UNIVERSIDADE D
COIMBRA

Filipe Gomes Arrais

UM OPERADOR FEDERADO PARA *KUBERNETES*

Dissertação no âmbito do Mestrado em Engenharia Informática, especialização em engenharia de software, orientada pelo Professor Doutor Filipe Araújo e o Dr. Miguel Guerreiro apresentada ao Departamento de Engenharia Informática da Faculdade de Ciências e Tecnologia da Universidade de Coimbra.

Setembro de 2023

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D
COIMBRA

DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

Filipe Gomes Arrais

Um operador federado para *Kubernetes*

Dissertação no âmbito do Mestrado em Engenharia Informática, especialização em engenharia de software, orientada pelo Professor Doutor Filipe Araújo e o Dr. Miguel Guerreiro apresentada ao Departamento de Engenharia Informática da Faculdade de Ciências e Tecnologia da Universidade de Coimbra.

Setembro de 2023

Agradecimentos

Independentemente dos obstáculos e contratempos que possam surgir, a presença de pessoas que nos apoiam torna a jornada mais simples e com mais significado.

Começo por reconhecer e gratificar toda ajuda e acompanhamento que obtive pela parte do meu Orientador, Professor Doutor Filipe Araújo, sem o qual não teria sido capaz de executar esta dissertação. Agradeço todo o tempo, empenho e conhecimentos comigo partilhados, levando desta experiência uma enorme admiração, tanto pela pessoa, como pelo profissional, pelo modo como se dedicou integralmente a este projeto.

Uma palavra muito especial ao Dr. Miguel Guerreiro, meu Coorientador, por toda a sapiência com que me brindou, guiando-me sempre por entre os momentos de maior dificuldade.

A Coimbra, à sua Universidade, à Faculdade de Ciências e Tecnologia e, em particular, a todos os Professores com quem tive o gosto de cruzar vida, o meu sincero obrigado.

Um especial obrigado à minha mãe e irmão por todo o apoio incansável, por todos os conselhos e por fazerem de mim a pessoa que sou hoje. A eles estou eternamente grato por todos os sacrifícios e esforços que permitiram que eu tivesse a oportunidade de ter acesso ao ensino superior. O mérito da conquista não é unicamente meu, mas nosso.

À minha Gabriela, a minha namorada, agradeço por todo o seu apoio incondicional, sempre presente nos bons e maus momentos, nas derrotas e nas vitórias, encorajando-me nos tempos difíceis e celebrando as conquistas. Sem dúvida um pilar ao longo deste percurso.

Não posso deixar passar os meus amigos, a família que esta cidade me deu, porque sem eles Coimbra não tinha tido o mesmo encanto. Agradeço-lhes todo o companheirismo, apoio, paciência e amizade com que caminharam ao meu lado ao longo destes últimos cinco anos. Obrigado por terem partilhado este caminho comigo.

Saliento a importância da participação do Platform Engineering Group da Web-Summit, sem o qual não teria sido viável o desenvolvimento deste projeto. A vós, o meu reconhecimento e gratificação pela disponibilização de um *cluster* e pela permissão para a criação de outro na conta da Amazon Web Services (AWS) fornecida.

Este trabalho é financiado pelo projeto POWER (bolsa POCI-01-0247-FEDER-070365), cofinanciado pelo Fundo Europeu de Desenvolvimento Regional (FEDER), através do Portugal 2020 (PT2020), e pelo Programa Operacional Competitividade e Internacionalização (COMPETE 2020).

POWER

empowering a digital future

Co-financed by:

COMPETE 2020 PORTUGAL 2020 UNIÃO EUROPEIA Fundo Europeu de Desenvolvimento Regional

Partners:

altice labs · U C · Universidade de Coimbra · it instituto de telecomunicações · IPN Instituto Pedro Nunes

Abstract

Multi-cluster architectures are becoming increasingly popular due to the many benefits they offer. Despite that, they also pose several significant challenges, especially when it comes to efficient cluster management. However, Kubernetes lacks the notion of a federated cluster, relying solely on metrics such as RAM and CPU to make cluster management decisions, limiting its ability to manage multi-cluster environments efficiently. One way to overcome this problem is to have well-defined service level objectives (SLOs) that can support decisions regarding the multi-cluster environment, as these provide a clear vision of how the application should behave, facilitating the cluster management process. The tools available on the market for operating multi-cluster configurations offer easy implementation and management but do not provide adequate support for meeting the SLOs defined for the applications deployed by them.

This dissertation addresses the challenges of managing a multi-cluster environment, focusing on maintaining SLOs. The aim is to develop a federated operator, which brings observability to the system by visualizing the SLOs and enabling it to act according to a set of rules in the event of violations to remedy these violations, thus bringing reliability to the system. With this, we hope to contribute to a system that minimizes violations of SLOs, which is crucial for organizations and clients to provide a consistent service that meets the client's expectations. It is also hoped to reduce the need for human intervention when operating the system.

In this dissertation, we developed a federated operator guided by a cost service level objective (SLO), which proved to be effective in minimizing SLO violations. This operator demonstrated a concrete implementation of a federated operator guided by SLOs, proving to be a valid solution for managing multi-cluster environments and federating clusters within the Kubernetes ecosystem.

Keywords

Kubernetes, Federated Operator, Operator Framework, SLOs, Multi-cluster Architecture, Multi-cloud Architecture

Resumo

As arquiteturas *multi-cluster* estão cada vez mais a ser usadas devido aos diversos benefícios que fornecem, caminhando, ainda assim, a par com diversos desafios significativos, principalmente quando se trata de uma gestão eficiente dos *clusters*. No entanto, *Kubernetes* carece da noção de *cluster* federado, baseando-se apenas em métricas como *RAM* e *CPU* para a tomada das suas decisões na gestão dos *clusters*, o que limita a sua capacidade para gerir ambientes *multi-clusters* de forma eficiente. Uma forma de colmatar este problema passa por ter objetivos do nível de serviço (SLOs) bem definidos, que possam apoiar as decisões relativas ao ambiente *muti-cluster*, pois estes fornecem uma visão clara de como a aplicação se deve comportar, facilitando o processo de gestão dos *clusters*. As ferramentas disponíveis no mercado para operar configurações *multi-cluster* oferecem uma implementação e gestão fácil destes, não facultando, contudo, um suporte adequado para o cumprimento dos SLOs definidos para as aplicações *deployed* pelos mesmos.

Posto isto, esta dissertação aborda os desafios de gerir um ambiente *multi-cluster*, com foco na manutenção de SLOs. Pretende-se então desenvolver um operador federado, que permite trazer observabilidade ao sistema, ao visualizar os SLOs e possibilitando que, em caso de violação destes, aja segundo um conjunto de regras, de forma a colmatar essas mesmas violações trazendo, consequentemente, confiabilidade ao sistema. Com isto, espera-se contribuir com um sistema que minimize as violações de SLOs que são cruciais para as organizações e clientes, de modo a prestar um serviço consistente e que vá de encontro às expectativas do cliente. Também se espera reduzir a necessidade de intervenção humana ao operar o sistema.

Nesta dissertação, desenvolveu-se um operador federado orientado por um objetivo do nível de serviço (SLO) de custo, que demonstrou eficácia na minimização das violações deste. Este operador demonstrou uma implementação concreta de um operador federado orientado por SLOs, mostrando-se uma solução válida para a gestão de ambientes *multi-clusters* e para a federação de *clusters* dentro do ecossistema do *Kubernetes*.

Palavras-Chave

Kubernetes, Operador federado, *Operator Framework*, *SLOs*, Arquitetura *Multi-cluster*, Arquitetura *Multi-cloud*

Conteúdo

Acrónimos

AWS Amazon Web Services.

CISUC Centro de Informática e Sistemas da Universidade de Coimbra.

CR Custom Resource.

CRD Custom Resource Definition.

CRs Custom Resources.

EKS Elastic Kubernetes Service.

FCTUC Faculdade de Ciências e Tecnologia da Universidade de Coimbra.

GKE Google Kubernetes Engine.

OLM Operator Lifecycle Manager.

SLAs acordos do nível de serviço.

SLO objetivo do nível de serviço.

SLOs objetivos do nível de serviço.

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

O presente documento insere-se no âmbito da dissertação de Mestrado em Engenharia Informática com especialização em Engenharia de *Software* pela Faculdade de Ciências e Tecnologia da Universidade de Coimbra (FCTUC), no ano curricular de 2022/2023. O trabalho decorreu no Centro de Informática e Sistemas da Universidade de Coimbra (CISUC) no Grupo de Engenharia de *Software* e Sistemas no Departamento de Engenharia Informática da Universidade de Coimbra.

1.1 Contexto

Olhando para o passado, conseguimos observar uma constante evolução no desenvolvimento e arquitetura do *software*, sendo que, com o evoluir dos anos, diversas coisas mudaram. O espelho dessas mudanças são, nomeadamente, as arquiteturas de micro-serviços, a massiva adoção de *containers* e as aplicações *cloud-native* que, cada vez mais, são tendências crescentes na área do *software*.

Todos estes fatores contribuíram para que *Kubernetes*, o orquestrador centrado em *containers*, se tornasse, para além de muito popular, o padrão para dar *deploy* e gerir aplicações em *containers*. Isto porque as arquiteturas de micro-serviços são compostas geralmente por vários serviços empacotados em *containers*, o que faz com que grandes aplicações que utilizem esta arquitetura sejam constituídas por imensos *containers*, introduzindo uma grande complexidade na gestão dos mesmos. As aplicações *cloud-native* também contribuíram para esta popularização do *Kubernetes*, pois assentam em três princípios, o uso de arquiteturas de micro-serviços, a utilização de *containers* e o facto de esta ser dinamicamente orquestrada, conduzindo este último princípio à necessidade de utilizar um orquestrador de *containers*. Adicionalmente, existe também uma adoção massiva da tecnologia de containerização entre a comunidade.

Posto isto, importa agora entender que em *Kubernetes* podemos optar por um arquitetura de apenas um *cluster* ou uma arquitetura *multi-cluster*, dependendo dos objetivos que se pretende alcançar para determinado serviço. As arquiteturas *multi-clusters* do *Kubernetes*, como o nome sugere, são aquelas em que o

Kubernetes é implementado em mais do que um *cluster*. De salientar que estas são particularmente úteis para cenários em que é necessário ter uma alta disponibilidade, pois possuem a capacidade de a aumentar, mesmo em cenários de falha, como por exemplo a interrupção do provedor de *cloud*, de estar em conformidade com as exigências de residência de dados dos clientes e também para tirar partido das capacidades específicas dos provedores de *cloud*, evitando o "vendor lock-in", conceito apresentado mais adiante. Devido a estes benefícios, as empresas muitas vezes optam por este tipo arquiteturas, já que estas lhes permitem prestar um melhor e mais confiável serviço aos seus clientes.

1.2 Problema e Motivação

Apesar das vantagens anteriormente elencadas, as configurações *multi-cluster* trazem consigo uma complexidade adicional quando se trata de orquestrar serviços executados sobre estes, especialmente num ambiente *multi-cloud*. Esta complexidade deve-se à diferença entre os diversos provedores de *cloud*. Além disso, gerir manualmente um ambiente *multi-cluster*, tal como monitorizar o sistema e fazer escolhas com base em dados de infra-estrutura, é difícil, dispendioso e torna impossível uma resposta imediata na eventualidade do surgimento de problemas.

Um dos desafios do *Kubernetes* prende-se com o facto de este se basear apenas em métricas e limites de *hardware*, como *CPU* e *RAM*, para acionar eventos de escalonamento para cima/baixo ou para agendar um determinado serviço entre os nós do *cluster*. No entanto, carece de outros fatores importantes, como congestionamento de rede, topologia da rede *multi-cluster*, degradação de serviços externos do provedor *cloud*, entre outros. A ausência de consideração destes fatores evidencia, neste cenário, a falta de noção de um *cluster* federado quando necessita de tomar decisões para aumentar o desempenho, confiabilidade e a disponibilidade dos serviços. Uma maneira de o contornar passa por ter objetivos do nível de serviço (SLOs) bem definidos, que poderão ajudar as equipas de engenharia nos processos operacionais e, para além disso, podem também ser usados para desencadear ações de auto-remediação, especialmente a um nível federado, onde as condições do *cluster* podem variar muito devido a *hardware* heterogéneo.

Os SLOs são métricas que definem um certo nível do serviço que se espera que o sistema forneça aos seus utilizadores, com vista em medir a qualidade do serviço que está a ser fornecido. Ao defini-los e observá-los, as organizações podem garantir que estes atendem às necessidades dos seus utilizadores. Como estes fornecem garantias de que a qualidade de determinado serviço corresponde ao esperado, respeitar SLOs é crucial para as organizações, de modo a não haver discrepâncias entre a expectativa do cliente e a sua real experiência ao usar o serviço e, para além disso, fornecendo, adicionalmente, uma maneira quantificável de medir a performance e qualidade do mesmo.

Para implementar plenamente os *clusters* federados, é necessário garantir que as aplicações adiram aos SLOs. Para que isto aconteça, um *control plane*, componente que faz as decisões globais sobre o *cluster*, deve ser capaz de fornecer tais capacidades, a fim de reduzir a exigência de intervenção manual e aumentar a

fiabilidade dos sistemas *multi-cluster*. Além disso, tendo em conta o contexto de arquiteturas multi-*cluster* e *multi-cloud*, seria benéfico utilizar os vários *clusters* de forma federada para garantir o cumprimento dos SLOs. Assim, caso um *cluster* por algum motivo fosse incapaz de cumprir os SLOs, seria possível usar o conceito de federação para utilizar outro *cluster* capaz de os cumprir.

Para lidar com configurações *multi-cluster*, existem tecnologias, como *Kubefed* ou *Cluster API*, que estão presentemente disponíveis graças às atuais ferramentas de *go-to-market* e à investigação. Estas *APIs* controlam *deployments* federados e servem como *API Gateways* para cada membro do *cluster*, sendo projetadas para facilitar a implementação e gestão de vários *clusters*. Adicionalmente, permitem que seja possível dar *deploy* das aplicações de forma consistente e gerir os *clusters* de forma centralizada, no entanto, até ao momento estas são incapazes de se adaptar às condições do *cluster* e de suportar sistemas de controlo inteligentes com a capacidade de fornecer ações de auto-cura, aptas para responder a mudanças e prever violações de SLOs. Como estas apenas se focam mais na gestão de configurações *multi-clusters* e não oferecem um suporte adequado para SLOs , é difícil para os fornecedores de determinado serviço controlar e respeitar os SLOs acordados. Utilizar métricas como a latência torna-se mais adequado para manter um SLO de, por exemplo, tempo de resposta. Além disso, a conjugação de métricas de alto nível com mecanismos que permitem a deteção de violações de SLOs, trazem uma maneira mais eficaz de respeitar os mesmos, pois permitem desencadear ações que impeçam a sua violação.

No contexto deste problema em específico, por exemplo, um controlador dentro do operador federado pode iniciar as instâncias de nós *Kubernetes* menos dispendiosas num dos *clusters* disponíveis, se for provável que uma aplicação ultrapasse o seu tempo de resposta SLO definido. Portanto, o operador gere os *clusters* de acordo com os SLOs definidos para a aplicação, adaptando-o de maneira a que a aplicação esteja sempre dentro dos parâmetros de performance e qualidade definidos de forma consistente e, assim, melhorar a confiabilidade do serviço. A observação de SLOs permite melhorar a observabilidade, ao fornecer um melhor entendimento de como os serviços do sistema se estão a comportar, sendo mais fácil identificar o problema que os causa e, consequentemente, a decisão a tomar.

Um operador federado é um tipo de software que faz a gestão e coordenação dos vários *clusters* de *Kubernetes*, num ambiente *multi-cluster*. Este pode ajudar a gerir a carga de trabalho entre os vários *clusters*, de modo a respeitar os SLOs, a dar *deploy* das diversas instâncias da aplicação e a fornecer uma visão unificada sobre todos os seus *clusters* e recursos. Desta forma, o operador federado consegue evitar a necessidade de intervenção humana, ao automatizar uma resposta quando este deteta violações dos SLOs, minimizando-as de forma imediata. O facto de um Operador Federado fazer parte do ecossistema *Kubernetes* facilita aos programadores o acrescento de novas funcionalidades, devido à facilidade com que pode ser integrado com a *API Kubernetes*, que é utilizada por todos os componentes num ambiente *multi-cluster*.

1.3 Objetivo

O objetivo final desta dissertação é a implementação de um operador federado extensível, para controlar configurações *multi-cluster* que são orientadas por um SLO. Pretende-se provar este conceito através da evidência do comportamento federado com a implementação de um operador que seja capaz de gerir um ambiente *multi-cluster* e *multi-cloud* de acordo com um SLO. Neste caso específico, o operador deve olhar para um SLO de custo definido para a aplicação a ser *deployed*, isto é, um limite de *budget* que a aplicação pode custar por mês. Com isto, tenciona-se otimizar o custo da aplicação nos *clusters*, mantendo-o dentro do nível definido. Portanto, o operador deve observar o custo de forma a tomar um conjunto de decisões para o otimizar, sempre respeitando o valor definido pelo SLO. Se a aplicação em determinado *cluster* atingir o SLO, o operador deve tomar uma decisão federada, como por exemplo migrar a carga da aplicação para outro *cluster* que respeite o SLO de custo definido. Deste modo, o operador federado fornece uma maneira de gerir e operar o ambiente *multi-cluster* e *multi-cloud* como um só sistema, para assim tirar partido de certos benefícios como os melhores preços das *clouds*, permitindo, desta forma, a redução do *vendor lock-in* e o respeito pelo SLO definido para a aplicação.

Assim, é preciso montar uma estrutura para recolher e armazenar as métricas necessárias e desenvolver o operador que utilize essas mesmas métricas para gerir e tomar decisões em relação aos vários *clusters* de acordo com SLO de custo.

1.4 Resultados

Dado por concluído o estágio, os resultados que produzi foram os seguidamente elencados:

- Criação de *clusters Kubernetes*, subsequente implementação do operador federado e *deployment* do mesmo num ambiente *multi-cluster* e *multi-cloud*, que demonstrou o conceito de operador federado orientado por SLOs;
- Uma minimização efetiva de violações do SLO de custo, tendo o operador demonstrado a capacidade de o manter dentro do limite de custo acordado;
- O operador federado permite uma resposta imediata quando deteta violações do SLO, ao tomar as medidas necessárias para resolvê-las, sem a necessidade de intervenção humana;
- O operador federado faz parte do ecossistema *Kubernetes* e, desta forma, contribui para uma solução federada para a plataforma que carece desta noção;

Em suma, a dissertação resultou num sistema que atende aos requisitos estabelecidos. O operador federado implementado é uma prova de conceito concreta

do conceito de operador federado orientado a SLOs. Este está projetado para monitorizar um SLO de custo específico, isto é, um limite de *budget* mensal, por exemplo de 100 euros por mês, definido para uma aplicação a ser *deployed* pelo mesmo que, no caso, é um servidor *Nginx*. Ao observar o SLO, o operador irá detetar eventuais violações e irá proceder às medidas necessárias para as minimizar, no próximo mês, ao migrar a aplicação para outro *cluster* que consiga cumprir mensalmente o SLO definido. Desta forma, está claramente evidenciado o comportamento federado do operador, pois as suas várias instâncias presentes em cada *cluster* trabalham em conjunto, como uma só, para gerir configurações *multi-cluster* e *multi-cloud* orientadas pelo um SLO de custo.

1.5 Estrutura do Documento

O documento encontra-se dividido nas seguintes secções:

- O **capítulo 2** é focado em toda a pesquisa realizada sobre o tema, contendo tópicos de grande importância para a realização desta dissertação. Também é feita uma análise sobre o funcionamento das tecnologias a usar no trabalho;
- O **capítulo 3** debruça-se nas funcionalidades que se pretendem para o sistema, ao fornecer uma descrição dos requisitos funcionais e atributos de qualidade do mesmo;
- O **capítulo 4** contém a arquitetura do sistema, apresentada com recurso ao modelo C4;
- O **Capítulo 5** descreve as ferramentas e processos utilizados no desenvolvimento do sistema proposto, além de fornecer uma descrição detalhada de como este funciona e foi implementado;
- O **Capítulo 6** descreve os testes realizados ao sistema e uma análise dos resultados;
- O **Capítulo 7** apresenta o planeamento do primeiro e segundo semestre através de diagramas de *Gantt* e as condições necessárias para o sucesso da dissertação. É também realizada uma análise de como o plano de trabalhos divergiu do planeado
- Por último, o **Capítulo 8** é uma análise pós-estágio, onde são tiradas ilações sobre o trabalho desenvolvido, dificuldades e algumas considerações finais.

Capítulo 2

Conceitos e Estado da Arte

Neste capítulo é descrita toda a pesquisa efetuada sobre os conceitos e tecnologias relacionadas com o projeto. É feita uma passagem pelos conceitos de micro-serviços, *containers*, orquestração de *containers*, aplicações *cloud-native*, *multi-cloud* e operadores, estabelecendo sempre uma relação entre os mesmos. São também analisadas as tecnologias relevantes e que serão usadas nesta dissertação, sendo estas o *Docker*, o *Kubernetes*, *Helm*, *Kubecost*, *Prometehus* e a *Operator Framework*.

2.1 Micro-serviços

Uma aplicação de micro-serviços é um conjunto de pequenos e independentes serviços (Figura 2.1), que correm cada processo da aplicação como um serviço separado e comunicam entre si, na maior parte das vezes, através de recursos *http* expostos por *APIs*. Pelo contrário, uma aplicação monolítica é vista como um único serviço (Figura 2.1), isto é, todas as funções e processos são agrupados na mesma aplicação. Neste último caso corre como um todo, o que traz uma grande desvantagem em relação a uma arquitetura de micro-serviços em que, se um serviço falhar, o resto da aplicação não fica comprometida (os restantes serviços continuarão a funcionar), ao contrário do que acontece no caso de uma aplicação monolítica pois, em caso de ocorrer alguma falha, toda esta ficará indisponível. A arquitetura de micro-serviços permite, desta forma, reduzir o risco de uma aplicação ficar totalmente indisponível.

