Universidade de Évora - Escola de Ciências e Tecnologia

Mestrado em Engenharia Informática

Dissertação

Metodologia para Conteinerização de micro-serviços

Andrei Brinza

Orientador(es) | Teresa Gonçalves
Pedro Salgueiro

Évora 2022

Universidade de Évora - Escola de Ciências e Tecnologia

Mestrado em Engenharia Informática

Dissertação

Metodologia para Conteinerização de micro-serviços

Andrei Brinza

Orientador(es) | Teresa Gonçalves

Pedro Salgueiro

Évora 2022

A dissertação foi objeto de apreciação e discussão pública pelo seguinte júri nomeado pelo Diretor da Escola de Ciências e Tecnologia:

Presidente	|	Lígia Maria Ferreira (Universidade de Évora)
Vogais	|	Vitor Beires Nogueira (Universidade de Évora)

Évora 2022

Agradecimentos

A realização desta dissertação não teria sido possível sem o incentivo e apoio de pessoas chave, aos quais estou muito grato.

Ao Pedro Pessoa pelo apoio e orientação prestado durante todo o percurso do trabalho prático. Neste sentido também agradeço aos docentes, Teresa Gonçalves e Pedro Salgueiro pela participação e colaboração.

Por ultimo agradeço a minha família principalmente ao meu irmão Mihail, por todo apoio e incentivo ao logo deste percurso.

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Excertos de Código

Lista de Acrónimos

CI *Continuous Integration*

CD *Continuous Delivery*

SOA *Service-Oriented Architecture*

VM *Virtual Machine*

VMM *Virtual Machine Monitor*

UI *User Interface*

NIST *National Institute of Standards and Technology*

API *Application Public Interface*

RPC *Remote Procedure Call*

REST *Representational State Transfer*

SOAP *Simple Object Access Protocol*

CPU *Central Processing Unit*

RAM *Random Access Memory*

OS *Operating System*

GKE *Google Kubernetes Engine*

IP *Internet Protocol*

HPC *High Performance Computing*

SaaS *Software as a Service*

PaaS *Platform as a Service*

IaaS *Infrastructure as a Service*

Sumário

Nas últimas décadas a internet tem experienciado um crescimento exponencial. Para acompanhar esse crescimento, as aplicações *web* têm vindo a ser cada vez mais complexas. As aplicações, que inicialmente começaram com uma arquitetura monolítica, tornaram-se ingeríveis devido ao seu grande crescimento. Para mitigar este problema, tem-se optado cada vez mais por arquiteturas em microsserviços. Esta arquitetura, ao contrário da monolítica, permite ter as funcionalidades desacopladas, facilitando a escalabilidade individual. No entanto, devido ao facto de muitas vezes os microsserviços correrem em ambientes ou máquinas diferentes, surgiu a necessidade de uniformizar os ambientes de execução. Inicialmente a solução passou por usar máquinas virtuais e posteriormente evoluiu para *containers*. Um *container* é um *package* independente que contém apenas o que é estritamente necessário para correr a aplicação.

Devido ao grande número de microsserviços que algumas aplicações têm, surgiu a necessidade de automatizar o processo de orquestração de *containers*, ou seja, escalar e alocar recursos automaticamente a cada instância do microsserviço.

Esta dissertação estuda e aplica uma metodologia de conteinerizar à uma aplicação com uma arquitetura de microsserviços, que corre em máquinas virtuais. Adicionalmente, também são discutidos e aplicados conceitos de *Continuous Integration* (CI)/*Continuous Delivery* (CD) e DevOps, criando uma *pipeline* automatizada com Git Actions.

Palavras chave: Microsserviços, Conteinerização, Orquestração, Docker, Kubernetes, Continuous Integration, Continuous Delivery

Abstract

Methodology for containerization of microservices

In the last decades, we have witnessed an exponential growth in internet usage in our lives. This led to increasingly more complex web applications. The applications usually start as a monolithic architecture that naturally grows in size, making them hard to maintain and understand. To mitigate this problem, microservices-based architectures have seen an increase in popularity. In contrast to the monolithic architecture, microservices architectures allow the development of decoupled functionalities, which can be then scaled individually. However, since most of the time the microservices run in different environments or machines, there is the need to unify the runtime environments. Initially, the solution was to use virtual machines, and then the trend transitioned to containers, which is just a small package that only has what is absolutely needed to run the application.

Due to the high number of microservices that some applications have, orchestration tools that automatically scale and allocate resources to the microservices are needed.

This thesis studies and applies a containerization methodology to a real-world microservices application that currently runs on virtual machines. Furthermore, it also introduces and applies the concept of CI/CD pipelines and DevOps, using Git Actions.

Keywords: Microservices, Containerization, Orchestration, Docker, Kubernetes, Continuous Integration, Continuous Delivery

Capítulo 1

Introdução

Nas últimas décadas temos vindo a assistir a um crescimento exponencial da utilização de internet na vida das pessoas. Estamos numa era tecnológica em que podemos fazer virtualmente quase tudo através da internet e em que passamos cada vez mais tempo *online*, quer seja em ambiente de lazer, estudo ou trabalho (Ruzgar, 2005).

Essa evolução na forma como usamos a internet foi sempre sustentada pelas aplicações *web*, que se têm vindo a tornar cada vez mais sofisticadas e complexas. No entanto, isso tem trazido vários desafios na forma como as empresas criam *software* e o disponibilizam aos seus clientes. No contexto de um único serviço, o aumento da complexidade e do número de funcionalidades de uma aplicação leva a um aumento drástico do tamanho do código fonte da aplicação, tornando-se numa arquitetura monolítica - arquitetura caracterizada por ter o código todo unificado num único bloco. Isso traz várias desvantagens já que, quanto mais complexo o código de uma aplicação, mais difícil é compreendê-lo, tornando mais difícil fazer alterações, uma vez que, ao alterar o código existe o risco de introduzir *bugs* noutros módulos. Para além disso, não é possível escalar partes individuais da aplicação, ou seja, se houver muitos pedidos de um determinado tipo é necessário adicionar uma máquina nova com a aplicação e distribuir os pedidos entre as máquinas (*horizontal scaling*) ou aumentar os recursos da mesma máquina (*vertical scaling*), o que leva a um uso de recursos ineficiente.

De modo a evitar estes problemas, nos meados da década de dois mil surgiu o conceito de arquitetura em microsserviços (Dragoni et al., 2017). Este tipo de arquitetura consiste em dividir a aplicação em pequenos módulos, idealmente independentes ou com o mínimo de dependências, possibilitando assim um desenvolvimento autónomo. Desta forma, quando surge um requisito para uma nova funcionalidade basta criar um novo microsserviço, sem impactar o que está em produção naquele momento, ao contrário da arquitetura monolítica. Na Secção 2 são apresentados detalhadamente os conceitos da arquitectura monolítica e de microsserviços, a evolução de uma arquitetura para a outra e respectivas vantagens e desvantagens.

A arquitetura em microsserviços tem vindo a ser cada vez mais utilizada. Habitualmente as empresas começam com uma aplicação que vai crescendo naturalmente para um monólito,

o qual é dividido em microsserviços (e.g., Netflix, que tem mais de mil microsserviços no seu produto) (Dragoni et al., 2017; Thönes, 2015). O elevado número de microsserviços que geralmente as empresas possuem levou também a uma mudança na forma como as empresas trabalham, evoluindo para equipas mais pequenas e tendo um ciclo de entregas muito mais rápido, adotando metodologias de trabalho *agile* e uma cultura de DevOps, discutidas na Secção 2.7.

Paralelamente à evolução das arquiteturas de *software*, houve também uma mudança na forma como as aplicações são *deployed*, ou seja, na forma como são instaladas e disponibilizadas aos clientes, sendo que esta mudança de *deploy* se baseou na evolução de máquinas físicas para máquinas virtuais[1] e posteriormente para *containers* (Sharma et al., 2016). A Secção 2.4 descreve mais em detalhe as vantagens e desvantagens de cada abordagem. Enquanto as máquinas virtuais permitem virtualizar os recursos de uma máquina física (*hypervisor-based virtualization*) possibilitando a execução concorrente de vários sistemas operativos, os *containers*, por outro lado, abstraem a nível do sistema operativo, não tendo como objetivo emular um sistema de *hardware*, mas sim, permitir que o *kernel* do Linux faça a gestão do isolamento entre aplicações, consentindo, dessa forma, que vários sistemas de Linux isolados (*containers*) corram num único *host*, partilhando a mesma instância de *kernel* (Li and Kanso, 2015). Um *container* é uma unidade de *software* que contém apenas os *packages* necessários para a execução da aplicação, o que faz com que seja bastante mais pequeno do que uma máquina virtual. Esta evolução permite um uso mais eficiente dos recursos, uma vez que um *container* é abstraído por um ou vários processos nativos. Para além disso, também permite reduzir os erros (e.g., diferenças de versões das dependências entre ambientes), uma vez que as aplicações correm sempre a partir do mesmo binário em todos os ambientes, tornando a depuração e despiste muito mais rápidos.

Por fim, houve também uma migração de sistemas *on-premises* para *cloud*, o que desencadeou que muitas empresas deixassem de comprar e gerir as suas próprias máquinas (*on-premises*) e passassem a alugar recursos na internet (*cloud*). A Secção 2.6 apresenta em mais detalhe alguns dos principais fornecedores de *cloud* e como estes funcionam.

A evolução de *software* e da forma com fazemos *deploy* fazem com que hoje em dia existam arquiteturas muito granulares compostas por centenas de microsserviços que correm em *containers*, na *cloud* ou *on-premises*. E, se por um lado, DevOps (que consiste na junção entre o desenvolvimento e as operações) automatiza vários processos, desde *build* até ao *deploy* e monitorização, é necessário também ter mecanismos de orquestração, ou seja, mecanismos que consigam automaticamente escalar os microsserviços horizontalmente em função da carga que estes recebem, usando, por exemplo, Kubernetes (Sayfan, 2017). A secção 2.5 explora o tema da orquestração dando como exemplos várias tecnologias.

[1] Do inglês, *Virtual Machine* (VM)

1.1 Objetivos

O presente trabalho tem como objetivo definir uma metodologia de conteinerização de microsserviços genérica, de modo a que possa ser aplicada a qualquer aplicação e ser conteinerizada. Para isso será estudado o estado da arte de vários temas relacionados com o desenvolvimento, conteinerização e *deployment* de *software*, nomeadamente:

- Arquiteturas de *software* (e.g., monólito, microsserviços)
- Evolução das metodologias de trabalho dentro das empresas
- Evolução das formas de *deployment*
- Ferramentas de orquestração de *containers* e de automatização dos processos do ciclo de vida do *software*
- *Cloud* e os seus fornecedores

Para além da proposta de metodologia, este trabalho inclui, também a sua aplicação a um problema concreto que consiste na migração de uma aplicação existente, inicialmente a correr em máquinas virtuais *linux on-premises*, para a *cloud* (*Google Kubernetes Engine* (GKE)). Este processo foi automatizado através de uma *pipeline* utilizando Git, Git Actions, Docker e Container Registry.

1.2 Contribuições

A principal contribuição desta dissertação é a definição de uma metodologia de conteinerização de microsserviços, a qual é sustentada pelo estado de arte e pela sua aplicação num trabalho prático, nomeadamente a conteinerização de uma aplicação específica.

1.3 Estrutura da dissertação

O Capítulo 2 começa por apresentar a evolução das arquiteturas de *software* e as formas de fazer *deploy* introduzindo, também, o conceito de orquestração de microsserviços, e concluindo com a análise do modelo de operação na *cloud* e as metodologias de trabalho. No Capítulo 3 são apresentadas de forma mais técnica as tecnologias Docker e Kubernetes e é descrita uma metodologia de conteinerização. O Capítulo 4 apresenta a aplicação da metodologia resultante na migração de uma aplicação e são discutidos os resultados. Por último, no Capítulo 5 são enunciadas as considerações finais do presente trabalho.

1.4 Enquadramento

O presente trabalho foi desenvolvido na empresa Ângulo Sólido, uma empresa de consultoria de informática com sede na cidade de Évora e com uma filial em Lisboa. A empresa foi fundada pelo Eng. Pedro Pessoa e Eng. Gustavo Homem em Fevereiro de 2005.

Capítulo 2

Estado da Arte

Grande parte das aplicações começam com um conjunto restrito de funcionalidades que, com o passar do tempo, vai sendo expandido devido ao surgimento de novos requisitos e de novas ideias. Numa fase inicial, essas novas funcionalidades são fáceis de adicionar na mesma aplicação, fazendo com que, ao longo do tempo, haja um crescimento inevitável do código fonte da aplicação, tornando-se numa aplicação com arquitetura monolítica. Quanto mais complexo e extenso o código fonte, mais difícil é conseguir perceber e manter a aplicação. De modo a tornar as aplicações mais modulares e isolar funcionalidades as aplicações evoluíram para uma arquitetura orientada a serviços[1] (Sprott and Wilkes, 2004; Erl, 2005) e posteriormente para microsserviços (Valipour et al., 2009).

2.1 Arquitetura monolítica

Tal como mencionado, a arquitetura monolítica tem sido preponderante no desenvolvimento de *software*, sendo muito adequada a projetos com dimensões pequenas. No entanto, com bastante frequência, as aplicações começam com uma dimensão relativamente pequena e, devido ao surgimento de novos requisitos, vão crescendo em escala, atingindo elevadas dimensões, e é aí que as desvantagens desse tipo de arquitetura se tornam mais evidentes (Kazanavičius and Mažeika, 2019).

