FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Observabilidade e telemetria em arquiteturas de micro-serviços

António Pedro dos Santos Carvalho

DISSERTAÇÃO DE MESTRADO

DISSERTACÃO DE MESTRADO

Orientador: António Miguel Pimenta Monteiro

31 de outubro de 2022

Abstract

The adoption of *cloud* computing in recent years has grown exponentially. To get the best use of this architecture, software developers have been using kubernetes in order to form and manage containers. As a result, the use of kubernetes has been increasing in recent years.

With this, there is a need for constant motorization of these microservices, so that the system operates with the greatest efficiency and clarity, so that no anomalies occur in the systems and they do not evolve into disruptive situations for the service.

To achieve this goal, a market study was done to find the best tools for solving the problems that arise on a daily basis.

A monitoring system that fits the company's product *Tlantic* was architected and implemented. This system uses Grafana, Grafana Loki, Prometheus and Jaeger. The system was also alarmed, using the AlertManager from Prometheus so that the Tlantic team receives notifications in Slack and thus reduces the impact time that errors have on the system.

This system was architected to be implemented automatically, i.e., it is possible to install the monitoring system and the Tlantic application instantly, using Terraform.

Finally, this dissertation leaves some doors open for the future in order to update and improve the system in several aspects.

Resumo

A adoção de computação *cloud* nos últimos anos tem crescido exponencialmente. Para conseguir a melhor utilização desta arquitetura, os programadores de software tem vindo a utilizar kubernetes, de modo a formar e gerir containers. Por consequência, a utilização de kubernetes tem vindo a aumentado nos últimos anos.

Com isto, é necessário que haja uma motorização constante destes micro-serviços, de modo a que o sistema opere com a maior eficiência e clareza, para que não ocorram anomalias nos sistemas e estas não evoluam para situações perturbadoras do serviço.

Para atingir este objetivo, foi feito um estudo de mercado para conseguir encontrar as melhores ferramentas para colmatar os problemas que vão surgindo no dia-a-dia.

Foi arquitetado e implementado um sistema de monitorização que se ajuste ao produto da empresa *Tlantic*. Este sistema utiliza o Grafana, Grafana Loki, Prometheus e Jaeger. Foi também feita a alarmística do sistema, utilizando para o efeito o AlertManager do Prometheus de modo a que a equipa da Tlantic receba notificações no Slack e assim se reduza o tempo de impacto que os erros têm no sistema.

Este sistema foi arquitetado de modo a ser implementado de forma automático, ou seja, é possível instalar o sistema de monitorização e a aplicação da Tlantic de forma instantânea, utilizando o Terraform.

Por fim, esta dissertação deixa algumas portas abertas para o futuro com vista a uma atualização e melhoraria do sistema em vários aspetos.

Agradecimentos

Queria agradecer ao orientador André Pinto e à restante equipa da Tlantic, que sempre estiveram disponíveis para ajudar no que fosse preciso.

Pretendo também agradecer ao meu orientador, António Monteiro, pela ajuda na dissertação.

Consciente que sozinho nada disto seria possível, dirijo um agradecimento especial à minha família, mais em concreto aos meus pais, irmã e avós.

Por último, queria agradecer aos meus amigos, que sem eles não estaria onde estou hoje.

António Carvalho

You can't just turn on creativity like a faucet. You have to be in the right mood.
What mood is that? Last-minute panic.

Bill Watterson

Conteúdo

1 Introdução **1**
- 1.1 Contexto . . . 1
- 1.2 Motivação . . . 1
- 1.3 Objetivos . . . 2
- 1.4 Estrutura da dissertação . . . 2

2 Infraestrutura **5**
- 2.1 Micro-serviços . . . 5
- 2.2 *Containers* . . . 7
- 2.3 *Container management* . . . 8
  - 2.3.1 *Kubernetes* . . . 8
- 2.4 *Cloud Hosting* . . . 11
- 2.5 *Helm* . . . 11
- 2.6 Infrastructure Build Tools . . . 12
  - 2.6.1 *Terraform* . . . 12
  - 2.6.2 Ansible . . . 13
  - 2.6.3 Chef . . . 14
  - 2.6.4 Comparação . . . 14

3 Sistema de monitorização **17**
- 3.1 Monitorizar . . . 17
- 3.2 Observabilidade . . . 18
- 3.3 Arquitetura de sistemas de monitorização moderno . . . 19
- 3.4 Visualização . . . 20
  - 3.4.1 *Kibana* . . . 20
  - 3.4.2 Grafana . . . 21
- 3.5 Métricas . . . 21
  - 3.5.1 Prometheus . . . 21
  - 3.5.2 New Relic . . . 22
  - 3.5.3 cAdvisor . . . 22
- 3.6 Logs . . . 23
  - 3.6.1 Grafana Loki . . . 23
  - 3.6.2 Logstash . . . 23
  - 3.6.3 Fluentd . . . 24
- 3.7 Tracing . . . 24
  - 3.7.1 Jaeger . . . 24
  - 3.7.2 Zipkin . . . 25
  - 3.7.3 AWS X-Ray . . . 25

3.8 Escolha de software para o projeto 26

4 Arquitetura do sistema **27**
4.1 Desenvolvimento do Sistema 27
4.2 Ambiente de desenvolvimento 28
4.2.1 Mobile Retail Suit 28
4.2.2 RabbitMQ 29
4.2.3 Nginx 30
4.3 Ambiente de monitorização 30
4.3.1 Prometheus 30
4.3.2 Grafana 32
4.3.3 Grafana Loki 33
4.3.4 Jaeger 35
4.3.5 Terraform 36

5 Implementação e Resultados **37**
5.1 Terraform 37
5.2 Prometheus e Grafana 40
5.2.1 Alertmanager 44
5.3 Grafana Loki 45
5.4 Jaeger 46
5.5 Análise de Resultados 47

6 Conclusão **49**

A Código utilizado **51**

Referências **59**

Lista de Figuras

2.1 Diferenças entre aplicação monolítica e aplicação em micro-serviços. 6
2.2 Diferenças entre máquinas virtuais e *Containers*. 8
2.3 Exemplo do kubectl get pods. 9
2.4 Arquitetura de Kubernetes. 10
2.5 Exemplo de código de configuração de Terrafom. 13

3.1 Arquitetura de um sistema de monitorização moderno. 20
3.2 Arquitetura de um sistema de *ELK Stack*. 21
3.3 Arquitetura de um sistema de monitorização usando o Prometheus. 22
3.4 Arquitetura exemplo do Jaeger. 24
3.5 Arquitetura do Zipkin. 25

4.1 Arquitetura de um sistema de monitorização usando Grafana, Loki e Alertmanager. 28
4.2 Aplicação modelo onde vai ser implementado o sistema de monitorização. 29
4.3 Exemplo de a aplicação de Nginx como Load balance. 30
4.4 Arquitetura do Prometheus, Grafana e o Ambiente de desenvolvimento. 31
4.5 Arquitetura de um sistema de recolha e visualização de logs usando o Grafana Loki e o Grafana. 35
4.6 Arquitetura de visualizaçao de Traces utilizando o Jaeger. 35
4.7 Arquitetura de um sistema de monitorização usando Grafana, Loki e Alertmanager. 36

5.1 Resultado da aplicação dos ficheiros Terraform. 40
5.2 Dashboard criada para a visualização de métricas do nó do cluster no Grafana. . . 42
5.3 Dashboard criada para a visualização de métricas do cluster no Grafana. 43
5.4 Dashboard criada para a visualização de métricas do pod RabbitMQ no Grafana. 43
5.5 Dashboard criada para a visualização de métricas da aplicaçao RabbitMQ no Grafana. 44
5.6 Dashboard criada para a visualização de métricas da aplicaçao Nginx no Grafana. 44
5.7 Notificação do Slack pela Regra "number_of_queues". 45
5.8 Exemplo de uma query de logs a um componente da aplicação da Tlantic. 46
5.9 Interface do Jaeger com querys com vista geral sobre o sistema. 47
5.10 Interface do Jaeger com querys com vista sobre uma query. 47

Listings

5.1 Código do ficheiro KUBECONFIG. 37
5.2 Ficheiro de Terraform providers 38
5.3 Configuração de variável para os providers. 38
5.4 Ficheiro de criação de namespaces, namesapces.tf. 39
5.5 Ficheiro exemplo dos recursos do MRS neste caso o mrs-integration-pricing. . . 39
5.6 Ficheiro de Terraform do RabbitMQ 39
5.7 Código de configuração de datasources no Grafana 41
A.1 Código para implementação das regras de Prometheus. 51
A.2 Configuração do AlertManager. 53
A.3 Parte do Código para a formação de Tracers. 54

Capítulo 1

Introdução

1.1 Contexto

Com a aparição da doença Covid-19, o mundo em 2019 foi obrigado a fazer confinamento,esta situação obrigou as empresas adotarem um regime de trabalho remoto. Segundo o *Cloud Microservices Market Research Report* é esperado que haja um grande aumento nos *cloud services* devido à falta de mão de obra e à necessidade de trabalhar a partir de casa. Desta forma, é esperado um aumento de 21.7% por ano até 2026 fazendo o mercado crescer de 831.45 milhões *USD* para 2701.36 milhões de *USD*.[1]

A acompanhar este aumento de mercado as grandes empresas como a *Microsoft*, *Google* e *Amazon* decidiram criar serviços de Infraestruturas e de plataforma, o IaaS e PaaS, designadamente a *Microsoft Azure*, *Google Cloud platform*, *Amazon Web Services*, havendo outros concorrentes no mercado. Estes serviços tem visto o numero de utilizadores de baixa escala a aumentarem bem como de grandes utilizadores como *Netflix*, *Facebook*, *BBC*, *Adobe*, *Twitter*,*Spotify*, *HSBC*, entre outros. Esta forte adoção tecnologias *cloud* tem sido uma tendência mundial visto que as grandes empresas conseguem ter um produto mais estável e assim aumentarem o número de clientes, e por consequência, também os seus lucros.

Assim, para acompanhar esta direção mundial é necessário uma forte aposta na monitorização e análise de produtos. De forma a certificar que os sistemas estão a funcionar corretamente, faz-se uma prevenção de modo a conseguir encontrar comportamentos irregulares e a resolvê-los antes que se tornem problemas que reflitam na experiência do utilizador e, por sua vez, na queda de lucros da empresa em questão.

1.2 Motivação

A empresa *Tlantic* desenvolve serviços de retalho para as empresas pertencentes ao grupo *Sonae* e outras no mercado internacional. Precisa de ter o sistema confiável pois tem de computar a transação de mais de 50 milhões de artigos por mês. Para isto ser possível é necessário que a aplicação não tenha falhas graves para que não prejudique os clientes e o seu próprio negócio.

Para isto, o sistema atual possui um ambiente de monitorização bastante simples e bastante pesado. É um sistema bastante simples que tem de ser alterado de forma a não criar tanta carga de trabalho, necessita assim de ser expandido de forma a cobrir todos serviços e a ser mais eficiente. Precisa também de alarmar a equipa de suporte se algo estiver errado de forma a ser possível corrigir os erros para que os produtos não atinjam os consumidores finais.

1.3 Objetivos

Esta dissertação foi realizada com o objetivo de introduzir um sistema de monitorização no sistema da empresa tlantic. Para isso foi necessário a realização de um estudo de mercado. Este estudo tem como missão encontrar as soluções mais recentes e com maior rentabilidade para a monitorização de aplicação baseada micro-serviços. Depois de encontrar as melhores ferramentas disponíveis, posteriormente foram implementas num cluster fornecido pela Tlantic de forma poder a implementar e experimentar todos os recursos e assim, comprovar que o novo sistema de monitorização funciona como expetável.

Este modelo necessita de conseguir obter os *logs* criados pelos *containers* da aplicação da *tlanic*, bem como as suas métricas. Para conseguir fazer a observação dos logs e das métricas, é imprescindível a criação de gráficos de visualização destas informações de modo facilitado e que seja percetível padrões de erros para facilitar o comportamento do sistema. Para além destes dados, também é necessário a criação de tracers que tem como objetivo de tentar identificar se está a ocorrer algum problema na rede do sistema, sabendo assim a localização da perturbação no sistema e o seu motivo. O atual modelo tem de conseguir também notificar a equipa de suporte. Na verdade sempre que existe um erro que comprometa o estado do sistema é necessário enviar uma notificação para a aplicação de comunicação entre os elementos da equipa neste caso, o slack.

Para além desta componente monitorização, esta dissertação tem como objetivo automatizar criação e o versionamento da infraestrutura do sistema completo desde a monitorização e à própria aplicação da empresa.

Depois do trabalho concluído, a empresa *Tlantic* poderá aplicar o sistema de monitorização a toda a sua estrutura de forma automática e assim, obter um sistema que gera informação imprescindível que consiga ter uma visão geral do sistema de forma alertar a equipa responsável em serviço no caso de haver alguma anomalia.

