Orquestração e Escalabilidade: Uma Análise das Funcionalidades do *Kubernetes*

João Miguel Gomes Vieira

Dissertação para obtenção de Grau de Mestre em
Engenharia Informática
(2º ciclo de estudos)

Orientador:Profª. Doutora Paula Prata

Outubro de 2023

.

Dissertação elaborada no Instituto de Telecomunicações - Delegação da Covilhã e no Departamento de Informática da Universidade da Beira Interior por João Miguel Gomes Vieira, Licenciado em Engenharia Informática pela Universidade da Beira Interior, sob orientação da Doutora Maria Paula Prata de Sousa, Investigadora do Instituto de Telecomunicações e Professora Auxiliar do Departamento de Informática da Universidade da Beira Interior, e submetida à Universidade da Beira Interior para discussão em provas públicas.

Trabalho integrado em projeto parcialmente financiado por:

- FCT - Fundação para a Ciência e a Tecnologia através do projeto número UIDB/50008/2020 FCT/MCTES.

Declaração de Integridade

Eu, João Miguel Gomes Vieira, que abaixo assino, estudante com o número de inscrição m9959 de/o Engenharia Informática da Faculdade de Engenharia, declaro ter desenvolvido o presente trabalho e elaborado o presente texto em total consonância com o Código de Integridades da **Universidade da Beira Interior**.

Mais concretamente afirmo não ter incorrido em qualquer das variedades de Fraude Académica, e que aqui declaro conhecer, que em particular atendi à exigida referenciação de frases, extratos, imagens e outras formas de trabalho intelectual, e assumindo assim na íntegra as responsabilidades da autoria.

Universidade da Beira Interior, Covilhã 09/10/2023

Assinado por: **João Miguel Gomes Vieira**
Num. de Identificação: 14761211
Data: 2023.10.10 12:27:28+01'00'

Agradecimentos

Em primeiro lugar, agradeço à minha orientadora. Pela sua insistência durante todo o ano, foi também ela que acompanhou todo o processo e proporcionou uma ajuda essencial para o término deste percurso.

Agradeço aos meus amigos, pois foi com eles que partilhei bons e maus momentos. Os meus amigos de infância, os que o meu percurso académico trouxe e até aos que o meu pequeno percurso profissional me deu a conhecer, todos eles fazem com que me sinta acarinhado e apoiado nos momentos mais difíceis.

Agradeço aos meus pais, que sempre foram a minha fonte de inspiração e apoio incondicional. Eles são os verdadeiros heróis desta jornada e cada conquista que tive até agora deve-se muito a eles e aos ensinamentos que me passaram.

Agradeço ao meu irmão e à minha cunhada, que são os meus companheiros em muitas das jornadas da minha vida. São eles que tem sempre um conselho quando a situação está mais difícil e que me apoiam incondicionalmente.

Agradeço ao meu afilhado que apesar de ainda nem sequer ter um ano. É também por ele que luto para ser uma pessoa melhor e para que ele, quando se lembrar, sinta orgulho na pessoa que chama seu padrinho.

E por fim, à minha namorada que é a minha parceira em todas as aventuras, é ela que aguenta os piores e os melhores momentos, que me garante que tudo vai tomar o caminho certo. Esta conquista é também dela por todo o apoio que ela me deu.

A todos estes que mencionei e aos que pareceram ficar escondidos, agradeço.

Resumo

A gestão de implantações de projetos, em inglês *deployments*, é uma parte essencial em qualquer tipo de projeto, desde a gestão de recursos alocados ao projeto até à monitorização das máquinas virtuais responsáveis por alojar estes mesmos projetos.

Depois das empresas passarem por uma era onde o *deployment* era feito diretamente em máquinas físicas e onde era impossível de definir o limite de recursos alocados a projetos, causando problemas de performance, começamos a ver as primeiras implantações em máquinas virtuais. Com as máquinas virtuais, a divisão de recursos, dentro de uma mesma máquina, era mais fácil de se fazer e o problema que a abordagem anterior apresentava, tinha sido resolvida.

Com a virtualização, acabou por surgir o conceito de contentores, em inglês *containers*, que são muito similares a máquinas virtuais no entanto são muito mais leves, necessitam de muito menos recursos visto que não necessitam dos recursos que uma emulação de um sistema operativo necessita para funcionar. Num contentor também fica guardado tudo o que seja necessário para executar qualquer tipo de *software* incluindo código, bibliotecas, configurações etc... Com este conceito de *containers*, surgiram tecnologias como o *docker* que tornaram estes *containers* o *standard* em empresas de informática com os mais variados projetos, pela sua facilidade em manter a consistência das aplicações e de reproduzir os mais variados ambientes dos sistemas. Com o *docker* é mais fácil de desenvolver, testar e fazer o *deploy* de aplicações assim como a sua gestão e escalamento em ambientes produtivos.

Com o aparecimento e o ganho de popularidade dos *containers*, o orquestrador de *containers* Kubernetes (K8s) aparece como uma solução para gerir ambientes complexos com vários *containers*. Com o K8s é possível automatizar, escalar e gerir aplicações dentro de *containers* [Goo23d]. Neste trabalho, é feito um estudo das principais características dos sistemas de orquestração de *containers*, com particular detalhe para o K8s. Fez-se uma instalação local de K8s para criar um ambiente de testes e analisar o impacto do *autoscaling* no desempenho de um conjunto de aplicações exemplo e também nos recursos do sistema.

Os resultados obtidos mostram que o *autoscaling* é uma ferramenta que deve ser parametrizada tendo em conta o tipo de aplicação e que deve ser ajustada ao longo do tempo.

Palavras-chave

Deployment de projetos, *Docker*, Escalonamento automático, *Kubernetes*, Orquestração de *containers*, Virtualização

Abstract

Project deployment management is an essential part of any type of project, from the management of resources allocated to the project to the monitoring of virtual machines responsible for hosting these same projects.

After companies went through an era where deployment was done directly on machines and where it was impossible to define the limit of resources allocated to projects, causing performance issues, we began to see the first deployments in virtual machines. With virtual machines, the division of resources within the same machine was easier and the problem that the previous approach presented, had been solved.

With virtualization, the concept of containers emerged, which are very similar to virtual machines however they are much lighter, require much less resources since it do not need the resources that an emulation of an operating system needs to function. Containers also store everything that is necessary to run any kind of software including code, libraries, configurations etc... With this concept of containers, technologies like Docker made these containers the standard in IT companies with the most varied projects, due to its ease in maintain application consistency and reproduce the most varied system environments. With docker it is easier to develop, test and deploy applications as well as its management and escalation in productive environments.

With the appearance and popularity of containers, container orchestrators K8s appears as a solution to manage complex environments with multiple containers. With K8s it is possible to automate, scale and administer applications inside containers [Goo23d]. In this work, a study is conducted on the main characteristics of container orchestration systems, with particular detail on K8s. A local installation of K8s was performed to create a testing environment and analyze the impact of autoscaling on the performance of a set of example applications as well as on system resources. The results obtained demonstrate that autoscaling is a tool that should be parameterized considering the type of application and should be adjusted over time

Keywords

Autoscaling, Container orchestraction, Docker, Kubernetes, Project Deployment, Virtualization,

Conteúdo

1 Introdução **1**
1.1 Motivação e Objetivos . . . 1
1.2 Contribuições . . . 2
1.3 Organização da Dissertação . . . 2

2 Estado da Arte **3**
2.1 Virtualização . . . 3
2.1.1 Máquinas Virtuais . . . 3
2.1.2 *Hyper-v* . . . 4
2.2 *Containers* . . . 4
2.2.1 *Container* vs Máquina Virtual . . . 5
2.2.2 Docker . . . 6
2.3 Orquestração de *Containers* . . . 7
2.3.1 Kubernetes . . . 8
2.3.2 Outros orquestradores . . . 11
2.4 Conclusão . . . 12

3 Plano de trabalho **13**
3.1 Plano de trabalho . . . 13
3.2 Conclusão . . . 13

4 Contexto experimental **15**
4.1 Introdução . . . 15
4.2 Máquina Virtual . . . 15
4.3 Microk8s . . . 17
4.4 *Deployment* da aplicação exemplo no *Microk8s* . . . 18
4.4.1 *Dockerfile* e *Docker image* . . . 19
4.4.2 *Deployments* . . . 22
4.4.3 *Services* . . . 24
4.5 *Autoscaling* . . . 25
4.6 A ferramenta de teste e monitorização: *JMeter* . . . 26
4.6.1 *JMeter* . . . 27

5 Testes e Resultados **31**
5.1 Testes . . . 31
5.2 Resultados . . . 31

6 Conclusão e Trabalho Futuro **37**
6.1 Conclusão . . . 37
6.2 Trabalhos Futuros . . . 37

Bibliografia **39**

Lista de Figuras

2.1 Arquitetura do *container* segundo [Fre20] 6
2.2 Arquitetura da máquina virtual segundo [Fre20] 6
2.3 Ilustração da arquitetura do *docker* segundo [Goo23b] 7
2.4 Ilustração da arquitetura de um orquestrador de *containers* segundo [IMAJ19] . . 8
2.5 Exemplo Arquitetura do K8s segundo [Pan22] 10

4.1 Definições iniciais da máquina virtual 16
4.2 Definição do utilizador 16
4.3 Recursos disponibilizados para a máquina virtual. 16
4.4 Espaço em disco disponibilizado para a máquina virtual 17
4.5 Sumário das definições 17
4.6 *Add-ons* disponíveis 18
4.7 Nós disponíveis no ambiente K8s 18
4.8 Imagens disponíveis no repositório local 20
4.9 Página de criação do repositório 21
4.10 Nova imagem *docker* disponibilizada 22
4.11 *Deployments* no K8s 24
4.12 Serviços do K8s 25
4.13 *Horizontal Pod Autoscaler* 26
4.14 *Interface* do *JMeter* 27
4.15 Criação do *Thread Group* 27
4.16 Configuração do *Thread Group* 28
4.17 Criação da *HTTP Request* 28
4.18 Configuração da *HTTP Request* 29
4.19 *Listeners* disponíveis 29
4.20 *Summary Report* 29

Lista de Tabelas

3.1 Planeamento do trabalho. 13

5.1 Testes de carga para a aplicação *v1* 32
5.2 Valores de *CPU* e Memória para a aplicação *v1* 33
5.3 Testes de carga para a aplicação *v2* 33
5.4 Valores de *CPU* e Memória para a aplicação *v2* 34
5.5 Ganhos do tempo de execução obtidos com a replicação 34

Lista de Siglas e Acrónimos

K8s	Kubernetes
VMM	Virtual Machine Monitor
CNI	Container Network Interface
SDN	Software-defined networking
AKS	Azure Kubernetes Service
GKE	Google Kubernetes Engine
API	Application Programming Interface
RAM	Random Access Memory
YAML	Yet Another Markup Language
SGBD	Sistema de gestão de Base de Dados
KPI	Key Performance Indicator
SO	Sistema operativo

Capítulo 1

Introdução

Ano após ano, o mundo empresarial tem vindo a sofrer alterações importantíssimas em tudo o que possa ajudar as mais variadas equipas de desenvolvimento, com foco em melhorar processos e otimizar os custos em projetos cada vez maiores e onde os clientes são cada vez mais exigentes no que toca a tempos de resposta.

O número de empresas com equipas de *Dev-Ops* dedicadas (que são responsáveis por um conjunto de práticas que visam a redução do tempo entre as modificações em ambientes de desenvolvimento para os ambientes produtivos enquanto asseguram a qualidade destes mesmos ambientes [BWZ15]) é cada vez maior e o desenvolvedor comum cada vez tem de se preocupar menos com questões relacionadas com o *deployment* da aplicação ou as *pipelines* dedicadas para gerir todo o código desde o *commit* mais simples até ao *merge request* para produção mais complexo.

No contexto de *Dev-Ops*, o Kubernetes (K8s) (chamado desta forma devido ao número de letras compreendido entre a primeira e última letra), pela sua facilidade na automação de *deployments* bem como a capacidade de escalar as aplicações à necessidade do cliente e também pelo facto de ser *open-source*, é a ferramenta mais utilizada em todo o mundo para orquestração de *containers* [Fle22].

1.1 Motivação e Objetivos

A necessidade que as empresas têm em encontrar profissionais que compreendam as várias áreas de competência requeridas para o bom desenvolvimento dos seus projetos, leva a que os profissionais sejam obrigados a compreender tecnologias fora das suas áreas de competência com vista a tornar o seu trabalho mais ágil e o trabalho das equipas mais fluido.

A motivação deste trabalho é aprofundar e adquirir conhecimentos que possam ser uma mais valia no futuro, numa área que está em ascensão. É também para ganhar mecanismos que possam trazer, a nível profissional, uma mais valia, com vista, a ter conhecimento nas mais variadas áreas de competência, neste caso, na área de *Dev-Ops*.

O objetivo principal é a criação de um ambiente de virtualização com base no K8s e desta forma explorar como fazer o *deploy* de aplicações. Irá ser feito um estudo sobre *autoscaling* onde, para isso, se irão criar algumas aplicações exemplo. Será feita toda a configuração do *deployment* das aplicações e, posteriormente, a configuração de um *Horizontal Pod Autoscaler* de forma a testar as funcionalidades de escalonamento e o porquê de ser importante.

Por fim, serão feitos testes de carga e serão recolhidos dados para perceber as diferenças entre uma aplicação com uma única réplica e a mesma aplicação mas com múltiplas réplicas.

1.2 Contribuições

Neste trabalho foi feito:

- O estado da arte sobre máquinas virtuais, *containers* e orquestradores de *containers*, com especial destaque para o *Docker* no caso dos *containers* e um ênfase adicional no K8s para os orquestradores de *containers*.
- Uma explicação detalhada sobre o *deployment* de aplicações num ambiente K8s, abrangendo todo o processo desde a criação da imagem *Docker* até à configuração do *deployment* na plataforma K8s.
- Análise e criação de ficheiros de configuração para os *HorizontalPodAutoscaler*.
- Avaliação do desempenho de aplicações que estão sujeitas a testes de carga, com e sem escalonamento.
- Avaliação da utilização de recursos, com e sem escalonamento.