A arquitetura monolítica é um modelo de arquitetura de *software* que consiste numa aplicação que tem todo o seu código no mesmo programa, tal como podemos ver na Figura 2.1. Esta característica faz com que as aplicações monolíticas tenham de ser executadas na mesma máquina, *i.e.,* não é possível correr diferentes partes da lógica em máquinas diferentes. Os monólitos são úteis e adequados para aplicações pequenas, *i.e.*, que ainda não atingiram grande escala, uma vez que o seu planeamento exige menos esforço do que o planeamento de uma arquitetura modular de microsserviços. Para além disso, visto que as funcionalidades são todas adicionadas no mesmo código fonte, a reutilização de código

[1] Do inglês, *Service Oriented Architecture* (SOA)

também se torna mais fácil do que em arquiteturas distribuídas, em que é preciso recorrer à criação de *packages*, *e.g.* NuGet, Maven, Npm.

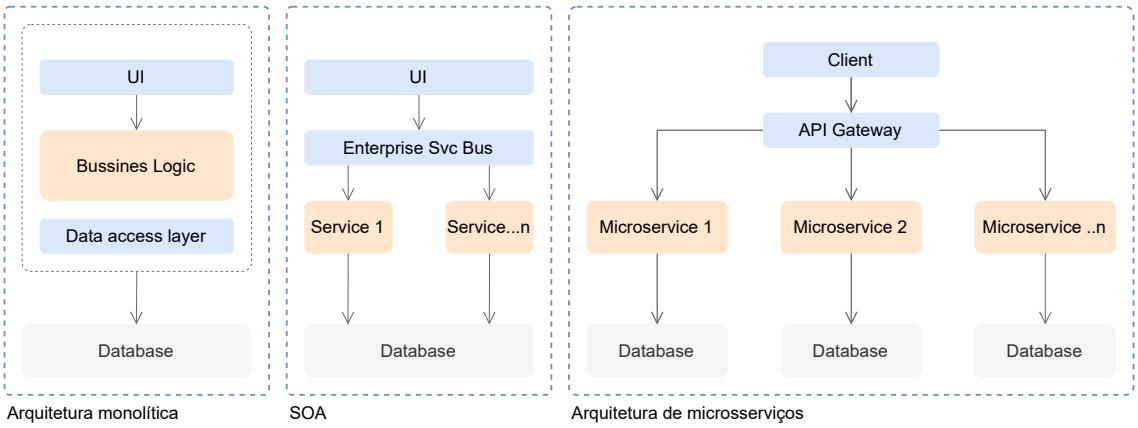

Figura 2.1: Evolução da Arquitetura de *software*: Monólito - SOA - Microsserviços

As desvantagens desta arquitetura surgem principalmente quando a complexidade do código se torna muito elevada, tornando difícil a adição de funcionalidades e manutenção do código fonte. Além disso, existe também o problema da escalabilidade, e.g., se tivermos uma aplicação *web* em que só uma parte do site tem muita carga, é necessário escalar toda a aplicação de forma horizontal ou vertical (Newman, 2017), fazendo com que a utilização de recursos seja muito ineficiente.

Para colmatar estes problemas e atingir um nível superior de modularidade, muitas empresas começaram a adotar arquiteturas orientadas a serviços (SOA) (Sprott and Wilkes, 2004; Erl, 2005), uma arquitetura aprofundada na próxima secção.

2.2 Arquitetura Orientada a Serviços

Uma arquitetura orientada a serviços tem como objetivo separar as funcionalidades da aplicação em serviços que comunicam através de uma interface[2] bem definida. A comunicação pode ser feita através de um *Enterprise Service Bus* (Chappell, 2004), como se pode ver na Figura 2.1, ou diretamente usando protocolos *web* (e.g., *Representational State Transfer* (REST) ou *Simple Object Access Protocol* (SOAP) (Mumbaikar et al., 2013)).

Um serviço pode, essencialmente, ser qualquer coisa; no caso de uma aplicação que lide com utilizadores, pode existir, por exemplo, um serviço que trata de todas as operações sobre os utilizadores (e.g. mudar nome, data de nascimento, etc.) Assim, numa empresa podem existir várias aplicações que necessitam de invocar essas operações, e podem fazê-lo através de uma interface bem definida.

Esta arquitetura apresenta várias vantagens em relação à arquitetura monolítica. Em

[2]Do inglês, *Application Public Interface* (API)

primeiro lugar, a divisão por serviços faz com que a lógica fique mais modular e mais bem dividida dentro da empresa, tornando-a mais simples de compreender e manter. Para além disso, os serviços podem ser desenvolvidos por equipas independentes, resultando num maior isolamento a nível de desenvolvimento. A nível operacional a principal vantagem é a possibilidade de escalar os serviços de forma independente. Desta forma, podemos escalar apenas os serviços que recebem mais carga, (e.g. no exemplo anterior, se existirem muitas aplicações que precisam de comunicar com o serviço que gere utilizadores, então esse serviço pode ter mais instâncias do que outros serviços com menos carga).

Esta arquitetura é geralmente aplicada ao nível das empresas, em que, por sua vez, cada serviço pode ser um monólito com centenas de milhares de linhas de código, sendo assim uma arquitetura de alto nível.

2.3 Arquitetura de Microsserviços

Para colmatar os problemas da arquitetura monolítica e tirar proveito das vantagens da arquitetura SOA, surgiu na última década o conceito de "microsserviços". A arquitetura em microsserviços surgiu como uma versão mais granular da arquitetura SOA (Kazanavičius and Mažeika, 2019). Se por um lado SOA é aplicada a um alto nível, os microsserviços estão num nível mais baixo, ao nível da aplicação. Em teoria, uma empresa pode ter uma uma arquitetura orientada a serviços em que cada serviço tem uma arquitetura de microsserviços.

No paradigma dos microsserviços existe um grande nível de granularidade, traduzindo-se, muitas vezes, num microsserviço por funcionalidade (e.g., numa aplicação *web* de *e-commerce* pode haver um microsserviço que trata das operações do carrinho, um que lista os produtos, outro que processa os pagamentos etc). Tal como na arquitetura orientada a serviços, os microsserviços também podem comunicar entre si, através de protocolos de *Remote Procedure Call* (RPC), REST, SOAP, etc. Geralmente, cada microsserviço é bastante simples, tendo poucos milhares de linhas de código. Isso faz com que as funcionalidades estejam isoladas, o que diminui o risco de uma alteração no código introduzir um *bug* noutra parte da aplicação, quando comparada com uma arquitetura monolítica. Este desacoplamento permite desenvolvimentos paralelos com ciclos de entregas mais rápidos, discutidos mais em detalhe na Secção 2.7.

Os microsserviços têm sido adotados em cada vez mais aplicações de computação na *cloud* (Krylovskiy et al., 2015; Knoche and Hasselbring, 2018). Vários fornecedores de *software* utilizam esta arquitetura, incluindo a IBM e Microsoft, além de inúmeros fornecedores de conteúdos tais como Amazon, Netflix, Spotify, The Guardian, Twitter, PayPal, SoundCloud ou BBC. O sistema de serviço *online* da Netflix usa mais de 1000 microsserviços, envolvendo 5 biliões de interações de serviços por dia (Mo et al., 2019; CloudZero, 2021). No caso da Amazon, cada página requer entre 100 a 150 chamadas a microsserviços (Xu et al., 2020).

Com o aumento do número de microsserviços surgem outros desafios. Se por um lado a

arquitetura de microsserviços traz muitas vantagens a nível do desenvolvimento, por outro, traz muitos desafios a nível operacional comparativamente com os monólitos. Num monólito temos a aplicação toda a correr numa máquina: se houver uma falha na máquina é relativamente fácil reiniciar a máquina, ou instanciar outra. Quanto aos microsserviços a situação é bastante diferente, uma vez que uma aplicação pode ser constituída por centenas de microsserviços que podem ter falhas de forma aleatória e que precisam de ser escalados de forma independente. Para agilizar esses processos as empresas optaram por inicialmente correr os microsserviços em máquinas virtuais e usar *scripts* para automatizar o escalonamento e detetar falhas. Mais tarde, evoluiu-se para *containers* e ferramentas que fazem orquestração dos microsserviços de forma automática (e.g., Kubernetes ou Docker Swarm). A Secção 2.4 apresenta a evolução nas formas de correr aplicações e a Secção 2.5 descreve mais em detalhe o conceito da orquestração de microsserviços usando *containers*.

2.4 Evolução das formas de *deploy*

Esta secção explora mais em detalhe as várias opções existentes para fazer *deploy* e a forma como evoluíram no tempo: começando pela forma mais antiga, *deploy* em máquina física, passando depois pela virtualização, as máquinas virtuais, e finalizando com conteinerização.

2.4.1 *Deploy* em máquina física

Inicialmente as aplicações eram instaladas num servidor (partilhado ou não entre vários clientes) que tinha um sistema operativo no qual corriam as aplicações. Isto trazia vários problemas: primeiro, os recursos eram utilizados de forma muito ineficiente (por exemplo, em alturas em que um servidor tivesse pouca carga os recursos do servidor não podiam ser alocados a outras tarefas); segundo, era necessária uma pessoa para fazer a administração do servidor, os processos não eram automatizados; por último, quando era necessário escalar a aplicação, era necessário ligar uma máquina nova e dividir a carga através de algum *load balancer* de forma manual.

2.4.2 Máquinas virtuais

Na década de 50 surgiu o conceito de virtualização, e no início da década de 60 surgiu a primeira máquina virtual (VM) desenvolvida pela IBM (Randal, 2020; Chiueh and Brook, 2005). A virtualização consiste em criar um ambiente simulado dentro de uma máquina física. A Figura 2.2 ilustra o seu funcionamento.

Para correr VMs numa máquina física é necessário que a máquina tenha um sistema operativo instalado (*host operating system*) e um programa que cria e corre VMs (*hypervisor*) (e.g., VirtualBox, VMWare (Li, 2010)). Este *hypervisor* faz a gestão do acesso aos recursos (*Random Access Memory* (RAM), *Central Processing Unit* (CPU), Disco) das VMs. Desta forma, tal como ilustrado na Figura 2.2, podemos ter uma máquina física a correr

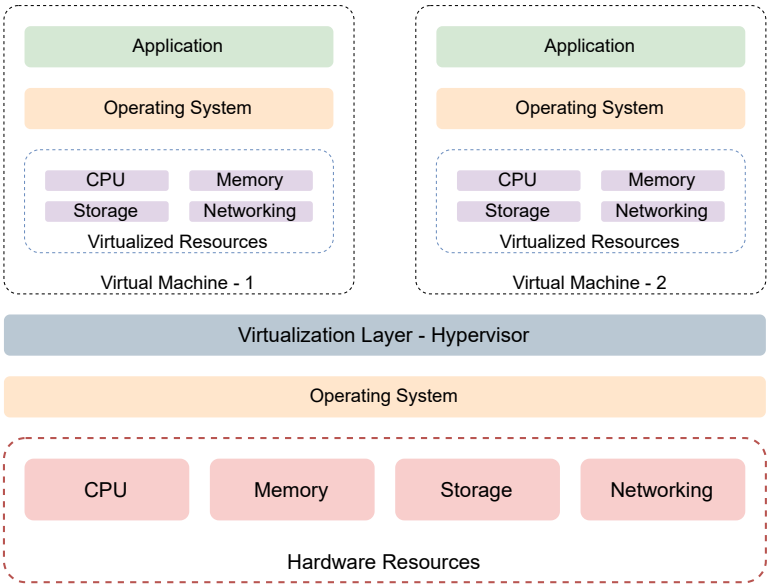

Figura 2.2: Arquitetura de máquinas virtuais

várias máquinas virtuais diferentes. Inclusivé, não há limitação para o tipo de VM que se pode correr: numa máquina Windows podemos correr várias máquinas virtuais Linux, por exemplo.

Do ponto de vista de utilização, uma VM é muito semelhante a uma máquina física, com a vantagem de que o cliente não tem de se preocupar com o *hardware*.

Desta forma, o *deployment* utilizando VMs tem muitas vantagens em relação ao *deployment* em máquina física, tais como:

1. Segurança: o facto das máquinas virtuais serem um ambiente isolado tem muitas vantagens de segurança; e.g., se uma VM for comprometida por um ataque, esses ataques, em princípio, não afetam outras VMs que possam estar eventualmente a correr nesse servidor.
2. Utilização eficiente dos recursos: tal como mencionado, numa máquina física podemos ter n VMs, possibilitando, assim, uma divisão mais eficiente dos recursos.
3. Escalabilidade: é possível escalar automaticamente uma aplicação utilizando *scripts* para instanciar novas VMs nos períodos com maior carga.

Por outro lado, a utilização de máquinas virtuais origina alguns problemas práticos, nomeadamente o *bundle size* (composto pelo sistema operativo mais a aplicação a executar com as suas dependências) e o *boot time* da VM mais o tempo de arranque da aplicação.

Adicionalmente a virtualização apresenta uma sobrecarga nos seguintes aspetos: virtualização de CPU, virtualização da memoria e virtualização de I/O. A sobrecarga da virtualização de CPU tem vindo a ser minimizada, havendo ambientes de virtualização como Xen

ou VMware que são capazes de atingir baixos custos de virtualização do CPU, uma vez que grande parte das instruções podem ser executadas dentro da VM. Por outro lado, a virtualização de I/O apresenta uma sobrecarga elevada porque os dispositivos de I/O normalmente são partilhados por todas as VMs instaladas numa máquina, o que provoca que a *Virtual Machine Monitor* (VMM) intervenha em todos os processos de I/O aumentando a latência de I/O e produzindo uma sobrecarga no CPU devido à alteração do contexto entre VMs e VMM (Huang et al., 2006). Adicionalmente as VMs dão origem a uma sobrecarga na partilha de dados entre as VMs ou entre as VMs e o *hypervisor*, exigindo, normalmente, oprações muito caras neste tipo de chamada (Felter et al., 2015).