1.4 Estrutura da dissertação

Este documento está divido em vários capítulos. Este capítulos são:

- O capitulo 1 Introdução, faz a introdução ao tema, bem como seu o motivo e objetivos.
- O capitulo 2 Infraestrutura, aborda os conceitos teóricos necessários á compreensão do sistema da dissertação como micro-serviços,*containers*,*kubernetes*, entre outros. Também é feita a apresentação das ferramentas estudadas para automatizar a criação do ambiente.

- O capitulo 3 Sistema de monitorização, faz apresentação dos conceitos teóricos necessários ao entendimento de monitorização e de observabilidade das tecnologias de monitorização atuais que foram estudadas.
- No capitulo 4 Arquitetura do Sistema é feita uma descrição da arquitetura do sistema a desenvolver.
- O capitulo 5 Implementação e Resultados, é feita uma explicação sobre a implementação da arquitetura e a demonstração dos resultados obtidos.
- Por último, no capitulo 6,Conclusões onde estão presentes as conclusões da dissertação e o possível trabalho para futuro.

Capítulo 2

Infraestrutura

2.1 Micro-serviços

De modo, a conseguir programar uma aplicação, os programadores de *software* optavam por uma abordagem de estrutura de única unidade de código base. Desenvolver uma aplicação nesta arquitetura chama-se criar uma aplicação monolítica. Esta arquitetura fundamenta-se em três partes, a primeira é interface de utilizador (*UI*) que utiliza principalmente páginas *HTML*, *java script* e *php* para um *UI* na *web*. Outra secção é á base de dados que manipula *SQL* para criar e modificar tabelas de dados, a última secção é a parte do servidor que lida com os pedidos *HPPT*, e consegue fazer consultas a base de dados de modo otimizado. Esta abordagem tem várias desvantagens como o crescimento do sistema, como é realizado numa escrita linear existem grandes dificuldades para aumentar uma aplicação complexa. Outra desvantagem é o tempo em que o serviço fica *offline* caso ocorra uma falha do sistema ou uma atualização do código.

Assim, sempre que existe uma manutenção, ou a atualização do código a aplicação tem de estar *offline* e por consequência afeta os utilizadores baixando a qualidade de experiência do serviço, fazendo as empresas perderem receita. Devido a este problema, os programadores evitam o lançamento de novas funcionalidades, adiando a sua atualização criando mais problemas para o futuro.[2]

Para combater este problema, os programadores mudaram a abordagem à estrutura da aplicação para micro-serviços. Micro-serviços não é um conceito exato, é uma forma de projetar uma aplicação de modo a dividir o código completo em divisões mais pequenas, serviços. Estes serviços comunicam entre si de forma independente, de forma a conseguir obter um produto final que não necessita que todas as funcionalidades estejam operacionais. Não existe uma definição exata de micro-serviços mas quase todos têm estas características:

1. Divisão do sistema em serviços mais pequenos onde cada serviço é substituível, atualizável e independentemente.

2. Criação de equipas multifuncionais que não se dedicam apenas ao trabalho numa área, por exemplo, uma equipa trabalha apenas na *UI* enquanto outra equipa trata da base de dados.

3. A equipa faz um produto e não um projeto, porque a equipa está, no dia a dia, em contacto com a aplicação corrigindo erros e melhorando o sistema até o produto ser descontinuado em vez de apenas entregar a aplicação completa ao cliente.
4. A utilização de *APIs* e do protocolos como *HTTP* de modo a fazer a conexão entre os serviços.
5. A possibilidade de utilização de vários standards na aplicação, como por exemplo a utilização de várias línguas de programação.
6. Utilização de várias bases de dados.
7. A automatização da infraestrutura e da testarem da mesma, são utilizados mecanismos de virtualização de modo a ser automático a distribuição dos serviços pelas máquinas.
8. Cada serviço é projetado em caso de falha de outros serviços .
9. O sistema é projetado de forma a ter uma evolução continua ao longo do tempo de vida do produto.

[3]

Este modo de programação tem ficado popular devido à natureza simples e à facilidade de aumentar o sistema, existem limites claros o que permite que haja responsabilidades bem definidas para o serviço e para a equipa de produção. Existe uma implementação independente, cada parte pode ser implementada sem ser necessário ter que se preocupar com os restantes elementos do código, melhorando a velocidade de atualização do mesmo.[4]. A figura 2.1 mostra a diferença aplicação monolítica e aplicação em micro-serviços.

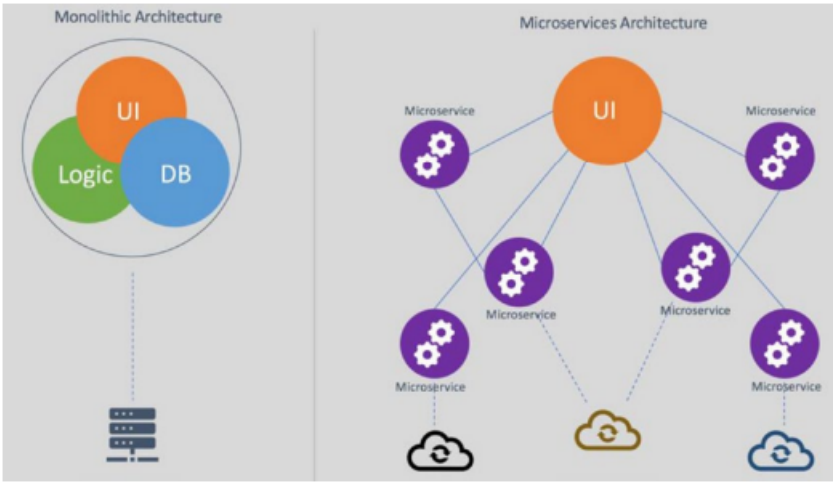

Figura 2.1: Diferenças entre aplicação monolítica e aplicação em micro-serviços.
[2]

Todavia, o facto de ser uma tecnologia avançada, requer a utilização de várias técnicas avançadas como manutenção de API, sistemas distribuídos e outras. Faz requer ainda programadores

experientes. Que seja necessária uma aprendizagem rigorosa de modo a desenvolver aptidões na área de engenharia de *software*. Outra desvantagem é que o aumento do números de micro-serviços faz com que aumente a complexidade do sistema.

Independentemente das vantagens e desvantagens é necessário, ao criar uma aplicação de micro-serviços, ter atenção aos problemas que podem ocorrer. Um dos problemas mais comuns é a escolha errada de estratégia para divisão entre cada serviço porque não é uma escolha clara. Outros problemas, são testar o sistema, devido a ser uma tarefa complexa à medida que vai aumentado a aplicação, e a monitorização tem de ser realizada de forma distribuída ao contrário das aplicações monolíticas.[4]

Assim, com esta estrutura, os micro-serviços mudaram a forma de programar os serviços de computação de *cloud*. Como estes serviços são muito dependentes da qualidade de serviço, como o tempo de latência, a taxa de transferência de rede e a disponibilidade e confiabilidade que o sistema tem de ter, a estrutura de micro serviços é mais apelativa devido às vantagens anteriormente faladas, a possibilidade de utilizar várias linguagens de programação e vários *frameworks*. Isto só é possível com a realização de *application programming interface* (*API*) entre plataformas como *remote procedure calls (RPC)* ou *RESTful API*. Outo motivo é a melhor correção e performance que este sistema consegue criar de forma a conseguir gerir a comunicação entre os vários serviços diminuindo a quantidade de computação requerida pelos mesmos. [5]

Deste modo, as aplicações que usam *cloud* têm nos últimos tempos aumentado a utilização da arquitetura micro-serviços comparativamente com a arquitetura monolítica.

2.2 *Containers*

As tecnologias de virtualização tem verificado um aumento de importância em *Cloud computing* ganhado um papel chave nos últimos anos. Existem vários motivos de adoção destas tecnologias, um deles é a ampliação de dados trocados entre serviços[6], o *hardware* mais recente tem uma capacidade processamento muito superior que é utilizada pelas aplicações sendo necessário que cada processador faça várias funções de modo a poupar recursos e tempo.[2] Estas tecnologias são a chave para conseguir aumentar a performance, e, por isso, têm visto um aumento de utilização em certas áreas como *Cloud Environments, Internet of Things*, e *Network FunctionVirtualization*.[6]

Existem várias formas de conseguir a virtualização. Nos últimos anos ficaram populares duas formas de conseguir correr aplicações em ambiente virtuais, *Hypervisor-based* que permite criação de várias máquinas virtuais sobre o mesmo *hardware* e, mais recentemente, os *containers*.[7]

Um *container* tem uma arquitetura diferente das máquinas virtuais, mas ambas criam uma camada de abstração entre o *hardware* e o *sofware*, contudo, o tipo de abstração é diferente.[6] A máquinas virtuais necessitam do um sistema operativo (OS) e *drivers* completos para que seja possível correr a aplicação por cima de um *hardware*. Por outro lado, os *containers* partilham o OS, principais bibliotecas, *drivers* e binários sendo a virtualização feita ao nível do *kernel*. Por isso os processos das aplicações necessitam de ser compatíveis com este.[8]

Os *Containers* têm ganho preferência às máquinas virtuais no paradigma das aplicações *cloud* devido a serem mais leves e flexíveis comparativamente. Criando uma performance melhor e utilização de menos memória, o que leva a terem menor custo. Além disso, têm um desempenho melhor o que leva a ser possível o aumento do sistema com uma melhor utilização dos recursos de um *cluster*.[2] A figura 2.2 mostra a diferença entre máquinas virtuais e *Containers*.

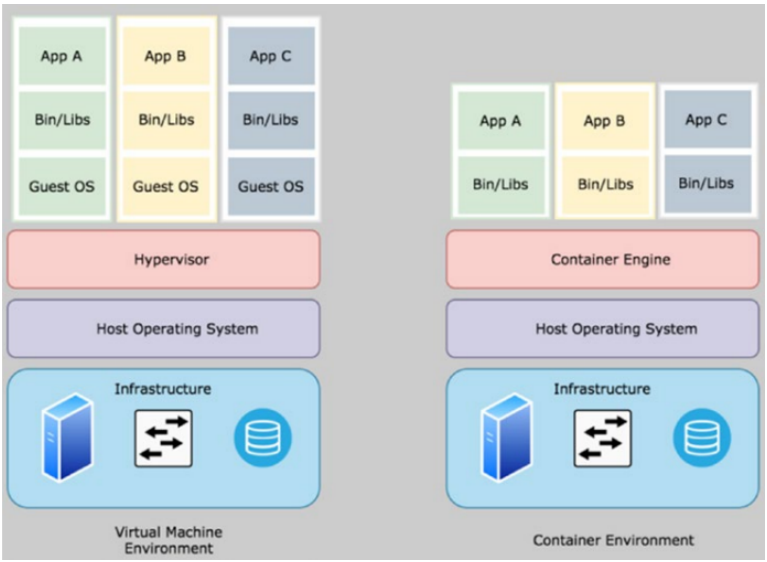

Figura 2.2: Diferenças entre máquinas virtuais e *Containers*. [2]

2.3 *Container management*

De forma a conseguir gerir os containers é necessário um motor de geração de container que tenha a capacidade de levantar e correr containers em ambiente de virtualização. Existem várias formas de conseguir este processo, existem várias opções como docker, LXC (Linux), kubernetes, entre outros. Sendo a mais popular ao longo dos anos o docker.[9]

2.3.1 *Kubernetes*

Para a conseguir controlar os *containers* de forma eficiente, a *Google* em 2014 criou o *kubernetes*.[9] *Kubernetes* é uma plataforma de acesso aberto de gestão de *containers*, que ao longo do tempo tem ganho popularidade entre os seus concorrentes como *Docker Swarm*, *Red Hat OpenShift Container Platform (Red Hat OCP)*, *Amazon Elastic Container Service (Amazon ECS*. [10]

Kubernetes pode usar o *Docker*, como motor para correr os *containers*, onde uma das características mais importantes é o aumento automático de containers e manutenção do sistema, o que permite o sistema correr em perfeitas condições sem que seja necessário a intervenção humana.[11] Este tipo de serviço consegue ser resiliente e escalável de forma a ser perfeito para utilização com micro-serviços e de containers. [12]

As principais características de *kubernetes* são:

1. A criação automática dos containers e alocação de recursos de modo a que a aplicação corra com os requisitos impostos e que use apenas os necessários, fazendo o balanceamento entre a melhor opção e o estado crítico.

2. O *kubernetes* tem um sistema de redes interno que faz com que o utilizador não precise de se preocupar com a comunicação entre os containers. Kubernetes define automaticamente o endereço de IP dos containers fazendo um balanceamento da rede.

3. Existe a opção de estabelecer o sistema de armazenamento localmente ou num serviço de *cloud* publica, como o *AWS*, *GPC* entre outros.

4. O kubernetes tem a capacidade de reiniciar containers que falharam, que não respondem ou os que falharam os mínimos requeridos pelo utilizador. Caso o nó desapareça o kubernetes tem a capacidade de alocar recursos para que sejam levantados noutro nó se existir essa possibilidade.

5. A implementação de segredos e gestor de configurações permite que haja modificação de configurações de forma automática sem ser preciso voltar a levantar o sistema completo e sendo possível encriptar estas definições para não ficarem visíveis no sistema.