1.3 Organização da Dissertação

O presente trabalho está organizado da seguinte forma:

- Capítulo 1 - Introdução - uma breve introdução ao tema desta dissertação, as motivações e objetivos, as contribuições e a organização do documento.
- Capítulo 2 - Estado da Arte - No estado de arte iremos abordar os conceitos de virtualização, *Containers* e também os orquestradores de *Containers* .
- Capítulo 3 - Plano de trabalho - No plano de trabalho irá ser feita a proposta de trabalho a ser realizado e a sua calendarização.
- Capítulo 4 - Contexto Experimental - Aqui descreve-se a criação da plataforma de K8s para testes e análise do impacto do escalonamento de aplicações.
- Capítulo 5 - Testes e Resultados - Descrição dos testes realizados e os resultados obtidos.
- Capítulo 6 - Conclusão e trabalho futuro - Conclusão e direções para trabalho futuro.

Capítulo 2

Estado da Arte

Neste capítulo, apresentamos as mais recentes tecnologias de virtualização, criação de *containers* e da sua orquestração. De forma a entender o K8s melhor, iremos abordar temas que estão diretamente ou indiretamente relacionados ao *Kubernetes*. Inicialmente irá ser estudado o que é a virtualização, que é a tecnologia central para tudo o que vem a seguir. Nesta secção sobre a virtualização iremos abordar o que são máquinas virtuais e iremos estudar brevemente o *Hyper-V*, que é uma tecnologia de virtualização da *Microsoft*, com o intuito de estender o estudo da virtualização e dar um exemplo concreto de uma tecnologia de virtualização.

De seguida, iremos abordar o conceito de *container*, mais especificamente, o *docker* por ser uma das ferramentas mais utilizadas em 2022 [Fle22] no que diz respeito à criação de *containers*. Irá ser feita uma introdução ao tema, explicando conceitos envolvidos na arquitetura do *docker*.

Iremos apresentar o modelo de orquestração de *containers* dando um maior ênfase no *K8s* onde iremos abordar em maior detalhe a sua arquitetura e também brevemente detalhar outros orquestradores.

Por fim, fez-se o levantamento das várias técnicas de escalonamento e também os critérios a ter em conta aquando da escolha do método mais apropriado e de métricas de avaliação.

2.1 Virtualização

Pela forma como as tecnologias foram evoluindo, a Virtualização irá ser o primeiro tema em análise deste capítulo. Iremos começar por definir máquina virtual, falar de algumas das suas utilidades e formas de uso e por fim dar o exemplo de uma tecnologia de virtualização.

2.1.1 Máquinas Virtuais

Uma máquina virtual é, como o nome indica, um ambiente virtual que funciona exatamente como um computador que possui um CPU, memoria etc..., através de máquinas virtuais é possível partilhar recursos de uma máquina mãe, com vários ambientes virtuais.

De acordo com o [Cou22], com recurso a máquinas virtuais, podemos criar ambientes de desenvolvimento para testar aplicações que necessitem de estar isoladas dando assim forma aos desenvolvedores de testar código ou software sem impactar o resto do ambiente. É possível também fazer *disaster management*, por exemplo, os utilizadores de *iPhones* fazem *backups* dos seus dados ao conectar com a *iCloud* e esta, por sua vez, guarda uma cópia virtual dos seus dados para que estes possam ser recuperados caso os dados sejam perdidos. Uma outra utilidade das máquinas virtuais é a capacidade de testar software com hardware incompatível. Podemos dar o exemplo de um software antigo que não sofre alterações há vários anos e necessitamos de

testar se ainda é compatível com sistemas mais recentes. Com recurso a uma máquina virtual podemos configurar todo o sistema para fazer este teste, desde o tipo de memória que ele usa até ao sistema operativo que pretendemos usar.

Segundo a página da VMWare [VMw23c], apesar de existiram inúmeras vantagens sobre as máquinas físicas, as máquinas virtuais também têm as suas desvantagens, por exemplo, correr múltiplas máquinas virtuais numa única máquina física pode resultar em instabilidades na performance caso os requisitos da infraestrutura não sejam correspondidos. As máquinas virtuais também são menos eficientes e mais lentas que um computador físico.

Associado ao termo máquina virtual, podemos muitas vezes encontrar o termo *hypervisor*. Este *hypervisor*, que também é conhecido por Virtual Machine Monitor (VMM) é o *software* que cria e corre as máquinas virtuais. O *hypervisor* permite que um computador hospede múltiplas máquinas virtuais ao partilhar os recursos da máquina mãe.

Existem dois tipos principais de *hypervisors*, os de tipo 1 que agem como um sistema operativo que corre diretamente no *hardware* da máquina em que corre, e o de tipo 2 que corre como um *software* como um programa habitual do computador [VMw23b]. Um exemplo de um *hypervisor* tipo 1 é o *Hyper-v* que será estudado na próxima secção.

2.1.2 *Hyper-v*

O *Hyper-v* é uma tecnologia de virtualização desenvolvido pela *Microsoft* que permite correr vários sistemas operativos numa única máquina física, foi introduzida pela primeira vez como um componente do *Windows Server 2008* e desde então tem sido desenvolvido como um produto *standalone* que pode ser utilizado com vários sistemas operativos.

Segundo Jadran Torbić, Ivan Stanković, Borislav Đorđević e Valentina Timenko o *Hyper-v* [TSDT18] representa uma camada de *software* que fica localizada imediatamente a cima do hardware. O seu propósito é permitir a diferentes sistemas operativos que simultaneamente corram no mesmo equipamento. Tal como o *kernel*, o *Hyper-V* gere a memória disponível, processa e influencia o funcionamento de todo o sistema.

Para a [Acr20], uma das maiores razões para a utilização do *Hyper-v* é que é uma solução com uma boa relação qualidade-preço. Algumas das ferramentas mais simples do *Hyper-v* não necessitam de licenciamento pago, no entanto, para funcionalidades mais avançadas é necessário uma licença paga.

2.2 *Containers*

Segundo Bora Basyildiz [Bas19] a evolução dos *containers* deu um grande passo em frente durante o desenvolvimento do *chroot* em 1979 como parte da versão 7 da *Unix*. Este *chroot*, marcou o inicio do isolamento de processos ao restringir o acesso de um ficheiro a uma diretoria em especifico. Apesar de grandes melhorias entre 2000 e 2011, a introdução do *Docker* foi o que fez com que os *containers* explodissem em popularidade e a sua popularidade continuou a

subir em 2017 através da introdução do K8s.

Segundo Stefano Sebastio, R. Ghosh e Tridib Mukherjee [SGM21], um *container* é um processo virtualizado e leve que oferece um ambiente independente da camada de *hardware*, ao contrário de uma máquina virtual, o *container* não tem integrado um sistema operativo. Com os *containers* é possível partilhar os recursos de uma só máquina com várias aplicações sem que, ao contrário do que faz uma máquina virtual, seja necessário instalar um sistema operativo para correr estas aplicações.

Para a [Aqu23] algumas vantagens de uma arquitetura conteinerizada são:

- Baixos custos de infraestrutura devido à possibilidade de correr vários *containers* numa única máquina virtual.
- A escalabilidade ao nível de microserviços elimina a necessidade de escalar o número de máquinas virtuais.
- Ser independente de sistema operativo.
- A sua rapidez entre iniciar e parar um *container* em segundos.
- Tendo em conta que não contém um sistema operativo, passa a ser uma solução leve.

2.2.1 *Container* vs Máquina Virtual

Para a [VMw23c] os *containers* são semelhantes às máquinas virtuais no sentido em que executam aplicações de forma isolada numa única plataforma. Enquanto que as máquinas virtuais virtualizam o *hardware* para criar um computador, os *containers* empacotam apenas o que uma aplicação necessita para funcionar.

Ainda segundo a [VMw23c], um beneficio chave dos *containers* consiste no facto de este ser muito mais leve quando comparado com máquinas virtuais, devido a apenas necessitar da aplicação de que se quer fazer o *deploy* e as suas dependências. Como resultado disto, um *container* inicia muito mais rapidamente fazendo com que a entrega de aplicações seja muito mais eficiente.

Em contraste, as máquinas virtuais são muito mais pesadas (em recursos) e mais lentas ao iniciar que um *container*, elas estão isoladas umas das outras e podem ter o beneficio de ter um sistema operativo completamente separado. As máquinas virtuais são mais indicadas para correr aplicações maiores e mais pesadas como por exemplo, monolíticos, em vez de microserviços que requerem uma maior separação e iria obrigar à criação de um número maior de máquinas virtuais fazendo com que sejam necessários mais recursos.

Segundo [Fre20] e tendo em conta as figuras 2.1 e 2.2, é possível observar que na arquitetura dos *containers*, estes não utilizam um sistema operativo próprio e utilizam também o *kernel* do sistema operativo onde o *container* está hospedado, já as máquinas virtuais, possuem um sistema operativo e um *kernel* próprio.

Uma questão também explorada em [Fre20] é se máquinas virtuais e *containers* podem coexistir no mesmo ambiente, sendo que a resposta é sim e podemos verificar isso num caso de uso onde

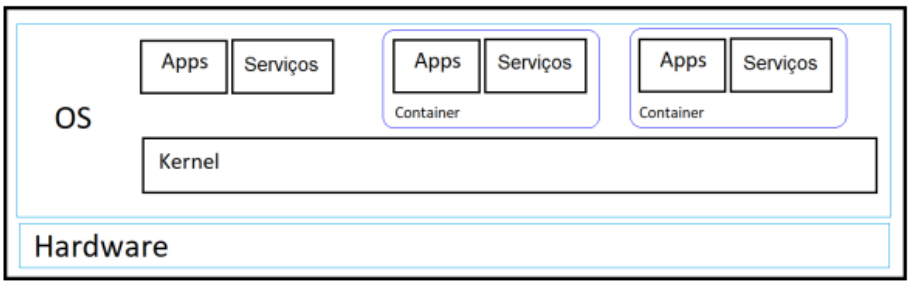

Figura 2.1: Arquitetura do *container* segundo [Fre20]

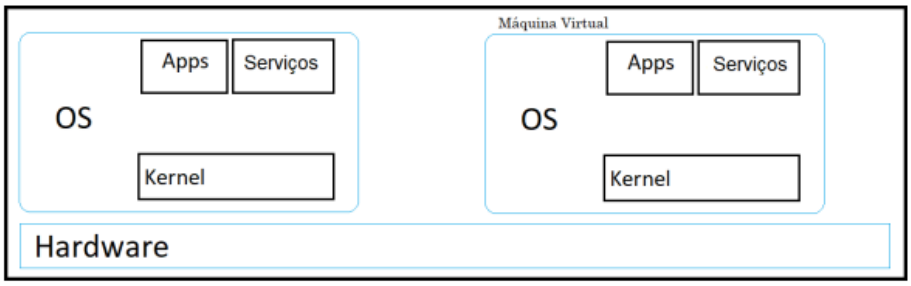

Figura 2.2: Arquitetura da máquina virtual segundo [Fre20]

se faça a instalação de, por exemplo, o *docker* numa máquina virtual. Neste caso poderíamos ter uma máquina virtual que iria simular todo um sistema operativo e em nesse mesmo sistema operativo poderíamos instalar um sistema de *containers*.

2.2.2 Docker

Segundo os autores Amit Potdar, Narayan D. G., Shivaraj Kengond e Mohammed Moin Mulla [PGKM20] a conteinerização é uma tecnologia que combina a aplicação, dependências relacionadas e bibliotecas do sistema para agregar tudo o que é necessário para executar a aplicação sob a forma de um recipiente. As aplicações que são construídas e organizadas podem ser executadas e implantadas como um *container*. Uma plataforma que implementa este conceito é conhecida como *Docker*, e garante que a aplicação funciona em todos os ambientes.

Segundo o autor Joshua Cook [Coo17], podemos olhar para o motor do *Docker* como o *daemon*, i.e. o que corre o processo em segundo plano. O *Docker*, como um todo, consiste num motor e num *hub*, sendo que o *hub* é onde ficam armazenadas as imagens *Docker* (que serão explicadas mais abaixo), sendo que estas são um conjunto de instruções para construir um *container*.

Para melhor entendermos o *docker*, a sua arquitetura é ilustrada na figura 2.3.

Tendo em conta a figura 2.3, o *docker client* comunica com o *docker daemon* para fazer todo o trabalho importante de construir, correr e gerir os *containers*. Outros aspetos importantes da arquitetura, segundo a [Goo23b] são:

- *Docker Client* - o *docker client* (comando *docker*) é a principal forma de comunicar com o *docker*. Quando se utiliza, por exemplo, o comando *docker run*, o *client* envia este comando ao *dockerd* que executa as instruções necessárias.

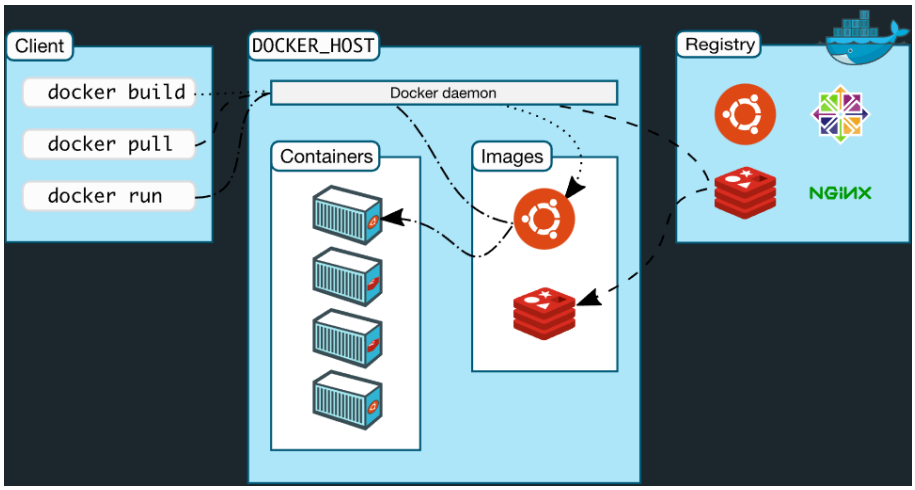

Figura 2.3: Ilustração da arquitetura do *docker* segundo [Goo23b]

- *Docker Daemon* - o *daemon* (comando *dockerd*) está á escuta das *APIs* do docker para gerir objetos como as imagens, *containers*, volumes e para fazer gestão de *networking*.
- *Docker Registries* - o *registry* é onde são guardadas as imagens *docker*. Um exemplo conhecido de um *registry* é o próprio *Docker Hub*, este é público e qualquer utilizador pode usar. Por omissão, o *docker* está configurado para utilizar o *Docker Hub* como *registry* para as suas imagens, no entanto, é possível ter um *registry* privado para as imagens.
- *Docker objects* - quando se utiliza o *docker*, está-se a criar e a usar imagens, *containers*, *networks*, volumes e outros tipos do objetos. Os objetos mais importantes a ter em conta são:
  - *Images* - uma imagem é um *template* de instruções para a criação de um *container docker*. É possível criar uma imagem à medida de uma certa necessidade, ou então, utilizar imagens de *registries* de terceiros. Para construir uma imagem é necessário criar um *Dockerfile* com as instruções necessárias para criar uma imagem.
  - *Containers* - A definição de *container* é a mesma que já foi abordada neste relatório na secção 2.2. É possível criar, iniciar, parar, mover ou apagar *containers* utilizando as *APIs* do *docker*.