Numa arquitetura micro-serviços, acrescentar uma nova funcionalidade passa muitas vezes por criar um novo serviço e implementar nesse mesmo serviço toda a lógica necessária para a concretização da mesma, enquanto que, numa arquitetura monolítica, uma nova funcionalidade é adicionada ao código que compõe toda a aplicação, fazendo com que este se torne muito grande e complexo. A arquitetura de micro-serviços é suportada por *containers*, pois cada serviço pode ser *deployed* sem interferir com os outros. Estes fornecem um ambiente adequado para o *deployment* de serviços em termos de rapidez, isolamento e facilidade com que se dá *deployment* às suas novas versões [1].

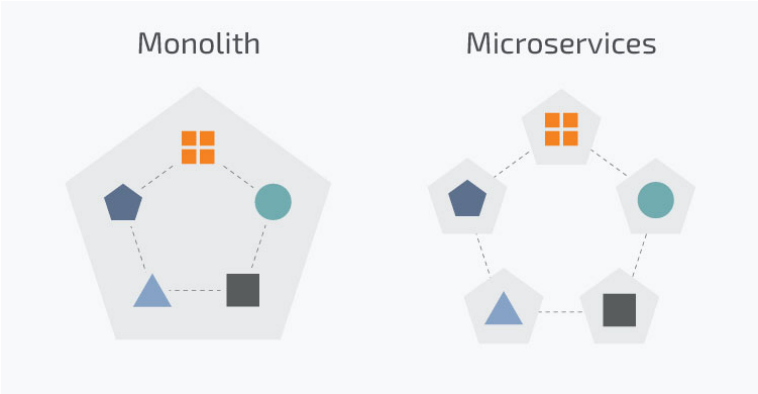


Figura 2.1: Micro-serviços *versus* Monolítico [2]

2.2 *Containers*

Houve uma evolução na forma como o *deployment* de aplicações é feito, sendo dividida em 3 eras, a tradicional, a virtualizada e a era dos *containers* [3]. Na era tradicional de *deployment*, as aplicações corriam em servidores físicos, não existindo, desta forma, maneira de definir limites nos recursos para as aplicações, o que originava problemas de alocação dos mesmos. Tomemos como exemplo duas aplicações a correr no mesmo servidor. Neste caso, pode haver momentos em que uma pode afetar o desempenho da outra por se encontrar a utilizar a maior parte dos recursos.

Uma solução para este problema seria correr as aplicações em servidores diferentes, o que, para além muito dispendioso, faria também com que as aplicações não usassem os recursos todos, surgindo a era de virtualização como resposta a estes problemas. Esta permitia correr várias máquinas virtuais dentro do mesmo servidor, o que proporcionava uma melhor gestão e uso dos recursos. As aplicações correm isoladas umas das outras, não existe partilha de informação entre as aplicações, pois cada máquina virtual executa os seus próprios componentes, incluindo o seu sistema operativo, que se encontra a correr em cima do *hardware* virtualizado.

Por fim, surgiu a era dos *containers*, que são similares às máquinas virtuais, mas com um nível de isolação mais baixo, uma vez que partilham o mesmo sistema operativo sendo, assim, considerados mais leves. Além disso, ao contrário das máquinas virtuais, os *containers* virtualizam sistemas operativos em vez de *hardware*, o que os torna mais portáveis, escaláveis e eficientes. Isto porque, esta partilha do mesmo sistema operativo por parte dos *containers*, oferece uma vantagem em relação às máquinas virtuais visto que, desta forma, há um melhor aproveitamento dos recursos de *hardware*, devido ao facto de não ser necessário correr os diversos sistemas operativos das máquinas virtuais (Figura 2.2). Assim, podem-se correr mais *containers* num servidor físico do que em máquinas virtuais.

Um *container* é uma unidade padrão de *software* que agrupa código juntamente com os ficheiros, as bibliotecas e as dependências necessárias para a aplicação funcionar corretamente, de forma a isolar a mesma para que esta corra em qual-

quer ambiente de forma confiável [4]. Uma das principais vantagens da containerização prende-se por esta separar um programa das suas dependências, numa entidade portável e autónoma que pode correr em qualquer lado [5]. O facto de ser portável faz com que seja possível mover a aplicação entre vários serviços *cloud*, por exemplo, sem que seja necessário fazer quaisquer alterações ao código, assegurando que a aplicação irá correr como esperado, facilitando assim o seu *deployment*. Para além disto, torna o desenvolvimento mais rápido, uma vez que evita que os programadores se tenham de preocupar com dependências, versões e ambientes.

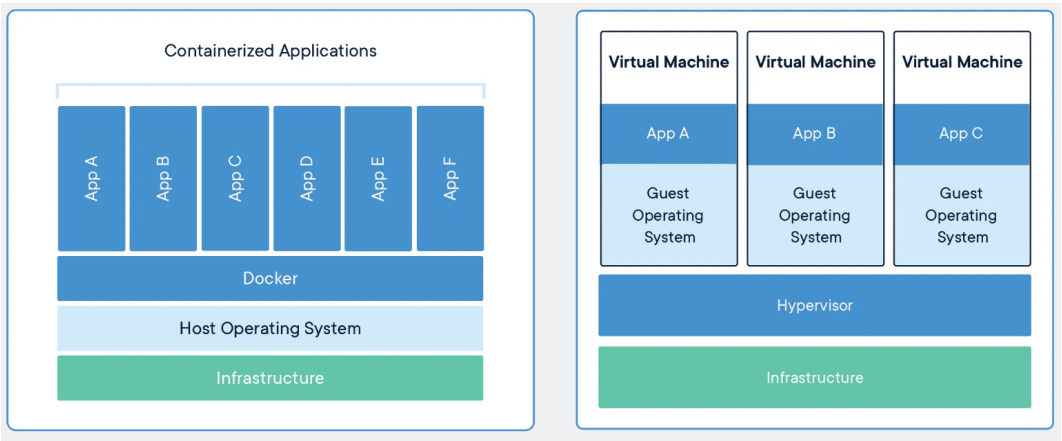


Figura 2.2: *Containers versus* Máquina Virtual [6]

2.3 Orquestração de *containers*

Os *containers* são muitas vezes utilizados em arquiteturas micro-serviços, sendo que grandes aplicações irão ser compostas por centenas ou até milhares de *containers*. Ora, isto introduz uma grande complexidade, já que gerir manualmente esta grande quantidade de *containers* é um grande desafio e é aí que entra o orquestrador. Este faz com seja possível controlar esta gestão de *containers*, pois proporciona uma forma declarativa de automatizar grande parte do trabalho [7].

Os orquestradores são ferramentas que permitem automatizar a gestão, o *deployment*, o escalonamento para cima/baixo e a comunicação dos *containers* através de descoberta de serviços. O uso de orquestradores vem ajudar, de uma forma automatizada, a manutenção das aplicações, a responder de forma dinâmica a mudanças, a permitir que, quando um *container* falhe, este seja substituído automaticamente e a que as atualizações e novas configurações dos *containers* seja feita automaticamente em todos *containers* [8].

A orquestração traz diversos benefícios às organizações que trabalham com *containers* ao simplificarem operações, uma vez que estes podem introduzir uma grande quantidade de complexidade o que, sem um orquestrador para os gerir, facilmente pode sair do controlo. As ferramentas de orquestração podem começar, recomeçar ou escalar um *container*. Além disso, como substituem uma pessoa ao fazer as diversas tarefas de forma automatizada, reduzem a possibilidade de

erro humano, criando assim uma segurança extra [7].

De acordo com Asif Khan [5], uma plataforma eficaz de orquestração de *containers* é aquela que possibilita agilidade no desenvolvimento, ao fornecer um conjunto de funcionalidades apropriadas para a aplicação, infraestrutura e ferramentas de equipa. Além disso, este argumenta também que devem permitir uma gestão e agendamento fácil do estado do *cluster* e fornecer mecanismos para alcançar uma disponibilidade alta e tolerância a falhas, redes eficientes em escala e elevada segurança para garantir a integridade dos serviços.

2.4 Aplicações *Cloud-native*

Uma aplicação *cloud-native* é concebida para satisfazer, entre outras, as exigências *cloud* de escalonamento automático, auto-cura, atualizações contínuas e reversões [9]. Para além disto, importa destacar que as tecnologias de virtualização de *containers* servem de base para estas aplicações *cloud-native*, que são uma abordagem para construir e correr aplicações que tomem partido dos benefícios de usar a *cloud*.

Segundo a Cloud Native Computing Foundation (CNCF), *cloud-native* é descrita como usar uma *stack* de *software* de código aberto com as seguintes características: A aplicação é empacotada em *containers*, dinamicamente orquestrada e orientada a micro-serviços [4]. As arquiteturas de micro-serviços são as mais comuns usadas [10], tornando-se, por isso, mais flexíveis, visto que podem ser facilmente modificadas e atualizadas sem afetar o sistema inteiro. Adicionalmente, a utilização de *containers* traz-lhes portabilidade, de modo a permitir que estas corram em todo o lado, ao mesmo tempo que tornam o *deployment* mais fácil. Para alcançar uma gestão dinâmica é utilizado um orquestrador de *containers*, por exemplo *Kubernetes*.

Para concluir, Fedor Y. Chemashkin *et al* [11] define, então, desta forma, as aplicações *cloud native* como utilização de *containers*, baseadas em micro-serviços e dinamicamente geridas, indo de acordo ao anteriormente referido.

2.5 *Multi-cloud*

Multi-cloud é uma solução informática de utilização de dois ou mais provedores de *cloud*, com o objetivo de obter certos benefícios, nomeadamente uma otimização de custos em infraestruturas, alcançar uma maior flexibilidade e portabilidade reduzindo, desta forma, o *vendor lock-in*.

O *vendor lock-in* traduz-se por uma situação onde um cliente se encontra preso a um provedor *cloud* porque o custo, duração e complexidade de mudar para um provedor diferente são muito altos. Isto traz algumas preocupações como o serviço ser afetado pela qualidade do provedor *cloud*, levando a que certos objetivos do nível de serviço (SLOs) não sejam cumpridos e também a que o provedor

possa aumentar os preços dos seus serviços, por saber que os clientes estão presos ao mesmo.

Além disto, *Multi-cloud* traz a vantagem de ser possível tirar partido dos melhores preços e recursos disponíveis entre os vários provedores, bem como alocar estrategicamente cargas de trabalho, de forma a obter os melhores custos. O *downtime* das aplicações é mínimo, pois se houver uma interrupção do provedor de *cloud*, os outros fornecedores podem prestar o serviço pretendido na mesma.

Portanto, quando se fala em orquestração de *containers* em contexto *multi-cloud*, estamos a referir-nos ao processo de utilizar uma ferramenta de orquestração para gerir *containers* em vários ambientes *cloud*, em oposição a apenas um [7]. Soluções multi-cloud são, muitas vezes, construídas em cima de tecnologias de código aberto como, por exemplo, o *Kubernetes*, que é suportado por todos os provedores de *cloud* públicos.

2.6 *Docker*

O *Docker* [6] é uma plataforma de código aberto para desenvolver, distribuir e correr aplicações. Este permite uma entrega rápida de *software*, visto que separa as aplicações da infraestrutura por baixo das mesmas [12]. Tal possibilita aos desenvolvedores criar um ambiente consistente para as aplicações, o que significa que estas podem correr em qualquer lado, eliminando, assim, o problema de apenas a aplicação correr no computador do programador, coisa que muitas vezes acontece, principalmente com aplicações grandes e complexas que têm muitas dependências. Isto é possível porque o *Docker* utiliza *containers* que, com as vantagens acima referidas, oferecem uma maneira rápida de criar e correr aplicações.

Este disponibiliza também uma partilha fácil das aplicações com outros desenvolvedores. Os *Docker containers* podem ser partilhados através de um *resgitry*, um sistema de armazenamento e distribuição para imagens do *Docker*. Este permite controlar as versões das imagens em repositórios. O *Docker Hub* [13] é um exemplo de um repositório público para imagens *Docker*, que faculta aos utilizadores a possibilidade de guardar, partilhar e descarregar imagens em repositórios públicos ou privados. Assim, os desenvolvedores podem partilhar ou colaborar o seu trabalho com outros, o que facilita a contribuição por parte dos mesmos.

As imagens do *Docker* são *templates* que contém todo o código, bibliotecas e dependências necessárias para correr a aplicação e são utilizadas para criar os *containers*. É possível criar de raiz ou utilizar imagens já publicadas em repositórios por outros. Para criar uma imagem, necessitamos de um *Dockerfile*, que é um ficheiro que contém, para além de instruções para a sua criação, instruções para a correr [14].

Docker daemon é o componente responsável por construir, correr e gerir imagens e *containers* através do uso de comandos do cliente. Este comunica com o *Docker daemon* através de uma *API Rest* que, por sua vez, utiliza o *Resgistry* para puxar ou empurrar imagens, onde por predefinição o *Docker* está configurado de forma

a procurar imagens no *Docker Hub*, mas é possível ser configurado para utilizar outro [12]. Na Figura 2.3 pode observar-se este funcionamento e como os diversos componentes interagem entre si.

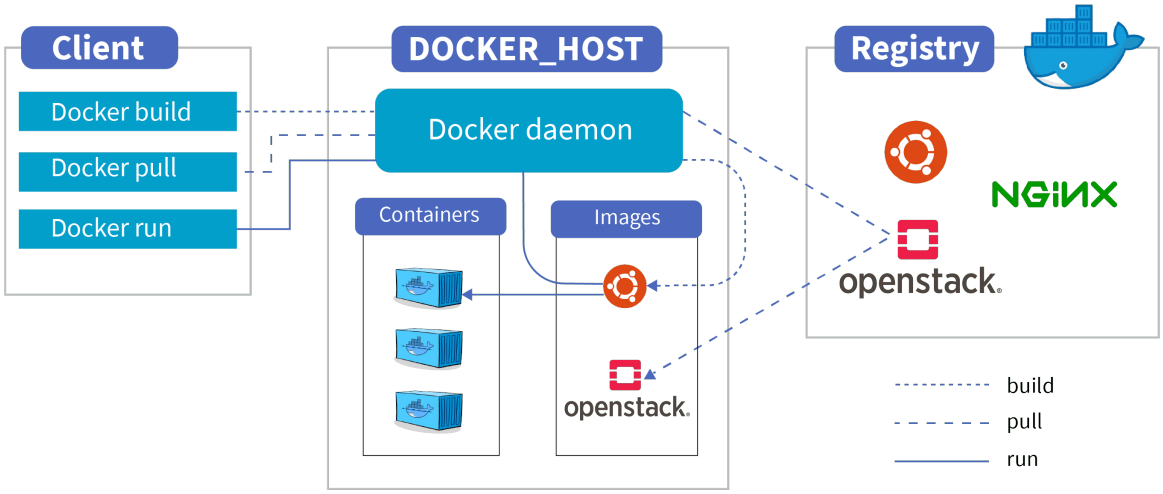


Figura 2.3: Arquitetura do *Docker* [12]

2.7 *Kubernetes*

Kubernetes, também conhecido como K8s devido a existirem 8 caracteres entre a letra K e S, é descrito de acordo com o seu *website* oficial como "um sistema de código aberto utilizado para automatizar o *deployment*, o escalonamento e a gestão de aplicações em *containers*" mas, antes de se mergulhar nos detalhes do mesmo, importa, primeiramente, abordar a sua história e a forma como surgiu.

A empresa Google foi uma das empresas que desde cedo entendeu os benefícios de usar os *containers* anteriormente referidos, começando a desenvolver as suas aplicações em *containers* [15]. Mais tarde, apercebeu-se da necessidade de desenvolver uma aplicação que permitisse, tanto a sua gestão, como das máquinas que os corriam, um orquestrador. Este projeto chama-se Borg e foi a inspiração para, mais tarde, a criação de *Kubernetes*, à medida que o interesse em *containers* crescia. Nasceu então o *Kubernetes*, que foi um projeto de código aberto criado pela Google, escrito em linguagem *Go* e, mais tarde, doado à fundação Cloud Native Computing (CNCF), apresentando, desde aí, um grande crescimento [16].

Kubernetes fornece um conjunto de funcionalidades para orquestrar *containers* [3]:

- **Lançamentos e reversões automatizadas:** *Kubernetes* lança progressivamente atualizações e alterações de configuração de aplicações, monitorizando constantemente a saúde da aplicação de forma a certificar-se que não mata todas

as suas instâncias. Isto permite evitar qualquer tipo de *downtime* e, em caso de alguma coisa correr mal, o próprio *Kubernetes* reverte automaticamente as alterações.

- **Descoberta de serviços e *load balancing*:** O *Kubernetes* evita que seja necessário modificar a aplicação para usar um mecanismo de descoberta de serviços, pois atribui aos seus *containers* um *IP* e um único *DNS* ao conjunto de *containers*. Assim, é possível fazer o *load balancing* dos mesmos.
- **Orquestração de armazenamento:** *Kubernetes* monta automaticamente soluções de armazenamento, quer seja a partir de armazenamento local, de fornecedores externos de *cloud* ou de um armazenamento em rede.
- **Auto-cura:** Este possui mecanismos de auto-cura, ou seja, sempre que um nó falha, os *containers* desse nó são substituídos e reagendados noutro. Também, no caso de os *containers* não corresponderem ao estado de saúde pretendido pelas políticas definidas, estes são terminados e não são anunciados ao cliente até serem reiniciados e estarem prontos a serem usados.
- **Gestão e configuração de *secrets*:** A gestão e atualização de dados sensíveis e da configuração da aplicação é feita separadamente da imagem da mesma, evitando assim reconstruí-la. Os *secrets* são informação confidencial, transmitida à aplicação sem revelar conteúdo sensível à configuração da *stack*.
- **Embalagem automática de *containers*:** *Kubernetes* coloca automaticamente *containers* com base nas suas necessidades e restrições de recursos, de modo a maximizar a utilização sem sacrificar a disponibilidade.
- ***Batch execution*:** *Kubernetes* suporta *batch execution* e permite substituir *containers* em falha.
- **Escalonamento horizontal:** É possível fazer escalonamento para cima/baixo da aplicação manualmente, através de comandos, ou automaticamente, com base em utilização de *CPU*.
- ***IPv4/IPv6 dual-stack*:** *Kubernetes* suporta alocação de endereços *IPv4* e *IPv6*.
- **Concebido para ser extensível:** *Kubernetes* permite a adição de novas funcionalidades, sem que seja necessário modificar o código fonte.

Agora que se compreende a necessidade de um orquestrador de *containers*, a forma como *Kubernetes* surge para resolver essa necessidade e o que este oferece, estamos preparados para mergulhar nos conceitos e no funcionamento deste projeto.

2.7.1 Arquitetura e Conceitos *Kubernetes*

Em relação à sua arquitetura, *Kubernetes* é composto por diversos componentes, como se pode verificar na Figura 2.4, onde cada um é responsável por uma determinada tarefa, a fim da execução e gestão de aplicações em *containers*. Para

compreender o papel de cada componente, irá ser passada uma visão geral sobre cada um.

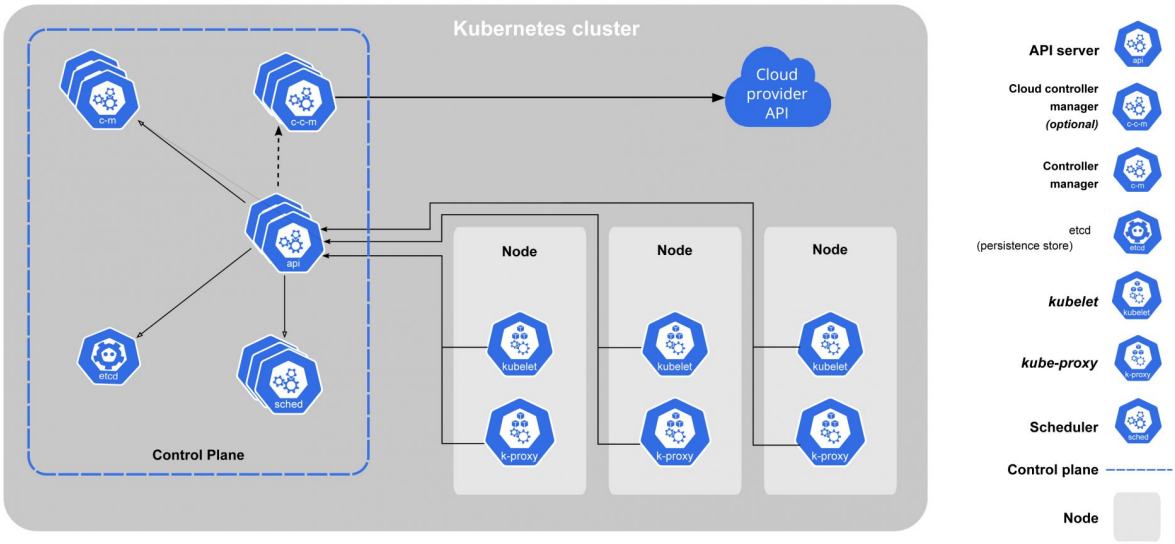


Figura 2.4: Arquitetura *Kubernetes* [3]

Um *cluster* de *Kubernetes* é constituído por, pelos menos, um nó mestre que é responsável por manter o seu estado e por vários nós de trabalho que correm as aplicações em *containers*. O nó mestre, não só controla o estado do *cluster* através dos componentes do *control plane* que são responsáveis por tomar todas as decisões globais relativas ao mesmo como, por exemplo, o agendamento, como também é responsável por detetar e responder a eventos do *cluster* [3]. Esses componentes são *kube-apiserver*, *etcd*, *kube-scheduler*, *kube-controller-manager* e *cloud-controller-manager*.

O *Kube-apiserver* é o componente encarregue por expor a *API* do *Kubernetes*, passando por ela todas as comunicações entre os diversos componentes. Uma instrução para o *Kubernetes* é dada através de um pedido *HTTP* para o *kube-apiserver*. A *API* do kuberntes é uma *API Restful*, que consome ficheiros *json* e *yaml* que descrevem o estado desejado do *cluster*. Todos os pedidos realizados à *API* estão sujeitos a autenticação e autorização e, só após isto, os ficheiros *yaml* ou *json* são validados, armazenados e o trabalho é agendado para o *cluster* [9].