6. O kubernetes consegue fazer a atualização de certos módulos sem ser necessário fazer com que o sistema fique completamente *offline* e, se ocorrer alguns erros, o kubernetes consegue reverter de modo a corrigi-los conseguindo fazer automaticamente *Rollbacks* e *Rollouts*. [13]

Kubernetes é definido por um nó mestre e os seus nós trabalhadores. O mestre é responsável por gerir o cluster de Kubernetes, onde estão os principais processos e serviços do ambiente. Os principais processos disponíveis pelo nó mestre são o API server que expõem os serviços a utilizadores externos. Estes utilizadores utilizam o comando kubectl para conseguir estabelecerem ligação ao servidor. As funções principais deste comando são o kubectl get que lista os recursos de kubernetes, como os pods, os servicos, entre outros; o kubectl apply que cria recursos no cluster; kubectl delete que apaga recursos; kubectl port-forward que faz uma ponte entre o utilizador e serviço para seja possível haver uma ligação entre o serviço e a porta escolhida. A figura 2.3 mostra um exemplo de utilização do comando kubectl get pods.

```
C:\Users\apsc0>kubectl get pods -n monitoring
NAME                                                     READY   STATUS    RESTARTS   AGE
alertmanager-kube-kube-prometheus-alertmanager-0         2/2     Running   10         21d
grafana-584bc8b86c-7thtc                                 1/1     Running   0          20d
kube-kube-prometheus-blackbox-exporter-5bc6cf6bbd-4cdtt  1/1     Running   0          21d
kube-kube-prometheus-operator-85cd5d95c4-d7zv2           1/1     Running   0          21d
kube-kube-state-metrics-5ddcff856b-n7cl7                 1/1     Running   0          21d
```

Figura 2.3: Exemplo do kubectl get pods.

O nó mestre consegue também gerir os processos e os serviços que estão a correr no *cluster* devido a ter o Controller Manager. Além disso, tem o *Scheduler* que assegura que todos deployments estão programados, e têm uma base de dados, o *ETCD*.

Os nós trabalhadores são constituídos por o Kubelet que é o motor de criação de containers. Este componente é essencial pois quando o kubenetes indica que é necessário outro container é com o kubelet que comunica, ou seja, o kubelet cria e corre containers. Dentro dos nós trabalhadores existem também pods.

Outro componente é kube-proxy que implementa o sistema de rede do kubernetes, ele permite a comunicação entre pods e nós e de dentro e fora do ambiente de kubernetes. A figura 2.4 mostra a arquitetura de Kubernetes.

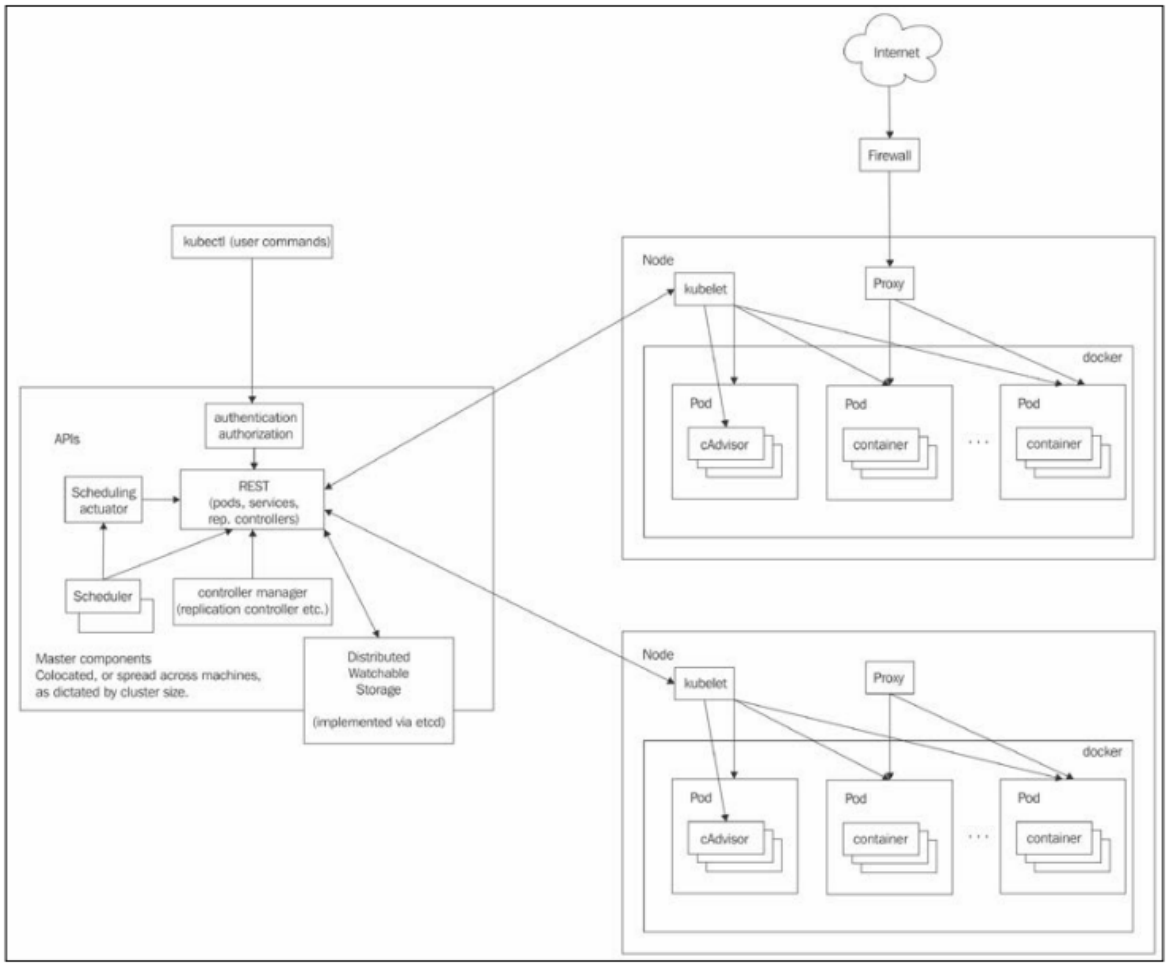

Figura 2.4: Arquitetura de Kubernetes.
[14]

Os principais componentes do kubernetes são Pods, a unidade básica de trabalho de *kubernetes*, um pod contem pelo menos um cointainer e ele pode acessar à memoria local do nó todos os containers num nó. Labels são pares de key-values que são usados para agrupar objetos como

pods. As labels são importantes pois existem componentes que necessitam de ser identificados de modo a facilitar a iteração entre componentes.

Serviços são usados para expor a funcionalidade de um pod a um utilizador ou a outro pod. Estes podem ser controlados por um IP virtual atribuído pelo *kubernetes*, estes serviços normalmente estão expostos por *endpoints*. Estes normalmente abrangem um grupo de pods com a mesma label. O namespace é um cluster virtual, estes clusters estão isolados e apenas podem comunicar entre si com interfaces. Um cluster físico pode conter vários namespaces e cada nó pode ter objetos de vários namespaces.

Segredos são pequenos objetos com informação confidencial como palavras-passe, tokens. Este tipo de objeto é guardado no etcd em texto onde certos elementos devem estar encriptados de modo a serem protegidos e são acedidos pelo Kuberntees API server de modo a que os pods tenham acesso às informações. Um daemonset é um tipo de instância que faz com que um pod corra em cada nó, se um nó novo aparecer o kubernetes de forma automática cria um pod nesse nó.

Os operators que são um tipo de controladores. Estes permitem aos utilizadores de kubernetes a extensão e a criação de funcionalidades não existentes no nó mestre e o API.[15] Estes componentes conseguem configurar e gerir aplicações bem como monitorizar o seu estado. O operator adiciona um endpoint a API do kubernetes chamada custom resource.[16] Os custom resource, CRs são uma extensão do mecanismo de API do kubernetes estes fornecem endpoits para a leitura e escrita de estrutura de dados. A diferença entre Crs e recursos oficiais de API do kubernetes é que os CRs não existem em todos os clusters sendo necessário a sua definição.

2.4 *Cloud Hosting*

A implementação desta tecnologia de *containers* é ideal em contexto de ambiente de cloud pois, devido ao custo é proibido ter um micro serviço alocado a cada máquina virtual. Foi por isso que o Kubernetes foi desenhado para aplicações nativas em cloud. Assim, pela escalabilidade e sua operação várias empresas utilizam infraestruturas como serviço, IaaS, e plataforma como serviço, PaaS, como o Amazon Web Services,AWS, o Microsoft Azure,Kubernetes Engine (GKE). A empresa Tlantic utiliza o AWS para correr o seu ambiente de trabalho onde tem a suas aplicações e onde foi realizado o trabalho desta dissertação.[17]

2.5 *Helm*

O kubernetes organiza e orquestra *conteiners* mas falta organização ao mais alto nível de modo a agrupar imagens. De modo a conseguir automatizar o processo de instalação e de configurações de serviços e *deployments* de *kubernetes* é usado uma ferramenta de gestão de pacotes, *Helm* que dá uma maior simplicidade ao processo de implementação e ao versionamento de aplicações bem como facilita a partilha de informação em repositórios públicos e privados.[18]

Uma das principais utilizações é o *Helm Chart*. Um Chart é um conjunto de ficheiros escritos, maioritariamente escrito em *YALM*, que pode descrever uma aplicação complexa. A utilização de *charts* tem como princípio a facilidade de instalação e desinstalação, a facilidade de alteração de configurações, bem como o agrupamento de *chart* num arquivo *tgz*.[19]

As principais linhas de comando desta ferramenta são:

1. Helm search que faz um pesquisa por predefinição no repositório oficial de *Kubernetes*.

2. helm repo tem várias ferramentas que podem ser add, index remove e update e estas instalam, removem, indexam e melhoram o repositório respetivamente.

3. helm install faz a instalação do *chart* no sistema.

4. helm upgrade faz a melhoria da versão atual do *chart* e helm rollback reverte essa melhoria.

5. entre outros...[18]

2.6 Infrastructure Build Tools

Kubernetes pode ter imensas vantagens mas traz também alguns desafios sendo um deles a possibilidade de uma instalação baseada em código não ser possível.[20] Para combater este problema o kubernetes e mais especificamente o AWS permite ferramentas de infraestrutura como código, IaaC, onde é possível ter um continuo planeamento, automatização e segurança. [21]

Estas ferramentas reduzem a complexidade relacionada com a criação de infraestruturas. Aumentam a facilidade na transição entre diferentes ambientes, bem como diferentes configurações.[22]

Para isto existem várias ferramentas no mercado: o Terraform, Jeakins, Ansible, entre outras.

2.6.1 *Terraform*

Terraform é uma ferramenta grátis de orquestração, foi criada pela empresa HashiCorp e está escrita na linguagem Golang. Esta ferramenta utiliza um tipo de código declarativo, ou seja, o código para implementar a infraestrutura é escrito de forma a declarar o seu estado final, e não os passos para obter esse mesmo estado.

Terraform utiliza um código binário na máquina do utilizador de forma a criar a infraestrutura necessária. Para isto, o Terraform faz chamadas de API aos fornecedores de infraestrutura, como por exemplo, o AWS. Para conseguir fazer estas chamadas o Terraform necessita dos Tokens de autenticação. Estes conteúdos são guardados num ficheiro de configurações do Terraform. A figura 2.5 mostra um exemplo de código de configuração de Terrafom.

```
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
}
```

Figura 2.5: Exemplo de código de configuração de Terrafom.
[23]

Os comandos principais do Terraform são o "terraform plan", o qual planeia as mudanças no sistema de forma a que o utilizador verifique se não existe erros no código. O "terraform appy"cria essas mudanças no sistema e "terraform delete"apaga o conteúdo formado pelo Terraform.
[23]

2.6.2 Ansible

Ansible é uma ferramenta de configuração que consegue automatizar as tarefas de infraestruturas, correr comandos ad hoc e instalar aplicações em múltiplas máquinas.[24] Utiliza uma linguagem de domínio específico (DSL) que descreve o estado final do sistema com pacotes certos que necessitam de ser instalados. Estes contêm os ficheiros de configuração e as permissões necessárias para que o Ansible consiga fazer modificações no sistema.

A principais características do *Ansible* são:

1. Os ficheiros de configuração são chamados playbooks são escritos em *YAML* e têm uma sintaxe fácil de ler.

2. Não é necessário instalar agentes ou outro software em cada máquina, apenas é necessário Python.

3. Esta ferramenta é baseada num sistema de push. A principal vantagem deste sistema é que o utilizador pode verificar quando existem mudanças no estado servidor. Esta caraterística faz com que seja superior a um sistema pull em escalabilidade para grande número de nós e para ambientes em que os nós sejam removidos e adicionados dinamicamente.

