Faculdade de Ciências e Tecnologia

Departamento de Engenharia Informática

Extending the Elastic Microservices Platform

António Miguel Guedes de Campos dos Santos Sequeira

Dissertação no contexto do Mestrado em Engenharia Informática, Especialização em Engenharia de Software orientada pelo Professor Filipe João Boavida Mendonça Machado de Araújo apresentada à Faculdade de Ciências e Tecnologia / Departamento de Engenharia Informática

Janeiro 2020

UNIVERSIDADE D
COIMBRA

Esta página foi intencionalmente deixada em branco.

Resumo

A utilização de microserviços como forma de construir aplicações veio mitigar muitos dos problemas apresentados pelos monólitos. No entanto trouxe outros, como a dificuldade de monitorizar e gerir.

As soluções existentes atualmente para a realização de *deploy* e gestão de microserviços não apresentam funcionalidades de monitorização e de gestão automática como resposta a acontecimentos imprevistos, tendo que haver intervenção por parte dos utilizadores para se conseguirem as capacidades escaláveis e elásticas.

Neste documento, a plataforma que se descreve tem como objetivo corrigir os problemas que as soluções existentes apresentam, recorrendo à recolha de métricas e *traces* como ponto de partida para uma análise automática, que permitirá a realização de escalabilidade e elasticidade automáticas, tendo em conta também as exigências do utilizador da plataforma em relação à sua aplicação. Permitirá ainda a apresentação visual de métricas ao utilizador, para que este possa acompanhar o que acontece na sua aplicação.

Palavras-Chave

Microserviços, Monitorização, Elasticidade, Escalabilidade, Rastreamento Distribuído

Esta página foi intencionalmente deixada em branco.

Abstract

The use of microservices as a mean of building applications has mitigated many of the problems presented by monoliths. However, it has brought others, such as the difficulty of monitoring and managing.

The existing solutions currently available for the deployment and management of microservices do not have monitoring and automatic management functionalities as a response to unforeseen events, and users must intervene to achieve scalable and elastic capabilities.

In this document, the platform that is described aims to correct the problems that the existing solutions present, using the collection of metrics and traces as a starting point for an automatic analysis, which will allow the achievement of automatic scalability and elasticity, also taking into account the requirements of the platform user in relation to its application. It will also allow the user to visually see metrics, so that they can follow what is happening in their application.

Keywords

Microservices, Monitoring, Elasticity, Scalability, Distributed Tracing

Esta página foi intencionalmente deixada em branco.

Agradecimentos

A realização desta tese não seria possível sem a ajuda de diversas pessoas, às quais eu não poderia deixar de mostrar a minha gratidão.

Agradeço em primeiro lugar ao meu orientador, Professor Filipe Araújo. Sem a sua orientação e ajuda, a conclusão da plataforma desenvolvida, bem como a escrita deste relatório não teriam chegado a bom porto.
Deixo ainda uma palavra de agradecimento aos meus colegas, alunos de doutoramento, Jaime Correia e André Bento pela paciência e disponibilidade nas inúmeras vezes que se reuniram comigo para me ajudarem a resolver problemas e ultrapassar obstáculos.
Agradeço ainda aos meus amigos e colegas de mestrado Joel Fernandes e João Ferreira, pela ajuda prestada e pelo apoio que me deram ao longo dos dois semestres da tese.
Queria ainda agradecer aos meus pais e às minhas irmãs pela força e incentivo constantes ao longo desta minha jornada em Coimbra.
Não podia faltar um agradecimento à minha namorada, Juliana Campos, pois sem o seu amor e carinho não teria conseguido chegar tão longe.

Expresso também a minha gratidão ao INCD - Infraestrutura Nacional de Computação Distribuída, pelo acesso fornecido aos seus recursos computacionais.

Esta página foi intencionalmente deixada em branco.

Conteúdo

Lista de Figuras

Esta página foi intencionalmente deixada em branco.

Lista de Tabelas

Esta página foi intencionalmente deixada em branco.

Capítulo 1

Introdução

O documento apresentado corresponde ao relatório final que expõe o trabalho desenvolvido pelo aluno António Miguel Guedes de Campos dos Santos Sequeira nos dois semestres da Tese de Mestrado em Engenharia Informática, Especialização em Engenharia de Software, orientada pelo Prof. Filipe Araújo, com lugar no Departamento de Informática, na Faculdade de Ciências e Tecnologias da Universidade de Coimbra.

Este trabalho foi parcialmente realizado no âmbito do projeto PTDC/EEI-ESS/1189/2014 - Data Science for Non-Programmers, suportado pelo COMPETE 2020, Portugal 2020-POCI, UE-FEDER e a FCT.

1.1 Motivação

Os microserviços são a resposta aos problemas apresentados pelos monólitos. Os monólitos são e sempre foram a estrutura mais comum na construção de *software*. No entanto, com o passar dos anos, as aplicações necessitaram se de tornar cada vez maiores para acomodar um maior número de funcionalidades e satisfazer requisitos mais rígidos. Isto levou a um aumento exponencial da complexidade do *software*, principalmente dos monólitos. Por esta razão, houve a necessidade de construir estas aplicações de maneira diferente, para garantir que certos aspetos relacionados com a construção, gestão e manutenção destas não são afetadas.
A solução mais utilizada para combater o acima descrito, são os microserviços. Consiste em dividir a aplicação em *software* mais pequenos que funcionam em conjunto, mas que, no entanto, não têm que ser embalados juntos. Para uma descrição mais detalhada da arquitetura de microserviços, ver a secção 2.1.

O aparecimento dos microserviços que veio mitigar os problemas dos monólitos é algo relativamente recente. A divisão dos monólitos em microserviços trouxe várias vantagens a nível de produção e utilização. No entanto, fazer uma gestão adequada destes de forma automática revelou-se um dos principais desafios. Atualmente, várias das ferramentas existentes para fazer esta gestão são proprietárias, e na sua maioria necessitam de intervenção do utilizador para definir parâmetros para que possam ser utilizados como base para a gestão, não havendo uma análise automática do que acontece com a aplicação de microserviços. Assim, existe um buraco no mercado para uma plataforma de gestão de aplicações de microserviços, que através da recolha e análise de *traces* e/ou métricas encontra os problemas das aplicações, conseguindo geri-los de maneira a que estas possam ter a maior disponibilidade e desempenho possíveis.

1.2 Objetivos

Com a tese de mestrado pretende-se, numa primeira fase, analisar conceitos, ferramentas e soluções já existentes, para de seguida se definir o que se pretende fazer, mostrando a sua relevância. Esta dissertação teve como objetivo final a implementação de uma plataforma de gestão de aplicações de microserviços, que através da análise de métricas e *traces* retirados da aplicação, permite tomar decisões de maneira automática relativas a como escalar os microserviços. Para se chegar a este ponto, é preciso realizar as seguintes etapas, descritas de seguida.

- Utilização de uma ferramenta de monitorização para se obterem métricas
- Implementação de um serviço de tracing para a obtenção de *traces*
- Utilização de uma ferramenta de visualização de modo a permitir uma representação visual das métricas tiradas
- Utilização de armazenamento permanente para acesso a dados posteriormente à sua recolha
- Criação de um algoritmo para tomada de decisão sobre a escalabilidade da aplicação, embora de maneira rudimentar, pois este algoritmo cai fora do *scope* da tese
- Criação de um controlador da aplicação de maneira a controlar e realizar alterações na aplicação

Após a realização do descrito anteriormente, foi preciso proceder à escrita do relatório final, onde é descrito tudo o que foi feito ao longo do ano de tese.

1.3 Resultados

O trabalho realizado no âmbito desta tese de mestrado satisfaz quase totalmente os objetivos propostos na secção anterior. Foi implementada uma plataforma de gestão de microserviços que consegue efetivamente realizar alterações na aplicação que se monitoriza, reagindo a dados retirados através de métricas.

Esta plataforma faz uso da tecnologia *Kubernetes* como orquestrador de microserviços, sobre o qual os restantes componentes são implementados. Para realizar o encaminhamento de pedidos, bem como para conseguir a monitorização necessária para a obtenção de métricas é utilizada a tecnologia *Istio*, uma *service mesh* que coloca *sidecars* entre os microserviços e os pedidos, ajudando no *load balancing* bem como registando os pedidos realizados a cada um destes microserviços. Estas métricas retiradas podem ser visualizadas numa ferramenta de visualização denominada *Grafana*, que permite que o utilizador observe dados sob a forma de gráficos e tabelas, dando assim possibilidade ao utilizador de compreender o que está a acontecer com a sua aplicação. Estes dados são armazenados de forma permanente numa base de dados de séries temporais denominada *InfluxDB*. A utilização desta base de dados é o que permite que o utilizador possa consultar os dados a longo prazo. Através das métricas retiradas, um controlador implementado em *Flask*, utilizando um algoritmo de decisão *dummy* criado como prova de conceito, pois este algoritmo foge ao *scope* desta tese, controla a aplicação de microserviços, escalando o número de instâncias do microserviço que precisa de ser alterado.

1.4 Planeamento

Nesta secção irá ser abordado como é que o trabalho se desenrolou ao longo dos dois semestres, mostrando a divisão do tempo das tarefas realizadas.

Relativamente ao primeiro semestre, o trabalho pode ser descrito pelo diagrama de *Gantt* da figura 1.1.

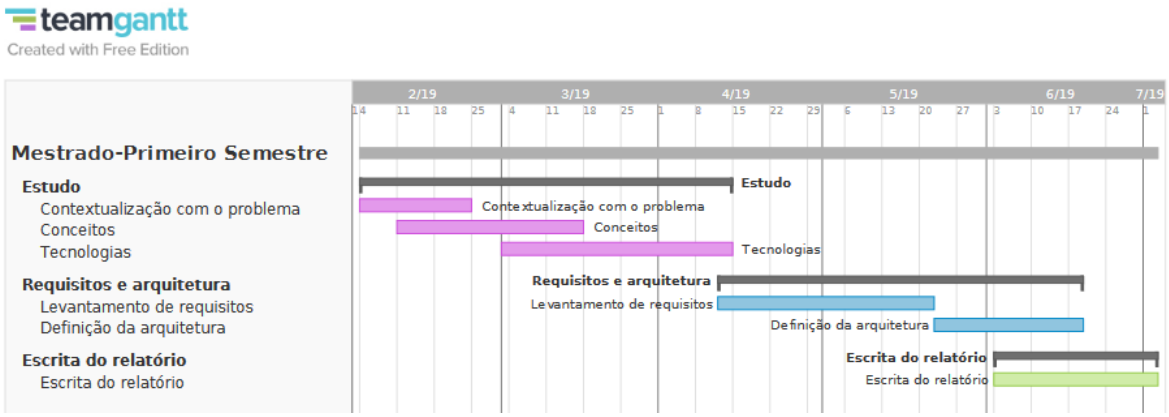

Figura 1.1: Diagrama de *Gantt* do trabalho desenvolvido no primeiro semestre

O primeiro semestre começou com a contextualização com o problema, para uma melhor compreensão do que iria ser necessário fazer no semestre. De seguida foi preciso estudar alguns conceitos fundamentais para ter um melhor entendimento dos temas que foram abordados durante a contextualização. Ao mesmo tempo que se fazia este estudo, algumas tecnologias foram começando a ser analisadas, conforme estas estavam ligadas aos conceitos estudados. Posteriormente, procedeu-se ao levantamento de requisitos funcionais e atributos de qualidades desejados para a plataforma. Por fim, foi feita a definição da arquitetura da plataforma, juntamente com a escrita do relatório intermédio.
Para uma constante orientação, acompanhamento e monitorização do progresso ao longo do primeiro semestre foram realizadas reuniões a cada duas semanas, em que se encontravam presentes eu, António Sequeira, o Professor Filipe Araújo, o Professor Jorge Cardoso por chamada telefónica, o estudante de doutoramento Eng. Jaime Correia, e os colegas André Bento e Joel Fernandes.

No segundo semestre as reuniões continuaram a realizar-se bissemanalmente, com a presença dos mesmos intervenientes, exceptuando o Professor Jorge Cardoso. Estas reuniões foram essenciais para a definição do rumo a seguir, bem como para a resolução de problemas e esclarecimento de dúvidas.
Numa tentativa de espelhar o trabalho realizado ao longo do segundo semestre, o seguinte diagrama de *Gantt* foi criado (figura 1.2).

O segundo semestre começou com a criação de uma conta *Google* para ser possível utilizar o *Google Kubernetes Engine*. Após a concretização deste passo, seguiu-se a implementação do *Istio*, uma *service mesh* que permite o redirecionamento de pedidos, através de *load balancing* por exemplo, bem como o registo dos acontecimentos na aplicação de microserviços que se pretende monitorizar e controlar. Para testar o funcionamento do *Istio* foi utilizada uma aplicação de microserviços disponibilizada pelos mesmos, a aplicação *Book-Info*.
O passo seguinte foi a apresentação ao utilizador das métricas retiradas. Para isso foi utilizada um tecnologia de *dashboards* denominada *Grafana*, que configurada com diversos *dashboards* apresenta os dados de maneira intuitiva e fácil de compreender como analisado na secção 5.7.1.

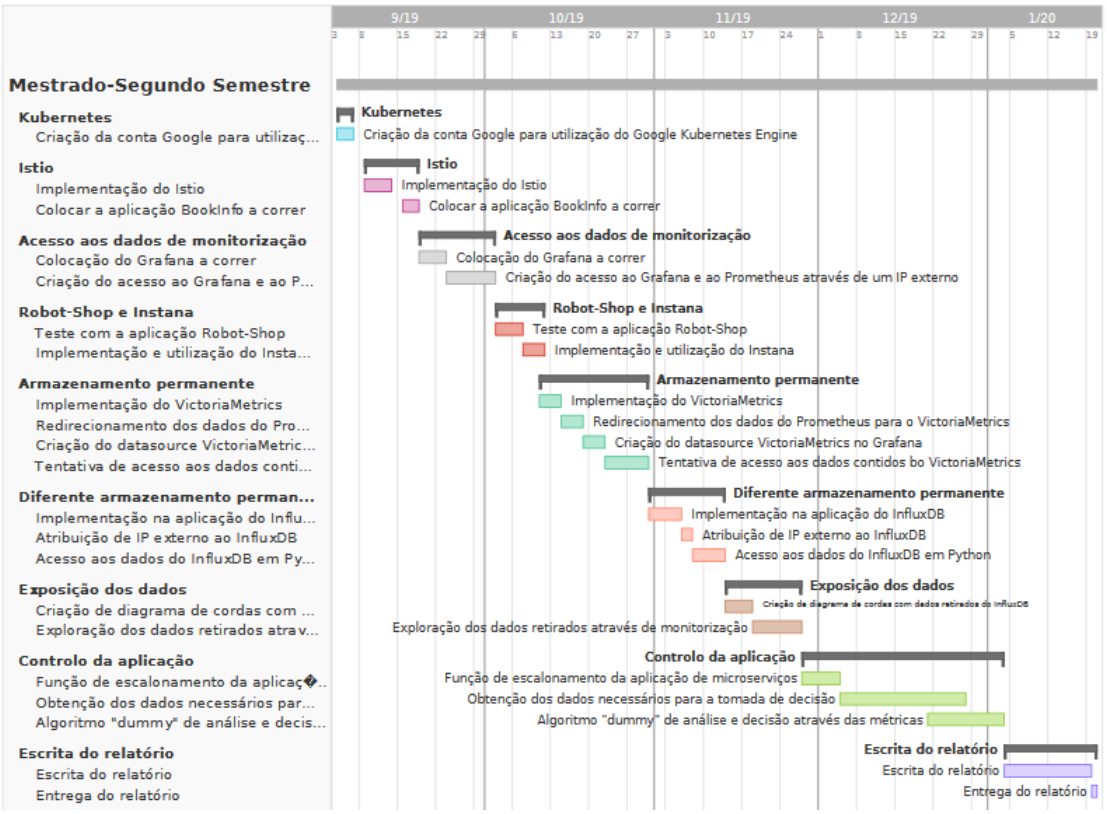

Figura 1.2: Diagrama de *Gantt* do trabalho realizado no segundo semestre

Após se ter verificado que o sistema estava a funcionar, e que as métricas estavam a ser retiradas e apresentadas corretamente, era necessário perceber se o *Istio* estava a funcionar por ser com sua aplicação de teste, ou se o que tinha sido implementado até àquele momento funcionaria com outra aplicação de microserviços, e já que se fazia a troca, com uma aplicação maior e mais complexa. Assim foi utilizada a aplicação de microserviços *Robot-Shop* que se trata de uma aplicação mais complexa, como foi analisado na secção 5.2. Juntamente com esta aplicação foi testada a ferramenta *Instana*, que acabou por ser descartada como parte constituinte da solução final.
De seguida foi preciso armazenar os dados de forma permanente. Para isso começou-se por usar uma base de dados denominada *VictoriaMetrics*. Após a sua implementação, bem como o redirecionamento dos dados do *Prometheus* para a base de dados, não era possível aceder a estes dados para a análise e posterior decisão para controlar a aplicação. Assim, foi necessário procurar outra solução de armazenamento, tendo sido escolhido o *InfluxDB*. Após a sua implementação, e tendo sido feito o redirecionamento dos dados para esta base de dados, foi criada a função de acesso aos dados em *Python*.
Foi ainda criada uma outra maneira de visualizar os pedidos feitos dentro da aplicação, através da utilização de um diagrama de cordas. Isto permite ter uma visualização mais clara de como os microserviços se ligam.
Em penúltimo lugar foi preciso criar o controlo da aplicação, através de funções implementadas em *Python*, para ser possível tanto a tomada de decisão em relação ao escalonamento da aplicação, bem como a alteração efetiva do número de instâncias do microserviço a modificar.
Por fim, foi redigido o presente relatório.

1.5 Organização do documento

O presente relatório encontra-se organizado da seguinte maneira:

- Capítulo 2 - Conceitos fundamentais
- Capítulo 3 - Trabalho Relacionado
- Capítulo 4 - Descrição da Arquitetura
- Capítulo 5 - Implementação
- Capítulo 6 - Resultados
- Capítulo 7 - Conclusão e Trabalho Futuro

No Capítulo 2 são analisados os conceitos fundamentais para a correta compreensão do problema a que se pretende dar resposta, juntamente com a exploração de algumas soluções e tecnologias para um melhor entendimento dos conceitos associados a estas, bem como para a sua possível utilização na solução final.

O Capítulo 3 serve para descrever os trabalhos desenvolvidos pelos colegas Fábio Pina e Fábio Ribeiro, bem como outras soluções, pois os problemas que se visaram resolver com esses projetos estão interligados com o problema apresentado neste relatório.

No Capítulo 4 são explicitados os requisitos funcionais, os atributos de qualidade e ainda a arquitetura da plataforma a desenvolver no segundo semestre.

O Capítulo 5 refere-se ao trabalho desenvolvido no segundo semestre da presente tese. Consta deste uma descrição detalhada de tudo o que foi implementado, dando uma visão geral mas detalhada do resultado final.

No Capítulo 6 foi feita uma explicação de como é que se procedeu à avaliação da plataforma, de forma a demonstrar que as funcionalidades pretendidas estavam efetivamente implementadas.

Por fim, no Capítulo 7 é feito um resumo do documento, bem como tecidos alguns pensamentos sobre o trabalho desenvolvido, e futuro trabalho a desenvolver.

Capítulo 2

Conceitos Fundamentais

Nesta secção, serão abordados vários conceitos relevantes para o trabalho, tais como o que são Microserviços e Service Mesh, Monitorização (Rastreamento distribuído, Monitorização de registos), Contentores e *Cluster Managers*.

2.1 Arquitetura de Microserviços

Microserviços são a mais recente tendência no desenvolvimento e entrega de serviços de software. Estes constituem uma abordagem de arquitectura de sistemas e *software*, que se apoia num conceito bem estabelecido de modularização, mas que enfatiza fronteiras técnicas. Cada módulo - cada microserviço - é implementado e operado como um sistema pequeno mas independente, oferecendo acesso à sua lógica e dados internos através de um interface bem definida. Isto aumenta a agilidade de *software* porque cada microserviço se torna uma unidade independente de desenvolvimento, operações, versão e escalabilidade.[1] Esta arquitetura surgiu de uma necessidade de colmatar a falta de maneiras de desenhar arquiteturas que funcionassem bem para sistemas muito complexos e de tamanho elevado. As linguagens de programação mais usadas no desenvolvimento de aplicações, como Java, C/C++ e Python, estão pensadas de maneira a levar à criação de artefactos executáveis singulares, também conhecidos como monólitos. As suas abstrações modulares dependem da divisão de recursos da mesma máquina. Assim, devido a esta dependência, não podem ser executados independentemente.[2] Uma solução monolítica, é um sistema que tem algumas vantagens associadas, mas que, no entanto, tem muitas desvantagens. Fala-se então a seguir de algumas destas vantagens e desvantagens.

