mais rapidamente possível. Tal como com a integração contínua, com a entrega contínua permite detectar problemas mais rapidamente, tendo como consequência disso uma maior confiabilidade dos produtos. É muito raro existir entrega

contínua sem que exista a integração contínua.

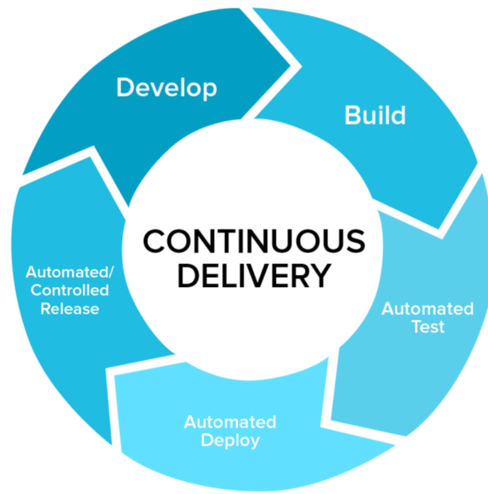

Figura 3-8: *Entrega contínua (adaptado de* [7])

3.1.5 Auto Scaling

O *Auto scaling* é a forma de aumentar ou diminuir automaticamente o número de recursos em computação, que estão atribuídos a uma aplicação tendo como base as necessidades em um determinado momento (figura 3-9).

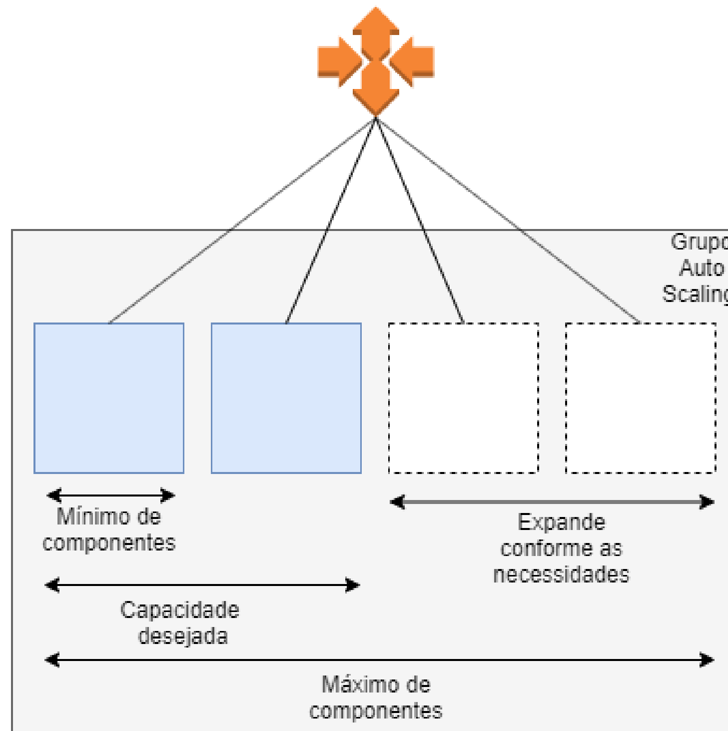

Figura 3-9: *Auto Scaling*

Surgiu com o aparecimento das tecnologias *cloud*, que proporcionou a revolução relativamente à maneira como os recursos são alocados, com a possibilidade de criar aplicações totalmente escaláveis na *cloud*.

Quando uma aplicação necessitar de mais poder de computação, é possível lançar recursos adicionais sob demanda de forma automática e utilizar os mesmos durante o tempo necessário e diminui esse número de recursos quando não são mais necessários, também de forma totalmente automática.

Existe um número limitado de recursos para lidar com a carga da aplicação. Quando o número de solicitações aumenta, a carga nesses recursos também aumenta, podendo

levar a latência na aplicação e até mesmo a falhas. Com o *Auto scaling* é possível superar essas falhas, uma vez que ele garante que os recursos sejam suficientes para conseguirem responder à carga na aplicação.

Normalmente no *Auto scaling* é possível a configuração do número mínimo de recursos. É possível especificar um número máximo, para garantir que os recursos nunca ultrapassem esse valor (contenção de custos). E é possível definir com que percentagem de carga será lançado um novo recurso ou até mesmo o número de recursos de cada vez, dependendo da criticidade da aplicação. Por exemplo é possível configurar que caso o *CPU* nos recursos ultrapasse os 70% de utilização, e caso isso aconteça serão lançados 2 novos recursos ao mesmo tempo.

3.2 Tecnologias utilizadas

As principais tecnologias pesquisadas neste estágio serão apresentadas nesta seção. Elas serão analisadas em detalhe para fornecerem uma boa visão geral dos seus recursos.

3.2.1 Docker

O *Docker* é uma solução de *containers open source*, tendo como foco a virtualização do nível dos sistemas operativos, em que as aplicações são empacotadas, distribuídas e executadas através de *containers Docker*.

O *Docker* utiliza os recursos subjacentes do *kernel* do *Linux*, que permitem a conteinerização. O diagrama da figura 3-10 demonstra os *drivers* e os recursos do *kernel* utilizados pelo *Docker*.

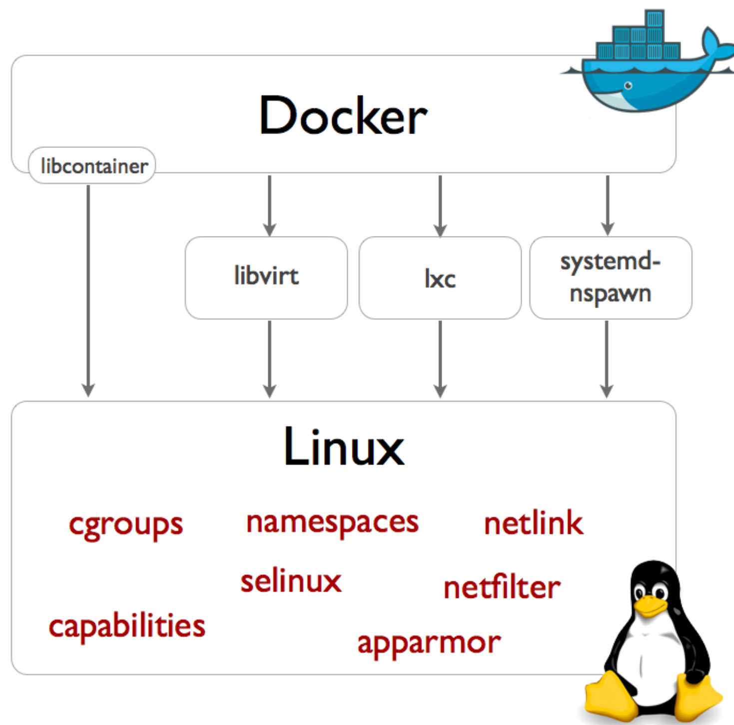

Figura 3-10: *Drivers* e os recursos do *kernel* utilizados pelo *Docker* (adaptado de [36])

Os *Namespaces* são os blocos de construção de um *container*. Existem diferentes tipos de *namespaces* como o *user*, *mnt*, *pid* dentre outros, sendo que cada um deles isola aplicações um do outro. Eles são criados com a chamada do sistema *clone* e podem anexar-se a outros *namespaces* já existentes.

Os *Cgroups* fornecem limitações de recursos e contabilização de *containers*. Em vez de definirem um limite de recursos para um único processo, os *cgroups* permitem a limitação de recursos para um grupo de processos.

Os *Union filesystem* criam camadas, tornando as mesmas muito leves e rápidas. O *Docker Engine* utiliza o *UnionFS* para fornecer blocos de construção para *containers*.

O *Docker Engine* combina os *namespaces*, *Cgroups* e o *UnionFS* num *wrapper* chamado formato de *container*. O formato padrão do *container* é *libcontainer* [10].

O Docker utiliza uma arquitectura cliente-servidor (figura 3-11). O cliente *Docker* conversa com o *Docker daemon*, que realiza o trabalho pesado de construir, executar e distribuir os *containers* [10].

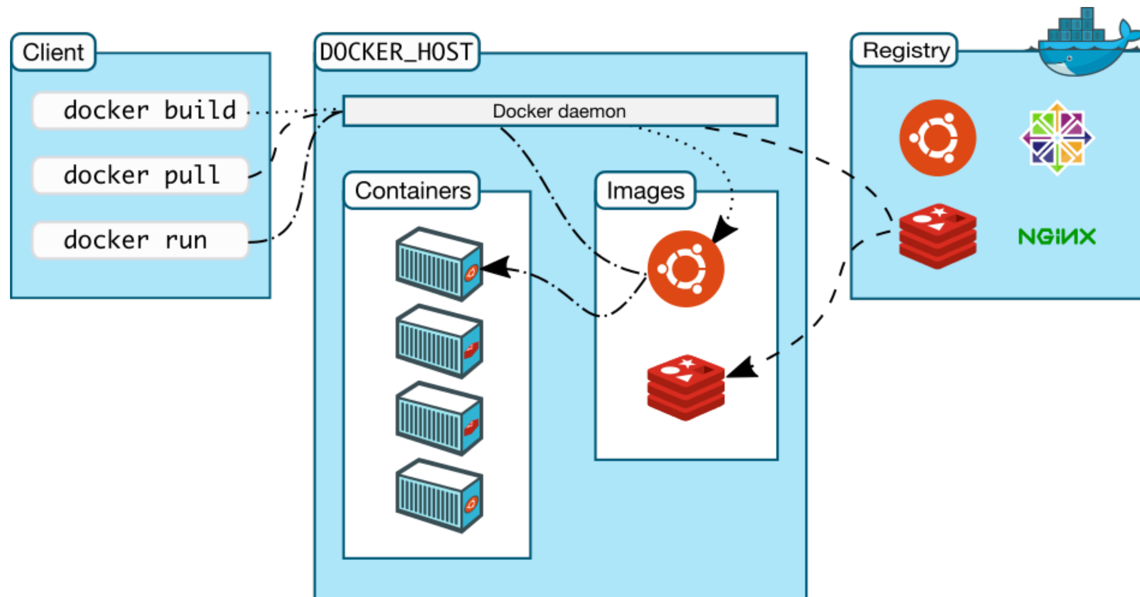

Figura 3-11: Arquitectura do *Docker* [10]

O *Docker daemon* escuta solicitações da *API* do *Docker* e gere objetos do *Docker*, como imagens, *containers*, redes e volumes. Um *daemon* também pode comunicar com outros *daemons* para gerir os serviços do *Docker*.

O cliente *Docker* fornece uma *CLI* que permite executar comandos de compilação, execução ou de paragem para um *Docker daemon*. Comandos como *docker run*, é o cliente que os envia para o *dockerd*, que os executa.

O *Docker* utiliza uma arquitetura cliente-servidor. O cliente do *Docker* conversa com o *Docker daemon*, que é quem realiza o trabalho pesado de construir, executar e distribuir os *containers*. O cliente e o *Docker daemon* podem ser executados no mesmo sistema. O cliente pode conectar-se a um *Docker daemon* remoto, utilizando para isso uma *API REST*, *sockets UNIX* ou uma interface de rede [10].

As *Docker Images* são templates que são gerados a partir dos *DockerFiles*. Incluem os passos para a instalação e execução da aplicação, o que permite a criação do *docker*

container.

3.2.2 Kubernetes

O kubernetes é um sistema de código aberto que foi inicialmente desenvolvido pelo Google que disponibiliza maneiras de gerir aplicações em contentores em toda a sua extensão. Essa gestão inclui desde a escalabilidade, balanceamento de carga dos serviços, garantir que os contentores sejam mantidos em execução, rede, entre outros. Essa gestão é alcançada através de uma variedade de sistemas independentes, interagindo em curtos ciclos de controlo por meio de uma API comum fornecida por um serviço central. Essa API é declarativa, significando que descreve o estado desejado do sistema em vez das mutações do estado existente.

Algumas características importantes do Kubernetes [32]:

- **Portável:** pode ser executado em ambientes de *cloud* públicas, privadas, híbridas ou em ambientes *multi-cloud* (por exemplo, a aplicação ser executada em duas ou mais *clouds* em simultâneo).
- **Extensível:** é modular, permite a utilização de *plugins*.
- **Permite autocorrecção:** reinício automático (em caso de falha na aplicações, por exemplo), posicionamento automático, replicação automática, dimensionamento automático.

De entre outras funcionalidades, o Kubernetes permite:

- Orquestração de contentores em múltiplas máquinas, em *clouds* privadas, públicas ou híbridas.
- Garante a integridade e a auto recuperação de aplicações em contentores, com o reinício, replicação e escalabilidade automática.
- Optimização dos recursos das máquinas, maximizando a disponibilidade de recursos de *hardware* para a execução das aplicações.
- Maior agilidade para escalar aplicações em contentores.
- Gestão e automatização dos *deployments* e das actualizações das aplicações.

Por omissão o Kubernetes é um sistema com um único mestre (pode conter mais) e um número variável de nós de trabalho. Um nó é um único servidor que faz parte de um *cluster* de Kubernetes (figura 3-12).

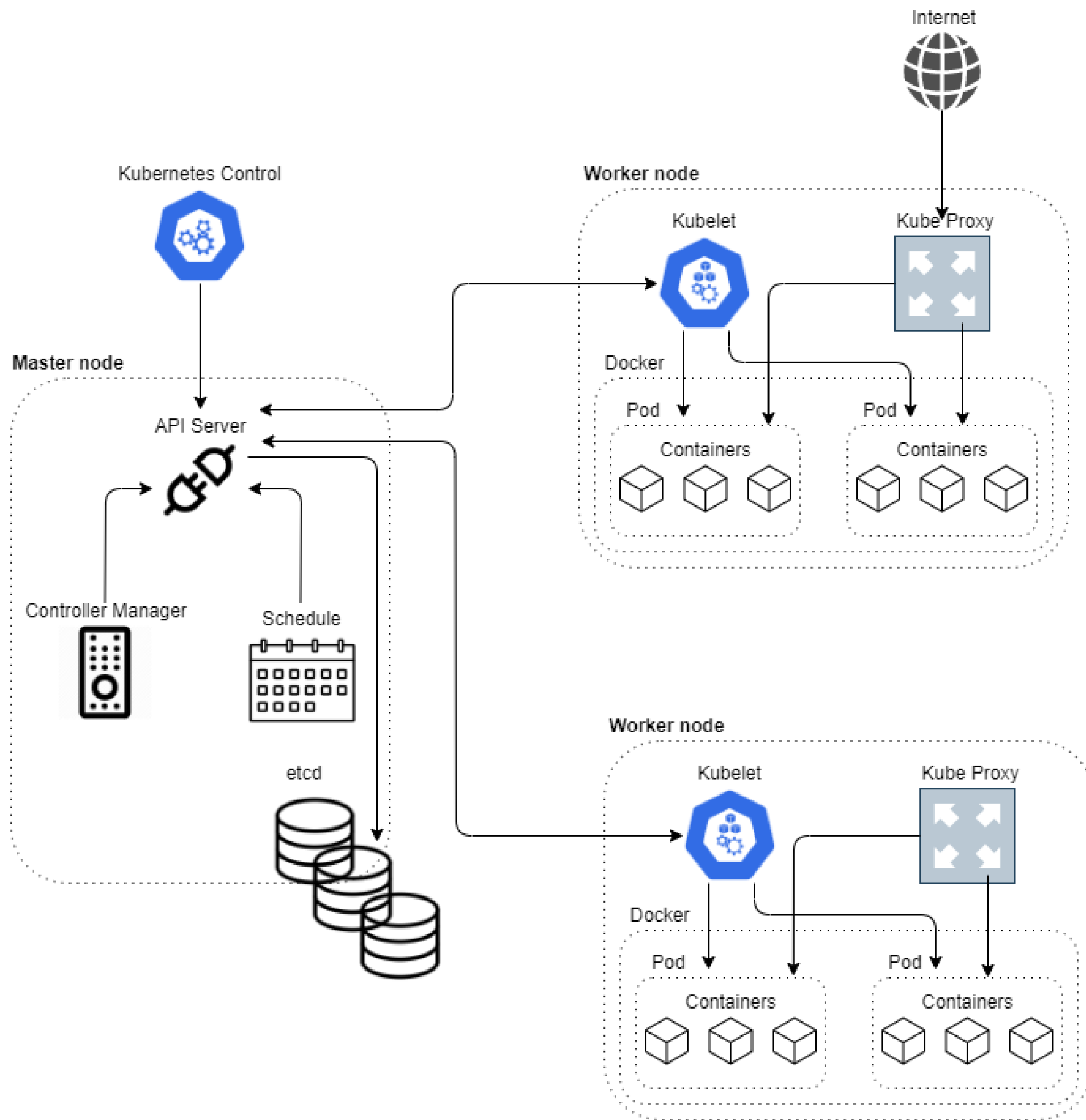

Figura 3-12: Arquitectura do kubernetes

Mestre e os seus componentes são responsáveis pela gestão de todo o *cluster* de Kubernetes.