4. O processo de funcionamento desta ferramenta é por predefinição criar um playbook, correr a linha de comando ansible-playbook, e por último o ansible conectar-se aos servidores e executar os módulos que fazem modificação no estado do servidor.[25]

5. O ansible consegue gerir inúmeros nós, mas administrar um simples nó é bastante simples.

6. O ansible pode ser usado para executar comandos shell em servidores remotos mas a principal vantagem é possibilidade de instalar aplicações completas com ajuda de helm charts.[26][25]

2.6.3 Chef

Chef é uma ferramenta de configuração e de automatização de infraestrutura. Este software providencia o sistema de capacidades de visibilidade, automatização do sistema de forma continua desde o desenvolvimento até a produção. Esta ferramenta está divida entre vários componentes que são: o Chef server, Chef client, Chef workstation, Chef repo.

O Chef server é o servidor responsável por manter a configuração disponível guardando cookbooks, repices e regras que têm de ser aplicadas aos nós. Os cookbook são repositórios escritos em Ruby para ficheiros, modelos, bibliotecas e receitas entre outros. Repices são padrões simples de configurações para ficheiros e aplicações sendo elementos fundamentais nas configurações de um cookbook. Este servidor permite uma utilização de uma interface visual entre a consola de gestão e de pesquisa.

O Chef client pode ser instalado em diferentes nós de um sistema de Kubernetes de AWS e garante localmente que este tem a configuração pretendida. O Chef client comunica com o Chef server regularmente de forma a conseguir obter os cookbooks e configurações sem atraso relevante.

Chef Workstation facilita ao utilizador a criação, teste e manutenção de cookbooks e interaja com o Chef server e o nós do sistema. A Worksation tem incorporada o Chef development kit que é um um pacote de comandos para conseguir utilizar todas funcionalidades do Chef.

Chef Repo é um repositório onde são guardados todos ficheiros do Chef como Cookbooks entre outros. É gerido como outros repositórios como o GitHub.[27]

2.6.4 Comparação

Estas foram a principais ferramentas estudadas, todas elas são ferramentas usadas por grandes empresas e têm as suas vantagens e desvantagens. Para se conseguir observar as diferenças das ferramentas, a tabela 2.1 menciona o custo que tem para o utilizador, se existe a possibilidade de utilizar fornecedor cloud, se a infraestrutura é constante ou não, o tipo de arquitetura da ferramenta, o número de contribuidores em 2019 e quantidade de maturidade da ferramenta,

	Chef	Ansible	Terraform
Custos	Grátis	Grátis	Grátis
Cloud	Todas	Todas	Todas
Infraestrutura	inconstante	inconstante	constante
Linguagem	processual	declarativa	declarativa
Arquitetura	cliente/servidor	apenas cliente	apenas cliente
Contribuidores(nº)	4,386	562	1,261
Maturidade	Alta	Media	baixa

Tabela 2.1: Tabela que compara as ferramentas de infraestrutura como código.[28][23]

O Chef pelo facto de ser uma arquitetura cliente/servidor faz com que seja mais pesado para o AWS, criando um custo escondido em comparação com as outras ferramentas e por isso o Terraform vai ser usado no trabalho realizado.

O Ansible não foi escolhido por ter infraestrutura inconstante, ou seja, é possível que haja diferenças entre o código e sua implementação no ambiente. Este item faz com que seja um fator negativo, e por esta razão, nesta dissertação foi utilizado o Terraform para fazer a sua automatização da infraestrutura que será explicada nos próximos capítulos.

Capítulo 3

Sistema de monitorização

3.1 Monitorizar

Monitorizar é o processo de ganhar visibilidade do estado do sistema, ou seja, é o processo de verificação que consegue observar as métricas e *logs*, e sendo possível, avisar os administradores se ocorrer algum problema.[29] Este sistema tem a responsabilidade de conseguir verificar a saúde do ambiente, corrigir alguns problemas que se manifestaram no passado e que não sejam muito complexos como resolução de certos comportamentos irregulares constantes. Para isto ser possível o programa de monitorização tem de conseguir verificar o estado em várias áreas do sistema, como:

1. *User Experience*
2. *Application*
3. *Services*
4. *Containers*
5. *Cloud Infrastructure*

Cada uma destas áreas vai ter diferentes prioridades de dados que são importantes de verificar. Na área do *User Experience* e *Application* é interessante de observar o tempo de resposta, bem como a percentagem de ações falhadas, no caso do *User Experience* o tempo reposta do utilizador. No cenário de *Services*, o importante é a percentagem do tempo em que o serviço está disponível bem como se está ocupado e o número de pedidos que ficaram em fila de espera. No *Containers* o tempo em que *CPU* está a ser limitado bem como a taxa de utilização de memória e pacotes perdidos. Por ultimo, *Cloud Infrastructure* é importante na percentagem de recursos usados e a percentagem de tempo em que os recursos são utilizados[2].

No que respeita a forma como os serviços de monitorização conseguem recolher a informação existem duas formas, *pull* e *push*. Sistema desenhados com *pull* são sistemas onde o programa de monitorização envia pedidos ao serviço principal de maneira conseguir receber os dados, ou seja, este método dá um foco ao programa que quer receber os dados. Este princípio traz vários

problemas, é muito complicado para os programadores aumentarem o programa sem perderem imenso tempo a desenhar um novo programa de monitorização. Desta forma, o método de *pull* tem desvantagens, mas consegue ser recomendado para verificar a disponibilidade dos serviços bem como reduzir o tempo em que o sistema está indisponível.

O outro método, *push* tem outra filosofia de desenvolvimento, é a própria aplicação que envia ao sistema de monitorização os dados quando os tem disponíveis. Os dados são enviados por emissores que podem utilizar qualquer tipo de protocolo de transporte não sendo requisitados a usarem um específico. Assim, *push* é recomendado a recolher a informação de componentes e dos serviços para descobrir desempenho.[30]

3.2 Observabilidade

Observabilidade é um conceito que se foca em interpretar os dados recebidos de um sistema externo de forma que se consiga prever um erro à mínima falha nos dados, o objetivo de uma plataforma de observabilidade é encontrar o ponto de desempenho normal de um sistema e depois melhorá-lo. Claro que isto é num caso perfeito, mas a observabilidade permite eliminar possibilidades porque um erro está a ocorrer. Para conseguir atingir este objetivo é necessário a relação de três pilares, que são métricas, *logs* e *traces*.

Métricas são dados numéricos fáceis de obter de um certo programa que se está a observar. Estes dados normalmente são reunidos em gráficos de modo a ser uma fácil leitura. Existem dois tipos de métricas, os *work metrics* que fornecem a informação do sistema, por exemplo, *Throughput*, e os *resource metrics* que fornecem informação sobre o recursos do sistema, por exemplo, a percentagem de utilização de *CPU*. Os eventos são outro parâmetro muito importante da observabilidade mas normalmente incluídos nas métricas, eles são um acontecimento que ocorreu num determinado momento ao contrário das métricas que são recolhidas por intervalos de tempo Um exemplo de um evento é um acionamento de um alarme.[2]

Os *logs* são coleções de dados semiestruturados ou apenas strings que detalham erros ou anomalias presentes no sistema ao contrário das métricas. Estes são usados para conseguir ter uma perceção do que aconteceu no caso de erro, de forma a chegar uma solução mais rápida.

Traces são dados semiestruturados como os logs, mas têm informação de um caminho de rede, o quais comunicam com o programa de modo a obter essa informação. É criado um documento parecido com um *log* de forma a mostrar detalhadamente um erro. Os principais parâmetros dos tracers são em que função do sistema ocorreu o erro, a sua duração, entre outros.[31]

No caso de micro serviços, como é uma arquitetura divida em vários módulos que comunicam entre si, o traces ganham outra importância. Estes tracers mostram o caminho entre os vários módulos, o tempo que estes demoraram e eventos que encontraram pelo caminho. Estes tracers chamam-se tracers distribuídos e são diferentes que o tracers normais. Utilizam cabeças de pacote adicionais HTTP para conseguir propagar entre os vários componentes do serviço.[32][33]

3.3 Arquitetura de sistemas de monitorização moderno

Um sistema de monitorização tem de conseguir observar os dados do sistema, alertar a equipa responsável de modo a resolver o problema, caso ocorra um problema grave, guardar os *logs* das anomalia, conseguir a visualização dos dados de forma clara, para que a equipa responsável não perca tempo.

De modo a conseguir um sistema atual de monitorização é necessário que o sistema tenha várias propriedades. Tem de conseguir a coleção de dados, as métricas como já foi referido anteriormente.

Tem de conseguir fazer o armazenamento de dados, a utilização de um sistema monitorização faz que sejam geradas quantidades enormes de dados por dia, o que leva a obrigação de gravar os dados de forma estruturada para optimizar a memória disponível. Para isso são utilizadas várias estruturas. O *Log-structured merge-tree (LSM tree)* esta estrutura de dados e pode ser utilizada no *NoSQL* e é otimizada para escrita de informação como inserções de dados.[34] O problema desta estrutura é necessidade de maior requisitos de memória, pois como *LSM-Tree* não tem uma chave de *index* global a procura de informação resulta num excesso de operações de disco de memória. Outro problema é que não existe qualquer ferramenta de ordenação de memória o faz com que os tempos de memória de procuras sejam bastante altos.

Por causa destes problemas a escolha mais popular é de *Time-Series Data Base(TSDB)*. Esta usa uma coleção de valores observados por um período de tempo criando os dados com a data. A estrutura é utilizada de modo a conseguir uma representação dos dados de modo reduzido, ou seja fazer o *sampling* da informação.[35] É utilizada devida à alta taxa em que os dado são escritos, é possível fazer o *downsampling* de modo que a informação mais antiga e irrelevante seja agregada e assim ocupar menos espaço.

Para conseguir visualizar os dados de forma clara e crítica são utilizados vários tipos de gráficos que compõem uma dashboard. Normalmente são utilizados gráficos, mapas, tabelas e painéis de disposição de dados.

Quando existem valores de alguma métrica fora dos padrões normais que condicionam o desempenho normal da aplicação é necessário avisar a equipa responsável para que seja resolvido o mais rápido possível. Para isso é necessário um sistema de alarme, este tem de conseguir examinar os dados e tem de ser definido um valor, que quando ultrapassado envia um alerta.

Este sistema tem de ser arquitetado a monitorizar os limites das métricas certas e não ser demasiado conservador pois se existirem uma quantidade excessiva de falsos positivos a equipa pode começar a ignorar os falsos positivos e as falhas críticas. Para isso é necessário encontrar o limite do alarme ideal, para que o alarme seja acionado mas não haja erros no sistema e assim, criar o mínimo de falsos positivos possíveis.

Um sistema de monitorização também tem de conseguir fazer a agregação de *logs*. Com a utilização de micro-serviços apareceram vários desafios, entre eles a possibilidade de existir demasiados serviços a gerar o seu próprio *logs* onde o seu tipo pode variar. Para resolver um problema é necessário saber as interações entre os serviços. Outro problema é a existência de *logs*

pouco úteis pois os *logs* só dão a visão que o serviço tem do problema, como os serviços podem estar separados é necessário reunir os *logs*. A figura 3.1 mostra um exemplo de arquitetura de um sistema de monitorização moderno.

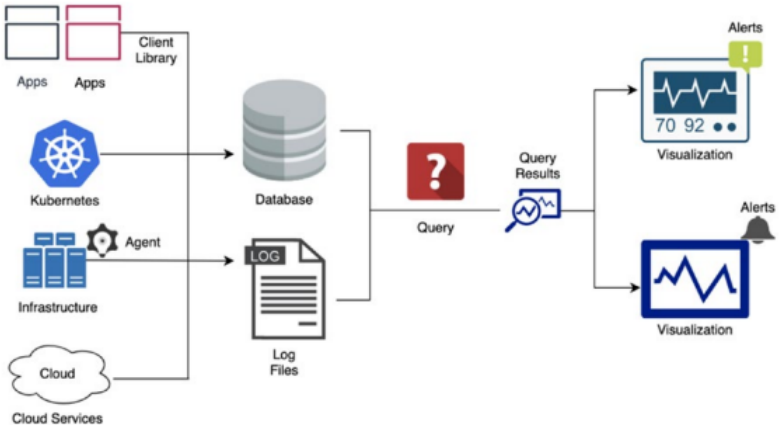

Figura 3.1: Arquitetura de um sistema de monitorização moderno.
[2]

3.4 Visualização

Para conseguir fazer a visualização dos dados é necessária uma visualização clara das métricas, logs e tracers. Para este efeito foi estudado duas ferramentas de visualização, o Kibana e o Grafana. Estas duas aplicações são bastante usadas por utilizadores em pequenas e grandes escalas.

3.4.1 *Kibana*

Kibana é uma plataforma de visualização de *dashboards* que integra o ELK Stack juntamente com o Elasticsearch e o Logstash onde é possível criar gráficos, mapas e histogramas, usando o *Elasticsearch*. Consegue fazer consulta de dados em tempo real utilizando *Beats*[36]. Estes são coletores de informação que transportam esta informação do *Elasticsearch*, ou outro servidor qualquer como por exemplo um servidor *apache*, a informação. Existem dois tipos de *Beats*, os que recolhem os *logs*, *Filebeats*, e os que recolhem informação mais básica, como métricas, estes chamam-se os *Metricbeat*. [37] A figura 3.2 mostra a arquitetura de um sistema de monitorização usando o ELK Stack, ou seja, o Kibana, o Elastic search e o Logstash.