Ora, é necessário que o *kube-apiserver* armazene este estado do *cluster* em algum lado e é aí que entra o *ectd*. O *ectd* é uma base de dados consistente e altamente disponível de dados chave-valor que guarda o estado do *cluster*, bem com as configurações do mesmo. Trata-se de um componente sensível para o *cluster*, pois se ocorrer uma perda dos dados que o *ectd* guarda, o *cluste*r não iria conseguir recuperar o seu estado, tornando-se inutilizável e, por esta razão, devemos ter um plano de recuperação [9][3].

Segue-se o *kube-scheduler*, o componente do *control plane* responsável por observar o servidor da *API* para novas tarefas e designá-las aos nós de trabalho apropriados. Tem apenas a responsabilidade de escolher os nós para correr as novas tarefas que ainda não têm um nó atribuído. Estas novas tarefas são geralmente

pods, que representam um conjunto de *containers* [3]. Mais à frente no capítulo irá ser apresentado o conceito de *pod*, ficando, para já, apenas a ideia geral.

O *kube-controller-manager* é responsável por executar os processos dos controladores [3]. Supervisiona e garante que o *cluster* está no estado desejado através dos diversos controladores que o compõem, sendo que cada um deles é um processo separado responsável por determinado componente ou tarefa do *cluster*. Este pressuposto de agrupar os diversos controladores vai contra a filosofia da arquitetura de micro-serviços, visto que cada controlador tem a sua responsabilidade, sendo que estes serviços deveriam ser separados, mas em ordem, para reduzir a complexidade eles são compilados num único binário e executados num único processo [3]. *Deployment controller*, *ReplicaSet controller*, *Node controller* e *Service-Account controller*, são alguns dos controladores que compõem o *kube-controller-manager*. Mais à frente no decorrer do capítulo, a questão do que é exatamente um controlador, como desempenha as suas funções e o seu objetivo será mais detalhada.

Por fim, o último componente do *control plane* é o *cloud-controller-manager*, cujo objetivo é facilitar a integração com os provedores de *cloud* e os seus serviços como, por exemplo, *load-balancers* e armazenamento. Para além disso, separa os componentes que interagem só com o *cluster*, dos que interagem com a plataforma de *cloud* [9]. À semelhança do *kube-controller-manager*, também compila os diversos controladores para um único binário e corre-os com um processo único.

Relativamente aos nós de trabalho, estes têm a responsabilidade de correr os *containers* e a carga de trabalho de acordo com as instruções que recebem do *control plane*. Os nós de trabalho são compostos por três componentes principais, o *kubelet*, o *kube-proxy* e um *container runtime*. O *kubelet* é o principal agente do *Kubernetes* e corre em cada nó do *cluster*, certificando-se que cada *container* está a correr num *Pod*. Comunica com o servidor da *API* para novas tarefas e, cada vez que encontra uma, tenta executá-la. Se não conseguir, simplesmente reporta ao *control plane* e é este que é responsável por tomar as decisões acerca do que fazer. Contrariamente, se conseguir executá-la, mantém uma comunicação com o *control plane* [9]. O *kube-proxy* é um componente que também corre em cada nó e é responsável por manter as regras da rede. Estas regras são o que permite a comunicação com os *Pods* quer dentro ou fora do *cluster*. Por fim, é necessário um *container runtime* que seja responsável por correr, parar e puxar imagens dos *containers* como, por exemplo, *Docker*, *CRI*, entre outros.

Em *Kubernetes* não se trabalha diretamente com os *containers*, em vez disso interage-se com *Pods*. Para criar, apagar ou atualizar um *container* é através destes, as unidades mais pequenas do *Kubernetes* que se podem criar e gerir. São um conjunto de um ou mais *containers* que partilham recursos de armazenamento, de rede e uma especificação de como correr os *containers* [3]. Portanto, fornecem assim um ambiente de execução partilhado [9], isto é, todos os elementos necessários para que uma determinada aplicação corra. *Containers* dentro do mesmo *Pod* partilham todos o *IP* do *Pod*, tendo todos, desta forma, o mesmo endereço. São também a unidade de escalonamento do *Kubernetes*, pois se quisermos escalar uma aplicação é através da adição e remoção de *Pods*.

Como cada *Pod* tem o seu *IP*, isto implica que, quando um *Pod* morre e é criado outro, este nasce com um *IP* diferente do anterior ou, até mesmo quando se quer escalar para baixo, estar-se a apagar *Pods* que poderão estar a ser utilizados por clientes, trazendo isto desafios da perspetiva de rede. Posto isto, como *Pods* podem ser criados a qualquer altura ou morrer, utilizar o endereço *IP* dos *Pods* não é viável. Como solução, podem utilizar-se Serviços do *Kubernetes* que permitem expor um ou um conjunto de *Pods*, de forma a possibilitar que estes sejam acedidos internamente e externamente no *cluster*, através da atribuição de um *IP* e de um nome *DNS* estável, que nunca muda. Um *Pod* pertence a um serviço através de *labels*. Se estas forem iguais às definidas no serviço, este irá reencaminhar o tráfego para esse *Pod*. Existem três tipos de serviços, o *ClusterIp*, o *NodePort* e o *LoadBalancer*, sendo que o primeiro é apenas acessível dentro do *cluster*, em contrapartida aos outros dois que permitem comunicação fora do mesmo.

Os *Pods* não oferecem qualquer tipo de mecanismo de auto-cura, de escalonamento, nem permitem atualizações contínuas e reversões de forma fácil [9]. Em caso de ocorrência de uma falha de um *Pod*, este é perdido e, por estas razões, utilizamos *Deployments*. Estes fornecem atualizações de forma declarativa para os *Pods* e *ReplicaSets* [3]. Descrevemos o estado desejado num *Deployment* e o *Deployment Controller* é responsável por executá-lo e trazer o estado atual para o estado desejado. O *ReplicaSet* está incumbido por manter um número de réplicas de *Pods* em execução [3]. Na verdade, é o *ReplicaSet* que é responsável pela auto-cura e o escalonamento, pois é este que está encarregue de garantir a disponibilidade de um número específico de *Pods*. O *Deployment* faz a gestão dos *ReplicaSet* e das atualizações contínuas e reversões.

DaemonSet é outro objeto cujo propósito é garantir que todos, ou alguns nós, caso especificado dessa forma, correm uma cópia de um *Pod*. À medida que os nós são adicionados, automaticamente vai ser criado e adicionado um *Pod* aos mesmos. Similarmente, se um nó é removido, a réplica do *Pod* é terminada. O benefício de usar *DaemonSets* é que este certifica-se que os *Pods* estão sempre a correr em cada nó do *cluster*, o que é especialmente importante para tarefas como recolha e monitorização de *logs* e gestão do *cluster*. Existem vários casos de uso para *DaemonSets* mas, normalmente, existe a necessidade de os usar para monitorização do nó no *cluster*, recolha de *logs* de nós individuais e, às vezes, de *Pods* nesses mesmos nós e para gerir o armazenamento do *cluster*.

Kubernetes Namespaces fornecem um mecanismo para isolar grupos de recursos num único *cluster* [3], permitindo assim dividi-lo em partes mais pequenas, tornar mais fácil a sua organização e oferecer um isolamento dos recursos de forma a que os demais recursos de outros *namespaces* não sejam afetados quando se correm comandos *kubectl*. Os nomes dos recursos têm de ser únicos dentro de um *namespace*, mas podem repetir-se entre os vários *namespaces*. O *Kubectl* é a ferramenta de linha de comandos que nos permite comunicar com o *control plane* através da *API* do *Kubernetes*.

Kubernetes foi desenhado de uma forma a possibilitar ser altamente configurável e extensível, de modo a minimizar as alterações ao código fonte principal. Existem vários pontos extensíveis no *Kubernetes* mas, neste contexto, apenas nos interessa a extensão do servidor da *API* através de *Custom Resources* e a adição de contro-

ladores personalizados para o controlo dos mesmos. Um recurso em *Kubernetes* trata-se de um *endpoint* na *API* do *Kubernetes*, na qual esta contém vários recursos. Um *custom resource* refere-se à adição de um *endpoint* à *API* padrão do *Kubernetes*, com o qual os utilizadores podem interagir como se fosse um recurso normal. Ao combinarmos os dois fornecem uma *API* declarativa, que nos declara o estado desejado para o recurso. O controlador personalizado é responsável por manter o estado atual do objeto *Kubernetes* em correspondência com o estado desejado [11].

Custom Resource Definition (CRD) é o que se usa para definir um Custom Resource (CR), sendo semelhante a um esquema que contém a definição e o tipo dos parâmetros do CR.

A combinação de Custom Resources (CRs) e de controladores personalizados introduziu um novo conceito, o padrão operador, que nos permite estender o comportamento do *cluster* sem modificar o código do *Kubernetes*.

2.8 Operadores

Um operador de *Kubernetes* é um controlador específico que estende a funcionalidade da *API* do *Kubernetes* para criar, configurar e gerir instâncias de aplicações, em nome de um utilizador do *Kubernetes* [17].

O padrão operador visa capturar o objetivo principal de um operador humano, responsável pela gestão de um serviço ou um conjunto de serviços [18]. O estado do *cluster* pode sofrer alterações, como ser necessário lançar uma nova versão da aplicação, adicionar uma nova funcionalidade, *Pods* a falharem e, consequente, degradação da performance da aplicação, sendo estes alguns dos cenários que necessitam de intervenção humana. Este esforço requer tempo que poderá ser agravado em caso de uma falha inesperada, pois existe uma urgência em recuperar o estado desejado para a aplicação, o que implica um engenheiro disponível a qualquer altura e que tenha um bom conhecimento da aplicação.

A função do padrão operador é capturar as intenções de como uma pessoa humana iria gerir o serviço. A criação de um operador começa muitas vezes com automatizar a instalação de aplicações e autoconfigurações e segue para automatizações mais complexas [17]. A implementação de um operador deve conter o estado desejado da aplicação, devendo este observar continuamente o estado real desta e tomar as decisões necessárias para a manter no estado saudável (estado desejado). Como se pode ver, este funcionamento é análogo ao conceito de controlador em *Kubernetes*, pelo que um operador é essencialmente um controlador com a diferença que usa conhecimento específico da aplicação para gerir os seus recursos.

O operador tem como objetivo reduzir a quantidade de trabalho manual que é necessário para manter a aplicação íntegra e saudável, como atualizações, cópias de segurança, escalonamento, entre outros, conseguindo alcançar isto ao capturar o tal conhecimento específico da aplicação no código e ao expô-lo de forma

declarativa. Este é constituído pelo operando, a aplicação que queremos gerir, pela linguagem de programação que permite ao utilizador, de forma declarativa, especificar o estado desejado para a aplicação e por um controlador que corre continuamente, responsável por ler, manter-se a par do estado da aplicação, reportar esse estado e por realizar de forma automatizada todas as ações necessárias quando o estado muda.

De forma simples, o operador lê o estado atual através do servidor da *API Kubernetes*, procede às modificações necessárias através de operações *CRUD* e atualiza a informação sobre o estado no servidor da *API* num constante *loop*, como se pode observar na Figura 2.5. Este *loop* de reconciliação (*control loop*) presente no operador, garante que o estado que o utilizador declara usando um CR coincide com o estado atual da aplicação.

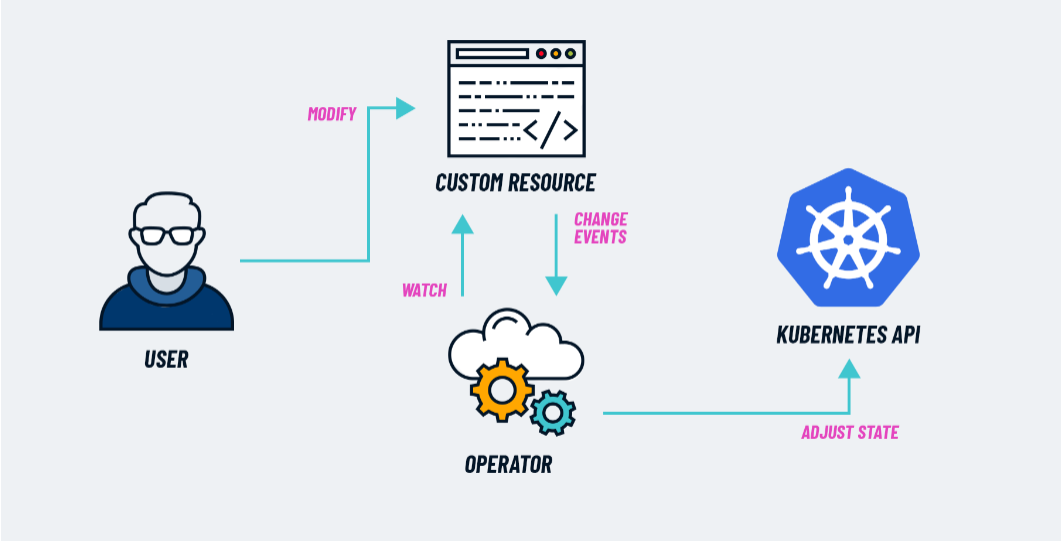


Figura 2.5: *Loop* de reconciliação do operador [19]

Fedor Y. Chemashkin *et al* [11] argumenta que o padrão operador surge da teoria de controlo e da robótica, onde o *control loop* é um *loop* infinito que controla o trabalho do sistema. Também refere o facto de os operadores poderem ser usados para fornecer acordos do nível de serviço (SLAs), por garantir um certo nível de qualidade para operações do sistema sobre certas suposições do ambiente operacional. Os SLAs são o acordo entre a identidade, que fornece o serviço, e os utilizadores, sobre o que se pretende do mesmo, enquanto que SLOs são os objetivos específicos para cumprir esse mesmo acordo.

Os autores ainda realçam a importância de ter um *software* capaz de adaptar o seu comportamento em tempo de execução real sem se interromper a si mesmo, devido ao ambiente *cloud* ser dinâmico e incerto.

Das propriedades acima referidas destaca-se a observabilidade, definida pelos autores como a que mostra se é possível restaurar completamente a informação sobre os estados do sistema ou não. Os operadores podem usar esta propriedade para reunir informações sobre o estado atual do sistema e tomar decisões com base nessa informação.

A teoria do controlo deu origem à ideia de observabilidade que, segundo Octavian Mart *et al* [20], é definida como uma medida de quão efetivamente os estados internos do sistema podem ser deduzidos de *outputs* externos. Trazer visibilidade aos sistemas é o objetivo da observabilidade, que pode ser alcançada com base em métricas e *logs*.

Podem utilizar-se as métricas observadas juntamente com *forecasting*, com o intuito de prever o futuro baseado em dados anteriores, o que pode ser útil para antecipar eventuais desvios daquilo que se pretende para a aplicação [20].

2.8.1 *Operator Framework*

O *Operator Framework* [21] é, de acordo com o seu *website* oficial, um conjunto de ferramentas de código aberto para gerir aplicações nativas *Kubernetes*, os operadores, de uma forma eficaz, automatizada e escalável. Este conjunto de ferramentas e os componentes *Kubernetes*, ajudam no desenvolvimento dos operadores e na gestão central de um *cluster*. A *framework* é composta por 3 componentes principais:

- **O *Operator SDK*:** Fornece um conjunto de bibliotecas e ferramentas de linhas de comando para construir um operador do zero. Este inclui *APIs* predefinidas, funções comuns, geradores de código e ferramentas de estruturação de projeto que permitem inicializar e estruturar projetos, criar *APIs* e controladores padrão, gerar código, construir e dar *deploy* de um operador num *cluster*.

- **O Operator Lifecycle Manager (OLM):** É um componente concebido para instalar, executar e atualizar operadores num *cluster* de *Kubernetes*. Os desenvolvedores do operador escrevem ficheiros que descrevem os meta-dados relevantes do operador de forma a que o OLM possa automatizar o seu *deployment*. Este também funciona como um catálogo dos operadores instalados dentro do *cluster* e pode ser usado para garantir que não existem conflitos entre as *APIs* dos operadores em questão.

- **Operator Hub:** Repositório de operadores que foram testados e certificados pela equipa do *Operator Framework*. Fornece um catálogo de operadores existentes e orientações para contribuir com novos. Os desenvolvedores podem submeter ou procurar operadores em `https://operatorhub.io/`.

Em termos de linguagem, um operador tecnicamente pode ser escrito em qualquer linguagem, desde que esta suporte os clientes e chamadas *API* necessárias para interagir com um *cluster Kubernetes*. Mas, o *Operator SDK* apenas oferece suporte para operadores escritos em *Go* ou construídos com ferramentas como *Helm* ou *Ansible*. A escolha da linguagem deve ser tomada de acordo com as necessidades das funcionalidades do operador a desenvolver, isto porque, *Helm* e *Ansible* apenas permitem a construção de operadores mais limitados, em contrapartida à linguagem *Go* que permite a construção de operadores mais customizados.

Em [22], os autores fornecem um conhecimento profundo do *Operator Framework* e os seus componentes. No livro está presente um guia prático de como desenhar e desenvolver um operador em *Go*, utilizando o *Operator Framework*, como publicá-lo no OperatorHub.io e diferentes maneiras de dar *deploy* como, por exemplo, a utilização do OLM. Além disso, são apresentados casos de estudo baseados em operadores reais.

O *Operator Framework* também define um modelo de maturidade (Figura 2.6) para os operadores de *Kubernetes*, onde estes são categorizados com base em cinco níveis que se baseiam nas funcionalidades do mesmo, através de perguntas quantificáveis. A classificação do operador juntamente com o modelo, permite a utilizadores reunirem informação sobre o que podem esperar de determinado operador.

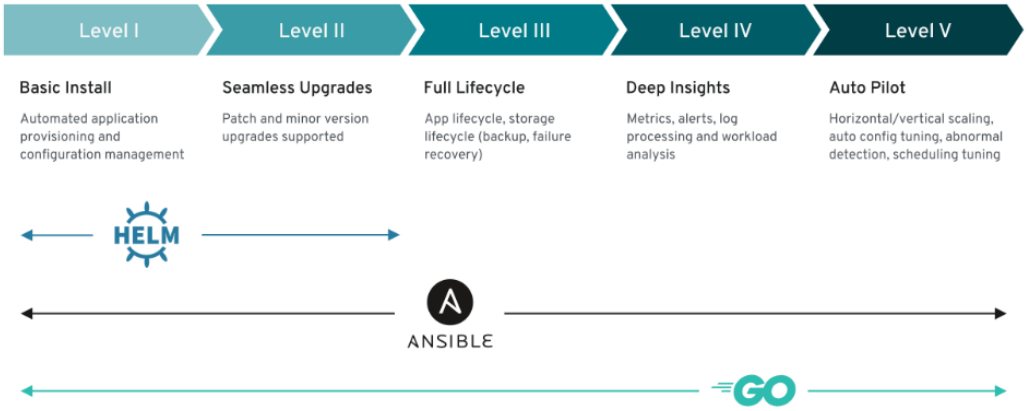


Figura 2.6: Modelo de maturidade do operador [21]

Os 5 níveis são [22]:

- **Nível 1 - Instalação básica:** Operadores que podem instalar um operando e, se necessário, expor as opções de configuração para essa instalação.
- **Nível 2 - Atualizações Contínuas:** Operadores que podem atualizar-se a si próprios e aos seus operandos sem perturbar a função.
- **Nível 3 - Ciclo de vida completo:** Operadores que podem lidar com criação ou restauração de cópias de segurança do operando, recuperação de falhas, opções de configuração mais complexas e escalar o operando automaticamente.
- **Nível 4 - *Insights* profundos:** Operadores que fornecem *insights* sobre o operando que estão a monitorizar. Estes *insights* podem ser métricas sobre eles ou sobre o operando.
- **Nível 5 - Piloto Automático:** Operadores capazes de lidar com tarefas complexas como escalonamento automático, auto-cura, afinação automática, ou deteção de anormalidade. O escalonamento automático traduz-se na capacidade do operador aumentar ou apagar réplicas do operando, de acordo

com as suas necessidades. A auto-cura é a capacidade do operador recuperar de um cenário de falha automaticamente com base em métricas, alertas ou *logs*. A afinação automática consiste em realocar *Pods* do operando para nós mais adequados. Por fim, a deteção de anormalidade é verificar que o operador se desvia do desempenho normal da aplicação quando esta se encontra num estado saudável.

Os autores em [23] apresentam o modelo de maturidade com maior detalhe, onde pegam num operador para uma aplicação demo "The Visitors Site"que se categoriza no nível 2. Estes implementam alterações no operador para o categorizar nos seguintes níveis do modelo, até chegarem ao nível 5.

2.8.2 Operador *Prometheus*

O *Prometheus* é um sistema de código aberto de monitorização, que é usado de forma comum para monitorizar *clusters* de *Kubernetes*. Este pode retirar métricas de vários componentes, incluído nós, *Pods* e serviços dentro de um *cluster*, e guardá-las numa base de dados temporal (*Time series database*). Desta forma, torna possível acompanhar o desempenho dos *clusters*. Este fornece algumas funcionalidades como [24]:

- Um modelo de dados temporais identificados pelo nome da métrica e pares chave/valor;
- *PromQl*, uma linguagem para consultar métricas;
- Vários tipos de gráficos e *dashboards* para visualização;
- Uma infraestrutura para recolher dados;
- Um protocolo para expor os dados e as consultas através da rede.

O operador *Prometheus* [25] automatiza a configuração e gestão da *stack* de monitorização do *Prometheus* que corre num *cluster Kubernetes*. Este utiliza *custom resources* para gerir e dar *deploy* do mesmo e dos seus componentes. Também pode ser configurado para enviar alertas com base em determinadas condições como, por exemplo, quando um serviço se torna indisponível ou quando um *Pod* não está a responder como esperado através do seu componente *AlertManager*.

As métricas são fundamentais para entender o porquê de determinada aplicação estar a correr de certa forma, ao fornecer informação sobre a mesma e sobre o que está a acontecer no sistema.

2.9 *Helm*

Helm [26] é um gestor de pacotes para *Kubernetes* que automatiza o ciclo de vida das aplicações, permitindo definir, instalar e atualizá-las de uma maneira simples e consistente.