A configuração e o estado de todo o sistema são tratados pelo mestre e persistidos no *etcd*.

O mestre disponibiliza o *kubectl* que é uma interface de linha de comando para executar comandos nos *clusters* do *kubernetes* (figura 3-13) [19].

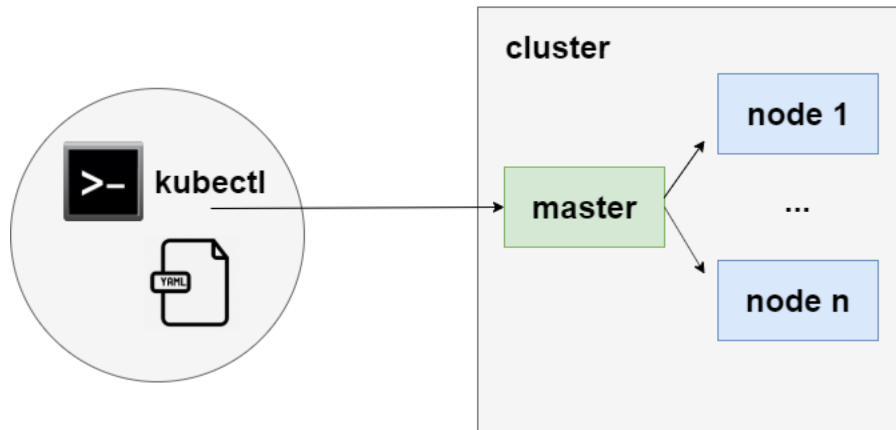

Figura 3-13: Arquitectura do kubernetes - mestre/nós

O mestre possui vários componentes que constituem a lógica do *cluster*. Os componentes mais relevantes são os seguintes:

- ***etcd***: é uma base de dados de chave-valor, que armazena os dados de configuração do cluster e o estado do cluster;
- ***api server***: é um componente do plano de controlo do *kubernetes* que expõe a API do *kubernetes*. O servidor da API é o *frontend* do plano de controlo do *kubernetes*. A principal implementação de um servidor de API do *kubernetes* é o *api server*, que foi desenvolvido para ser escalado horizontalmente, ou seja, é dimensionado com a implantação de mais instâncias. [14]
- ***schedule***: tem a responsabilidade de executar as tarefas que estão agendadas, como por exemplo a execução de *containers* nos vários nós, tendo como base a disponibilidade de recursos.

  Ele procura *pods* que ainda não estão agendados e atribui a um nó.
- ***controller manager***: tem como objectivo primário a execução dos controladores. Logicamente, cada controlador é um processo separado, mas para reduzir a complexidade, todos eles são compilados para um único binário e executados todos eles no mesmo processo [14].

Existem vários tipos de controladores, sendo os mais relevantes os seguintes:

- **Controlador de nó**: responsável por perceber e responder quando os nós são desactivados [14].
- **Controlador de replicação**: responsável por manter o número correcto de *pods* para cada objecto do controlador de replicação no sistema [14].
- **Controlador de *endpoints***: tem como responsabilidade popular os objectos de *endpoint* [14].
- **Conta de serviço e controladores de *tokens***: crie contas padrão e *tokens* de acesso à *API* para novos *namespaces* [14].

Os componentes dos nós têm como principal finalidade o manuseamento de novos *pods*, garantindo que eles sejam iniciados e assegurando a comunicação entre os vários *pods*.

Embora os componentes específicos possam diferir dependendo da configuração de cada *cluster*, a lista seguinte refere os principais componentes fornecidos pelo *kubernetes* para os nós.

- ***kube proxy***: é um *proxy* de rede que é executado em cada nó, permitindo assim a comunicação entre os vários *pods*.

  Em várias implementações de *proxy*, o modo padrão configura o *proxy* com *iptables* onde estão os mapeamentos de IP entre o *IP* (estático) do *proxy* e os *IPs* dos *pods* [13].

  O *kube proxy* utiliza a camada de filtragem de pacotes do sistema operativo, caso esteja disponível. Caso contrário, encaminha o próprio tráfego.
- ***kubelet***: é agente que é executado em cada nó do *cluster*. Ele garante que os *containeres* permaneçam em execução em um *pod*.

  O *kubelet* utiliza um conjunto de *PodSpecs* que são fornecidos por vários mecanismos e garante que os *containeres* descritos nesses *PodSpecs* permaneçam em funcionamento corretamente. O *kubelet* não faz a gestão de *containeres* que não tenham sido criados pelo *kubernetes* [14].
- ***container runtime***: é o software responsável pela execução de *containeres*.

  O *kubernetes* suporta vários tempos de execução de *containeres*, sendo o mais

relevante o *docker*.

Os *pods* são as menores unidades de aplicações no *kubernetes*. É um grupo de um ou mais *containers* (sendo os mais utilizados os *containers docker*), com armazenamento/rede partilhados, e uma especificação que indica como deve ser executado (figura 3-14).

O contexto que é compartilhado de um *pod* é um conjunto de *namespaces*, *cgroups* e outras funcionalidades de isolamento de um sistema operativo.

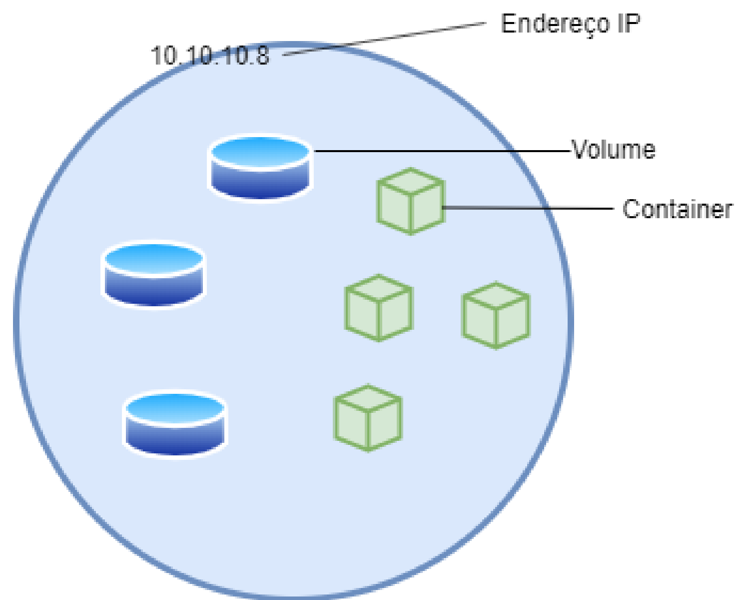

Figura 3-14: *Pod*

O fluxo da criação num *pod* em um *cluster* de *Kubernetes* pode ser observado na figura 3-15 e explicado de seguida.

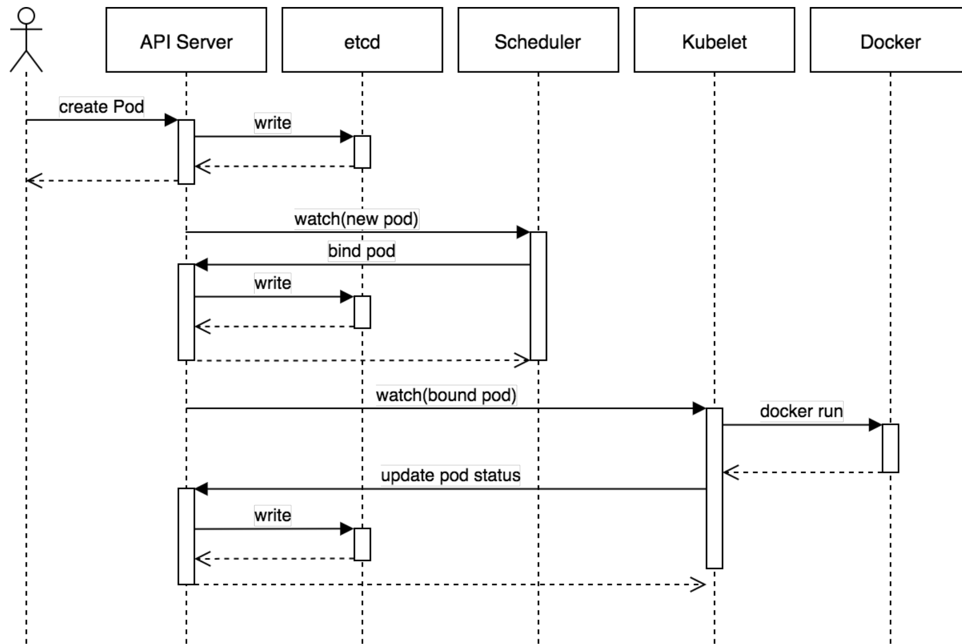

Figura 3-15: Fluxo de um *pod* (adaptado de [15])

De seguida são enumerados por ordem temporal o fluxo de criação de um *pod*.

1. O *kubectl* envia o pedido para o servidor de *API*.

2. O servidor de *API* valida o pedido e grava o mesmo no *etcd*.

3. O *etcd* notifica novamente o servidor *API*.

4. O servidor de *API* chama o *schedule*.

5. O *schedule* decide onde executar o *pod* e devolve essa informação ao servidor de *API*.

6. O servidor de *API* grava essa informação no *etcd*.

7. O *etcd* notifica novamente o servidor de *API*.

8. O servidor de *API* envia o pedido de criação do *pod* ao *Kubelet* do nó correspondente.

9. O *Kubelet* envia o pedido para a *API* do *soquete* do *Docker* para criar o *container* através do *daemon* do *Docker*.

10. O *Kubelet* envia a actualização do estado do *pod* para o servidor de *API*.

11. O servidor da *API* grava o novo estado do *pod* no *etcd*.

A nível de rede, qualquer *pod* recebe um endereço *IP* único. Cada *container* dentro de um *pod* partilha o *namespace* da rede, incluindo o *IP* e as portas de rede e podem comunicar entre eles utilizando o *localhost*, quando a comunicação é para fora do *pod* há coordenação para eles utilizarem os recursos de rede partilhados [20].

A figura 3-16 demonstra como funciona o acesso a um serviço que é executado dentro do *pod* de um determinado nó do *cluster Kubernetes*.

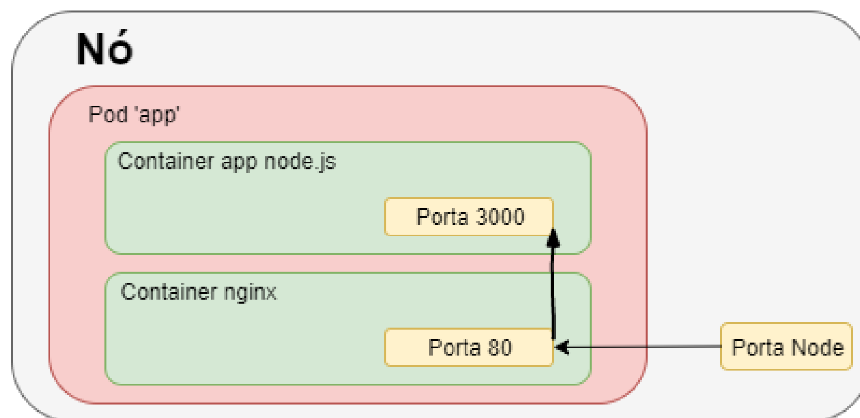

Figura 3-16: Rede em um *pod*

Um *pod* pode especificar um conjunto de volumes partilhados. Todos os *containers* que fazem parte desse *pod* podem aceder aos volumes partilhados, permitindo assim que os *containers* partilhem dados (figura 3-17). Eles também permitem persistência de dados, para o caso de um *container* necessite de ser reiniciado.

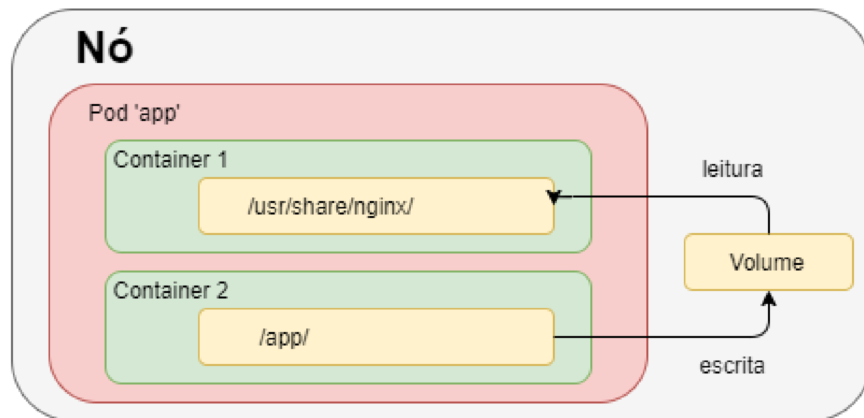

Figura 3-17: Armazenamento em um *pod*

O armazenamento dentro do *pod* é efémero e desaparece caso o *pod* também desapareça. Caso seja importante que os dados sobrevivam caso o *pod* desapareça, é necessário implementar um volume. O *kubernetes* utiliza os seus volumes separados.

Existem muitos tipos de volumes, principalmente para ambientes na *cloud*, sendo vários sistemas de arquivos e até repositórios *Git*. Quando combinamos as classes de armazenamento e os provedores de *cloud*, podemos obter resultados bastante interessantes (figura 3-18).