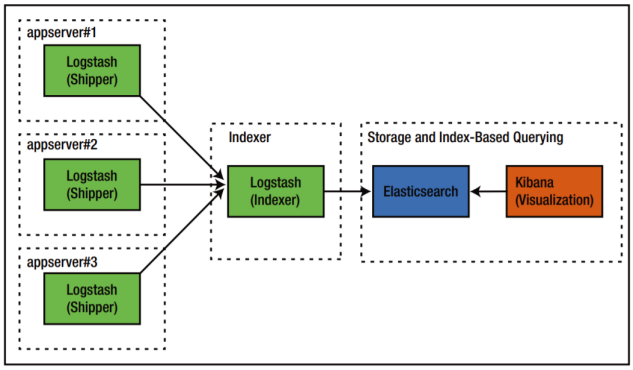

Figura 3.2: Arquitetura de um sistema de *ELK Stack*.
[38]

3.4.2 Grafana

Grafana é uma plataforma grátis que consegue fazer a visualização, pesquisa e alarmística de dados. Este software consegue fornecer ao utilizar ferramentas para que seja possível a visualização dos dados de forma mais fácil com a utilização de gráficos, histogramas e de heat maps, entre outros. Para conseguir estes dados pode utilizar mais de 30 fontes de dados com o intuito de criar dashboards.

Este *software* é aberto à comunidade, ou seja, qualquer pessoa consegue fazer um *plugin* ou um dashboard e partilhá-lo para melhorar a experiência dos utilizadores.[39][2] O Grafana também tem acesso a uma ferramenta de alarmística, consegue criar alarmes com regras definidos para várias métricas.

Ao contrário do *kibana* que necessita do *ElasticSearch*, o *Grafana* é mais versátil conseguindo com a utilização de *plugins* a integração com outras base de dados como *RDBMs*, *MySQL*,*TSDBs*, *PostgresQL*, *InfluxDB* e *Prometheus* sendo também possível combinar dados com origem em várias base de dados.[39]

3.5 Métricas

Para conseguir obter métricas do ambiente foram estudas três aplicações: o Prometheus, New Relic, e o cAdvisor.

3.5.1 Prometheus

Prometheus é um plataforma aberta de monitorização que usa uma base de dados de *TSDB*. É capaz de agregar métricas com um modelo *pull* via *HTTP* de várias origens em tempo real, tem o suporte do modelo *push* de métricas, tem o gestor de alarmes usando uma *HTTP API*.[40]

O Prometheus tem duas possibilidades para obter dos valores das métricas, ou faz um pedido HTTP para a aplicação e esta envia os valores para o Prometheus, ou recolhe estes dados com a ajuda a um exportador. Estes dados são guardados e podem ser usados em gráficos ou fazer

alarmes com recurso à componente do AlertManager. Para conseguir fazer com que as aplicações consigam enviar as métricas o prometheus tem à sua disposição bibliotecas oficiais e não oficiais nas principais linguagens.

Os exportadores são essenciais por existir casos em que não existe a possibilidade conseguir fazer com que o objeto que é necessário monitorizar envie dados formatados para o prometheus. Os exportadores são um pequeno software que se corre juntamente com a aplicação que se quer obter as métricas. Este recolhem e enviam a informação do Prometheus quando é requisitado consoante do tempo for definido pelo utilizador. Para conseguir encontrar estes exportadores de forma automática e assim estabelecer e manter conexão, o Prometheus utiliza um serviço de descoberta. O Prometheus tem a possibilidade de produzir gráficos e painéis com a informação guardada localmente numa base de dados.[41]

Na figura 3.3 é demonstrado um sistema de monitorização usando o Prometheus.

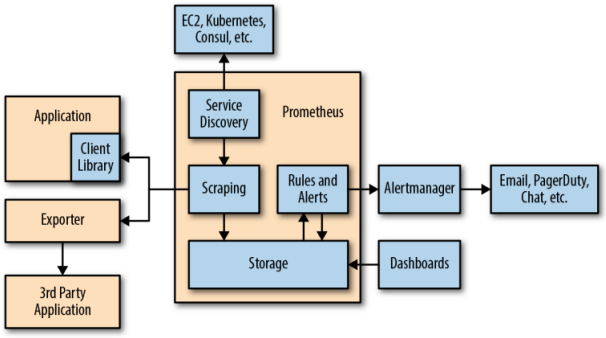

Figura 3.3: Arquitetura de um sistema de monitorização usando o Prometheus. [41]

3.5.2 New Relic

O New Relic é uma plataforma de monitorização focada na performance das aplicações onde a principal vantagem da utilização desta aplicação é o diagnostico de performance e de problemas dentro de aplicações e de sistemas distribuídos. Este sistema oferece um sistema de monitorização das métricas de infraestrutura, alertas e análise de software entre outros. Oferece um bom suporte de infraestrutura como código e suporta a criação de dashboards para a visualização das métricas.[42]

New Relic para conseguir obter as métricas, implementa um agente que recolhe as métricas em tempo real onde é necessário que esteja um em cada plataforma.[43]

A principal desvantagem deste serviço é o seu custo, que é muito elevado para cada container.

3.5.3 cAdvisor

cAdvisor é uma ferramenta de análise de métricas de containers. Esta ferramenta coleta, agrega e exporta métricas de todos os conteiners que estão a funcionar num ambiente de kubernetes.

Para conseguir estas métricas o cAdvisor é intalado junto ao kubelet um agente e assim consegue obter as métricas de cada container. O software tem uma interface para se conseguir visualizar as métricas mas é uma interface bastante menos sofisticada que o Grafana ou o Kibana.[2]

3.6 Logs

De modo a encontrar a melhor ferramenta para se ser possível recolher, agregar, visualizar logs foram estudadas as seguintes aplicações: Grafana Loki, Logstash, Fluentd.

3.6.1 Grafana Loki

Grafana Loki é um sistema de agregação de logs inspirado no prometheus. Este sistema recolhe os *logs* em texto simples em vez de *JSONs* sendo um nome e valor indexados a cada *log*. [40] As principais vantagens deste sistema são:

1. Conseguir guardar os logs de forma eficiente e com custo baixo.
2. Conseguir gerar métricas e alertas a partir dos logs.
3. Recolher os logos em tempo real, conseguindo fazer a sua pesquisa por o período em que foram criados.
4. A integração nativa com Kubernetes, o Prometheus e Grafana conseguindo no mesmo UI fazer a visualização de métricas, Logs e traces [44]

O *Grafana Loki* utiliza o *Promtail* que é um agente que encaminha os *logs* para o *Loki*.[45] Assim, este sistema tem como principal procedimento descobrir o alvo de busca logs, adicionar labels necessárias aos fluxos de logs e reenviá-los para o Loki.[46]

3.6.2 Logstash

O *Logstash* é uma ferramenta de envio de *logs* e de eventos que fornece uma estrutura integrada para a coleção de *logs*, para a centralização, análise e procura. Consegue receber a informação necessário por ficheiros, *TPC/UDP*, *Syslog*, *stdin*, entre outros mecanismos e enviar os dados por *TCP/UDP*, email, *HTTP*, entre outros serviços de rede. Este *software*, além de conseguir fazer a agregação dinâmica dos *logs* e de eventos de várias fontes, habilita a indexação de tempo aos ficheiros de modo a ser possível fazer uma busca mais eficiente.[38][36]

O *Elasticsearch* é um *server* de procura baseado no *Apache Lucene*. Este consegue assegurar um motor de busca de texto em tempo real e distribuído, e foi criado de modo a ser utilizado com o logstash. Várias grandes empresas multinacionais como a *Netflix, Microsoft, EBay* tem uma implementação do *Elasticsearch* no seu sistema de forma a conseguir fazer a busca de dados e a sua analise. Este *software* é leve e pode ser utilizado por computadores simples ou por *clusters* complexos, fornece uma interface *RESTful* interface usando *JSON* sobre *HTTP* sendo toda a informação guardada em *JSONs*.[38]

3.6.3 Fluentd

Fluentd é uma ferramenta de agregação de logs, ou seja, reúne logs produzidos em várias fontes e os junta usando apenas uma aplicação. Esta ferramenta é utilizada, por norma, com o Elasticsearch em substituição do Logstash. Este ferramenta tem uma arquitetura baseada em plugins e, assim, recebe os logs e os reencaminha para uma base de dados.

Esta ferramenta em comparação com o Logstash é parecida em termos de desempenho sendo o Fluentd um pouco mais leve. [47]

3.7 Tracing

Para encontrar a melhor solução de obter os logs foram estudas três ferramentas: o Jaeger, Zipkin, o AWS X-Ray.

3.7.1 Jaeger

Jaeger é uma aplicação grátis de tracing distribuido que suporta integrações para REST API. Esta ferramenta permite a visualização das traces através de uma interface de web e consulta de logs com querys.

As bibliotecas do Jaeger são baseadas no OpenTracing que é o standard de coletar dados, o OpenTracing é incluído no Cloud Native Computing Foundation.

Jaeger utiliza agentes para conseguir validar, indexar e guardar os traces onde o Jager disponibiliza um serviço de API endpoints stateless. Para conseguir guardar a informação nos casos normais são usados base de dados como por exemplo o Elasticsearch ou o Cassana.[48] A figura 3.4 mostra a arquitetura do Jaeger.

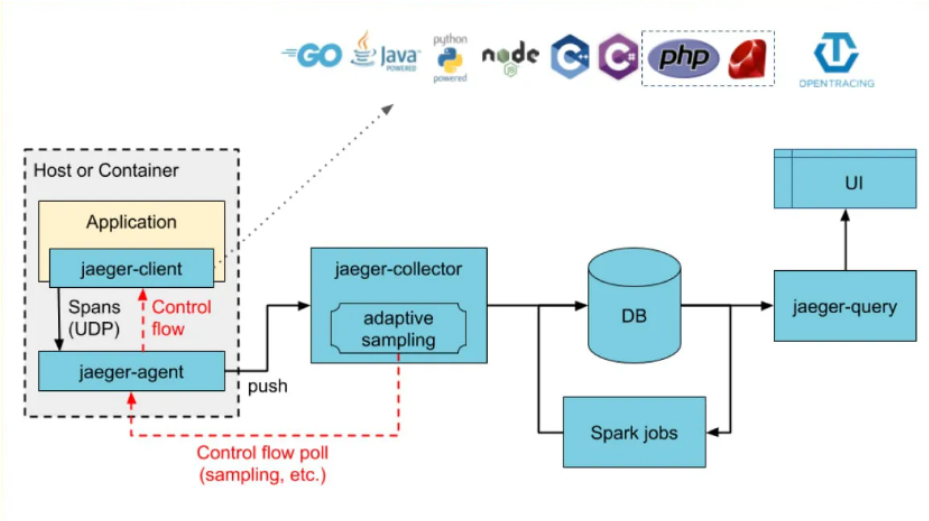

Figura 3.4: Arquitetura exemplo do Jaeger.
[48]

3.7.2 Zipkin

Zipkin é uma aplicação grátis de tracers distribuído esta aplicação ao contrário do Jaeger não utiliza o OpenTracing mas sim as suas próprias bibliotecas de instrumentação. Também utiliza agentes que recolhem e indexam a informação.

Este software utiliza uma API JSON para encontrar o tracers na base de dados como o Elasticsearch ou o Cassana igualmente como o Jaeger. O Zipkin também tem uma interface web que pode ser usada para visualizar o tracers do utilizador.[49]

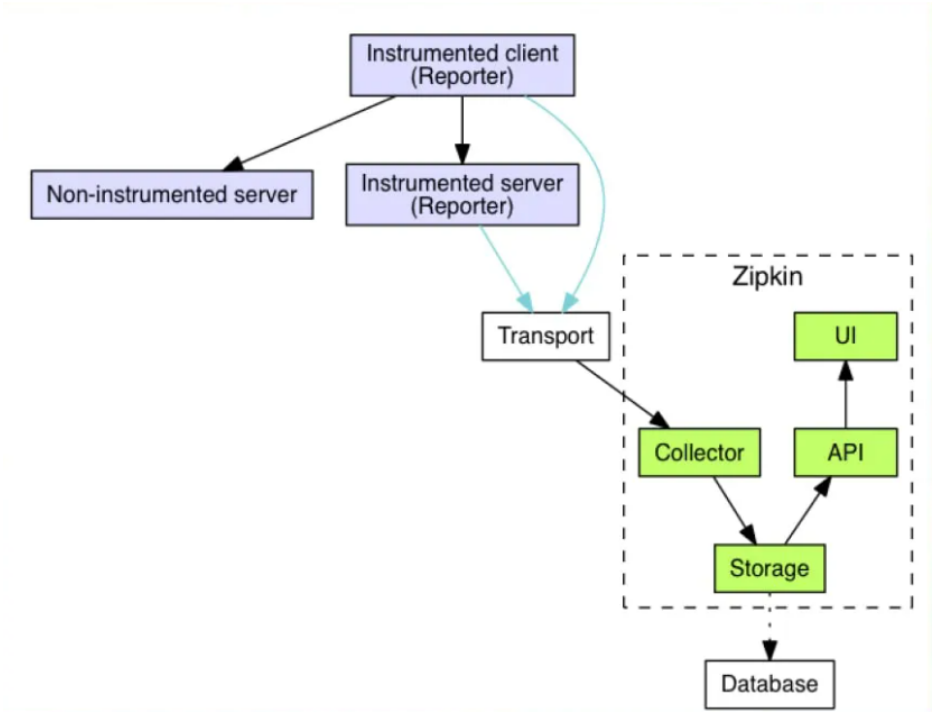

Figura 3.5: Arquitetura do Zipkin.
[49]

3.7.3 AWS X-Ray

O AWS X-Ray é uma aplicação paga fornecida pela Amazon e tem como objetivo identificar e solucionar erros de performance em aplicações de arquitetura de micro serviços. Esta ferramenta consegue criar tracers distribuídos, agregando os dados de modo a expor ao utilizador uma visão completa do desempenho do ambiente e da aplicação.[50]

Para conseguir esta visão o X-Ray organiza os dados por segmentos e utiliza uma interface web para os mostrar ao utilizador.