2.3 Orquestração de *Containers*

A orquestração de *containers* é o processo de automatização de grande parte das operações necessárias para correr aplicações dentro de *containers*. Isto inclui a automatização de processos como os de *deployment* e provisionamento, *scaling* e de questões relacionadas com o *networking* das aplicações entre outros aspetos que geralmente as equipas de *Dev-ops* têm de resolver, no que diz respeito a aplicações e ambientes produtivos e não produtivos. Através desta orquestração é possível diminuir de forma significativa a complexidade de todos estes processos [VMW23a].
Pela imagem 2.4 e segundo o Isam Mashhour Al Jawarneh [IMAJ19], a camada de *Resource Management* gere os recursos, como a memoria *CPU*/*GPU*, espaço no disco ente outros. O seu objetivo é maximizar a utilização e minimizar a interferência entre *containers* que estejam a

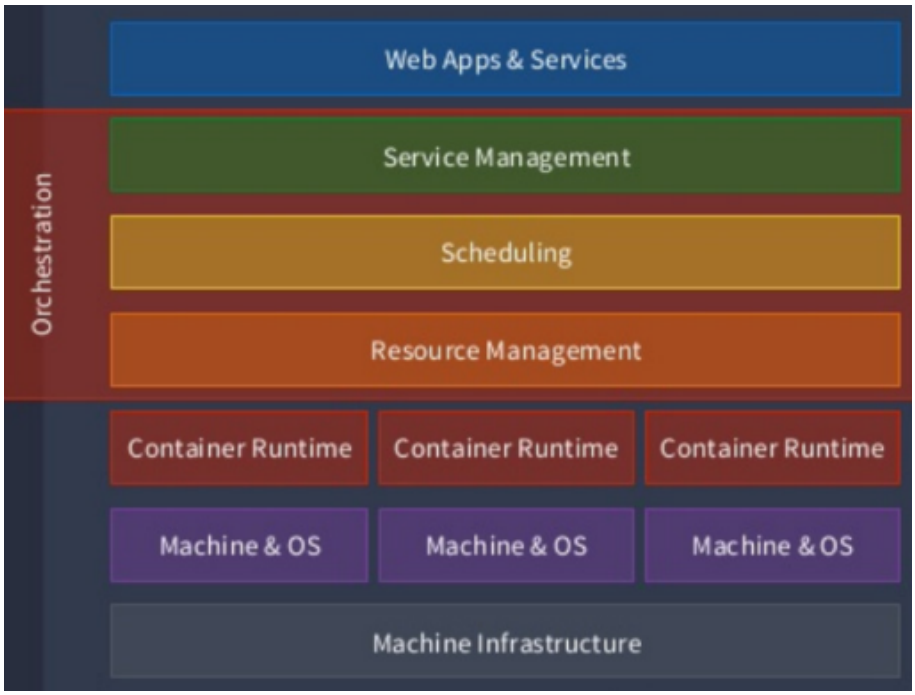

Figura 2.4: Ilustração da arquitetura de um orquestrador de *containers* segundo [IMAJ19]

competir por recursos.

O *scheduling layer* procura utilizar os recursos do *cluster* eficientemente. Tipicamente, este recebe informações por parte do utilizador, como os recursos de um *container* ou o número de réplicas, e então decide como distribuir os recursos pelos vários *containers* que constituam uma aplicação.

Por fim, a camada do *service management* oferece a capacidade de construir e entregar aplicações complexas. Este *service management* gere aspetos que incluem:

- *labels* que serão juntos aos metadados do *container*;
- grupos/*namespaces* para isolar os *containers*;
- *load-balancing* para dividir carga que seja imposta ao *container*;
- verificar se a aplicação está pronta para receber pedidos.

2.3.1 Kubernetes

Segundo os autores Jay Shah e Dushyant Dubaria [SD19], a orquestração de *containers* é o conceito central por trás de K8s. Então, o K8s está encarregue de todos os *containers*. Incluído nos seus deveres de gestão de *containers* estão a colocação de *containers*, o *scaling* e *descaling*, e o *load-balancing* do *container*. O K8s é usado para automatizar a implementação, *scaling* [DTR+18], [ZKU+22] e orquestração de aplicações em *containers*, bem como a gestão de cargas e serviços inseridos em *containers*.

O *scaling* das aplicações pode ser feito de várias formas [TVK22]. A escolha do método de *scaling* inclui processos de análise de dados como a carga à qual a aplicação está a ser submetida, o

tipo de arquitetura que a aplicação tem, o *timing* em que queremos fazer o escalonamento e também se deve ter em conta que tipo de métricas estamos a utilizar para monitorizar a aplicação. Estas análises podem levar à escolha de um de três tipos de escalonamento:

- *Vertical Scaling* - O escalonamento vertical refere-se ao aumento ou diminuição de recursos alocados à aplicação. Este método depreende que se desligue a aplicação para reconfigurar os recursos alocados.

- *Horizontal Scaling* - O escalonamento horizontal permite o aumento ou diminuição de réplicas de cada aplicação. Este método pode ser configurado de forma manual ao definir valores máximos para o consumo de, por exemplo, *CPU* por parte da aplicação.

- *Hybrid Scaling* - O escalonamento híbrido é uma combinação de ambos os métodos de escalonamento a cima e permite a utilização de ambos os métodos para criar condições ideais da utilização de ambos, ou seja, podemos fazer um escalonamento vertical para determinar quais os recursos mais apropriados a alocar à aplicação numa primeira fase e depois utilizar o escalonamento horizontal para mudar, de forma dinâmica, o número de réplicas.

Ainda segundo [TVK22], saber quando fazer o escalonamento é algo a ter em conta e pode ser feito de duas formas:

- *Reactive Scaling* - No escalonamento reativo o sistema monitoriza o tráfego existente ou os recursos utilizados e se algum dos dois atingir um valor máximo definido, então o sistema irá fazer uma avaliação de qual escalonamento fazer.

- *Proactive Scaling* - O escalonamento proativo utiliza técnicas de prever necessidades futuras para organizar os recursos necessários. Inteligência artificial e *machine-learning* tem sido utilizados para fazer este tipo de previsão.

[TVK22] indica também os tipos de métricas que podem ser avaliadas no processo de escolha de método de escalonamento. Entre eles, pode-se encontrar os próprios recursos da máquina, como o *CPU*, *RAM* e memória disponível, os tempos de resposta da aplicação e o número de chamadas feitas à aplicação. Estas métricas podem ser classificadas como de baixo nível, como por exemplo, os recursos físicos, ou então de alto nível, como por exemplo o tráfego de pedidos feitos à aplicação e o tempo que estes demoram a ser processados.

Na figura 2.5, podemos ver alguns dos componentes da arquitetura do K8s explicados mais abaixo.

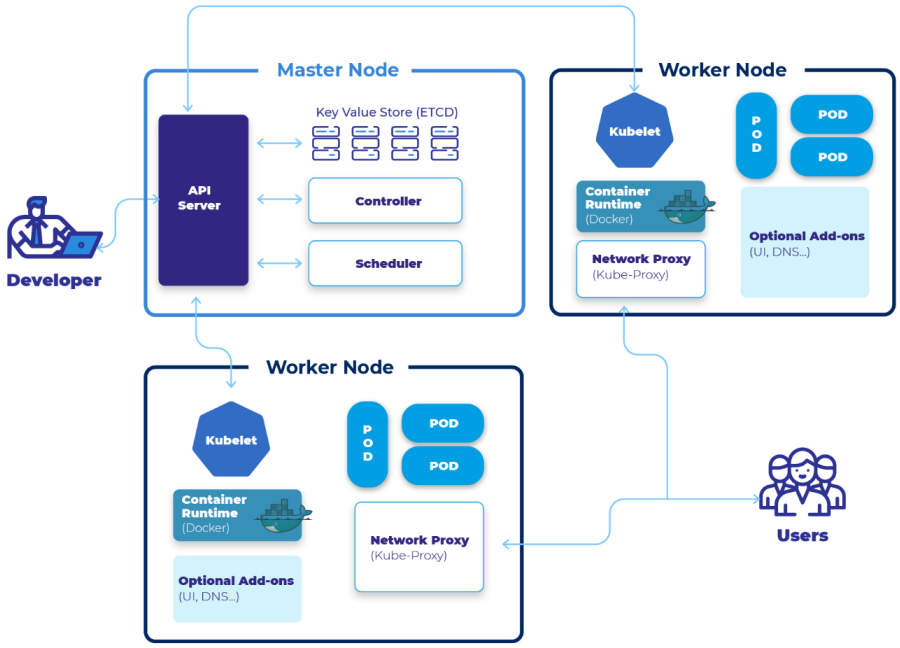

Figura 2.5: Exemplo Arquitetura do K8s segundo [Pan22]

Na figura 2.5 podemos ver os seguintes módulos do K8s:

- *Master Node* - O *master node* é responsável pela gestão do *cluster* K8s. É sobretudo o ponto de entrada para todas as tarefas administrativas. O *master node* está responsável pelos processos de controlo que estão disponíveis para todo o ambiente.
- *Worker nodes* - Os *worker nodes* correm aplicações via *pods* enquanto que o *master node* controla estes *pods*. Para aceder às aplicações é necessário conectar ao *worker node*.
- *Pod* - Os *pods* são a unidade mais pequena de execução do K8s, aqui serão organizados os *containers* que vão constituir a aplicação.
- *API Server* - O *API server* é o ponto central para expor funcionalidades para utilizadores externos. Quando estamos a utilizar os comandos do *kubectl* estamos na realidade a comunicar com o *API server*.
- *Etcd* - O *etcd* é a base de dados utilizada para guardar valores no formato de *key-value* e onde é guardada toda a informação sobre o *cluster*.
- *Controller-Manager* - O *controller* é um componente do painel de controlo do K8s, este corre sob a forma de um *container* dentro de um pod em cada *master node* do *cluster*. Este é uma coleção de diferentes controladores cuja principal função é verificar o estado dos objetos e fazer com que o estado dos objetos esteja no estado desejado.
- *Scheduler* - Este componente também corre no painel de controlo do K8s e tem a responsabilidade de atribuir *pods* para o *cluster*. Este determina que nós estão aptos para receber um *pod* dependendo de, por exemplo, os recursos disponíveis num certo nó.
- *Kubelet* - Cada *worker node* tem um processo chamado de *kubelet*, este cria, destroi ou atualiza *pods* do seu nó.

- *Kube-proxy* - Outro serviço que corre em todos os nós, este é responsável por todo o *networking* do nó.
- *Container Runtime* - Esta é a peça de software responsável por correr os *containers* nos *pods*.
- *Add-ons* - De forma a extender funcionalidades do K8s é possível também instalar *add-ons*.

Múltiplos estudos sobre sistemas K8s têm sido publicados na literatura cientifica:

- Em [ZG22] é estudado como automatizar a segurança de um sistema K8s.
- [MHR21] estuda questões relacionadas com a migração de dados para *Deep Learning*.
- Em [KAGP21] é analisada a performance de *plugins* Container Network Interface (CNI).
- No trabalho [TKVL+20] são estudadas diferenças entre arquiteturas para vários clientes de K8s.
- Em [GSC+22] é feito um estudo para compreender a interação entre o *scheduler* do K8s com o Software-defined networking (SDN) aquando do *deployment* de *pods*.
- Em [BR22] é estudado o *deployment* em K8s utilizando *Helm Charts*.

2.3.2 Outros orquestradores

Para além do K8s existem outros orquestradores de *containers*:

- *Docker swarm* - *Docker swarm* é uma outra tecnologia de orquestração de *containers*, sendo que este, é o sistema nativo do próprio *Docker*. Com este é também possível fazer a gestão de todo um *cluster* de *containers* oferecendo ferramentas para escalar os *pods* existentes, *load balancing*, ferramentas de gestão de *networking* etc...[Doc23b]
- *Rancher* - Apesar de estar neste capítulo, o *Rancher* não é em si um orquestrador. O *Rancher* é uma ferramenta que, quando utilizada com o K8s, pode facilitar a interação com os comandos do próprio K8s. O *Rancher* oferece ferramentas de autenticação e controlo de acessos para todos os *clusters* que possam existir num ambiente de K8s e oferece ferramentas de integração com repositórios para ficar diretamente ligado a *pipelines* de *deployment* [Ran22].

Segundo [Fre20], numa perspetiva de comparação do *docker swarm* e o K8s, verificou-se que o *docker swarm* é uma ferramenta de orquestração superior se tivermos ambientes mais simples, com um número de *containers* reduzido e também é mais fácil de instalar e configurar, no entanto, para ambientes mais complexos, com necessidade de haver um conjunto de *workers* para implementar um ambiente de *containers* mais complexo, o K8s é o mais indicado.

Existem também tecnologias *cloud* que oferecem ferramentas de criação de *clusters* K8s na *cloud*.

Entre eles:

- Azure Kubernetes Service (AKS) - Serviço de K8s para a *cloud* da *Microsoft* [Mic23]
- Google Kubernetes Engine (GKE) - Serviço de K8s para a *cloud* da *Google* [Goo23c]

Ambos os serviços oferecem a possibilidade de criação de um *cluster* de K8s.