As vantagens que estão associadas a esta solução monolítica prendem-se com a simplicidade com que a aplicação resultante é concebida, e consequentemente a facilidade de testar tal aplicação. Também o *deploy* desta aplicação se torna mais simples, pois esta encontra-se concentrada num só pacote, não tendo que haver preocupação com a orquestração de serviços. Por fim, a escalabilidade horizontal desta aplicação torna-se direta, correndo várias cópias atrás de um *load balancer*.
Porém, muitas desvantagens vêm acopladas. As desvantagens que são a seguir referidas fazem com que este tipo de arquitetura seja cada vez mais colocada de lado quando se pensa em desenvolver um *software*, principalmente um em que o tamanho seja grande. Novas exigências na maneira de pensar no desenvolvimento destas aplicações levaram à criação de novos modos de olhar para os requisitos, fomentando o aparecimento de novos *design patters*, como os Microserviços. Este tipo de arquitetura vem colmar muitos dos problemas relacionados com a "arquitetura monolítica".

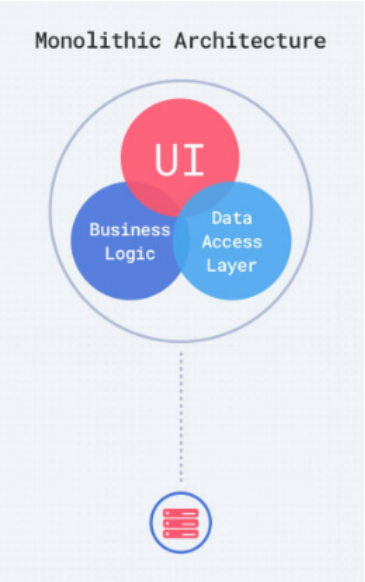

Figura 2.1: "Arquitetura monilítica" [3]

Primeiramente, este tipo de aplicação, baseada numa "arquitetura monolítica", sofre de dificuldades na sua manutenção. Conforme cresce e fica mais complexa, a aplicação deixa de ser totalmente compreensível, o que faz com que a descoberta e posterior correção de erros ou a realização de alterações necessárias seja mais difícil de fazer rapidamente e de maneira correta. Esta é a primeira área onde os Microserviços têm vantagem. Cada microserviço é pequeno, e focado em fazer tarefas específicas. O desenvolvimento que é seguido para os microserviços vai ao encontro da definição de Robert C. Martin, *Single Responsibility Principle*, que estabelece que "Junte-se o que muda pela mesma razão, e separe-se o que muda por razões distintas".[4] Assim, a correção de erros ou a implementação de atualizações nos microserviços torna-se muito mais rápida e eficaz, pelo maior conhecimento do código por parte das equipas de desenvolvimento, e pelo reduzido tamanho do microserviço em comparação a uma "arquitetura monolítica".
O aumento desmesurado de uma solução monolítica leva a que o *start-up time* seja bem mais elevado do que o de serviços pequenos. A facilidade e rapidez de realizar o *deploy* do microserviço faz com que esta tarefa seja menos complicada de concretizar.
Ainda associado ao que foi dito anteriormente, num monólito, atualizações que sejam feitas obrigam a um *redeploy* de todo o sistema, o que associado ao ponto anterior, faz com que seja um processo complexo. Pelo contrário, a modularidade com que se faz a introdução de alterações nos microserviços, apenas obrigando ao *redeploy* do microserviço alterado, leva a que este processo seja simples e de reduzido consumo temporal.
Em relação à fiabilidade nestes monólitos, esta é dependente de qualquer módulo da aplicação, pois um módulo com um erro pode levar o sistema inteiro abaixo. E a disponibilidade é também afetada, pois caso se dê o referido no ponto anterior, poderá acontecer em todas as cópias por de trás do *load balancer*, deixando o sistema de estar disponível. Analisando agora os microserviços, a modularidade com que se faz *deploy* destes leva a que a fiabilidade seja muito mais elevada. O sistema não fica dependente de cada microserviço, e mesmo que haja uma falha num destes, o sistema não falha como um todo. Isto leva, por consequência, a uma maior disponibilidade do sistema.
Finalmente, a aplicação resultante deste tipo de desenvolvimento monolítico tem dificuldades em se adaptar a novas tecnologias, pois está muito dependente de *frameworks* e linguagens de programação, pelo que não é fácil alterá-la.[5] Este problema não se coloca em relação aos microserviços. Sendo cada microserviço independente, apenas criando cone-

xão com outros microserviços através de uma interface, estes não têm requisitos em relação à tecnologia ou linguagem de programação utilizadas. Assim, não se encontram limitados a uma estrutura específica, podendo cada equipa de desenvolvimento escolher o que melhor se adapta à função que o serviço precisa de desempenhar.

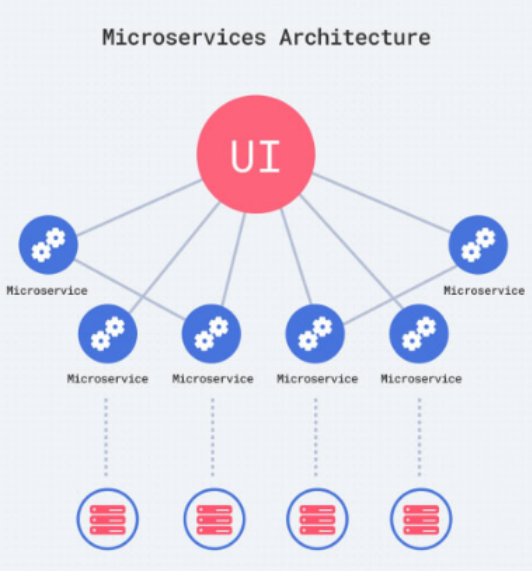

Figura 2.2: Arquitetura de microserviços [3]

No entanto, os microserviços não tem só vantagens, havendo alguns pontos negativos a evidenciar.
A criação de uma aplicação baseada nesta arquitetura leva, obrigatoriamente, a um sistema mais complexo, pois este encontra-se distribuído, e cada microserviço tem que ligar a outros. Isto faz também com que realizar testes neste tipo de sistemas se torne uma tarefa mais complicada.
Devido à modularidade, e à distribuição desses módulos, os *developers* são obrigados a implementar um serviço de comunicação entre microserviços para que estes possam comunicar.
Regra geral, um sistema composto por microserviços irá necessitar de uma maior quantidade de recursos, isto porque cada microserviço poderá estar num contentor individual, até para garantir o isolamento de cada instância, o que leva a um maior uso de recursos[5].
Assim, pode-se concluir que existem vantagens e desvantagens associadas aos dois tipos de sistemas. A escolha de qual usar vai depender de cada situação em concreto, sendo que para um sistema muito complexo e de tamanho elevado se aconselham microserviços, e para um mais pequeno e menos complexo, uma solução monolítica poderá ser a escolha mais lógica.

2.2 Contentores

Como falado na secção anterior, os microserviços baseiam-se na ideia da separação de serviços, tornando o código mais organizado, e com código mais especializado em cada microserviço, havendo também uma separação de responsabilidades. Para conseguir esta separação de serviços são utilizados *containers* (contentores). Contentores são um mecanismo de virtualização leve, autónomo e executável que serve para embalar a aplicação, para ajudar na distribuição dessa aplicação e é parte integrante na orquestração de microserviços[6]. Nos contentores faz-se a junção do código, das configurações e ferramentas

necessárias e das dependências exigidas. Isto permite fazer implementações rápidas e confiáveis, independentemente do ambiente[7].

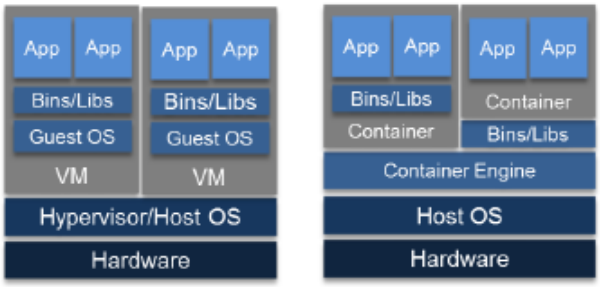

Figura 2.3: Máquinas virtuais *vs* Contentores [8]

Os contentores correm sobre o sistema operativo, pelo que vários contentores na mesma máquina partilham o OS, fazendo com que estes precisem de menos espaço, ganhando também na velocidade com que realizam as operações, pois não necessitam de fazer a tradução de código máquina para o OS.
Assim, os contentores apareceram para tentar ser uma solução mais viável do que, por exemplo, máquinas virtuais. De seguida será feita uma comparação entre estas a nível de desempenho, segurança, tempo de inicialização, armazenamento e sistema operativo.
A nível do sistema operativo, cada máquina virtual corre no seu hardware virtualizado, e o *Kernel* é carregado para a sua própria memória. Ao revés, os contentores partilham o mesmo sistema operativo e *Kernel*, sendo que a imagem do *Kernel* é carregado para memória física. Isto quer dizer que poderá apenas ser necessário um sistema operativo, sendo por isso menos pesado.
Relativamente à segurança, nas máquinas virtuais esta depende da implementação do *Hypervisor*. Nos contentores, controlo de acesso mandatório pode ser alterado.
No que diz respeito ao desempenho, as máquinas virtuais sofrem de uma carga vinda do necessidade de traduzir as instruções máquina do hóspede para o sistema operativo, enquanto que os contentores permitem performance quase nativa comparado com o sistema operativo *host* subjacente.
A inicialização destes componentes deve ser o mais rápido possível, para garantir que o sistema é iniciado sem atrasos. Assim, as máquinas virtuais não são a opção preferencial, pois estas podem demorar minutos a inicializar. Já os contentores podem ser ligados em segundos, sendo a escolha preferencial.
Por fim, a armazenamento necessário para correr os componentes é uma característica importante a ter em conta. Como dito em cima, as máquinas virtuais correm o seu sistema operativo, e todas têm que ter este completo, juntamente com os programas necessários que têm que ser instalados em cada máquina virtual, levando a uma necessidade de maior armazenamento. Em contraste, os contentores precisam de menos espaço, pois estes partilham o sistema operativo[9].
Assim, pode-se concluir que os contentores apresentam melhores características, sendo uma solução mais leve e fácil de orquestrar, justificando a sua escolha sobre as máquinas virtuais.

2.3 Monitorização

A monitorização é um dos mais importantes componentes associados aos microserviços. Para uma análise do que se passa com a arquitetura enquanto esta corre é preciso retirar

métricas que possam ser analisadas.
É prática comum em monitorização de *clouds* reunir métricas de baixo nível de sistemas virtuais tais como o uso de CPU e RAM, bem como métricas de mais alto nível e métricas especificas à aplicação tais como tempos de resposta e capacidade de resposta aos pedidos[10]. No entanto, esta informação pode ser um pouco limitante na informação que nos fornece, pois só algumas conclusões podem ser retiradas das métricas.
Assim, existem várias maneiras de fazer a monitorização de uma arquitetura de microserviços, as quais irão ser abordadas aseguir.

2.3.1 Rastreamento distribuído

Rastreamento distribuído (*Distributed Tracing*) é um método utilizado para criar um perfil e monitorizar aplicações, especialmente aplicações construídas como microserviços. Este método permite localizar com mais precisão onde ocorrem falhas e quais são as causas de baixo desempenho.[11] Trata-se do processo de rastrear a atividade resultante de um pedido à aplicação. Com este método consegue-se [12]:

- Ver o caminho que um pedido percorre enquanto viajou pelo sistema, sendo este sistema muitas vezes complexo
- Saber a latência dos componentes durante o caminho percorrido
- Saber que componente no caminho cria estrangulamento no sistema

Cada rastreamento (*trace*) permite ver o caminho percorrido pelo pedido, e é constituído por múltiplos *spans*, que representam o tempo despendido em serviços ou recursos desses serviços. Mais especificamente, um *span* tem informações como a hora a que o pedido começou e acabou, o nome do microserviço utilizado, a função que foi executada e mais informações se for necessário.
A título de exemplo, tenha-se em conta um serviço que tem como objetivo apresentar mensagens quando o utilizador se conecta. Na figura 2.4 podemos observar um *trace* com diversos *spans* correspondentes ao mesmo. Para responder ao serviço "/messages", vários outros pedidos internos são feitos. Neste caso, primeiramente tem que se verificar se o utilizador se encontra autenticado através do pedido "auth". De seguida, verifica-se se as mensagens se encontram em *cache*, através do "cache.Get". Neste exemplo específico, as mensagens não se encontravam em *cache*. Por isso é preciso fazer um pedido *MySQL* para se ter acesso a estas, com "mysql.Query". Por fim, guardam-se as mensagens em *cache* com "cache.Put" e responde-se ao pedido inicial.[13]
É possível observar pelo exemplo anterior, que um *trace* mantém um registo do caminho percorrido pelo pedido.

Assim, tem-se acesso a um conjunto mais detalhado de informações sobre o que se passa dentro dos microserviços, permitindo que se consiga realizar uma análise informada sobre o que é necessário mudar no comportamento, ou onde é preciso escalar o sistema para lidar com cargas elevadas de pedidos.
De seguida serão abordadas algumas das tecnologias existentes para a recolha de *traces*.
No entanto estas tecnologias obrigam a uma instrumentação dos microserviços, pois é cada microserviço que é responsável por enviar as informações para um ponto de recolha, para que estas possam ser agregadas e processadas, e posteriormente armazenadas, acrescentando assim uma responsabilidade aos microserviços, não sendo o rastreamento distribuído uma técnica desprovida de peso para os microserviços.

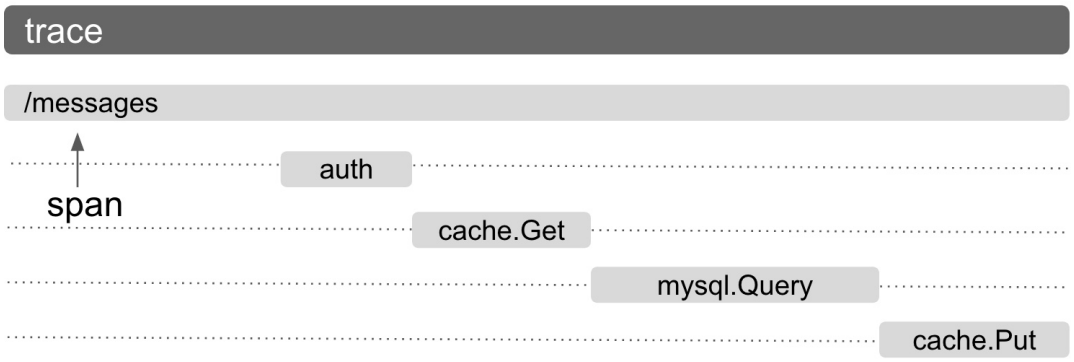

Figura 2.4: *Trace* [13]

OpenCensus

O *OpenCensus* "torna fácil a obtenção de telemetria crítica de serviços. O *OpenCensus* atualmente fornece bibliotecas para várias linguagens que permitem capturar, manipular e exportar métricas e *traces* para o(s) *backend*(s) à sua escolha."[14]
Para entender melhor como funciona esta solução, algumas das funcionalidades que esta oferece irão ser abordadas.

Funcionalidades

- Propagação de Contexto
- Recolha de *Traces*
- Recolha de Métricas de Séries Temporais
- *APIs*

A "Propagação de Contexto" é a perpetuação de um ID de correlação ou o formato de *trace* que está a ser utilizado, de modo a interligar atividades e pedidos entre serviços a partir de uma ação inicial do cliente. No entanto, conseguir realizar esta perpetuação automaticamente mostra-se desafiante, fazendo com que a propagação automática de contexto seja apontado como um dos principais obstáculos à utilização de rastreamento distribuído. O *OpenCensus* consegue fornecer esta funcionalidade automática nas linguagens e infraestruturas suportadas, e também guarnece os *developers* de uma *API* simples para propagação manual e manipulação de contexto.[14]
Relativamente ao segundo ponto, a solução *OpenCensus* recolhe e propaga *traces* pelo sistema, permitindo visualizar como é que os pedidos dos clientes viajam pelos serviços, fazendo rapidamente uma análise profunda da raiz do problema e permitindo observar melhor a latência num conjunto distribuído de serviços.[14]
O *OpenCensus* retira também estatísticas de séries temporais, incluindo a latência, o número de pedidos e o tamanho do pedido para cada *endpoint*. Uma vez captadas, estas estatísticas podem ser agregadas em métricas com janelas temporais.[14]
APIs para todo o tipo de telemetria são também disponibilizadas. A título de exemplo, o utilizador pode definir e captar métricas especificas, adicionar spans ou anotações aos *traces*, mudar contextos de propagação, entre outros.[14]
Após a exposição destas funcionalidades, é possível perceber que esta é uma solução que se encontra bastante desenvolvida, e que permite a integração desta em diversos tipos de linguagens e infraestruturas.

OpenTracing

OpenTracing é uma especificação *API*, infraestruturas e livrarias que implementam essa especificação, e documentação para o projeto. Esta ferramenta permite aos *developers* adicionar instrumentação ao código das suas aplicações usando *APIs* que não os prendem a nenhum produto ou vendedor em particular.[11]
Esta ferramenta traz funcionalidades de rastreamento distribuído que vão ao encontro às oferecidas pelo *OpenCensus*.
O *OpenTracing* permite a recolha de *traces*, constituídos por *spans*, para a aquisição de informação sobre os pedidos nos serviços. De maneira semelhante à ferramenta anterior, o *OpenTracing* recorre à propagação de contexto para a serialização de informação.
As funcionalidades são disponibilizadas pelas *OpenTracing APIs*, que se encontram em *packages* públicas para diferentes tipos de linguagens de programação. As principais construções que são disponibilizadas ao utilizador final são:

- *ITracer* - a interface que expõem todas a capacidades de rastreamento distribuído diretamente ao utilizador final [15]
- *ISpanBuilder* - a interface construtora que utilizada para criar e iniciar instâncias de *ISpans* utilizadas para recolher dados relativos a operações específicas [15]
- *ISpan* - a estrutura de dados utilizada para conter e modificar todos os dados contextuais relacionados com o *span* [15]
- *IScopeManager* - a ferramenta utilizada para ajudar na correlação da actividade e ocorre assincronamente dentro de um processo individual [15]

2.3.2 Monitorização de registos

A monitorização de registos (*Log Monitoring*) é a ação de rever registos retirados do sistema enquanto estes estão a ser retirados. Normalmente isto envolve a utilização de uma ferramenta de *software* de análise. *Software* de gestão de registos podem ser configurados para registar eventos específicos relacionados com a aplicação, e alertar a equipa de desenvolvimento que algo se passou.[16]
No entanto, seria interessante fazer com que esta análise leva-se à resolução automática de problemas, em especial em microserviços, podendo repor serviços que morreram, ou aumentar automaticamente o número de instâncias em caso de carga elevada.

2.4 *API Gateway*

Uma *API Gateway* "pega em todos os pedidos *API* dum cliente, determina que serviços são necessários e combina-os numa experiência síncrona para o utilizador."[17].

Num monólito, o cliente utiliza um só serviço, o próprio monólito. No entanto nos microserviços, os serviços encontram-se distribuídos por contentores diferentes, pelo que é preciso saber quais são os serviços disponíveis e para qual serviço(s) reencaminhar o pedido.
A exposição destes serviços diretamente ao cliente pode levar a alguns problemas, como um aumento na complexidade do código do cliente, pois este tem que manter o controlo de vários pontos de extremidade e processar falhas de forma resiliente. Esta exposição também obriga a um acoplamento entre o cliente e os serviços, isto porque o cliente precisa de saber como são os serviços decompostos, levando a uma dificuldade na manutenção do cliente.