Dar *deploy* de aplicações manualmente pode ser difícil, principalmente para aplicações complexas, compostas por vários objetos *Kubernetes* (*Deployments*, *Pods*, *Services*, entre outros), isto porque é necessário criar um ficheiro *Yaml* para cada objeto *Kubernetes*, criação esta indispensável para o funcionamento das aplicações. Além disso, é ainda necessário configurar os valores de como se pretende dar *deploy* do *workload* e, em caso de *upgrade* ou regressão de uma atualização, alterar cada objeto um a um. Por este motivo, *Helm* é uma ferramenta que visa simplificar este tipo de problemas através de *Helm charts*, que são um conjunto de manifestos *Yaml* e *templates* que descrevem os vários objetos *Kubernetes* e definem as configurações necessárias para uma aplicação correr num *cluster Kubernetes*. Isto permite ajustar as configurações da aplicação, como portas de serviços, recursos necessários, entre outros, sem ser necessário modificar diretamente os valores *Yaml* dos *charts*. É possível correr várias versões da aplicação com o *Helm*, sendo apenas necessário escolher o *release*. Ressalta-se ainda que os *charts* podem ser criados pelo programador ou obtidos em repositórios do *Helm* como, por exemplo, o *ArtifactHub*.

2.10 *Kubecost*

Apesar dos seus benefícios, a *cloud* traz desafios em relação ao controlo de custos. Infelizmente, medir e reportar os custos relacionados ao ambiente *cloud* é difícil, pois cada uma tem o seu sistema de preços. *Kubecost* é uma ferramenta *open-source* que permite ter visibilidade sobre os gastos do *cluster Kuberntes* e permite monitorizar e gerir custos num ambiente *Kubernetes*. É integrado com a infraestrutura de forma a ajudar as equipas de programadores a rastrear, gerir e reduzir gastos. Desta forma, fornece visibilidade e *insights* de custos em tempo real, de maneira a ajudar as equipas a reduzir custos [27].

O *Kubecost* depende da *stack* do *Prometheus*, nomeadamente do *kube-sate metric*, que fornece métricas relacionada com os objetos *Kubernetes*, do *node exporter* que coleta métricas relacionadas com o uso do nó, do *cAdvisors* para métricas relacionadas com os *containers* e do *coreDNS* para métricas da rede. A ferramenta também permite a previsão de custos futuros com base nas tendências atuais de utilização. O servidor *Kubecost* depende de um *pod cost-analyser* que inclui um *web frontend* que, por sua vez, contém *dashboards* dos dados de um servidor que gere todas as chamadas *API* entre os vários componentes ou o *cluster* e um modelo de custo que fornece cálculos e métricas de alocação de custos.

Das principais *features* do *Kubecost*, destaca-se o facto de este permitir a visualização das despesas de alocação em conceitos nativos do *Kubernetes*, ou seja, permite ver os gastos de forma detalhada, nomeadamente, por *deployments*, *services*, *namespaces* e *labels*, o que permite fornecer custos precisos e de acordo com a fatura real da *cloud* em que está integrado. Além disso, fornece *insights* de otimização personalizados, baseados no ambiente e padrões de comportamento observados e permite ainda a configuração de alertas para violações de *budgets*. Esta ferramenta foi desenvolvida especificamente para *Kubernetes* e ambientes *cloud-native* e sendo um projeto de código aberto, mantém uma forte integração com este

ecossistema [27].

2.11 Correr *Kubernetes Clusters*

Existem várias maneiras de correr *Kubernetes*, entre elas, localmente e na *cloud*, sendo que cada forma tem o seu propósito e casos de uso diferentes. Correr localmente é uma forma que permite desenvolver e testar rapidamente e sem custos a validação das aplicações antes de dar *deploy* para ambientes mais complexos. Para isso existem ferramentas como o *Minikube* [28] e o *Kind* [29], sendo que a primeira permite criar um *cluster Kubernetes* dentro de uma máquina virtual com apenas um nó, o que é útil e rápido para testar cenários de *Kubernetes*, não permitindo, no entanto, testar cenários de vários nós. O *Kind* é uma solução semelhante à anterior com uma grande diferença, já que esta permite criar *clusters* com vários nós. Este utiliza *docker containers* para criar os diversos nós, o que permite a testagem de cenários mais complexos, o que não é possível com *Minikube*. Estas soluções, de forma geral, fornecem uma maneira simples de criar um *cluster* para um ambiente de desenvolvimento e de testes para aplicações, antes destas entrarem num ambiente de produção.

Relativamente a *Kubernetes* na *cloud*, quase todas fornecem um serviço de *Kubernetes*, estando estas encarregues de gerir grande parte da infraestrutura pelo utilizador. Temos como alguns exemplos:

- Amazon Web Services (AWS): Elastic Kubernetes Service (EKS) [30]
- Azure: Azure Kubernetes Service (AKS) [31]
- Linode: Linode Kubernetes Engine (LKE) [32]
- DigitalOcean: DigitalOcean Kubernetes (DOKS) [33]
- Google Cloud Platform: Google Kubernetes Engine (GKE) [34]

Cada fornecedor de *cloud* oferece as suas vantagens, desvantagens e custos, sendo que, tipicamente, soluções *cloud* oferecem soluções escaláveis e geridas com uma sobrecarga operacional reduzida. Estas lidam com provisionamento, dimensionamento e atualizações de *cluster*, permitindo que o foco esteja no desenvolvimento das aplicações.

Capítulo 3

Requisitos

Este capítulo contém toda informação sobre o sistema a desenvolver e os requisitos necessários para alcançar os objetivos da dissertação, nomeadamente os requisitos funcionais e os atributos de qualidade. Estes clarificam e ajudam a ter um melhor entendimento do que se pretende implementar no sistema, podendo ter um grande impacto no desenho da arquitetura. Os requisitos funcionais são listados na Secção 3.1 e os atributos de qualidade na Secção 3.2.

3.1 Requisitos Funcionais

Esta secção é dedicada às funcionalidades necessárias para implementar o operador federado e a estrutura, que permitirá a observação das métricas. Estão nas Tabelas 3.1, 3.2, 3.3, 3.4, 3.5 e 3.6 apresentadas abaixo, os requisitos funcionais de todo o sistema.

REQ-1 Monitorização do custo mensal da aplicação	
Nivel	
Ator	Operador Federado
Objetivo	Recolher os custos de alocação da aplicação no *namespace*
Pré-condição	O *Kubecost* encontra-se devidamente configurado, em execução e com os dados de alocação de custos disponíveis
Pós-condição	O operador federado recebe os dados de custo de alocação corretamente
Cenário Principal	1. O operador efetua uma chamada à *API Allocations* do *kubecost*; 2. O *kubecost* fornece os dados da alocação para determinado *namespace*; 3. O operador trata os dados para extrair o custo total da alocação da aplicação; 4. O operador comunica os dados às várias instâncias dos operadores nos diferentes *clusters* através de *nats*; 5. O operador lê as diversas informações recolhidas dos outros operadores através de *nats*; 6. O operador armazena todos esses dados, mantendo um mapa global de informação de todas as instâncias dos operadores e dos seus *clusters*, respetivamente.

Tabela 3.1: REQ-1 Monitorização do custo mensal da aplicação

REQ-2 Previsão do custo mensal da aplicação	
Nivel	
Ator	Operador Federado
Objetivo	Prever o custo da aplicação mensal baseado no seu *deployment*
Pré-condição	O *Kubecost* encontra-se devidamente configurado, em execução e com os dados necessários à previsão disponíveis
Pós-condição	O operador federado recebe previsão mensal de custo da aplicação corretamente
Cenário Principal	1. O operador efetua uma chamada à *API Spec cost Prediction* do *Kubecost*; 2. O *Kubecost* fornece a previsão; 3. O operador trata os dados para extrair o custo final da aplicação naquele *namespace*; 4. O operador comunica os dados às diversas instâncias dos operadores nos diferentes *clusters* através de *Nats* ; 5. O operador lê as várias informações recolhidas pelos outros operadores através de *nats*; 6. O operador armazena todos esses dados, mantendo um mapa global de informação de todas as instâncias dos operadores e dos seus *clusters*, respetivamente.

Tabela 3.2: REQ-2 Previsão do custo mensal da aplicação

REQ-3 Monitorização do custo mensal em relação ao objetivo do nível de serviço (SLO) definido	
Nivel	
Ator	Operador Federado
Objetivo	Verificar se o SLO definido está a ser cumprido
Pré-condição	Os dados de alocação de custos estão disponíveis para consulta
Pós-condição	Violação detetada, caso exista
Cenário Principal	1. O operador consulta os valores recolhidos dos custos; 2. Compara os dados com o SLO definido; 3. Deteta violação; 4. O operador comunica a violação ao operador líder através de *nats*.

Tabela 3.3: REQ-3 Monitorização do custo mensal em relação ao SLO definido

REQ-4 Eleição do operador líder	
Nivel	
Ator	Operador Federado
Objetivo	Eleger um operador líder
Pré-condição	*Zookeeper* devidamente configurado e em execução
Pós-condição	O operador líder é devidamente determinado
Cenário Principal	1. O operador federado regista um nó efémero e sequencial num diretório do *zookeeper*; 2. Consulta o *zookeeper* para determinar qual o operador com *ID* mais baixo; 3. O operador com *ID* mais baixo é designado líder; 4. Operador líder fica à escuta de eventuais violações de SLO para efetuar as devidas decisões.

Tabela 3.4: REQ-4 Eleição do operador líder

REQ-5 Migração da aplicação	
Nivel	
Ator	Operador líder
Objetivo	Migrar a aplicação para um *cluster* que respeite o SLO
Pré-condição	Violação de SLO detetada
Pós-condição	Aplicação migrada para um *cluster* compatível com o SLO
Cenário Principal	1. O operador líder recebe mensagem de violação do SLO por parte de uma instância do operador; 2. Verifica que *clusters* estão disponíveis e se se encontram dentro dos parâmetros definidos para o SLO para receber a carga; 3. De entre os *clusters* disponíveis escolhe o que a previsão de custos para a aplicação é menor; 4. O líder emite uma ordem para o operador no *cluster* onde o SLO está a ser violado para apagar o *deployment* da aplicação; 5. O líder emite uma ordem para o operador do *cluster* selecionado dar *deploy* da aplicação; 6. A aplicação é migrada e continua a ser executada de acordo com o SLO definido.
Cenário Secundário	2.a. Não existem *clusters* disponíveis; 2.a.1. Mantém a aplicação até haver um *cluster* disponível, que se encontre dentro dos parâmetros definidos para a migração.

Tabela 3.5: REQ-5 Migração da aplicação

REQ-6 *Deploy* e gestão da aplicação	
Nivel	
Ator	Operador Federado
Objetivo	Dar *deploy* e gerir o estado da aplicação
Pré-condição	Recebe instrução para dar *deploy* da aplicação
Pós-condição	Aplicação *deployed* com sucesso
Cenário Principal	1. O operador recebe solicitação de *deploy* da aplicação; 2. Inicia o processo de *deploy* da aplicação no *cluster*; 3. O operador monitoriza e gere o estado da aplicação para garantir que a sua execução é correta e de acordo com o especificado.
Cenário Secundário	2.a. Não consegue dar *deploy* da aplicação; 2.a.1 Criar condição com o estado de erro; 2.a.2 Dar *retriger* do *loop* de reconciliação.

Tabela 3.6: REQ-6 *Deploy* e gestão da aplicação

Com isto, pretende-se, por um lado, que o operador consiga observar o custo real da aplicação no *namespace* em que está *deployed* (Tabela 3.1) e, por outro, que também seja capaz de prever o custo da mesma para o seu *deployment* em outro *cluster* que não contenha o serviço (Tabela 3.2). Tendo em conta isto, o operador deverá ser capaz de observar eventuais violações do SLO definido para a aplicação (Tabela 3.3) e tomar a decisão de migrar a carga para um *cluster* que esteja disponível (Tabela 3.5), ou seja, sem o serviço *deployed* e através da previsão de custos para correr a aplicação num que esteja dentro do limite do SLO. Estas decisões de gestão sobre os *clusters* devem ser tomadas apenas por uma instância do operador federado, de modo a evitar inconstâncias em relação ao estado dos mesmos e, como tal, é necessário proceder à eleição de uma instância líder (Tabela 3.4). Por fim, o operador também deve ser responsável por criar e gerir o *deployment* de acordo com o estado desejado para a aplicação e para o *cluster* em que esta se encontra (Tabela 3.6).

3.2 Atributos de Qualidade

Nesta secção são apresentados os atributos de qualidade, também conhecidos com requisitos não funcionais, bem como uma descrição dos mesmos. Estes serão classificados de acordo com uma escala (E de elevado, M de médio, B de baixo) da forma como vão impactar a arquitetura.

- Portabilidade (M): O operador federado é projetado para cenários *multi-cluster* e *multi-cloud*, logo é necessário que este seja capaz de correr com facilidade nos vários ambientes e plataformas, quer seja *on-premises* ou na *cloud* sem ser necessário modificar o código fonte deste.
- Confiabilidade (E): O operador deve ser confiável e capaz de gerir a aplicação *deployed* por ele de maneira consistente. Este deve lidar adequadamente com falhas da aplicação, mantendo sempre a mesma a correr. Além disso, em caso de falha da instância líder do operador ou falha temporária de obtenção de custos, deve manter o seu funcionamento correto, de forma a garantir que este execute as suas funções essenciais na gestão da aplicação a seu cargo, de acordo com os objetivos do nível de serviço (SLOs) definidos. O sistema também deve permitir à aplicação respeitar ao máximo os SLOs definidos, de modo a que este apresente um serviço com uma qualidade consistente e dentro dos parâmetros determinados.
- Manutenibilidade (M): O sistema deve ser projetado para facilitar a manutenção e atualizações. Ele deve aderir às melhores práticas de desenvolvimento e deve ser elaborado de forma modular para permitir atualizar e ajustar o comportamento do operador sem a necessidade de alterar o código fonte. Isto pode envolver parâmetros como a definição do SLO, réplicas e porto da aplicação

Capítulo 4

Arquitetura

Este capítulo descreve a arquitetura do sistema, que foi desenvolvida de acordo com os requisitos previamente definidos. O modelo C4 foi utilizado para a documentar, resultando isto em vários diagramas.

4.1 Modelo C4

O modelo C4 descreve a arquitetura do *software* em quatro níveis de detalhe, são eles o diagrama de contexto do sistema, o de *container*, o dos componentes e código. Dos quatro níveis referidos, serão apresentados todos à exceção do código, devido a ser necessário ter um grande nível de detalhe numa fase inicial do projeto.

4.1.1 Diagrama de contexto do sistema

Nesta subsecção, é apresentado o diagrama de contexto do sistema, diagrama este com o nível mais elevado, que mostra como o sistema a construir interage com os sistemas externos e utilizadores.

Neste caso, como se pode verificar na Figura 4.1, o sistema interage com dois utilizadores, sendo estes o programador e o cliente. O primeiro é responsável por configurar o sistema de acordo com os objetivos pretendidos (configuração dos *clusters*, *deplpoy* do operador federado e criação dos Custom Resources (CRs)), estando também encarregue por monitorizar o sistema através das interfaces disponibilizadas pelo *Grafana* e *Kubecost*. Já o cliente é quem vai utilizar o serviço fornecido pelo sistema, neste caso a aplicação que será *deployed*.

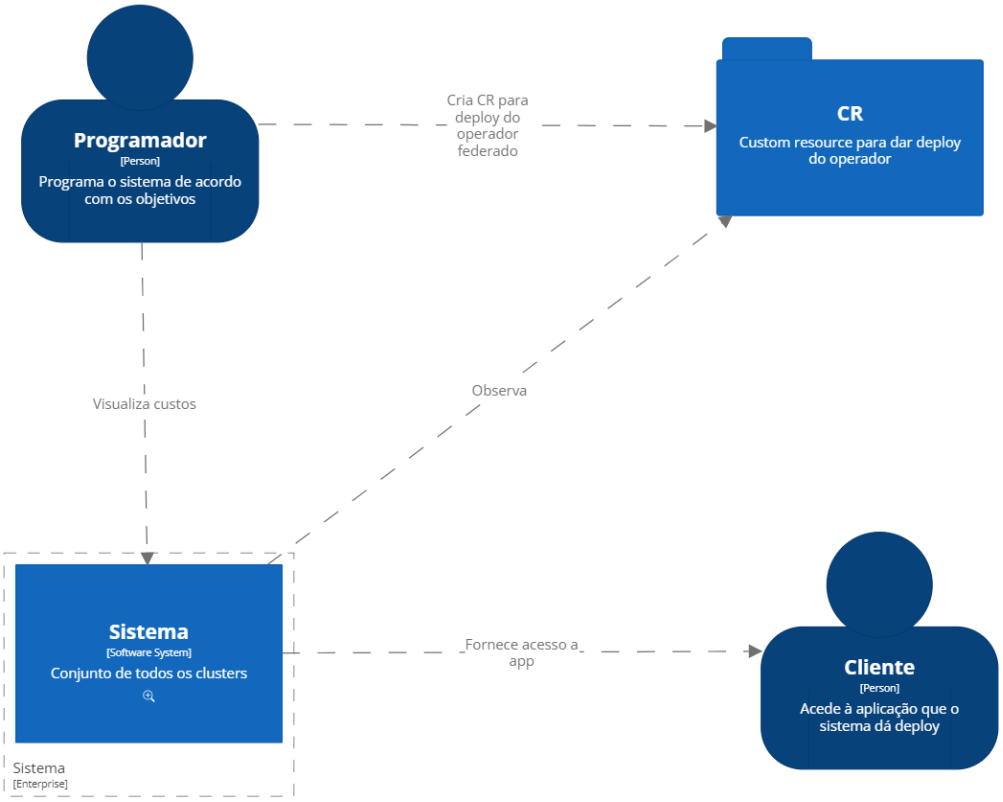


Figura 4.1: Diagrama de contexto

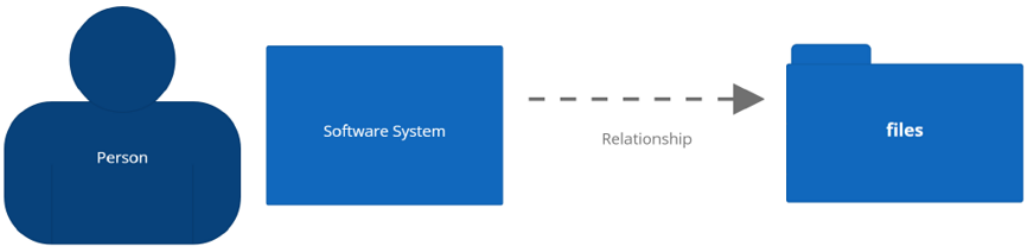


Figura 4.2: Legenda dos elementos do diagrama de contexto

4.1.2 Diagrama de *container*

O próximo diagrama apresentado contém maior detalhe do que o anterior, representando o sistema na forma de *containers*. Como se pode observar, o sistema é composto por um conjunto de *clusters Kubernetes*. Esses *clusters* poderão formar uma arquitetura *multi-cluster* e *multi-cloud*, dependendo das necessidades do negócio.

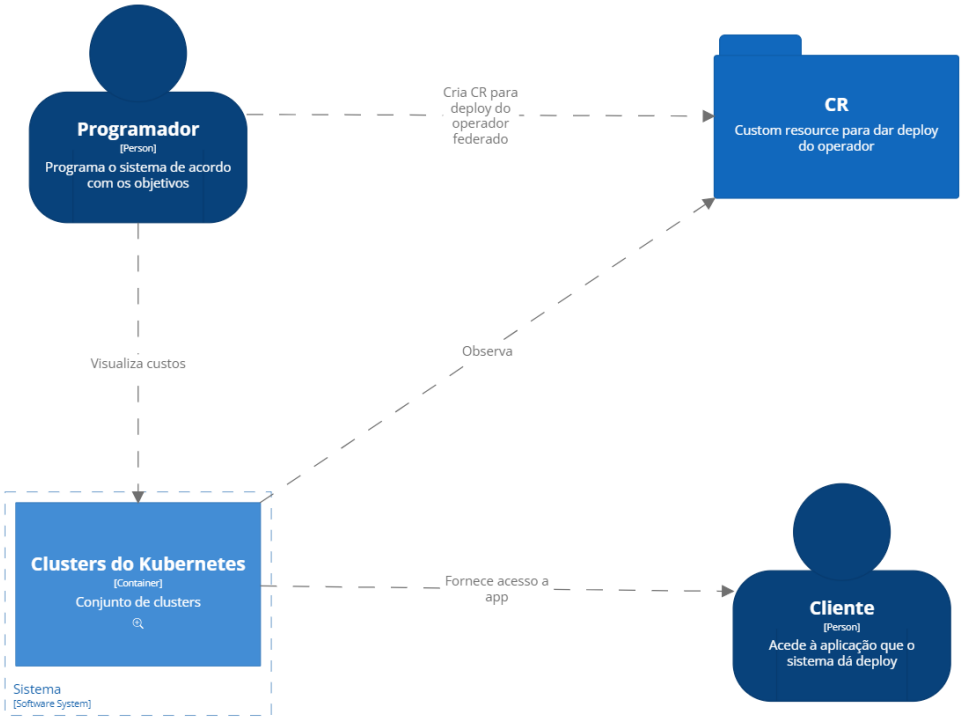


Figura 4.3: Diagrama de *container*

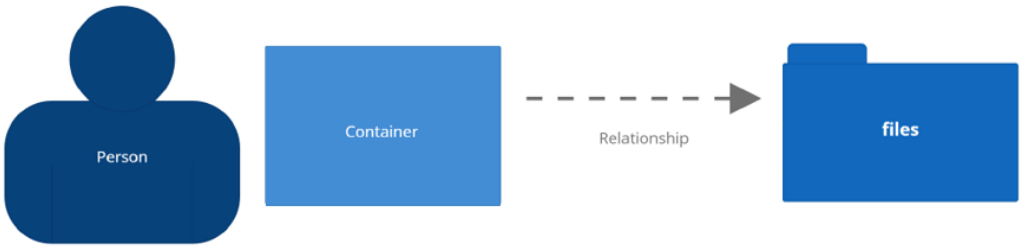


Figura 4.4: Legenda dos elementos do diagrama de *container*

4.1.3 Diagrama dos componentes

Uma vez mais, este diagrama entra em maior detalhe em relação aos anteriores, sendo uma expansão de um *container* individual, neste caso de um *cluster Kubernetes*. Adicionalmente, permite ter uma visão dos componentes dentro do mesmo que, no caso, são dez.