2.4 Conclusão

Neste capítulo foi feito um estudo sobre máquinas virtuais, *containers*, *docker*, orquestradores de *containers* e por fim o K8s.

Máquinas virtuais, *containers* utilizando o *docker* e orquestradores de *containers* utilizando o K8s são tecnologias inter-relacionadas que transformam a forma como aplicações modernas são construídas e implantadas.

As máquinas virtuais oferecem um ambiente virtual que pode correr múltiplos sistemas operativos, sendo flexíveis com a utilização de recursos de uma máquina mãe. Por outro lado, os *containers* são uma solução leve de ambientes isolados para correr aplicações, tornando fácil a sua manutenção e *deployment*. O *docker* é uma plataforma para construir, implantar e correr *containers*. Por fim, o orquestrador de *containers*, K8s, oferece formas de automatizar o *deployment*, escalonamento e gestão das suas aplicações.

Estas tecnologias juntas, oferecem soluções para a construção, *deployment* e gestão de aplicações modernas, dando às empresas formas eficientes de oferecer soluções de grande qualidade aos seus clientes.

Neste trabalho, vamos estudar como o escalonamento de aplicações numa plataforma de K8s tem impacto no desempenho do sistema.

Capítulo 3

Plano de trabalho

3.1 Plano de trabalho

Após a criação do ambiente de K8s pretende-se fazer vários testes a uma, ou várias, aplicações exemplo de forma a perceber o seu comportamento com uma e múltiplas réplicas utilizando uma ferramenta que permita a realização de testes de carga à aplicação. Com estes testes, o objetivo é também verificar os tempos de resposta e de processamento da aplicação, bem como, como aumenta a utilização de recursos perante o aumento de processamento de dados. Esta dissertação inclui as seguintes tarefas:

- Criar um conjunto de aplicações.
- Estudar ferramentas para testes de carga, nomeadamente o *JMeter*.
- Estudar ferramentas de monitorização para verificação do desempenho dos recursos sob carga.
- Fazer uma instalação de K8s local.
- Fazer o *deploy* das aplicações criadas na instalação local do K8s.
- Medir o impacto do *scaling* no desempenho das aplicações.
- Medir o impacto do *scaling* nos recursos sistema.
- Comparar esse impacto para diferentes tipos de aplicações.

Na tabela 3.1 está o planeamento do trabalho a ser realizado para os próximos meses.

Tarefa	Mês 1	Mês 2	Mês 3	Mês 4
Escrita da dissertação			X	X
Estudo do K8s e ferramentas de teste	X	X		
Criar aplicações exemplo		X		
Instalação do K8s local	X	X		
Estudo da escalabilidade		X	X	
Testes da plataforma		X	X	X

Tabela 3.1: Planeamento do trabalho.

3.2 Conclusão

Os orquestradores de *containers* como o K8s trouxeram ferramentas centralizadas para gerir os *containers*, possibilitando a construção de ambientes produtivos de grande escala, por exemplo, ambientes com centenas de microserviços disponibilizados para milhares de utilizadores.

Neste trabalho, devido à limitação de recursos não foi possível explorar a criação de um *cluster* de K8s com vários nós.

Capítulo 4

Contexto experimental

4.1 Introdução

Neste capítulo, é apresentado o ambiente experimental onde foram realizados os testes para o estudo do *autoscaling* no K8s. Para isso foi usada uma máquina virtual onde foi instalado o *microk8s* que permite uma instalação local do K8s. De seguida foi criada uma aplicação exemplo, que designamos por *API Challenge Java* da qual foram implantadas duas versões, v1 e v2, no K8s e que posteriormente serão sujeitas a testes de carga para estudo do comportamento do sistema de *autoscaling* para diferentes configurações do mesmo. De seguida descrevemos como configurar o *autoscaling*, mostrando um ficheiro Yet Another Markup Language (YAML) exemplo, e a ferramenta *JMeter* que foi escolhida para simular os acessos à aplicação exemplo, para testes de carga.

4.2 Máquina Virtual

Para a criação da máquina virtual, que será utilizada para o contexto experimental, foi escolhido o *Virtual box* [Ora23], um software desenvolvido pela *Oracle* e cujo objetivo é a criação e utilização de máquinas virtuais. Esta máquina virtual foi instalada numa máquina base com 32*Gb* de RAM um processador *i7-12700* 2.10*GHz* com 12 cores e 1*Tb* de memória.

Para a instalação do K8s foi criada uma máquina virtual *Ubuntu* na versão 22.04.2 com 8*Gb* de Random Access Memory (RAM) com um processador de 4 núcleos e 50*Gb* de memória e foi escolhida a ferramenta *microk8s* [Can23] visto que esta é indicada para instalações locais de K8s com um *cluster* único, que é adequado para testes de funcionalidades do K8s.

As figuras 4.1, 4.2, 4.3 e 4.4 descrevem a criação da máquina virtual, incluindo respetivamente a atribuição do nome(Teste), a escolha do Sistema operativo (SO) (*Ubuntu* 22.04.2), definição do utilizador (teste), os recursos da máquina (número de cores e memória) e o espaço em disco. A figura 4.5 sumariza as definições da máquina criada.

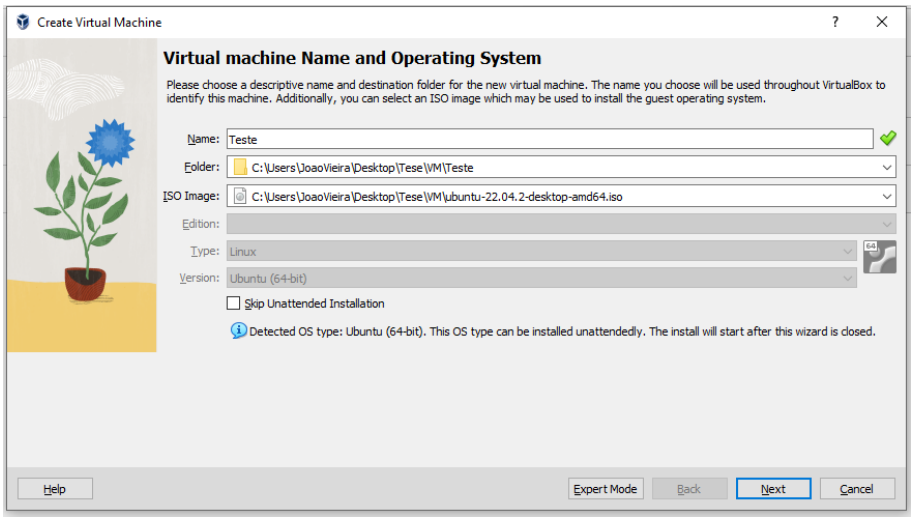

Figura 4.1: Definições iniciais da máquina virtual

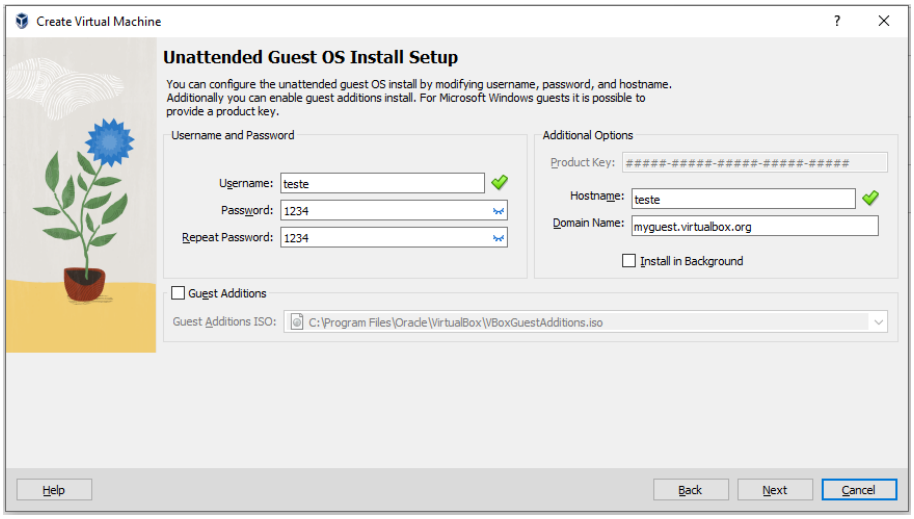

Figura 4.2: Definição do utilizador

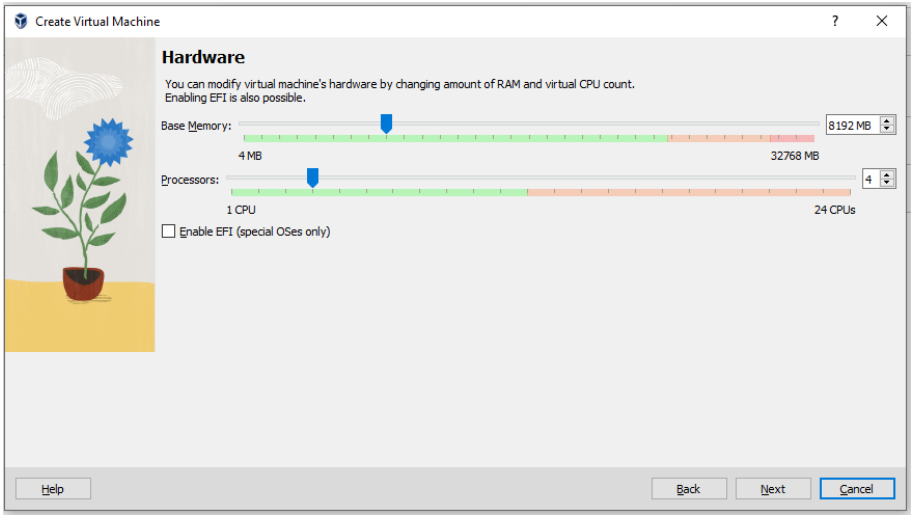

Figura 4.3: Recursos disponibilizados para a máquina virtual.

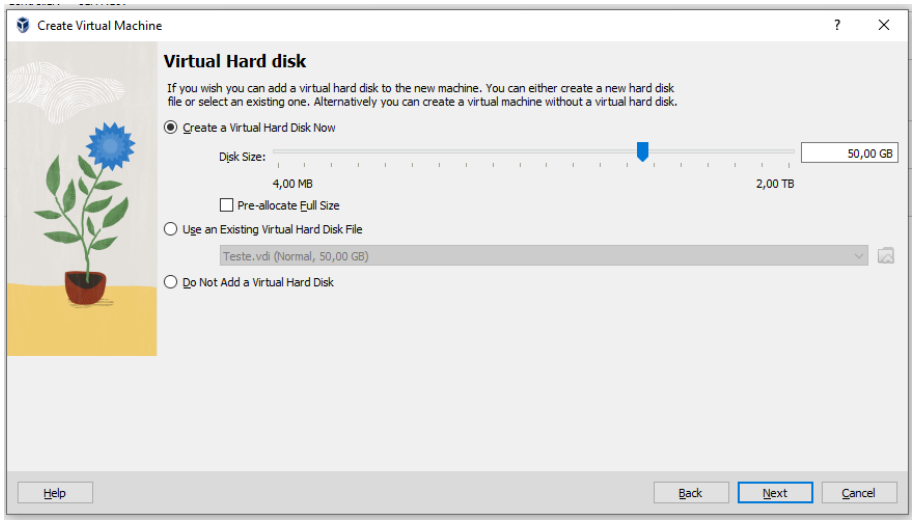

Figura 4.4: Espaço em disco disponibilizado para a máquina virtual

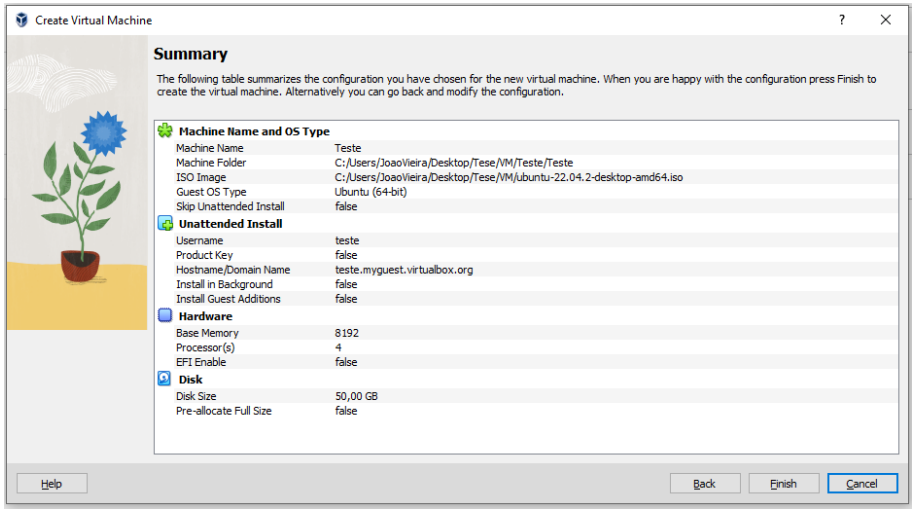

Figura 4.5: Sumário das definições

4.3 Microk8s

O *Microk8s* é uma ferramenta leve cujo objetivo é correr *clusters* de K8s em ambientes locais. Este oferece uma maneira fácil e rápida de começar a utilizar o K8s, possibilitando a desenvolvedores e equipas uma forma rápida de experimentar e desenvolver aplicações baseadas em *containers* sem que haja necessidade de configurar um ambiente de K8s complexo.