2.4.3 *Containers*

Os conceitos de *containers* e máquinas virtuais têm origens comuns: surgiram com a mudança na arquitetura do *software* em direção ao *multiprocessing* e *multitasking* nos finais dos anos 50, início dos anos 60, e também pela motivação do isolamento dos processos, aumento da portabilidade, utilização eficiente dos recursos partilhados e minimização da complexidade em ambientes UNIX/Linux (Randal, 2020). Ao longo do tempo, o termo de *container* tem vindo a estar cada vez mais presente; por exemplo, em 1999 Banga et al. (1999) propõem um novo sistema de abstração do sistema operativo[3]: o *resource container*. "*A resource container is an abstract operating system entity that logically contains all the system resources being used by an application to achieve a particular independent activity.*" (Banga et al., 1999) O termo tem também surgido em vários trabalhos; no entanto, só mais tarde, com o lançamento do Docker em 2013, o conceito de conteinerização começou a ser amplamente usado na indústria (Merkel et al., 2014; Randal, 2020).

Os *containers* Linux são construidos em cima do *kernel namespaces*. Uma das características de Linux *namespaces* é permitir isolamento ao nível do *kernel*, possibilitando encapsular redes, recursos, processos, etc. Portanto, no caso dos *containers*, para criar o isolamento, geralmente é adicionado a cada processo um *container* ID e uma verificação de acesso para cada chamada. Desta forma, é possível criar um *container* isolado que não tem acesso a processos ou recursos fora dele próprio (Felter et al., 2015). Se por um lado uma máquina virtual contém todo o sistema operativo e virtualiza ao nível do *hardware*, um *container* pode ter apenas um único processo, fazendo com que seja bastante mais leve e compacto.

Na literatura, o isolamento dos *containers* também é conhecido como "*container-based virtualization*" ou "*lightweight virtualization*" (Sun et al., 2018; Felter et al., 2015).

A Figura 2.3 ilustra as diferenças duma aplicação que é executada numa VM e um *container*. Dois *containers* que corram na mesma máquina não sabem que estão a partilhar recursos, uma vez que cada um pode ter uma abstração da rede, dos processos, dos recursos, sistema de ficheiros etc (Felter et al., 2015).

Um *container* agrega a aplicação e todas as suas dependências, podendo assim ser instalado facilmente em qualquer ambiente (e.g., produção, testes, etc). Desta forma, os *containers*

[3]Do inglês, *Operating System* (OS)

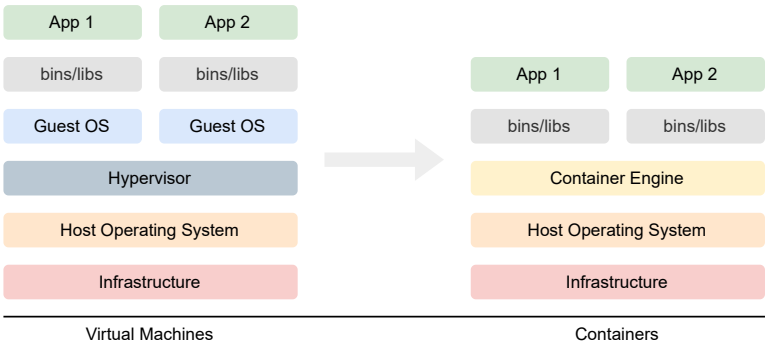

Figura 2.3: Arquitetura de máquinas virtuais e *containers*

solucionam os seguintes problemas:

- Conflitos de dependências: se quisermos correr uma aplicação que precisa de Java 8 e outra que precisa de Java 11 é bastante simples uma vez que cada aplicação corre num *container* separado.
- Dependências em falta: instalar uma aplicação num novo ambiente é direto com *containers* uma vez que não há o risco de num dado ambiente uma dependência não existir ou estar com uma versão incompatível.
- Divergência nas plataformas: se tivermos o ambiente de produção a correr os servidores com Ubuntu e o ambiente de testes com Fedora, as aplicações vão correr de igual forma desde que ambos os ambientes estejam a correr o mesmo *container engine*, e.g., Docker.

Adicionalmente, em comparação com as máquinas virtuais, os *containers* apresentam vantagens a nível de desempenho, principalmente, porque os processos são nativos do *host*. Uma vez que são mais leves, o seu tempo de inicialização é bastante inferior, fazendo com que num cenário em que seja necessário escalar horizontalmente um serviço, o seu tempo de instanciação seja menor. O desempenho e a sua uniformidade faz com que os *containers* consigam ser geridos por ferramentas de orquestração automática, discutidas na Secção 2.5.

No entanto, ao contrario das VM, os *containers* não podem aplicar políticas de segurança locais, dependendo das políticas globais definidas no *host* OS. Isto pode levar a uma falha de segurança quando são feitas chamadas de sistema que não são *namespace-aware*. Também, ao contrário das VMs, os processos que correm num *container* não têm conhecimento do limite dos recursos; como o *container* partilha o `kernel` com o OS da máquina, os processos dentro do *container* podem ver, por exemplo, todos os CPUs no sistema ou a memória (Sun et al., 2018; Felter et al., 2015). No entanto, o Docker tem mecanismos para limitar os recursos usados por cada *container* (Docker, 2013).

Para além do Docker Engine que será analisado em detalhe no Capítulo 3, existem outras ferramentas de conteinerização. São exemplo:

- Singularity: uma solução de conteinerização *open-source* frequentemente utilizada em sistemas *High Performance Computing* (HPC), ideal para computação científica (Sylabs, 2016). Mais tarde, em 2021, o projeto separou-se dando origem ao Apptainer e SingularityCE. A versão (3.8.7) estável do SingularityCE foi lançada em Março de 2022.

- Shifter: é uma alternativa ao Docker, com a possibilidade de converter as imagens Docker para imagem Shifter. Após o processo de conversão permite correr as imagens em sistemas HPC (nextflow.io, 2020).

- Podman: é uma solução *open-source*, nativa do Linux com o objetivo de facilitar a procura, execução, *build* e *deploy* de aplicações utilizando *Open Containers Initiative*. Ao contrario do Docker, o Podman tem uma arquitetura *daemon-less* permitindo assim lançar *containers* em *child processes* separados (podman.io, 2019).

- Linux Containers (LXC): Esta solução abstrai a nível do OS permitindo correr vários sistemas Linux (*containers*) no mesmo *host* tendo como base o mesmo *kernel*. "*The goal of LXC is to create an environment as close as possible to a standard Linux installation but without the need for a separate kernel.*" (Containers, 2008)

Para além das ferramentas mencionadas existem outras ferramentas de conteinerização tais como OpenVZ (OpenVZ, 2006), containerd (Containerd, 2019), Buildah (Buildah, 2018), rtk também conhecido por CoreOS Rocket (CoreOs, 2017), entre outras.

2.5 Orquestração de *containers*

A orquestração de *containers* consiste em automatizar ao máximo o esforço operacional necessário para correr aplicações conteinerizadas. Isso inclui automatizar várias operações do ciclo de vida de um *container* (Khan, 2017):

1. Instanciação ou *deployment*: instanciar um *container* com base num ficheiro de configuração.

2. Escalonamento: escalar o serviço de forma horizontal, i.e., instanciar réplicas do serviço com base numa métrica definida (e.g., quando se atinge uma determinada percentagem de CPU, RAM, pedidos por intervalo de tempo, etc). Também é necessário conseguir escalar verticalmente um serviço, i.e., alocar mais CPU ou RAM à medida que o serviço vai tendo mais carga. Este escalonamento tem que ser tanto para cima como para baixo, i.e., conseguir instanciar réplicas quando elas são necessárias e apagá-las quando o deixam de ser, para conseguir ter um uso eficiente dos recursos.

3. Rede: aquando da instanciação de um *container* este deve estar inserido automaticamente numa rede virtual definida num ficheiro de configuração.

4. Balanceamento da carga: nos casos em que um serviço está a ser prestado por 10 réplicas, a ferramenta de orquestração deve ser capaz de distribuir automaticamente a carga entre as 10 para não sobrecarregar nenhum deles. Geralmente são suportadas várias técnicas de balanceamento, como *round-robin* ou de forma aleatória.

As plataformas de orquestração mais populares hoje em dia são Kubernetes, Docker Swarm, Apache Mesos e Cattle (Al Jawarneh et al., 2019). Neste trabalho, o foco será em Kubernetes, que é a uma das tecnologias mais populares à data da escrita (Pan et al., 2019).

2.6 *Cloud Computing*

Até aqui discutimos a evolução de *"como?"* as aplicação são executadas e disponibilizadas aos clientes; nesta secção iremos discutir a evolução do *"onde?"*.

Para mitigar os problemas e as dificuldades da instalação *on-premises*, na década de 2000 surgiu o conceito de *cloud computing*. O primeiro grande *cloud-provider*, i.e. uma empresa que oferece serviços de *cloud*, foi a Amazon em 2006, juntando-se depois a Google e a Microsoft em 2008 (Bairagi and Bang, 2015; Qian et al., 2009).

Antes do lançamento dos 3 maiores *cloud providers*, qualquer empresa que pretendia disponibilizar serviços aos seus clientes necessitava ter servidores ligados à rede pública e correr as aplicações em questão localmente ou *on-premises*. No entanto, isso tem muitas desvantagens: primeiro, é necessário um grande planeamento e investimento para comprar ou alugar máquinas suficientes para aguentar a carga; adicionalmente, traz grandes custos operacionais e de manutenção; depois, caso seja um serviço global, é preciso ter geo-redundância, ou seja, ter servidores espalhados pelo mundo, para estarem mais perto dos clientes de modo a diminuir a latência e aumentar a qualidade do serviço; se por um lado uma empresa grande que tenha muito capital até pode suportar esses custos, uma pequena empresa tem imensas dificuldades em crescer desta forma.

Segundo *National Institute of Standards and Technology* (NIST) Mell et al. (2011) a computação na nuvem é um modelo que permite aos seus clientes alugar e posteriormente libertar de forma rápida e conveniente, com um baixo custo operacional, serviços de armazenamento, rede, computação, etc., como por exemplo, alugar uma máquina virtual com 2 núcleos, durante meia hora (e pagar apenas pela meia hora).

Paralelamente à *cloud*, mas sem cumprir com a definição acima, existem também fornecedores (*Data Centers*) que permitem o aluguer de máquinas físicas (OVHcloud, 1999). Este tipo de serviço é geralmente utilizado por clientes com necessidades específicas como, por exemplo, requisitos de localização das máquinas típicas em aplicações governamentais.

2.6.1 Características que definem o *cloud computing*

As características definidas pelo NIST para um *cloud provider*, i.e., uma empresa que oferece serviços de *cloud* são as seguintes:

1. *On-demand self-service*: possibilidade de alocar e libertar recursos de forma automática, sem intervenção humana.
2. *Broad network access*: serviços acessíveis através da internet pública e através de dispositivos diferentes, e.g., telemóvel, computador, etc.
3. *Resource pooling*: possibilidade do utilizador escolher a localização geográfica do recurso que quer alocar.
4. *Rapid elasticity*: possibilidade de escalar os recursos numa dada região, dando ao utilizador a impressão de que os recursos são ilimitados.
5. *Measured service*: possibilidade de monitorizar e controlar os custos de forma transparente tanto para o cliente como para o *cloud provider*.

2.6.2 Modelos de operação

Existem vários tipos de operação de *cloud*, apresentando, cada um deles, níveis diferentes de responsabilidade, retirando, também, alguma liberdade aos programadores, sendo assim necessário saber identificar qual o mais adequado. São eles:

1. *On-premises*: tal como discutido anteriormente, este é o modelo de um *datacenter* gerido pela empresa, com responsabilidades de gestão desde a rede até à aplicação.
2. *Infrastructure as a Service* (IaaS): é tipicamente o modelo em que um utilizador pode alocar recursos e escolher o sistema operativo, e.g., máquinas virtuais.
3. *Platform as a Service* (PaaS): este é um modelo de operação em que o utilizador fornece a aplicação, não pode escolher o *runtime* ou o sistema operativo mas consegue configurar várias opções e tem controlo sobre a aplicação a correr. Exemplo disso é seria um cluster de Kubernetes, em que o *cloud provider* opera o *cluster* e o utilizador limita-se a fazer *deploy* da aplicação.
4. *Software as a Service* (SaaS): este é o modelo que dá menos liberdade ao utilizador mas que lhe retira o máximo de responsabilidade de operação. O utilizador geralmente fornece o código fonte e o *cloud provider* trata de manter o serviço operacional.

A Figura 2.4 ilustra, para cada tipo, quais os serviços geridos pelo cliente e fornecedor do serviço.

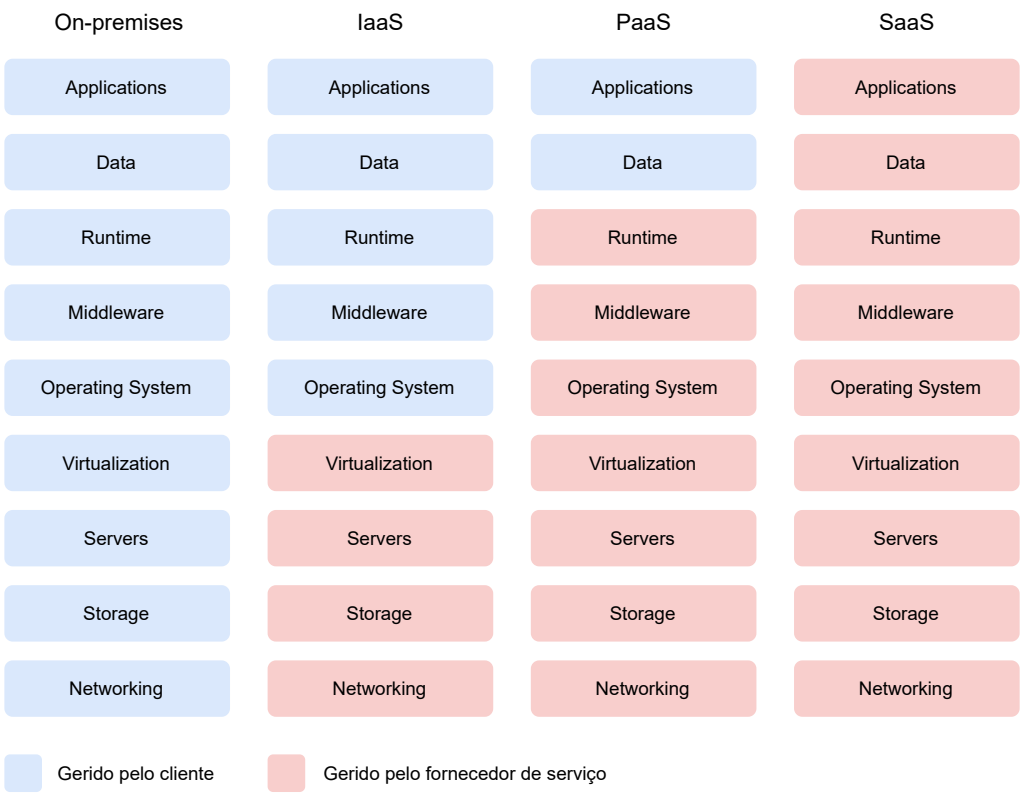

Figura 2.4: Serviços de *cloud computing*

2.7 Metodologias de trabalho

Nas últimas décadas temos vindo a assistir a uma transição nas metodologias usadas no desenvolvimento de *software*, passando de abordagens *plan-driven* (e.g., *waterfall*) para abordagens mais iterativas e ágeis (e.g., *Scrum, eXtreme programming* (Schwaber, 1997; Beck, 1999)).