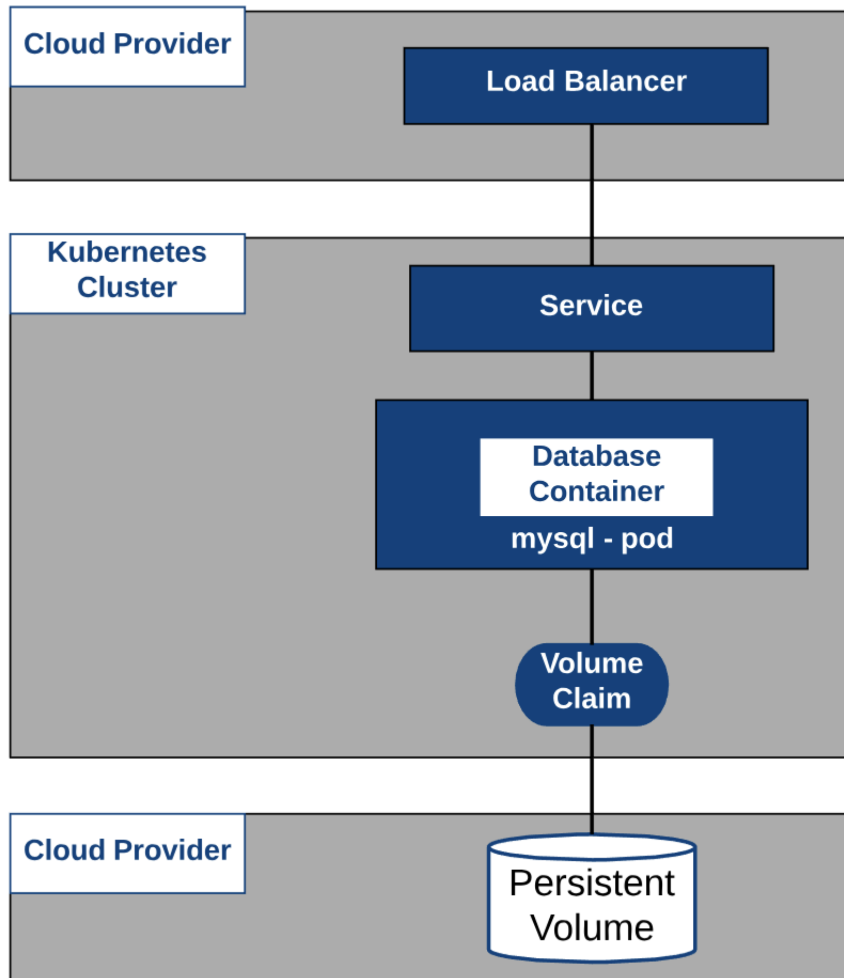

Figura 3-18: Armazenamento persistente [16]

Os serviços são utilizados para expor *pods*. Com o *kubernetes* não é necessário modificar a aplicação para utilizar um mecanismo de descoberta de serviço. O *kubernetes* fornece aos *pods* endereços *IP* e um único nome *DNS* para um *pod* ou um conjunto de *pods*, e pode balancear a carga entre esse conjunto.