Este software utiliza exportadores que recebem os segmentos JSON e os reencaminha para o modulo principal do X-Ray.[51]

3.8 Escolha de software para o projeto

Para se conseguir um sistema de monitorização completo é necessário ter um sistema que consiga recolher métricas, logs e fazer traces. Para este efeito, foi escolhido das tecnologias acima referidas, o Grafana para fazer a visualização das métricas e os logs, o Prometheus para recolher as métricas e concretizar a alarmística, o Grafana Loki para recolher e indexar o logs e o Jaeger para realizar e observar os traces.

O Prometheus foi escolhido pois é uma ferramenta grátis ao contrário do New Relic e sendo este projeto baseado numa ausência de custos, dúvidas não há em relação à utilização daquele em detrimento deste. Em relação cAdvisor, este não foi escolhido porque existem métricas importantes no sistema que este não as recolhe, como por exemplo, o número de pacotes que o message broker transporta, o que é essencial com o uso da aplicação da Tlantic.

O Grafana foi a ferramenta de visualização escolhida, para se observar os dados uma vez que, é uma ferramenta mais versátil que o Kibana, o qual não consegue receber os dados do Prometheus.

Para coletar logs foi eleito o Grafana Loki pois é uma ferramenta que tem uma completa integração com o Grafana.

Para o tracers optou-se pelo Jaeger por ser uma aplicação grátis ao contrário do AWS X-RAY, cujo custo total não faz sentido num perspetiva global e por ser escalável ao contrário do Zipkin.

Capítulo 4

Arquitetura do sistema

Neste capítulo vai ser abordado a arquitetura do sistema de monitorização com o software que foi escolhido no capitulo anterior, ou seja, um sistema de monitorização formado pelo Grafana, Grafana Loki, Prometheus e Jaeger e as suas interações com os componentes da aplicação da Tlantic. Para conseguir fazer a implementação deste sistema também é discutida a arquitetura do Terraform de forma a automatizar a instalação do sistema.

4.1 Desenvolvimento do Sistema

O sistema de monitorização foi implementado num sistema Kubernetes utilizando um cluster de AWS Could para fazer a implementação e configuração da infraestrutura de forma automática. Para conseguir esta automatização foi utilizado o Terraform. Foram implementados dois namespaces diferentes para diferenciar as aplicações da Tlantic: o ambiente de desenvolvimento e o ambiente das aplicações de monitorização.

Neste ambiente de monitorização foi implementado o Prometheus, o Grafana Loki e o Jaeger para recolher métricas, Logs e Tracers, respetivamente. Para conseguir fazer a alarmística foi utilizado o Alertmaneger que é uma componente do Prometheus para fazer a notificação do Slack. Para conseguir fazer a visualização dos Logs e métricas foi implementado o Grafana.

Na figura 4.1, sem pormenorizar as comunicações entre os principais componentes, é possível observar qual o elemento que interage para que os objetivos do trabalho sejam alcançados. O ambiente de desenvolvimento é secção do cluster que foi implementado pela equipa da Tlantic e é constituído por o Mobile Retail Suit, o RabbitMQ e o Nginx onde o ambiente de monitorização vai interagir para conseguir obter as métricas.

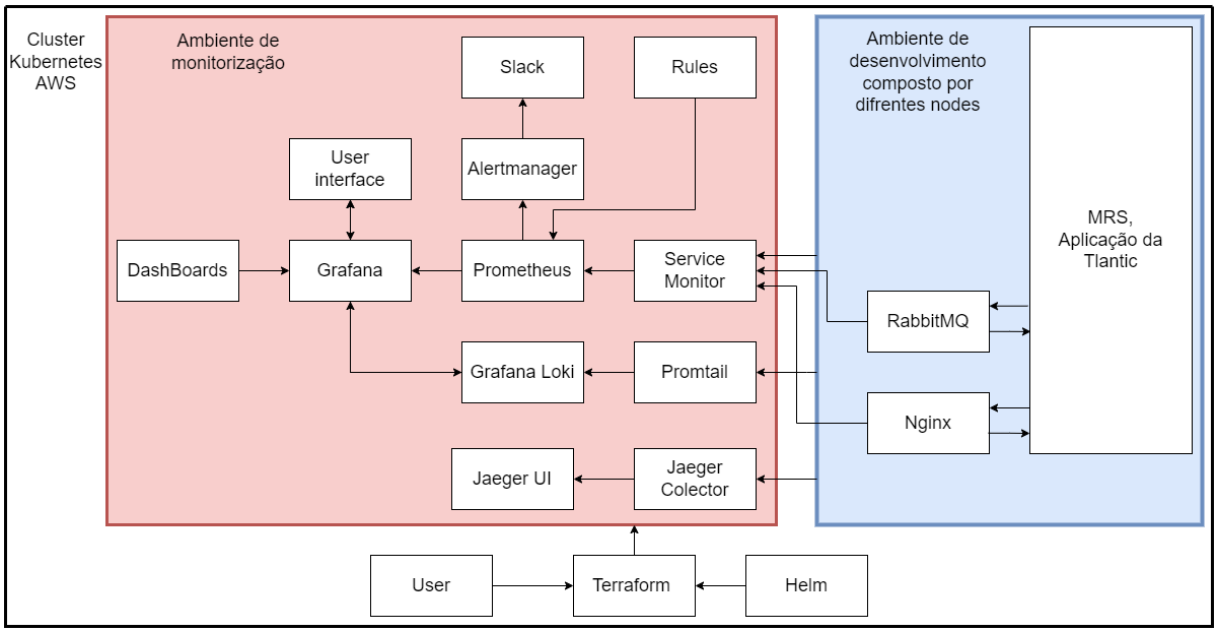

Figura 4.1: Arquitetura de um sistema de monitorização usando Grafana, Loki e Alertmanager.

Nas próximas secções vai ser abordado o ambiente de desenvolvimento e o o ambiente de monitorização bem como uma análise mais profunda à figura 4.1.

4.2 Ambiente de desenvolvimento

4.2.1 Mobile Retail Suit

O Mobile Retail Suit, MRS, é uma das aplicações da Tlantic que faz um sistema de gestão e de monitorização para empresas de retalho como super-mercados, tem como funções gestão de inventários, gestão de stock, reabastecimento, gestão de etiquetas de preços e produtos e redução das quebras e ruturas, entre outros.[52] Esta aplicação tem uma arquitetura em micro-serviços e foi desenvolvida de modo a ser implementada num ambiente de Cloud AWS com utilização de Kubernetes.

Esta aplicação é constituída por vários módulos como base de dados, gateways, entre outros, que estão implementados em vários pods e é necessário que estes comuniquem entre si. Para conseguir esta comunicação é utilizado um broker de mensagens, o RabbitMQ. Para conseguir fazer o balanceamento da rede é utilizado o Nginx. A figura 4.2 demonstra isso mesmo os vários módulos a comunicar entre si e com o message broker (nesta imagem o load balancer não está representado). Esta aplicação esta divida por 75 módulos cada um com o seu helm chart.[52]

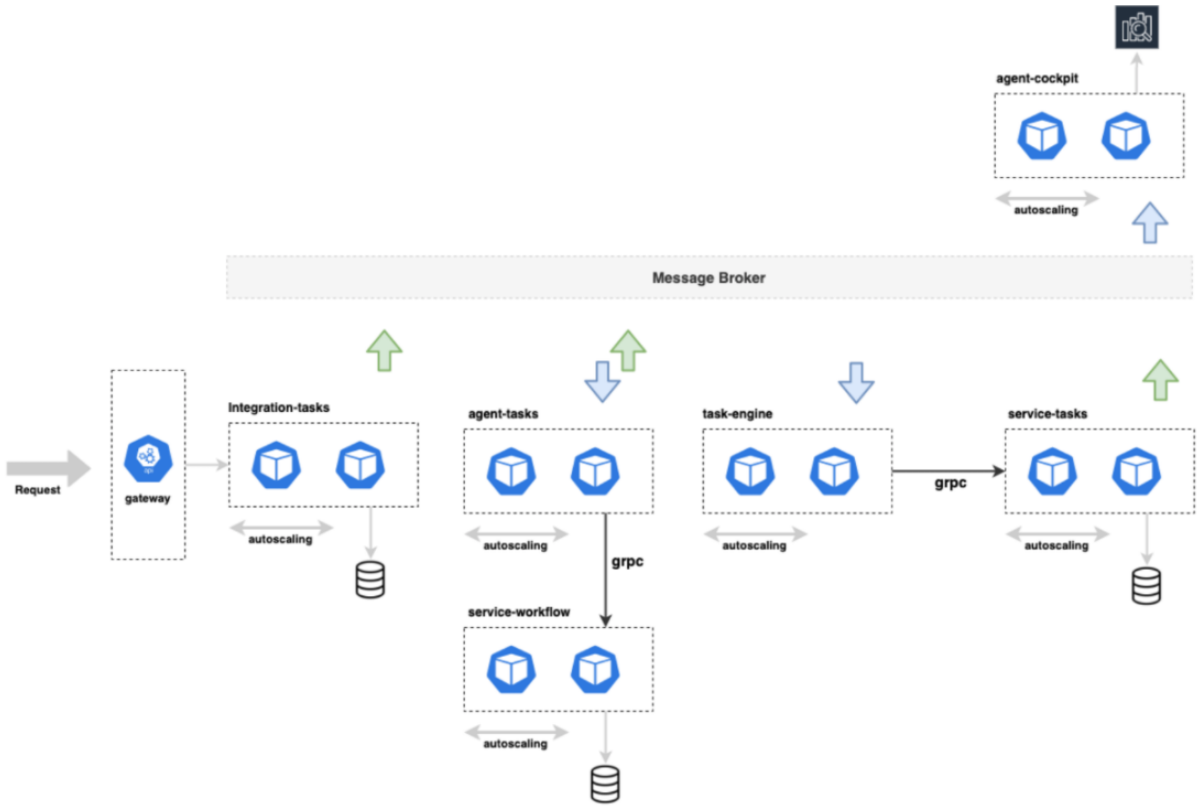

Figura 4.2: Aplicação modelo onde vai ser implementado o sistema de monitorização. [53]

4.2.2 RabbitMQ

Para conseguir fazer o transporte de mensagens a Tlantic usa no MRS o RabbitMQ que é um dos brokers de mensagens open-source mais amplamente usados. Este software é usado como intermediário entre aplicações independentes, foi originalmente baseado no AMQP (Advanced Message Queuing Protocol).

Este software cria várias filas de mensagens onde cada módulo consegue enviar e receber aquelas de modo a separar o que cada um recebe.

Estas filas de mensagens são frequentemente usadas em arquitetura de micro serviços devido à necessidade de comunicação entre os diferentes módulos. Para fazer esta comunicação entre serviços, a Tlantic escolheu usar o RabbitMQ.[54]

Uma questão muito importante, numa aplicação de micro-serviços, é o seu broker de mensagens. Com efeito, existem várias métricas que necessitam de ser monitorizadas como por exemplo o número de conceções a cada fila ou o número de mensagens em espera, tudo a denunciar que existem problemas no ambiente.

4.2.3 Nginx

O Nginx é um servidor reverso de proxy ou seja, é uma aplicação que redireciona pedidos do utilizador para uma rede privada de forma a criar uma linha de abstração e controlo entre o servidor e o cliente, sendo possível a utilização de DNS para facilitar a utilização de pedidos HTTPs. Este software consegue analisar pedidos baseando-se no seu URI e decidir como proceder com o pedido de modo a fazer o roteamento do tráfego de uma rede.