Outro problema que pode surgir desta exposição será o aumento da latência na resposta aos pedidos, devido ao facto de que certos pedidos necessitam de mais do que um serviço, pelo que tem que haver várias comunicações entre o cliente e os serviços, aumentando o tempo de resposta ao pedido original. Ainda a exposição dos serviços leva a perigos para a segurança, pois estes estão diretamente acessíveis.[18]

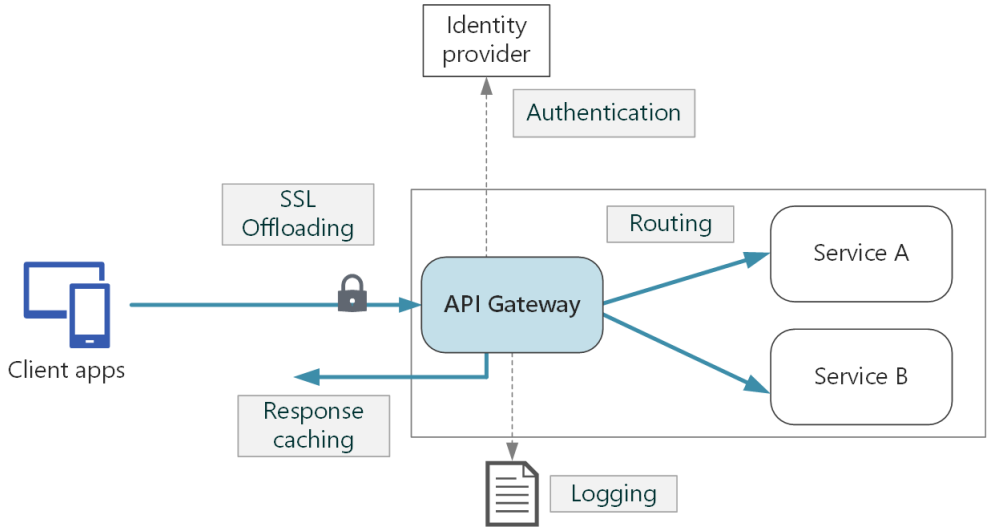

Figura 2.5: *API Gateway* [18]

Assim uma *API Gateway* é uma camada de abstração que se coloca entre o cliente e os serviços que este pretende utilizar. Esta *API* tem associado algumas funcionalidades que vão permitir mitigar os problemas acima mencionados.[18]

- Encaminhamento de pedidos. Com a *API Gateway* consegue-se o encaminhamento de pedidos para um ou mais serviços necessários, com encaminhamento a nível da camada de aplicação. Assim, oferece-se um ponto único para os clientes consumirem os serviços, ajudando a desacoplar os clientes dos serviços.
- Agregação. A *API* permite a agregação de vários pedidos individuais num único pedido. Assim, um pedido do cliente que exija pedidos a vários serviços vai ter apenas uma resposta agregada, reduzindo as ligações entre o cliente e os serviços, diminuindo por consequência a latência.
- Abstração de funções. A *API* permite que certas funções que existiam em diversos serviços lhes sejam abstraídas, pois esta serve de ponto de entrada. Assim funções como autenticação e autorização podem ser feitas pela *API Gateway*.

Outras tarefas podem ser abstraídas dos serviços e do seu serviço de orquestração, como a monitorização e *load balancing*.

2.5 Service Mesh

Uma *service mesh* é uma rede de microserviços que oferece, de maneira consistente capacidades críticas, incluindo descoberta de serviços (*service discovery*), *load balancing*, encriptação, observabilidade, capacidade de rastreamento de pedidos, autenticação e autorização.[19][20] Esta rede garante que a comunicação entre serviços que se encontram em contentores é rápida, confiável, e segura.

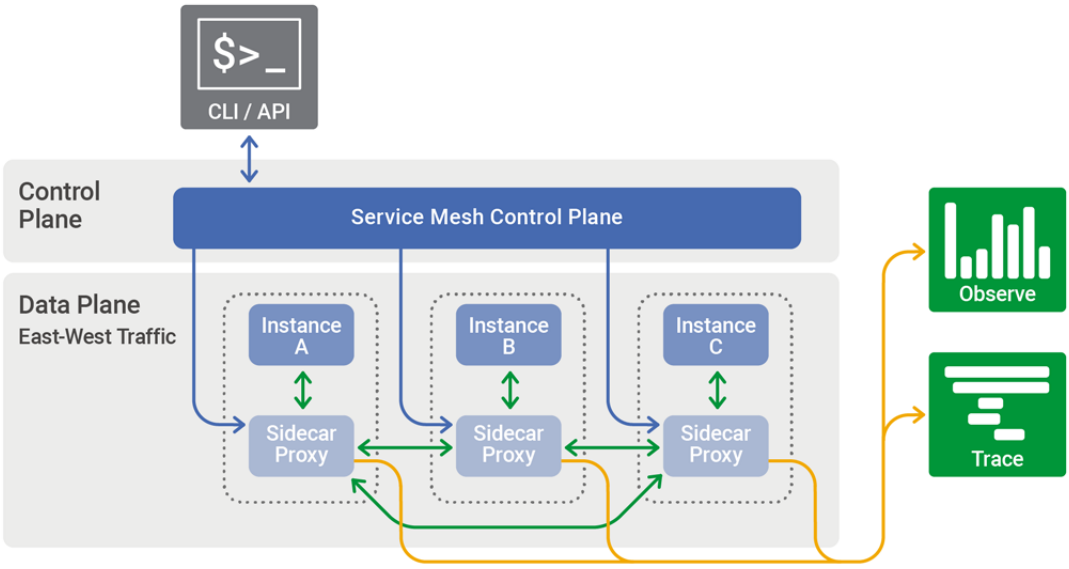

Figura 2.6: *Service Mesh* [19]

Estas capacidades são possíveis de obter, devido à utilização de um intermediário entre o pedido e o serviço, o *sidecar*. A *service mesh* é normalmente implementada com a utilização de um *sidecar proxy* para cada instância/contentor/*pod* dum serviço. Estes lidam com as comunicações entre serviços, monitorização e questões relacionadas com segurança. Ou seja, tudo o que possa ser abstraído dos serviços. Desta maneira, os *developers* podem cuidar do desenvolvimento, suporte e manutenção do código da aplicação no serviço, enquanto equipas de operação podem fazer a manutenção da *service mesh* e correr a aplicação.
As *service mesh* ajudam também, como já referido, com o *load balancing*. Várias ferramentas de orquestração (algumas das quais falaremos mais tarde) já incorporam capacidades de *load balancing* a nível da camada de transporte. No entanto as *service meshes* trazem a capacidade de se realizar *load balancing* a nível da camada da aplicação.[19]
O funcionamento de microserviços poderá beneficiar muito da combinação destes com uma *service mesh*. Como dito em cima, as *service mesh* trazem funcionalidades que retiram algumas responsabilidades do orquestrador de microserviços e dos microserviços em si, levando a uma mais fácil gestão e monitorização do funcionamento destes.
No entanto, as ferramentas que ajudam na implementação de *service meshs* ainda se encontra num estado pouco maduro, não conseguindo fornecer esta funcionalidade com todas as suas potencialidades.

2.6 Time Series Database

Uma *Time Series Database* (*TSDB*) é uma base de dados otimizada para dados com um carimbo temporal. Estes dados são medições ou eventos simples que são rastreados, monitorizados e agregados ao longo do tempo. Os dados podem ser métricas de um servidor, monitorização da performance de uma aplicação, dados de rede, dados de sensores, entre outros.[21]
Este tipo de base de dados é desenvolvido especificamente para lidar com métricas e eventos que estão marcados temporalmente, estando otimizada para lidar com a mudança ao longo do tempo. As propriedades que tornam estes dados muito diferentes são a gestão do ciclo de vida destes, a compactação e revista alargada de muitos registos.[21]

A importância das *TSDB* prende-se com o facto de as condições fundamentais da computação terem mudado na última década. Anteriormente, estas já eram utilizadas na análise financeira e na bolsa. No entanto com a mudança de paradigma de monólitos para *software* dividido em microserviços, tudo ficou compartimentalizado. Assim, cada uma destas partes, componentes, gera dados individualmente, sendo criada uma quantidade enorme destes dados. Para se poder visualizar estes é preciso a ajuda destas bases de dados, que permitem a análise temporal e a representação organizada dos dados conforme estes foram gerados, devido ao seu carimbo temporal. As *TSDB* estão especialmente otimizadas para tratar os dados de modo a rapidamente mostrar valores como percentis ou distribuições ao longo do tempo, algo que é pesado no caso de haverem muitos dados.[21]

2.6.1 Tecnologias

InfluxDB

O *InfluxDB* é uma *time series database* de código aberto. Foi desenvolvida para lidar com uma grande quantidade de escrita de dados e de *queries* e tem uma linguagem de *query* própria parecida a *SQL*, denominada *InfluxQL*, para interagir com os dados.[22]
Trata-se de uma solução estável preparada para ser utilizada em produção. Suporta milhões de escritas na base de dados por segundo e consegue acompanhar até as maiores exigências de monitorização. Para além da linguagem parecida com *SQL* para a realização de *queries*, dispõe também de funções específicas que permitem adiantar trabalho no processamento de dados.[22]
Ao conseguir recolher tantos dados em tão pouco tempo, algumas preocupações com espaço surgem. Para colmatar este problema, o *InfluxDB* compacta automaticamente os dados para minimizar o espaço utilizado. Para isso, é guardado por um periodo limitado de tempo os dados com todos os pormenores, e a longo prazo são guardados dados com menos precisão, mas permitindo na mesma que estes possam ser consultados no futuro.[22]

Por estas razões, o *InfluxDB* aproxima-se de uma ferramenta ideal com as características necessárias para o armazenamento a longo prazo de dados, mas que mantêm a capacidade de permitir a análise desses dados, não se desfazendo dos dados necessários para a fazer.

2.7 Cluster Manager

Com a utilização de contentores, é conseguido o isolamento desejado, separando funcionalidades. No entanto, alguns desafios advêm da utilização desta tecnologia. Para começar, a gestão de dependências entre contentores em aplicações de várias camadas é um problema. Mas mais importante no contexto do problema apresentado nesta tese, o desafio de definir, fazer *deploy* e operar o conjunto de contentores que fazem parte de uma aplicação. Para isto, surgem soluções que têm como objetivo fazer esta gestão, chamados *Cluster Managers*.[8]
Uma arquitetura baseada em grupos de contentores agrupa *hosts* - servidores virtuais sobre *hypervisors* ou possivelmente servidores *baremetal* - em *clusters*. Cada *host* pode conter vários contentores com serviços comuns como agendamento, *load balancing* e aplicações. Serviços de aplicação são grupos lógicos de contentores pertencentes à mesma imagem. Estas permitem escalabilidade de uma aplicação entre *hosts*.

O *deployment* de aplicações distribuídas através de contentores é suportado usando um serviço node virtual escalável (*cluster*) com alta complexidade interna (suportando escala-

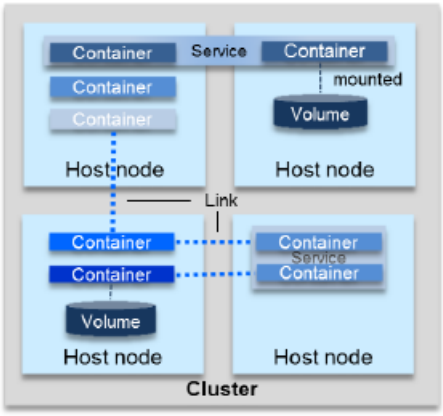

Figura 2.7: *Cluster* [8]

bilidade, *load balancing* e *failover*) e reduzida complexidade externa. Um API permite a gestão de *clusters* apartir da criação de serviços e contentores.[8]
Uma arquitetura de *cluster* é composta por motores que partilham a descoberta de serviços, orquestração e *deployment* (*load balancing*, monitorização, escalabilidade e também armazenamento de ficheiros). Assim, ficam satisfeitos alguns requisitos avançados por Nane Kratzke.[8][23]

Requisitos
Elevada disponibilidade de serviços
Comunicação segura entre serviços
Escabilidade automática e *load balancing*
Orquestração e *service discovery* escalável e distribuído
Transferência de serviços entre *clusters*

Tabela 2.1: Requisitos de uma arquitetura de *clusters* satisfeitos

Atualmente já existem algumas soluções que pretendem preencher este campo da gestão de *clusters*, mas que se encontram num estado embrionário[24] , pois não apresentam qualquer funcionalidade automática. No geral, tudo tem que ser configurado à mão, não havendo uma resposta automática destas soluções, principalmente a nível da elasticidade.
Algumas soluções existentes serão agora analisadas, para permitir uma melhor compreensão deste conceito.

2.7.1 Tecnologias

Kubernetes

A ferramenta *Kubernetes* "é uma plataforma portátil, extensível e *open source* para gestão de cargas de trabalho e serviços em contentores, que facilita tanto a configuração declarativa como a automação".[25]

O *Kubernetes* faz uso dos contentores como a técnica de agrupar e executar aplicações. Assim, serve como gestor destes contentores, garantindo que não existe paragem (*downtime*) da aplicação.[25]
"O *Kubernetes* fornece uma estrutura para executar sistemas distribuídos de forma resiliente. Este cuida dos seus requisitos de dimensionamento, *failover*, padrões de *deploy* e mais".[25] Assim, o *Kubernetes* proporciona:

- Descoberta de serviços e balanceamento de carga
- Orquestração de armazenamento
- *Rollouts* e *rollbacks* automatizados
- Permite a especificação de quanto *CPU* e memória *RAM* cada contentor precisa
- Auto-regeneração (reposição de contentores que falharam, terminação de contentores que não respondem)
- Gestão de configurações e dados secretos sem perigo de serem expostos

Iremos agora explorar um pouco mais a fundo os componentes desta ferramenta. Alguns conceitos chave do sistema *Kubernetes* são:

- *Nodes*
- *Pods*
- *Deployments*
- *Services*

Começamos por analisar os *Nodes*. Estes são as máquinas no *Kubernetes*, tanto físicas como virtuais. Estes *Nodes* contêm várias informações que podem ser observadas. Primeiramente, é possível ver os endereços, sendo estes o *hostname* e os *IP's* internos e externos do *Node*. Em segundo lugar, as condições. Estas podem ser duas: *Ready* e *OutOfDisk*. Se a condição for *OutOfDisk*, quer dizer que existe espaço insuficiente para a adição de novos *Pods*. Já se for *Ready*, isto significa que se trata de um *Node* saudável e que está pronto para receber mais *Pods*. Por fim, se o controlador não conseguir aceder ao *Node* durante 40 segundos, a condição *Ready* muda para *Unknown*. Em terceiro lugar, a capacidade. Com esta informação consegue-se saber os recursos disponíveis no *Node*, tais como *CPU*, memória e o número máximo de *Pods* permitidos. Em quarto e último lugar, informação geral. Daqui consegue-se observar qual a versão do programa que está a correr, bem como o nome do *OS*. Para controlar estes *Nodes* existem os chamados *Node Controllers*. É a sua responsabilidade garantir que o *IP* certo é atribuído. Também acompanham a relação entre a lista de *Nodes* disponíveis e a lista de máquina disponíveis. Quando um *Node* é considerado não saudável, o *Controller* verifica também a saúde do *Node*. Se for um caso de estar pouco saudável, o *Node* é removido da lista de *Nodes* disponíveis.[26]
Viramos a atenção agora para os *Pods*, que são um grupo de um ou mais contentores, do seu armazenamento partilhado e as suas configurações. Vários tipos de contentores são suportados pelo *Kubernetes*, sendo o mais comum os do *Docker*.
Estes *Pods* não devem ser considerados como entidades persistentes. No caso de acontecer alguma falha estes serão destruídos, pelo que não se deve guardar informação importante nestes. Esta deve ser guardada em armazenamento próprio. Para entender melhor este conceito da efemeridade dos *Pods*, será importante perceber o ciclo de vida destes.
Cada *Pod* estará numa fase, que é um resumo do estado atual do *Pod*. Estas fases são: "Pendente", o *Pod* foi aceite pelo *Kubernetes*, mas algumas imagens ainda estão a ser criadas; "A correr", todos os contentores foram criados e o *Pod* está ligado a um *Node*; "Bem sucedido", o *Pod* terminou todos os contentores com sucesso; "Falhado", o *Pod* terminou e pelo menos um dos contentores falhou; ou "Desconhecido", que quer dizer que não é possível aceder ao estado do *Pod*.[26]
Em terceiro, os *Deployments*. Estes são utilizados para fazer atualizações no *Pods*. Podem

ser usados para criar novos *Pods*, mudar a versão de um contentor e até recrear o estado anterior se alguma coisa correr mal. Quando se cria um *Deployment*, define-se um estado desejado e o *Kubernetes* irá manter o ambiente nesse estado. A título de exemplo, se for definido através de um *Deployment* que é desejável ter 3 réplicas sempre presentes, caso uma vá abaixo ou falhe, esta é reposta, para manter o número desejado de 3.
A cada *Deployment* está uma associado um estado. Acedendo a este, saberemos se tudo está a funcionar como pretendido. Assim, quando se observa o estado de um *Deployment*, pode-se obter as seguintes informações: "Desejada", o número de *Pods* que se pretende ter; "Corrente", diz o número de replicas que o *Deployment* está a gerir; "Atualizados", o número de *Pods* que têm a versão mais recente; e "Disponível", quantos *Pods* estão no estado *Ready*.[26]
Por fim, os *Services*. Estes são definidos como um conjunto de *Pods* e a política que define o controlo de acesso. Estes servem para combater a efemeridade dos *Pods*. Como os *Pods* não são estáticos, é preciso definir como estes podem ser encontrados.
Quando se publicam *Kubernetes Services*, pode-se definir como estes vão ser expostos. Os tipos de *Services* que podem ser definidos são: "*ClusterIP*", o serviço vai ser exposto dentro do *cluster*, com um *IP* local, e não será acessível de fora do *cluster*; "*NodePort*", expõem o serviço num dado porto, usando o IP do *Node*. A título de exemplo, se um *Node* corre no *IP* 10.0.5.5, e o *NodePort* for 1234, o serviço pode ser utilizado no endereço 10.0.5.5:1234; "*LoadBalancer*", expõem o serviço utilizando um *load balancer*; e por fim, "*ExternalName*", o serviço será exposto utilizando o nome configurado numa propriedade. A título de exemplo, "projeto.com".[26]

Após a análise feita em cima, consegue-se perceber como o *Kubernetes* funciona, e as vantagens que lhe estão associadas na orquestração de contentores, bem como na disponibilização de serviços para uso externo.