A figura 3-19 demonstra a criação de um serviço em *Kubernetes*.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: ipv-estgv-service
  namespace: backend
spec:
  selector:
   app: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9677
```

Figura 3-19: Exemplo da criação de um serviço

Os *ReplicaSet* são utilizados para definir a quantidade de instâncias de um *pod* devem estar em execução em um determinado *cluster* de *Kubernetes*.

Cada *ReplicaSet* tem uma configuração definida com os dados dos novos *pods*, esses *pods* são identificados através de rótulos (campo metadata.ownerReferences).

Um *ReplicaSet* garante que um número específico de réplicas de *pods* estejam em execução a qualquer momento.

Os *ReplicaSet* podem ser dimensionados manualmente, ou automaticamente por meio de escalonamento horizontal definindo na configuração esse número (podemos definir ainda limites para a utilização de memória, *cpu* para cada recurso).

Os *Deployments* são utilizados para gerir um conjunto dimensionado de *pods* em várias versões das aplicações. Além disso fornecem as funcionalidades dos *ReplicaSet* através da utilização dos mesmos.

Permitem ainda o lançamento de novas versões, como por exemplo a alteração de lançamentos de aplicações, disponibilizam um histórico de lançamentos. Permitem estratégias de recriar *pods* e de RollingUpdate.

Com a estratégia de RollingUpdate uma aplicação fica sempre disponível, por exemplo uma aplicação que está a ser executada em um *pod* e recebe um *update*, é lançado um *pod* novo, e o antigo apenas é apagado quando o novo estiver pronto, garantindo com isto a disponibilidade das aplicações (figura 3-20).

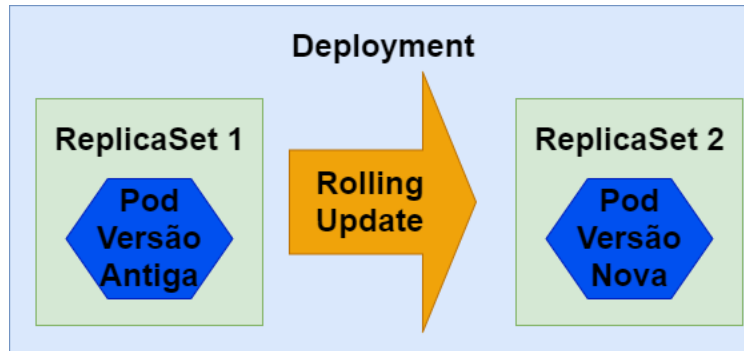

Figura 3-20: *RollingUpdate*

Resumidamente, o recriar simplesmente altera o *ReplicaSet* para a nova versão, enquanto que com o *RollingUpdate* é adicionado um novo *ReplicaSet*, em que os dois são dimensionados ao mesmo tempo em sentido inverso.

A figura 3-21 demonstra a utilização de vários *ReplicaSet* no contexto de um *Deployment*.

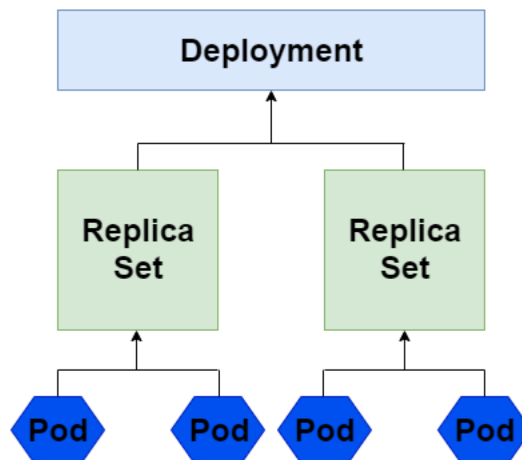

Figura 3-21: *ReplicaSet e Deployment*

Um *namespace* pode ser pensado como um *cluster* virtual. É possível ter um *cluster* físico que contém vários *clusters* virtuais segregados por *namespaces* (figura 3-22). Cada *cluster* virtual é totalmente isolado dos restantes *clusters* virtuais e apenas pode existir comunicação entre eles através de interfaces públicas. [37]

Os nós e os volumes persistentes não vivem dentro de um *namespace*. O *kubernetes* pode agendar *pods* de *namespaces* diferentes para serem executados dentro do mesmo nó. Da mesmo forma, os *pods* de *namespaces* diferentes podem utilizar o mesmo armazenamento persistente [37].

Com os *namespaces* é possível definir políticas de rede assim como cotas de recursos para garantir o acesso e a distribuição adequada dos recursos físicos no *cluster*.

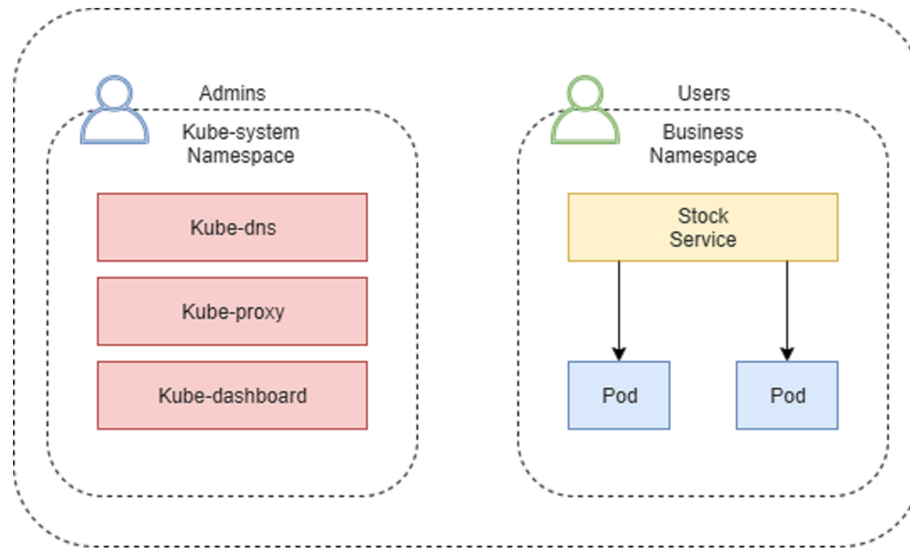

Figura 3-22: *Namespaces*

3.2.3 Microsoft Windows Azure

A Plataforma *Microsoft Windows Azure* é a plataforma de computação em nuvem da *Microsoft*, tendo um conjunto cada vez maior de serviços para ajudarem as organizações a enfrentarem os desafios de negócios. Permite criar, gerir e implantar aplicações em uma enorme rede global utilizando várias ferramentas e estruturas.

O *Windows Azure* consiste em vários serviços *on demand*. A figura 3-23 ilustra os vários componentes e recursos do *Windows Azure*. Todos esses componentes e serviços são agrupados em três principais componentes, serviços de aplicações, serviços de dados e computação.

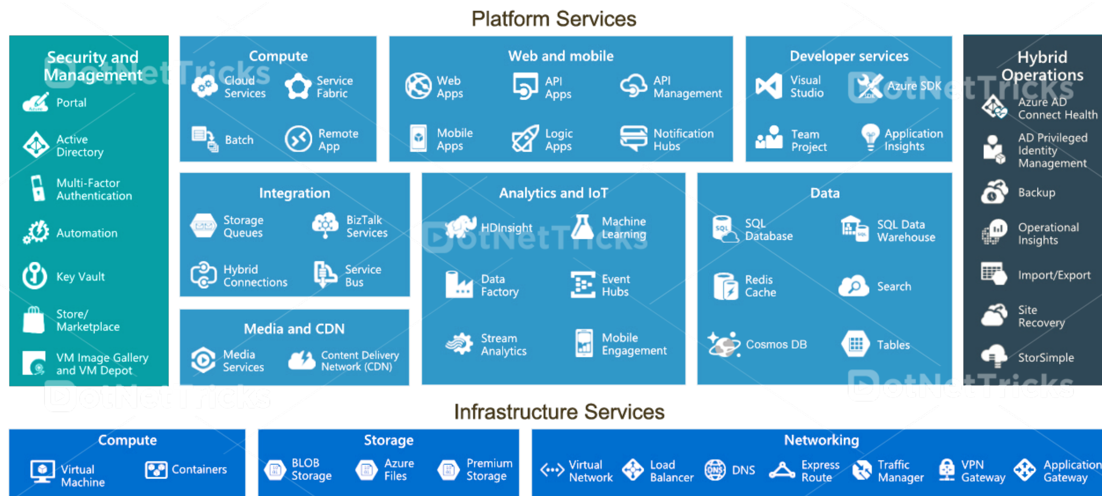

Figura 3-23: Serviços do *Microsoft Azure* (adaptado de [33])

3.2.4 Azure Kubernetes Service

O *Azure Kubernetes Service (AKS)* é um serviço de orquestração de *containers*, baseado no sistema *open source Kubernetes*, que está disponível na *cloud Microsoft Azure*.

O *AKS* é um *cluster* de *Kubernetes* em que as máquinas (mestres e nós) são geridas pelo *provider*, facilitando assim a implantação e gestão de aplicações em *containers*. Podemos pensar no *AKS* como um *Kubernetes* sem servidores, e com ferramentas para CI/CD, tendo como principais vantagens a flexibilidade e a automação [4].

O *AKS* configura os mestres e os nós do *Kubernetes* durante o processo de implantação, e permite a gestão de outras tarefas como a integração com o *Azure Active Directory* (figura 3-24) ou permitir a configuração de rotas HTTP.

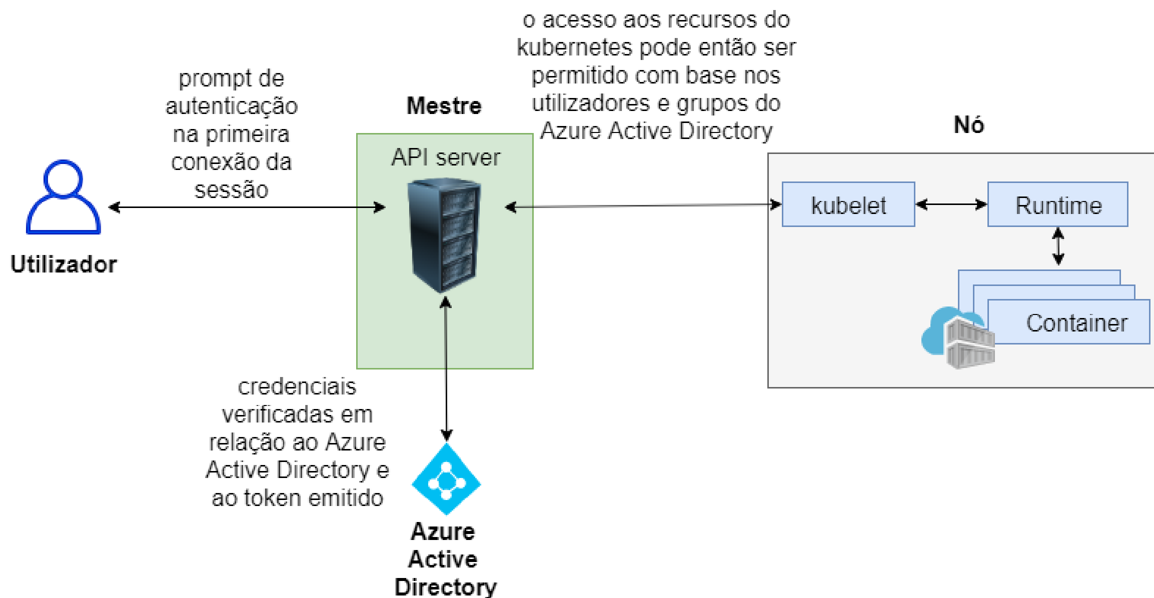

Figura 3-24: *AKS* com *Azure Active Directory*

O *AKS* permite aumentar ou diminuir facilmente o número de nós, para facilitar a gestão de carga nos recursos. Uma das críticas que faço é que o *AKS* actualmente apenas permite lançar novos nós com os mesmos recursos dos existentes. Por exemplo, se na configuração inicial os nós forem de 16GB de RAM, apenas suporta novos nós também com 16GM de RAM, o que nem sempre é necessário escalar logo de forma tão agressiva.

Os utilizadores podem aceder ao *AKS* através do portal de gestão do serviço, ou através de CLI.

3.2.5 Container Registry

Um *container registry* é uma colecção de repositórios que permitem o armazenamento de imagens de *containers*. Uma imagem de *container* é um arquivo composto por várias camadas que permitem a execução de aplicações em uma única instância. O armazenamento de todas as imagens de uma aplicação em um *container registry* permite aos programadores a confirmação, identificação e a extracção de imagens quando necessário.

Os *container registry* além de armazenarem imagens, podem ainda implementar regras de acesso. Têm ainda a opção de serem públicos ou privados.

O *Docker Hub* é o maior repositório de imagens de *containers* do mundo, com bastantes variedades de conteúdo, incluindo aplicações de programadores das comunidades de *containers*, projectos de código aberto e fornecedores independentes de *software* que constroem e distribuem o seu código em *containers* [9].

O *Docker Hub* permitem acesso grátis a repositórios públicos para armazenarem e compartilharem imagens ou podem escolher um plano de assinatura para repositórios privados [9].

A figura 3-25 demonstra um exemplo da utilização de um *Container Registry* com a utilização de *pipelines* de *CI/CD*, em que o *pipeline* de *CI* cria uma imagem *Docker* e armazena a mesma no *Container Registry*, e posteriormente o *pipeline* de *CD* acede ao *Container Registry* para obter acesso à imagem criada anteriormente.

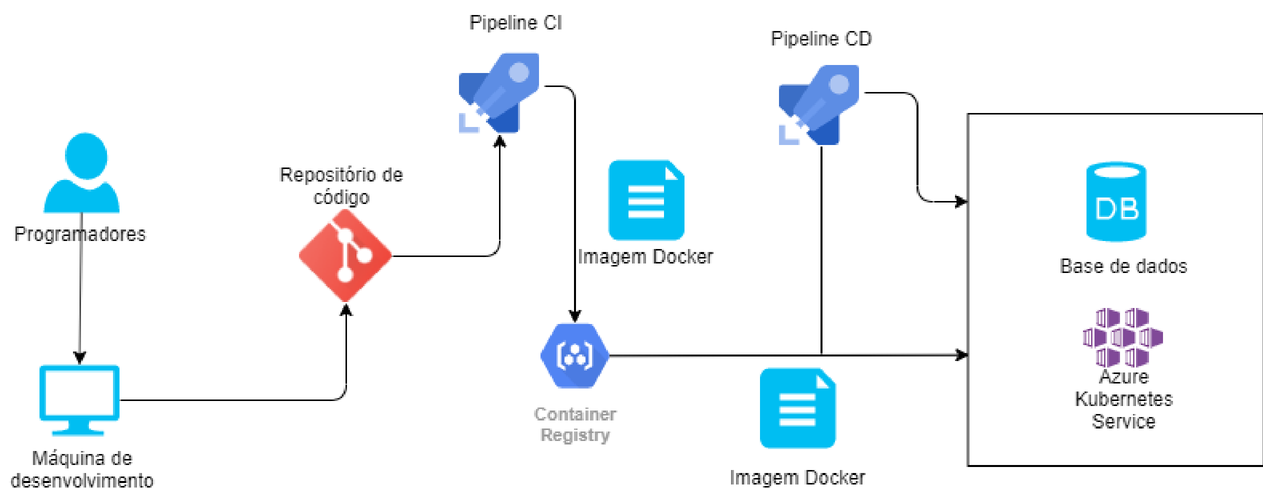

Figura 3-25: Exemplo de ambiente com *Container Registry*

3.2.6 Azure Devops

O *Azure DevOps* foi lançado pela *Microsoft*, e é um conjunto de serviços alojados na *cloud*, dos quais fazem parte testes, *pipelines CI/CD*, quadros, dentre outros.

Os *Azure Repos* são os repositórios no *Azure Devops* são "ilimitados". Eles funcionam como repositórios *Git*, permitindo assim a colaboração através de *pull requests* e um avançado sistema de gestão de ficheiros [3].

Nos *Azure Artifacts* é possível alojar, partilhar ou criar pacotes de artefactos da equipa. Também é possível a adição de artefactos aos *pipelines* de *CI/CD*, controlo de versões ou testes [3].

O *Azure Test Plans* é uma autêntica suíte de testes e lançamento de versões de *software* que conta com várias ferramentas de testes. Conta ainda com um sistema de *logs* dos testes realizados [3].

O *Azure Boards* permite o acompanhamento do trabalho, pendências, relatórios personalizados, entre outros. É possível o rastreamento de trabalho entre várias equipas.

O *Azure Pipelines* é uma área de desenvolvimento propriamente dita. *Build*, testes e *deploy* com *CI/CD*.

É possível trabalhar com várias linguagens e conectar-se por exemplo a *GitHub* ou ao *Bitbucket*.

Tem vários tipos de servidores como compilações com *Windows*, *Linux*, *MacOS*, entre outros.

3.2.7 Terraform

O *Terraform* é uma ferramenta *open source* utilizada para o provisionamento de infraestruturas, que foi criada pela empresa *HashiCorp* [25].

O *Terraform* permite que a infraestrutura seja escrita em código em uma linguagem simples e legível chamada *HCL* (*HashiCorp Configuration Language*). O *Terraform CLI* lê os arquivos de configuração e fornece um plano de execução das alterações, que podem ser revistas por segurança e depois aplicadas e provisionadas (figura 3-26) [25].

Com o *Terraform* é possível a gestão da infra-estrutura em uma variedade de prove-

dores de *cloud*, públicas e privadas.

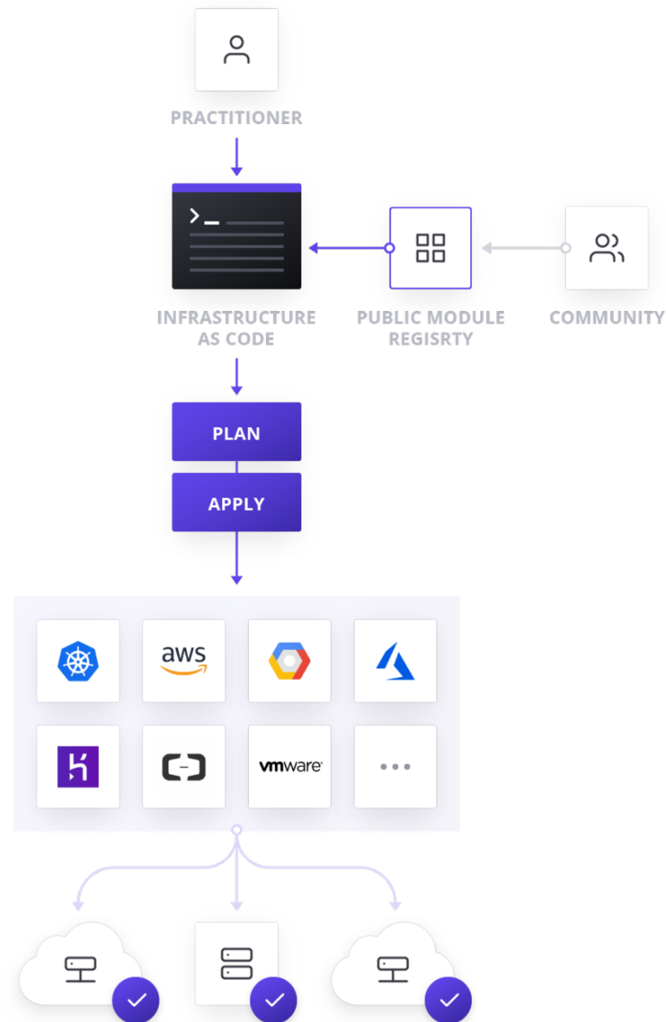

Figura 3-26: Arquitectura *CLI* do *Terraform* [25]

3.3 Comparação de soluções

Nesta seção, são apresentadas e avaliadas as várias soluções alternativas de orquestração de *container* e avaliadas. Finalmente, são comparadas as várias soluções de

orquestração apresentadas.

3.3.1 Docker Swarm

O *Docker Swarm* é uma funcionalidade quem vem nativa do *Docker*, disponível no seu *Engine*, que permite a orquestração de *clusters Docker*. Basicamente consiste em múltiplos *Dockers Hosts* que são executados em modo *swarm* e funcionam como gestores (gerindo a associação e delegação) e como trabalhadores (executando serviços de *swarm*). Um nó pode ser apenas gestor, apenas trabalhador ou executar ambas as funções [24].

Na criação de um serviço é configurado a estado do *cluster*, como o número de réplicas, recursos de rede, portas, etc. A partir daqui o *Docker* irá tentar manter a configuração em execução. Por exemplo, caso um nó fique indisponível, serão agendadas as tarefas desse nó em outros nós.

Uma das grandes vantagens do *Docker swarm* é a possibilidade de modificação das configurações dos serviços, sem a necessidade de reiniciar o serviço manualmente, uma vez que o *Docker* actualiza as configurações, interrompendo de seguida as tarefas dos serviços com as configurações desactualizadas e cria novas com as novas configurações [24].

3.3.2 Nomad

O *Nomad* é um orquestrador para gestão de máquinas num *cluster* e executar aplicações dentro dessas máquinas [11].

É enviado um ficheiro binário com tarefas de agendamento internas e tarefas de gestão dos recursos do *cluster*, não sendo necessários outros serviços para armazenar o estado do *cluster* e/ou aplicação. É ainda distribuído e suporta nativamente *failover*. Não suporta gestão secreta nem descoberta de serviços, sendo necessário para isso ferramentas externas como o *Vault* ou o *Consul*, criando assim um desafio acrescido na sua configuração e monitorização [11].

O *Nomad* é suportado em sistemas operativos *windows*, *linux* e *macOS*.

No entanto o *Nomad*, foi desenhado inicialmente para gestão de *clusters* e nãp pensado para orquestração de *containers*.

3.3.3 DC/OS

Tem como base o *kernel* do *Apache Mesos*, sendo um sistema distribuído e *open source*.

O *DC/OS* consegue gerir várias máquinas a partir de uma única *interface*, gerir *containers* e fornece recursos de rede, descoberta de serviços, entre outros [31].

O *DC/OS* tem a versão grátis e uma versão empresarial paga, sendo que a versão grátis é bastante limitada, faltando recursos como gestão secreta ou autenticação.

3.3.4 Comparação de sistemas de orquestração

Após uma comparação de sistemas de orquestração de *containers* (tabela 3.1), *kubernetes* foi visto como o melhor para as necessidades deste projecto. O *DC/OS* chegou perto, mas a comunidade é menor e para oferecer os mesmos serviços que o *kubernetes* necessita de assinatura paga, além de que é um sistema mais focado em *big data*.

	Kubernetes	**Docker Swarm**	**Nomad**	**DC/OS**
Open source	X	X	X	X
Alta disponibilidade	X	X	X	X
Descoberta de serviço	X	X	-	X
Gestão secreta	X	X	-	-/X
Balanceamento de carga	X	X	-	X
Suporta volumes	X	X	-	X
Atualizações contínuas	X	X	X	X
Self-healing	X	X	X	X
Auto scaling	X	-	-	X

Tabela 3.1: Comparação de sistemas de orquestração de *containers* (X é um recurso suportado)

Esta página foi propositadamente deixada em branco.

Capítulo 4

Implementação

Neste capítulo, será descrito o trabalho realizado ao longo do estágio curricular. Uma vez que existem dados confidenciais, não será possível a demonstração de dados nem código que seja considerado confidencial.

4.1 Fluxos de implantação

No desenvolvimento de *software*, uma versão é um novo *software* ou um *software* existente que foi modificado, e o processo da criação do mesmo.

A maneira como esse *software* é disponibilizado em cada um dos vários ambientes (testes, produção, ...) é uma das partes mais importantes da engenharia de *software*, uma vez que é cada vez mais arriscado a existência de lançamentos incompletos. No entanto grande parte dos problemas são resultado de alterações no *software* ou no ambiente.

Para evitar atrasos dispendiosos e manter as operações das organizações a funcionar, é necessário investir em um processo robusto de gestão de lançamentos de versões de *software*.

Por conseguinte, a proposta que está neste momento em funcionamento na organização é a da figura 4-1.

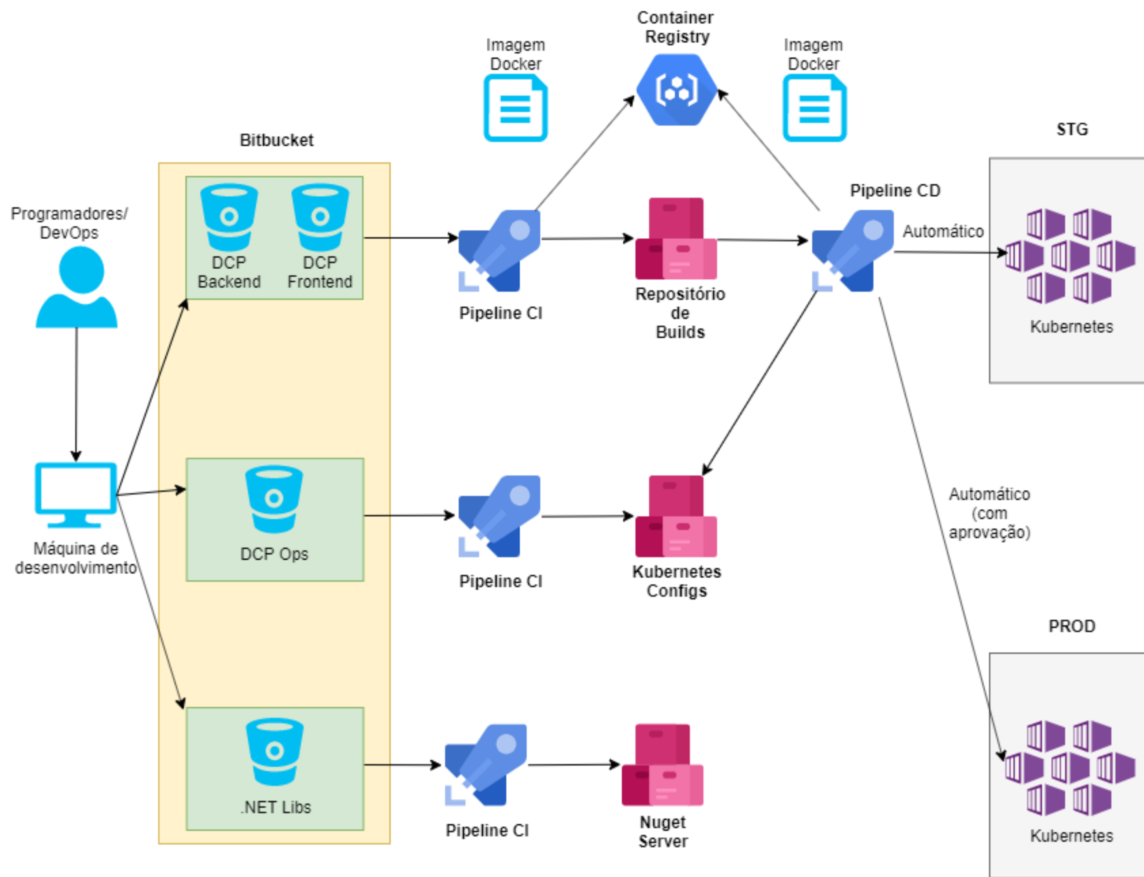

Figura 4-1: Desenho dos fluxos de implantação

A nível de fluxos de *CI*, decidiu-se partir em três, uma vez que eram independentes entre si ao nível das entregas de *software*.

Neste momento existem dois ambientes, *STG* que é onde a equipa de *QA* faz os testes e *PROD*, que é o ambiente de produção. Existe o ambiente de desenvolvimento, mas é local e não é da responsabilidade dos *pipelines*.

A gestão de versões de código e a maneira como o sistema de *branches*, *pull requests*, etc é pensado é crucial principalmente em desenvolvimento *agile*. Vou dar o exemplo de um problema que aconteceu no inicio do estágio. Foi feito um *branch* para o desenvolvimento A, após o desenvolvimento foi feito o *pull request* e após o code review pelos membros da equipa foi feito o *merge* para *master* e o *branch* fechado. De seguida foi feito o mesmo procedimento para o desenvolvimento B. Sendo o desenvolvimento

B algo mais urgente teria que ser feita uma *release* desse desenvolvimento (na altura o desenvolvimento A ainda não estava aprovado pela equipa de *QA*). Com esta forma de gerir as versões ao colocar o desenvolvimento B em produção obrigatóriamente seria colocado o A, uma vez que quando foi feito o *branch* B, já o A estava em *master*. Normalmente este tipo de problemas não acontece em metodologias não *agile*, como por exemplo no modelo em cascata, uma vez que normalmente é colocado o produto final em produção apenas no final dos desenvolvimentos. No modelo *agile*, são colocados desenvolvimentos em produção normalmente em cada *sprint*. O modelo de gestão de versões proposto e que está neste momento a ser utilizado nas equipas é o seguinte:

- Todos os *branches* são feitos apartir do *master*.
- Após o *pull request* é feito o *merge* para um *branch* de *staging* mas o *branch* do desenvolvimento continua aberto.
- Existe um *branch* de *release* de produção em que são feitos os *merges* dos *branches* dos desenvolvimentos que irão entrar em produção.
- As *releases* para os ambientes não produção são automáticas.
- As *releases* para o ambiente de produção são automáticas mas com aprovação.
- Após a *release* de produção é feito o *merge* do *branch release* produção para *master*.

4.2 Conteinerização dos Micro-serviços

É possível a criação de *Docker images* através de um processo iterativo (executar comandos isolados em um *CLI*). Mas a melhor opção é colocar esses passos em *Dockerfiles*, que são arquivos de compilação simples que contêm os passos para a criação de *Docker images*. Algumas dessas vantagens é a possível gestão de versões, previsibilidade uma vez que ajudam a mitigar erros humanos, entre outras.

No estágio a primeira abordagem sugerida para criar as *Docker images* foi seguindo a tradicional de um *Dockerfile* por cada micro-serviço (figura 4-2), sendo que cada

um dos *Dockerfiles* era criado na raiz de cada um dos projectos do respectivo micro-serviço.