O operador federado é responsável por dar *deploy* e gerir as várias instâncias da aplicação de acordo com o Custom Resource (CR) criado pelo utilizador. Este está também encarregue de agir segundo um conjunto de regras, aquando da observação de uma violação dos SLOs, de forma a respeitá-los. O *Prometheus* guarda as métricas exportadas pelo *node exporter*, que recolhe métricas relativas ao uso de recursos no *cluster* como *RAM*, *CPU*, entre outros, dentro de cada nó do *cluster* e pelo *Kube State Metrics* que recolhe métricas relacionadas com os objetos do

Kubernetes, *Pods*, *deployments*, entre outros. Foi utilizado o *Grafrana* para expor *dashboards* das métricas retiradas por estes componentes e para trazer observabilidade para o sistema, tendo estas sido retiradas uma vez que são necessárias para o funcionamento do *Kubecost*, que as usa para calcular custos relativos ao *cluster*. Desta forma, o *Kubecost* foi configurado para usar as métricas guardadas no *Prometheus*. O *Kubecost* ainda expõe uma interface que permite analisar e observar os custos associados aos recursos de um *cluster Kubernetes*, fornecendo informações detalhadas sobre o seu consumo, a capacidade de alocação e tendências de custos. Adicionalmente, expõe ainda um conjunto de *APIs*, das quais importa salientar a *Spec Cost Predict* e a *Allocation*. A primeira permite prever o custo de determinado *deployment* da aplicação num *namespace* e como irá ser alterado em temos de custo esse *namespace* e a segunda possibilita saber o custo real de determinada aplicação num *namespace*. Assim, o operador federado interage com estas *APIs* para obter informações relativas ao custo, de forma a conseguir medir se o SLO está a ser cumprido e, no caso de não estar, ter forma de prever o custo da aplicação em outros *clusters*, para poder efetuar uma migração de carga mantendo o SLO dentro do nível definido.

Como não podemos ter mais que um operador a tomar as decisões, uma vez que se corre o risco de elas diferirem e, consequentemente, acabar num estado dessincronizado para os vários *clusters*, foi necessário efetuar uma eleição de líder, para que apenas um e um só operador tome as decisões de como gerir o ambiente *multi-cluster*. E é aí que entra o *Zookeeper*, que permite que cada operador crie um nó efémero e sequencial, isto é, um nó que se apaga a partir do momento em que o operador deixa de estar operacional e um nó com uma sequência atribuída. O operador consulta, então, o *Zookeeper*, onde conseguirá saber da existência de todos os operadores, sendo eleito o líder com menor *ID*.

Por fim, *Nats* é o componente utilizado pelos operadores para comunicarem e, de forma federada, gerirem o ambiente *multi-cluster* e *multi-cloud*. De notar que os operadores apenas interagem com os componentes do próprio *cluster*, sendo que o *Nats* é que é responsável por encaminhar as mensagens para os outros *Nats*, em outros *clusters*, para que os operadores consigam ler e enviar mensagens entre eles.

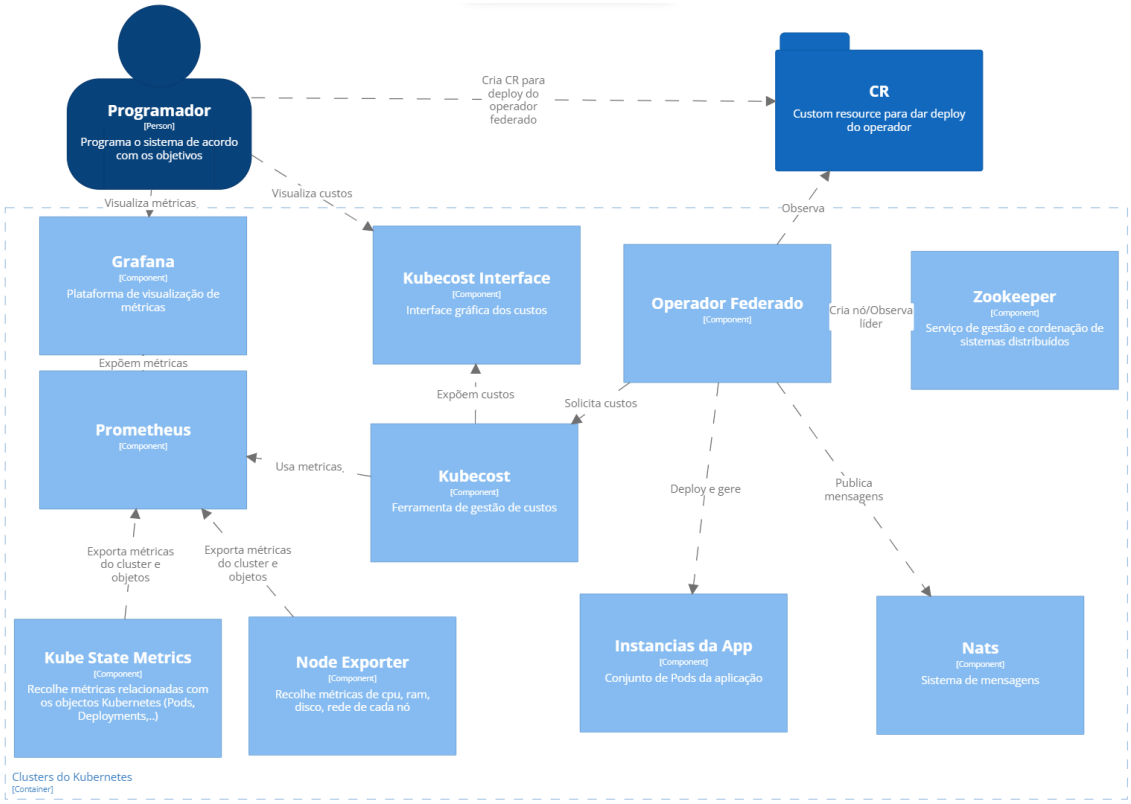


Figura 4.5: Diagrama dos componentes

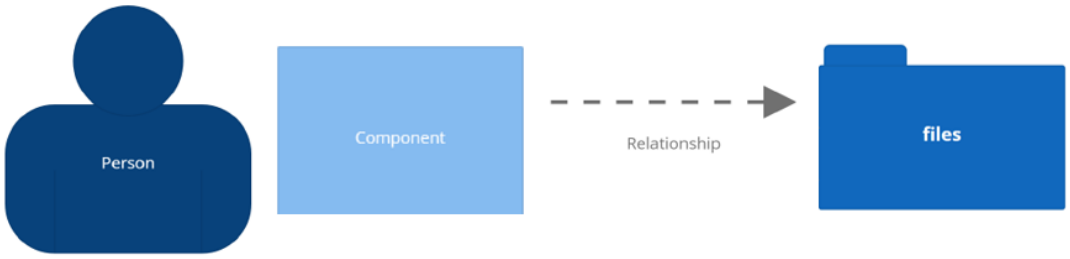


Figura 4.6: Legenda dos elementos do diagrama dos componentes

Capítulo 5

Implementação

Este capítulo descreve a implementação e todos os passos necessários ao desenvolvimento do operador federado para gerir configurações *multi-cluster* e *multi-cloud* orientadas a um objetivo do nível de serviço (SLO) de custo, definido para um servidor *Nginx*.

5.1 Abordagem

Esta secção é dedicada às ferramentas e procedimentos utilizados para desenvolver o sistema proposto. Para esse fim, explica as escolhas relativamente ao editor de código, ao sistema de controlo de versões, à *framework* e à linguagem usadas para a construção do operador e os *clusters* criados e utilizados.

5.1.1 *Framework* e linguagem do operador

A *Operator Framework* [21] foi a *framework* escolhida, oferecendo inúmeras vantagens desde bibliotecas, ferramentas e estruturas que ajudam os programadores a criar operadores eficientes e de acordo com um padrão. Desta forma, permite que os mesmos se concentrem apenas na lógica da aplicação, em vez de ter de lidar com complexidades de baixo nível do *Kubernetes*, como já referido em maior profundidade na Secção 2.8.1 do Capítulo 2

A *framework* apenas oferece suporte para a construção de operadores com ferramentas como *Helm* e *Ansible* ou escritos na linguagem de programação *Go*, sendo que *Go* permite a criação de operadores mais complexos, tendo-se optado, por essa razão, pela escolha desta linguagem para o seu desenvolvimento.

5.1.2 Sistema de Controlo de Versões

De forma a guardar e gerir o código do operador deste projeto, foi utilizado um sistema de controlo de versões, nomeadamente o *Git*, que é o padrão nos tempos

atuais devido ao seu bom desempenho e extensa documentação. Um sistema de controlo de versões, como o próprio nome indica, permite rastrear o histórico do projeto. Assim, cada alteração é organizada em *commits*, que criam uma versão do projeto, o que é particularmente útil para reverter ou restaurar versões anteriores, bem como para manter um seguimento das alterações ao código. Foi criado um repositório dedicado ao projeto na plataforma *Github*, plataforma esta que utiliza o sistema de Controlo de Versões do *Git*.

5.1.3 Editor de código

Durante a realização desta dissertação, o editor de código usado foi o *VS Code*, pois oferece suporte para a linguagem *Go*, com recursos de sintaxe, formatação e sugestões de código inteligentes que tornam o processo de escrita mais fácil e rápido. Além disso, é possível estender o *VS Code*, o que possibilita a personalização do ambiente de desenvolvimento através de *plugins* disponíveis, que permitem aprimorar o suporte para a linguagem *Go*, para *Kubernetes* e também para outras ferramentas essenciais para o projeto, como suporte para ficheiros *Yaml* com suporte integrado de sintaxe do *Kubernetes*, extremamente útil para a criação de objetos do mesmo. Outra característica do *VS Code* é a integração direta com o *GitHub*, que facilita a gestão do repositório de código do projeto.

5.1.4 Aplicação e *Clusters Kubernetes*

Para o desenvolvimento do sistema pretendido, foram utilizados três *clusters*. Um deles foi fornecido pelo grupo *SSE*, tendo sido criado e configurado na Amazon Web Services (AWS) através de *Kops*, uma ferramenta de código aberto que simplifica a criação e gestão de *clusters Kubernetes* em fornecedores de *cloud*. Os restantes foram criação pessoal, um utilizando o Elastic Kubernetes Service (EKS) da AWS e outro o serviço Google Kubernetes Engine (GKE) da Google. O *cluster* EKS insere-se no financiamento do estágio e o *cluster* GKE tem como propósito mostrar o funcionamento do sistema, além de num ambiente *multi-cluster*, num ambiente *multi-cloud*, sendo que a sua escolha se prendeu pela oferta de um período de experimentação gratuito oferecido pela Google, nomeadamente 300 dólares para usufruir durante dois meses o que permitiu, desta forma, criar um *cluster* com as necessidades básicas requeridas para a dissertação. As configurações de cada *cluster* são as seguintes:

- *Cluster* AWS *Kops*: Este *cluster* é composto por seis nós no total, dos quais três são *worker nodes* e os restantes são *master nodes*. Os *worker nodes* são executados em instâncias *t3.medium*[1], enquanto os *master nodes* utilizam instâncias *t3.xlarge*[2]. Os *master nodes* são responsáveis por conter os *control planes*

[1]Máquina virtual que oferece 2 vCPUs, 4 GB de RAM e que é capaz de ajustar os recursos conforme necessário.

[2]Máquina virtual que oferece 4 vCPUs, 16 GB de RAM e que é capaz de ajustar os recursos conforme necessário.

do *cluster*, que são essenciais para a gestão e coordenação das operações relativas ao mesmo. O *cluster* foi criado na região *eu-west* da AWS.

- *Cluster* AWS EKS : Este *cluster* é composto por dois *worker nodes* do tipo *t3.medium*. Os *master nodes* não são criados explicitamente pelo utilizador, pois a AWS encarrega-se da sua gestão, sendo que estes correm numa conta ao cargo da AWS e é apenas exposta a *API* do *Kubernetes* através de um *endpoint* associado ao *cluster*. Este *cluster* foi criado na região *eu-west*, com recurso à ferramenta *eksctl* [35], ferramenta de comandos de linha que facilita a gestão e criação de *clusters* no serviço EKS.
- *Cluster* Google GKE : Criado na região *europe-west1-b* da Google, é constituído por dois *worker nodes* executados em instâncias do tipo *e2-medium*[3]. À semelhança do serviço GKE da AWS, os *master nodes* são geridos pela Google, sendo apenas acessíveis através da *API* do *Kubernetes* associada ao *cluster*.

No contexto da aplicação utilizada na dissertação, que é *deployed* pelo operador, esta é um servidor *Nginx*. O *Nginx* é um servidor *web* de código aberto, conhecido pela sua alta performance, escalabilidade e baixo consumo de recursos. É frequentemente utilizado como um servidor *web* reverso, *proxy HTTP* e *proxy* de e-mail, além de ser uma opção popular para servir conteúdo estático e dinâmico em aplicações *web* [36].

5.2 Desenvolvimento

Esta secção descreve o desenvolvimento do operador, bem como o *deployment* dos vários componentes necessários ao funcionamento do mesmo, nos *clusters* anteriormente referidos.

5.2.1 Componentes *deployed* com *Helm*

A fim de instalar determinadas *stacks* de alguns componentes presentes na arquitetura como o *Prometheus*, *Kubecost*, *Nats* e *Zookeeper*, utilizou-se o *Helm*, devido à facilidade que oferece em criar os objetos *Kubernetes* necessários para o funcionamento destes no *cluster*, sendo preciso apenas passar os parâmetros de acordo com o que se pretende para este caso. Todos foram instalados com recurso aos comandos *helm add repo* e *helm install*, sendo que em alguns dos casos foi necessário fazer algumas configurações extra.

Nats

O operador federado é composto por múltiplas instâncias do mesmo operador *deployed* em diversos *clusters*. Para que estas instâncias possam exibir um com-

[3]Máquina virtual de custo otimizado que oferece 1 vCPU e 4 GB de RAM

portamento federado, é essencial que elas trabalhem em conjunto, como uma só, e de forma coesa para tomar decisões em relação ao *cluster*, de maneira a evidenciar, assim, esse comportamento. Portanto, foi necessário implementar um sistema de comunicação entre essas instâncias, sendo que tal foi alcançado através da utilização do sistema de mensagens *Nats*, com recurso aos conceitos de *subject* e fila de mensagens. Um *subject* é basicamente um tópico para o qual os remetentes podem publicar mensagens e os destinatários podem subscrever-se para as receber. Durante a implementação foram definidos *subjets* específicos para cada mensagem que necessita de ser transmitida, de acordo com o seu conteúdo.

A escolha do *Nats* como meio de comunicação entre os operadores foi motivada por uma particularidade sua, que permite, através de *gateways*, isto é, componentes que possibilitam conectar vários *clusters Nats*, estabelecer a comunicação entre os mesmos. Os *gateways* são responsáveis pelo roteamento das mensagens para os vários *clusters*, que podem ou não se encontrar em diferentes redes e locais. Isto permite formar um *super cluster Nats*, onde as mensagens são reencaminhadas para vários *clusters Nats*. Ora, isto torna-se bastante útil neste cenário em questão, pois cada operador encontra-se num *cluster Kubernetes* e rede distinta e, desta forma, o operador apenas necessita de ter conhecimento do *Nats* que se situa no seu *cluster* e comunicar através dele, sendo que o *Nats* é o responsável por reencaminhar as mensagens para os outros *clusters Nats*, que juntos formam o *super cluster Nats*.

Super cluster nats

Num ambiente federado, onde as diversas instâncias do operador estão distribuídas por vários *clusters* e redes, a descoberta de serviços é um desafio complexo. Existe uma necessidade de permitir que as diferentes instâncias se consigam comunicar de forma eficiente, independentemente da sua localização, sendo isto crucial para a coesão do sistema. Neste contexto, os *gateways* do *Nats* surgem como uma abordagem para a solução deste problema, ao permitirem a criação de um *super cluster*. Desta forma, a descoberta de serviços é simplificada, sendo que as instâncias do operador podem comunicar com outras instâncias em *clusters* diferentes, sem precisar de saber onde exatamente estão localizadas. Isso elimina a necessidade de configurar manualmente rotas de comunicação entre as várias instâncias, resolvendo o problema de descoberta de serviços.

Ora, para permitir a comunicação foi necessário criar o *super cluster Nats*, tirando vantagem dos *gateways*. Um *super cluster Nats* é um conjunto de *clusters Nats* que estabelecem conexões entre eles, através de *gateways* que propagam as mensagens para os outros *gateways*. No ficheiro de configuração presente na Figura 5.1 (superclusters.yaml), definiram-se as configurações necessárias para configurar o *Nats* e os seus *gateways*. No ficheiro, cada *gateway* tem o nome do *cluster Kubernetes* em que se encontra e o *url* do *Nats* correspondente. Desta forma, quando um operador publica uma mensagem no *Nats* presente no seu *cluster Kubernetes*, este tem os *gateways* configurados para encaminhar a mensagem para todas as restantes instâncias do operador presentes no sistema. Para adicionar um operador ao *super cluster Nats* basta acrescentar um *gateway* ao ficheiro e aplicá-lo em todos

os *clusters* do sistema, para que eles passem a conseguir comunicar com o novo operador e este também o consiga fazer.

Como cada instância do operador só interage com os componentes do seu *cluster*, o operador apenas publica e lê mensagens do seu *Nats*, com o qual comunica através de *localhost*, não precisando de conhecer as redes e *clusters* em que se encontram os outros operadores, ficando essa tarefa ao encargo do *Nats*.

```
gateway:
  enabled: true

  # NOTE: defined via --set gateway.name="$ctx"
  # name: $ctx

  gateways:
  - name: dev-cluster.sysobs.dei.uc.pt
    urls:
    - nats://a72b113bdf5714bf295a4f26aaa7ba89-159225451.eu-west-1.elb.amazonaws.com:7522

  - name: test.eu-west-1.eksctl.io
    urls:
    - nats://a7715da3511ed4ff1b1b9b5869a31e95-1481916464.eu-west-1.elb.amazonaws.com:7522

  - name: gke_coherent-server-389015_europe-west1_filipe-arrais-gke
    urls:
    - nats://35.233.38.233:7522


natsbox:
  enabled: true
```

Figura 5.1: Ficheiro de configuração do *super cluster Nats*

Para possibilitar a comunicação com as outras instâncias *Nats*, para além das configurações dos *gateways*, foi necessário expor o *Nats* com recurso a um serviço *Kubernetes* do tipo *load balancer*, como apresentado na Figura 5.2, que expõe as portas necessárias para a comunicação do *Nats*. Este garante que o *Nats* seja acessível fora da rede pelos *gateways* presentes nos diferentes *clusters*, permitindo a troca de mensagens entre as várias instâncias do operador federado.

```
apiVersion: v1
kind: Service
metadata:
  name: my-nats-load-balancer
  namespace: nats
spec:
  type: LoadBalancer
  selector:
    app.kubernetes.io/instance: my-nats-aws
    app.kubernetes.io/name: nats
  ports:
  - name: client
    port: 4222
    targetPort: 4222
    nodePort: 31000
  - name: cluster
    port: 6222
    targetPort: 6222
    nodePort: 31001
  - name: monitor
    port: 8222
    targetPort: 8222
    nodePort: 31002
  - name: metrics
    port: 7777
    targetPort: 7777
    nodePort: 31003
  - name: leafnodes
    port: 7422
    targetPort: 7422
    nodePort: 31004
  - name: gateways
    port: 7522
    targetPort: 7522
    nodePort: 31005
```

Figura 5.2: Serviço do tipo *Load Balancer* que expõe o *Nats* no *cluster* da AWS

Zookeeper

A eleição de líder é um conceito de sistemas distribuídos que se refere ao processo de eleição de um único líder num grupo de nós. Sendo o sistema desenvolvido um sistema distribuído, foi necessário implementar o conceito de eleição de líder no operador federado, devido ao facto de, se deixarmos que cada instância do operador tome decisões relativas aos vários *clusters*, poderá levar a inconsistências e conflitos relativamente ao estado dos mesmos. Devido a isso, podemos cair numa situação onde as cargas de trabalho são duplicadas num determinado *cluster* ou migradas de um *cluster* para vários em vez de apenas um, devido a diferentes instâncias do operador escolherem *clusters* distintos para a migração. De forma a evitar estes conflitos e inconsistências, a eleição de um líder é essencial, para que apenas este e só este tome decisões em relação aos vários *clusters*.

A escolha do *Zookeeper* parte da consistência e confiabilidade que o serviço oferece para resolver problemas de coordenação em sistemas distribuídos e também pela facilidade de uso e redução da complexidade do código, em contrapartida ao que seria se se implementasse um algoritmo de eleição de líder no próprio operador, que é uma tarefa que traz uma grande complexidade. Além disso, o *Zookeeper* já possui mecanismos de deteção de falhas implementadas, que detetam quando um operador se torna indisponível para a eleição do líder e permite também ter um repositório centralizado com todos os operadores ativos com um

identificador único atribuído.

O *Zookeeper* é o único componente que, como já referido anteriormente, se encontra apenas *deployed* num dos *clusters*, devido à complexidade exigida para sincronizar várias instâncias num ambiente *multi-cloud* e *multi-cluster* e como a eleição de líder não é o foco, o *Zookeeper* foi apenas um serviço utilizado como uma solução para este problema, optando-se por se manter assim. De forma a permitir que todas as instâncias do operador se consigam conectar ao servidor, foi necessário expô-lo através de um serviço *Kubernetes* do tipo *load balancer*, para que este seja acessível pelas outras instâncias de operadores que não se encontrem nesse *cluster*, tendo sido criado o serviço presente na Figura 5.3.

```
apiVersion: v1
kind: Service
metadata:
  annotations:
    meta.helm.sh/release-name: my-release
    meta.helm.sh/release-namespace: zookeeper
  name: zookeeper-loadbalancer
  namespace: zookeeper
spec:
  selector:
    app.kubernetes.io/component: zookeeper
    app.kubernetes.io/instance: my-release
    app.kubernetes.io/name: zookeeper
  ports:
  - name: client
    port: 2181
    targetPort: 2181
  - name: follower
    port: 2888
    targetPort: 2888
  - name: leader
    port: 3888
    targetPort: 3888
  type: LoadBalancer
```

Figura 5.3: Serviço *load balancer* que expõe o *Zookeeper*

Prometheus

O *Prometheus* desempenha um papel fundamental no sistema, pois é o responsável por recolher e armazenar as métricas dos vários componentes do *Kubernetes*, que serão usadas posteriormente pelo *Kubecost* para calcular os custos de alocação e prever custos futuros. É importante realçar que, para a recolha das métricas necessárias para o *Kubecost* efetuar os seus cálculos, este depende de outros componentes como o *Node Exporter* e o *Kube State Metrics*. Desta forma, o *Helm chart* utilizado permite o *deployment* do *Prometheus* no *cluster Kubernetes*, com os componentes *node exporter* e *kube state metrics*. Além disso, ainda cria um *Grafana* que permite a observação das métricas recolhidas em *dashboards*.