Docker Swarm

Antes de se puder analisar o que é o *Docker Swarm*, tem que se falar do que é o *Docker*. O *Docker* é "uma plataforma aberta para desenvolvimento, envio e execução de aplicações. Este permite que se separe as aplicações da infraestrutura para que se possa fornecer software rapidamente.". A plataforma *Docker* permite embalar e correr aplicações num ambiente isolado conhecido como "contentor". O isolamento e segurança permitem correr vários contentores simultaneamente no mesmo *host*, quer seja este físico ou virtual.[27]
O *Docker* baseia-se numa arquitetura cliente-servidor. O *Docker client* comunica com o *Docker daemon*, que se encarrega de construir, correr e distribuir os contentores *Docker*. Tanto o *client* como o *daemon* podem correr no mesmo sistema, ou podem estar ligados remotamente comunicando através de uma *REST API*.[27]
O *Docker daemon* ouve pedidos através do *Docker API* e gere objetos *Docker* como imagens, contentores, redes e volumes. Um *daemon* também pode comunicar com outros *daemons* para gerir serviços *Docker*.[27]
O *Docker client* é a maneira principal que os utilizadores usam para comunicar com o *Docker*. Quando se faz uso de comandos, o *client* envia esses comandos para o *daemon*, que trata de os executar. O *client* pode comunicar com mais do que um *daemon*.[27]
O *Docker registry* guarda imagens *Docker*. Uma imagem *Docker* é um molde para criar contentores, que por vezes é baseada numa outra imagem, com personalização extra. Existe um registo público de imagens denominado *Docker Hub* que qualquer um pode utilizar. Por defeito, o *Docker* procura automaticamente imagens neste *Hub*. Caso se deseje, pode-se correr um registo privado.[27]
Por fim, um *Docker Engine* é tudo o que foi falado anteriormente. Trata-se da aplica-

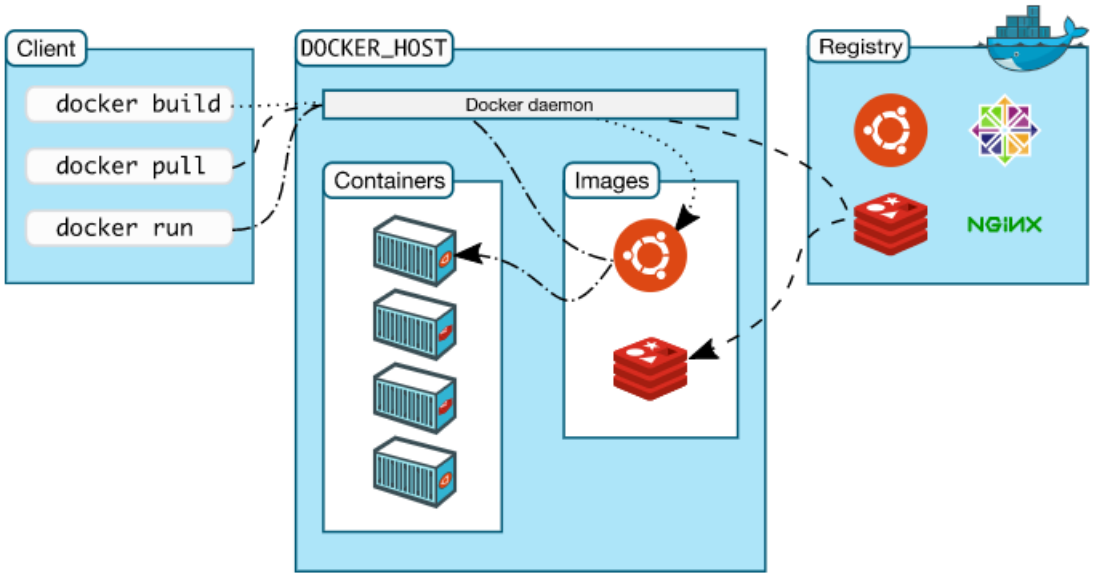

Figura 2.8: Arquitetura *Docker* [27]

ção cliente-servidor que contêm os diferentes componentes referidos, exceptuando o *Docker Hub*. Engloba o *Docker daemon*, o *Docker API* e a *CLI* (*Command Line Interface*) que permite a interação com o *daemon*.
Explicado um pouco melhor o que é o *Docker*, pode-se avançar para o que é o *Docker Swarm*.

O *Docker Swarm* é um modo que permite a gestão de *clusters* de *Docker Engines*, nativamente com a plataforma *Docker*. Pode-se utilizar o *Docker CLI* para criar um *swarm*, para fazer *deploy* de aplicações no *swarm* e gerir o comportamento deste.[28]
Isto permite que equipas de desenvolvimento, bem como administradores, possam realizar coordenação entre contentores e alocar tarefas a grupos de contentores; realizar verificação do estado de saúde de contentores e gerir os ciclos de vida de contentores individuais; possibilitar redundância e *failover* no caso de *nodes* que falham; escalar o número de contentores, aumentando e diminuindo o número de acordo com a carga; e ainda realizar atualizações sobre múltiplos contentores.[28]
Esta ferramenta permite a realização de várias tarefas críticas relativas à orquestração de microserviços, sendo uma tecnologia a considerar quando se procura uma solução para este fim.

Capítulo 3

Trabalho relacionado

O presente trabalho tem como objetivo a especificação de uma plataforma para a gestão automática de aplicações de microserviços. Assim, neste capítulo irão ser analisadas ferramentas e trabalhos já realizados que se encontram relacionados com a plataforma que se pretende implementar por objetivos ou conceitos.

3.1 Monitoria de Arquiteturas de Microserviços

O trabalho desenvolvido pelo colega Fábio Pina é uma plataforma de monitorização de microserviços. Esta plataforma consiste na utilização de uma *API Gateway* que monitoriza os pedidos, e observa também os microserviços que estão envolvidos em cada pedido. Como cada um destes pedidos tem que passar obrigatoriamente pela *API*, várias métricas são retiradas e guardadas em bases de dados. Com estes dados guardados, as métricas podem ser observadas com a ajuda de uma ferramenta de *dashboards* revelando o que acontece dentro da aplicação.

Para conseguir fazer *deploy* da aplicação de microserviços, bem como o sistema de monitorização, foi utilizado o *Docker Swarm*, um sistema de gestão de *clusters* e de agendamento, que tem como objetivo gerir os *nodes* do *swarm* criado. Estes *nodes* podem ser máquinas físicas ou virtuais, pelo que se trata de uma ferramenta bastante versátil. Esta solução foi analisada na secção 2.7.1.

Para a visualização das métricas retiradas, é utilizada a ferramenta *Grafana*, que permite a visualização das métricas, levando a um melhor entendimento destas. É uma ferramenta bastante mutável, que possibilita a criação de diversas *dashboards* diferentes para se conseguir um visualização diversificada.[29]

A nível de monitorização e *API*, o colega Fábio Pina utilizou um projeto de código aberto da *Netflix* denominado *Zuul*[30]. Este projeto permite a implementação de uma *gateway* de entrada para os pedidos que chegam à plataforma, à qual se podem aplicar filtros para realização de monitoria, entre outras funcionalidades.
Várias métricas foram retiradas com recurso a esta *gateway*, sendo informação crítica para analisar os pedidos. Alguns dos dados retirados são o tempo de inicio do pedido, tempo do fim, duração, *IPs* de origem e destino, serviço de destino, instância de destino, método utilizado e código de resposta.

3.2 Plataforma de Microserviços Elásticos

O objetivo do trabalho do colega Fábio Ribeiro era a criação de uma plataforma de gestão automática de aplicações de microserviços. Com recurso a rastreamento distribuído (isto num trabalho futuro) e a utilização de um algoritmo a ser desenvolvido pelo Eng. Jaime Correia que analisaria os *traces*, ir-se-ia conseguir realizar automaticamente elasticidade na aplicação de microserviços que estava a ser monitorizada. Com a análise dos *traces* retirados, conseguir-se-ia perceber onde é que a aplicação estava a ter problemas, quer fossem falhas, quer fossem engarrafamentos, e de maneira automática o sistema realizava escalonamento dos microserviços que precisassem, aumentando ou diminuindo o número de instância desses microserviços.

O trabalho que acabou por ser realizado foi a base para esta plataforma, tendo o colega Fábio Ribeiro conseguido implementar uma plataforma que acolhe a aplicação de microserviços que se pretende monitorizar, e disponibilizar um ponto de entrada para o algoritmo falado anteriormente. Assim, a fundação para uma plataforma que escala automaticamente ficaram lançadas.

Para este fim, o colega utilizou o *Kubernetes*, uma ferramenta de gestão de serviços e trabalhos em contentores, que permite configurações declarativas ou automatizadas. Esta solução foi analisada mais ao pormenor na secção 2.7.1.

3.3 *Amazon Elastic Compute Cloud* (*Amazon EC2*)

O *Amazon Elastic Compute Cloud* (*Amazon EC2*) é "um serviço web que fornece capacidade computacional segura e redimensionável na *cloud*. É projetado para tornar a computação *web* em *cloud* em escala mais fácil para os *developers*".[31]

Esta plataforma fornece capacidade computacional escalável na *Amazon Web Services Cloud* (*AWS*). A *Amazon EC2* permite lançar tantos servidores virtuais quantos necessários, configurar segurança e redes, e gerir armazenamento. Assim, é possível escalar para lidar com mudanças nos requisitos da aplicação que está a correr ou na popularidade desta.[32] Várias funcionalidades são providenciadas por esta plataforma, sendo de seguida enumeradas algumas destas[32]:

- Ambientes de computação virtual, conhecidas como *instances*
- Modelos pré-configurados para as *instances*, conhecidos como *Amazon Machine Images*, que empacotam tudo o que é necessário para o servidor do utilizador (como o sistema operativo e software necessário)
- Várias configurações de *CPU*, memória, armazenamento e capacidade de rede para as *instances*, conhecidas como *instances types*
- Entre outras

O *Amazon EC2* permite também que o utilizador defina condições para o funcionamento da sua aplicação. Com a auto-escalabilidade que este fornece (*EC2 Auto Scaling*), a plataforma escala automaticamente para lidar com um aumento na carga, mantendo o desempenho. Quando a carga diminui, o número de instâncias ativas também diminui, baixando o custo associado à utilização do *Amazon EC2*.[33] Esta plataforma põem à disposição

do utilizador alguns sistemas operativos, sendo que alguns têm um custo associado à sua utilização, como o sistema operativo da *Amazon*, o *Amazon Linux*.

O mencionado anteriormente faz com que esta plataforma seja uma opção que apela aos utilizadores, pois estes não precisam de se preocupar em gerir a sua própria infraestrutura, podendo usar os serviços da *Amazon* para desenvolver, correr e manter as suas aplicações.

3.4 *Amazon Elastic Container Service* (*Amazon ECS*)

Para a correr aplicações de microserviços, um serviço como o *Amazon ECS* pode ser mais indicado. Trata-se de um serviço de orquestração de contentores que suporta contentores *Docker*, permitindo corre e escalar facilmente aplicações constituídas por contentores no *AWS*. Este serviço retira a responsabilidade do utilizador ter que instalar o seu próprio serviço de orquestração, eliminando a necessidade de gerir e escalar um *cluster* de máquinas virtuais, ou programar contentores nessas máquinas virtuais.[34]

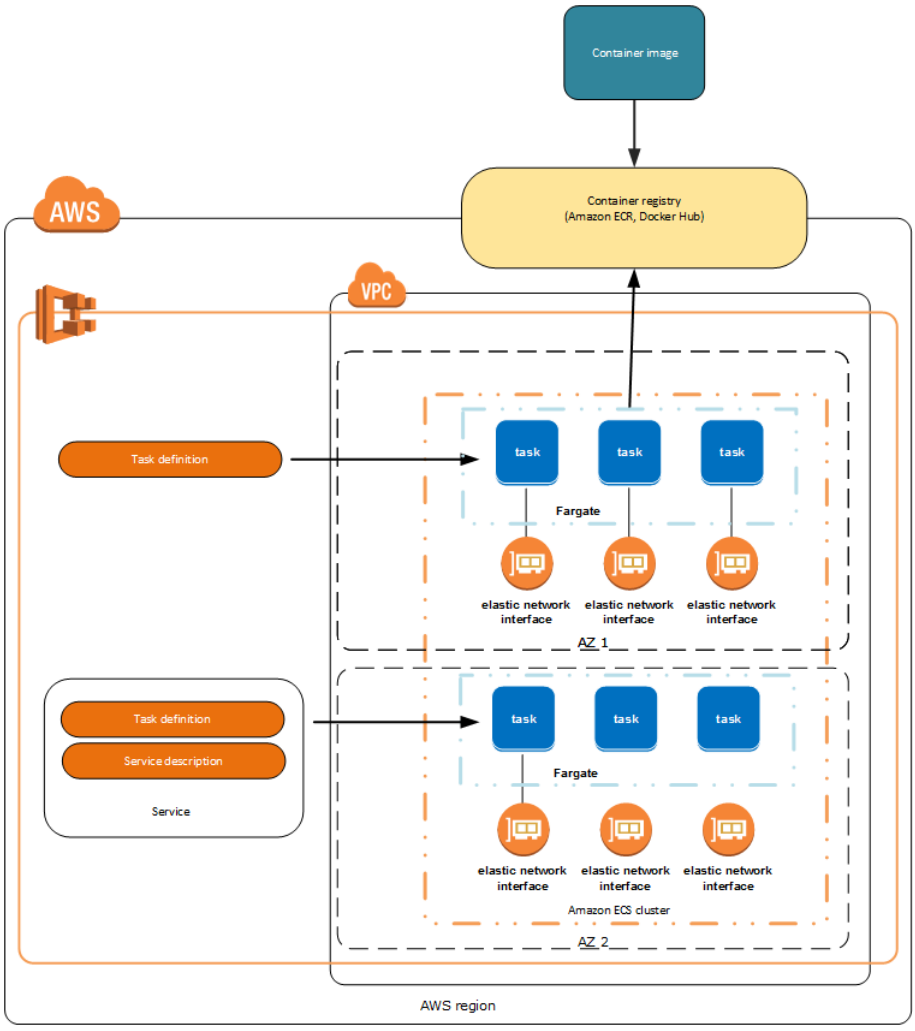

Figura 3.1: Ambiente *Amazon ECS* [35]

Existem várias partes constituintes desta plataforma quando se cria uma infraestrutura, sendo a mais central o *cluster*.
Clusters são um grupo de instâncias de contentores, sendo que um *cluster* pode conter diferentes tipos de instâncias de contentores, o que significa que se pode realizar diferentes

tarefas dentro do mesmo *cluster*. Quando se cria um *cluster* novo é preciso configurar várias propriedades, como o tipo de instâncias *EC2*, o número de instâncias que devem ser lançadas, o armazenamento *EBS* definindo a quantidade, um par de chaves para acesso *SSH* e ainda a rede *VPC* (*Virtual Private Cloud*).[36]

Esta plataforma fornece várias funcionalidades:

- Desenvolvimento
- Gestão
- Programação
- Rede
- Monitorização e Registo

A nível de desenvolvimento, com o *Amazon ECS* é possível correr contentores *Docker* sem alterações nas configurações. Permite também a utilização de contentores *Windows*, que é suportada através de uma *AMI* (*Windows Amazon Machine Image*). O *Amazon ECS CLI* permite simplificar a experiência de desenvolvimento local, bem como facilitar a configuração e *deploy* de contentores. O *Amazon ECS* é compatível ainda com repositórios de imagens de outras empresas, como o *Docker Hub*.[37]
É possível gerir vários aspetos do *Amazon ECS*, através da definição de tarefas, denominadas *Task Definition*. Com estas definições, é possível especificar um ou mais contentores que são precisos para uma tarefa, incluindo memória e requisitos de *CPU*, partilha de armazenamento e como os contentores ligam entre si. Estas definições também permitem o lançamento de quantas tarefas forem necessárias de apenas um único *Task Definition*. O *Amazon ECS* também facilita o lançamento de atualizações nos contentores, começando a lançar novos contentores com a versão mais recente, retirando os contentores com a versão anterior. A plataforma dispõe ainda de recuperação automática, substituindo contentores não saudáveis, para manter o número desejado de contentores.[37]
O *Amazon ECS* dispõem de múltiplas estratégias de programação que colocam os contentores nos *clusters* tendo em conta os recursos necessários (como por exemplo *CPU* ou *RAM* e requisitos de disponibilidade. Também é possível configurar como é que as tarefas são dispostas num *cluster* de instâncias *EC2* baseado em atributos como o tipo de instâncias, zona de disponibilidade ou atributos definidos pelo utilizador.[37]
A nível de rede, o *Amazon ECS* integra *service discovery*, para facilitar a descoberta e ligação de contentores. Isto é conseguido através da definição de nomes para os recursos da aplicação, mantendo uma localização atualizada destes recursos que mudam dinamicamente, aumentando assim a disponibilidade da aplicação. Para além do referido, a plataforma tem também capacidade de *load balancing* que permite a distribuição da carga pelos contentores.[37]
Por último, o *Amazon ECS* permite a realização de monitorização, através da recolha de métricas como a utilização de *CPU* e memória de tarefas que estão a correr. Também é possível o armazenamento de registos (*logs*), que contêm informações como as chamadas à *API*, a hora de entrada da chamada, o *IP* de origem da chamada, os parâmetros da chamada e a resposta enviada pelo *Amazon ECS*.[37]

Capítulo 4

Descrição da arquitetura

Nesta secção, serão abordados os requisitos do sistema que se pretende implementar, e será apresentada uma proposta de arquitetura que serviu de guia para concretização do sistema no segundo semestre.

Numa primeira fase, nas secções 4.1.1 e 4.1.2, vão ser mencionados os requisitos funcionais e os atributos de qualidade nos quais a arquitetura de baseará. Numa segunda fase, na secção 4.2, vai ser explanada a arquitetura, sendo que os detalhes desta serão discutidos e explicados, bem como as tecnologias a utilizar.

Neste trabalho queremos abordar o problema da realização automática de elasticidade em aplicações de microserviços. Para esse fim pretende-se proceder à criação de uma plataforma de gestão de microserviços, que através da coleção de métricas e *traces*, os quais são obtidos através de recolha normal e técnicas de rastreamento distribuído (*tracing*), consegue realizar uma análise destas para obter a elasticidade pretendida.
A análise das métricas e *traces* para a realização de elasticidade é feita conforme a disponibilidade dos mesmo. A utilização de *tracing* implica a instrumentalização dos microserviços, pelo que estes têm que implementar as ferramentas para enviar os *traces* para o ponto de recolha. Por esta razão, pode acontecer que nem todos os microserviços enviem *traces*, fazendo com que estes possam não estar disponíveis. Assim, a plataforma precisa de conseguir realizar uma análise tendo em conta o que tem à sua disposição, seguindo uma espécie de caminho de degradação. Existindo *traces*, optará por utilizar estes, pois são mais completos. Na sua ausência, a plataforma utilizará as métricas retiradas.

4.1 Requisitos

4.1.1 Requisitos Funcionais

Nesta secção vão ser estabelecidos os requisitos funcionais que serviram de base para o desenvolvimento da plataforma no segundo semestre.

Estes são de extrema importância, pois só com base nestes é que se consegue uma arquitetura que satisfaça o desejado, permitindo que se consiga a definição de testes para validar a plataforma.

Criação de conta na plataforma

Descrição e Prioridade
Descrição: Para que o utilizador da plataforma possa fazer *deploy* da sua aplicação, e de seguida possa fazer a sua gestão, precisa de se autenticar na plataforma. Mas para isso precisa de registar uma conta de utilizador.
Prioridade: Alta

Estímulos/Sequência de Respostas

Estímulo	Resposta do sistema
O utilizador da plataforma escolhe a opção de se registar	A plataforma apresenta uma interface para a realização do registo
O utilizador da plataforma preenche corretamente os dados obrigatórios	A plataforma apresenta uma mensagem de sucesso
O utilizador da plataforma preenche incorretamente os dados obrigatórios	A plataforma apresenta uma mensagem de insucesso

Requisitos Funcionais
REQF-1: Registo na plataforma

Login na plataforma

Descrição e Prioridade
Descrição: Para que o utilizador da plataforma possa fazer *deploy* da sua aplicação, e de seguida possa fazer a sua gestão, precisa de se autenticar na plataforma.
Prioridade: Alta

Estímulos/Sequência de Respostas

Estímulo	Resposta do sistema
O utilizador da plataforma escolhe a opção de fazer *login*	A plataforma apresenta uma interface para a realização do *login*
O utilizador da plataforma preenche corretamente os dados obrigatórios	A plataforma apresenta uma mensagem de sucesso
O utilizador da plataforma preenche incorretamente os dados obrigatórios	A plataforma apresenta uma mensagem de insucesso

Requisitos Funcionais
REQF-2: *Login* na plataforma

***Deploy* de novas aplicações na plataforma**

Descrição e Prioridade
Descrição: A plataforma tem que permitir o *deploy* de aplicações de microserviços .
Prioridade: Alta

Estímulos/Sequência de Respostas

Estímulo	**Resposta do sistema**
O utilizador da plataforma escolhe a opção de fazer *deploy* da sua aplicação	Uma vez feito o *deploy*, a plataforma cria um API Gateway por onde os utilizadores da aplicação podem fazer uso da aplicação

Requisitos Funcionais
REQF-3: *Deploy* de novas aplicações
REQF-4: Providenciar uma API Gateway para que os utilizadores da aplicação possam utilizar a aplicação

Gestão da aplicação - Iniciar uma aplicação

Descrição e Prioridade
Descrição: O utilizador da plataforma deve poder iniciar a aplicação.
Prioridade: Média

Estímulos/Sequência de Respostas

Estímulo	**Resposta do sistema**
O utilizador da plataforma escolhe a opção "Iniciar a aplicação"	A plataforma apresenta uma mensagem a pedir a confirmação por parte do utilizador para garantir que pretende mesmo iniciar a aplicação
O utilizador da plataforma confirma que quer iniciar a aplicação	A plataforma procede à inicialização da aplicação
O utilizador da plataforma não confirma que quer iniciar a aplicação	A plataforma cancela o processo de inicialização da aplicação

Requisitos Funcionais
REQF-5: Permitir que o utilizador da plataforma inicie uma aplicação que esteja parada

Gestão da aplicação - Parar uma aplicação

Descrição e Prioridade
Descrição: O utilizador da plataforma deve poder parar a aplicação.
Prioridade: Média