```
FROM microsoft/dotnet:2.2-sdk AS build-env
WORKDIR /source

# Copy csproj and restore as distinct layers
COPY src/SonaeSF.Integrations.Dropbox.FileDiscovery SonaeSF.Integrations.Dropbox.FileDiscovery
COPY src/SonaeSF.Core SonaeSF.Core
COPY src/SonaeSF.Integrations.Dropbox SonaeSF.Integrations.Dropbox
RUN cd SonaeSF.Integrations.Dropbox.FileDiscovery && dotnet restore

# Copy everything else and build
COPY . ./
RUN cd SonaeSF.Integrations.Dropbox.FileDiscovery && dotnet publish -c Release -o /app/

# Build runtime image
FROM microsoft/dotnet:2.1-aspnetcore-runtime
WORKDIR /app
COPY --from=build-env /app .

ENTRYPOINT ["dotnet", "SonaeSF.Integrations.Dropbox.FileDiscovery.dll", "-v", "-t 300", "-p 60"]
```

Figura 4-2: *Dockerfile* primeira abordagem

Apesar de funcionar, esta abordagem não é a melhor, uma vez que existe imensa redundância de código, além de que nem todos os programadores percebem de *Docker*, sendo que nesses casos tinham que abrir um *ticket* para a equipa de *DevOps* por cada novo serviço criado.

Devido a esses problemas, foi seguida uma abordagem diferente (figura 4-4), a de um único *Dockerfile* presente na raiz (figura 4-3), o ficheiro passa a estar presente apenas na pasta *src* e consegue ler as configurações de cada um dos projectos que estão contidos na solução e não ao nível do projecto, e utilizando variáveis de ambiente para controlar a criação de cada uma das *Docker images*.

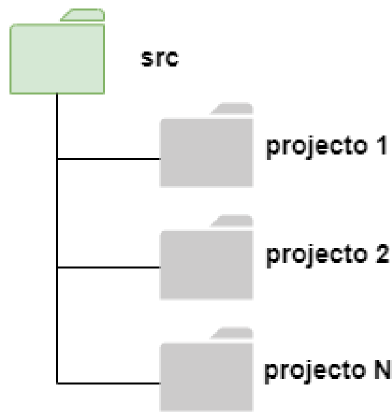

Figura 4-3: Árvore da solução

```
FROM microsoft/dotnet:2.2-sdk AS builder
WORKDIR /source

ARG FULLY_QUALIFIED_APP_NAME
ARG FEED_URL
ARG PAT

COPY . /source

# download and install latest credential provider. Not required after https://github.com/dotnet/dotnet-docker/issues/878
RUN wget -qO- https://raw.githubusercontent.com/Microsoft/artifacts-credprovider/master/helpers/installcredprovider.sh | bash

# Environment variable to enable seesion token cache. More on this here: https://github.com/Microsoft/artifacts-credprovider#help
ENV NUGET_CREDENTIALPROVIDER_SESSIONTOKENCACHE_ENABLED true
ENV DOTNET_SYSTEM_NET_HTTP_USESOCKETSHTTPHANDLER 0
ENV VSS_NUGET_EXTERNAL_FEED_ENDPOINTS {\"endpointCredentials\": [{\"endpoint\":\"${FEED_URL}\", \"username\":\"build\",
\"password\":\"${PAT}\"}]}

# Environment variable for adding endpoint credentials. More on this here: https://github.com/Microsoft/artifacts-credprovider#help
# Add "FEED_URL" AND "PAT" using --build-arg in docker build step. "endpointCredentials" field is an array, you can add multiple
#endpoint configurations.
# Make sure that you *do not* hard code the "PAT" here. That is a sensitive information and must not be checked in.
RUN dotnet restore src/$FULLY_QUALIFIED_APP_NAME/$FULLY_QUALIFIED_APP_NAME.csproj --source $FEED_URL
--source https://api.nuget.org/v3/index.json

RUN dotnet publish src/$FULLY_QUALIFIED_APP_NAME --output /app/

# Stage 2
FROM microsoft/dotnet:2.2-aspnetcore-runtime AS runner
WORKDIR /app

COPY --from=builder /app .
```

Figura 4-4: *Dockerfile* segunda abordagem

Para o projecto de *Front-end* uma vez que apenas é composto por um único micro-serviço, um único *Dockerfile* funciona perfeitamente (figura 4-5).

```
FROM node:12-alpine

# Create app directory
RUN mkdir -p /app
WORKDIR /app

# Install app dependencies
COPY package.json /app
COPY yarn.lock /app
RUN yarn install

# Set environment variables
ENV NODE_ENV production
ENV HOST 0.0.0.0
ENV PORT 3000

# Bundle app source
COPY . /app
RUN yarn build

# Clear the cache
RUN yarn cache clean

EXPOSE 3000
CMD [ "yarn", "start" ]
```

Figura 4-5: *Dockerfile* serviço de *Front-end*

Para facilitar a utilização de variáveis de ambiente nas máquinas de desenvolvimento, pode ser criado um ficheiro .env (*snippet* 4.1) com essas variáveis.

```
NODE_ENV=development
DCP_API_URL=http://192.168.56.1:5000
HOST=0.0.0.0
PORT=3000
AZURE_AD_TENANT_ID=10c1e88b-309d-4d2c-aed2-582dca3843bd
AZURE_AD_CLIENT_ID=3a82f4f6-9cf0-4272-87ec-91d458d59488
AZURE_AD_CLIENT_SECRET=cbba576a-1b4a-4bb9-8d62-4ebde2999d79
```

Snippet 4.1: Configuração de variáveis de ambiente

Na empresa utilizam *Git*, que é um sistema de controlo de versões distribuído. Para não existir risco de enviar esses ficheiros para os repositórios de código, basta criar um ficheiro *.gitignore* e adicionar a entrada do *snippet* 4.2 ao ficheiro.

```
# Sensible information
.env
```

Snippet 4.2: Configuração do ficheiro *.gitignore*

Ambiente de desenvolvimento

As máquinas de desenvolvimento são VMs com o sistema operativo *Ubuntu Server*. Para criar o ambiente de desenvolvimento local foi utilizado o *Docker Compose*. É uma ferramenta para definir e executar aplicações Docker em vários *containers*. Num ficheiro são configurados os vários serviços *Docker*, e de seguida com um único comando são criados e iniciados todos os serviços configurados.

```
sonae-sf-core-kafka-rest-proxy:
    container_name: sonae-sf-core-kafka-rest-proxy
    hostname: sonae-sf-core-kafka-rest-proxy
    image: confluentinc/cp-kafka-rest
    ports:
        - "8082:8082"
    environment:
        # KAFKA_REST_ZOOKEEPER_CONNECT: zookeeper:2181
        KAFKA_REST_LISTENERS: http://0.0.0.0:8082/
        KAFKA_REST_HOST_NAME: sonae-sf-core-kafka-rest-proxy
        KAFKA_REST_BOOTSTRAP_SERVERS: PLAINTEXT://sonae-sf-core-kafka:19092
    depends_on:
        - sonae-sf-core-zookeeper
        - sonae-sf-core-kafka
```

Figura 4-6: Exemplo de serviço no *Docker Compose*

Como é possível ver na figura 4-6, é possível especificar os serviços do qual o serviço em questão depende. Nesse caso ao executar apenas o serviço da figura 4-7, ele irá criar as *Docker images* e executar os *containers* dos serviços *"sonae-sf-core-zookeeper"* e do *"sonae-sf-core-kafka"*.

```
brunobold@ubuntuserver:~/share/dcp-backend$ docker-compose up -d sonae-sf-core-kafka-rest-proxy
WARNING: The ARTIFACT_FEED_URL variable is not set. Defaulting to a blank string.
WARNING: The ARTIFACT_PAT variable is not set. Defaulting to a blank string.
sonae-sf-core-zookeeper is up-to-date
Starting sonae-sf-core-kafka ... done
Starting sonae-sf-core-kafka-rest-proxy ... done
```

Figura 4-7: Executar serviço no *Docker Compose*

Nas figuras 4-8 e 4-9 é possível observar as *Docker images* criadas e os *containers* em execução, respectivamente.

```
brunobold@ubuntuserver:~/share/dcp-backend$ docker images
REPOSITORY                TAG                 IMAGE ID            CREATED             SIZE
confluentinc/cp-kafka-rest   latest              27066ec8dcc2        2 months ago        984MB
confluentinc/cp-kafka     5.3.0               f9936a4fcbce        3 months ago        589MB
zookeeper                 3.4.9               3b83d9104a4c        2 years ago         129MB
```

Figura 4-8: *Docker images*

```
brunobold@ubuntuserver:~/share/dcp-backend$ docker ps -a
CONTAINER ID        IMAGE                          COMMAND                  CREATED
d46a94e34671        confluentinc/cp-kafka-rest     "/etc/confluent/dock…"   About a minute ago
xy
5ca34f1d946a        confluentinc/cp-kafka:5.3.0    "/etc/confluent/dock…"   About a minute ago
19dec9a35d32        zookeeper:3.4.9                "/docker-entrypoint.…"   About a minute ago
```

Figura 4-9: *Docker Containers*

Para executar todos os serviços basta executar o comando do *snippet* 4.3.

```
docker-compose up
```

Snippet 4.3: Iniciar os serviços do *Docker Compose*

Para facilitar a limpeza de um ambiente e o lançamento de um novo, é recomendada a criação de um *Makefile* com os vários comandos em ficheiros *bash script* (*snippet* 4.4).

```
restore-all:
	ops/sbin/restore-all.sh
run-dev-environment:
	DRYRUN=$(DRYRUN) KAFKA_REST_PROXY=$(KAFKA_REST_PROXY) ./ops/
	   ↪ sbin/run-dev-environment.sh $(WORKSPACE_DIR)
add-sln:
	for p in src/*; do dotnet sln hipsters.sln add $p; done
```

Snippet 4.4: *Makefile* para criação de ambiente local

4.3 Container registry

A empresa tem a maioria dos sistemas alojados na *cloud Microsoft Azure*. O *Azure* disponibiliza um *container registry* privado, com suporte para *containers docker*, permite replicação geográfica (gerir um único registo em várias regiões), segurança integrada com o *Azure Active Directory*, *Docker Content Trust* e integração de rede virtual [6]. Assim sendo, não foi difícil a escolha pelo *container registry* do *Azure*.

O *container registry* é o único serviço que a equipa tem actualmente no *Microsoft Azure* que foi criado através de *az cli (Azure command-line interface)*, uma vez que é partilhado entre os vários ambientes e caso no futuro a infraestrutura seja migrada para outro provedor, a probabilidade de se utilizar *Docker Hub* é grande, então não existe a necessidade de automatizar a sua criação com *Terraform*.

Nas máquinas de desenvolvimento utilizamos sistemas operativos *ubuntu*, então a demonstração do que foi feito no estágio nas máquinas de desenvolvimento é feita tendo como base esse sistema.

É possível com um único comando instalar o *az cli* (*snippet* 4.5).

```
curl -sL https://aka.ms/InstallAzureCLIDeb | sudo bash
```

Snippet 4.5: Instalação do *az cli*

De seguida correr o comando do *snippet* 4.6 para obter acesso ao *Azure*.

```
az login
```

Snippet 4.6: Autenticação no *Microsoft Azure*

Para criar o *container registry* foi executado o comando do *snippet* 4.7.

```
az acr create --resource-group SonaeFashionGeneric --name
    ↪ SonaeFashionContainerRegistry --sku Premium
```

Snippet 4.7: Criação do *container registry*

Os *Resource Groups* no *Azure* servem para agrupar uma colecção de recursos, normalmente relacionados, que partilham o mesmo ciclo de vida. Com isso é mais fácil de gerir custos e gerir acessos [5]. Todos os recursos no *Azure* pertencem a um *Resource Group*. Neste caso não foi necessário criar um novo para o *container registry*, uma vez que foi utilizado um existente (SonaeFashionGeneric) que serve para os recursos que são partilhados entre ambientes ou aplicações.

Após a criação do *Container Registry* (figura 4-10) já é possível aceder ao mesmo, utilizando para isso o comando do *snippet* 4.8.