Kubecost

O *Kubecost*, como já apresentado, foi um *fit* perfeito para a obtenção de métricas de custos. É uma ferramenta bastante útil para ajudar a alcançar métricas detalhadas sobre os gastos dos recursos em *clusters Kubernetes*. Ao invés de depender de métricas de uso de recursos recolhidas pelo *Prometheus* e depois ter de consultar os preços da *cloud* em que o *cluster* está em execução e por fim ter de realizar os cálculos para obter os gastos do mesmo, o *Kubecost* automatiza todo este processo, o que facilita a visualização e análise dos gastos de cada *cluster*. Este deteta a instância do nó em que está a correr e calcula os gastos de cada *cluster* utilizando os preços da *cloud* correta, disponibilizados por objetos *Kubernetes*, isto é, é possível consultar os gastos por aplicação, *namespace* ou por qualquer outra dimensão relevante. O *Kubecost* foi integrado com a *stack* do *Prometheus deployed* no *cluster*, sendo necessário configurar certos valores do *Kubecost chart* para o próprio *Kubecost* não dar *deploy* de uma *stack* do *Prometheus* e utilizar a já existente no *cluster*.

Estes foram os componentes necessários para dar *deploy* no *cluster* para o funcionamento do operador federado, tendo sido organizados no *cluster* nos seguintes *namespaces*: *Nats* no *namespace nats*, *Zookeeper* no *namespace zookeeper* e *Prometheus* e *Kubecost* no *namespace prometheus*. O facto de as instâncias do operador só interagirem cada uma com a sua *stack* de componentes aumenta a tolerância a falhas, pois se um dos componentes ou determinado *cluster* ficar indisponível, apenas afetará uma instância do operador, permitindo que as outras continuem a funcionar sem serem afetadas. Além disso, possibilita uma adição ou remoção fácil de uma instância sem afetar nenhuma das outras, porque para adicionar uma instância à rede de operadores basta configurar o *gateway Nats*, adicioná-lo ao ficheiro de configuração do *Nats* e aplicá-lo a todos os *clusters*, sem ser necessário modificar o código do operador. Da mesma forma, para remover é suficiente apagar o *gateway* do ficheiro.

5.2.2 Operador federado

O primeiro passo para a criação do operador passou pela iniciação do projeto utilizando a *Operator Framework*, nomeadamente a ferramenta *operator-sdk* que permite gerar uma estrutura padrão do projeto com recurso ao comando:

operator-sdk init –domain example.com –repo github.com/example/nginx-operator

Após isso, foi necessário criar a *API* do operador, que é a definição de como o operador é representado num *cluster Kubernetes*. A *API* traduz-se diretamente para uma Custom Resource Definition (CRD), que descreve o esquema para um Custom Resource (CR) com o qual o operador irá interagir. Sem isto, não haverá maneira para o operador ler os valores passados no CR, portanto criar a *API* é um passo necessário antes de escrever qualquer lógica do operador. A criação da *API* envolve o comando:

operator-sdk create api –group operator –version v1alpha1 –kind NginxOperator –resource –controller

Este comando gera uma estrutura base em *Go*, pronta a completar com os parâmetros que pretendemos no objeto, que posteriormente irá gerar a CRD do operador. Após isto, teremos um projeto com vários ficheiros e diretórios com o aspeto da Figura 5.4, dos quais se destacam dois ficheiros principais, nginxoperatortypes.go e nginxoperatorcontroller.go, preenchidos com a lógica que se pretende para operador. O primeiro contém a estrutura do objeto referida anteriormente e o segundo a lógica do controlador que será apresentada adiante. Os outros diretórios contêm ficheiros que tratam de outros aspetos de baixo nível do *Kubernetes*.

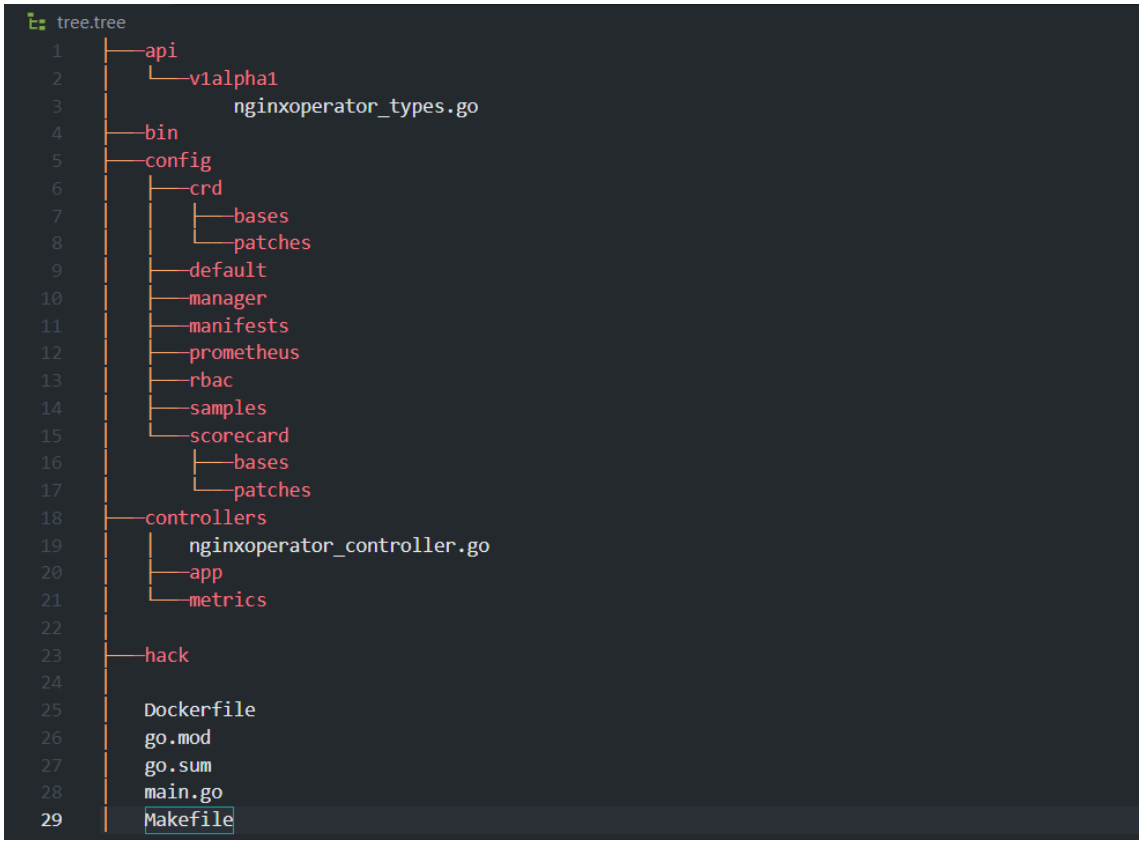


Figura 5.4: Estrutura do projeto

O desenvolvimento do operador federado começa então com a definição da sua CRD num objeto, no ficheiro nginxoperatortypes.go anteriormente referido. Este contém todas as especificações desejadas para o estado do recurso que o operador irá observar e controlar. A definição da CRD inclui os parâmetros apresentados na Figura 5.5. De notar que a estrutura apresentada apenas representa a definição da *spec* da CRD e não toda a totalidade da mesma. A *spec* é o campo que define os Custom Resources (CRs) que pretendemos criar.

```go
// NginxOperatorSpec defines the desired state of NginxOperator
type NginxOperatorSpec struct {

    // Port is the port number to expose on the Nginx Pod
    Port *int32 `json:"port,omitempty"`
    // Replicas is the number of deployment replicas to scale
    Replicas *int32 `json:"replicas,omitempty"`
    // ForceRedploy is any string, modifying this field
    // instructs the Operator to redeploy the Operand
    ForceRedploy string `json:"forceRedploy,omitempty"`
    //AppLimitCost is the limit for the deployment of an app
    AppLimitCost *int32 `json:"appLimitCost,omitempty"`
    //IsDeployed indicates if the app is deployed on this cluster
    IsDeployed bool `json:"isDeployed,omitempty"`
}
```

Figura 5.5: Estrutura que define a *Spec* da CRD do operador

- ***Port*:** Este parâmetro, que é um valor inteiro de 32 bits ('int32'), permite definir o número da porta no *deployment* em que será exposto o *Pod* da aplicação. Desta forma, o operador pode utilizar esse valor para configurar a porta de serviço da aplicação. É opcional, pois este parâmetro pode ser definido no *yaml* do *deployment* da aplicação.
- **Replicas:** Este campo permite definir através de um ponteiro de um inteiro de 32 bits ('int32') o número de *pods* para a aplicação. O operador usa este valor para ajustar a quantidade de réplicas do *Deployment* da aplicação, sendo, à semelhança do parâmetro *port*, opcional pelas mesmas razões.
- **ForceRedploy:** É um string opcional que permite instruir o operador a dar *redeploy* do operando. Útil para forçar uma ação de *redeploy*, quando necessário, sem ser preciso alterar as especificações atuais.
- **AppLimitCost:** É o parâmetro na *spec* da CRD que nos permite definir o SLO de custo para a aplicação. É um ponteiro de valor inteiro de 32 bits ('int32'), que o operador irá utilizar para verificar se o *deployment* se encontra dentro do valor definido, sendo um valor de comparação para perceber se o *deployment* pode ou não continuar no *cluster Kubernetes*.
- **IsDeployed:** É um booleano ('bool') que indica se a aplicação está *deployed* no *cluster*. Numa situação inicial, quando estamos a dar *setup* do sistema, permite escolher os *clusters* onde pretendemos que a aplicação seja *deployed*. Após isso, este campo é apenas alterado pelo próprio operador, para ele controlar em que *clusters* as aplicações devem ou não estar a correr. O operador usa este valor para perceber se tem ou não de dar *deploy* da aplicação.

Com esta definição, é possível criar um CR com as configurações apresentadas na Figura 5.6. Os campos *port* e *forceredeploy* são opcionais e, por essa razão, não foram incluídos no exemplo. O *namespace* e o *name* no campo *metadata* permitem definir o nome do *namespace* onde irão ser executados os *Pods* e o seu nome, respetivamente.

```yaml
apiVersion: operator.example.com/v1alpha1
kind: NginxOperator
metadata:
  name: pod-nginx
  namespace: teste3
spec:
  replicas: 10
  appLimitCost: 7
  isDeployed: true
```

Figura 5.6: Exemplo de um CR

Após a definição da CRD do operador, segue-se a lógica principal do mesmo, representada pelo *loop* de reconciliação. Conforme descrito em capítulos anteriores, os operadores funcionam na premissa de reconciliar o estado atual do *cluster* com o estado desejado para o mesmo. A execução deste *loop* é feita periodicamente ou aquando das alterações no estado atual, que acionam um determinado evento. A função principal do operador é a função *reconcille()*, presente no ficheiro do controlador (nginxoperatorcontroller.go), parte fundamental do operador. É neste ficheiro que delineamos grande parte da lógica do operador, onde definimos o controlador que irá observar os CRs e as decisões e a forma como as realiza para garantir que o estado atual desses recursos corresponda ao estado desejado, que é especificado através do CR do operador. A função *reconcille()*, como o próprio nome indica, é a função principal de reconciliação do controlador, chamada sempre que ocorrem eventos relevantes nos recursos controlados pelo operador. Esta é a função responsável por realizar as ações necessárias para garantir que o estado atual coincida com o estado desejado. Tendo em conta que este ficheiro é a base da lógica por trás do operador, irão ser focadas as funções nele presente.

O ficheiro é ainda composto por outra função, a função *setupWithManager()* que é das primeiras a ser chamada quando o operador é *deployed* no *cluster*. Esta, além de configurar o *manager* que vai ser responsável por responder a eventos e chamar a função *Reconcille*, é também onde foi criada a lógica para o operador estabelecer uma comunicação com o *cluster Nats* e com o servidor *Zookeeper*. Desta forma, conseguimos manter uma comunicação sem interrupções, visto que o operador reage a eventos. Ainda nesta função, é criado um nó sequencial e efémero num diretório de operadores no *Zookeeper*, pois ao fim da execução desta função o operador encontra-se pronto e disponível no *cluster*. Nessa função, existe código para observar alterações aos objetos *NginxOperator*, que são os objetos criados pela aplicação dos CRs e também para observar o *deployment* do servidor *Nginx*, visto que o operador também está responsável pela gestão do mesmo. Assim, se houver, alterações no *deployment* da aplicação, o operador sabe que tem de executar a função *Reconcille*.

Relativamente a esta mesma função, a *Reconcille*, esta é iterada da seguinte forma:

O operador está encarregue de recolher os custos relativos à aplicação no *cluster*. Para isso, este utiliza duas *APIs* do *Kubecost*, a *Allocation API* e a *Predict API*. Na primeira *API*, o operador extrai os custos agregados por *namespace*, filtrando-os

pelo *namespace* onde a aplicação está em execução. Neste pedido, é sinalizado que se pretende a soma acumulada, ou seja, a soma da execução da aplicação nos diversos dias acumulados ao longo do tempo em que esta se encontra em execução no *cluster*, numa janela específica que, neste caso é de trinta dias. Desta forma, o operador obtém o custo total da aplicação durante o período considerado.

Na segunda *API*, o operador envia uma solicitação, passando o ficheiro *yaml* que contém as informações do *deployment Nginx*, onde são descritos os recursos de *CPU* e *RAM* necessários, bem como uma descrição de como esta deve ser *deployed* no *cluster*, apresentado na Figura 5.7. Posteriormente, o *Kubecost* analisa o impacto desse *deployment* no *namespace* pretendido, onde a aplicação poderá ser executada e para a realização dessa análise, o *Kubecost* utiliza dados de comportamentos históricos, em termos de recursos e consumo destes. Com base nessas informações, o *Kubecost* realiza uma estimativa do custo mensal esperado para o *namespace* em questão, após a execução do *deployment* da aplicação. Desta forma, o operador consegue ter informações importantes para a tomada de decisões, de modo a minimizar as violações do SLO ao utilizar a informação da *Allocation API* para perceber se a aplicação se encontra dentro do *budget* definido para a mesma e a *Predcit API* para tomar decisões relativas à escolha de outro *cluster* que consiga correr a aplicação dentro do valor do SLO definido. Após isso, o operador comunica esses dados através de *Nats*, nomeadamente para o tópico *PredictCost*, e lê também as mensagens desse tópico, mantendo um mapa global com todas as instâncias dos operadores e os seus dados de custo. Neste tópico, tem acesso a um conjunto de informações como, se a aplicação está *deployed* naquele operador, o custo real e a previsão do custo mensal da aplicação naquele *cluster*.

O operador analisa os seus dados em relação ao SLO definido, observando se a aplicação se encontra ou não dentro do limite. Caso ocorra uma violação do SLO, o operador comunica-a através de *Nats* para o tópico *Decisons*, onde o operador líder estará à escuta de mensagens para efetuar a devida decisão, de modo a manter o SLO dentro do limite. Para eleição do líder, é utilizado o *Zookeeper* para estabelecer um diretório centralizado, onde cada instância do operador federado regista um nó efémero e sequencial, sendo que o operador líder é eleito consultando esse diretório e o que tiver o identificador mais baixo é selecionado. O operador líder analisa, então, os dados recolhidos de cada operador e, dentro dos *clusters* disponíveis para receber a aplicação, escolhe o mais barato e inicia as ordens de migração, enviando duas mensagens para o tópico *Orders*, sendo uma delas para o *cluster* que violou o SLO retirar a aplicação e outra para o *cluster* escolhido iniciar o *deployment* da aplicação. A escolha do *cluster* é feita com base em se o *cluster* tem a aplicação *deployed*, na previsão de custos e no SLO. Para um *cluster* estar disponível, ele não pode ter a aplicação *deployed* e a previsão para o *deployment* da aplicação tem de estar dentro do SLO de custo definido, caso contrário o *cluster* não está elegível para receber a aplicação. Dentro dos *clusters* que reúnem essas condições, é escolhido o que tiver previsão mais baixa.

De referir ainda que, para leitura de mensagens do *Nats*, foi utilizado o conceito de *QueueSubscribe* para garantir que todas as instâncias do operador federado recebem as mensagens. Através do *QueueSubscribe*, as mensagens são distribuídas de forma equitativa entre as instâncias, o que permite um processamento

uniforme e evita a sobrecarga numa única instância do *Nats* (cada mensagem é lida por uma instância do *Nats*). Ainda para evitar perdas de mensagens, como a função *reconcille* é um *loop* que executa periodicamente, foi necessário reduzir esse tempo de execução para dez segundos, evitando, assim, que as mensagens enviadas pelo operador sejam perdidas.

Por fim, o operador contém também uma lógica para criar ou atualizar o *deployment* da aplicação (Figura 5.7), neste caso o servidor *Nginx*, responsável por manter a aplicação. Nesta lógica, caso o *deployment* seja eliminado ou ocorram alterações no mesmo, o operador é responsável por o trazer de volta e também é aqui que são alterados os valores de *port* e de *replica*, caso sejam especificados no CR do operador. Os valores do ficheiro *yaml* do *deployment* da aplicação são sobrepostos pelos valores do CR do operador.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 10
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
        resources:
          limits:
            cpu: "400m"
            memory: "512Mi"
          requests:
            cpu: "200m"
            memory: "256Mi"
```

Figura 5.7: *Deployment* do *Nginx*

Capítulo 6

Testes

De forma a assegurar que o operador federado cumpre com as funcionalidades especificadas no Capítulo 3, foram realizados testes. Entre eles, testes manuais para verificar se o operador cumpre o seu propósito, descrito na secção 6.1, e um teste de quase ambiente de produção, na secção 6.2. Segue-se a secção 6.3 que contém a conformidade do sistema com os atributos de qualidade da secção 3.2 do Capítulo 3.

6.1 Testes Manuais

Para a validação do sistema foram efetuados testes, de modo a verificar se o operador federado tem o comportamento esperado, de acordo com os seus requisitos. De recordar que o operador é executado como um *loop*, sendo que todas as suas funcionalidades formam uma grande, pois são executadas de uma só vez e, por essa razão, é importante testar o operador como um todo.

6.1.1 Teste em ambiente *multi-cluster*

Neste teste, inicialmente começou-se por testar o operador num ambiente *multi-cluster*, tendo sido utilizados os dois *clusters* da Amazon Web Services (AWS) anteriormente referidos, o criado com o Elastic Kubernetes Service (EKS) e o criado com recurso à ferramenta *Kops*. Aqui foram definidos os Custom Resources (CRs) presentes nas Figuras 6.1 e 6.2, contendo estes indicação para um *deployment* de 10 réplicas, um objetivo do nível de serviço (SLO) de limite mensal de 60 dólares de *budget* e a indicação de se a aplicação deve ser *deployed* no *cluster*. Apesar disto, neste teste a definição do SLO é irrelevante, pois o que aqui se efetuou foi simular a violação do SLO através da alteração do parâmetro do SLO, isto é, o Custom Resource (CR) do operador que continha a indicação para o *deployment* da aplicação era atualizado, sendo o SLO configurado para o valor zero, o que forçava a sua violação o que, consequentemente, desencadeava o conjunto de operações que o operador necessita de realizar para migrar a carga para outro *cluster* que cumpra o SLO. Para que isso acontecesse, o valor do SLO no outro *cluster* teria

de ser superior ao custo do *deployment* que, no caso, era o valor de 60 dólares. Isto porque o *deployment* utilizado tem um custo de 40 a 50 dólares e, para um *cluster* ser elegível para a migração da aplicação ele tem de respeitar o SLO, ora portanto, se o SLO fosse inferior ao custo mensal do *deployment*, o *cluster* que não contém a aplicação não poderia receber a carga uma vez que não o respeitaria e, como tal, não seria possível observar a migração. Através da manipulação do valor do SLO, testaram-se todas as combinações possíveis de migração da carga repetidamente, ou seja, as duas combinações várias vezes.

O teste apresentou resultados positivos, já que o operador detetou sempre as violações de SLO quando o limite era reduzido para zero, iniciando todo o processo para manter a aplicação num *cluster* que cumprisse o SLO e, consequentemente, efetuando com sucesso todas as migrações da mesma.