In a plan-driven approach, the system increments are identified in advance; if an agile approach is adopted, the early increments are identified but the development of later increments depends on progress and customer priorities. (Sommerville, 2011)

Todas estas metodologias têm em comum quatro processos que são fundamentais na engenharia de *software*, nomeadamente: a especificação; o design e a implementação; a validação e a evolução do *software* (Sommerville, 2011). Em seguida serão abordados os modelos *waterfall* e *agile* no processo de desenvolvimento de *software*.

2.7.1 Waterfall

O modelo *Waterfall* é um exemplo de um processo *plan-driven* do ciclo de vida de um *software*. É composto por várias etapas, tal como podemos ver na Figura 2.5, nomeadamente:

- Definição dos requisitos
- Elaboração de uma solução técnica
- Implementação do *software*
- Verificação do produto e testagem
- Instalação do *software* para o ambiente de produção

onde cada etapa requer uma entrega/conclusão da fase anterior; as fases são dependentes entre elas, e só se avança para uma fase seguinte quando a anterior está concluída. O desenvolvimento em *waterfall* consiste em encadear todas as fases do processo de desenvolvimento do *software*, obtendo no final uma versão do produto.

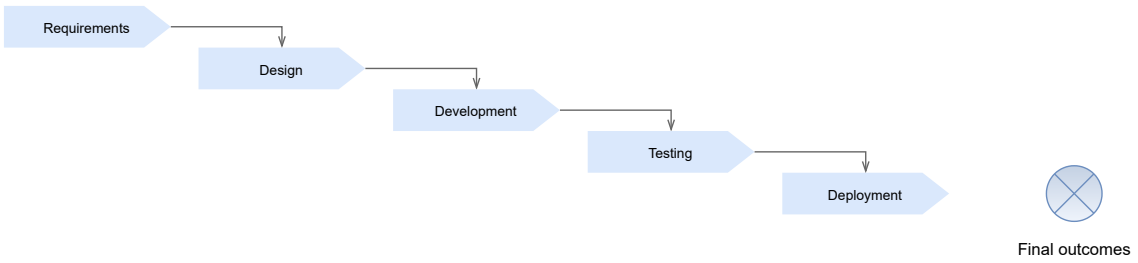

Figura 2.5: Metodologia *waterfall*

Habitualmente, cada etapa do processo é feita por uma equipa diferente. A duração deste processo depende da complexidade e dimensão do projeto. Uma das principais vantagens da metodologia *waterfall* é ter prazos e orçamentos fixos, isto porque os objetivos do projeto são especificados no início e também porque o cliente tem pouco envolvimento no processo de desenvolvimento; normalmente o *feedback* é dado na conclusão de cada fase. Por outro lado, o surgimento de problemas inesperados pode tornar complicado o desenvolvimento do projeto e levar a uma derrapagem do prazo e inclusive do orçamento. Isto porque, como as fases do processo dependem entre si, por exemplo, um problema nos requisitos pode levar a reformulação de todas as fazes seguintes. Esta metodologia é adequada para projetos com requisitos bem definidos onde o resultado final esperado é claro (Sommerville, 2011).

2.7.2 Agile

Agile é uma abordagem incremental e contém fases de desenvolvimento iguais ou semelhantes à metodologia *waterfall*. Enquanto na abordagem *waterfall* as fases dependem uma

da outra, na metodologia *agile* podem ser executadas em simultâneo, permitindo assim um rápido *feedback* ao longo das atividades. Por outras palavras, *agile* é um conceito de gestão de projetos composta por um conjunto de princípios e que incentiva uma abordagem iterativa da gestão, dá ênfase à flexibilidade em vez do seguimento de um plano rígido. O conceito *agile* é concretizado em várias metodologias de gestão como Scrum ou Kanban.

As abordagens *agile*, por exemplo a metodologia Scrum, dividem o processo de desenvolvimento em pequenos ciclos (também conhecidos por *sprints*), com duração entre 2 a 4 semanas, em que cada *sprint* contém todas as fases de desenvolvimento. No final de cada *sprint* há uma pequena entrega, fazendo assim com que o desenvolvimento do produto seja iterativo.

Este tipo de abordagem é indicado para projetos onde é previsto haver mudanças nos requisitos ao longo do desenvolvimento. Em suma, o objetivo das metodologias ágeis é entregar de forma rápida *software* funcional para que os clientes possam propor alterações aos requisitos a serem entregues em versões futuras (Sommerville, 2011). A Figura 2.6 ilustra a metodologia *agile*.

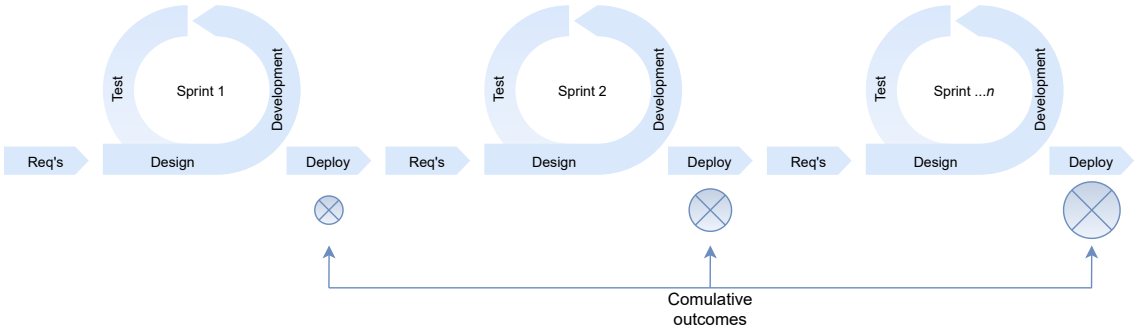

Figura 2.6: Metodologia *agile*

Devido à natureza iterativa da metodologia *agile*, os conceitos de *Continuous Integration* (CI) e *Continuous Delivery* (CD) estão frequentemente associados ao processo de entrega. Portanto, é importante introduzir estes conceitos. O conceito de CI, consiste em integrar automaticamente o código num repositório partilhado, várias vezes ao dia, passando, tipicamente, por uma fase de *build* e de testes, que valida o *software*. Este tipo de automatização é feita com a ajuda de ferramentas tais como Jenkins, Travis, TeamCity, entre outros. Estas ferramentas permitem, através de *scripts* ou de ficheiros de configuração, definir *pipelines* personalizados que criam um *workflow* de entrega automatizado e definir as etapas necessárias do fluxo de entrega (Fowler and Foemmel, 2006). Estas etapas são exploradas na Secção 2.7.3.

Paralelamente ao CI também temos CD (Neely and Stolt, 2013) e *continuos deployment* (Savor et al., 2016) que estendem o conceito anterior adicionado à automatização as seguintes etapas do processo: as etapas de *release* e *deploy*, respetivamente.

As diferenças entre metodologias enunciadas também implicam alterações substanciais na estrutura e tamanho das equipas. Devido à natureza do modelo *waterfall* em que cada

equipa é responsável por uma etapa, as empresas que trabalham neste modelo dividem as suas equipas por especialidade, ou seja, equipas de testes, de desenvolvimento, de design etc. Por contraste, nos modelos *agile*, devido ao facto de cada *sprint* englobar as várias fases de desenvolvimento, as equipas são mistas e geralmente mais pequenas, ou seja, numa equipa há pessoas que se dedicam ao *design*, desenvolvimento e testes.

Estas diferenças na abordagem ao desenvolvimento e na morfologia das equipas do modelo *agile* trazem várias vantagens comparativamente ao modelo *waterfall*, nomeadamente:

- Comunicação entre as várias fases do projeto: Estando as pessoas que desenvolvem e que testam na mesma equipa, a comunicação é intra-equipa ao contrário de inter-equipa. Essa proximidade é uma enorme vantagem e acelera muito o processo.
- Introdução de novos requisitos: É bastante mais fácil introduzir novos requisitos num modelo *agile*; basta ter isso em conta no *sprint* seguinte. Em contra-partida, no modelo *waterfall* todo o processo tem que recuar para uma etapa anterior.
- Receção de *feedback* mais cedo: a existência de entregas regulares na metodologia *agile* permite ao cliente dar *feedback* sobre o que está a ser feito numa fase mais embrionária do produto, o que torna mais fácil ajustar os requisitos.
- Noção do progresso: devido às entregas em cada *sprint*, o cliente consegue ver o ritmo ao qual está a avançar o projeto.

Apesar de todas as vantagens e das melhorias que a metodologia *agile* introduz, também apresenta algumas desvantagens relevantes:

- O resultado final pode não ser o que foi inicialmente projetado, isto porque, uma das características relevantes da metodologia *agile* é o acompanhamento pelo cliente do processo de desenvolvimento do *software* e, durante este processo, o cliente pode não ser totalmente claro e sucinto à medida que os requisitos forem incrementados.
- Pode levar a uso ineficiente dos recursos devido ao número de alterações dos requisitos, isto é, se os clientes não estiverem satisfeitos com uma entrega duma iteração e mudarem os requisitos, então essa iteração é inútil.
- A degradação do código, que tende a acontecer devido ao crescente número de incrementos, o que torna mais difícil futuros processos de manipulação ou adição de funcionalidades.

2.7.3 DevOps

No mundo empresarial, o que acontece muitas vezes é que as empresas têm equipas dedicadas que tratam de monitorizar e colocar o *software* em produção. Desta forma, as equipas que desenvolvem, quer estejam num modelo *waterfall* ou *agile*, entregam o seu código a

outra equipa que vai tratar de o colocar em produção para os clientes. Esta abordagem tem dois grandes problemas. Por um lado, as equipas de operações têm pouco conhecimento sobre o código e sobre a forma como o *software* se comporta, fazendo com que a resposta a possíveis erros em produção seja mais lenta. Por outro lado, a equipa que desenvolveu o *software* está muito desligada do aspeto operacional do seu produto. Isto é principalmente problemático porque o acesso à monitorização em produção muitas vezes diz-nos o que é necessário melhorar em *releases* futuras.

Para colmatar estas dificuldades, no final dos anos dois mil surgiu uma nova cultura de trabalho, chamada DevOps (Ebert et al., 2016; Zhu et al., 2016). DevOps é um conjunto de práticas que têm como objetivo reduzir o tempo entre fazer o *commit* duma alteração e essa alteração entrar em produção, sem pôr em causa a qualidade da entrega (Bass et al., 2015). Esta estratégia é conseguida através da integração e automatização de todas as etapas de desenvolvimento e *deployment* do *software*, desde a sua conceção até à operação e monitorização em produção. A Figura 2.7 ilustra o enquadramento do DevOps e os passos que este acrescenta em relação à metodologia *agile*.

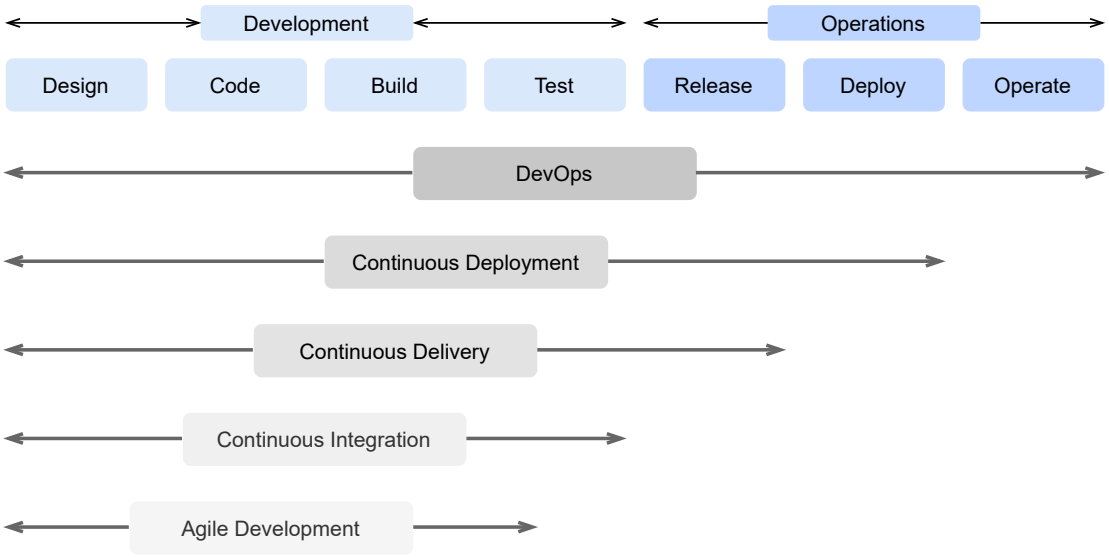

Figura 2.7: Ciclo de várias metodologias de desenvolvimento

DevOps tem vindo a impor-se na indústria como uma metodologia que agiliza e facilita a entrega de *features* e de produtos ao mercado, devido ao seu grande nível de automatismo. Esse automatismo pretende não só fazer com que o processo seja mais rápido mas também reduzir a intervenção humana que, por consequência, reduz a quantidade de erros introduzidos.