```
az acr login --name SonaeFashionContainerRegistry
```

Snippet 4.8: Autenticação no *container registry*

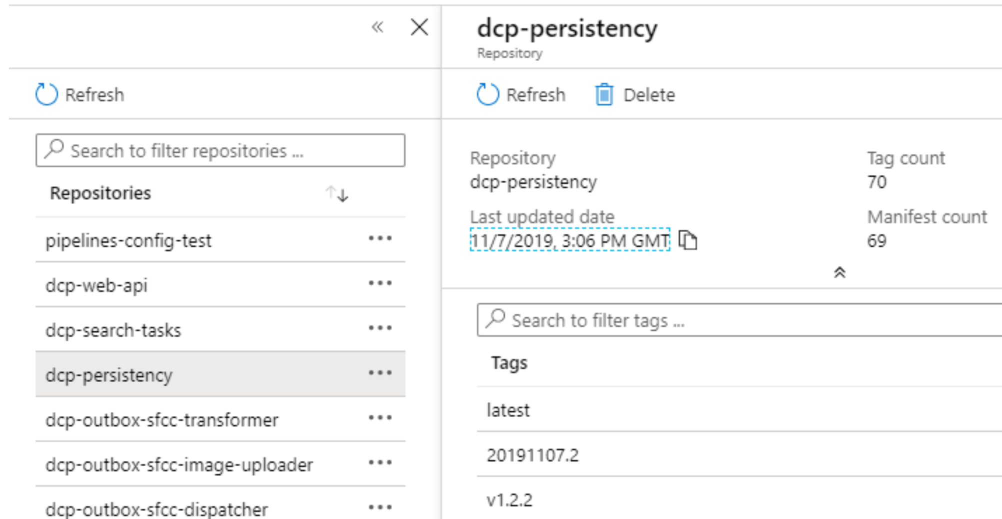

Figura 4-10: *Container Registry - Digital Catalog Platform*

E de seguida podemos enviar *containers images*, aceder entre outras tarefas. No caso deste projecto todas as *containers images* são criadas e geridas pelos *pipelines* de *CI/CD*.

4.4 Nuget Server

Apesar da criação e configuração de um *Nuget Server* não fazer parte dos objectivos iniciais, cedo se percebeu que era algo crucial para o projecto.

O *Nuget* é um repositório central de pacotes para as plataformas *.NET Framework* e *.NET Core*. O *Nuget.Server* é o nome desse repositório e é público.

Neste projecto surgiu a necessidade de criar pacotes *nuget* para serem disponibilizados para as várias empresas da organização (*Worten*, *Sonae MC*, etc), assim como para outros projectos da *Fashion*. Basicamente bibliotecas feitas pela nossa equipa que podem ser reutilizadas por outras equipas.

Daí surgiu a necessidade de existir um *Nuget server* privado. O que antes era necessário alojar por exemplo um sistema *BaGet* em um servidor, hoje já é possível configurar um *artifact* do *Azure DevOps* para servir como um *Nuget Server*.

Para isso basta após criar o *artifact* no *Azure DevOps* (figura 4-11), aceder às conexões e escolher a de *nuget*. De seguida gerar uma *API Key* no *Azure DevOps* e já é possível o acesso ao *artifact*.

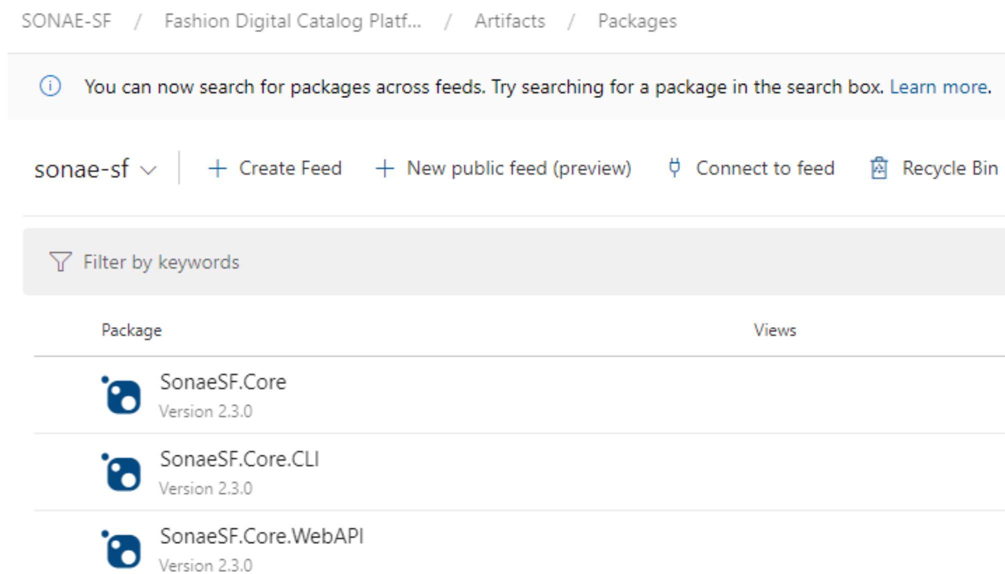

Figura 4-11: *Artifact - Nuget Server*

O *Visual Studio* é o *IDE* utilizado na equipa para desenvolvimento *.NET Core*, para aceder ao *Nuget Server*, basta seguir os seguintes passos:

1. No menu **Ferramentas**, selecione **Opções**.
2. Expanda o *NuGet Package Manager* e selecione **Origens** do pacote .
3. Selecione o sinal de mais verde no canto superior direito.
4. Na parte inferior da caixa de diálogo, insira o nome do *feed* e o *URL* obtido na conexão do *artifact*.
5. Selecione **Actualizar**.

Com isto a equipa de desenvolvimento já tem acesso aos pacotes *nuget* de forma segura como se pode ver na figura 4-12.

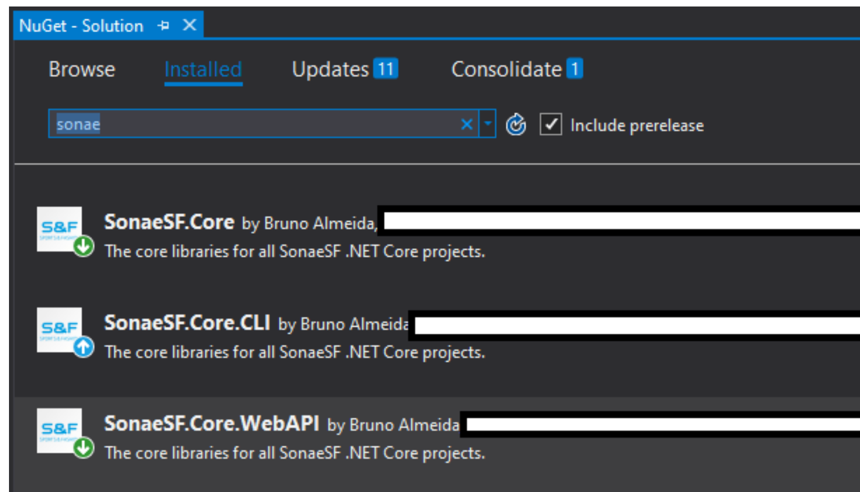

Figura 4-12: Pacotes *nuget* no VS

4.5 Azure Kubernetes Service

Terraform

Para a criação do *AKS* foi utilizado o *Terraform*, com o objectivo de automatizar o processo de criação de recursos para os vários ambientes e de controlo de versões.

Os ficheiros do *Terraform* fazem parte do repositório *DCP Ops*, e para a sua execução foi decidido a utilização de um *storage* do *Azure*, com o objectivo de ser mais fácil a execução e partilha para os vários ambientes, além do *storage* permitir controlo de acessos e integração com o *Azure DevOps* se necessário.

Primeiro executar o ficheiro de provider (figura 4-13).

```
provider "azurerm" {
  version         = "~> 1.29"
  subscription_id = ""
  tenant_id       = ""

  client_id     = "${var.ARM_CLIENT_ID}"
  client_secret = "${var.ARM_CLIENT_SECRET}"
}
```

Figura 4-13: *Povider do Terraform*

Após essa execução já é possível executar o ficheiro para a criação do *AKS* (figura 4-14).

```
resource "azurerm_kubernetes_cluster" "hip-aks" {
  name                = "hip-aks-${var.az_environ}"
  location            = "${var.az_region}"
  resource_group_name = "${azurerm_resource_group.hip-rg-aks.name}"
  dns_prefix          = "hip-aks-${var.az_environ}"

  agent_pool_profile {
    name            = "agentpool"
    count           = 1
    vm_size         = "Standard_B2s"
    os_type         = "Linux"
    os_disk_size_gb = 30
  }

  service_principal {
    client_id     = "${var.ARM_CLIENT_ID}"
    client_secret = "${var.ARM_CLIENT_SECRET}"
  }

  tags = {
    environment = "${var.az_environ}"
  }
}
```

Figura 4-14: *Terraform - AKS*

Após a criação do *AKS* é possível aceder ao *dashboard*, basta para isso instalar o *kubectl*, e de seguida executar os comandos do *snippet* 4.9.

```
az aks install-cli

# Para obter as credenciais de acesso ao cluster
az aks get-credentials --resource-group NOME_RESOURCE_GROUP --name
    ↪ NOME_AKS

# Para abrir o dashboard do Kubernetes
az aks browse --resource-group NOME_RESOURCE_GROUP --name NOME_AKS
```

Snippet 4.9: Criação e acesso ao *AKS*

Após esses comandos já é possível aceder ao *dashboard* do *Kubernetes* (figura 4-15).

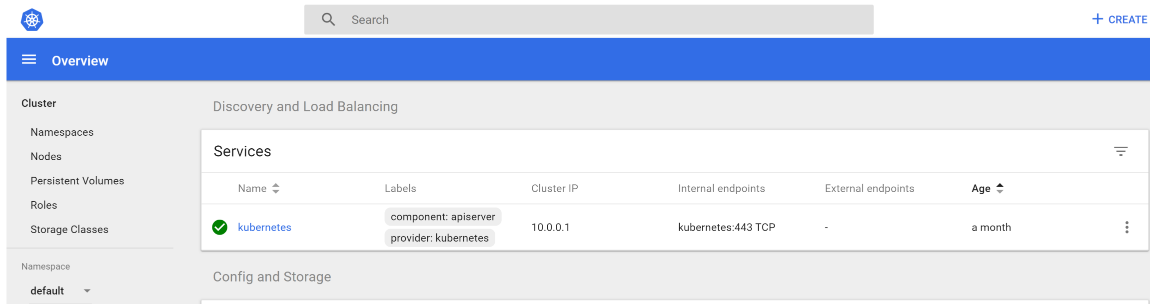

Figura 4-15: *Dashboard - AKS*

4.6 Kubernetes

Os ficheiros de configuração do *Kubernetes* fazem parte do repositório *DCP Ops*, e estão distribuídos por ambiente.

Foi decidido a criação de um ficheiro de *Kubernetes* (figura 4-16) por cada micro-serviço, uma vez que facilita a configuração e a gestão individual. Os vários micro-serviços têm configurações diferentes e necessitam de recursos diferentes.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
    name: persistency-product
    namespace: backend
spec:
    selector:
        matchLabels:
            app: persistency-product
    strategy:
        type: RollingUpdate
    template:
        metadata:
            labels:
                app: persistency-product
        spec:
            containers:
                - name: persistency-product
                  image: hipregistry.azurecr.io/dcp-persistency
                  volumeMounts:
                      - name: certificates
                        mountPath: "/certificates"
                        readOnly: true
                  imagePullPolicy: Always
                  env:
                      - name: SONAESF_ENVIRONMENT...
                      - name: SONAESF_Database__DCP__User...
                      - name: SONAESF_Database__DCP__Password...
                      - name: SONAESF_Cache__Options__Password...
                      - name: SONAESF_Elasticsearch__Credentials__Username...
                      - name: SONAESF_Elasticsearch__Credentials__Password...
                      - name: VERBOSE_MODE...
                  command: ["/bin/sh"]
                  args:
                      [
                          "-c",
                          "dotnet SonaeSF.DCP.Persistency.dll -f product ${VERBOSE_MODE}",
                      ]
            restartPolicy: Always
            volumes:
                - name: certificates
                  secret:
                      secretName: certificates
```

Figura 4-16: *Ficheiro Kubernetes*

Foram utilizados os recursos de *Deployment*, uma vez que criam o *ReplicaSet*, o que facilita a gestão dos *pods* principalmente ao nível de escalar os serviços.

Volumes

A nível de volumes foram utilizados volumes não persistentes, uma vez que grande parte dos serviços necessitam de certificados para comunicar com os sistemas de *Kafka*. Para isso é definido um volume e posteriormente é montado esse volume no *Deployment*, que tem a mesma vida útil que o *pod*.

Variáveis de ambiente

Para as variáveis de ambiente foram utilizados dois recursos do *Kubernetes*, para dados não sensíveis foram utilizados os *ConfigMap*, sendo que para os dados sensíveis como *passwords* foram utilizados os *Secrets*.

Um *ConfigMap* pode ser criado através de *CLI* com o *kubectl*, ou através de ficheiro. Por *CLI* foi preterido uma vez que não dá para gerir versões.

Foi criado um ficheiro com a configuração do *ConfigMap* (*snippet* 4.10).

```
{
  "kind": "ConfigMap",
  "apiVersion": "v1",
  "metadata": {
    "name": "config",
    "namespace": "backend"
  },
  "data": {
    "APP_ENV": "Staging",
    "BLOBS_CONTAINER_REFERENCE": "hip",
    "DROPBOX_ORIGIN_FOLDER": "/Ecommerce/SFPhotoUploadStg",
    "DROPBOX_WORKER_FOLDER": "/Ecommerce/SFPhotoUploadWorkerStg",
    "VERBOSE_MODE": "-v"
  }
}
```

Snippet 4.10: Configuração de um *ConfigMap*

Para criar o *ConfigMap* executar o comando do *snippet* 4.11.

```
kubectl apply -f NOME_FICHEIRO
```

Snippet 4.11: Criação do *ConfigMap* no *cluster* de *Kubernetes*

Os *Secrets* têm uma estrutura parecida com os *ConfigMap*, com a diferença que nos dados chave-valor, o valor é obrigatoriamente preenchido em *base64* e os dados aparecem encriptados no *dashboard* do *kubernetes*. Para converter texto para *base64* em sistemas *Linux*, ver figura 4-17.

```
brunobold@ubuntuserver:~/share$ echo -n 'admin' | base64
YWRtaW4=
brunobold@ubuntuserver:~/share$ echo -n 'teste123456789' | base64
dGVzdGUxMjM0NTY3ODk=
```

Figura 4-17: Converter texto para *base64* em sistemas *Linux*

De seguida é mostrado um exemplo (*snippet* 4.12), uma vez que não é possível a demonstração do original devido a conter dados sensíveis da organização.

```
{
  "kind": "Secret",
  "apiVersion": "v1",
  "metadata": {
    "name": "secrets",
    "namespace": "backend",
    "selfLink": "/api/v1/namespaces/backend/secrets/secrets",
  },
  "data": {
    "ELASTICSEARCH_PASSWORD": "dGVzdGUxMjM0NTY3ODk=",
    "ELASTICSEARCH_USERNAME": "YWRtaW4="
  },
  "type": "Opaque"
}
```

Snippet 4.12: Configuração de um *Secret*

Para criar o *Secret* executar o comando do *snippet* 4.13.

```
kubectl apply -f NOME_FICHEIRO
```

Snippet 4.13: Criação do *Secret* no *cluster* de *Kubernetes*

Para utilizar os dados dos *ConfigMaps* e dos *Secrets* nos *pods*, basta mapear na secção *env* como demonstrado de seguida (*snippet* 4.14).

```
env:
    - name: SONAESF_ENVIRONMENT
      valueFrom:
        configMapKeyRef:
          name: config
          key: APP_ENV
    - name: SONAESF_Elasticsearch__Credentials__Username
      valueFrom:
        secretKeyRef:
          name: secrets
          key: ELASTICSEARCH_USERNAME
```

Snippet 4.14: Utilização de *ConfigMap* e *Secret*

Serviços

No *Namespace* de *frontend*, há a necessidade de aceder aos serviços de fora da rede. Foram exploradas três formas de expor esse serviço, sendo que a primeira abordagem foi através de *ClusterIP* (figura 4-18). Esta abordagem oferece um serviço dentro do *cluster* do *Kubernetes*, que as restantes aplicações podem aceder dentro do *cluster*. Se é apenas de dentro do *cluster* porque foi tentada essa abordagem? Pode ser acedido utilizando para isso um *proxy Kubernetes*.