```yaml
apiVersion: operator.example.com/v1alpha1
kind: NginxOperator
metadata:
  name: pod-nginx
  namespace: teste
spec:
  replicas: 10
  appLimitCost: 60
  isDeployed: false
```

Figura 6.1: Custom resource (CR) criado no *cluster Kops* AWS

```yaml
apiVersion: operator.example.com/v1alpha1
kind: NginxOperator
metadata:
  name: pod-nginx
  namespace: teste
spec:
  replicas: 10
  appLimitCost: 60
  isDeployed: true
```

Figura 6.2: Custom resource (CR) criado no *cluster eks*

6.1.2 Teste em ambiente *multi-cluster* e *multi-cloud*

Após o teste do operador num cenário *multi-cluster*, adicionou-se, além disso, o cenário *multi-cloud*, através da junção do *cluster* criado com o Google Kubernetes Engine (GKE) da Google aos *clusters* anteriormente referidos, com a criação de um CR igual ao presente na Figura 6.1.

O processo do teste utilizado foi equivalente ao teste referido na subsecção anterior, tendo-se testado, de igual forma, todas as combinações possíveis de migração da aplicação que, no caso dos três *clusters*, são seis, através da alteração do campo do CR *appLimitCost* que é referente ao SLO.

Neste teste também se simulou o caso de em alguma eventualidade não haver nenhum *cluster* com disponibilidade para receber a aplicação. De relembrar que, para que um *cluster* esteja disponível para receber a aplicação, ele não pode ter a aplicação já a correr e a previsão de custo para a mesma, naquele *cluster*, tem de ser inferior ao SLO definido. Portanto, para simular este cenário alterou-se o valor do SLO para zero em todos os operadores.

O resultado do teste foi novamente positivo, tendo o operador federado detetado todas a violações do SLO e efetuado todas as migrações da aplicação com sucesso. Para o cenário em que não existem *clusters* disponíveis para receber a aplicação, o operador também teve o comportamento esperado, pois manteve a aplicação na mesma no *cluster* e esperou até haver outro que pudesse recebê-la.

Ainda neste teste, através da observação dos *logs* do operador, verificou-se se este escolhia o *cluster* cuja previsão era a mais baixa para a migração da aplicação e se o líder era elegido corretamente, juntamente com a observação direta do repositório centralizado do *Zookeeper*.

6.2 Teste de quase ambiente de produção

O teste apresentado nesta secção foi bastante aproximado do que seria a utilização do operador num ambiente de produção. Ainda assim, não o é considerado completamente porque o operador está pensado para ter em conta um SLO de custo mensal, significando isto que, caso o teste fosse efetuado totalmente em ambiente de produção, o lógico seria definir um SLO de 50 a 60 dólares, dado que o *deployement* custaria entre 40 a 50 dólares mensais, sendo necessário um maior período de tempo para a realização dos testes. Portanto, ao invés disso e como forma de encurtar esse mesmo período, definiu-se um SLO de limite de 7 dólares por mês para a aplicação. Por outras palavras, desta forma o SLO era atingido dentro de sensivelmente uma semana, sendo assim possível, num período de tempo mais curto, observar o comportamento do operador. Caso contrário, ou seja, se se tivesse um SLO mensal adequado ao *deployment* da aplicação e, assim, um teste em ambiente de produção, a espera teria de ser de sensivelmente um mês para observar uma eventual violação do SLO que poderia nem se verificar, pois a aplicação poderia, nesse mês, ficar abaixo do SLO, não se observando, nesse caso, o comportamento do operador para manter a aplicação dentro

do SLO, sendo exigível assim um teste de vários meses. Ao definirmos um SLO pouco adequado ao custo do *deployment* da aplicação, garante-se que as violações do mesmo ocorreriam num espaço de tempo muito mais curto. Apesar disto, ressalta-se que os *clusters* que não tinham a aplicação a correr teriam de ter o campo *appLimitCost* configurado para 60 euros, caso contrário não estariam elegíveis para a migração, pelas razões já referidas anteriormente. Por outro lado, o que continha a aplicação a correr era sempre configurado com um limite de 7 dólares.

Destaca-se ainda que foram utilizados os três *clusters* referidos ao longo da dissertação, o *cluster Kops*, EKS e GKE e definidos os seguintes CRs respetivamente, presentes nas Figuras 6.3 e 6.4. Como se pode observar nas mesmas, a aplicação inicialmente foi *deployed* no *cluster Kops* e deixou-se o sistema a correr para se observar o comportamento do operador. Em vez de se mexer no valor do SLO, como se fez nos testes apresentados anteriormente para provocar violações do SLO, esperou-se que a aplicação o atingisse.

```yaml
apiVersion: operator.example.com/v1alpha1
kind: NginxOperator
metadata:
  name: pod-nginx
  namespace: teste3
spec:
  replicas: 10
  appLimitCost: 7
  isDeployed: true
```

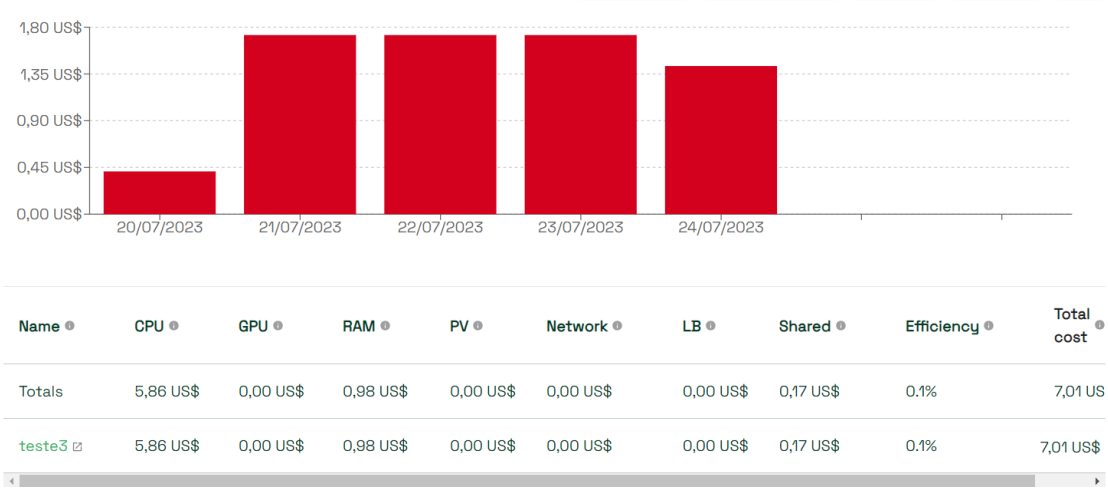
Figura 6.3: CR criado no *cluster Kops* AWS