Este software cria logs com códigos do estado dos pedidos pedidos HTTPs respondendo com 2xx caso o pedido seja um sucesso, 3xx na condição de haver uma redirecção do pedido, 4xx na hipótese de erro por parte e 5xx se houver um erro por parte do servidor. Por isso, caso ocorra um erro no serviço existirá um número anormal de logs com o código 5xx sendo necessário fazer uma monitorização cuidada para que este valor não seja um valor alto. [55] A figura 4.3 mostra como o Nginx se comporta num servidor web.

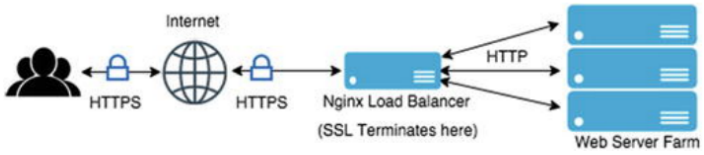

Figura 4.3: Exemplo de a aplicação de Nginx como Load balance.
[55]

Esta aplicação é importante pois demonstra que o MRS está funcional e conectável em face de métricas importantes. Estas métricas mostram o estado da aplicação a partir do tempo de resposta, taxa de erros, conceções aceites, entre outros.

4.3 Ambiente de monitorização

4.3.1 Prometheus

O Prometheus, como já foi referido, foi o software de monitorização escolhido para fazer a obtenção de métricas.

A figura 4.4 mostra as ligações entre os componentes necessários para a visualização das métricas em Dashboards no Grafana e a criação de notificações para o Slack. As principais ligações são entre o Prometheus e os nós e entre o Prometheus e o Grafana. De modo a que o Prometheus consiga saber que estes módulos existem e a que software necessita de obter as métricas, este utiliza um sistema de Service discovery e o Prometheus Operator.

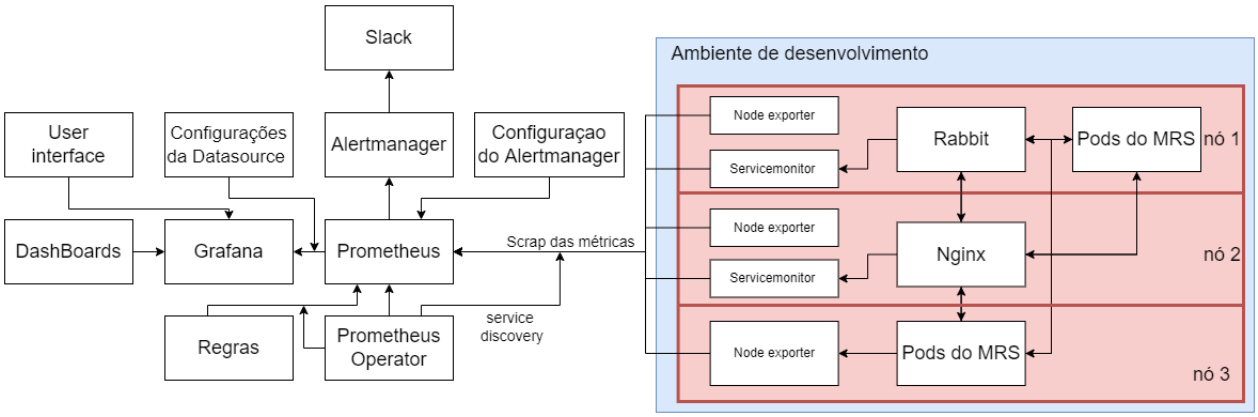

Figura 4.4: Arquitetura do Prometheus, Grafana e o Ambiente de desenvolvimento.

4.3.1.1 Service discovery e Prometheus Operator

Service discovery é a forma abstrata que a aplicação tem identificar outra, ou seja, o Prometheus necessita de identificar o serviço que precisa de monitorizar. Como cada serviço ou máquina pode desaparecer a qualquer momento, o Prometheus necessita saber o que precisa de monitorizar a todos os instantes, caso algum componente deixa de responder.

Para conseguir fazer este reconhecimento o Prometheus tem à sua disposição várias técnicas. Pode fazer a configuração estática do alvo, onde é ajustado no Prometheus um código estático manualmente, que utiliza as portas de rede. Outra técnica é a utilização de ficheiros que o Prometheus lê e assim, reconhece os alvos conforme os ficheiros, não necessitando de usar a rede. Esta técnica consegue integrar alvos que não estão preparados para o Prometheus automaticamente. Estas duas técnicas tem uma desvantagem enorme com a utilização de kubernetes pois, caso um nó ou serviço reinicie e o seu número de porta porta de entrada de rede mude é necessário modificar essa configuração.

Para conseguir fazer automatização do Service discovery foi escolhido utilizar o Prometheus Operator, esta técnica fornece uma implantação nativa de Kubernetes onde o operator faz a implementação e a configuração de forma automática dos componentes do Prometheus.

As principais características do Prometheus Operator são:

1. Utilizar kubernetes CR para criar e gerir os recursos como o Prometheus, Alertmanager e outros.

2. A simplicidade de configuração destes recursos num ambiente nativo de kubernetes utilizando réplicas, políticas de retenção entre outros.

3. Automatizar a criação de alvos de monitorização utilizando o sistema de labels do kubenetes.

Outra principal vantagem é a monitorização do servidor de API do kubernetes para mudanças em custom resources, CR, e assegurar o que Prometheus varie com estas mudanças. Estes Cr são:

1. Prometheus, define a implementação de Prometheus pretendida.
2. Alertmanager, define a implementação de Alertmanager pretendida.
3. ThanosRuler, define a implementação de Thanos Ruler pretendida. Thanos Ruler é o componente que permite o processamento de regras de alertas e regras de monitorização por múltiplas plataformas de Prometheus.
4. ServiceMonitor, declara os grupos de serviços de Kubernetes que necessitam de ser monitorizados. O Operator gera automaticamente a configuração de scrape baseada na configuração atual dos objetos do servidor de API.
5. PodMonitor, declara os grupos de pods de Kubernetes que necessitam de ser monitorizados, tal como o ServiceMonitor.
6. Probe, declara os grupos de alvos estáticos de Kubernetes que necessitam de ser monitorizados, tal como o ServiceMonitor e o PodMonitor.
7. PrometheusRule, define a Regras de alarmes ou de gravação. O Operator cria o ficheiro da regra automaticamente que são utilizadas pelo Prometheus e o Alertmanger.
8. AlertmanagerConfig, declara a configuração do Alertmanger possibilitando a criação de vários caminhos para recetores variados.

Para conseguir extrair as métricas do Kubernetes, o Prometheus utiliza o Node exporter. Node exporter é um exportador de métricas oficial do Prometheus que captura métricas de sistema baseadas em Linux. Este software apenas captura as métricas da própria máquina que está implementado, ou seja não captura métricas de serviços ou processos na máquina, captura sim métricas como a utilização do CPU, espaço disponível no disco, utilização de rede entre outros.

No caso deste problema, com a utilização de Kubernetes, o Prometheus utiliza um daemonset para confirmar a utilização de instância em cada nó de modo a conseguir obter métricas do nó em detalhe. [56]

Para conseguir fazer a obtenção das métricas, o Prometheus faz o scrape, isto é, envia um pedido HTTP de modo a que seja reenviado as métricas pretendentes, algumas métricas úteis também são adicionadas, como se o scrape foi sucedido e o tempo que demorou. Estes scrapes são configuráveis e acontecem em casos normais entre 10 e 60 segundos em cada alvo. Os scrapes são feitos ao node exporter e aos serviceMonitor do Nginx e do Rabbit.

4.3.2 Grafana

Grafana é o software que fez a criação e a visualização de dashboards, estas dashboards são constituídas por tabelas, gráficos, outros tipos de visualização das métricas e do logs . Para fazer esta visualização de interface o Grafana utiliza no Kubernetes um serviço para onde é possível fazer o comando kubectl port-forward para criar ligação entre o utilizador e o Grafana.

É possível criar dashboards a partir da interface e com a utilização de configmaps é possível carregar automaticamente estas dashboards previamente criadas e assim sempre que se instalar o Grafana. Estes painéis são formatados em ficheiros json.

Para o Grafana obter a informação necessária para as dashboards utiliza data sources. Estes são usados para criar a ligação entre os programas com a informação necessária e assim criar os gráficos. Existem vários tipos de data sources como o OpenTSDB, PostgreSQL, Grafana Loki, Prometheus entre muitos outros. Para conseguir obter esta informação, o Grafana necessita do endereço de IP, a porta do servidor, as credenciais de acesso e o nome da aplicação. Esta Data sources podem ser configurados diretamente no Grafana ou pela interface, como um dos objetivos do trabalho é automatizar o sistema completo foi feita a configuração dos data sources no Grafana.[57]

4.3.2.1 Alertmanager

O Alertmanager é um componente do Prometheus e recebe alarmes e transforma-os em notificações. Estas notificações incluem email, aplicações de conversa como o Slack e serviços como o Pagerduty.

O Alertmanager não transforma apenas alarmes em notificações, tendo várias funções, as principais são:

1. Caso ocorra um erro grave, as notificações de erros com menor preocupação têm de ser inibidas, pois a equipa não pode perder o foco no alarme mais importante.
2. Este software consegue agregar conjuntos de alarmes numa só uma notificação, de modo a que não ocorra uma descarga desnecessária de notificações num utilizador.
3. A possibilidade de conseguir silenciar certos alarmes que sejam esperados.
4. Cada equipa está focada em objetivos diferentes, por isso é necessário, caso ocorra um erro que as diferentes equipas obtenham diferentes notificações.
5. Caso as notificações sejam esquecidas ou perdidas o Alertmanager tem a possibilidade de configuração do tempo em que elas serão reenviadas.
6. As notificações podem ter templates com o seu conteúdo customizado modo de a facilitar a perceção da leitura por parte dos utilizadores. [56]

Estas notificações podem ser configuradas para diferentes rotas e diferentes estilos de notificações. As configurações são implementadas com a utilização de um ficheiro YAML normalmente chamado *alartmangaer.yaml*.

4.3.3 Grafana Loki

O Grafana Loki como já foi referido é uma ferramenta de recolha e visualização de logs para conseguir fazer esta visualização. O Grafana Loki foi implementado por diferentes módulos entre eles o Distributor, Ingester, Querier, Promtail e a base de dados.

O módulo Distributor recebe e valida os dados recolhidos pelo Promtail. Para validar os dados o Distributor verifica cada log para assegurar que está de acordo com a especificação necessária, este processo inclui a verificação das Labels e verifica se o tempo de criação dos logs é valido, ou seja, se é muito antigo ou demasiado recente. Depois de esta validação os logs são transmitidos para o Ingester.[58]

O módulo do Ingester escreve os dados recebidos na base de dados, antes de os registar, o Ingester indexa os logs em vez de os guardar, log a log, na base de dados. Quando existe uma pesquisa na memória também existe uma pesquisa no Ingester e, caso o Ingester possua essa informação, ele retorna os dados, como mostra as setas azuis da figura 4.5. Os Logs são divididos por categorias, usando as labels, e são guardados em chunks que depois são armazenados na base de dados.

O modulo querier é usado para receber e processar as queries do utilizador e enviar os resultados. Este modulo permite a realização de pesquisas na base de dados e no Ingester. Ao analisar uma quantidade tão extensiva de memória o modulo vai ter que lidar com dados duplicados. Devido a este fator existem mecanismos internos que apenas retornam queries com os mesmos dados uma vez.

Para conseguir escrever um log na base de dados, o distributor recebe um pedido HTTP do promtail para guardar streams de dados. Cada stream é codificado usando um hash ring. O distributor envia para o ingesters cada stream de dados e este cria um chunk ou junta a stream um chunk já criado anteriormente.[59] Para finalizar, o distributor responde com um código de sucesso via conexão HTTP. Este caminho é mostrado com as setas pretas na figura 4.5.

Para conseguir fazer a leitura de um log, o querier recebe um pedido para dados, query, via HTTP. O modulo querier lê e retorna dos dados caso exista correspondência. Depois o querier carrega os dados da base de dados e corre a query para verificar se existem logs, caso haja, faz uma outra verificação para confirmar que não existem duplicados. Por fim, o querier envia os dados finais via HTTP. Este caminho é mostrado pelas setas azuis na figura 4.5.

Promtail é o agente que encaminha o logs para uma instância de Grafana Loki. Este agente é implantado em cada máquina, ou nó, para conseguir recolher os logs desse ambiente. O Promtail é implementado num sistema de Daemonset, ou seja,é colocado em todos os nós. Os principais objetivos do Promtail são descobrir os alvos, anexar às streams de dados labels e enviar para estes conjuntos de dados para uma instância de Grafana Loki.

O Promtail para conseguir conseguir obter o logs utiliza o sistema de Service discovery do Prometheus ou seja, usando o Kubernetes API encontra o que necessita usando as labels.

Para fazer a visualização dos log o utilizador utiliza o Grafana da mesma forma que já foi explicado no Prometheus, sendo a principal diferença a introdução de querys no Grafana para se conseguir obter esta visualização.

A figura 4.5 mostra a interação entre os vários módulos do Grafana Loki, o Grafana e o Ambiente de desenvolvimento.