DevOps no mundo real

A adoção do DevOps nem sempre é fácil para as empresas, visto ser necessária a fusão entre o desenvolvimento (Dev) e a parte de operações (Ops) e uma mudança de menta-

lidade das pessoas (Riungu-Kalliosaari et al., 2016). Para além disso, é preciso que haja investimento na criação de *pipelines* para automatizar cada etapa do processo, tal como referido anteriormente (Leite et al., 2019). Em parte, esta transição tem sido facilitada pela quantidade de ferramentas diferentes que têm surgido. A Tabela 2.7.3 ilustra algumas delas.

Apesar das dificuldades, hoje em dia, para muitas empresas, o DevOps é já algo intrínseco à sua forma de trabalhar, tendo as maiores empresas de tecnologia (e.g., Google, Amazon, Microsoft) adotado o paradigma. Para além disso, essas empresas também são os maiores *cloud providers* do mercado, e o DevOps é integrado nativamente nas suas plataformas.

Processos automatizados

Apesar da automatização do fluxo de trabalho não ser necessariamente específica das metodologias *agile*, é nessas metodologias que são mais aplicadas devido às entregas incrementais e à forte relação com o cliente. Na cultura DevOps esta automatização de processos é ainda mais presente. Estes fluxos de trabalho[4] podem ser automatizados de modo a que um *commit* num repositório de controlo de versões (e.g., GitHub, GitLab, SVN) possa colocar uma nova versão do *software* no ambiente de produção. A Figura 2.7 ilustra os passos automatizados no DevOps.

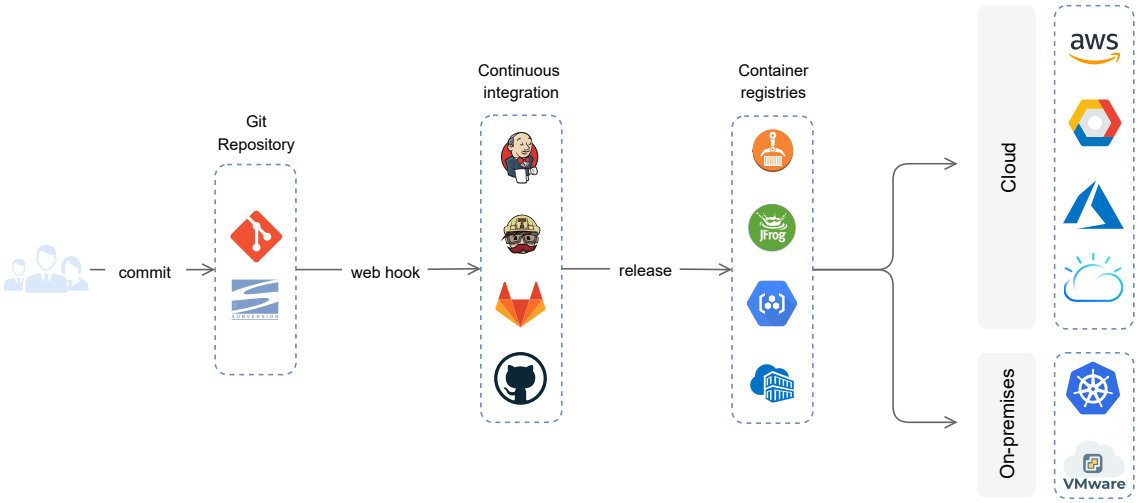

Figura 2.8: *Pipeline* DevOps

Introduz-se, de seguida, os passos de uma *pipeline* e como são habitualmente integrados:

Sistema de controlo de versões. O sistema de controlo de versões tem como uma das funções base a identificação de alterações em ficheiros. Quando são feitas alterações em ficheiros, o sistema identifica e permite fazer *commit* destas alterações. No ato de *commit* é criada uma nova versão destes ficheiros, permitindo assim seguir as

[4]Referente as *pipelines*

alterações feitas ao longo do tempo. Uma outra característica é permitir que várias pessoas trabalhem sobre o mesmo repositório.

Portanto, o sistema de controlo de versões é o ponto de entrada da *pipeline* descrita anteriormente. No ato de um *push* para o repositório remoto podem ser despoletadas as ações descritas em seguida, nomeadamente *Build, Test, Release* e *Deployment*, sem haver necessidade de intervenção humana neste processo.

Build. Habitualmente é o passo inicial da *pipeline*, podendo ser despoletado por um *commit* ou pela criação de uma *tag git*, que depois vai invocar um *webhook* para a ferramenta de *continous integration* começar a *pipeline*.

Para automatizar a etapa de *build* usa-se primeiro uma ferramenta que consiga obter o código fonte (e.g., `Git`) e posteriormente, uma ferramenta que crie um artefacto com base no código fonte. Esta última é dependente da linguagem de programação em que o código está escrito (e.g., Maven ou Gradle para Java). Se houver um erro de *build* a pipeline habitualmente pára, uma vez que o código não poderá ser corrido.

Test. Esta etapa tem como objetivo validar a qualidade do código. Habitualmente consiste em correr os testes unitários e os testes de integração para validar a funcionalidade. Opcionalmente, poderá também ser aplicada uma ferramenta de análise estática para detetar *bugs* ou más praticas (e.g., SonarQube (Campbell and Papapetrou, 2013)). Neste passo são também definidos vários limites, como por exemplo a percentagem de *code coverage*[5] ou número de testes falhados, e caso os limites não sejam cumpridos, a *pipeline* irá parar.

Release. Consiste em produzir um artefacto pronto para ser *deployed*. Este artefacto pode ser de diferentes tipos (e.g., um Jar, uma imagem Docker ou até o próprio código fonte se for uma biblioteca). Cada *release* está associada a uma versão, que a permite identificar inequivocamente.

Deployment. Esta etapa utiliza o artefacto gerado no passo anterior e faz *deploy*, que consiste em instalar e colocar em execução a nova versão do *software* num determinado ambiente (e.g., desenvolvimento, produção, etc.). Esta instalação pode ter várias formas, e.g., numa máquina virtual ou ao iniciar um novo *container* num *cluster* de Kubernetes.

Logging and Monitoring. Apesar do DevOps focar principalmente na automatização dos processos de *build* e *deploy*, é muito importante também assegurar o correto funcionamento do *software*. Para isso usam-se várias ferramentas de *logging* e de monitorização (e.g., Grafana (Grafana, 2014), New Relic (Relic, 2008)). Essas ferramentas possibilitam a recolha e visualização de métricas de vários parâmetros de funcionamento, tais como latência, número de pedidos por intervalo de tempo ou número de erros. Com base nessas métricas é possível ver se o sistema se comporta de acordo com os requisitos ou não.

[5] Percentagem de código coberto pelos testes unitários

Tool	DevOps phase	Tool type	Configuration format	License
Ant	*Build*	*Build*	XML	Apache
Maven	*Build*	*Build*	XML	Apache
Rake	*Build*	*Build*	Ruby	MIT
Gradle	*Build*	*Build*	Based on Groovy	Apache
Jenkins		*Continuous integration*	UI	MIT
Travis CI		*Continuous integration*	YAML	MIT
Git Actions		*Continuous integration*	YAML	Commercial/Free
Chef	*Deployment*	*Configuration management*	Ruby-based DSL	Apache
Ansible	*Deployment*	*Configuration management*	YAML	GPL (GNU General Public License)
Loggly	Operations	Logging		Commercial
GrayLog	Operations	Logging		Open Source
Nagios	Operations	Monitoring		Commercial
New Relic	Operations	Monitoring		Commercial
Cacti	Operations	Monitoring		GPL
Grafana	Operations	Monitoring	UI, YAML	

Tabela 2.1: Ferramentas DevOps

Capítulo 3

Metodologia de conteinerização

Com base no estudo efetuado no Capítulo 2, este capítulo irá analisar de forma mais técnica as tecnologia Docker e Kubernetes (uma vez que foram estas as ferramentas usadas no trabalho) e propor uma metodologia para conteinerizar uma aplicação.

3.1 Docker

Docker é uma plataforma de conteinerização *open-source* que permite separar as aplicações da infraestrutura de modo a agilizar a entrega de *software*. Esta separação é possível através de *containers* que correm a aplicação em um ambiente isolado.

Quando se fala de Docker, é importante compreender os conceitos de *Docker Image* e *Docker Container*. Uma *Docker Image* é criada a partir de um ficheiro, chamado Dockerfile que contém um conjunto de instruções (que serão analisadas mais à frente nesta secção).

Após executar o comando de *build* do Dockerfile o resultado é uma imagem. Uma imagem é um ficheiro imutável (ficheiro apenas de leitura) que contém todos os ficheiros necessários para uma aplicação ou serviço, e.g., código fonte, bibliotecas, dependências, etc... Por sua vez, um *container* é um ambiente *run-time* isolado e que representa uma instância de uma imagem. Podem ser criados vários *containers* a partir da mesma imagem, sendo isto útil quando temos um serviço com uma carga elevada. Neste caso levanta-se mais um *container* para distribuir o tráfego através de um *load balancer*. Na Figura 3.1 são exemplificados os passos desde o Dockerfile até ao *container* (Docker, 2013).

Desta forma, para criar um *container* precisamos de três elementos. O Excerto de Código 3.1 contém um exemplo de um Dockerfile duma aplicação em Python:

1. Dockerfile: um ficheiro que contém uma série de instruções para criar uma imagem. No exemplo, a linha 1 define qual a versão da sintaxe a usar, seguida da especificação da plataforma e a sua versão (neste caso é um ambiente Linux, que já traz o Python instalado); na linha 3 copia-se tudo do diretório atual para o diretório `/app` dentro da

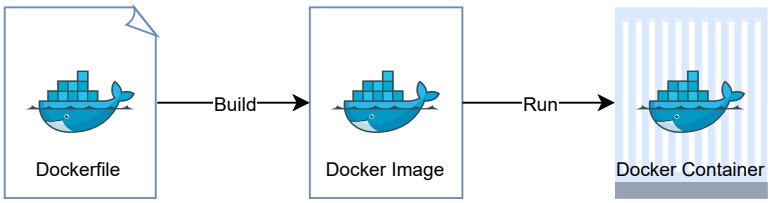

Figura 3.1: Etapas para criar um Docker *Container*

imagem; por fim, na linha 4, compila-se o código e na linha 5 a aplicação é executada.

2. Contexto: representa aquilo que será conteinerizado, geralmente é uma aplicação.
3. Execução dos seguintes comandos:
   (a) `docker build` : comando que assume a existência de um Dockerfile presente na pasta corrente e faz *build* duma imagem;
   (b) `docker run` *id-imagem*: para levantar o *container*

```
1 # syntax=docker/dockerfile:1
2 FROM python:3
3 COPY . /app
4 RUN make /app
5 CMD python /app/app.py
```

Excerto de Código 3.1: Exemplo de Dockerfile duma aplicação Python

3.2 Kubernetes

Kubernetes é um orquestrador de *containers* (Docker, Podman, entre outros) *open-source* originalmente desenvolvido pela Google e mantido atualmente pela Cloud Native Computing Foundation (Sayfan, 2017; Burns et al., 2016). Um *cluster* de Kubernetes pode ser instalado tanto em máquinas físicas, i.e., diretamente no *hardware*, ou em máquinas virtuais. Um requisito necessário é existir interconetividade entre as máquinas, uma vez que a arquitetura do *cluster*, ilustrada na Figura 3.2, é dividida em nós que precisam de comunicar entre si. Existe também um painel de controlo que tem a lógica de orquestração.