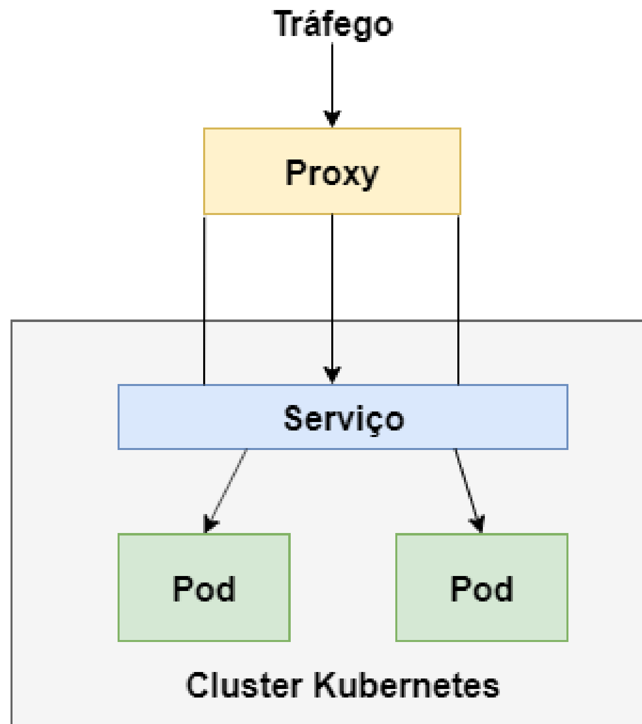

Figura 4-18: *ClusterIP Kubernetes*

Executar de seguida o *proxy* (*snippet* 4.15) com abertura da porta 8080.

```
kubectl proxy --port=8080
```

Snippet 4.15: Execução de um *proxy* no *Kubernetes*

Esta foi a abordagem escolhida para expor os serviços a comunicarem entre eles, por exemplo os serviços de *API* a serem acedidos pelos serviços de *front-end* (figura 4-19).

```
apiVersion: v1
kind: Service
metadata:
    name: web-api
    namespace: backend
spec:
    type: ClusterIP
    ports:
        - port: 80
          targetPort: 8000
    selector:
        app: web-api
```

Figura 4-19: *ClusterIP* para a *WebAPI* do DCP

O problema desta abordagem é que para ser executado o *proxy*, é necessário executar o *kubectl* com um utilizador autenticado, não suporta roteamento nem balanceamento de carga. Por estes motivos foi descartada esta abordagem para expor os serviços para o exterior da rede.

A segunda abordagem foi a de utilizar um balanceador de carga para expor os serviços. Esta abordagem é muito semelhante à anterior a nível de arquitectura, simplesmente o *proxy* é substituído por um balanceador de carga que como o nome indica suporta balanceamento de carga. O balanceador fornece um único endereço *IP* e encaminha todo o tráfego de rede que bate nesse *IP* para os serviços do *Kubernetes*.

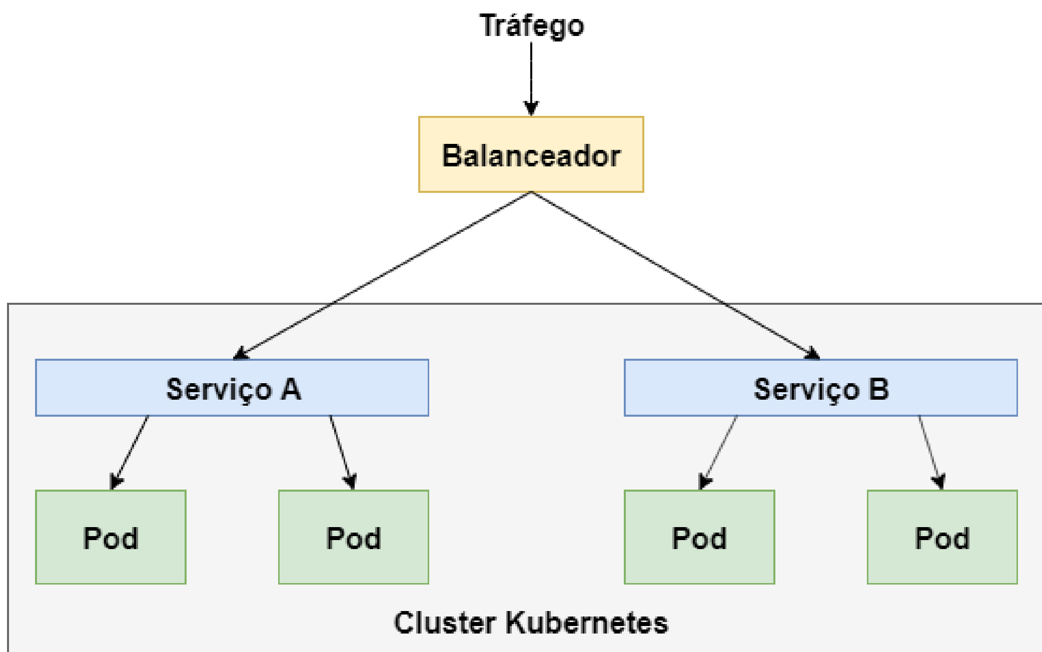

Figura 4-20: *Balanceador com Kubernetes*

Pelo observado na figura 4-20, existe balanceamento entre os mesmos serviços, por exemplo dois serviços iguais replicados (serviço A e B), caso existisse um serviço diferente era necessário mais um balanceador.

Para criar um balanceador no *Azure* utilizando *az cli* (*snippet* 4.16).

```
#criar resource group
az group create \
    --name NOME_RESOURCE_GROUP \
    --location eastus

#criar IP Publico
az network public-ip create --resource-group NOME_RESOURCE_GROUP --
    ↪ name sonaeIPVIPPublico



#criar load balancer
az network lb create \
    --resource-group NOME_RESOURCE_GROUP \
    --name sonaeIPVLoadBalancer \
    --public-ip-address sonaeIPVIPPublico \
    --frontend-ip-name sonaeIPVFrontEndPool \
    --backend-pool-name sonaeIPVBackEndPool
```

Snippet 4.16: Criação e configuração do balanceador

Após a criação basta criar uma *health probe* com o *cluster Kubernetes* e as regras de balanceamento para o *cluster*.

A desvantagem desta abordagem é de que não suporta roteamento inteligente, e necessita obrigatóriamente de um balanceador externo.

A terceira abordagem foi o *Ingress*. Ao contrário das abordagens anteriores, o *Ingress* não é um tipo de serviço. O *Ingress* fica à frente dos vários serviços e funciona como um roteador ou ponto de entrada no *cluster*.

O *Ingress* (figura 4-21) apesar de ser a forma mais poderosa de expor serviços no *Kubernetes*, é também a mais complicada de utilizar. Existem vários tipos de *Ingress*

Controllers como por exemplo *Kong*, *Skipper* ou *NGINX*. Aqui houveram dois que sobressaíram, o *HAProxy* devido aos seus algoritmos avançados de balanceamento de carga, e o *NGINX* devido à comunidade ser maior, e devido aos serviços serem expostos apenas para as equipas da organização, a escolha acabou por ser o *NGINX*.

O *Ingress* é também a forma mais útil de expor vários serviços no mesmo *IP* e que todos eles utilizem o mesmo protocolo de camada 7 (normalmente *HTTPS*).

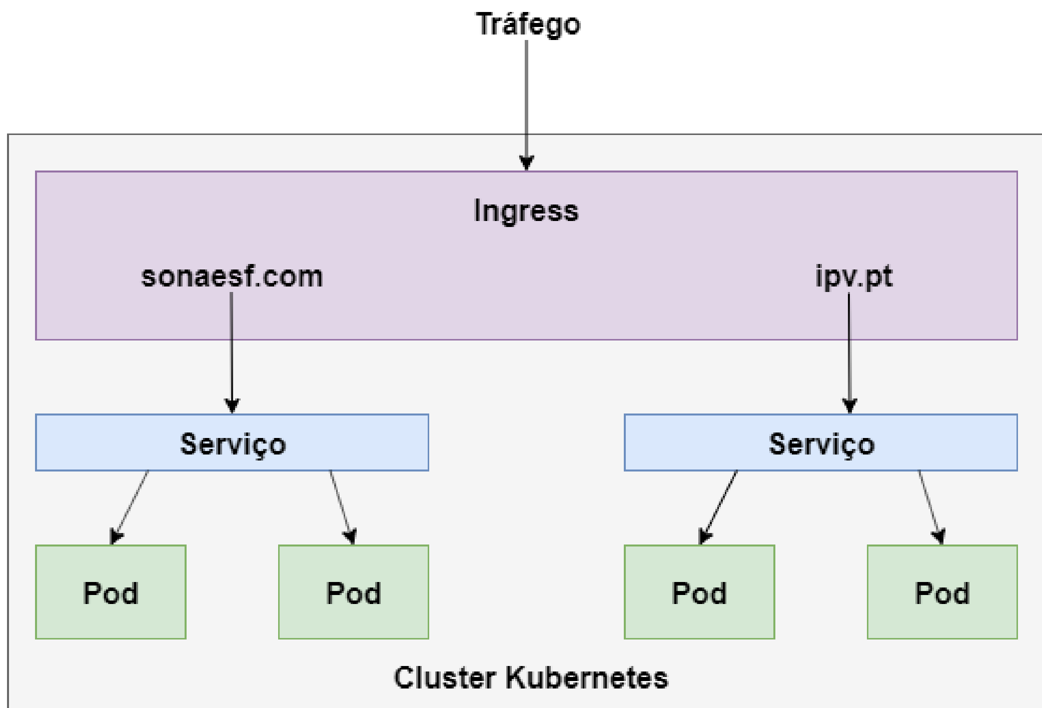

Figura 4-21: *Ingress controllers*

Para instalar o *Ingress* primeiro executar os seguintes comandos (*snippet* 4.17).

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-
    ↪ nginx/master/deploy/static/mandatory.yaml

#Comando para executar caso o provider seja o Azure
kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-
    ↪ nginx/master/deploy/static/provider/cloud-generic.yaml
```

Snippet 4.17: Instalação *Ingress controller*

De seguida basta adicionar o serviço de *Ingress*, por exemplo no ficheiro de configuração do *Deployment*, figura 4-22.

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
    name: ingress-nginx
    namespace: frontend
    annotations:
        kubernetes.io/ingress.class: nginx
        certmanager.k8s.io/cluster-issuer: letsencrypt-prod
        certmanager.k8s.io/acme-challenge-type: http01
spec:
    tls:
        - hosts:
              - dcp.sonaesf.com
          secretName: letsencrypt-prod
    rules:
        - host: dcp.sonaesf.com
          http:
              paths:
                  - path: /
                    backend:
                        serviceName: dashboard-ui
                        servicePort: 3000
```

Figura 4-22: *Ingress controllers* DCP

Autoscaling dos pods

Primeiro foram definidos os recursos, na figura 4-23 para o *pod* são reservados 0.25 de *CPU* e no máximo poderá utilizar 0.5 de *CPU*.

```
resources:
    requests:
      cpu: "250m"
    limits:
      cpu: "500m"
```

Figura 4-23: Configuração de recursos para um *pod*

De seguida executar o comando do *snippet* 4.18 para criar o *Horizontal Pod Autoscaler*.

```
kubectl autoscale deployment NOME_SERVICO --cpu-percent=50 --min=3 --
    ↪ max=10
kubectl get hpa
```

Snippet 4.18: Configuração do *HPA*

Com isso irá manter o *CPU* dos *pods* em cerca de 50%. Por exemplo caso estejam em execução quatro *pods* e a carga de *CPU* seja superior a 50%, será lançado um novo *pod*. No estágio foram definidos limites de 75% nos serviços de *Back-end*.

O problema de um *HPA* é quando são atingidos os limites dos nós. Não adianta apenas escalar os *pods* se não existirem máquinas (nós) com recursos para aguentarem a carga. Neste caso o *HPA* vai continuar a tentar lançar novos *pods*, o que vai resultar em *pods* no estado pendente.

Autoscaling do cluster

Para a configuração de *autoscaling* no *cluster AKS* temos que definir os seguintes parâmetros no comando *az* do *snippet* 4.19.

```
 --enable-cluster-autoscaler \
 --min-count 1 \
 --max-count 3
```

Snippet 4.19: Configuração inicial do *autoscaling*

Habilitar o *autoscaler*, definir um número mínimo de nós, e definir o número máximo (questões de segurança por questões de preços).

Para mais tarde actualizar esses valores executar o comando do *snippet* 4.20.

```
az aks update \
--resource-group sonaeIPVResourceGroup \
--name sonaeIPVAKSCluster \
--update-cluster-autoscaler \
--min-count 1 \
--max-count 10
```

Snippet 4.20: Actualização dos valores do *autoscaling*

Em algumas regiões do Azure ainda é possível aceder às máquinas dos nós. E embora o AKS utilize o autoscaling de VM para os nós, não é recomendado mexer em qualquer configuração do nó. É recomendado deixar ser o AKS a gerir as suas VMs.

4.7 Configuração dos pipelines CI/CD

No ciclo de desenvolvimento moderno, preparar as releases e colocar esses desenvolvimentos nos vários ambientes continuamente tornou-se um factor de extrema importância do processo. Os *pipelines* de *CI/CD* ajudam a reagir imediatamente a um possível problema de código.

Pipelines CI

No caso do estágio foram criados *pipelines* de *CI* para três grupos de repositórios.

O *pipeline* de *CI* ligado ao repositório *.NET Libs* tem como responsabilidade preparar os pacotes *nuget* e enviar os mesmos para um *Nuget Server* privado.

O *pipeline* de *CI* do repositório *DCP Ops*, tem como responsabilidade fornecer os ficheiros de configuração do *Kubernetes* para um *artifact* que pode ser acedido pelo *pipeline* de *CD*.

Os *pipelines* de *CI* dos repositórios do *DCP* (*Back-end* e *Front-end*) executam a compilação automática e inserem essa compilação em *artifacts* que podem ser acedidos

pelos *pipelines* de *CD*.

A primeira abordagem ao nível dos *pipelines* de *CI* foi o que pareceu racional. Para preparar as *releases* era necessário executar de previamente uma série de comandos (compilação, preparar as *Docker images*, enviar as *Docker images* para o *Docker registry*, etc), então foi tentado uma abordagem de colocar esses comandos a serem executados em cada *pipeline* de *CI*. Cedo foi notório de que essa abordagem de racional tinha muito pouco, uma vez que para criar *pipelines* para um ou dois projectos é exequível, já para mais de uma dezena é extremamente difícil de gerir.

A segunda abordagem foi tentar seguir a mesma lógica dos *Dockerfiles*, uma vez que os *pipelines* do *Azure DevOps* suportam configurações em ficheiros *YAML* (figura 4-24), foi criado um único ficheiro para todos os *pipelines* e controlar o pipeline através de variáveis de ambiente.

```yaml
pool:
    name: "Hosted Ubuntu 1604"
trigger:
    - release/staging
    - release/production/v*
pr: none
variables:
    - group: BUILD
steps:
    - bash: |
          # If branch contains version number, lets catch it
          VERSION=$BUILD_BUILDNUMBER
          if [[ $BUILD_SOURCEBRANCHNAME =~ v[0-9]+\.[0-9]+\.[0-9]+ ]]; then
            VERSION=$BUILD_SOURCEBRANCHNAME
          fi

          # Set variable
          echo "##vso[task.setvariable variable=version]$VERSION"

          # Print version
          echo "Source Branch ==> ${BUILD_SOURCEBRANCH}"
          echo "Source Branch Name ==> ${BUILD_SOURCEBRANCHNAME}"
          echo "Docker Image Tag ==> ${VERSION}"
      displayName: "Set docker image tag version"
    - bash: "docker build --build-arg FULLY_QUALIFIED_APP_NAME=$(App.Name) --build-arg FEED_URL=$(Nuget.FeedUrl) --build-arg PAT=$(Nuget.PAT)
    -t $(Acr.Host)/$(App.ImageName):$VERSION $(System.DefaultWorkingDirectory)"
      displayName: "Build Image"
    - bash: "docker login $(Acr.Host) -u $(Acr.Username) -p $(Acr.Password)"
      displayName: "Login to Docker Registry"
    - bash: "docker push $(Acr.Host)/$(App.ImageName):$VERSION"
      displayName: "Push Image"
```

Figura 4-24: Ficheiro de configuração das *pipelines*

É definida a *pool*, basicamente será a máquina em que irá ser executado o *pipeline* (figura 4-25), se uma máquina *windows*, *MacOS*, para este projecto são máquinas com *SO Ubuntu*. De seguida foram especificados os *triggers*, que serão os *branches* do repositório de código em que alterações ao código irão fazer com que o *pipeline*

seja iniciado. Nos *steps* foi implementado *bash script* para validar se é uma *release* com a versão definida, caso contrário utiliza a por omissão do *pipeline*, que será uma data. De seguida irá tratar de criar as *Docker images* e de seguida irá autenticar-se no *Docker Registry* e enviar as *Docker images*.