Estímulos/Sequência de Respostas

Estímulo	Resposta do sistema
O utilizador da plataforma escolhe a opção "Parar a aplicação"	A plataforma apresenta uma mensagem a pedir a confirmação por parte do utilizador para garantir que pretende mesmo parar a aplicação
O utilizador da plataforma confirma que quer parar a aplicação	A plataforma procede à paragem da aplicação, tomando as devidas precauções para não haver perda de dados
O utilizador da plataforma não confirma que quer parar a aplicação	A plataforma cancela o processo de paragem da aplicação

Requisitos Funcionais
REQF-6: Permitir que o utilizador da plataforma pare uma aplicação que esteja a correr

Gestão da aplicação - Iniciar um microserviço

Descrição e Prioridade
Descrição: O utilizador da plataforma deve poder iniciar um microserviço.
Prioridade: Média

Estímulos/Sequência de Respostas

Estímulo	Resposta do sistema
O utilizador da plataforma escolhe a opção "Iniciar um microserviço"	A plataforma apresenta uma lista com os microserviços que estão parados na aplicação
O utilizador da plataforma escolhe o microserviço que pretende iniciar	A plataforma apresenta uma mensagem a pedir a confirmação
O utilizador da plataforma confirma que quer iniciar o microserviço	A plataforma procede à inicialização do microserviço
O utilizador da plataforma não confirma que quer iniciar o microserviço	A plataforma cancela o processo de inicialização do microserviço

Requisitos Funcionais
REQF-7: Permitir que o utilizador da plataforma inicie um microserviço que esteja parado

Gestão da aplicação - Parar um microserviço

Descrição e Prioridade
Descrição: O utilizador da plataforma deve poder parar um microserviço.
Prioridade: Média

Estímulos/Sequência de Respostas

Estímulo	**Resposta do sistema**
O utilizador da plataforma escolhe a opção "Parar um microserviço"	A plataforma apresenta uma lista com os microserviços que estão a correr na aplicação
O utilizador da plataforma escolhe o microserviço que pretende parar	A plataforma apresenta uma mensagem a pedir a confirmação
O utilizador da plataforma confirma que quer parar o microserviço	A plataforma procede à paragem do microserviço, tomando as devidas precauções para não haver perda de dados
O utilizador da plataforma não confirma que quer parar o microserviço	A plataforma cancela o processo de paragem do microserviço

Requisitos Funcionais
REQF-8: Permitir que o utilizador da plataforma pare um microserviço que esteja a correr

Gestão da aplicação - Adicionar um microserviço à aplicação

Descrição e Prioridade
Descrição: O utilizador da plataforma deve poder adicionar um microserviço à aplicação.
Prioridade: Média

Estímulos/Sequência de Respostas

Estímulo	**Resposta do sistema**
O utilizador da plataforma escolhe a opção "Adicionar um microserviço" e submete os ficheiros necessários	A plataforma adiciona o microserviço, realizando as alterações necessárias

Requisitos Funcionais
REQF-9: Permitir que o utilizador da plataforma adicione um microserviço à aplicação

Gestão da aplicação - Remover um microserviço da aplicação

Descrição e Prioridade
Descrição: O utilizador da plataforma deve poder remover um microserviço da aplicação.
Prioridade: Média

Estímulos/Sequência de Respostas

Estímulo	**Resposta do sistema**
O utilizador da plataforma escolhe a opção "Remover um microserviço"	A plataforma apresenta uma lista de microserviços da aplicação
O utilizador escolhe o microserviço a remover	A plataforma remove o microserviço, realizando as alterações necessárias

Requisitos Funcionais
REQF-10: Permitir que o utilizador da plataforma remova um microserviço da aplicação

Gestão da aplicação - Definição de métricas de qualidade

Descrição e Prioridade
Descrição: O utilizador da plataforma deve poder definir métricas de qualidade.
Prioridade: Média

Estímulos/Sequência de Respostas

Estímulo	**Resposta do sistema**
O utilizador da plataforma escolhe a opção "Definir métricas de qualidade"	A plataforma apresenta uma nova interface com a opção de definir percentis
O utilizador da plataforma define o percentil desejado	A plataforma tem em conta estas métricas na gestão da aplicação

Requisitos Funcionais
REQF-11: Permitir que o utilizador da plataforma possa definir métricas de qualidade

Gestão da aplicação - Acesso a informação sobre a aplicação

Descrição e Prioridade
Descrição: O utilizador da plataforma deve ter acesso a informações sobre a aplicação.
Prioridade: Média

Estímulos/Sequência de Respostas

Estímulo	**Resposta do sistema**
O utilizador da plataforma escolhe a opção "Informações sobre a aplicação"	A plataforma apresenta uma nova interface com informações, sendo estas

Requisitos Funcionais
REQF-12: Permitir que o utilizador da plataforma tenha acesso a informação detalhada sobre a sua aplicação, sendo esta informação o tempo de resposta (através do tempo de início e o tempo do fim do pedido), IP da máquina invocada e IP da máquina invocadora, as portas de origem, o serviço de destino, o método chamado, o URL do pedido, a resposta ao pedido

Monitorização

Descrição e Prioridade
Descrição: A plataforma deve coletar métricas associadas ao uso da aplicação.
Prioridade: Alta

Estímulos/Sequência de Respostas

Estímulo	**Resposta do sistema**
Um novo pedido chega à aplicação	A plataforma deve coletar métricas, como tempos de resposta, caminho percorrido pelo pedido dentro da aplicação, ID's dos microserviços utilizados, e *traces*

Requisitos Funcionais
REQF-13: Obter métricas que permitam uma análise do que acontece na aplicação, como o tempo de resposta (através do tempo de início e o tempo do fim do pedido), IP da máquina invocada e IP da máquina invocadora, as portas de origem, o serviço de destino, o método chamado, o URL do pedido, a resposta ao pedido
REQF-14: Obter *traces* que permitam uma análise do que acontece na aplicação

Métricas

Descrição e Prioridade
Descrição: A plataforma deve disponibilizar as métricas retiradas através de um *endpoint/interface*.
Prioridade: Média

Estímulos/Sequência de Respostas

Estímulo	Resposta do sistema
O algoritmo de análise de *traces* precisa de acesso aos novos *traces* que foram gerados	A plataforma deve disponibilizar esses *traces* através dum *endpoint*

Requisitos Funcionais
REQF-15: A plataforma disponibiliza *traces* através de um *endpoint*

Escalabilidade

Descrição e Prioridade
Descrição: A plataforma deve escalar automaticamente o número de instâncias de um ou mais microserviços da aplicação como resposta uma carga elevada de pedidos.
Prioridade: Alta

Estímulos/Sequência de Respostas

Estímulo	Resposta do sistema
Um número elevado de pedidos chega à aplicação	A plataforma tenta recolher métricas e *traces*
O algoritmo de análise tenta processar os *traces*	A aplicação forneceu *traces*
O algoritmo de análise processa os *traces*	A plataforma aumenta o número de instâncias dos microserviços da aplicação com a carga demasiado elevada
O algoritmo de análise tenta processar os *traces*	A aplicação não forneceu *traces*
O algoritmo de análise processa as métricas	A plataforma aumenta o número de instâncias dos microserviços da aplicação com a carga demasiado elevada

Requisitos Funcionais
REQF-16: A plataforma deve escalar a aplicação automaticamente, baseado em parâmetros de qualidade como, a título de exemplo, a latência, o *throughput* ou a morfologia dos microserviços
REQF-17: A plataforma deve seguir um caminho de degradação na análise dos *traces* e métricas, sendo que se não existirem *traces* disponíveis, devem ser analisadas as métricas

4.1.2 Atributos de qualidade

Ao desenhar uma arquitetura de uma aplicação, tem que se ter em atenção os atributos de qualidade. Estes têm um impacto grande na arquitetura, pois com base nestes conseguimos priorizar o que é mais importante quando se utiliza a aplicação. Assim, de seguida serão expostos os atributos de qualidades mais pertinentes, do mais importante para o menos, no formato de *Utility tree*. Posteriormente será feita uma descrição mais detalhada de cada atributo, explicando a sua importância para a plataforma.

Atributos de qualidade	**Refinamento**	**ASR**
Escalabilidade	Suportar um número elevado de aplicações	A plataforma precisa de conseguir crescer com o aumento do número de aplicações (A,A)
Elasticidade	Lidar com um aumento de carga	Com o aumento de carga, a plataforma tem que conseguir reagir de acordo, aumentando o número de instâncias da plataforma para suportar a carga. (M,A)
Desempenho	Executar os pedidos às aplicações em tempo útil	Os pedidos feitos às aplicações que correm na plataforma têm de ter resposta num tempo aceitável máximo de 3 segundos (M,A)
Disponibilidade	A plataforma tem que estar a funcionar	Sendo uma plataforma online que corre aplicações de clientes, esta tem que estar disponível 99.99% do tempo (A,A)
Confiabilidade	A plataforma tem que funcionar como esperado	A plataforma tem que ser capaz de responder em caso de falha das aplicações, lançando uma nova instância da aplicação caso seja necessário (M,A)

Tabela 4.1: Atributos de qualidade

A notação utilizada na tabela anterior tem como legenda A=Alto, M=Médio e B=Baixo. Esta notação serve para indicar, primeiramente, o impacto que cada atributo de qualidade tem na arquitetura, e em segundo lugar, o valor que cada atributo de qualidade tem no negócio.

Começando a analisar cada atributo de qualidade do mais importante para o menos, inicia-se com a Escalabilidade.
Este é o atributo que é colocado no topo pois trata-se do que mais tem mais impacto tanto na arquitetura como valor no negócio. Sendo uma plataforma que pode vir a ter muitos utilizadores que vão pôr as suas aplicações a funcionar, esta não pode falhar. A plataforma tem que conseguir crescer em número de nós presentes no sistema, não diminuindo a sua capacidade de resposta aos pedidos que chegam, bem como a sua capacidade de monitorização. A aplicação de cada utilizador da plataforma pode crescer com o tempo. Estes utilizadores querem ter confiança na plataforma, acreditando que esta consegue crescer com o aumento das suas aplicações, e com o aumento da carga, garantindo a disponibilização dos recursos necessários, bem como o desempenho esperado.

Logo a seguir a este, a Elasticidade é o atributo de qualidade mais importante. A plataforma, também relacionado com o referido em relação ao atributo de qualidade anterior, precisa de conseguir dar resposta ao a cargas elevadas feitas às aplicações que estão alojadas. Para isso, tem que ter propriedades elásticas, conseguindo aumentar os componentes

necessários para que a plataforma como um todo funcione sem problemas.

Em terceiro lugar aparece o Desempenho. Uma plataforma que tem como objetivo albergar diversas aplicações que podem ter requisitos de desempenho rígidos não pode ser a causa pela qual este desempenho é afetado. Assim, é preciso que a plataforma consiga garantir que este não é afetado, fazendo com que os pedidos às aplicações não demorem mais do que 3 segundos, um valor que no limite ainda pode ser considerado como aceitável por parte dos utilizadores. Este objetivo tem que ser cumprido ou então corre-se o risco de poder ser esta a razão pela qual a plataforma não tem sucesso.

Em penúltimo lugar, encontra-se o atributo de qualidade Disponibilidade. A plataforma tem que estar a funcionar corretamente a maior quantidade de tempo possível, porque como está a incorporar aplicações, qualquer *downtime* na plataforma, é *downtime* também nas aplicações. Assim, é de extrema importância que o tempo em que a plataforma está inoperacional seja reduzido ao mínimo possível, para causar o menor inconveniente nas aplicações. Ao se estabelecer que a plataforma tem que estar disponível 99.99% do tempo, isto quer dizer que ao longo de um ano, o *downtime* será de 0.01% x 365 dias = 3 dias, 15 horas e 36 minutos.

Por fim, analisa-se o atributo de qualidade Confiabilidade.
Este atributo é importante porque é o que procura que a plataforma mantenha um funcionamento normal, e que consiga responder a situações de extremo. No caso de uma das aplicações a corre falhar, a plataforma tem que conseguir reagir mantendo o funcionamento normal da aplicação. Isto pode ser alcançado lançando uma nova instância da aplicação, caso não se consiga resolver com o aumento, diminuição ou reposição de instâncias dos microserviços.

4.2 Arquitectura proposta

Para o desenvolvimento da arquitetura da plataforma que se pretende implementar, seguiu-se o modelo C4 de Simon Brown. Este é um modelo que tem como objetivo pensar na arquitetura em 4 níveis diferentes de abstração, sendo estes a nível do contexto (C1), contentores (C2), componentes (C3) e classes (C4)[38]. Irão ser utilizados os primeiros 3 níveis de abstração (C1 a C3), pois nesta fase ainda não estão definidas as classes (C4). Assim, nas próximas subsecções irão estar descritos os 3 diagramas de níveis de abstração diferentes.

4.2.1 Diagrama de Contexto (C1)

O Diagrama de Contexto (C1) é o ponto de partida no desenvolvimento de uma arquitetura. É o diagrama mais abstrato em relação ao software que se pretende implementar, permitindo uma visão geral dos principais componentes e intervenientes na arquitetura.

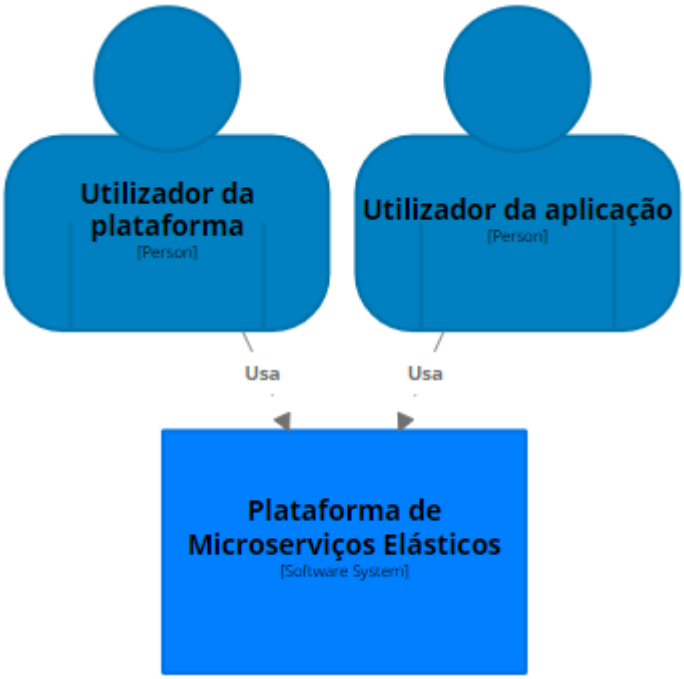

Figura 4.1: Diagrama de Contexto (C1)

A plataforma de microserviços elásticos é uma plataforma que permite que os seus utilizadores possa fazer *deploy* da sua aplicação, e permite que outros utilizadores possam fazer uso dessa aplicação. Assim, como se pode ver na figura 4.1, existem dois tipos de utilizadores: os utilizadores da plataforma e os utilizadores das aplicações que estão a correr na plataforma.
Estes utilizadores irão aceder às funcionalidades de duas maneiras diferentes:

- *API* de controlo
- *API* de entrada para a aplicação

Os utilizadores da plataforma acederão através da *API* de controlo, para realizarem a gestão da sua aplicação, como descrito nos requisitos funcionais na secção 4.1.1. Já os utilizadores das aplicações irão aceder a estas através da *API* de entrada para as aplicações.

4.2.2 Diagrama de Contentores (C2)

O Diagrama de Contentores (C2) é um aprofundar do que foi apresentado no Diagrama de Contexto (C1). Neste diagrama detalham-se melhor os contentores que fazem parte da Plataforma de Microserviços Elásticos.

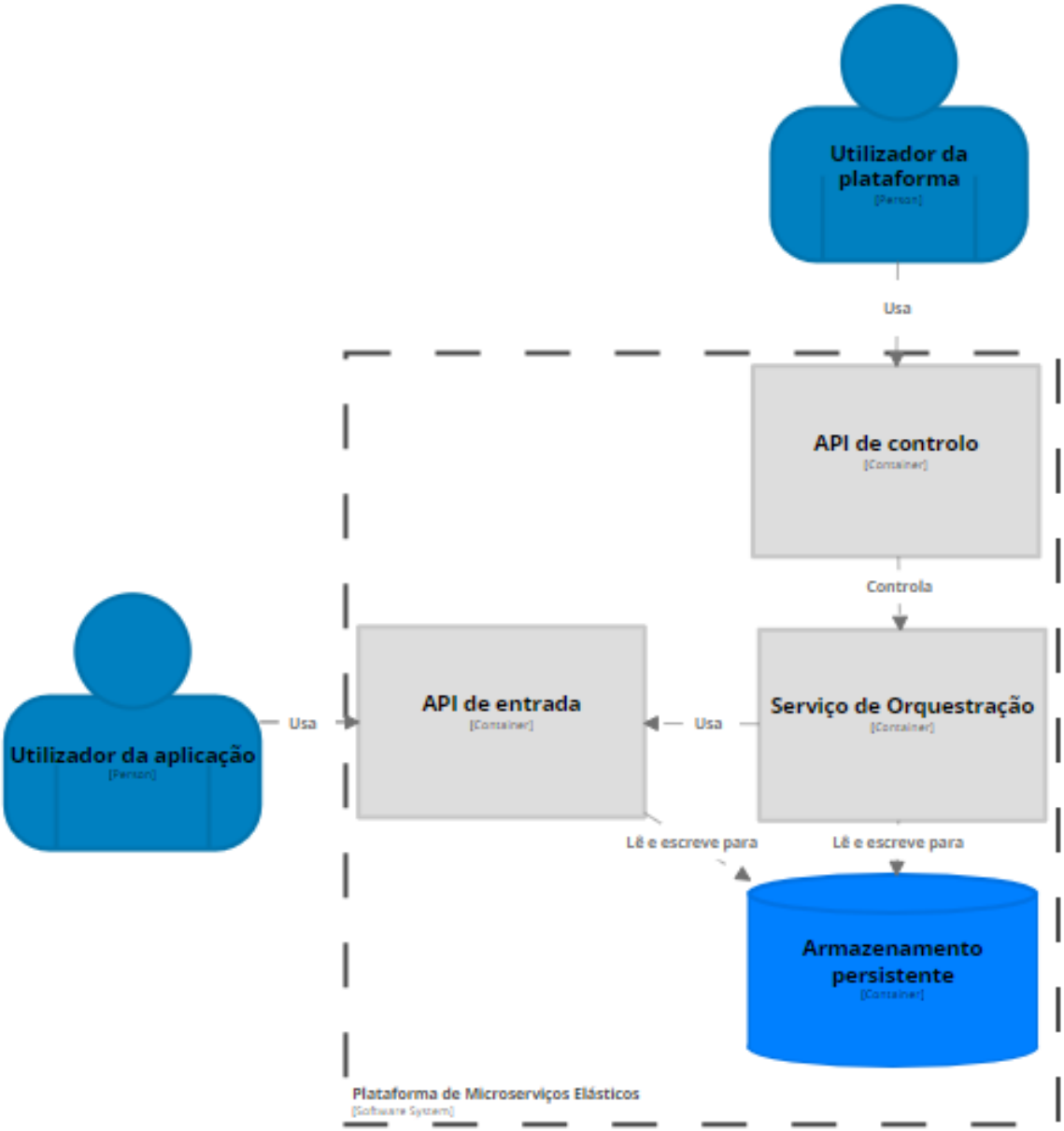

Figura 4.2: Diagrama de Contentores (C2)

A plataforma será constituída por 4 contentores principais:

- *API* de controlo
- Serviço de Orquestração
- Armazenamento persistente

- *API* de entrada

Cada um destes contentores terá funções específicas, que têm como objetivo permitir que o que ficou definido nos requisitos seja cumprido.

Relativamente ao *API* de controlo, este tem como função principal comunicar com o Serviço de Orquestração. Isto permite que seja possível fazer *deploy* ou gerir aplicações, entre outras funções que estão definidas nos requisitos funcionais na secção 4.1.1. Este *API* de controlo poderá ser implementado como um servidor *web*.

O Serviço de Orquestração é a peça central da Plataforma de Microserviços Elásticos. É este serviço que gere a infraestrutura, que programa e comanda os microserviços que se encontram dentro de contentores e que realiza o *load balancing*. Para além disto, o Serviço de Orquestração é responsável por disponibilizar a ligação entre os utilizadores das aplicações e as próprias aplicações.

Para criar essa ligação, é disponibilizada uma *API* de entrada para as aplicações, para que os utilizadores das aplicações possam fazer uso destas.