Os principais elementos que compõem a arquitetura de Kubernetes são:

- **Painel de Controlo / Master.** É a unidade que controla as operações de *scheduling* e que reage a eventos, e.g., quando é detetada demasiada carga e é necessário instanciar uma nova réplica. Os componentes que o constituem são os seguintes:
  1. API Server: é o componente principal que expõe as APIs de gestão do *cluster*;
  2. Scheduler: monitoriza a criação de novos *pods* e atribui-lhes um nó;

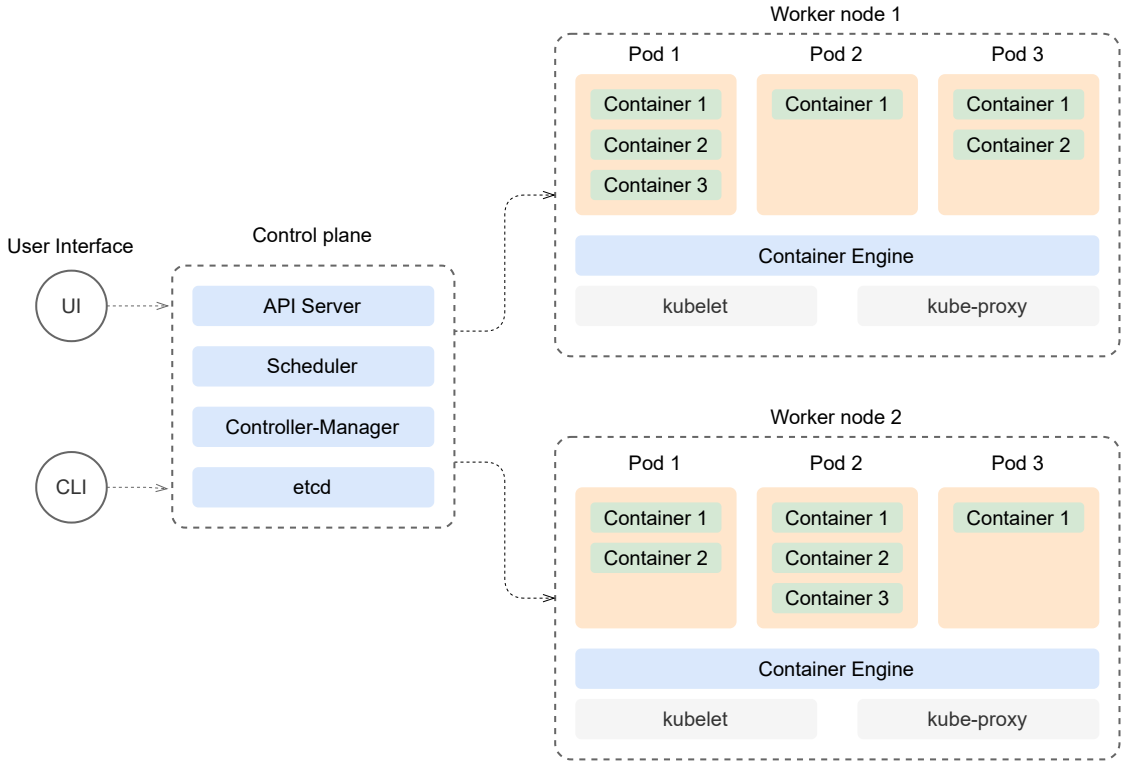

Figura 3.2: Arquitetura do Kubernetes

3. Controller Manager: monitoriza o estado do *cluster*, com o objetivo de o manter saudável, e.g., se um nó for abaixo por alguma razão, outro nó é automaticamente instanciado se houver recursos disponíveis
4. `etcd`: uma base de dados *key-value* que mantém os dados do *cluster*.

- Nó: pode ser uma máquina virtual ou física, e tem como objetivo correr *pods*. Um nó é composto pelos seguintes componentes:

1. Pod: é uma unidade de trabalho. Cada *pod* é composto por um ou mais *containers*, tendo todos o mesmo *Internet Protocol* (IP). Adicionalmente todos os *containers* dentro de um *pod* têm acesso a um armazenamento partilhado entre eles;
2. `kubelet`: agente presente em todos os nós, cujo objetivo é assegurar que todos os *pods* estão a correr;
3. kube-proxy: responsável por manter comunicação dentro dos elementos do *cluster*;
4. Container Engine: é um *runtime* que vai de facto correr os *containers* dentro dos *pods*. Por exemplo se os *containers* forem Docker, então o container enginer terá que ser Docker também.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

Excerto de Código 3.2: Exemplo de configuração de um Pod/Deployment

Para fazer *deploy* de uma aplicação em Kubernetes é necessário definir os seguintes elementos em ficheiros YAML:

1. Pod: define quais os containers a correr, as portas que deve expor, o nome do *pod*. Também podemos definir para cada *container* requisitos e limites de recursos, e.g., dizer que um determinado *container* precisa de 100Mb de RAM para correr e que no máximo pode chegar a 200Mb de RAM. Com esta informação o Kubernetes sabe que se ultrapassar esse limite o *pod* deve ser apagado. O Excerto de Código 3.2 mostra um exemplo de uma especificação de um *Deployment*, em que adicionalmente se pode especificar o número de réplicas, na linha 8.

2. Serviço: a definição deste elemento permite expor uma aplicação que corre numa série de *pods*, como um serviço de rede. O Excerto de Código 3.3 apresenta um exemplo duma declaração de um serviço que expõe o *deployment* discutido anteriormente.

3. Ingress: permite expor um serviço para fora do *cluster*.É útil quando queremos que os clientes tenham acesso a uma aplicação externa. O Excerto de Código 3.4 mostra um exemplo de um `yaml` que declara um *Ingress* que expõe o *nginx-service*.

A Figura 3.3 representa um *cluster* com 3 aplicações expostas na internet. Cada uma das aplicações é representada por um serviço com 3 *pods*.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```

Excerto de Código 3.3: Exemplo de configuração de um Serviço

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: nginx-ingress
spec:
  defaultBackend:
    service:
      name: nginx-service
      port:
        number: 80
```

Excerto de Código 3.4: Exemplo de configuração de um *Ingress*

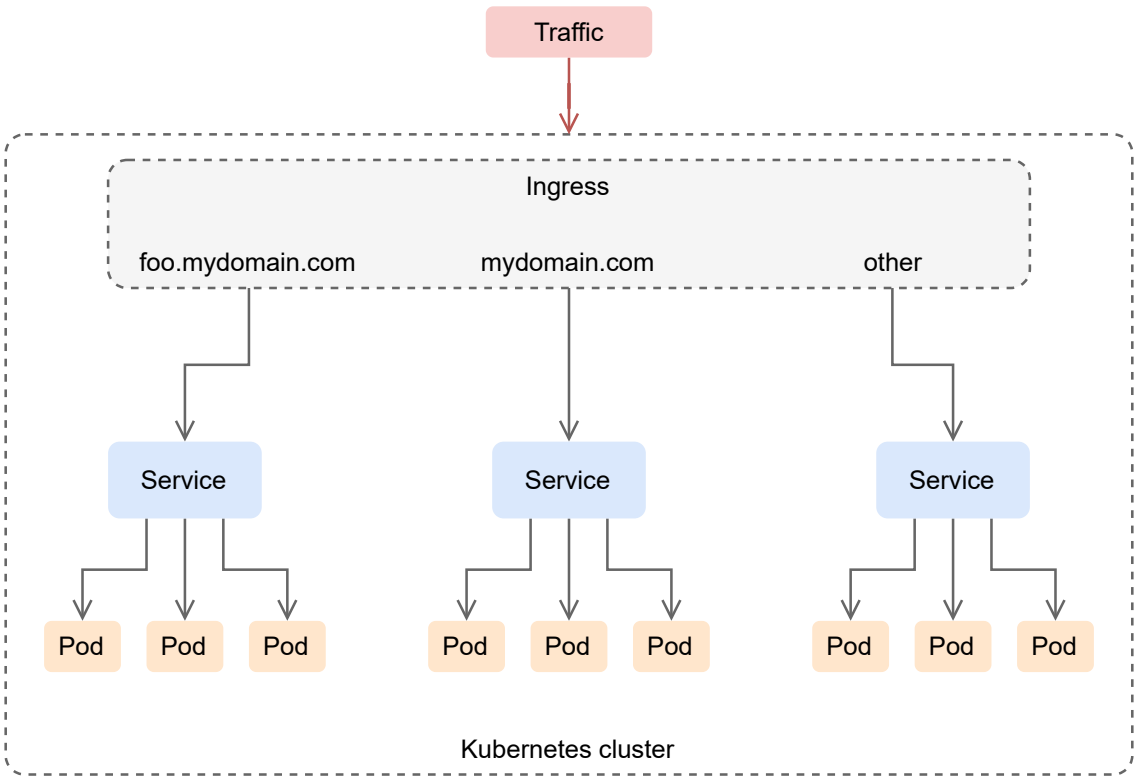

Figura 3.3: Componentes de uma aplicação a correr em Kubernetes

3.3 Metodologia de conteinerização

Esta Secção tem como objetivo, com base no estudo efetuado anteriormente, definir uma metodologia genérica para a conteinerização de aplicações.

A conteinerização tem que ser agnóstica à lógica do código, ficando apenas vinculada às tecnologias usadas para conteinerizar os serviços/aplicações. Adicionalmente, houve o desafio de estudar a possibilidade da existência de um Dockerfile genérico, que conseguisse ser usado para todos os serviços, evitando assim a repetição, em parte, do Dockerfile para cada serviço.

Após o estudo da tecnologia Docker e de como definir um Dockerfile concluí que não é possível ter um Dockerfile genérico que se aplique a todas as aplicações em diversas tecnologias. Isto porque, em primeira instância, é necessário definir a *keyword* `FROM` que define o ambiente do serviço. No exemplo do Excerto de Código 3.1 tem-se `FROM python:3`, mas poderia ser, por exemplo, `FROM php:5.6-fpm`, dependendo da tecnologia da aplicação a conteinerizar. Portanto, para um serviço em Python será necessário uma imagem base que tenha o Python instalado, e para um serviço em PHP será necessário uma outra imagem base que tenha o PHP instalado. Uma outra razão desta impossibilidade é que as aplicações em tecnologias diferentes têm dependências diferentes e podem necessitar de configurações de ambiente, também elas diferentes.

Desta forma, concluiu-se que só é viável ter um Dockerfile único genérico para todos os microsserviços quando estes apresentam diferenças mínimas entre eles. Ter um Dockerfile por microsserviço tem várias vantagens:

1. É mais fácil manter e compreender o Dockerfile, e implicitamente, a imagem que vai ser construída;
2. Não existe o risco de alterar o ficheiro e corromper o processo de *build* de outro serviço;
3. Estando o código no mesmo repositório que o Dockerfile facilita a construção de *pipelines*, uma vez que não é necessário ir buscar esse ficheiro a sítios remotos.

Portanto, para definir um Dockerfile, é preciso analisar, primeiramente, a aplicação a conteinerizar com o intuito de reunir os requisitos necessários, que são:

1. Identificar a tecnologia em que o serviço é desenvolvido, e.g., Python;
2. Identificar o tipo de servidor em que o microsserviço corre, e.g., Nginx, Apache;
3. Analisar as dependências necessárias para adicionar ao ambiente, e.g., instalar `pip` de modo a conseguir instalar as dependências de Python definidas no ficheiro `requirements.txt`. Analogamente para React instalar o *package manager* `Npm`;
4. Conhecer os portos necessários a expor na rede, de forma a que o microsserviço consiga receber conexões nesse porto, e.g., expor o porto 80 para ligações HTTP;
5. Saber o comando a invocar quando o *container* Docker arranca.

Posteriormente é necessário traduzir a análise obtida para a sintaxe do Docker. Para os serviços em Python um Dockerfile básico genérico poderia ser o indicado no Excerto de Código 3.5 e para Java no Excerto de Código 3.6.

```
1 FROM python:3.5 #tecnologia do serviço
2 WORKDIR /usr/src/app
3 COPY . /usr/src/app #copiar a aplicação para dentro do container
4 #instalar as dependências da aplicação
5 RUN pip install -r /usr/src/app/requirements.txt
6 #expor o porto 8080
7 EXPOSE 8080
8 #instalar a aplicação e correr o respetivo servidor - tornado
9 CMD python /usr/src/app/setup.py install && tornado
```

Excerto de Código 3.5: Dockerfile de um serviço em Python

```
1 #tecnologia do serviço
2 FROM openjdk:16-alpine3.13
3 WORKDIR /app
4 COPY .mvn/ .mvn
```