A instalação do *microk8s* é feita através da linha de comandos e começa-se por instalar os pacotes necessários no *Ubuntu* com recurso ao comando:

```
sudo snap install microk8s --classic
```

O *microk8s* oferece uma serie de *add-ons* que podemos utilizar no nosso K8s e tendo finalizado a instalação pode-se correr o seguinte comando para verificar estes *add-ons*:

```
microk8s status --wait-ready
```

A figura 4.6 demonstra os *add-ons* disponíveis pelo *microk8s* e o seu estado. Por exemplo, podemos observar que o *addon metrics-server* está *enabled* visto que irá ser usado para medir recursos em utilização.

```
joaovieira@Teste:~$ microk8s status --wait-ready
microk8s is running
high-availability: no
  datastore master nodes: 127.0.0.1:19001
  datastore standby nodes: none
addons:
  enabled:
    dashboard            # (core) The Kubernetes dashboard
    dns                  # (core) CoreDNS
    ha-cluster           # (core) Configure high availability on the current node
    helm                 # (core) Helm - the package manager for Kubernetes
    helm3                # (core) Helm 3 - the package manager for Kubernetes
    metrics-server       # (core) K8s Metrics Server for API access to service metrics
  disabled:
    cert-manager         # (core) Cloud native certificate management
    community            # (core) The community addons repository
    gpu                  # (core) Automatic enablement of Nvidia CUDA
    host-access          # (core) Allow Pods connecting to Host services smoothly
    hostpath-storage     # (core) Storage class; allocates storage from host directory
    ingress              # (core) Ingress controller for external access
    kube-ovn             # (core) An advanced network fabric for Kubernetes
    mayastor             # (core) OpenEBS MayaStor
    metallb              # (core) Loadbalancer for your Kubernetes cluster
    minio                # (core) MinIO object storage
    observability        # (core) A lightweight observability stack for logs, traces and metrics
    prometheus           # (core) Prometheus operator for monitoring and logging
    rbac                 # (core) Role-Based Access Control for authorisation
    registry             # (core) Private image registry exposed on localhost:32000
    storage              # (core) Alias to hostpath-storage add-on, deprecated
```

Figura 4.6: *Add-ons* disponíveis

O comando *kubectl* é utilizado para comunicação com Application Programming Interface (API) *server* do K8s e é com recurso a este que iremos interagir com o ambiente e iremos ter acesso às várias informações do mesmo.

Podemos utilizar o seguinte comando para verificar os nós do nosso K8s:

```
microk8s kubectl get nodes
```

Este irá devolver o nome dos nós que temos no nosso ambiente como demonstra a figura 4.7, neste caso podemos ver que o nó teste foi criado à 42 dias.

```
joaovieira@Teste:~$ microk8s kubectl get nodes
NAME     STATUS   ROLES    AGE   VERSION
teste    Ready    <none>   42d   v1.26.4
```

Figura 4.7: Nós disponíveis no ambiente K8s

4.4 *Deployment* da aplicação exemplo no *Microk8s*

Para testar as funcionalidades do *deployment* de aplicações foi criada uma aplicação em *Java*, na versão 11, que utiliza a *framework Spring Boot* na versão 2.7.10. Nesta aplicação foi incluída uma API que visa a criação de registos numa base de dados em memória, o Sistema de gestão de Base de Dados (SGBD) *H2*. O objetivo da utilização desta aplicação, para além de servir como base de teste para a criação do *deployment* no ambiente de K8s, é a criação de altas cargas no *pod* com vista à exploração das funcionalidades de *auto scaling*.

Nesta aplicação foi adicionada uma estrutura *Record* com os campos *id* (de tipo *Long*), *name* (de tipo *String*) e *createdDate* (de tipo *Instant*). Através desta classe (*Record*) é mapeada uma tabela para a base de dados relacional *H2*. Cada campo deste *Record* irá corresponder a uma coluna da tabela.

Foi também criada uma API com o seguinte caminho:

```
/api/v1/create-record
```

Esta *API* foi criada para dois casos, *v1* e *v2*, ambas aceitam um parâmetro *size* que irá determinar a quantidade de *Records* que irão ser gravados na base de dados de forma sucessiva e sem substituir linhas já existentes.

Quando fazemos um pedido à API, vai ser despoletado um processo que será iterado tantas vezes quantas for definido no parâmetro *size*. Este processo, no caso *v1*, consiste em popular os campos *name* e *createdDate* da estrutura *Record* para que em cada iteração a estrutura seja guardada em base de dados e adicionada a uma lista para ser posteriormente devolvida como resposta ao pedido à API. O valor *name* é populado com uma *string* com cinquenta caracteres gerada de forma aleatória e o campo *createdDate* assume a data e a hora do sistema.

Tendo em conta que a aplicação descrita a cima apenas tem um acesso à base de dados e não tem muito processamento de dados associado, foi criado o caso *v2*.

Esta nova aplicação, *v2*, tem também uma base de dados em memória, *H2*, uma API com o mesmo caminho que aceita o mesmo parâmetro *size* e possui também uma estrutura *Record* com o mesmo objetivo de ser guardada na base de dados. A esta nova estrutura *Record* foram adicionados vinte e seis novos campos do tipo *string* e foi replicada cinco vezes, ficando a base de dados com seis tabelas. Nesta versão, para cada pedido à API, serão adicionados *size* registos às seis tabelas de base de dados.

O processo a seguir é muito semelhante à aplicação *v1*, no entanto, em vez de apenas popularmos os campos *name* e *createdDate*, populámos também os vinte e seis novos campos adicionados com uma *string* com cinquenta caracteres gerados de forma aleatória. O resto do processo de popular o campo *createdDate* e gravação em base de dados, é igual à aplicação *v1*.

Tendo em conta que o restante processo de *deployment* é igual para ambas as aplicações, os processos irão ser explicados com base na aplicação *v1* de forma a não repetir processos que são iguais.

4.4.1 *Dockerfile* e *Docker image*

Nesta secção são descritos todos os passos para a criação da imagem *docker* que será utilizada para o *deployment* bem como a criação do ficheiro YAML para o mesmo e o serviço associado para que seja exposta uma porta para ser possível interagir com a aplicação.

O K8s baseia-se em imagens *Docker* para disponibilizar as aplicações. Tendo isto em conta é necessário criar a imagem *docker* da aplicação e disponibiliza-la num repositório para poder ter

a aplicação pronta a ser utilizada num *deployment*.

Para criar a imagem *docker* é definido um ficheiro denominado de ***Dockerfile*** e colocamos a seguinte especificação no ficheiro descrita abaixo:

```
FROM openjdk:11-jre-slim
WORKDIR /app
COPY target/apichallenge-0.1.2.jar /app
EXPOSE 8080
CMD ["java", "-jar", "apichallenge-0.1.2.jar"]
```

1. *FROM openjdk:11-jre-slim* - Este primeiro ponto define a imagem base da imagem, este serve de ponto inicial para a construção da imagem *docker* e serve também como um sistema operativo para a imagem.
2. *WORKDIR /app* - Aqui é definido a pasta onde os comandos do *docker* irão fazer efeito.
3. *COPY target/apichallenge-0.1.2.jar /app* - Nesta terceira linha, é copiado o código compilado, ou seja, o ficheiro *.jar* para a pasta definida no ponto 2.
4. *EXPOSE 8080* - Como o objetivo é disponibilizar uma API é necessário expor uma porta para interação com a aplicação.
5. *CMD [”java”, -jar”, ”apichallenge-0.1.2.jar”]* - Este último ponto tem como objetivo dizer quais os comandos a correr para executar a aplicação assim que o *container docker* seja inicializado.

Depois de ter o ficheiro *Dockerfile* criado no projeto é necessário compilar e gerar o ficheiro *.jar* da aplicação e assim correr o comando:

```
docker build -t <image_name> .
```

Este comando irá construir a imagem *docker* e guardá-la no nosso repositório de imagens local. Para verificar as imagens que temos disponíveis podemos correr o seguinte comando e verificar um *output* parecido ao da figura 4.8.

```
docker images
```

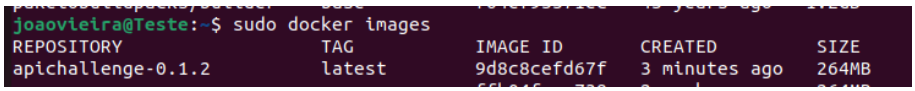

Figura 4.8: Imagens disponíveis no repositório local

Para disponibilizar a imagem de forma pública é necessário ter um repositório. O próprio *docker* disponibiliza de forma gratuita um repositório onde podemos guardar imagens num repositório público, que se chama *docker hub* [Doc23a].

Para podermos guardar a imagem, precisamos primeiro de fazer o *login* no *docker hub* através do comando:

```
docker login
```

Antes de enviarmos a imagem para o repositório do *docker hub* é necessário criar o repositório na página do mesmo. A figura 4.9 exemplifica a página de criação. Aqui atribui-se um *namespace* e dá-se um nome ao repositório.

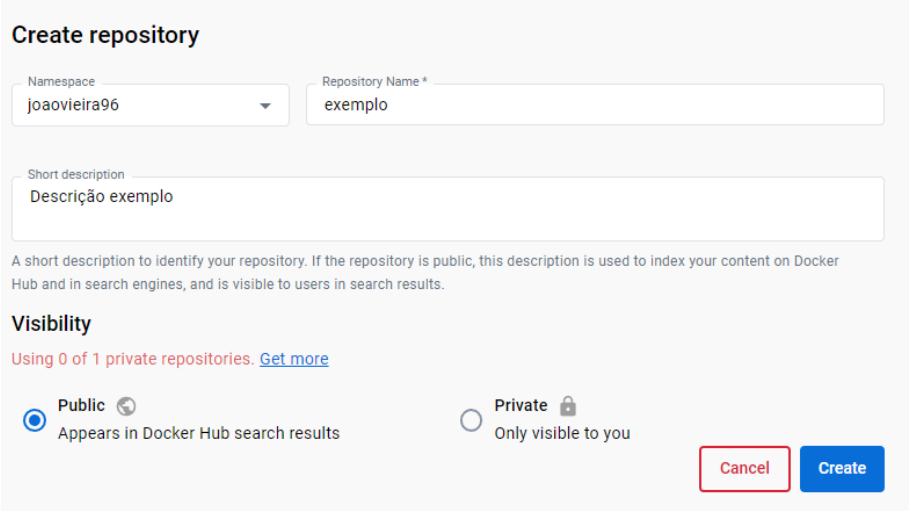

Figura 4.9: Página de criação do repositório

Depois da autenticação e da criação do repositório é necessário alterar o nome da imagem no repositório local, visto que, para enviarmos a imagem para o *docker hub* é necessário identificarmos o *namespace* da imagem dentro do repositório local para que o *docker* consiga identificar qual o *namespace* que irá utilizar para guardar a imagem. Isto é feito com recurso ao comando:

```
docker tag <imagem-local>:<tag-local> <namspace>/<nome>:<tag>
```

No comando são indicados o nome da imagem e a *tag* que já existe no repositório local e a segunda parte, o *namespace* configurado durante a criação do repositório no *docker hub*, o seu nome e *tag* que é habitualmente utilizada para definir versões das imagens.

Após todos estes passos podemos utilizar o comando *docker push* para guardar a imagem no repositório do *docker hub* podendo-se verificar na figura 4.10 que a imagem (*apichallenge:0.1.2*) está agora guardada no repositório público.

```
docker push joaovieira96/apichallenge:0.1.2
```

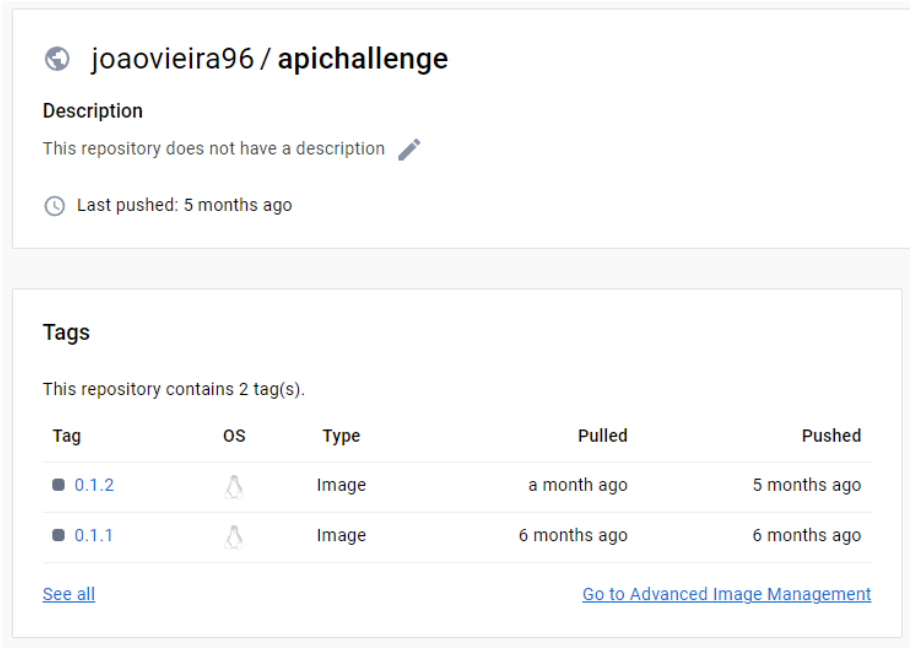

Figura 4.10: Nova imagem *docker* disponibilizada

4.4.2 *Deployments*

Para fazer o *deployment* de aplicações é necessário criar um ficheiro YAML. Este ficheiro irá ter toda a especificação necessária para o K8s fazer o *deployment* da aplicação. Desde o nome do *pod* até às variáveis de ambiente, todas as configurações podem estar definidas num único ficheiro, no entanto, cada aplicação necessita do seu próprio ficheiro de *deployment*.