Por fim, o Armazenamento permanente serve para guardar de forma segura os dados tanto das aplicações que estão a correr na plataforma, como também as métricas e os *traces* retirados pela Plataforma em relação às aplicações enquanto estas são monitorizadas.

4.2.3 Diagrama de Componentes (C3)

Por fim, é feito o Diagrama de Componentes (C3). Este é uma visão mais pormenorizada de como é constituído cada contentor, sendo explicitados os componentes chaves de cada um, e a maneira como estes interagem.
Assim, devido a limitações da ferramenta utilizada para a criação de diagramas, que não permitiu mostrar os componentes dos dois contentores ao mesmo tempo, dois diagramas foram criados, um para o Serviço de Orquestração e outro para o Armazenamento persistente.

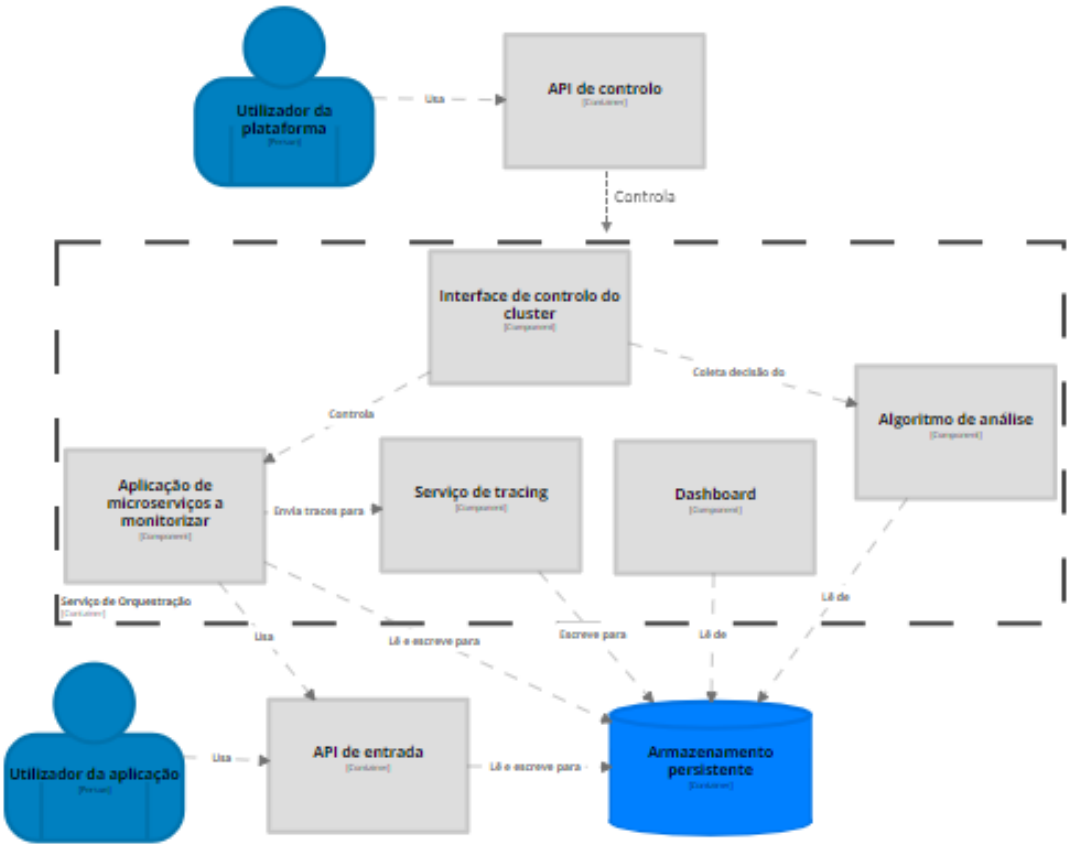

Figura 4.3: Diagrama de Componentes do Serviço de Orquestração (C3)

O Serviço de Orquestração é constituído por várias componentes fundamentais, que vão permitir a realização das funcionalidades especificadas na secção 4.1.1.
Para começar este serviço necessita de acomodar aplicações dos utilizadores da plataforma. Assim, o primeiro componente que é referido são as próprias aplicações de microserviços que vão ser monitorizadas. Estas são constituídas por microserviços que irão ser escalados conforme as necessidades detetadas com as métricas e *traces* retirados.
Embora não pertencente a este contentor do Serviço de Orquestração, para que os utilizadores das aplicações possam fazer uso destas, uma *API* de entrada tem que ser disponibilizada, sendo que esta *API* é um contentor fundamental. Esta vai também ajudar com a monitorização.
Para a realização da monitorização duas técnicas serão utilizadas, como referido em cima: métricas retiradas a partir da *API* de entrada e *traces* obtidos através de um serviço de rastreamento distribuído (*tracing*).
Outro componente que constitui o Serviço de Orquestração é um *Dashboard*. Este vai permitir ao utilizador da plataforma ter acesso à representação visual de métricas retiradas

da sua aplicação, para que este possa ter uma precessão do que está a acontecer.
Por fim para a plataforma conseguir cumprir a sua função principal, elasticidade dos microserviços que estão a ser monitorizados, é que preciso que esta consiga analisar as métricas e os *traces* de maneira a ser capaz de perceber onde é preciso escalar os microserviços. O algoritmo de análise sai fora do âmbito deste trabalho, pelo que irá ser implementado um algoritmo simbólico para testar se a plataforma funciona como é pretendido. As decisões tomadas pelo algoritmo irão ser recolhidas pelo componente Interface de controlo do *cluster*, que irá implementar o decidido, alterando o necessário na aplicação de microserviços.

Como dito anteriormente, devido a limitações da ferramenta utilizada para a criação dos diagramas, é agora analisado em separado o Armazenamento persistente.

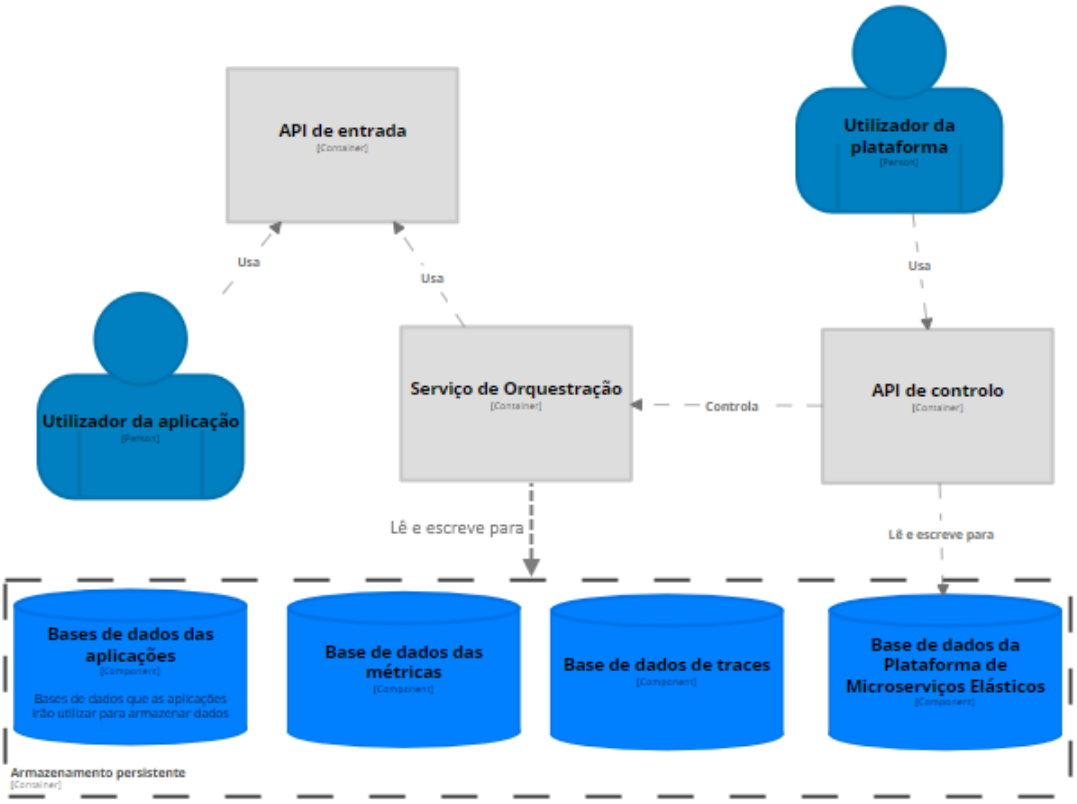

Figura 4.4: Diagrama de Componentes do Armazenamento persistente (C3)

Este Armazenamento persistente é a solução encontrada para a preservação de todos os dados, sendo estes dados pertencentes a diversos componentes do Serviço de Orquestração. O Armazenamento persistente encontra-se dividido em várias bases de dados de suporte aos componentes. Em primeiro lugar, é preciso uma base de dados da Plataforma de Microserviços Elásticos. Esta irá ter como função guardar dados relativos aos utilizadores da plataforma, entre outros dados. De seguida, serão necessárias bases de dados que as aplicações que correm necessitam para o seu correto funcionamento. Em terceiro lugar, serão essenciais pelo menos duas bases de dados para albergar os dados relativos à monitorização. Assim existirá uma base de dados das métricas retiradas pela *API* de entrada e uma base de dados que tem como objetivo manter os dados retirados pelo serviço de rastreamento distribuído, os *traces*. A partir destas bases de dados, o algoritmo de análise irá ter acesso à monitorização feita na aplicação.

4.2.4 Tecnologias a utilizar

Como analisado anteriormente, existem 4 grandes contentores na Plataforma de Microserviços Elásticos, que se apoiam em tecnologias para atingir os seus objetivos. O Serviço de Orquestração é constituído por diferentes componentes, os quais também necessitam de utilizar tecnologias para desempenharem a sua função. De seguida, irá ser feita uma exposição de possíveis tecnologias a utilizar.

Tecnologias terão que ser escolhidas para os seguintes contentores e componentes:

- *API* de controlo
- Serviço de Orquestração
- Armazenamento persistente
- *API* de entrada para a aplicação
- *Dashboard*
- Serviço de *tracing*

Relativamente ao *API* de controlo, como já dito em cima, pode ser implementada como um servidor *web*. Assim algumas tecnologias destacam-se para esta função, nomeadamente *Django*[39], *Flask*[40] ou *web2py*[41].
O Serviço de Orquestração tem que ser implementado com recurso uma tecnologia de *Cluster Management*. Isto porque o objetivo é albergar aplicações de microserviços, fazendo a sua gestão quer a nível de *load balancing*, quer a nível de elasticidade automática. Com esta tecnologia será também possível fazer a monitorização de maneira mais fácil, com a ajuda de ferramentas para esse fim. Alguma ferramentas já estudadas no Capítulo 2 são possíveis soluções para este serviço, como o *Kubernetes*[25] ou o *Docker Swarm*[42], que permitirão o cumprimento de vários atributos de qualidade como a Escalabilidade, Elasticidade e a Confiabilidade.
A utilização de um Serviço de Orquestração tem como consequência a utilização de contentores para o isolamento dos microserviços. Como estes contentores são efêmeres, e podem ser terminados e iniciados várias vezes, estes precisam de um maneira de guardar os dados de forma permanente. Por esta razão é preciso a utilização de Armazenamento persistente. Uma possível solução de armazenamento persistente, associado a instâncias a correr no *Google Compute Engine* ou no *Google Kubernetes Engine*, são os *Google Persistent Disk* que são uma solução de armazenamento em blocos duráveis e de alto desempenho na *Google Cloud Platform*[43].
Para a API de entrada para a aplicação, tecnologias como *Zuul*[30] servem de serviço de *edge* que fornecem encaminhamento dinâmico, monitorização, resiliência e segurança, criando assim um ponto único para consumo da aplicação, e permitindo também a monitorização dos pedidos.
Para a apresentação das informações aos utilizadores da plataforma sobre as suas aplicações, uma solução que permita a criação de diferentes vistas da informação recolhida em forma de métricas é precisa. Assim soluções como o *Grafana*[29] permitem a criação de diferentes dashboards, dando ao utilizador várias maneiras de observar a informação.
Por fim, um serviço de rastreamento distribuído (*tracing*) é necessário para a obtenção de *traces*. Existem várias soluções disponíveis para este fim, tendo duas delas sido faladas no Capítulo 2: *OpenTracing* e *OpenCensus*. Existem ainda outras opções como *Zipkin*[44] ou *Jaeger*[45].

No segundo semestre irá haver um período de tempo dedicado à experimentação de várias destas tecnologias, de maneira a perceber se estas são indicadas para o fim desejado, levando a uma posterior escolha.
A escolha terá como fundamento as funcionalidades de cada tecnologia testada, tendo estas que preencher o requerido pelos requisitos funcionais e atributos de qualidade. Outros aspetos a ter em conta serão a facilidade de utilização desses tecnologias, bem como o seu tamanho e compatibilidade entre estas.

Capítulo 5

Implementação

Neste capítulo é descrito e analisado o trabalho desenvolvido no segundo semestre desta dissertação, que diz respeito à implementação prática da proposta descrita no Capítulo 4.

Para uma descrição mais percetível, este capítulo foi divido nas partes constituintes da arquitetura desenvolvida.

5.1 Solução final

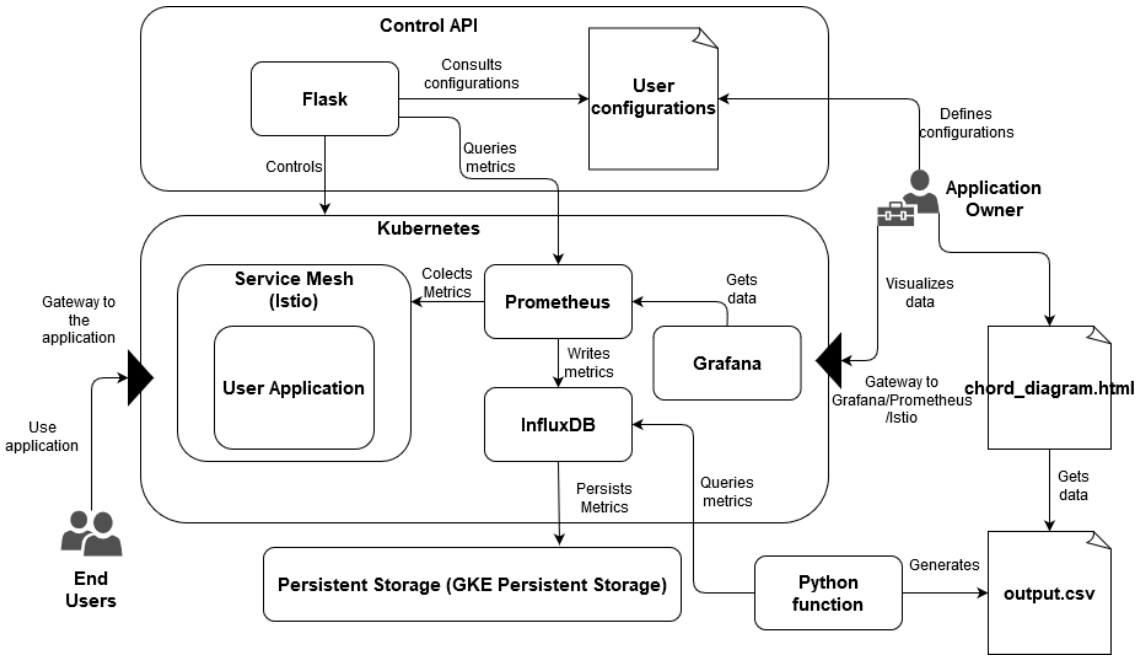

Figura 5.1: Diagrama da arquitetura da solução final

Para uma melhor compreensão do funcionamento da solução implementada, bem como para se ter noção de como é que cada um dos componentes se utilizam uns aos outros, enquanto a sua implementação é descrita ao longo deste capítulo, foi criado este diagrama (figura 5.1). Vai permitir ter uma visão geral de todos os componentes implementados ou utilizados, e também quais são os intervenientes na sua utilização.

5.2 Aplicações de microserviços

De maneira a se conseguir acompanhar e verificar o progresso do desenvolvimento desta plataforma, foi necessário o uso de aplicações de microserviços, para comprovar que as funcionalidades implementadas estavam a funcionar. Para isso, num primeiro momento, foi utilizada a aplicação de microserviços *BookInfo*, aplicação de microserviços desenvolvida e disponibilizada para testes pelos criadores do *Istio*. Num segundo momento, foi utilizada outra aplicação de microserviços, a *Robot-Shop*, uma aplicação desenvolvida e disponibilizada pelos criadores do *Instana*.

5.2.1 *BookInfo*

Como já foi mencionado, a primeira aplicação de microserviços utilizada foi a aplicação *BookInfo*. Esta aplicação é disponibilizada pelos criadores do *Istio* para facilitar a implementação e testes à *service mesh Istio*. Devido a ter sido utilizada esta tecnologia, como será analisado mais à frente, para os primeiros testes foi utilizada esta aplicação.

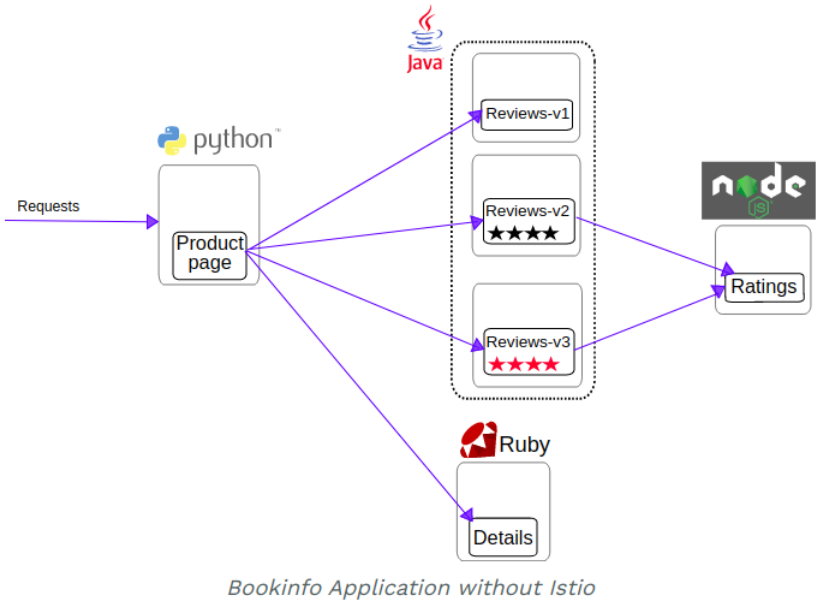

Figura 5.2: Aplicação de microserviços *BookInfo*[46]

Esta aplicação é constituída por 4 microserviços (figura 5.2:

- *Details*
- *Product Page*
- *Ratings*
- *Reviews*

Tendo em conta que a aplicação apenas é constituída por 4 microserviços, esta considera-se mais um protótipo do que uma aplicação, nem sempre espelhando as exigências de muitas situações reais. Por esta razão, foi utilizada uma segunda aplicação, um pouco mais complexa.

5.2.2 *Robot-Shop*

Como exemplo mais complexo, foi utilizada a aplicação de microserviços *Robot-Shop*. Trata-se de uma aplicação disponibilizada pela ferramenta de visualização de aplicações denominada *Instana*, para usar como teste de maneira a demonstrar as suas funcionalidades. No âmbito da presente tese, foi usada como um exemplo maior e mais completo do que poderá ser uma aplicação de microserviços realista, permitindo aferir que independentemente do tamanho da aplicação, a ferramenta desenvolvida no âmbito deste trabalho cumpre a sua função.
A *Robot-Shop* é constituída por diversos microserviços que comunicam entre si, sendo estes:

- *Cart*
- *Catalogue*
- *Dispatch*
- *Mongo*
- *MySQL*
- *Payment*
- *Ratings*
- *Shipping*
- *User*
- *Web*

Trata-se uma aplicação com 10 microserviços, envolvendo assim uma maior complexidade.

5.3 *Cluster Manager*

Para se poder realizar a orquestração dos microserviços, foi preciso recorrer aos já mencionados *Cluster Managers*. As duas opções analisadas no Capítulo 2 (*Kubernetes* e *Docker Swarm*) apresentam vantagens e desvantagens, e com base nestas foi tomada uma decisão sobre qual utilizar.
O *Cluster Manager* escolhido foi o Kubernetes, pelas suas capacidades de orquestração, bem como outros factores evidenciados mais à frente.