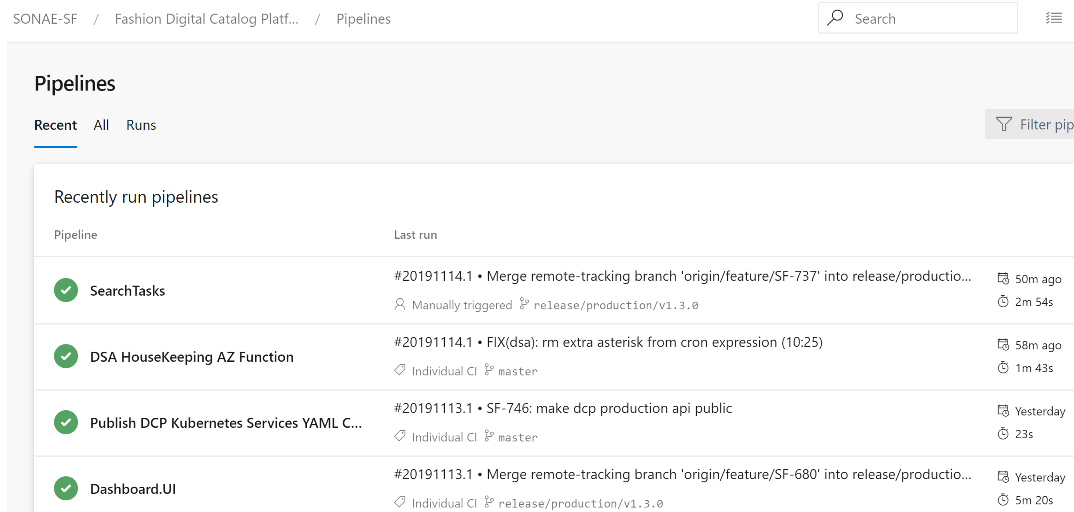

Figura 4-25: *Pepilines* do DCP

Pipelines CD

Para a configuração dos *pipelines* de *CD* são definidos os *artifacts* que fazem parte desse *pipeline* e os estágios, normalmente um estágio por cada ambiente (figura 4-26).

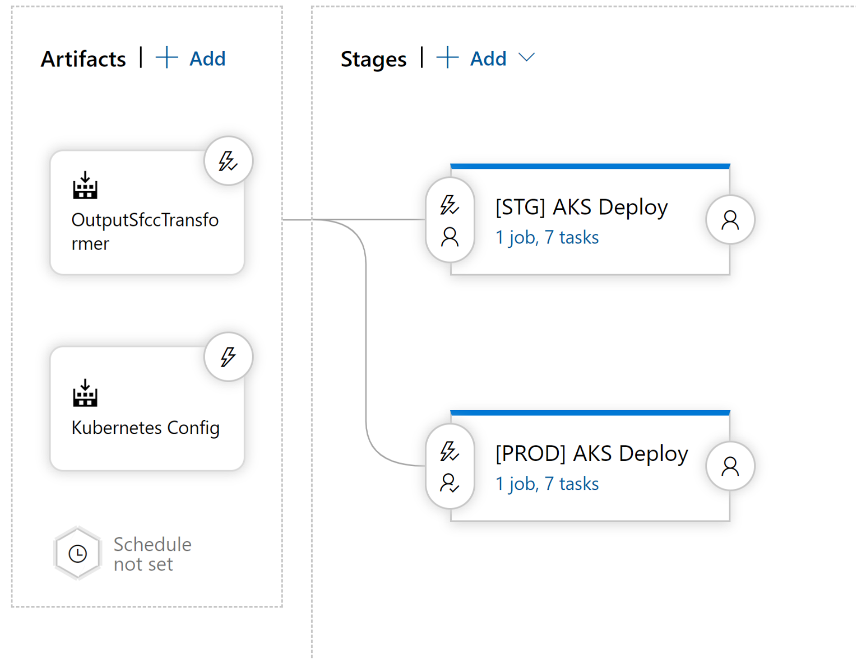

Figura 4-26: *Release* do DCP

Apenas o *artifact* "OutputSfcc.Transformer"irá fazer executar os estágios dos vários ambientes, o de *STG* de forma automática, e o de *PROD* automático mas com a necessidade de aprovação.

Dentro de cada estágio existem várias tarefas (figura 4-27), definir a versão da *release* (código do *snippet* 4.21), aplicar as configurações contidas no ficheiro *Kubernetes* correspondente ao *pipeline*, e actualizar as imagens nos *pods*.

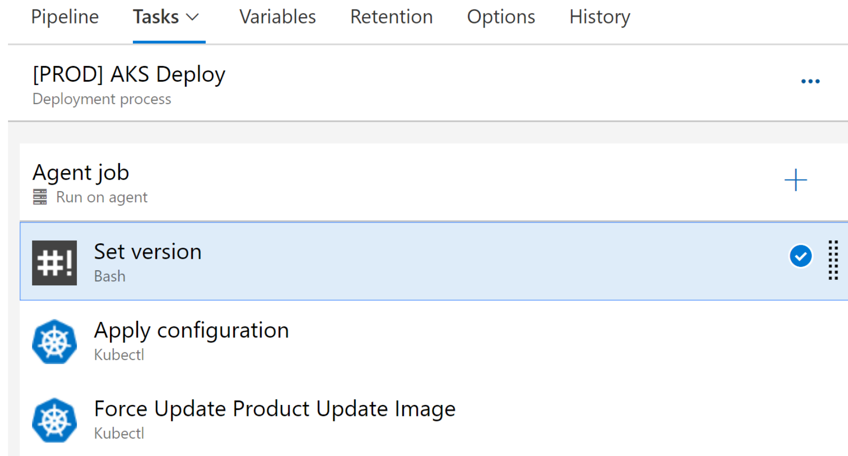

Figura 4-27: Estágio de um *pipeline*

```
#!/bin/bash

# If branch contains version number, lets catch it
VERSION=$BUILD_BUILDNUMBER
if [[ $BUILD_SOURCEBRANCHNAME =~ v[0-9]+\.[0-9]+\.[0-9]+ ]]; then
    VERSION=$BUILD_SOURCEBRANCHNAME
fi

# Set variable
echo "##vso[task.setvariable variable=version]$VERSION"

#Print version
echo "Source Branch ==> ${BUILD_SOURCEBRANCH}"
echo "Source Branch Name ==> ${BUILD_SOURCEBRANCHNAME}"
echo "Docker Image Tag ==> ${VERSION}"
```

Snippet 4.21: Inserir a versão da *release*

Para a utilização das variáveis de ambiente foi utilizada a *Library* do *Azure DevOps*, com os *Variable Groups*, que podem ser utilizados dentro dos *pipelines*.

Esta página foi propositadamente deixada em branco.

Capítulo 5

Conclusões

Com este capítulo chega o término deste relatório de estágio e as conclusões do mesmo. A primeira secção aborda o estágio em si e os conhecimentos adquiridos ao longo desse período. A segundo secção fala sobre os resultados obtidos no estágio. Por último a terceira secção apresenta alguns planos de trabalho futuros.

5.1 Estágio

O estágio curricular decorreu sem problemas de maior, tendo sido o mesmo realizado com acompanhamento permanente por parte dos orientadores, precavendo assim possíveis falhas no mesmo.

No decorrer do estágio foram exploradas várias tecnologias, assim como a integração com a equipa de trabalho.

Foi possível após o estágio desenvolver e fomentar competências nas várias áreas tecnológicas envolvidas no mesmo.

Ao longo do estágio tive sempre a liberdade e fui sendo incitado a propor ideias com o intuito de enriquecer o projecto. É sempre uma motivação extra estes incentivos e liberdade por parte da empresa.

Foi ainda bastante engrandecedor o companheirismo e a partilha de conhecimento por parte da equipa de trabalho, permitindo-me crescer e que sem a mesma os resultados alcançados neste projecto não teriam sido os mesmos.

5.2 Resultados

Durante o período de estágio foi possível a implementação de várias tarefas. Primeiro o desenho dos fluxos de implantação dos vários micro-serviços, assim como o desenho da gestão de versão de *software*. De seguida a conteinerização dos vários micro-serviços e a implementação desses serviços no ambiente de desenvolvimento, sendo de seguida criado e configurado um *Container Registry* para hospedar as várias *Docker images*. Posteriormente foi criado e configurado um *cluster* de *kubernetes* com recurso ao *Azure Kubernetes Service*. De seguida foi implementado a orquestração dos vários *containers* utilizando para isso o *Kubernetes*. Foram ainda criados e configurados os vários *pipelines* de *CI* e de *CD* no *Azure DevOps*.

Como complemento, foi criado e configurado um *nuget server* privado, uma ferramenta essencial para a equipa criar, compartilhar e consumir código útil.

Tudo o que foi implementado encontra-se ao dia de hoje em produção, o que permite o funcionamento das lojas *online* da Zippy e MO, nacionais e internacionais, assim como as aplicações das vendas em lojas físicas. Existe ainda a possibilidade de outras empresas da organização como os sistemas das lojas online do Continente e da Wells virem a utilizar os sistemas implementados durante o estágio, o que demonstra a robustez das implementações.

5.3 Trabalho futuro

Embora os objectivos tenham sido atingidos, seria interessante explorar a ferramenta *Pulumi* como possível alternativa ao *Terraform*. O *Terraform* obriga a aprendizagem de uma linguagem proprietária, o *Pulumi* por outro lado suporta linguagens como *Python, Go, JavaScript e TypeScript*, o que diminui o tempo de aprendizagem. Além disso existe uma ferramenta *tf2pulumi* que converte *Terraform HCL* para *Pulumi*.

Ao nível do *cluster Kubernetes*, será de valor explorar ferramentas para testes automáticos de segurança, como por exemplo o Kubesec.io. Antes que as configurações dos ficheiros Kubernetes sejam executadas, é efectuada uma chamada ao Kubesec.io para verificar se tudo está bem. O Kubesec.io verifica por exemplo se as políticas de segurança definidas são adequadas (fornece informações sobre o que pode estar errado).

Esta página foi propositadamente deixada em branco.

Bibliografia

[1] "15 benefits of microservices you need to know about", 2019. https://www.qat.com/15-benefits-microservices/, 2019-09-10.

[2] "agile development from a programmer's perspective", 2019. https://levelup.gitconnected.com/agile-from-a-developers-perspective-27b23ea665f0/, 2019-09-29.

[3] "azure devops", 2019. https://www.microsofttech.com.br/pt-br/microsofttech/azure-devops/, 2019-10-23.

[4] "azure kubernetes service", 2019. https://azure.microsoft.com/en-us/services/kubernetes-service/, 2019-10-25.

[5] "azure resource manager overview", 2019. https://docs.microsoft.com/en-us/azure/azure-resource-manager/resource-group-overview, 2019-10-15.

[6] "container registry", 2019. https://azure.microsoft.com/en-us/services/container-registry/, 2019-10-15.

[7] "continuous delivery and integration: Rapid updates by automating quality assurance", 2019. https://www.altexsoft.com/blog/business/continuous-delivery-and-integration-rapid-updates-by-automating-quality-assuranc 2019-09-22.

[8] "continuous integration", 2019. https://cloud.google.com/solutions/continuous-integration/, 2019-09-22.

[9] "docker hub", 2019. https://www.docker.com/products/docker-hub/, 2019-10-10.

[10] "docker overview", 2019. https://docs.docker.com/engine/docker-overview/, 2019-10-27.

[11] "hashicorp nomad", 2019. https://www.nomadproject.io/, 2019-10-12.

[12] "jira atlassian", 2019. https://www.atlassian.com/software/jira/guides/use-cases/what-is-jira-used-for#jira-for-agile-teams/, 2019-10-01.

[13] "kubernetes - service", 2019. https://kubernetes.io/docs/concepts/services-networking/service/, 2019-09-24.

[14] "kubernetes components", 2019. https://kubernetes.io/docs/concepts/overview/components/, 2019-09-23.

[15] "kubernetes master components: Etcd, api server, controller manager, and scheduler", 2019. https://medium.com/jorgeacetozi/kubernetes-master-components-etcd-api-server-controller-manager-and-scheduler-3a0 2019-09-24.

[16] "kubernetes – cloud providers and storage classes", 2019. https://theithollow.com/2019/03/13/kubernetes-cloud-providers-and-storage-classes/, 2019-09-20.

[17] "microservice architectures: What they are and why you should use them", 2019. https://blog.newrelic.com/technology/microservices-what-they-are-why-to-use-them/, 2019-09-10.

[18] "overview of docker compose", 2019. https://docs.docker.com/compose/, 2019-09-21.

[19] "overview of kubectl", 2019. https://kubernetes.io/docs/reference/kubectl/overview/, 2019-09-24.

[20] "pod overview", 2019. https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/, 2019-09-25.

[21] "scrum artifacts", 2019. https://www.scrumalliance.org/learn-about-scrum/scrum-elearning-series/scrum-artifacts/, 2019-09-29.

[22] "sonae", 2019. https://www.sonae.pt/pt/, 2019-09-19.

[23] "sonae - fashion", 2019. https://www.sonae.pt/pt/sonae/o-grupo-e-os-negocios/, 2019-09-19.

[24] "swarm mode overview", 2019. https://docs.docker.com/engine/swarm/, 2019-09-21.

[25] "terraform", 2019. https://www.terraform.io/, 2019-11-02.

[26] "the advantage of using scrum methodology", 2019. https://www.goodworklabs.com/the-advantage-of-using-scrum-methodology/, 2019-09-29.

[27] "the benefits of scrum & agile", 2019. https://www.thescrummaster.co.uk/scrum/benefits-scrum-agile/, 2019-09-29.

[28] "the scrum guide", 2019. https://www.scrumguides.org/scrum-guide.html, 2019-09-29.

[29] "understanding microservices", 2019. https://www.redhat.com/en/topics/microservices, 2019-09-15.

[30] "what is application orchestration?", 2019. https://www.mulesoft.com/resources/esb/what-application-orchestration/, 2019-09-21.

[31] "what is dc/os?", 2019. https://dcos.io/, 2019-10-23.

[32] "what is kubernetes", 2019. https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/, 2019-10-03.

[33] "what is microsoft azure? - an introduction to azure", 2019. https://www.dotnettricks.com/learn/azure/getting-started-with-microsoft-azure-platform, 2019-10-12.

[34] "what is scrum?", 2019. https://www.scrum.org/resources/what-is-scrum, 2019-09-29.

[35] "what is scrum methodology?", 2019. https://resources.collab.net/agile-101/what-is-scrum/, 2019-09-29.

[36] Neependra Khare. *Docker Cookbook*. Packt, 2015.

[37] Gigi Sayfan. *Mastering Kubernetes*. Packt, 2017.

[38] Tarek Ziadé. *Python Microservices Development*. Packt, 2017.