```yaml
apiVersion: operator.example.com/v1alpha1
kind: NginxOperator
metadata:
  name: pod-nginx
  namespace: teste3
spec:
  replicas: 10
  appLimitCost: 60
  isDeployed: false
```

Figura 6.4: CR criado no *cluster EKS e GKE*

Posto isto, o teste teve início no dia 20/07 e terminou a 31/07, tendo uma duração de 11 dias. Abaixo pode observar-se a evolução do custo ao longo do tempo, sendo sempre apoiada por dois gráficos, um com a linha temporal relativa a 7 dias e outro com a linha temporal relativa a 30 dias, cada um deles para os três

clusters. Como anteriormente referido, a aplicação foi inicialmente *deployed* no *cluster Kops*, tendo estado a correr, conforme as Figuras 6.5 e 6.6, desde 20/07, até atingir o SLO de custo definido a 7 euros no dia 24/07. Após atingir o limite, pode constatar-se que a aplicação foi imediatamente retirada pelo operador, não apresentando mais custos nos dias seguintes, significando que foi migrada para outro *cluster* que cumprisse o SLO.



Name	CPU	GPU	RAM	PV	Network	LB	Shared	Efficiency	Total cost
Totals	5,86 US$	0,00 US$	0,98 US$	0,00 US$	0,00 US$	0,00 US$	0,17 US$	0.1%	7,01 US
teste3	5,86 US$	0,00 US$	0,98 US$	0,00 US$	0,00 US$	0,00 US$	0,17 US$	0.1%	7,01 US$

Figura 6.5: Evolução do custo de alocação no *cluster Kops* num período de 7 dias

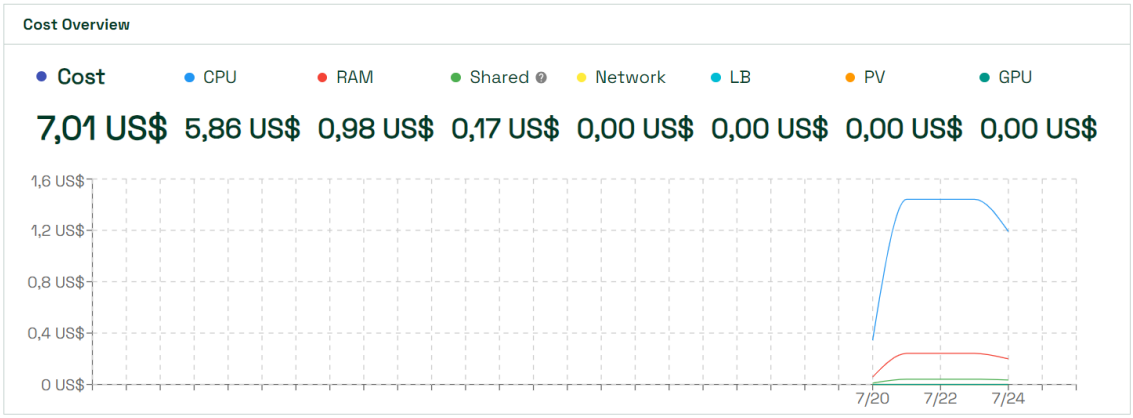
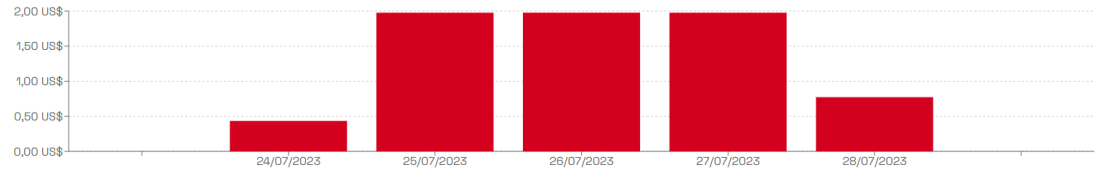


Figura 6.6: Evolução do custo de alocação no *cluster Kops* num período de 30 dias

No caso, a aplicação foi migrada para o *cluster* EKS, como se pode observar nas Figuras 6.7 e 6.8. No dia 24/07 foi então iniciado, por parte da instância presente no *cluster* EKS, o *deployment* da aplicação, tendo novamente estado a correr até atingir de novo o SLO neste *cluster*, o que se sucedeu no dia 28/07. Neste dia, voltou-se a iniciar uma migração da aplicação para outro *cluster*.



Name	CPU	GPU	RAM	PV	Network	LB	Shared	Efficiency	Total cost
Totals	2,75 US$	0,00 US$	0,46 US$	0,00 US$	0,00 US$	0,00 US$	3,92 US$	0.1%	7,14 US$
teste3	2,75 US$	0,00 US$	0,46 US$	0,00 US$	0,00 US$	0,00 US$	3,92 US$	0.1%	7,14 US$ N/A

Figura 6.7: Evolução do custo de alocação no *cluster EKS* num período de 7 dias

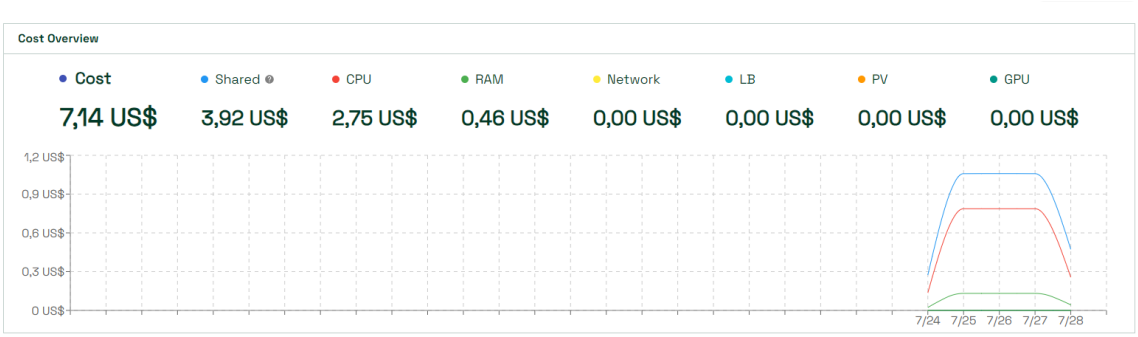
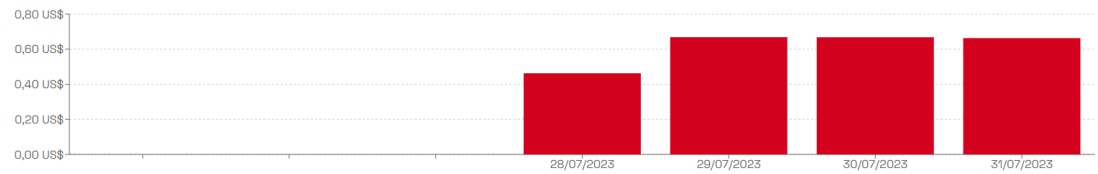


Figura 6.8: Evolução do custo de alocação no *cluster EKS* num período de 30 dias

Por fim, o *cluster* que concluiu o processo de migração ao iniciar o *deployment* da aplicação no dia 28/07 foi o *cluster* GKE da Google, como se pode observar nas Figuras 6.9 e 6.10. De notar que este, devido a ter sido criado com os créditos grátis oferecidos pela Google, não tinha capacidade para iniciar as dez réplicas de *Pods* da aplicação devido aos recursos reduzidos, tendo apenas conseguido iniciar dois *Pods*. Tal levou o *cluster* a ter os menores custos diários, sendo que era o que tinha previsão mais cara para o *deployment* da aplicação dando, desta forma, a ilusão de ser o mais barato quando não é.



Name	CPU	GPU	RAM	PV	Network	LB	Shared	Efficiency	Total cost
Totals	0,84 US$	0,00 US$	0,14 US$	0,00 US$	0,00 US$	0,00 US$	1,48 US$	0.1%	2,46 US$
teste3	0,84 US$	0,00 US$	0,14 US$	0,00 US$	0,00 US$	0,00 US$	1,48 US$	0.1%	2,46 US$ N/A

Figura 6.9: Evolução do custo de alocação no *cluster GKE* num período de 7 dias

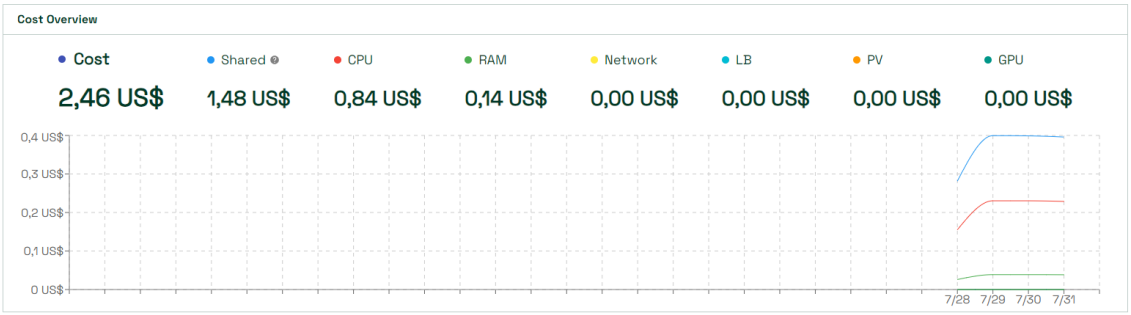


Figura 6.10: Evolução do custo de alocação no *cluster GKE* num período de 30 dias

Como se pode observar pelos gráficos apresentados, o operador realizou com sucesso a deteção da violação do SLO, tendo tomado as medidas necessárias para as minimizar. Claro que neste teste, devido ao SLO estar desadequado, o operador não consegue encontrar um *cluster* que efetivamente consiga cumprir o SLO mensalmente.

6.3 Conformidade dos atributos de qualidade

Esta secção contém uma análise da conformidade do sistema com os atributos de qualidade descritos na Secção 3.2.

- Portabilidade: Para cumprir com este atributo de qualidade recorreu-se à utilização de *Docker containers* para a construção do operador, o que permite que o mesmo seja *deployed* em qualquer *cluster Kubernetes*, quer na *cloud* ou *on-premisses*, sem que seja necessário alterar o código fonte. Durante a dissertação, diversos *clusters* foram utilizados, desde *clusters* locais como *Minikube* e *Kind* e soluções de *clusters* na *cloud*, como EKS e *kops* na AWS e GKE na Google. O operador foi *deployed* com sucesso em todos estes *clusters* e com o funcionamento correto, sem ser necessário efetuar qualquer modificação para que este corresse em todos os *clusters* referidos. Como tal, o operador está em conformidade com este atributo de qualidade.

- Confiabilidade: De forma a que o sistema esteja em conformidade com este atributo de qualidade foi necessário implementar mecanismos de recuperação. Primeiramente, o operador gere todo o ciclo de vida da aplicação, isto é, caso o *deployment* da aplicação falhe, seja eliminado, ou o número de réplicas não esteja correto, possui lógica para trazer de novo o *deployment*, no caso da sua eliminação, e lógica para manter o *deployment* no estado pretendido.

  Outro mecanismo de falha tem a ver com a eleição de líder. O operador utiliza o *Zookeeper* que deteta automaticamente quando um operador falha ou não está disponível, eliminando-o do repositório centralizado. Juntando a isto, o operador utiliza esse repositório para eleger um líder através de

uma lógica simples de escolha por meio de um *ID*, estando de forma constante a fazer esta verificação. Isto permite que exista sempre um líder ativo capaz de tomar decisões relativas aos vários *clusters* e que todas as instâncias estejam sincronizadas com o líder. O operador também armazena os últimos custos de alocação e de previsão, o que permite que, em caso de falha temporária da *API* do *Kubecost*, este consiga na mesma manter o seu funcionamento e a tomada de decisões sobre o *cluster*.

Para a verificação deste atributo foram efetuados vários testes: Para a primeira parte, o *deployment* da aplicação foi eliminado, tendo o operador conseguido detetar esta eliminação e lançar de novo o *deployment*. Ainda neste teste, também se alterou o número de réplicas e observou-se o operador a alterar o número de *Pods* para o correto. Para a segunda parte, deu-se *setup* das várias instâncias do operador nos diversos *clusters*, procedeu-se à terminação da instância líder e observou-se a eleição de outro líder após esse momento, tendo todas as instâncias ficado sincronizadas com o novo líder. Ao longo dos testes anteriormente referidos, também se observaram várias mudanças de líder, pois em *Kubernetes* é normal os *Pods* serem reiniciados muitas vezes e, devido a isso, esta lógica de tolerância à falha do líder é muito importante pois permite que o sistema esteja sempre disponível para cumprir com o seu propósito.

Por fim, para o último mecanismo, interrompeu-se temporariamente a ligação do operador com a *API* do *Kubecost* e observou-se que este consegue igualmente tomar uma decisão na mesma, pois consegue saber o custo previsto para a aplicação noutros *clusters*, ficando, no entanto, incapaz de detetar as violações do SLO, pois não consegue acompanhar o preço da alocação de recursos a aumentar, tendo apenas o valor referente à última chamada da *API*. Por outro lado, no caso de a ligação ser interrompida nos *clusters* onde a aplicação não está a correr, o operador irá conseguir ter o funcionamento completamente normal, pois irá detetar a violação do SLO e conseguirá tomar uma decisão com os valores de previsão armazenados pelos outros *clusters*.

- Manutenibilidade: O operador foi desenvolvido com a *operator framework*, o que permite criar um diretório e ficheiros padrão com uma determinada estrutura. Ora, isto facilita futuras modificações, pois o operador segue determinada estrutura que está bem documentada na *framework*, o que contribui para uma localização de código rápida, sem a necessidade de olhar para todos os ficheiros. A *framework* também segue as melhores práticas do ecossistema do *Kubernetes* e, desta forma, estamos a adotar as mesmas para o operador desenvolvido. O operador também foi construído de forma modular, isto é, é possível alterar ou configurar certos parâmetros sem a necessidade de alterar código ou de compilar o operador de novo. Para alcançar isto, foi criado uma Custom Resource Definition (CRD) com os parâmetros *Port, Replicas, ForceReploy, AppLimitCost* e *IsDeployed*, referidos anteriormente na secção 6.2.2, que permitem que a alteração dos mesmos seja modular.

Capítulo 7

Planeamento

Este capítulo fornece informação sobre o cronograma do estágio, dividido em dois semestres. Ambas as secções contém gráficos de *Gantt* desenvolvidos com recurso à ferramenta *Excel*, que ilustram o cronograma da realização da dissertação ao longo desse período de tempo.

7.1 Primeiro semestre

O planeamento para o primeiro semestre foi delineado pelos orientadores da dissertação. O trabalho relativo a este período de tempo contemplou uma vertente mais teórica, de pesquisa e aprendizagem. O gráfico de *Gantt* presente na Figura 7.1 ilustra o trabalho do primeiro semestre e como este foi divido ao longo do tempo.

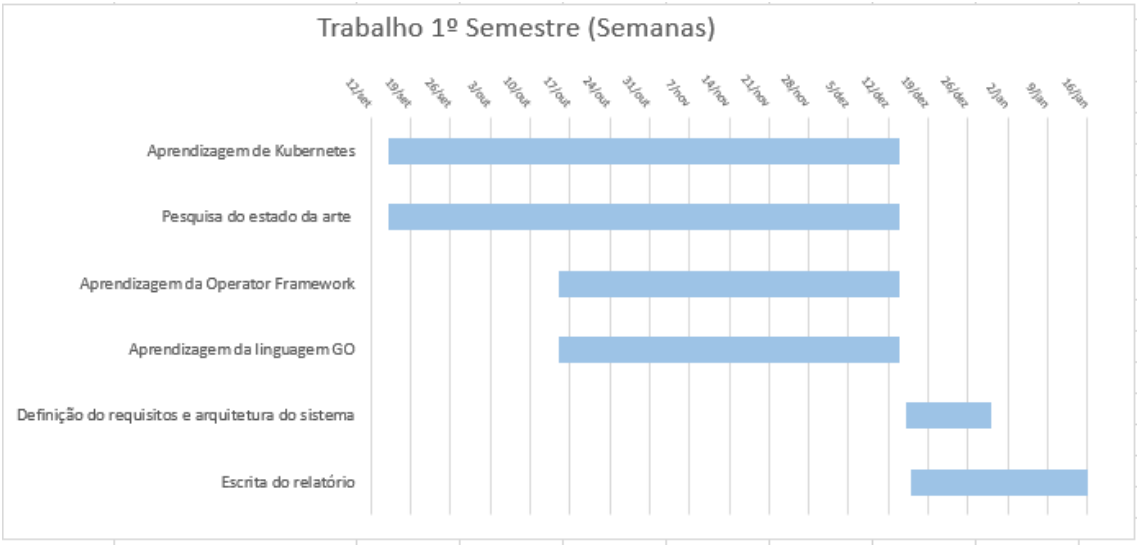


Figura 7.1: Diagrama de *Gantt* para o primeiro semestre

- **Aprendizagem de *Kubernetes*:** Começou-se por uma aprendizagem da tecnologia, tendo-se iniciado, primeiramente, com uma compreensão dos con-

ceitos fundamentais, o seu funcionamento, terminologia e arquitetura. Esta parte envolveu leitura da documentação oficial do *Kubernetes*, tutoriais e leitura de livros. À medida da progressão da aprendizagem, existiu uma vertente mais prática para a consolidação do conhecimento adquirido. Por fim, neste período também se exploraram maneiras de correr um *cluster Kubernetes* localmente.

- **Pesquisa do estado da arte:** Iniciou-se com leituras sobre conteúdos como micro-serviços, *containers*, orquestração de *containers*, aplicações *cloud-native* e *multi-cloud*, tendo sido revista literatura e artigos. Após isso, exploraram-se algumas tecnologias relacionadas, tanto com estes conceitos, como também com o conceito de operador aliado a questões de objetivos do nível de serviço (SLOs) e observabilidade.
- **Aprendizagem da *Operator framework*:** Primeiramente, foi realizado um estudo abrangente aos conceitos subjacentes à *framework*, nomeadamente a compreensão do conceito e princípios de operadores *Kubernetes*, controladores e Custom Resources (CRs). Foram explorados tutoriais de operadores simples, de forma a adquirir prática e familiarização com a estrutura destes, tendo-se efetivamente construído um operador, que serviu de base para criar o operador mais complexo proposto neste documento.
- **Aprendizagem da Linguagem *Go*:** Previamente, foi realizada uma exploração dos fundamentos da linguagem, a sua sintaxe, tipos de dados e estruturas de controlo, através da leitura da documentação oficial e com recurso a tutoriais práticos para consolidar os conceitos. Esta tarefa foi feita ao mesmo tempo que a anterior, pois aproveitou-se o facto de o operador ser escrito em *Go* para praticar a linguagem.
- **Definição de requisitos e arquitetura do sistema:** Durante este período foram levantados os requisitos do sistema a desenvolver e a realização de uma arquitetura usando o modelo C4.
- **Escrita do relatório:** Esta foi a última etapa do primeiro semestre, onde todo o conhecimento adquirido foi escrito, juntamente com a exposição do sistema proposto a desenvolver neste documento.

Durante o semestre, a cada duas semanas era realizada uma reunião com os orientadores da dissertação, o Professor Doutor Filipe Araújo e o Dr. Miguel Guerreiro. As reuniões permitiam a compartilha de ideias, esclarecimento de dúvidas, orientação e *insights* bastantes úteis para o desenvolvimento da presente dissertação.

7.2 Segundo semestre

O planeamento do segundo semestre teve uma vertente mais prática, tendo-se iniciado o processo de implementação do operador federado proposto no documento. O trabalho delineado para este semestre está presente na Figura 7.2, ilustrado por um diagrama de *Gantt*.

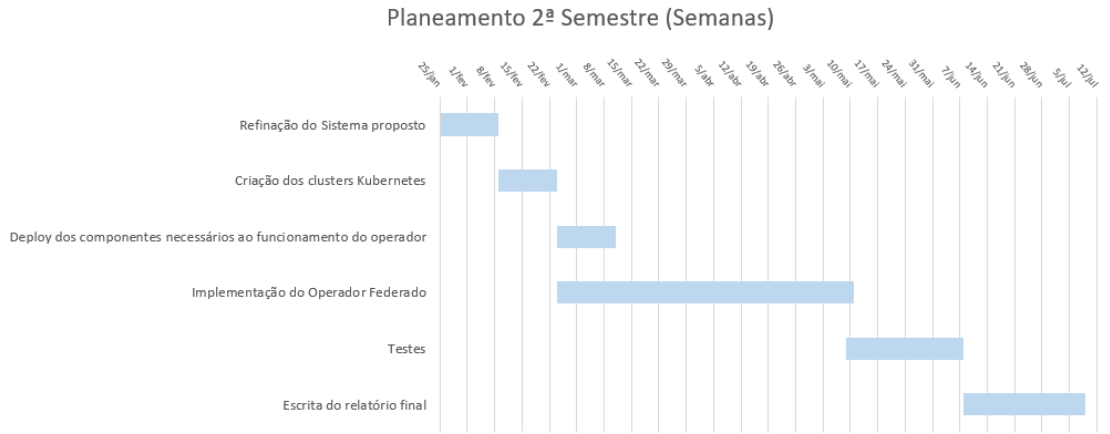


Figura 7.2: Diagrama de *Gantt* para o segundo semestre

- **Refinação do sistema proposto:** Esta tarefa consistiu numa especificação mais concreta e aprofundada do sistema a desenvolver, tendo envolvido uma refinação dos requisitos e arquitetura mais detalhada para um nível de abstração mais baixo do que o anteriormente apresentado. Durante esta tarefa também se concentrou em questões práticas, como os componentes necessários a utilizar para o funcionamento do operador federado. Desta forma, permitiu uma compreensão mais precisa do sistema a ser desenvolvido, fornecendo uma base mais sólida, necessária para a fase seguinte de implementação.
- **Criação dos *clusters Kubernetes***: Foi criado um cluster com o Elastic Kubernetes Service (EKS), feita uma pesquisa para a escolha de outro *cluster* e explorado o processo de criação do mesmo. O serviço escolhido foi o Google Kubernetes Engine (GKE), tendo sido criado posteriormente, devido ao período de experimentação ser curto, planeado-se a criação do mesmo apenas para quando o operador federado estivesse praticamente concluído.
- ***Deploy* dos componentes necessários ao funcionamento do operador:** Durante esta tarefa foi feita uma investigação de como dar *delpoy* dos componentes necessários ao funcionamento do operador, nomeadamente o *Nats*, *Zookeeper*, *Prometheus* e *Kubecost*. Foi analisada a ferramenta *Helm* que, posteriormente, foi utilizada para o *deployement* dos componentes anteriormente referidos nos *clusters*.
- **Implementação do Operador federado:** Consistiu na implementação dos requisitos definidos.
- **Testes:** Resumiu-se à realização de testes ao operador para verificar a conformidade do sistema com os requisitos definidos.
- **Escrita do relatório final:** Esta foi a última etapa da dissertação, onde todo trabalho e conclusões ao longo do ano foi descrito.

No segundo semestre, as reuniões de duas em duas semanas com os orientadores mantiveram-se nos mesmos moldes.

7.3 Limiar do Sucesso

Esta secção detalha todas as condições que devem ser atendidas na dissertação para ser considerada um sucesso.

Os seguintes objetivos devem, por isso, ser atingidos:

- O período de tempo reservado para a dissertação não pode ser excedido;
- Todas as funcionalidades do sistema descritas neste documento devem ser implementadas e testadas;
- O sistema criado deve servir como prova ao conceito de operador federado orientado a SLOs.

7.4 Cronograma Planeado *versus* Real

Em contraste com o cronograma original mostrado na Figura 7.2, o cronograma inicialmente planeado divergiu do real, conforme visto na Figura 7.3.

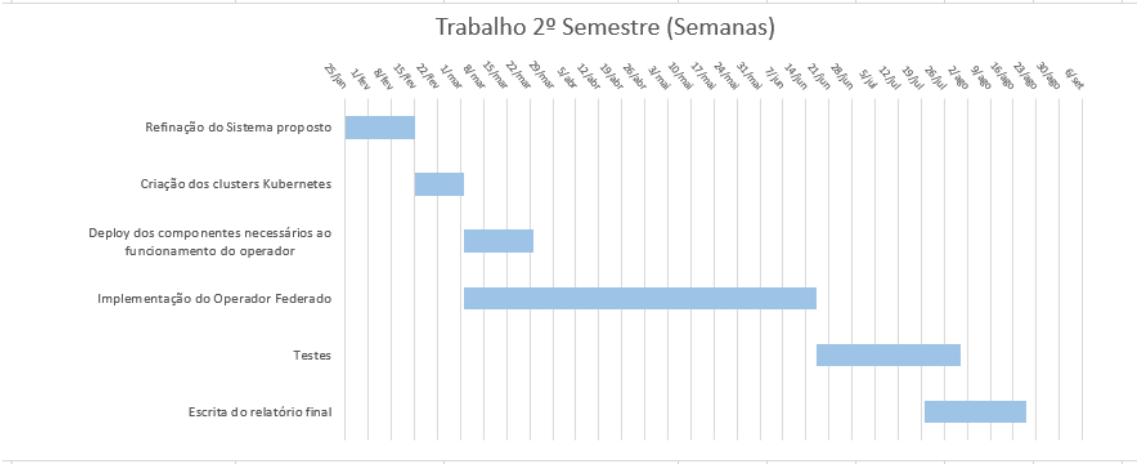


Figura 7.3: Diagrama de *Gantt* para o atual trabalho do 2º Semestre

O cronograma manteve a mesma ordem do inicialmente planeado, tendo havido, no entanto, atrasos em algumas tarefas, o que conduziu a um retardo temporal para a conclusão da dissertação. O primeiro atraso ocorreu na tarefa de refinação do sistema proposto, que tinha como objetivo melhorar a especificação do operador de forma mais concreta e aprofundada. Após uma investigação mais desenvolvida, foram levantadas várias questões como a eleição de líder, como seria efetuada a comunicação entre as instâncias do operador e como se iriam calcular os custos, o que exigiu a exploração de alguns algoritmos e ferramentas.

A tarefa de *deploy* dos componentes necessários ao funcionamento do operador, e aqui fala-se do *Nats*, *Zookeeper*, *Prometheus* e *Kubecost*, causou algumas dificuldades. O *deployment* de cada componente exigia muito conhecimento destes e,

como tal, foi necessário utilizar uma alternativa que facilitasse o *deployment* de todas estas *stacks*. Como alternativa, utilizou-se o *Helm* que ajudou neste processo.

Adicionalmente, também a tarefa de testes teve o seu período de tempo ligeiramente aumentado, devido ao teste de quase produção que demorava um período de tempo maior ter sido necessário repetir mais do que uma vez.

Por fim, o desenvolvimento do operador federado envolveu um prazo superior ao esperado devido à complexidade que exigiu.

Capítulo 8

Conclusão

As empresas estão sistematicamente à procura de melhorar os seus serviços e acompanhar a constante evolução das últimas tecnologias. Tal tem contribuído para que as arquiteturas *multi-cluster* se tornem cada vez mais uma tendência, muito devido aos benefícios que oferecem. Ainda assim, a complexidade que acarretam traz desafios significativos para a sua gestão.

Este trabalho focou-se então na abordagem dos desafios específicos associados à gestão de ambientes *multi-cluster*. Estes utilizam diversas vezes diferentes *clouds*, o que conduz a um ambiente *multi-cloud* que acrescenta uma ainda maior dificuldade na sua gestão, causada essencialmente pela heterogeneidade dos vários fornecedores de *cloud*. Focou-se também na manutenção de objetivos do nível de serviço (SLOs) para garantir um serviço confiável e de qualidade, aspetos deveras importantes neste contexto. Em resumo, dedicou-se a resolver os desafios críticos associados à gestão de aplicações num ambiente *multi-cluster* e *multi-cloud*, com um foco especial na manutenção de SLOs, devido à necessidade que existe na criação de soluções que permitam otimizar os custos, o desempenho, e a confiabilidade das aplicações.

O desenvolvimento deste operador federado para *Kubernetes*, orientado por um objetivo do nível de serviço (SLO) de custo e apoiado por tecnologias como *Kubecost*, *NATS* e *ZooKeeper*, demonstrou a viabilidade desta abordagem baseada em outras métricas que o *Kubernetes* não tem em conta, para otimizar os custos das aplicações em ambientes *multi-cluster* e *multi-cloud*. Desta forma, a aplicação do conceito de operador federado orientado por SLOs relevou-se uma estratégia eficaz para minimizar as suas violações. O operador desenvolvido tem a capacidade de monitorizar um SLO de custo, de reagir a eventuais violações do mesmo e de adaptar os *clusters* para as minimizar. A colaboração das diversas instâncias do operador federado em diferentes *clusters* evidencia o seu comportamento federado, onde decisões conjuntas resultam em uma gestão dos *clusters* orientadas por SLOs. Assim, este contribui para a noção de federação que o *Kubernetes* não possui, ao operar de maneira coordenada com os vários *clusters* para atender ao SLO.

Os resultados confirmaram que este operador é eficaz a monitorizar e ajustar dinamicamente a alocação de recursos entre *clusters*, mantendo a aplicação dentro

dos limites de custo acordados. Este operador destaca-se pela capacidade de resposta imediata à violação do SLO, automatizando a migração de aplicações entre *clusters*, quando necessário, eliminando a necessidade de intervenção humana na gestão deste ambiente complexo. O operador federado desenvolvido monitoriza constantemente o ambiente *multi-cluster* e *multi-cloud* para detetar violações do SLO, tomando medidas, neste caso, para mitigar o problema, como migrar a aplicação para um *cluster* mais adequado que cumpra o SLO de custo definido. Além disso, o operador federado faz parte do ecossistema *Kubernetes*, o que permite uma fácil integração com este, contribuindo para uma abordagem federada do *Kubernetes*.

Este trabalho não apenas resolveu um problema complexo, mas também abriu novas abordagens para o futuro. Nesta dissertação, criou-se um caso específico para provar o conceito de operador federado orientado por SLOs, ao implementar-se uma solução de gestão de ambientes *multi-cluster* e *multi-cloud* orientada por um SLO de custo, referente ao limite que uma aplicação pode custar mensalmente. Embora este seja um caso específico, esta abordagem orientada por SLOs pode servir de base para soluções mais complexas, visto que pode ser aplicada a várias métricas e outros cenários. À medida que as organizações continuam a adotar arquiteturas *multi-cluster* e *multi-cloud*, soluções como esta têm o potencial de desempenhar um papel crítico na garantia de otimização de custos, desempenho e eficiência das aplicações.

8.1 Dificuldades

Esta secção descreve as dificuldades encontradas durante o desenvolvimento do operador federado para *Kubernetes*, que tem como objetivo minimizar as violações de um SLO de custo para uma aplicação em ambientes *multi-cluster* e *multi-cloud*.

8.1.1 Inexperiência nas tecnologias

Uma das maiores dificuldades enfrentadas durante o desenvolvimento desta dissertação foi a necessidade de aprender rapidamente várias tecnologias e conceitos, das quais eu não tinha qualquer experiência como *Kubernetes*, operadores *Kubernetes* e *Operator Framework*, ao mesmo tempo que efetuava toda uma pesquisa relativa ao conceito de operador aliado a SLOs, o que me sobrecarregou bastante e exigiu um esforço extra para assimilar a informação de forma rápida, de modo a traduzi-la para uma solução prática.

8.1.2 Complexidade do Operador Federado

O operador federado projetado é de grande complexidade, o que provou ser bastante desafiador. Encontrar uma solução para provar o conceito de operador federado orientado a SLO foi difícil, perceber como garantir a sincronização das vá-

rias instâncias do operador em diferentes *clusters*, como seria a melhor forma de obter os custos da aplicação, como desenhar uma solução para que cada instância do operador fosse independente, de modo a que a falha de uma não comprometa o sistema, como proceder à eleição do líder e como resolver o problema de descobertas de serviços associado às várias instâncias do operador foi um processo difícil e moroso.

8.1.3 *Cluster* Adicional

Outra dificuldade foi apenas a existência de financiamento para a *cloud* Amazon Web Services (AWS), o que só me permitia utilizar o operador em ambiente *multi-cluster*. Como resultado, foi necessário recorrer às ofertas gratuitas de outras *clouds*, nomeadamente a Google. Devido às limitações do período de experimentação gratuito, foi necessária uma gestão cuidadosa dos recursos.

8.2 Considerações Finais

Para concluir, penso que este estágio foi realizado com sucesso, tendo sido provado o conceito de operador federado orientado por SLOs. Todas as funcionalidades foram implementadas e o sistema foi deixado num estado utilizável.

O desenvolvimento do operador federado permitiu-me experimentar e aplicar uma variedade de conceitos, desde a compreensão dos fundamentos da tecnologia *Kubernetes*, o conceito de operador *Kubernetes*, exploração de diversas soluções de *cloud* para *Kubernetes* e o conceito de federação aliado aos SLOs.

Por fim, a realização deste projeto foi bastante enriquecedora, mas também extremamente desafiante, pois não dispunha de qualquer experiência nas tecnologias utilizadas. Contudo, ao fim da realização desta dissertação, a experiência foi adquirida e será, indubitavelmente, uma mais-valia daqui para a frente.

References

[1] Sachchidanand Singh and Nirmala Singh. Containers & docker: Emerging roles & future of cloud technology. In *2016 2nd international conference on applied and theoretical computing and communication technology (iCATccT)*, pages 804–807. IEEE, 2016.

[2] Microservices vs monolith: Which is better - folio3 dynamics blog. `https://dynamics.folio3.com/blog/microservices-vs-monolith/`. (Accessed on 07/01/2023).

[3] Kubernetes. `https://kubernetes.io/`. (Accessed on 05/12/2022).

[4] Ying Mao, Yuqi Fu, Suwen Gu, Sudip Vhaduri, Long Cheng, and Qingzhi Liu. Resource management schemes for cloud-native platforms with computing containers of docker and kubernetes. *CoRR*, abs/2010.10350, 2020.

[5] Asif Khan. Key characteristics of a container orchestration platform to enable a modern application. *IEEE Cloud Comput.*, 4(5):42–48, 2017.

[6] Docker: Accelerated, containerized application development. `https://www.docker.com/`. (Accessed on 15/12/2022).

[7] What is container orchestration? | vmware glossary. `https://www.vmware.com/topics/glossary/content/container-orchestration.html`. (Accessed on 05/12/2022).

[8] What is container orchestration? `https://www.redhat.com/en/topics/containers/what-is-container-orchestration`. (Accessed on 05/12/2022).

[9] Nigel Poulton. *The kubernetes Book*. Independently published, London, 2021.

[10] Qingsong Jiao, Botong Xu, and Yunjie Fan. Design of cloud native application architecture based on kubernetes. In *IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress, DASC/PiCom/CBDCom/CyberSciTech 2021, Canada, October 25-28, 2021*, pages 494–499. IEEE, 2021.

[11] Fedor Y Chemashkin and Pavel D Drobintsev. Kubernetes operators as a control system for cloud-native applications. Technical report, Tech. rep., Peter the Great St. Petersburg Polytechnic University, 2021.

[12] Docker overview | docker documentation. https://docs.docker.com/get-started/overview/. (Accessed on 20/12/2022).

[13] Docker hub | docker. https://www.docker.com/products/docker-hub/. (Accessed on 25/12/2022).

[14] What is docker? | ibm. https://www.ibm.com/topics/docker. (Accessed on 20/12/2022).

[15] Borg: The predecessor to kubernetes | kubernetes. https://kubernetes.io/blog/2015/04/borg-predecessor-to-kubernetes/. (Accessed on 04/01/2023).

[16] Nassim Kebbani, Piotr Tylenda, and Russ Mckendrick. *The kubernetes Bible*. Packt Publishing Ltd, Birmingham, 2022.

[17] What is a kubernetes operator? https://www.redhat.com/en/topics/containers/what-is-a-kubernetes-operator. (Accessed on 10/12/2022).

[18] Operator pattern | kubernetes. https://kubernetes.io/docs/concepts/extend-kubernetes/operator/. (Accessed on 10/12/2022).

[19] Kubernetes operators: what are they? some examples | cloud native computing foundation. https://www.cncf.io/blog/2022/06/15/kubernetes-operators-what-are-they-some-examples/. (Accessed on 07/01/2023).

[20] Octavian Mart, Catalin Negru, Florin Pop, and Aniello Castiglione. Observability in kubernetes cluster: Automatic anomalies detection using prometheus. In *2020 IEEE 22nd International Conference on High Performance Computing and Communications; IEEE 18th International Conference on Smart City; IEEE 6th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 565–570. IEEE, 2020.

[21] Welcome to operator framework. https://operatorframework.io/. (Accessed on 10/12/2023).

[22] Michael Dame. *The Kubernetes Operator Framework*. Packt Publishing Ltd, Birmingham, 2022.

[23] Ruxiao Duan, Fan Zhang, and Samee U Khan. A case study on five maturity levels of a kubernetes operator. In *2021 IEEE Cloud Summit (Cloud Summit)*, pages 1–6. IEEE, 2021.

[24] Overview | prometheus. https://prometheus.io/docs/introduction/overview/. (Accessed on 01/01/2023).

[25] prometheus-operator/prometheus-operator: Prometheus operator creates/configures/manages prometheus clusters atop kubernetes. https://github.com/prometheus-operator/prometheus-operator. (Accessed on 16/01/2023).

[26] Helm. https://helm.sh/. (Accessed on 29/06/2023).

[27] Welcome to the docs! - kubecost documentation. https://docs.kubecost.com/. (Accessed on 26/07/2023).

[28] minikube start | minikube. https://minikube.sigs.k8s.io/docs/start/. (Accessed on 26/07/2023).

[29] kind. https://kind.sigs.k8s.io/. (Accessed on 26/07/2023).

[30] Serviço gerenciado do kubernetes – amazon eks – amazon web services. https://aws.amazon.com/pt/eks/. (Accessed on 26/07/2023).

[31] Serviço do kubernetes gerido (aks) | microsoft azure. https://azure.microsoft.com/pt-pt/products/kubernetes-service. (Accessed on 26/07/2023).

[32] Fast kubernetes clusters & deployments | linode, now akamai. https://www.linode.com/products/kubernetes/. (Accessed on 26/07/2023).

[33] Digitalocean kubernetes :: Digitalocean documentation. https://docs.digitalocean.com/products/kubernetes/. (Accessed on 26/07/2023).

[34] Google kubernetes engine (gke) | google cloud. https://cloud.google.com/kubernetes-engine?utm_source=google&utm_medium=cpc&utm_campaign=emea-pt-all-en-dr-bkws-all-all-trial-e-gcp-1011340&utm_content=text-ad-none-any-DEV_c-CRE_529445694694-ADGP_Hybrid%20%7C%20BKWS%20-%20EXA%20%7C%20Txt%20~%20Containers%20~%20Kubernetes%20Engine&hl=pt-br#v1-KWID_43700060416641562-aud-488003287395:kwd-920676122-userloc_1011721&utm_term=KW_gke-NET_g-PLAC_&gad=1&gclid=Cj0KCQjwiIOmBhDjARIsAP6YhSWAyTAhm_vTepV7lkS4oPraqQ2OEa-HwxqvH9a7VlZkOzmAr75-yMsaAlQIEALw_wcB&gclsrc=aw.ds&hl=pt-br. (Accessed on 26/07/2023).

[35] eksctl. https://eksctl.io/. (Accessed on 08/08/2023).

[36] Advanced load balancer, web server, & reverse proxy - nginx. https://www.nginx.com/. (Accessed on 08/08/2023).