5.3.1 *Kubernetes*

O *Kubernetes* trata-se de um serviço de orquestração de microserviços que pode ser corrido em *baremetal*, necessitando no entanto de um conjunto significativo de configurações, e de diversos recursos, algo que foi descrito pelo colega Fábio Ribeiro, sendo por esse motivo uma tarefa com exigência temporal muito elevada. Por esta razão, partiu-se logo para a utilização de *Kubernetes* num serviço de *cloud*, neste caso o *Google Kubernetes Engine*.
Embora o serviço em *cloud* facilite a instalação de um *cluster* em *Kubernetes*, este não retira a curva de aprendizagem que é preciso percorrer para uma utilização correta e funcional deste serviço de orquestração. A instalação e configuração dos microserviços continua a ter que ser feita manualmente, ou com recurso a ficheiros de configuração.

Google Kubernetes Engine

A utilização do *Kubernetes* num serviço de *Cloud* tem como vantagem evitar a definição de configurações, como definições de rede ou de máquinas físicas. Tudo isto é tratado automaticamente, permitindo poupar tempo, e no âmbito deste trabalho, deixando mais tempo para a implementação da plataforma.
Utilizando o *GKE UI*, a instalação de um *cluster* fica simplificada, sendo possível declarar definições como o número de núcleos deste *cluster*, capacidades de auto-escalonamento do *cluster* em caso de aumento do número de microserviços, evitando ter que definir os recursos necessários para o correto funcionamento do *cluster* (figura 5.3).

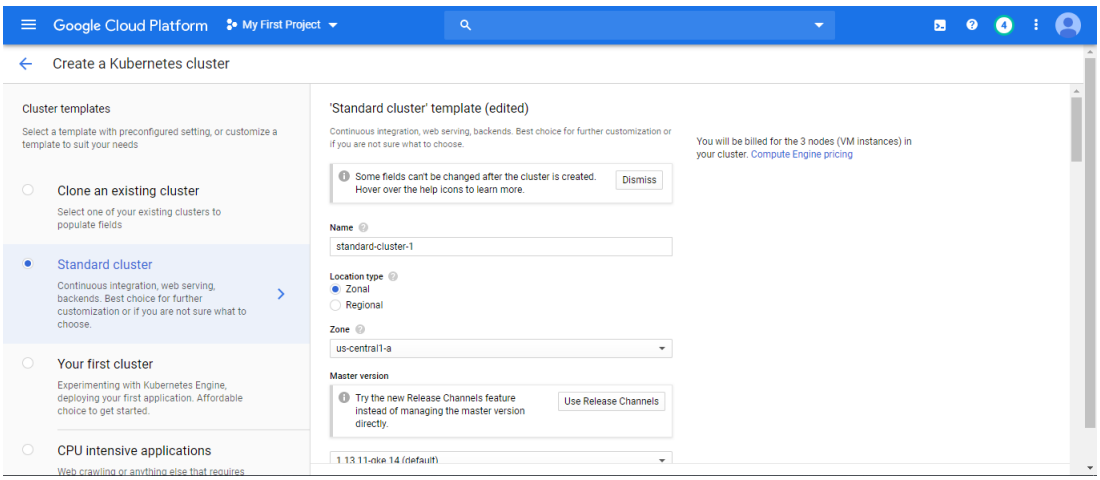

Figura 5.3: *Google Kubernetes Engine* - criação de um *cluster*

5.4 Monitorização

O primeiro passo para a implementação desta plataforma seria começar pela recolha de métricas que iriam permitir a tomada de decisão em relação ao *cluster*, de maneira a mitigar os problemas encontrados na sua análise. Assim, para a recolha de métricas haveria duas hipóteses: as *API Gateway* ou as *Service Mesh*. Por se tratar de uma tecnologia que surgiu mais recentemente, tendo vindo a conseguir realizar tudo o que as *API Gateway* conseguem, oferecendo informações ainda mais completas, foi tomada a decisão de trabalhar com as *Service Mesh*.

5.4.1 *Service Mesh*

No que toca a *Service Meshes*, existem algumas soluções disponíveis, sendo que a maior parte delas ainda se encontra num estado inacabado ou embrionário, pelo que a *service mesh* escolhida foi o *Istio*, pois é a única que se encontra num estado avançado o suficiente para ser utilizável.

Istio

O *Istio* é uma tecnologia que tem como objetivo conectar, controlar e permitir observação sobre um conjunto de microserviços.

Foi a primeira tecnologia configurada e corrida no decurso do desenvolvimento da ferramenta, juntamente com a aplicação de microserviços *Book-Info*. Isto permitiu perceber quais seriam as métricas que efetivamente se conseguiam retirar. O *Istio* utiliza *sidecars* colocados antes dos microserviços (como analisado na secção 2.5), controlando o tráfego que é gerado com os pedidos, bem como servindo de *load balancer*, distribuindo os pedidos conforme as instâncias disponíveis. Certos ficheiros de configuração tiveram de ser alterados de forma a providenciar as métricas necessárias.
As métricas são enviadas para o *Prometheus*, conseguindo-se assim realizar *queries* sobre estas. De forma a ter uma visualização destas métricas, em conjunção com o *Prometheus* é utilizada a ferramenta de *dashboards* denominada de *Grafana*.
Para colocar o *Istio* a correr foi criado um *script*, *base_ script.sh*, que faz o *deploy* do *Istio*, juntamente com o *deploy* da aplicação de microserviços que se pretende monitorizar. De modo a se poder aceder aos dados do *Prometheus* e à interface gráfica do *Grafana* foram criados ficheiros, *prometheus-gateway.yaml* e *grafana-gateway.yaml*. Estes ficheiros definem os caminhos que o utilizador pode usar para aceder ao descrito acima, utilizando a *gateway* já criada pelo *Istio*. Assim, para se aceder ao *Grafana* o utilizador usará o *IP* externo atribuído à *gateway* do *Istio*, juntamente com o caminho */dashboard* (A título de exemplo, "http://34.65.67.80:80/dashboard"). Já para se realizaram *queries* ao *Prometheus*, utiliza-se o mesmo *IP*, seguido de */api/datasources/proxy/1/api/v1/query_ range*.

Prometheus

Como parte constituinte do *Istio* é utilizado o *Prometheus*. O Prometheus é um *toolkit opensource* de monitorização e alerta de sistemas desenvolvido originalmente no *SoundCloud*[47] (figura 5.4). Tem um modelo de dados por séries cronológicas que se podem identificar pelo nome das métricas e pares chave/valor, tendo uma linguagem de *queries* própria, *PromQL*.
O *Prometheus* funciona retirando as métricas dos seus alvos, e guardando estes dados com carimbos temporais, permitindo a navegação por estes, bem com a sua análise em diversos pontos temporais.
Juntamente com a ferramenta na secção 5.7.1 descrita, *Grafana*, é possível obter uma visualização das métricas retiradas.

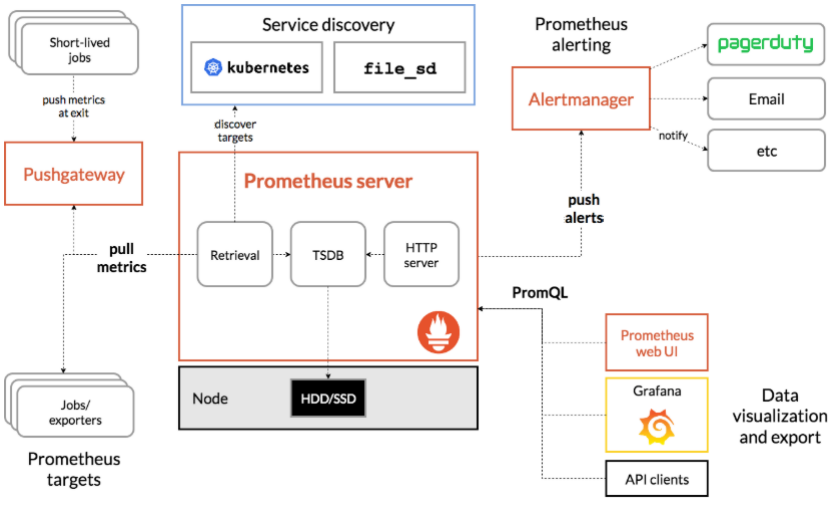

Figura 5.4: Arquitetura do *Prometheus*[47]

5.4.2 *Instana*

Com a intenção de tentar perceber quais são as capacidades da *APM* (*Application Performance Management*) denominada *Instana*, e que tipo de informações seria possível retirar e visualizar, esta foi aplicada ao *cluster* que corria a aplicação de microserviços que se pretende monitorizar.
A *Instana* é uma ferramenta que tem como objetivo a monitorização da *performance* de uma aplicação de microserviços, permitindo a visualização 3D da performance através de gráficos gerados com *machine learning*, criando também alertas sobre a *performance* da aplicação.
Com esta ferramenta foi possível obter uma visualização 3D de todo o *cluster*, bem como dados relativos à sua utilização de *CPU*, memória *RAM*, entre outros dados.
No entanto, para além de se tratar de uma solução paga, para funcionar corretamente é preciso correr *workers* paralelamente a cada microserviço, levando a um elevado aumento da quantidade de recursos necessários para correr o *cluster*, não trazendo vantagens em relação aos dados retirados com o *Istio*.

5.5 Armazenamento Persistente

A necessidade de utilização de armazenamento persistente advém do facto de que o *Prometheus*, embora tenha armazenamento temporário das métricas retiradas, não guarda os dados a longo prazo. Para este fim é necessário a integração com armazenamento fora da ferramenta, nomeadamente a utilização de outras bases de dados. Assim, o *Prometheus* está preparado para escrever os dados para outros armazenamentos, permitindo a utilização destes ao longo do tempo.

Figura 5.5: *Prometheus external storage*[48]

O *Prometheus* vem preparado para esta situação, e já fornece maneira de fazer um *remote write* (uma escrita remota para um endereço *IP*), exportando as métricas. De seguida, no caso da solução escolhida já estar preparada para dados provenientes do *Prometheus*, esta fará um *parse* dos dados, fazendo as alterações necessárias para que consiga armazenar os dados corretamente (figura 5.5).
Durante o desenvolvimento deste trabalho foram utilizadas duas soluções de armazenamento permanente, tendo no final ficado em uso uma das soluções.

5.5.1 *Victoria Metrics*

Em primeiro lugar foi feita a implementação do armazenamento permanente denominado *Victoria Metrics*. Trata-se de uma base de dados temporal, que armazena os dados como são exportados pelo *Prometheus*.
Esta apresenta as vantagens de suportar a linguagem de *query* nativa do *Prometheus*, *PromQL*, bem como a velocidade com que consegue ingerir dados do *Prometheus* (figura

5.6).

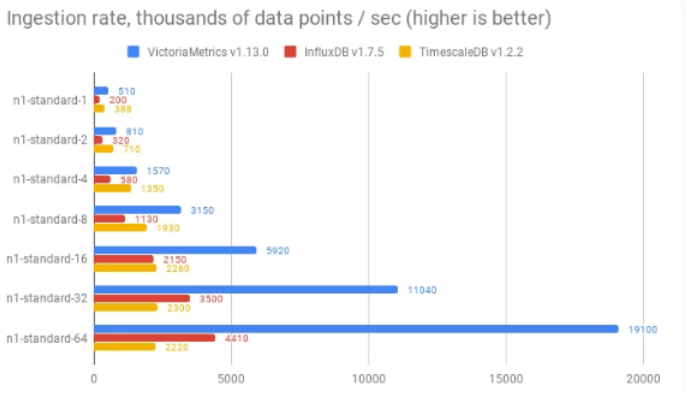

Figura 5.6: Taxa de ingestão de dados *Victoria Metrics*[49]

No entanto, após a implementação desta solução, embora se tenha conseguido colocar o *Grafana* a consumir as métricas a partir do *Victoria Metrics*, o mesmo não foi possível para a solução em *Flask*, no algoritmo de decisão. Por esta razão, teve que se repensar qual a solução de armazenamento que iria ser utilizada.

5.5.2 *InfluxDB*

Pela razão acima descrita, o *InfluxDB* foi a solução encontrada para o armazenamento a longo prazo. Trata-se de uma base de dados temporal que tem suporte nativo para armazenamento de dados exportados pelo *Prometheus*. Todavia, ao contrário do *Victoria Metrics*, o *InfluxDB* não armazena os dados na mesma estrutura do *Prometheus*, fazendo um *parse*, realizando as seguintes alterações[50]:

- O nome da métrica em *Prometheus* transforma-se no nome da *measurement* em *InfluxDB*
- O valor da amostra em *Prometheus* transforma-se num campo denominado *value* em *InfluxDB*, sendo sempre um *float*
- As *labels* de *Prometheus* tornam-se em *tags* de *InfluxDB*

Quando estas alterações são feitas, o suporte da linguagem *PromQL* deixa de existir, tendo que os dados ser obtidos através da linguagem *InfluxQL* (*Influx Query Language*). Este facto trouxe problemas na obtenção dos dados necessários, pois as linguagens não são iguais, nem permitem obter os dados da mesma maneira. No futuro seria ideal criar novos *dashboards* no *Grafana*, utilizando a linguagem *InfluxQL*, pois os dados apresentados com recurso a essa ferramenta são os dados que estão alojados no *Prometheus*, permitindo assim ver apenas as últimas seis horas de dados. Porém, o *datasource* do *InfluxDB* está configurado no *Grafana*, pelo que a ferramenta está pronta para a criação dos *dashboards*.

5.6 Algoritmo de decisão

Para a tomada de decisão, era necessário realizar o desenvolvimento de um controlador. Devido à facilidade inerente à linguagem de programação *Python* e por existirem

frameworks que permitem tanto realizar *queries* ao *InfluxDB*, como ao *Prometheus*, bem como realizar alterações ao *cluster* que está a correr no *GKE*, foi utilizado o *framework Flask*.
Com o *Flask* é possível criar serviços *web*, permitindo a criação de lógica num ambiente de *server*. As funções que foram desenvolvidas foram as seguintes:

- get_apps(name, namespace=None)
- scale_applications(name, namespace=None, number_of_replicas=None)
- dummy_decision_algorithm()

As duas primeiras funções utilizam a *framework* denominada *kubernetes* para *Python*, e fazem uso de dois ficheiros retirados do *GKE*: *kube-config* e *google-application-credentials.json*. Estes ficheiros permitem obter a autorização necessária para aceder à *API* da *Google* e ao *cluster* em si, podendo consultar-se informações sobre este e realizar alterações que, no nosso caso, é a alteração do número de instâncias de um microserviço.
Com a primeira função, *get_apps*, pretende-se obter o *deployment* do microserviço que se vai modificar, para que este possa ser utilizado na função *scale_applications*, alterando os seus atributos, para realizar as modificações necessárias no *cluster*. A utilização da função *api_instance.read_namespaced_deployment*, armada com o nome do microserviço, e o *namespace* onde este se encontra alojado, permitem tal obtenção.
A segunda função, *scale_applications*, tem como objetivo escalar o microserviço que precisa de ser ajustado. Atribuindo um novo valor ao atributo *deployment.spec.replicas* e de seguida chamando a função *api_instance.patch_namespaced_deployment*, altera-se o número de réplicas pretendidas no *cluster*, criando alterações efetivas aumentando o número de instâncias do microserviço.
Por fim, a terceira função, *dummy_decision_algorithm*, é a função que toma a decisão em relação aos microserviços. Tem em conta os percentis do tempo de resposta 95% e 99% dos pedidos realizados no últimos 5 minutos, e com base nestes valores tenta alterar o *cluster* como necessário para conseguir tentar atingir os valores definidos pelo utilizador. Passados 30 segundos, volta a fazer a mesma comparação, e assim sucessivamente.

5.7 Visualização de métricas

Para que o utilizador possa compreender o que está a acontecer com a sua aplicação, é importante que este possa visualizar alguns dados que indicam o estado do *cluster*. Assim, foi utilizada uma ferramenta, *Grafana*, que permite uma representação gráfica das métricas que são retiradas durante o funcionamento do *cluster*.

5.7.1 Grafana

O *Grafana*, como dito anteriormente, trata-se de uma ferramenta de *dashboards* que permite a visualização de dados sob forma de gráficos e tabelas, entre outros.
O *Istio* já vem incluído com os ficheiros de configuração base para um conjunto grande de diferentes tipos de gráficos e dados, como por exemplo os percentis de tempo de resposta (50%, 90% e 99%), a taxa de sucesso dos pedidos de um certo serviço, a utilização de CPU de cada microserviço, sendo estes apenas alguns dos exemplos configurados.
As *dashboards* que estão implementadas são as seguintes:

- *Istio Galley*
- *Istio Mesh*
- *Istio Mixer*
- *Istio Performance*
- *Istio Pilot*
- *Istio Service*
- *Istio Workload*

Estes *dashboards* representam diferentes tipos de dados que foram extraídos com a ajuda do *Istio* e do *Prometheus*. Analisando mais a fundo as *dashboards Istio Mesh* e a *Istio Service*, vemos que estas duas colocam ênfase aos dados de duas perspetivas diferentes, sendo que a primeira mais geral sobre todo o *cluster*, e a segunda mais específica, expondo dados relativos a cada microserviço, permitindo assim que o utilizador tenha muita e diversa informação sobre a sua aplicação.
Na primeira, *Istio Mesh*, podem-se observar dados como a taxa de sucesso e os percentis dos tempos de resposta dos microserviços que estão a ser utilizados no momento da consulta à página (figura 5.7).

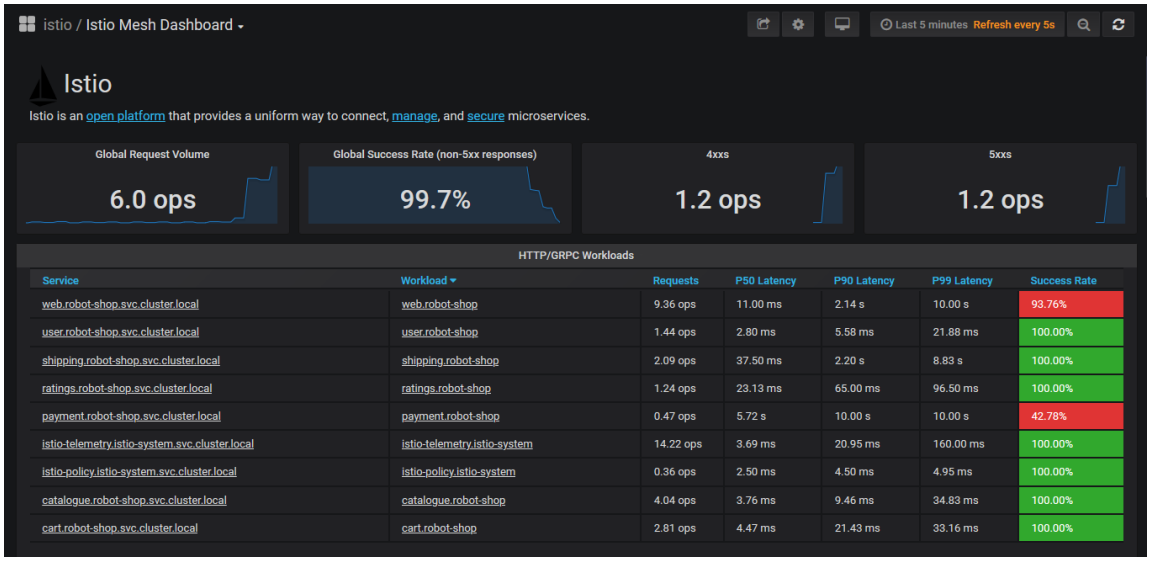

Figura 5.7: *Grafana - Istio Mesh dashboard*

Na segunda, *Istio Service*, o utilizador pode, através de um *dropdown menu*, escolher qual o serviço do qual pretende ver informação mais detalhada, e tem acesso a um conjunto de gráficos que mostram diferentes dados, como o tempo de resposta, percentis do tempo de resposta, o código das respostas e o tamanho dos pedidos (figura 5.8).

As restantes *dashboards* referem-se aos dados retirados sobre alguns dos microserviços que constituem o *Istio*, como o *Istio Pilot*, que permite ao *Istio* capacidades de gestão do tráfego de pedidos, ou o *Istio Galley*, que fornece serviços de gestão de configuração.