```
5 COPY mvnw pom.xml ./
6 COPY src ./src
7 EXPOSE 8080
8 #build do projeto e correr o servidor do spring-boot com o serviço
9 CMD ["./mvnw", "spring-boot:run"]
```

Excerto de Código 3.6: Dockerfile de um serviço em Java

Como podemos observar nos Excertos de Código 3.5 e 3.6, ao ter a informação necessária para conteinerizar uma aplicação, o processo torna-se relativamente simples: escolher a partir de que imagem começar o processo de conteinerização (e.g. `FROM openjdk:16-alpine3.13` para Java, ver o Excerto de Código 3.6), copiar os ficheiros necessários para dentro da imagem, instalar as dependências necessárias e fazer *build* da aplicação, expor os portos necessário e por último correr o comando para executar a aplicação.

Após a conteinerização de uma aplicação com uma arquitetura de microsserviços surge a necessidade de orquestrar estes microsserviços, isto é, automatizar os processos de: *deployment*, *auto-scaling*, *load balancing* e rede. Para este propósito será utilizada a plataforma Kubernetes (descrita na Secção 3.2).

Uma vez que o objetivo do trabalho deste trabalho foi estudar uma metodologia de conteinerização de microsserviços surge a possibilidade de, no futuro, estudar uma metodologia de orquestração de microsserviços de modo a tirar o máximo partido da tecnologia e automatizar os processos acima mencionados. Neste trabalho serão utilizadas as normas presentes na documentação de Kubernetes.

Capítulo 4

Aplicação da metodologia a um caso prático

Este Capítulo descreve a parte prática do trabalho, começando por enquadrar o trabalho e introduzir o ambiente técnico. De seguida, são apresentadas as metodologias de trabalho e a aplicação da metodologia de conteinerização resultante do Capítulo anterior a um caso prático.

4.1 Ambiente Técnico

Por questões de confidencialidade apenas é possível indicar que o *software* para aplicação da metodologia proposta consiste numa plataforma de monitorização de métricas para *cloud*, servidores, *containers* e *websites*. A aplicação apresenta uma arquitetura em microsserviços, programados em Python 2.7/3.5, PHP e React. Estes microsserviços correm em máquinas virtuais, utilizando Nginx e PHP-Fpm; a comunicação entre os serviços é feita utilizando REST (HTTP).

A plataforma é comercializada como *software* como um serviço (SaaS), conta, presentemente, com mais de 700 clientes e supervisiona dezenas de biliões de métricas diariamente.

A nível técnico, o trabalho foi desenvolvido em ambiente Linux, utilizando Git, Visual Studio Code e Docker.

4.2 Objetivos e metodologia de trabalho

O objetivo principal é uniformizar a forma como os microsserviços correm, passando de máquinas virtuais para *containers*. Para isto, será necessário conteinerizá-los usando Docker.

Para proceder à uniformização começou-se por estudar a aplicação, a sua arquitetura e as tecnologias usadas na sua conceção. A Figura 4.1 apresenta a arquitetura da aplicação.

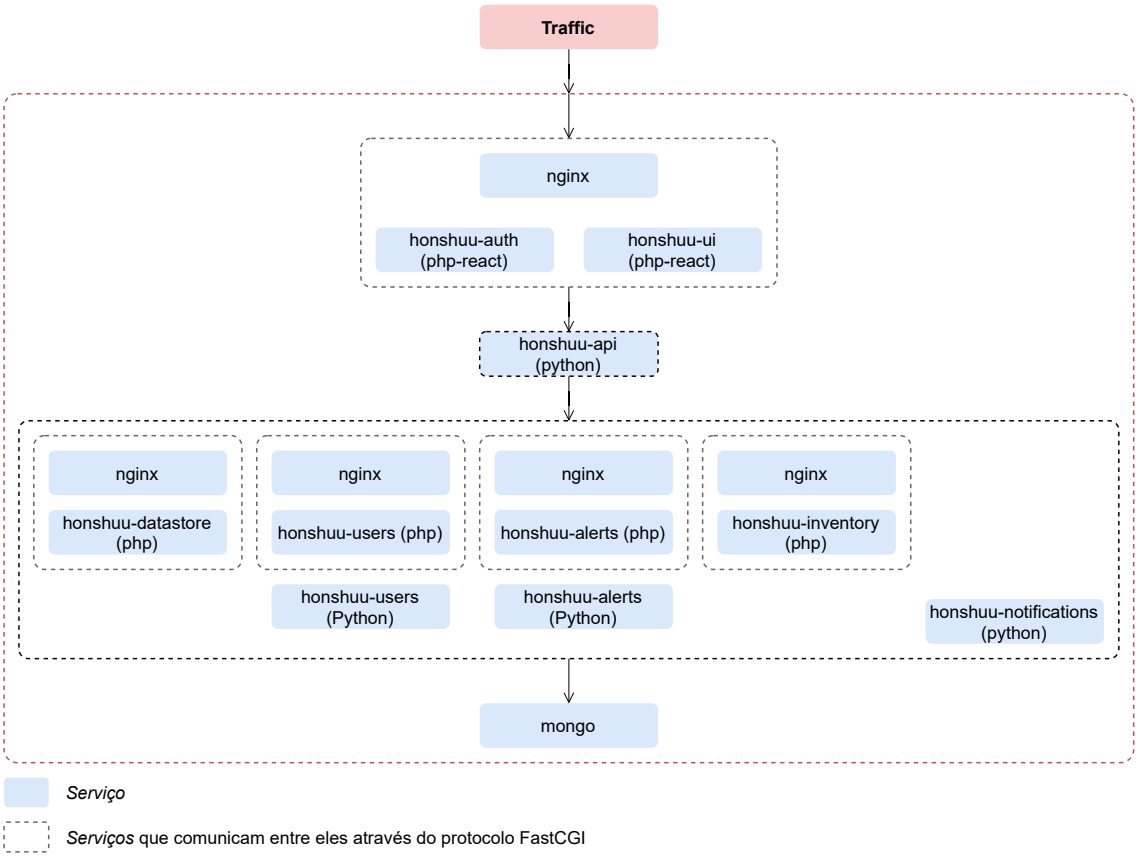

Figura 4.1: Arquitetura da aplicação

Inicialmente foi considerada a opção de fazer um Dockerfile genérico em que fosse possível definir a imagem base, as dependências e os *scripts* de início de sessão. Uma solução possível consiste na utilização de um ficheiro de *script bash* para parametrizar esse Dockerfile genérico. No entanto, chegou-se à conclusão que para microsserviços diferentes o esforço de utilizar o Dockerfile definindo todas as variáveis e o ficheiro *bash*, seria igual ou superior ao esforço de configurar diretamente um Dockerfile no microsserviço. Para além disso, a forma como se constrói a imagem ficaria acoplada ao Dockerfile genérico.

Posteriormente, diminuiu-se o *scope* e tentou-se fazer um Dockerfile genérico para os microsserviços que usam a mesma tecnologia, e.g., serem programados na mesma linguagem. No entanto, as desvantagens mantinham-se, uma vez que a diferença de versões das dependências limitava os benefícios do Dockerfile genérico.

4.3 Desenvolvimento prático

Depois de chegar à conclusão que não seria possível usar apenas um Dockerfile, optou-se por definir um por microsserviço. Desta forma evitaram-se algumas dificuldades como o conflito de dependências.

De seguida, são analisados alguns exemplos de Dockerfiles definidos aplicando a metodologia descrita na Secção 3.3.

4.3.1 Dockerfile para Python

O Excerto de Código 4.1 representa um dockerfile de um dos componentes em Python da aplicação. Como se pode observar, a primeira linha define a versão de Python, neste caso 2.7.9. Essa imagem é essencialmente uma imagem Linux que traz o Python instalado. De notar também na segunda linha a definição do argumento *GITHUB_ TOKEN*, que é passado no momento da criação da imagem e tem como propósito autenticar o *download* de dependências. Nas restantes linhas atualizam-se os *packages* do Linux. Na linha 7 é copiada a aplicação para dentro da imagem e depois são instaladas as dependências nas linhas 10-12 a partir do ficheiro `requirements.txt`, presente nos serviços Python. Por fim, é definido na linha 14 o comando que irá iniciar a aplicação aquando do arranque do *container*.

```
1  FROM python:2.7.9
2  ARG GITHUB_TOKEN
3  RUN apt-get update
4  RUN pip install --upgrade pip
5  RUN pip install typing
6  WORKDIR /usr/src/app
7  COPY . /usr/src/app
8  RUN echo -e "machine github.com\n\tlogin ${GITHUB_TOKEN}" \
9      >> ~/.netrc
10 RUN sed -i 's/git+ssh:\/\/git@/git+https:\/\//' \
11     /usr/src/app/requirements.txt && \ pip install \
12     --no-cache-dir -r /usr/src/app/requirements.txt
13 EXPOSE 8097
14 CMD python /usr/src/app/setup.py install && tornado
```

Excerto de Código 4.1: Dockerfile de um microsserviço em Python

4.3.2 Dockerfile para PHP

O Excerto de Código 4.2 mostra um Dockerfile de um microsserviço feito em PHP. Este microsserviço tem também uma componente de *User Interface* (UI) feita em React. Desta forma, esta imagem é construída utilizando outras duas imagens: *node:8.2.1* para compilar a aplicação React e *php:5.6-fpm* que contém o servidor e o *runtime* de PHP.

```
FROM node:8.2.1 AS node
FROM php:5.6-fpm
ARG GITHUB_TOKEN
ARG NEW_RELIC_URI
RUN apt-get update && apt-get install -y \
        git \
        libssl-dev \
    && docker-php-ext-install mbstring \
    && docker-php-ext-enable mbstring \
    && pecl install mongo xdebug-2.5.5 \
    && docker-php-ext-enable mongo xdebug \
    && apt-get clean

COPY --from=node /usr/local/lib/node_modules /usr/local/lib/node_modules
COPY --from=node /usr/local/bin/node /usr/local/bin/node
RUN ln -s /usr/local/lib/node_modules/npm/bin/npm-cli.js /usr/local/bin/npm

RUN mkdir /var/www/html/honshuu-auth
COPY . /var/www/html/honshuu-auth
WORKDIR /var/www/html/honshuu-auth

RUN npm install && npm run build

RUN php composer-bootstrap.php && \
    bin/composer config --global github-oauth.github.com ${GITHUB_TOKEN} && \
    bin/composer install

RUN ln -s /var/www/html/honshuu-auth/ /var/www/html/honshuu-auth/login
RUN ln -s /var/www/html/honshuu-auth/ /var/www/html/honshuu-auth/forgotten

COPY auth_www.conf /usr/local/etc/php-fpm.d/

RUN \
  curl -L ${NEW_RELIC_URI} | tar -C /tmp -zx && \
   export NR_INSTALL_USE_CP_NOT_LN=1 && \
    export NR_INSTALL_SILENT=1 && \
     /tmp/newrelic-php5-*/newrelic-install install && \
      rm -rf /tmp/newrelic-php5-* /tmp/nrinstall*
```

Excerto de Código 4.2: Dockerfile PHP

4.3.3 Dockerfile para Nginx

Por fim, o Excerto de Código 4.3 ilustra um Dockerfile de um servidor Nginx. Este Dockerfile é bastante simples, definindo apenas algumas configurações, e.g *nginx.conf*, *proxy.conf*.

```
FROM nginx:1.21.3
RUN rm -f etc/nginx/conf.d/*.conf
COPY datastore_rewrite_rules /etc/nginx/
COPY nginx.conf etc/nginx/nginx.conf
```

```
5 COPY proxy.conf etc/nginx/conf.d/
```

Excerto de Código 4.3: Dockerfile Nginx

4.3.4 Docker Compose

Depois de conteinerizados os microsserviços procedeu-se ao teste e validação do comportamento esperado. Para isso, optou-se por utilizar Docker Compose, que é uma ferramenta que permite correr uma aplicação como um conjunto de *containers* Docker Turnbull (2014).

Para utilizar o Docker Compose é necessário configurar o ficheiro *docker-compose.yaml*, representado no Excerto de Código 4.4. No ficheiro definimos quais os serviços que vão compor a aplicação (daí o nome *docker compose*); no exemplo apresentado são definidos 3 dos serviços da aplicação. Adicionalmente também se define uma rede para especificar como é que os serviços poderão comunicar entre si. Por omissão, os serviços podem comunicar entre si através do nome que define o serviço.

A título de exemplo, podemos observar na linha 4 a definição de um serviço chamado *nginx-inventory*. Para definir o serviço configuramos vários parâmetros, tais como:

- o nome do serviço, imagem e container: linhas 4 e 5, respetivamente;
- a política de reiniciar o *container*: linha 6, que diz que o *container* não deve ser reiniciado;
- os parâmetros de *build* da imagem: linhas 7-9.
- os *containers* dos quais depende: linhas 10 e 11. Este parâmetro serve para indicar quais os serviços que necessitam ser arrancados primeiro;
- definições de rede: linhas 12-15. Neste caso apenas se adiciona um *alias* do serviço à rede *main*. Este *alias* permite aos outros serviços dentro da mesma rede encontrar este serviço através dos *hostnames nginx-inventory* e *inventory*.

Os serviços *honshuu-inventory*, *mongodb* e os restantes que, por uma questão de espaço não estão presentes no Excerto de Código 4.4, são configurados de forma análoga.

As variáveis comuns a vários serviços, e.g., *GITHUB_ TOKEN*, são declaradas num ficheiro à parte, o ficheiro *.env*, no mesmo diretório em que se encontra o *docker-compose.yaml*.

Com a aplicação a correr localmente, foram feitos pedidos a cada microsserviço individualmente e também ao fluxo da aplicação como um todo, para garantir que o comportamento era mantido. Estes testes foram possíveis porque a aplicação cria um utilizador por omissão quando arranca.

```yaml
version: "3.7"
services:
    nginx-inventory:
        image: nginx-inventory
        container_name: nginx-inventory
        restart: unless-stopped
        build:
            context: ../honshuu-inventory/nginx
            dockerfile: Dockerfile
        depends_on:
            - honshuu-inventory
        networks:
            main:
                aliases:
                    - inventory

    honshuu-inventory:
        image: honshuu-inventory
        container_name: honshuu-inventory
        build:
            context: ../honshuu-inventory/
            dockerfile: Dockerfile-php
            args:
                GITHUB_TOKEN: ${GITHUB_TOKEN}
        depends_on:
            - mongodb
        environment:
            - HONSHUU_ENV=${HONSHUU_ENV}
        networks:
            main:

    mongodb:
        image: mongo
        container_name: mongodb
        networks:
            main:
                aliases:
                    - mongo
networks:
    main:
```

Excerto de Código 4.4: Excerto de *docker-compose.yaml*

4.3.5 Pipeline Git Actions

Para além de conteinerizar a aplicação, foi lançado o desafio de automatizar o processo de criação e publicação das imagens Docker num *registry* da Google Cloud (Google Container Registry), i.e., um repositório de imagens Docker. Desta forma, optámos por usar Git Actions como tecnologia de CI, uma vez que os microsserviços já estavam em repositórios no GitHub, sendo a integração mais transparente.

Para configurar uma *pipeline* em Git Actions é necessário primeiro ter uma estrutura de pastas *.github/workflows*, onde se encontra o ficheiro `yaml` de Git Actions. Nesse ficheiro, apresentado no Excerto de Código 4.5, definem-se os seguintes aspetos:

1. Nome da *pipeline*: linha 1;
2. Quando a *pipeline* deve ser despoletada: linhas 3-6. Neste caso, quando ocorre um *commit* para o *main branch*;
3. Variáveis de ambiente: linhas 8-10. Estas são, por exemplo, as chaves para aceder a repositórios ou valores usados em mais do que um sítio;
4. Tarefas que constituem a *pipeline*: linhas 12-32. Estas tarefas correm de forma síncrona e sequencial dentro do contexto duma imagem Docker. Nesta secção podemos definir o seguinte:
   (a) Nome da tarefa: linha 13;
   (b) Imagem Docker em que a tarefa vai executar: linha 15. Por exemplo, *ubuntu-latest*;
   (c) Um conjunto de passos que constituem a tarefa. Neste caso, temos 4 passos (*steps*), em que o primeiro, (*Checkout*), nas linhas 19-20, vai buscar o código do respetivo repositório em que a pipeline se encontra; de seguida, nas linhas 22-28, é feita a autenticação no Google Cloud. Nos últimos dois passos (*Build* e *Publish*) das linhas 30 e 35, é criada a imagem Docker, utilizando o Dockerfile presente no repositório, e de seguida a imagem é publicada no Google Container Registry.

Antes de correr a *pipeline* também foi necessário definir um conjunto de *secrets*. Este mecanismo de *secrets* tem a grande vantagem de não ter chaves de autenticação diretamente no código, visíveis a qualquer pessoa que tem acesso ao repositório. Estes *secrets* podem ser definidos ao nível do ambiente, repositório ou organização, sendo que estes três níveis são hierárquicos, podendo ter chaves comuns a todos os repositórios ao nível de organização.

4.3.6 Dificuldades encontradas

Ao longo da realização deste trabalho foram encontrados vários obstáculos, tais como:

```yaml
name: Build and Publish Docker Image

on:
  push:
    branches:
    - main

env:
  PROJECT_ID: ${{ secrets.GKE_PROJECT }}
  IMAGE: service-image

jobs:
  setup-build-publish-deploy:
    name: Setup, Build and Publish
    runs-on: ubuntu-latest
    environment: production

    steps:
    - name: Checkout
      uses: actions/checkout@v2

    - uses: google-github-actions/setup-gcloud@v0.2.0
      with:
        service_account_key: ${{ secrets.GKE_SA_KEY }}
        project_id: ${{ secrets.GKE_PROJECT }}

    - run: |-
        gcloud --quiet auth configure-docker

    - name: Build
      run: |-
        docker build \
          --tag "gcr.io/$PROJECT_ID/$IMAGE:$GITHUB_SHA" .