Abaixo podemos ver o YAML escrito para o *deployment* da aplicação já mencionada.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: apichallenge
spec:
  selector:
    matchLabels:
      app: apichallenge
  replicas: 1
  template:
    metadata:
      labels:
        app: apichallenge
    spec:
      containers:
        - name: apichallenge
          image: joaovieira96/apichallenge:0.1.2
          imagePullPolicy: IfNotPresent
          resources:
            requests:
              cpu: "125m"
```

```
        memory: "1024Mi"
      limits:
        cpu: "300m"
        memory: "1024Mi"
    ports:
      - containerPort: 8080
```

No exemplo a cima podemos encontrar as seguintes configurações:

1. *apiVersion* - Define a versão da API que o K8s irá utilizar para interpretar o ficheiro YAML
2. *kind* - Define o tipo de recurso que está a ser a criado
3. *metadata* - Define todos os meta dados do *deployment*. Alguns exemplos de meta dados que podemos incluir são:
   (a) *name* - Define o nome do *deployment*
   (b) *namespace* - Define o *namespace* em que o *deployment* vai ser criado. Este serve para organizar e isolar recursos dentro do *cluster*.
   (c) *labels* - Define pares chave-valor que são usados para identificar *deployments*
4. *spec* - *Spec* significa *specification* e este campo define várias configurações e parâmetros para a criação do *deployment*. Os seguintes elementos podem ser encontrados sob a definição da *spec*:
   (a) *selector* - Define rótulos para identificar os *pods*.
   (b) *replicas* - Define o número de réplicas que o *deployment* vai ter por omissão. De notar que esta é uma forma estática de definir o número de réplicas da aplicação.
   (c) *template* - Define um *template* que descreve a criação do *deployment* e este é usado como base para a criação de réplicas. Aqui podemos encontrar as seguintes especificações:
      i. *spec* - Aqui encontramos novamente a configuração de um especificação. Esta permite uma maior configuração e é aqui que vamos fazer a configuração do ***container*** que inclui as seguintes configurações:
         A. *image* - Define a imagem *docker* a ser utilizada no *deployment*. Se não for configurado nenhum repositório de imagens (como encontramos no exemplo), o K8s irá procurar a imagem no repositório do *Docker hub*.
         B. *imagePullPolicy* - Quando o K8s procura a imagem *docker*, este irá fazer o *download* da imagem para o *pod*. Este campo define como agir em relação à imagem. Podemos definir para fazer sempre o *download* da imagem, fazer apenas o *download* caso a imagem ainda não exista no *pod* ou nunca fazer o *download* da imagem.
         C. *resources* - Define os recursos computacionais do *pod*. Esta configuração tem dois sub campos, o ***limits*** e o ***requests*** e estes servem para definir os recursos máximos e iniciais do *pod*, respetivamente. No ficheiro exemplo para o *cpu* temos *125m* que corresponde a um oitavo de um *core* e para memória temos *1024Mi* que corresponde a *1024megabyte*.

(d) *ports* - Define configurações para *networking* do pod. No exemplo a cima ainda encontramos a configuração ***containerPort*** que define a porta em que a aplicação irá estar à escuta.

Por fim, para aplicar o ficheiro YAML apenas tem de ser executado o seguinte comando mencionando o nome do ficheiro criado com a *flag* -f que especifica que vamos utilizar o nome do ficheiro. A partir deste momento uma instância da aplicação *v1* será inicializada.

```
microk8s kubectl apply -f apichallenge-deployment.yaml
```

Existe também um comando para verificar que *deployments* existem a correr no *cluster*.

```
microk8s kubectl get deployments
```

Este comando terá um *output* semelhante ao da seguinte figura.

```
joaovieira@Teste:~/Desktop/KUBERNETES/app-deployments$ microk8s kubectl get deployments
NAME           READY   UP-TO-DATE   AVAILABLE   AGE
apichallenge   1/1     1            1           20d
```

Figura 4.11: *Deployments* no K8s

4.4.3 *Services*

Um *Service* no contexto de K8s é um recurso utilizado para estabelecer o *networking* das nossas aplicações. No exemplo da aplicação *v1* apesar de haver uma porta exposta durante o *deployment* esta não é acessível fora do *pod*, sendo então necessário criar uma ponte para se comunicar com a aplicação.

A configuração do um serviço é ligeiramente diferente da configuração do *deployment*. No serviço podemos definir os seguintes tipos de serviço.

1. *ClusterIP* - Este é o tipo *default* de um serviço, ele expõe uma porta interna que apenas é alcançável dentro do *cluster*.
2. *NodePort* - Este tipo expõe uma porta e permite o acesso exterior ao *cluster*.
3. *LoadBalancer* - Permite o balanceamento de pedidos para o *pod*.

Este recurso oferece configurações de *networking* e abaixo podemos encontrar o exemplo de um serviço que foi configurado aquando da criação da aplicação *v1*.

```
apiVersion: v1
kind: Service
metadata:
  name: apichallenge-nodeport
spec:
  type: NodePort
  selector:
    app: apichallenge
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 8080
```

Por fim, à semelhança dos *deployments*, tem que se aplicar o ficheiro YAML criado e pode-se verificar os serviços existentes com recurso ao comando:

```
microk8s kubectl get services
```

Este deverá ter um *output* parecido ao da figura 4.12. Como podemos observar está instalado um serviço do tipo *NodePort* para a aplicação. O serviço do tipo *ClusterIP* faz parte da instalação do próprio K8s.

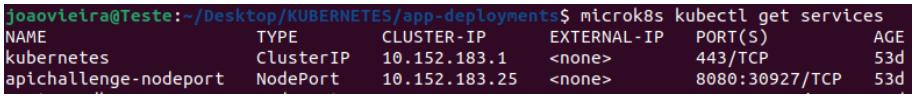

Figura 4.12: Serviços do K8s

4.5 *Autoscaling*

O *autoscaler* tem a responsabilidade de ajustar o número de réplicas de um *pod*, dependendo da utilização dos recursos, tais como *CPU* ou memória. Ele monitoriza continuamente a utilização dos recursos pelos *pods* e, ao detetar um elevado uso, aumenta o número de réplicas. Por outro lado, se a utilização diminui, o *autoscaler* reduz o número de réplicas. Este processo assegura uma utilização eficiente dos recursos do *cluster* e a manutenção da disponibilidade da aplicação.

Para a criação e configuração do *autoscaler* é necessário primeiro ter uma ferramenta de monitorização de recursos. O *microk8s* disponibiliza um *addon* chamado de *metrics-server* e este serve para esse mesmo efeito. Para ativar este *addon* podemos correr o comando:

```
microk8s enable metrics-server
```

Tendo o *metrics-server* ativo, tem de se criar o recurso *HorizontalPodAutoscaler* com a seguinte especificação:

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: apichallenge-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: apichallenge
  minReplicas: 1
  maxReplicas: 5
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
```

```
    type: Utilization
    averageUtilization: 50
```

Neste recurso podemos ver as configurações de número mínimo (*minReplicas: 1*) e máximo de réplicas (*maxReplicas: 5*) e também qual o recurso que o *auto scaler* irá ter em conta para decidir aumentar ou diminuir o número de réplicas, neste caso o recurso é o *cpu*. Alternativamente este recurso poderia ser a memória. Pelo ficheiro exemplo, quando for atingido 50% (*averageUtilization*) de utilização de *cpu* o K8s vai subir gradualmente o número de réplicas.

Se quisermos fazer *autoscaling* de várias aplicações teremos que criar um ficheiro de configuração para cada aplicação. O recurso que resulta da execução do ficheiro fica associado à aplicação através da configuração vista dentro do *scaleTargetRef*. O valor do *apiVersion* e do *kind* do *autoscaler* tem que ter valores iguais aos do *deployment* nas mesmas configurações, já o *name* no *autoscaler* tem que ter o mesmo valor visto dentro da configuração *spec.selector.matchLabels.app* do *deployment*.

Depois de criado o YAML pode-se criar o recurso, à semelhança do que se fez com o *deployment* e o *service*. Para verificar os *HorizontalPodAutoscaler* existentes podemos correr o seguinte comando:

```
microk8s kubectl get hpa
```

Este irá ter um *output* parecido ao da seguinte figura 4.13. Note-se que este último comando disponibiliza também a percentagem de utilização do recurso definido no YAML (*cpu*) que a aplicação está a consumir, sendo que, no exemplo dado na figura 4.13 a aplicação não está a sofrer qualquer tipo de carga.

```
joaovieira@Teste:~/Desktop/KUBERNETES/app-deployments$ microk8s kubectl get hpa
NAME               REFERENCE                 TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
apichallenge-hpa   Deployment/apichallenge   4%/50%    1         5         1          20d
```

Figura 4.13: *Horizontal Pod Autoscaler*

A decisão de aumento ou diminuição de réplicas é definida pela seguinte formula [Goo23a].

```
desiredReplicas = ceil[currentReplicas * ( currentMetricValue / desiredMetricValue )]
```

Nesta fórmula, o número de réplicas desejado (*desiredReplicas*), para o qual o sistema vai escalar, é dado pelo quociente entre o valor da métrica alvo (*desiredMetricValue* - neste caso a percentagem de *cpu*) e o valor atual (*currentMetricValue*) da mesma métrica multiplicado pelo o número de atual de réplicas.

4.6 A ferramenta de teste e monitorização: *JMeter*

Tendo as aplicações prontas a serem utilizadas, irá passar-se então para o teste das funcionalidades de *auto scaling*. Para isto, será necessária uma ferramenta para provocar uma grande quantidade de pedidos *REST* na aplicação tendo como objetivo aumentar os recursos que esta utiliza. A ferramenta escolhida foi o *JMeter* [Fou23].

O *JMeter* é uma ferramenta *open-source* para realização de testes de carga e medir a *performance* de aplicações. Estes testes de carga são, por exemplo, grandes quantidades de chamadas a APIs e por norma servem para perceber como é que as aplicações se comportam quando estão a ser sujeitas a um elevado processamento.

4.6.1 *JMeter*

O *JMeter* é uma ferramenta *open-source* que permite executar testes de carga em aplicações *web*, oferecendo uma *interface* gráfica para a configuração dos testes como ilustra a figura 4.14.

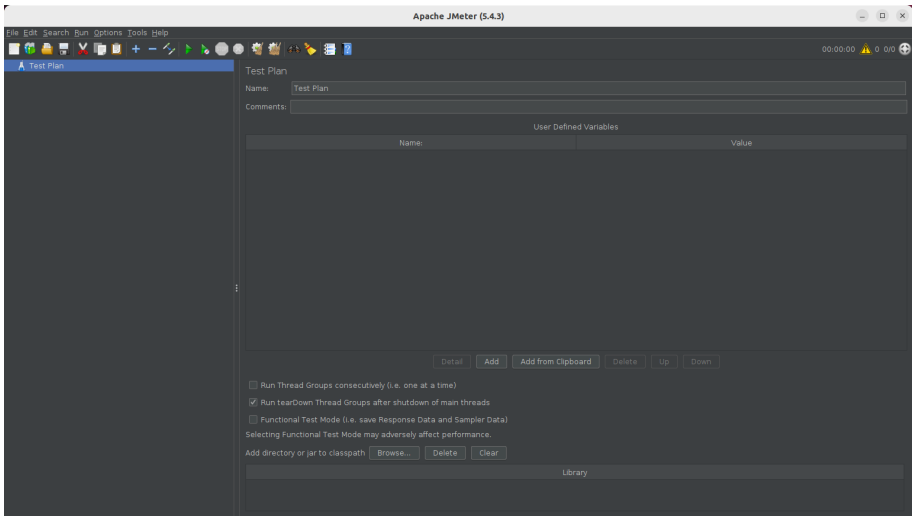

Figura 4.14: *Interface* do *JMeter*

Tendo o *JMeter* a correr, é possível criar uma serie de recursos para os testes, como por exemplo *listeners* para obtenção de resultados ou *timers* para calendarizar os testes. Para este caso, será criado um ***Thread Group*** como se pode ver na figura 4.15.

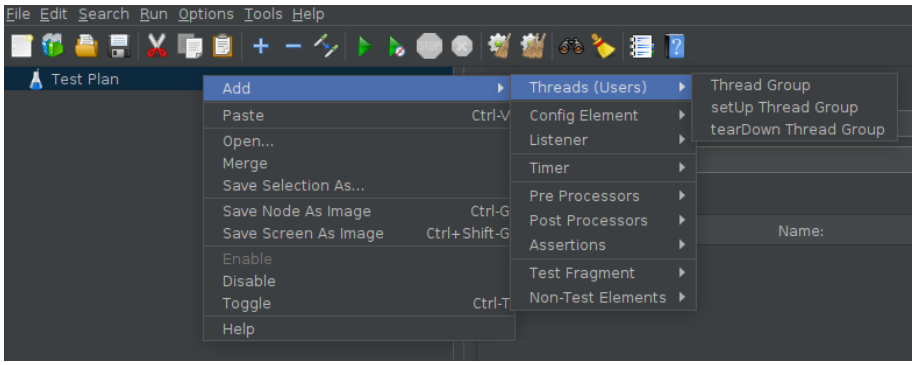

Figura 4.15: Criação do *Thread Group*

Um *Thread Group* é o elemento inicial de um plano de testes. O *Thread Group* é um grupo de *threads* que vão executar um mesmo cenário e é nesta configuração que definimos o número de repetições que a ação que pretendemos irá executar como mostra a figura 4.16. No campo *number of threads (users)* será indicado o número de *threads* que queremos criar.

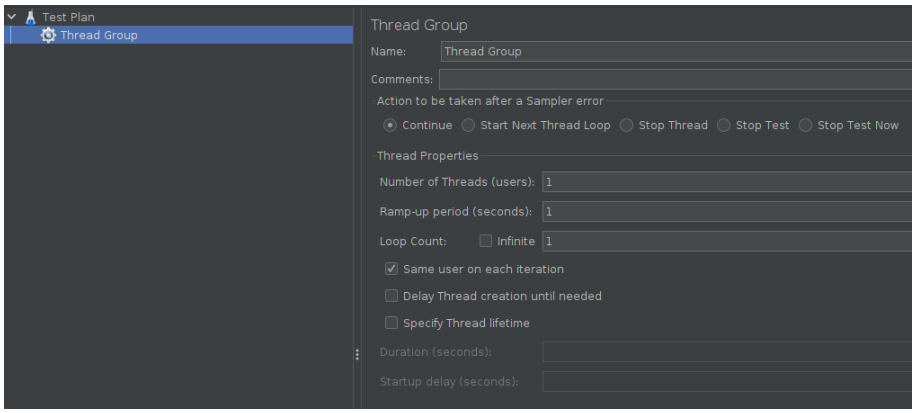

Figura 4.16: Configuração do *Thread Group*

Depois de configurar o *Thread Group* passamos à criação do recurso que queremos que se repita, neste caso, um ***HTTP Request*** (figura 4.17).