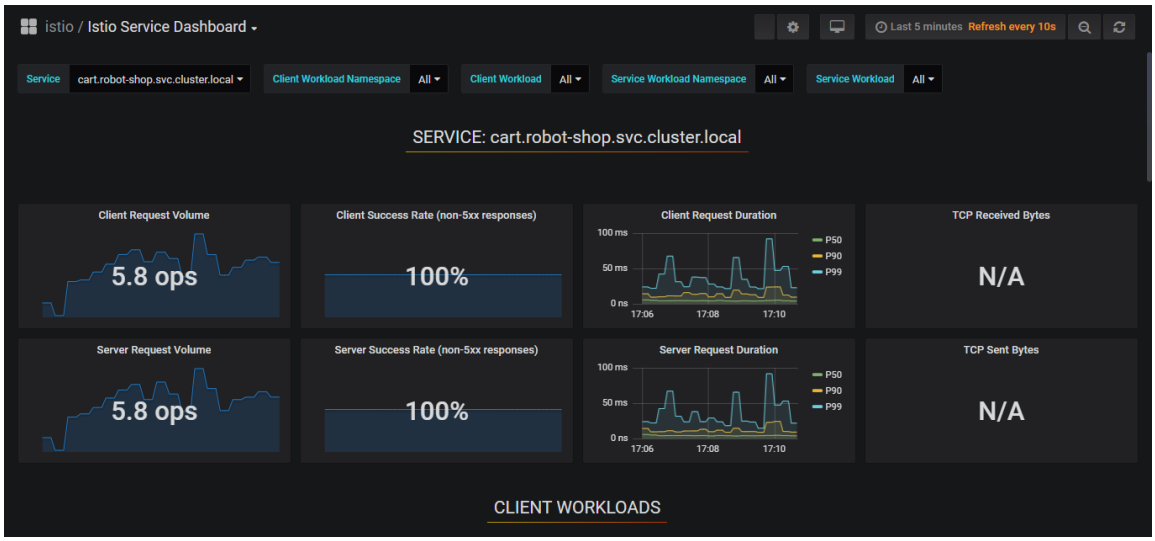

Figura 5.8: *Grafana - Istio Service dashboard*

5.7.2 Diagrama de cordas

Para uma visualização mais intuitiva, foi criado um diagrama de cordas. Este tipo de diagramas representa as ligações, neste caso, entre microserviços. Assim, é mais fácil perceber quais serão os microserviços que têm afluência por parte dos outros. Neste caso, o diagrama representa apenas as ligações, não incluindo os pesos das ligações (tendo em conta número de vezes que essa chamada foi feita).

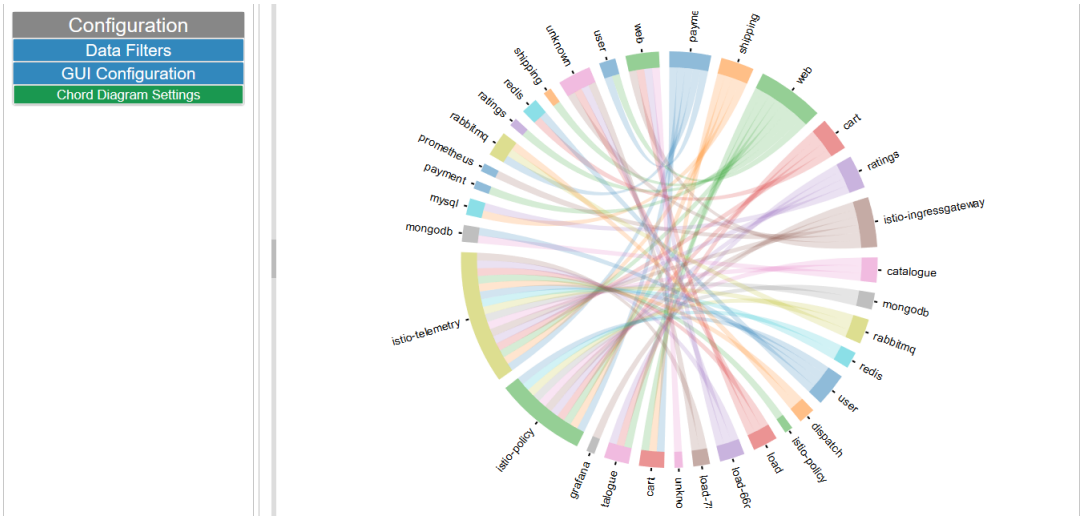

Figura 5.9: Diagrama de cordas

Para retirar os dados para utilizar no diagrama foi criada uma função em *Python* que comunica com o *InfluxDB* através da *framework* denominada *influxdb* para *Python*. Através de uma *query* de *InfluxQL* retiram-se os dados relativos aos pedidos realizados, e tratam-se os dados, escrevendo-os num ficheiro *output.csv*. Este ficheiro é posteriormente utilizado por código em *JavaScript* embutido em *HTML*, sendo possível visualizar o resultado num *browser*. Para a criação do diagrama de cordas foi utilizada uma *library* de *JavaScript* denominada *D3.js*, que foi criada para a manipulação de documentos baseados em dados.

Capítulo 6

Resultados

Para confirmar que a plataforma desenvolvida tem as funcionalidades necessárias para se poder considerar a presente implementação um sucesso, é preciso realizar alguns testes. Assim, este capítulo será dedicado a descrever alguns testes realizados, que serviram para confirmar o funcionamento da plataforma.

O primeiro teste realizado, ainda durante o desenvolvimento, foi a utilização de mais do que uma aplicação de microserviços. A plataforma tem como objetivo albergar aplicações de diferentes utilizadores. Por esta razão, foi necessário testar diferentes aplicações de microserviços, e como já foi descrito na secção 5.2, foram utilizadas duas aplicações distintas, com tamanhos e características diferentes, *BookInfo* e *Robot-Shop*. Isto permitiu confirmar que independentemente do tamanho das aplicações, tanto o *Kubernetes* como o *Istio* estão preparados para albergar as diferentes aplicações que possam aparecer.
Um segundo teste realizado foi relativo aos dados retirados com a ferramenta *Istio*. Após se ter feito o *deploy* de uma aplicação de microserviços, foi preciso observar se métricas estavam a ser retiradas. Para isso foram gerados pedidos, num primeiro momento no *browser*, utilizando a aplicação manualmente, e num segundo momento sendo empregue um gerador de pedidos, que automaticamente cria os pedidos que fazem uso dos diferentes microserviços contidos na aplicação. Com este tráfego de pedidos gerado, foi possível através da ferramenta *Grafana* perceber que dados estavam a ver retirados e guardados.
Para garantir que a aplicação estaria a escalar como esperado após correr as funções criadas em *Python*, vários testes foram feitos. Começou-se por aumentar num número específico de instâncias um microserviço. Após correr a função, foi verificado que o número de instâncias aumentou dentro do *cluster*. De seguida modificou-se outra vez o número, e chamando a função o número de instâncias diminuiu. Assim foi possível perceber que a função de escalonamento estava funcional.
Por questões de segurança, é preciso ter os ficheiros referidos na secção 5.6 para se poder, tanto obter informações, como realizar alterações no *cluster*. Tentando realizar um pedido de alteração (a título de exemplo, um pedido de mudança do número de instâncias de um microserviço) sem estes ficheiros mostra que é impossível fazer alterações neste. Assim, o *cluster* está protegido de alterações indevidas por parte de outros utilizadores.
Por fim, foi testado o *datasource* do *InfluxDB* dentro do *Grafana*, tendo sido feitas algumas *queries* de teste, obtendo resultados. Assim, fica confirmado que os dados estão a ser escritos para o *InfluxDB*. Ainda foram feitas duas *queries*, uma no *Prometheus* e outra no *InfluxDB*, equivalentes. Os dados obtidos foram os mesmos, o que prova que estes, para além de estarem a ser escritos, estão a sê-lo da forma correta.

Com os testes descritos, foi possível garantir que a aplicação efetivamente monitoriza a

aplicação de microserviços, guarda as métricas retiradas, tem acesso a estas para a sua análise e consegue, com as devidas medidas de segurança, realizar alterações no *cluster*, aumentando ou diminuindo assim o número de instâncias de microserviços.

Capítulo 7

Conclusão

Nesta secção é feita uma revisão de todo o trabalho realizado, o ponto a que se chegou e ainda o trabalho futuro associado a este projeto.

7.1 Conclusão e Trabalho Futuro

Este relatório é o culminar do trabalho realizado ao longo dos dois semestres de tese. Neste encontra-se descrito tudo o que foi feito, de maneira pormenorizada, começando pelo primeiro semestre, onde se fez o estudo de conceitos, seguido do estudo de algumas tecnologias relacionadas, o levantamento de requisitos e a definição da arquitetura. Relativamente ao segundo semestre, procedeu-se à implementação da arquitetura definida no primeiro semestre, sendo que maior parte dos objetivos foram atingidos. Assim, nesta secção vamos analisar resumidamente o trabalho desenvolvido, bem como trabalho futuro que faria deste projeto uma solução mais refinada e completa.

No segundo semestre desta tese foi implementada uma solução para o problema apresentado no início deste documento: uma plataforma que permite ao utilizador fazer *deploy* da sua aplicação de microserviços, possibilitando assim a sua monitorização, e por consequência a gestão automática desta. Com a ajuda de uma *service mesh* o tráfego é gerido e monitorizado, permitindo a recolha de métricas. A implementação de um algoritmo de decisão possibilita pegar nestas métricas, fazer a sua análise e tomar decisões automáticas sobre o escalonamento da aplicação, levando assim a colmatar os problemas encontrados na execução desta.
Esta solução mostra que o utilizador pode monitorizar e gerir a sua aplicação sem a necessidade de lidar com a infraestrutura, pois não precisa de lidar com a gestão de recursos, tráfego de pedidos e escalonamento. Para além disto, é dado ao utilizador a possibilidade de visualizar o que acontece na sua aplicação, com recurso a uma ferramenta que expõe os dados tratados em tabelas e gráficos.

Para uma solução mais completa e apurada existem algumas funcionalidades que podiam ser acrescentadas ou melhoradas num trabalho futuro, que são as seguintes:

- Criação de uma linha de comandos para o utilizador
- Implementação de *tracing* como uma fonte extra de dados para análise
- Implementação de um algoritmo de decisão mais complexo

- Criação de *dashboards* com os dados armazenados no *InfluxDB*

Uma das funcionalidades que seria interessante para permitir um maior controlo por parte do utilizador seria a criação de uma linha de comando personalizada para que o utilizador pudesse realizar certas ações que de momento necessitam de ser feitas alterando ficheiros de configuração.
Seria também importante a implementação de um serviço de *tracing*, pois este tipo de monitorização tem acoplada a si um maior espectro de informações do que as métricas, permitindo uma análise mais profunda e com maior contexto.
Um algoritmo de decisão mais complexo e completo está a ser desenvolvido pelo aluno de doutoramento Jaime Correia, e este irá integrar com este projeto, levando a capacidade análise e escalonamento a outro nível.
Seria ainda interessante utilizar os dados armazenados no *InfluxDB* para a criação de *dashboards* no *Grafana*, pois no estado atual o utilizador só tem acesso às últimas 6 horas de monitorização, ou seja, aos dados armazenados temporariamente pelo *Prometheus*.

No final, uma plataforma de gestão automática foi criada como resultado desta tese. No seu estado atual, esta plataforma alberga uma aplicação de microserviços, e é capaz de escalar esta aplicação conforme as necessidades desta, e tendo em conta as configurações do utilizador. Com o algoritmo que está a ser desenvolvido pelo aluno Jaime Correia, a plataforma irá escalar da melhor maneira para colmatar os problemas encontrados durante a análise das métricas, e futuramente de *traces*. Pelas razões acima descritas considero que a implementação de uma solução para o problema apresentado foi conseguida, tendo sido atingidos a quase totalidade dos objetivos.

Bibliografia

[1] Pooyan Jamshidi et al. "Microservices: The journey so far and challenges ahead". Em: *IEEE Software* 35.3 (2018), pp. 24–35. ISSN: 07407459. DOI: 10.1109/MS.2018.2141039.

[2] Nicola Dragoni et al. "Microservices: Yesterday, today, and tomorrow". Em: *Present and Ulterior Software Engineering* (2017), pp. 195–216. DOI: 10.1007/978-3-319-67425-4_12. arXiv: arXiv:1606.04036v4.

[3] *Microservices vs. Monolith Architecture - DEV Community*. URL: https://dev.to/alex{_}barashkov/microservices-vs-monolith-architecture-4l1m (acedido em 03/06/2019).

[4] Sam Newman. *Building Microservices DESIGNING FINE-GRAINED SYSTEMS*. 2015, p. 102. ISBN: 978-1-491-95035-7. DOI: 10.1109/MS.2016.64. arXiv: 1606.04036.

[5] *Introduction to Monolithic Architecture and MicroServices Architecture*. URL: https://medium.com/koderlabs/introduction-to-monolithic-architecture-and-microservices-architecture-b211a5955c63 (acedido em 01/06/2019).

[6] Claus Pahl e Pooyan Jamshidi. "Microservices: A Systematic Mapping Study". Em: 1.Closer (2016), pp. 137–146. DOI: 10.5220/0005785501370146.

[7] *O que é um contentor?* URL: https://aws.amazon.com/pt/containers/ (acedido em 03/06/2019).

[8] Claus Pahl e Brian Lee. "Containers and clusters for edge cloud architectures-A technology review". Em: *Proceedings - 2015 International Conference on Future Internet of Things and Cloud, FiCloud 2015 and 2015 International Conference on Open and Big Data, OBD 2015* (2015), pp. 379–386. DOI: 10.1109/FiCloud.2015.35.

[9] Rajdeep Dua, A. Reddy Raja e Dharmesh Kakadia. "Virtualization vs containerization to support PaaS". Em: *Proceedings - 2014 IEEE International Conference on Cloud Engineering, IC2E 2014* (2014), pp. 610–614. DOI: 10.1109/IC2E.2014.41.

[10] Giovanni Toffetti, Martin Bl e Florian Dudouet. "An architecture for self-managing microservices". Em: ().

[11] *What is Distributed Tracing?* URL: https://opentracing.io/docs/overview/what-is-tracing/ (acedido em 10/06/2019).

[12] *Introduction to distributed tracing | New Relic Documentation*. URL: https://docs.newrelic.com/docs/apm/distributed-tracing/getting-started/introduction-distributed-tracing (acedido em 10/06/2019).

[13] *OpenCensus - Tracing*. URL: https://opencensus.io/tracing/ (acedido em 21/06/2019).

[14] *OpenCensus - Introduction*. URL: https://opencensus.io/introduction/ (acedido em 21/06/2019).

[15] *Why You Should be Paying Attention to OpenTracing | Petabridge.* URL: https://petabridge.com/blog/why-use-opentracing/ (acedido em 22/06/2019).

[16] *Introduction to distributed tracing | New Relic Documentation.* URL: https://docs.newrelic.com/docs/apm/distributed-tracing/getting-started/introduction-distributed-tracing (acedido em 10/06/2019).

[17] *What is an API Gateway? | NGINX Learning.* URL: https://www.nginx.com/learn/api-gateway/ (acedido em 30/06/2019).

[18] *Designing microservices: API gateways.* URL: https://docs.microsoft.com/pt-pt/azure/architecture/microservices/design/gateway (acedido em 30/06/2019).

[19] *What Is a Service Mesh? - NGINX.* URL: https://www.nginx.com/blog/what-is-a-service-mesh/ (acedido em 02/06/2019).

[20] *Descomplicando Service Mesh 1/3 – Getup Cloud.* URL: https://blog.getupcloud.com/descomplicando-service-mesh-1-3-9abf298a4a?gi=300e934e3564 (acedido em 23/06/2019).

[21] *Time Series Database (TSDB) Explained | InfluxDB | InfluxData.* URL: https://www.influxdata.com/time-series-database/ (acedido em 15/01/2020).

[22] *InfluxDB 1.X: Open Source Time Series Platform | InfluxData.* URL: https://www.influxdata.com/time-series-platform/ (acedido em 15/01/2020).

[23] Nane Kratzke. "OPEN JOURNAL OF MOBILE COMPUTING AND CLOUD COMPUTING A Lightweight Virtualization Cluster Reference Architecture Derived from Open Source PaaS Platforms". Em: (), pp. 1–14.

[24] Emiliano Casalicchio. "Autonomic Orchestration of Containers: Problem Definition and Research Challenges". Em: (2017). DOI: 10.4108/eai.25-10-2016.2266649.

[25] *What is Kubernetes - Kubernetes.* URL: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/ (acedido em 19/06/2019).

[26] *Kubernetes — an overview – Just Another Dev Blog.* URL: https://justanotherdevblog.com/kubernetes-an-overview-bf47b0af1865?gi=e8208ce8f897 (acedido em 19/06/2019).

[27] *Docker overview | Docker Documentation.* URL: https://docs.docker.com/engine/docker-overview/ (acedido em 19/06/2019).

[28] *Docker Swarm 101.* URL: https://www.aquasec.com/wiki/display/containers/Docker+Swarm+101 (acedido em 19/06/2019).

[29] *Grafana Features | Grafana Labs.* URL: https://grafana.com/grafana (acedido em 19/06/2019).

[30] *GitHub - Netflix/zuul: Zuul is a gateway service that provides dynamic routing, monitoring, resiliency, security, and more.* URL: https://github.com/Netflix/zuul (acedido em 29/06/2019).

[31] *Elastic Compute Cloud - Amazon EC2 - AWS.* URL: https://aws.amazon.com/pt/ec2/?nc1=f{_}ls (acedido em 30/06/2019).

[32] *O que é o Amazon EC2? - Amazon Elastic Compute Cloud.* URL: https://docs.aws.amazon.com/pt{_}br/AWSEC2/latest/UserGuide/concepts.html (acedido em 30/06/2019).

[33] *Amazon EC2 Features - Amazon Web Services.* URL: https://aws.amazon.com/ec2/features/?nc1=f{_}ls (acedido em 30/06/2019).

[34] *Amazon ECS - Run containerized applications in production.* URL: https://aws.amazon.com/ecs/ (acedido em 30/06/2019).

[35] *What is Amazon Elastic Container Service? - Amazon Elastic Container Service.* URL: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/Welcome.html (acedido em 01/07/2019).

[36] *AWS EC2 Container Service — an overview – Just Another Dev Blog.* URL: https://justanotherdevblog.com/aws-ec2-container-service-an-overview-4b1460aeaf21 (acedido em 01/07/2019).

[37] *Amazon ECS Features - run containers in production.* URL: https://aws.amazon.com/ecs/features/?nc1=h{_}ls (acedido em 01/07/2019).

[38] *The C4 model for visualising software architecture.* URL: https://c4model.com/ (acedido em 28/06/2019).

[39] *The Web framework for perfectionists with deadlines | Django.* URL: https://www.djangoproject.com/ (acedido em 29/06/2019).

[40] *Welcome | Flask (A Python Microframework).* URL: http://flask.pocoo.org/ (acedido em 29/06/2019).

[41] *Weblet Importer.* URL: http://www.web2py.com/ (acedido em 29/06/2019).

[42] *Swarm mode overview | Docker Documentation.* URL: https://docs.docker.com/engine/swarm/ (acedido em 29/06/2019).

[43] *Persistent Disk: armazenamento local permanente | Persistent Disk | Google Cloud.* URL: https://cloud.google.com/persistent-disk/ (acedido em 29/06/2019).

[44] *OpenZipkin · A distributed tracing system.* URL: https://zipkin.io/ (acedido em 29/06/2019).

[45] *Jaeger: open source, end-to-end distributed tracing.* URL: https://www.jaegertracing.io/ (acedido em 29/06/2019).

[46] *Istio / Bookinfo Application.* URL: https://istio.io/docs/examples/bookinfo/ (acedido em 02/01/2020).

[47] *Overview | Prometheus.* URL: https://prometheus.io/docs/introduction/overview/ (acedido em 09/01/2020).

[48] *Storage | Prometheus.* URL: https://prometheus.io/docs/prometheus/latest/storage/ (acedido em 08/01/2020).

[49] *VictoriaMetrics | the best long-term storage for Prometheus.* URL: https://victoriametrics.com/ (acedido em 08/01/2020).

[50] *Prometheus endpoints support in InfluxDB | InfluxData Documentation.* URL: https://docs.influxdata.com/influxdb/v1.7/supported{_}protocols/prometheus/ (acedido em 08/01/2020).