    - name: Publish
      run: |-
        docker push "gcr.io/$PROJECT_ID/$IMAGE:$GITHUB_SHA"
```

Excerto de Código 4.5: Pipeline CI/CD em Git Actions

- *Hardcoded links*: encontraram-se muitos *links* que estavam diretamente definidos no código, não sendo configuráveis por ambiente. Temos como exemplo disso:
  - Links *deprecated* para bibliotecas, que foi preciso atualizar;
  - Endereços no código que em ambiente de desenvolvimento estavam como *localhost*
- Dependências inexistentes: alguns serviços em Python apontavam para versões de bibliotecas que já não se encontravam disponíveis *online*. Para resolver este problema, foi necessário recuperar a dependência no repositório do projeto, e atualizar o ficheiro *setup.py*;
- Ficheiros de Nginx em linguagem *Puppet*: inicialmente os ficheiros de configuração de Nginx foram disponibilizados em linguagem *Puppet*, que se revelou uma dificuldade devido à curva de aprendizagem para dominar a tecnologia. Posteriormente, por uma questão de facilidade, utilizaram-se os ficheiros gerados de nginx do ambiente de produção.
- Divergências entre o ambiente em máquinas virtuais e conteinerizado: atualmente, no ambiente de produção, os serviços de PHP utilizam o nginx e o Php-Fpm que se encontram ambos na mesma máquina. Na nova versão em *containers* os serviços foram separados, i.e., neste momento, o serviço de Php-Fpm utiliza um Nginx que se encontra noutro *container*. Ora, isso revelou-se um problema, uma vez que o *path* do *script* de PHP era referenciado de forma absoluta, que funciona quando se está na mesma máquina, mas não em máquinas/contextos diferentes. A solução, presente no Excerto de Código 4.6, foi adicionar o parâmetro `$fastcgi_script_name`.

```
1 fastcgi_param SCRIPT_FILENAME  \
2         /var/www/html/honshuu-auth$fastcgi_script_name;
```

Excerto de Código 4.6: Fix para referenciar o script de forma relativa

Capítulo 5

Conclusões e Trabalho Futuro

O presente Capítulo apresenta as conclusões finais deste trabalho e também ilustra as possíveis oportunidades de trabalho futuro.

5.1 Conclusões

Esta dissertação teve como objetivo estudar uma metodologia de conteinerização de microsserviços e aplicá-la a uma aplicação que corre em máquinas virtuais. Inicialmente, tentou-se encontrar uma forma genérica para conteinerizar aplicações; no entanto, depois de algum estudo, chegou-se à conclusão de que a abordagem não teria muitas vantagens e que inclusivé iria adicionar complexidade em alguns casos.

Desta forma, os microsserviços que compõem a aplicação foram conteinerizados seguindo uma série de passos, descritos neste documento. Para além disso, o ambiente foi replicado localmente usando Docker Compose. Adicionalmente, foi também automatizado o processo de criação e publicação das imagens Docker de cada microsserviço utilizando uma *pipeline* de Git Actions.

Em suma, a conteinerização revela-se a ser uma alternativa mais leve e prática às máquinas virtuais, não só pela vantagem a nível de performance mas também pela uniformização que traz aos ambientes. Podemos também afirmar que o esforço de conteinerizar uma aplicação é relativamente baixo; no entanto, a complexidade pode ser superior em aplicações com tecnologias *legacy*. Essencialmente, conteinerizar uma aplicação consiste em conhecer as tecnologias em que a aplicação é desenvolvida, a forma como a aplicação interage na rede, e as suas dependências e, posteriormente traduzir esta informação para a sintaxe do Docker e correr os respetivos comandos para construir a imagem e posteriormente instanciar o *container*.

5.2 Trabalho futuro

Este trabalho cria espaço para algum trabalho futuro, nomeadamente, correr a aplicação num *cluster* de Kubernetes, o que iria automatizar muito do trabalho operacional. Para além disso, também se deveria iterar sobre a *pipeline* de Git Actions atual para transformar numa pipeline de DevOps que incluísse todos os passos desde a criação da imagem Docker, até à sua publicação e *deploy* no *cluster* de Kubernetes, incluindo ainda a execução dos testes automatizados e unitários.

Opcionalmente, também se poderia melhorar a forma como são configurados os ambientes na aplicação, i.e., ter ficheiros de configuração por ambiente em que fosse possível configurar diversas variáveis, como por exemplo *hostnames* das dependências.

Bibliografia

Al Jawarneh, I. M., Bellavista, P., Bosi, F., Foschini, L., Martuscelli, G., Montanari, R., and Palopoli, A. (2019). Container orchestration engines: A thorough functional and performance comparison. In *ICC 2019-2019 IEEE International Conference on Communications (ICC)*, pages 1–6. IEEE.

Bairagi, S. I. and Bang, A. O. (2015). Cloud computing: History, architecture, security issues. In *National Conference CONVERGENCE*, volume 2015, page 28.

Banga, G., Druschel, P., and Mogul, J. C. (1999). Resource containers: A new facility for resource management in server systems. In *OSDI*, volume 99, pages 45–58.

Bass, L., Weber, I., and Zhu, L. (2015). *DevOps: A software architect's perspective*. Addison-Wesley Professional.

Beck, K. (1999). Embracing change with extreme programming. *Computer*, 32(10):70–77.

Buildah (2018). Buildah blogs | buildah.io. https://buildah.io/blogs/.

Burns, B., Grant, B., Oppenheimer, D., Brewer, E., and Wilkes, J. (2016). Borg, omega, and kubernetes. *Communications of the ACM*, 59(5):50–57.

Campbell, G. A. and Papapetrou, P. P. (2013). *SonarQube in action*. Manning Publications Co.

Chappell, D. A. (2004). *Enterprise service bus*. " O'Reilly Media, Inc.".

Chiueh, S. N. T.-c. and Brook, S. (2005). A survey on virtualization technologies. *Rpe Report*, 142.

CloudZero (2021). Netflix architecture: How much does netflix's aws cost? https://www.cloudzero.com/blog/netflix-aws.

Containerd (2019). containerd an industry-standard container runtime with an emphasis on simplicity, robustness and portability. https://containerd.io/.

Containers, L. (2008). Linux containers. https://linuxcontainers.org/.

CoreOs (2017). What is rkt? https://www.redhat.com/en/topics/containers/what-is-rkt#history-of-rkt.

Docker (2013). Docker documentation. https://docs.docker.com/.

Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., and Safina, L. (2017). Microservices: yesterday, today, and tomorrow. *Present and ulterior software engineering*, pages 195–216.

Ebert, C., Gallardo, G., Hernantes, J., and Serrano, N. (2016). Devops. *Ieee Software*, 33(3):94–100.

Erl, T. (2005). *Service-oriented architecture*. Pearson Education Incorporated.

Felter, W., Ferreira, A., Rajamony, R., and Rubio, J. (2015). An updated performance comparison of virtual machines and linux containers. In *2015 IEEE international symposium on performance analysis of systems and software (ISPASS)*, pages 171–172. IEEE.

Fowler, M. and Foemmel, M. (2006). Continuous integration. *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf*, 122(14):1–7.

Grafana (2014). Grafana: The open observability platform | grafana labs. https://grafana.com/.

Huang, W., Liu, J., Abali, B., and Panda, D. K. (2006). A case for high performance computing with virtual machines. In *Proceedings of the 20th annual international conference on Supercomputing*, pages 125–134.

Kazanavičius, J. and Mažeika, D. (2019). Migrating legacy software to microservices architecture. In *2019 Open Conference of Electrical, Electronic and Information Sciences (eStream)*, pages 1–5. IEEE.

Khan, A. (2017). Key characteristics of a container orchestration platform to enable a modern application. *IEEE cloud Computing*, 4(5):42–48.

Knoche, H. and Hasselbring, W. (2018). Using microservices for legacy software modernization. *IEEE Software*, 35(3):44–49.

Krylovskiy, A., Jahn, M., and Patti, E. (2015). Designing a smart city internet of things platform with microservice architecture. In *2015 3rd International Conference on Future Internet of Things and Cloud*, pages 25–30. IEEE.

Leite, L., Rocha, C., Kon, F., Milojicic, D., and Meirelles, P. (2019). A survey of devops concepts and challenges. *ACM Computing Surveys (CSUR)*, 52(6):1–35.

Li, P. (2010). Selecting and using virtualization solutions: our experiences with vmware and virtualbox. *Journal of Computing Sciences in Colleges*, 25(3):11–17.

Li, W. and Kanso, A. (2015). Comparing containers versus virtual machines for achieving high availability. In *2015 IEEE International Conference on Cloud Engineering*, pages 353–358. IEEE.

Mell, P., Grance, T., et al. (2011). The nist definition of cloud computing. *National Institute of Standards and Technology*.

Merkel, D. et al. (2014). Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2.

Mo, Q., Song, W., Dai, F., Lin, L., and Li, T. (2019). Development of collaborative business processes: a correctness enforcement approach. *IEEE Transactions on Services Computing*.

Mumbaikar, S., Padiya, P., et al. (2013). Web services based on soap and rest principles. *International Journal of Scientific and Research Publications*, 3(5):1–4.

Neely, S. and Stolt, S. (2013). Continuous delivery? easy! just change everything (well, maybe it is not that easy). In *2013 Agile Conference*, pages 121–128. IEEE.

Newman, S. (20 Apr 2017). Chapter 5: Splitting the monolith. In *Building Microservices*.

nextflow.io (2020). Shifter containers. https://singularity.lbl.gov/. Online; accessed on 2020-12-14.

OpenVZ (2006). Open source container-based virtualization for linux. https://openvz.org/.

OVHcloud (1999). Aluguer de servidores para empresas | servidores dedicados | ovhcloud. https://www.ovhcloud.com/pt/bare-metal/advance/.

Pan, Y., Chen, I., Brasileiro, F., Jayaputera, G., and Sinnott, R. (2019). A performance comparison of cloud-based container orchestration tools. In *2019 IEEE International Conference on Big Knowledge (ICBK)*, pages 191–198. IEEE.

podman.io (2019). What is podman? simply put: alias docker=podman. https://podman.io/whatis.html. Online; accessed on 2020-12-14.

Qian, L., Luo, Z., Du, Y., and Guo, L. (2009). Cloud computing: An overview. In *IEEE international conference on cloud computing*, pages 626–631. Springer.

Randal, A. (2020). The ideal versus the real: Revisiting the history of virtual machines and containers. *ACM Computing Surveys (CSUR)*, 53(1):1–31.

Relic, N. (2008). New relic - observability platform for developers | new relic. https://newrelic.com/.

Riungu-Kalliosaari, L., Mäkinen, S., Lwakatare, L. E., Tiihonen, J., and Männistö, T. (2016). Devops adoption benefits and challenges in practice: a case study. In *International conference on product-focused software process improvement*, pages 590–597. Springer.

Ruzgar, N. S. (2005). A research on the purpose of internet usage and learning via internet. *Turkish Online Journal of Educational Technology-TOJET*, 4(4):27–32.

Savor, T., Douglas, M., Gentili, M., Williams, L., Beck, K., and Stumm, M. (2016). Continuous deployment at facebook and oanda. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 21–30. IEEE.

Sayfan, G. (2017). *Mastering kubernetes*. Packt Publishing Ltd.

Schwaber, K. (1997). Scrum development process. In *Business object design and implementation*, pages 117–134. Springer.

Sharma, P., Chaufournier, L., Shenoy, P., and Tay, Y. (2016). Containers and virtual machines at scale: A comparative study. In *Proceedings of the 17th international middleware conference*, pages 1–13.

Sommerville, I. (2011). *Software Engineering, 9/E*. Pearson Education, Inc.

Sprott, D. and Wilkes, L. (2004). Understanding service-oriented architecture. *The Architecture Journal*, 1(1):10–17.

Sun, Y., Safford, D., Zohar, M., Pendarakis, D., Gu, Z., and Jaeger, T. (2018). Security namespace: making linux security frameworks available to containers. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1423–1439.

Sylabs (2016). Home | sylabs. https://sylabs.io/.

Thönes, J. (2015). Microservices. *IEEE software*, 32(1):116–116.

Turnbull, J. (2014). *The Docker Book: Containerization is the new virtualization*. James Turnbull.

Valipour, M. H., AmirZafari, B., Maleki, K. N., and Daneshpour, N. (2009). A brief survey of software architecture concepts and service oriented architecture. In *2009 2nd IEEE International Conference on Computer Science and Information Technology*, pages 34–38. IEEE.

Xu, X., Cao, H., Geng, Q., Liu, X., Dai, F., and Wang, C. (2020). Dynamic resource provisioning for workflow scheduling under uncertainty in edge computing environment. *Concurrency and Computation: Practice and Experience*, page e5674.

Zhu, L., Bass, L., and Champlin-Scharff, G. (2016). Devops and its practices. *IEEE Software*, 33(3):32–34.