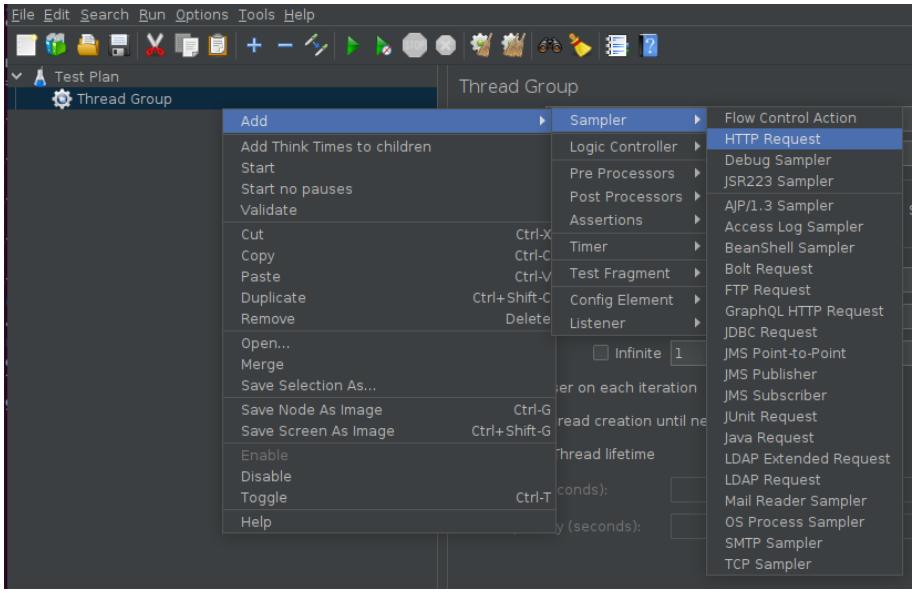

Figura 4.17: Criação da *HTTP Request*

A configuração do pedido *HTTP* define as várias especificações que são habituais num pedido *REST* como o *ip* e porta da aplicação, o tipo de pedido que pretendemos fazer, o *path* para a API e podemos configurar também parâmetros do pedido (figura 4.18). Como podemos ver o *ip* está definido para *localhost*, a porta para 32058, o tipo de pedido é um *GET*, o *path* é */api/v1/create-record* e o *size* é 200.

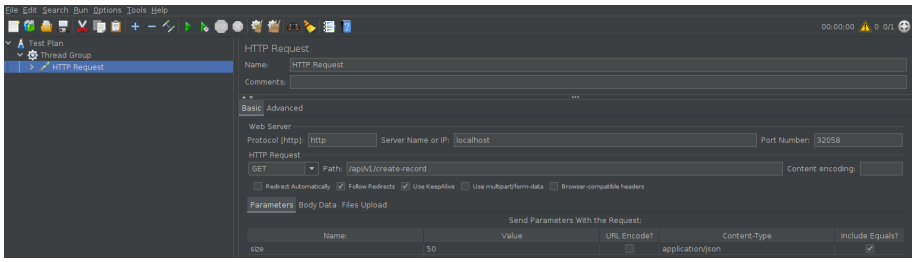

Figura 4.18: Configuração da *HTTP Request*

Por fim, para verificarmos o resultado dos pedidos que se vão fazer, tem que se criar um último recurso para estar à escuta destes resultados. Este recurso chama-se *listener* e como podemos ver na figura 4.19 o *listener* escolhido é do tipo ***Summary Report***. Este apresenta informações sobre o teste que foi feito pelo *Thread Group* como o número de pedidos e a percentagem de erro, que são métricas importantes para os testes que se pretendem realizar.

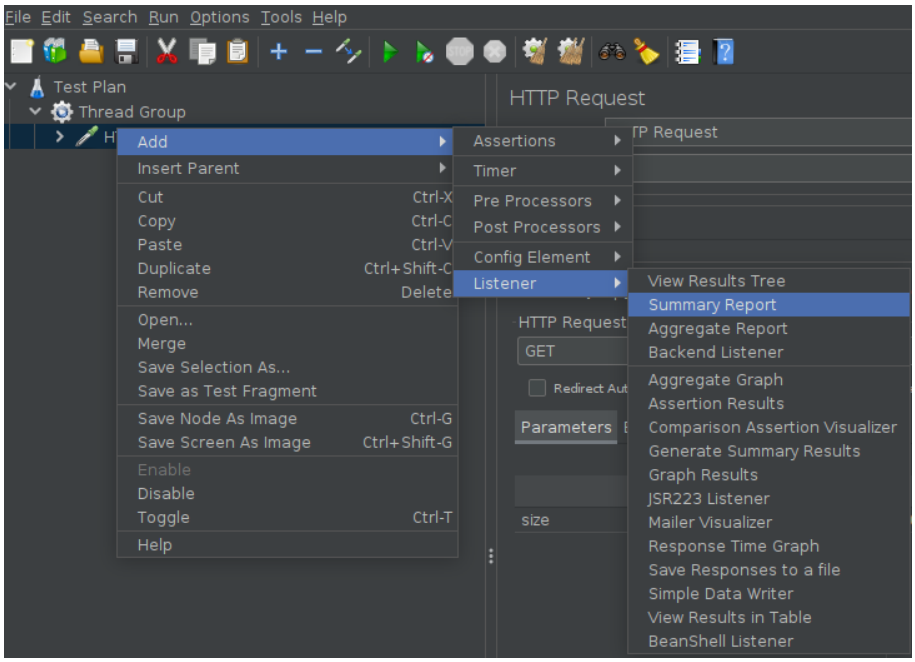

Figura 4.19: *Listeners* disponíveis

Tendo em conta que este *listener* é meramente informativo, este não necessita de configurações. Na figura 4.20 pode-se ver um exemplo com cem pedidos efetuados.

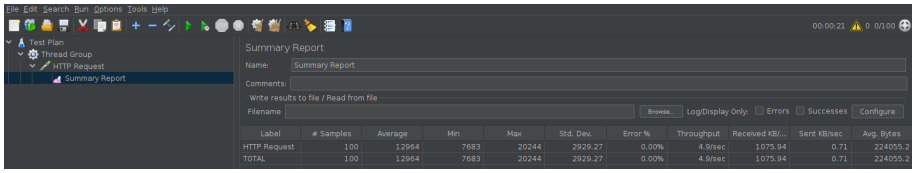

Figura 4.20: *Summary Report*

Neste capítulo apresentámos o ambiente experimental construido para a realização de testes de carga utilizando o *JMeter* para perceber como é que a aplicação se comporta com uma e várias réplicas. Os testes são apresentados no próximo capítulo.

Capítulo 5

Testes e Resultados

5.1 Testes

Para avaliar a escalabilidade proporcionada por uma plataforma de K8s comparamos o tempo de execução e os recursos usados pelas aplicações exemplo com uma única réplica e com várias réplicas, através de testes de carga feitos com o *JMeter*. Com esta ferramenta foi possível efetuar um grande número de chamadas à API da aplicação, simulando assim, momentos de grande processamento.

O objetivo destes testes de carga é provocar um processamento alto no *microserviço* e nas suas réplicas de forma a perceber os tempos de resposta e assim entender o quão benéfico pode ser a utilização de um *HorizontalPodAutoscaler* para aumentar o número de réplicas de um certo *pod*. Adicionalmente, tendo em conta que altas cargas de pedidos também podem originar problemas na conexão do *microserviço* à sua base de dados, acabou por se avaliar também a percentagem de erros que foi possível verificar durante os testes de carga.

Um dos problemas encontrados logo desde início foi o facto de ser difícil controlar o quantidade de pedidos que seriam necessários fazer às aplicações de forma a provocar a carga necessária para o *HorizontalPodAutoscaler* subir um número especifico de réplicas. Assim, em vez de configurar o *autoscaling*, de forma a simular os testes pretendidos com o número de réplicas desejado, foi modificada a configuração do *deployment* para ter o número de réplicas planeado, isto é, o número de réplicas foi definido de forma estática.

Para as duas versões, *v1* e *v2*, da aplicação exemplo foram feitos três casos de teste. Um primeiro caso em que a aplicação não é replicada, um segundo caso com três réplicas e um terceiro caso com cinco réplicas. Para cada um destes casos foram simulados diferentes números de pedidos. Para ter resultados mais fidedignos foram feitos dez testes para cada um dos casos estando depois na tabela representada a média desses resultados.

Para verificar a utilização de *CPU* e memória utilizou-se o comando *htop* que nos oferece os valores de utilização destes dois recursos.

5.2 Resultados

Na tabela 5.1 apresentam-se, para a aplicação *v1*, os tempos de execução, em segundos, e a percentagem de erros para os três casos de estudo tendo para cada caso sido simulados cem, quinhentos e mil pedidos. Estes pedidos consistem em chamadas à API e todas elas fazem uso do parâmetro *size* com um valor de duzentos.

Nº Replicas	Nº Pedidos	Tempo de processamento do teste de carga (s)	Erro (%)
1	100	4,7	0
1	500	25,8	0
1	1000	57,8	1,43
3	100	2.6	0
3	500	9.0	0
3	1000	19.3	0
5	100	1,7	0
5	500	5,5	0
5	1000	9,8	0

Tabela 5.1: Testes de carga para a aplicação *v1*

Na tabela 5.1 conseguimos verificar que os tempos de resposta descem de forma acentuada quando aumentamos o número de réplicas e vemos que existem alguns erros no caso de réplica única para mil pedidos devido a perda de ligação à base de dados pela quantidade de acessos que é feito.

Se olharmos para os tempos de resposta de um, três e cinco réplicas, conseguimos ver que o tempo diminui quase de forma proporcional ao número de réplicas, ou seja, para, por exemplo, mil pedidos, o tempo de resposta aproxima-se aos sessenta segundos, já para três réplicas o tempo se aproxima de vinte segundos o que é indicativo que cada uma das três réplicas processou tanto quanto a réplica única. Já para o caso de cinco réplicas, apesar de poder haver uma maior discrepância, o mesmo raciocínio pode ser feito.

Para analisar a utilização de recursos do sistema consideraram-se os mesmos três casos de estudo (uma, três e cinco réplicas) sem carga e com uma carga de quinhentos pedidos.

A tabela 5.2 apresenta a utilização dos recursos do sistema nos casos de uma, três e cinco réplicas. Aqui podemos ver a utilização de *CPU* sob a forma de intervalo e o valor de memória considerando a aplicação sem carga e com carga de quinhentos pedidos.

O que se consegue verificar na tabela 5.2 é que, no caso de uma réplica, sem estar sujeita a carga, os quatro *cores* de *CPU* tem valores entre os dez a quinze por cento de utilização enquanto que a utilização de memoria se mantém próxima dos 3.3*Gb*.

Para uma única replica, ao se aplicar um teste de carga de quinhentos pedidos consegue-se verificar picos de utilização dos quatro *cores* para valores próximos aos noventa por cento e os valores de memória aumentam para aproximadamente 4*Gb*.

Já para o caso de termos as cinco réplicas disponíveis, sem submeter os *pods* a nenhuma carga, tem os *cores* de *CPU* com valores entre os vinte e os trinta por cento e a memória em valores próximos dos 4.5*Gb*.

Para as cinco réplicas, ao submeter uma carga de quinhentos pedidos, os *cores* do *CPU* atingem picos de utilização de cerca de noventa por cento enquanto que a memória chega a valores perto de 5.5*Gb* de utilização.

Nº Replicas	Nº Pedidos	*CPU*	Memória
1	0	[10% - 15%]	3.3*Gb*
1	500	[80% - 90%]	4.0*Gb*
3	0	[10% - 15%]	3.6*Gb*
3	500	[80% - 90%]	4.3*Gb*
5	0	[20% - 30%]	4.5*Gb*
5	500	[80% - 90%]	5.5*Gb*

Tabela 5.2: Valores de *CPU* e Memória para a aplicação *v1*

O que se pode concluir da tabela 5.2 é que a utilização de *CPU* se mantém similar nos três exemplos de réplicas em carga, os valores oscilam entre os oitenta e noventa por cento. Para os casos sem carga, a utilização de *cpu* é ligeiramente superior para o caso das cinco réplicas (entre vinte a trinta por cento) em relação aos casos de uma e três réplicas (entre dez a quinze por cento), o que é justificável com o número superior de réplicas a utilizar recursos do sistema.

Já para a memória, conseguimos verificar que a quantidade de recursos vai aumentado nos três casos de réplicas, havendo uma maior utilização de recursos quando a aplicação está sob carga, como seria de esperar.

Depois de perceber o comportamento do sistema e das aplicações para a aplicação *v1*, foram aplicados os mesmos casos de teste para a aplicação *v2*. Nestes testes apenas se alterou o valor do parâmetro *size* para vinte em vez dos duzentos feitos nos testes para a *v1*. Esta alteração resultou do facto de ao realizarmos os testes com um *size* de duzentos, a percentagem de erros ser elevada.

A tabela 5.3 apresenta, para a aplicação *v2*, os resultados dos tempos de execução e percentagens de erro para os mesmos casos e número de pedidos considerados para a aplicação *v1*.

Nº Replicas	Nº Pedidos	Tempo de processamento do teste de carga (s)	Erro (%)
1	100	9.4	0
1	500	103.4	27.64
1	1000	214.3	47.10
3	100	4.2	0
3	500	28.8	0
3	1000	51.6	0.33
5	100	3.4	0
5	500	13.2	0
5	1000	30.5	0

Tabela 5.3: Testes de carga para a aplicação *v2*

Como se pode observar na tabela 5.3 os tempos de execução de *v2* são, como seria de esperar, sempre superiores aos tempos de *v1*. Considerando o caso das três réplicas com cem pedidos o tempo de execução é menos de metade do caso de uma réplica e com quinhentos e mil pedidos o tempo de execução está abaixo dos trinta por cento (é de apenas vinte e quatro por cento para mil pedidos). Em relação ao caso das cinco réplicas o tempo de execução com cem pedidos é de trinta e seis por cento do tempo com uma réplica e com quinhentos e mil pedidos está abaixo dos quinze por cento.

No que diz respeito à percentagem de erros observamos que, para *v2*, se só existir uma réplica com quinhentos pedidos, a percentagem de erro já é bastante elevada (cerca de vinte e oito por cento) passando para quase o dobro nos mil pedidos (quarenta e sete por cento). Quando a aplicação é replicada para três réplicas, para quinhentos pedidos não ocorrem erros, mas para mil pedidos voltamos a detetar percentagens de erro residuais (nem chega a um por cento). Já para cinco réplicas não são detetados nenhuns erros.

A análise da utilização de recursos é feita de forma igual à realizada para a aplicação *v1*, ou seja, utilizando os três casos de teste (uma, três e cinco réplicas) sem carga e com uma carga de quinhentos pedidos e a tabela 5.4 organiza os dados de forma igual à tabela 5.2 com a utilização de *CPU* sob a forma de intervalo e os valores de memória.

Na tabela 5.4 conseguimos ver que, de uma forma geral, tal como para *v1*, a utilização de memória vai aumentando à medida que é aumentado o número de réplicas e o número de pedidos.

Nº Replicas	Nº Pedidos	*CPU*	Memória
1	0	[10% - 15%]	4.0*Gb*
1	500	[70% - 90%]	4.3*Gb*
3	0	[10% - 15%]	4.6*Gb*
3	500	[75% - 90%]	5.5*Gb*
5	0	[10% - 15%]	4.7*Gb*
5	500	[80% - 90%]	6.2*Gb*

Tabela 5.4: Valores de *CPU* e Memória para a aplicação *v2*

Para melhor comparar os resultados obtidos nas duas aplicações apresentamos, na tabela 5.5, o ganho no tempo de execução (em percentagem) para cada uma das aplicações em relação ao tempo de execução com uma única réplica.

Nº Replicas	Nº Pedidos	Ganho de *v1* em relação a uma réplica (%)	Ganho de *v2* em relação a uma réplica (%)
3	100	45	55
3	500	65	72
3	1000	67	76
5	100	64	64
5	500	79	87
5	1000	83	86

Tabela 5.5: Ganhos do tempo de execução obtidos com a replicação

Como podemos observar na tabela 5.5 a aplicação com mais processamento, a *v2*, tem em todas as situações estudadas um ganho superior à *v1*. Para a aplicação *v2*, os ganhos no caso de cinco réplicas com quinhentos e mil pedidos são notáveis (perto dos noventa por cento).

Em relação à utilização de recursos entre *v1* e *v2* a comparação já é mais difícil. Enquanto que a utilização de memória é superior para a aplicação *v2* em todos os pontos de comparação, a utilização de *CPU* parece atingir os mesmos picos de utilização que a aplicação *v1*.

No mundo empresarial onde muitas vezes os clientes tem os tempos de processamento bem definidos através de Key Performance Indicator (KPI) pode ser útil utilizar esta abordagem para garantir que o número de réplicas permita atingir os índices desejados. Esta abordagem

demonstrou-se útil na avaliação de desempenho e na otimização de tempos de resposta, contribuindo para uma melhor experiência do utilizador e para a eficiência operacional do sistema em questão.

Para estas duas aplicações os resultados demonstram uma clara vantagem na replicação parecendo que as cinco réplicas são adequadas para o limite máximo a atribuir a um *HorizontalPodAutoscaler*.

Capítulo 6

Conclusão e Trabalho Futuro

6.1 Conclusão

Esta dissertação proporcionou uma maior compreensão sobre tecnologias de virtualização e orquestração de *containers*, com ênfase no K8s. Neste documento foi abordado o estado da arte da virtualização, *containers* e orquestradores de *containers* onde o K8s foi o maior objeto de estudo.

Após uma revisão teórica no estado da arte, procedeu-se à criação de uma máquina virtual onde se instalou a ferramenta *microk8s* para fazer todos os testes pretendidos. Finalizada a instalação foi feito o *deployment* de duas aplicações exemplo, passando por todo o processo de *deployment*, desde a criação das imagens *docker* até à criação do ficheiro YAML. Foi também explorado como criar um *service* e também foi abordado o *autoscaling* com a criação de um *HorizontalPodAutoscaler*.

Depois de ter o ambiente pronto, procedeu-se aos testes de carga às aplicações, utilizando o *JMeter*. Durante estes testes foi monitorizado o tempo de processamento dos testes de carga e os erros que dai resultavam, bem como, a percentagem de utilização de *CPU* e a memória utilizada pelo sistema que resultava destes testes. Daqui foi possível verificar que a utilização de múltiplas réplicas conduz à melhoria dos tempos de resposta das API. Esta análise poderá ser utilizada no futuro para perceber a necessidade de escalar aplicações e em caso de necessidade, perceber a quantidade de réplicas adequada para a aplicação em causa.

Esta dissertação representou um aprofundamento do meu conhecimento, permitindo a exploração de áreas até então não exploradas durante a minha trajetória profissional. Desde a criação *step-by-step* da imagem Docker - processo que pode inicialmente parecer complexo devido ao número de passos que têm que ser dados para a criação da imagem -, até a elaboração dos arquivos YAML essenciais para os *deployments* e *services*, esta experiência foi importante para compreender o significado de cada campo e sua aplicabilidade. Além disso, a aplicação prática do K8s nos testes conduzidos na plataforma será certamente uma experiência importante, uma vez que o K8s, em constante expansão, assume uma posição cada vez mais proeminente no universo de *Dev-Ops*.

6.2 Trabalhos Futuros

Esta dissertação lança as bases para a investigação de outros tópicos dentro do K8s. A exploração dos *services* foi maioritariamente superficial, concentrando-se principalmente no *NodePort*. Há uma oportunidade de aprofundar a compreensão e integração do recurso *LoadBalancer*, analisando como o implementar, vantagens e desvantagens, bem como sua aplicação em cenários

práticos.

Outro aspeto que merece uma análise mais detalhada é a configuração de um *cluster* de K8s com múltiplos nós. Dada a inclinação do K8s para ambientes de grande escala, a implementação de múltiplos nós assume uma relevância substancial na criação de uma infraestrutura mais robusta e responsiva às exigências dos clientes. Portanto, a exploração dessa configuração proporcionará uma compreensão aprofundada das estratégias para assegurar a resiliência e a eficácia operacional do sistema.

Por fim, tendo em conta que esta dissertação teve uma componente muito grande de estudo sobre *scaling* e *autoscaling*, seria também interessante tentar criar casos de teste adequados para que o *autoscaling* criado seja parte importante dos testes e não algo que fique no plano teórico, como acabou por acontecer neste trabalho. Para além disso, seria também interessante explorar outros tipos de *autoscaling*, como por exemplo, utilizando inteligência artificial.

Bibliografia

[Acr20] Acronis. What is hyper-v: The authoritative guide, 2020. [Online] https://www.acronis.com/en-sg/blog/posts/hyper-v-authoritative-guide/. Último acesso a 07 de Outubro de 2023. 4

[Aqu23] Aquasec. Containerized architecture: Components and design principles, 2023. [Online] https://www.aquasec.com/cloud-native-academy/container-security/containerized-architecture/. Último acesso a 07 de Outubro de 2023. 5

[Bas19] Bora Basyildiz. A brief history of container technology, 2019. [Online] https://www.section.io/engineering-education/history-of-container-technology/. Último acesso a 07 de Outubro de 2023. 4

[BR22] Agathe Blaise and Filippo Rebecchi. Stay at the helm: secure kubernetes deployments via graph generation and attack reconstruction. In *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, pages 59–69, 2022. 11

[BWZ15] Leonard J. Bass, Ingo Weber, and Liming Zhu. Devops - a software architect's perspective. In *SEI series in software engineering*, 2015. 1

[Can23] Canonical. Docker overview, 2023. [Online] https://microk8s.io. Último acesso a 07 de Outubro de 2023. 15

[Coo17] Joshua Cook. The docker engine. 2017. [Online] https://api.semanticscholar.org/CorpusID:56675905. Último acesso a 07 de Outubro de 2023. 6

[Cou22] Coursera. "introduction to virtualization: What is a virtual machine?". 2022. [Online] https://www.coursera.org/articles/what-is-a-virtual-machine. Último acesso a 07 de Outubro de 2023. 3

[Doc23a] Docker. Dockerhub, 2023. [Online] https://hub.docker.com. Último acesso a 07 de Outubro de 2023. 20

[Doc23b] Docker. Swarm mode overview, 2023. [Online] https://docs.docker.com/engine/swarm/. Último acesso a 07 de Outubro de 2023. 11

[DTR+18] Wito Delnat, Eddy Truyen, Ansar Rafique, Dimitri Van Landuyt, and Wouter Joosen. K8-scalar: A workbench to compare autoscalers for container-orchestrated database clusters. In *2018 IEEE/ACM 13th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 33–39, 2018. 8

[Fle22] Flexera. Flexera 2022 state of the cloud report, 2022. [Online] https://resources.flexera.com/web/pdf/Flexera-State-of-the-Cloud-Report-2022.pdf?elqTrackId=f3bb660986704d2980404386aa003141&elqaid=6925&elqat=2. Último acesso a 07 de Outubro de 2023. 1, 3

[Fou23] Apache Software Foundatio. Jmeter, 2023. [Online] https://jmeter.apache.org. Último acesso a 07 de Outubro de 2023. 26

[Fre20] João Emanuel Leitão Freire. Orquestração de containers usando kubernetes e docker swarm. 2020. [Online] http://hdl.handle.net/10400.6/11091. Último acesso a 07 de Outubro de 2023. xiii, 5, 6, 11

[Goo23a] Google. Algorithm details, 2023. [Online] https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/#algorithm-details. Último acesso a 07 de Outubro de 2023. 26

[Goo23b] Google. Docker overview, 2023. [Online] https://docs.docker.com/get-started/overview/. Último acesso a 07 de Outubro de 2023. xiii, 6, 7

[Goo23c] Google. Google kubernetes engine (gke), 2023. [Online] https://cloud.google.com/kubernetes-engine. Último acesso a 07 de Outubro de 2023. 11

[Goo23d] Google. Production-grade container orchestration, 2023. [Online] https://kubernetes.io/. Último acesso a 07 de Outubro de 2023. vii, ix

[GSC+22] Alessio Giorgetti, Davide Scano, Javad Chamanara, Mustafa Albado, Edgard Marx, Sean Ahearne, Andrea Sgambelluri, Francesco Paolucci, and Filippo Cugini. Kubernetes orchestration in sdn-based edge network infrastructure. In *2022 Optical Fiber Communications Conference and Exhibition (OFC)*, pages 1–3, 2022. 11

[IMAJ19] Filippo Bosi Luca Foschini Giuseppe Martuscelli Rebecca Montanari Amedeo Palopoli Isam Mashhour Al Jawarneh, Paolo Bellavista. Container orchestration engines: A thorough functional and performance comparison. *CC 2019 - 2019 IEEE International Conference on Communications (ICC)*, 2019. xiii, 7, 8

[KAGP21] Zhuangwei Kang, Kyoungho An, Aniruddha Gokhale, and Paul Pazandak. A comprehensive performance evaluation of different kubernetes cni plugins for edge-based and containerized publish/subscribe applications. In *2021 IEEE International Conference on Cloud Engineering (IC2E)*, pages 31–42, 2021. 11

[MHR21] Suchanat Mangkhangcharoen, Jason Haga, and Prapaporn Rattanatamrong. Migrating deep learning data and applications among kubernetes edge nodes. In *2021 IEEE 23rd Int Conf on High Performance Computing & Communications; 7th Int Conf on Data Science & Systems; 19th Int Conf on Smart City; 7th Int Conf on Dependability in Sensor, Cloud & Big Data Systems Application (HPCC/DSS/SmartCity/DependSys)*, pages 2004–2010, 2021. 11

[Mic23] Microsoft. Azure kubernetes service (aks), 2023. [Online] https://azure.microsoft.com/en-us/products/kubernetes-service/. Último acesso a 07 de Outubro de 2023. 11

[Ora23] Oracle. Oracle virtualbox, 2023. [Online] https://www.virtualbox.org. Último acesso a 07 de Outubro de 2023. 15

[Pan22] Nikola Pantic. Kubernetes architecture diagram: The complete explanation, 2022. [Online] https://www.clickittech.com/devops/kubernetes-architecture-diagram/. Último acesso a 07 de Outubro de 2023. xiii, 10

[PGKM20] Amit Potdar, Narayan D. G., Shivaraj Kengond, and Mohammed Moin Mulla. Performance evaluation of docker container and virtual machine. *Procedia Computer Science*, 171, 2020. 6

[Ran22] Rancher. What is rancher?, 2022. [Online] https://ranchermanager.docs.rancher.com. Último acesso a 07 de Outubro de 2023. 11

[SD19] Jay Shah and Dushyant Dubaria. Building modern clouds: Using docker, kubernetes & google cloud platform. *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 0184–0189, 2019. 8

[SGM21] Stefano Sebastio, Rahul Ghosh, and Tridib Mukherjee. An availability analysis approach for deployment configurations of containers. *IEEE Transactions on Services Computing*, 2021. 5

[TKVL+20] Eddy Truyen, Nane Kratzke, Dimitri Van Landuyt, Bert Lagaisse, and Wouter Joosen. Managing feature compatibility in kubernetes: Vendor comparison and analysis. *IEEE Access*, 8:228420–228439, 2020. 11

[TSDT18] Jadran Torbić, Ivan Stanković, Borislav Dorđević, and Valentina Timenko. Hyper-v and esxi hypervisors comparison in windows server 12 virtual environment. *2018 17th International Symposium INFOTEH-JAHORINA (INFOTEH)*, pages 1–5, 2018. 4

[TVK22] Minh-Ngoc Tran, Dinh-Dai Vu, and Younghan Kim. A survey of autoscaling in kubernetes. In *2022 Thirteenth International Conference on Ubiquitous and Future Networks (ICUFN)*, pages 263–265, 2022. 8, 9

[VMW23a] VMWare. Container orchestration, 2023. [Online] `https://www.vmware.com/topics/glossary/content/container-orchestration.html`. Último acesso a 07 de Outubro de 2023. 7

[VMw23b] VMware. What is a hypervisor?, 2023. [Online] `https://www.vmware.com/topics/glossary/content/hypervisor.html`. Último acesso a 07 de Outubro de 2023. 4

[VMw23c] VMware. What is a virtual machine?, 2023. [Online] `https://www.vmware.com/topics/glossary/content/virtual-machine.html`. Último acesso a 07 de Outubro de 2023. 4, 5

[ZG22] Hui Zhu and Christian Gehrmann. Kub-sec, an automatic kubernetes cluster apparmor profile generation engine. In *2022 14th International Conference on COMmunication Systems & NETworkS (COMSNETS)*, pages 129–137, 2022. 11

[ZKU+22] Johannes Zerwas, Patrick Krämer, Răzvan-Mihai Ursu, Navidreza Asadi, Phil Rodgers, Leon Wong, and Wolfgang Kellerer. KapetÁnios: Automated kubernetes adaptation through a digital twin. In *2022 13th International Conference on Network of the Future (NoF)*, pages 1–3, 2022. 8