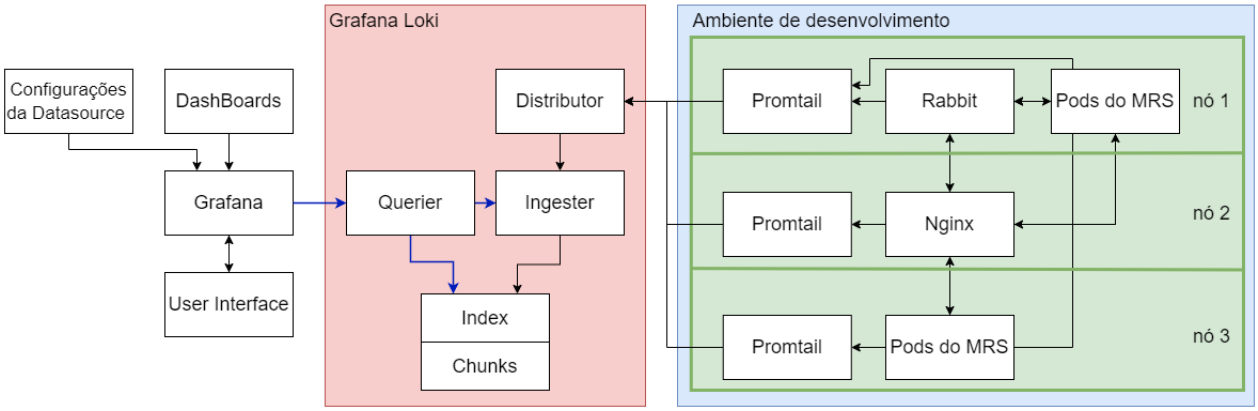

Figura 4.5: Arquitetura de um sistema de recolha e visualização de logs usando o Grafana Loki e o Grafana.

4.3.4 Jaeger

O Jaeger como já foi referido é o software escolhido para recolher logs. Este programa é constituído por dois módulos, o collector e o query. O Jaeger collector recebe traces a partir de um programa que utiliza as bibliotecas do opentelemetry, depois as traces passam através de uma pipeline de modo a serem validadas e melhoradas e são posteriormente guardadas na base de dados. O Jaeger tem suporte de várias bases de dados, neste projeto vai ser usado a base de dados Cassandra.

O modulo Jaeger query é o serviço que expõem as APIs para receber traces da base de dados. Este serviço tem uma interface Web que possibilita ao utilizador analizar e fazer a pesquisa de traces.

O trace é formado por um programa que necessita de ter acesso aos serviços de Kubernetes por isso para a sua utilização precisa de abrir conexão entre o serviço e o programa. Este programa foi escrito de modo a utilizar as bibliotecas do OpenTelemetry. A figura 4.6 mostra estes módulos do Jaeger e a sua interação.[60]

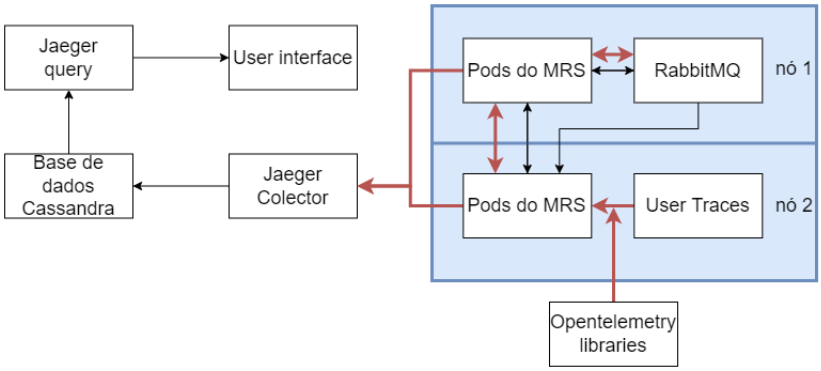

Figura 4.6: Arquitetura de visualizaçao de Traces utilizando o Jaeger.

4.3.5 Terraform

Terraform, como já foi dito, foi o software escolhido para automatizar o processo de instalação, configuração e upgrade da infraestrutura. Para isto, o terraform regista informação acerca da infraestrutura num ficheiro que contém um formato JSON personalizado. Este ficheiro que normalmente é chamado terraform.tfstate, é usado para registar um mapa de recursos terraform nos ficheiros de configuração, este mapa é uma representação desses recursos no próprio hardware. Para fazer mudanças no sistema, o Terraform compara essas configurações com o estado do sistema para determinar que alterações são necessárias fazer ao sistema. Assim, utiliza providers de modo a comunicar com o cluster.

Providers adiciona o tipo de recursos que iram ser implementados ou a fonte de dados que o terraform consegue gerir. Todos os tipos de recursos são implementados por um provider, ou seja, sem o Providers o Terraform não consegue gerir algum tipo de infraestrutura. A figura 4.7 mostra como o Terraform vai interagir com o cluster de AWS de modo a conseguir alterar os componentes.[23]

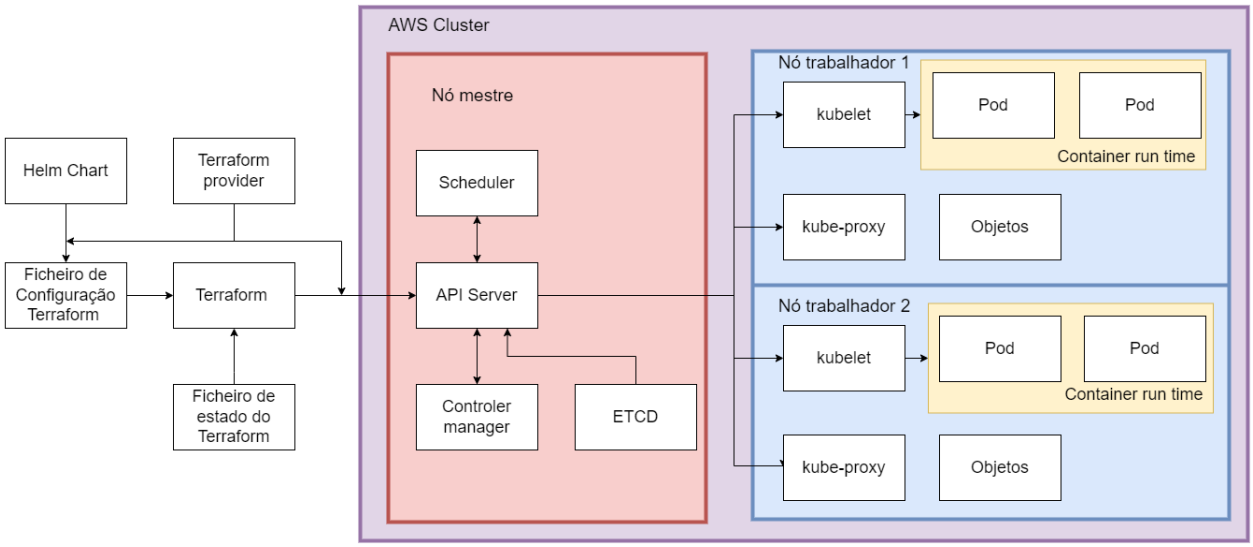

Figura 4.7: Arquitetura de um sistema de monitorização usando Grafana, Loki e Alertmanager. [40]

Capítulo 5

Implementação e Resultados

Neste capitulo é abordada a implementação e resultados da arquitetura previamente descrita no capítulo anterior. Foi feita uma implementação por fases onde se implementou cada componente individualmente e, assim, conseguir obter um sistema de monitorização completo.

5.1 Terraform

Para conseguir utilizar o Terraform foi instalado o kubectl a partir do URL https://kubernetes.io/docs/tasks/tools. Em seguida foi feita configuração necessária no ficheiro KUBECONFIG para se obter a interação e acesso do cluster com o Terraform. Este código é observável na lista 5.1

Listing 5.1: Código do ficheiro KUBECONFIG.

```
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: **********
    server: https://**********.gr7.eu-west-1.eks.amazonaws.com
  name: arn:aws:eks:eu-west-1:********:cluster/tlantic-dev-qual-n

contexts:
- context:
    cluster: arn:aws:eks:eu-west-1:*********:cluster/tlantic-dev-qual-n
    user: arn:aws:eks:eu-west-1:******:cluster/tlantic-dev-qual-n
  name: arn:aws:eks:eu-west-1:*********:cluster/tlantic-dev-qual-n

current-context: arn:aws:eks:eu-west-1:*******:cluster/tlantic-dev-qual-n
kind: Config
preferences: {}
users:
```

```
- name: arn:aws:eks:eu-west-1:***********:cluster/tlantic-dev-qual-n
  user:
    exec:
      apiVersion: client.authentication.k8s.io/v1beta1
      args:
      - --region
      - eu-west-1
      - eks
      - get-token
      - --cluster-name
      - tlantic-dev-qual-n
      command: aws
      env: null
      interactiveMode: IfAvailable
      provideClusterInfo: false
```

Depois foi instalado o Terraform a partir do URL oficial: https://learn.hashicorp.com/tutorials/terraform/install-cli.

Foram feitos ficheiros de configuração para que o Terraform consiga obter acesso ao cluster, o ficheiro 5.2 mostra o código utilizado para implementar estes providers. Este ficheiro tem dois providers um para fazer ligação ao cluster e outro para conseguir fazer obtenção dos helm chats.

Listing 5.2: Ficheiro de Terraform providers

```
provider "helm" {
  kubernetes {
    config_path = pathexpand(var.kube_config)
  }
}

provider "kubernetes" {
  config_path = pathexpand(var.kube_config)
}
```

No primeiro provider foram utilizados variáveis, como mostra o código 5.2. Foi preciso configura-la, utilizando uma variável para no futuro facilitar uma hipotética modificação. Como mostra o código 5.3.

Listing 5.3: Configuração de variável para os providers.

```
variable "kube_config" {
  type    = string
  default = "Kubeconfig"}
```

Em seguida foi feita a criação do namespace de desenvolvimento e do namespace de monitorização foi feito um ficheiro namespaces.tf. Este ficheiro tem o código mostrado pela 5.4.

Listing 5.4: Ficheiro de criação de namespaces, namesapces.tf.

```
resource "kubernetes_namespace" "dev" {
  metadata {
    name = var.namespace
  }
}
resource "kubernetes_namespace" "monitoring" {
  metadata {
    name = "monitoring"
  }
}
}
```

Para concluir a instalação do MRS foi feito um ficheiro Terraform para cada módulo, devido ao número alto de recursos necessários para a aplicação foi feito um Programa em Python de modo a automatizar a escrita dos ficheiros Terraform variando o nome e helm chart de cada módulo. O código 5.5 mostra um exemplo como estes ficheiros resultaram do programa.

Estes ficheiros tem dependência da criação do namespace de desenvolvimento para garantir que não haja erros, ou seja, a criação do infraestrutura do MRS não pode ocorrer sem existir estes namespaces. A dependência do RabbitMQ deve-se ao fato da ocorrência de erros com as filas de dados sendo necessário a instalação do RabbitMQ antes dos módulos do MRS.

Listing 5.5: Ficheiro exemplo dos recursos do MRS neste caso o mrs-integration-pricing.

```
resource "helm_release" "mrs-integration-pricing" {
  chart      = "mrs-integration-pricing"
  name       = "${var.namespace}-mrs-integration-pricing"
  namespace  = var.namespace
  repository = var.charts
  version    = "v1.0.0-build.5"

  depends_on = [kubernetes_namespace.dev, helm_release.rabbitmq]
```

Para concluir, os ficheiros de recursos do ambiente de desenvolvimento foram feitos ficheiros Terraform para o RabbitMQ e Nginx, a figura 5.6 mostra o ficheiro do RabbitMQ. O ficheiro de configuração values.yalm é utilizado para modificar as configurações normais do chart do RabbitMQ e do Nginx.

Listing 5.6: Ficheiro de Terraform do RabbitMQ

```
resource "helm_release" "rabbitmq" {
```

```
  chart      = "rabbitmq"
  name       = "rabbitmq-${var.namespace}"
  namespace  = var.namespace
  repository = "./rabbitmq/chart"

  depends_on = [kubernetes_namespace.dev]


  values = [
    "${file("rabbitmq/values.yaml")}"
  ]
}
```

Para se conseguir confirmar os resultados utilizou-se o comando kubectl get pods -A no terminal que mostra todos os pods em todos os namespaces. isso é mostrado pela figura 5.1 que mostra parte dos pods instalados pelo Terraform usando Helm charts.

```
C:\Users\apsc0>kubectl get pods -n dev
NAME                                                READY   STATUS             RESTARTS   AGE
dev-mrs-gateway-api-b5b7948db-94g42                 1/1     Running            0          8d
dev-mrs-service-gateway-integration-7d957f8b79-xpmdw 1/2    CrashLoopBackOff   658        8d
dev-mrs-service-gateway-service-f87876d85-g877p     2/2     Running            373        8d
dev-nginx-ingress-ingress-nginx-controller-6656c9b8cf-nbnl8 0/1 Pending        0          8d
dev-nginx-ingress-ingress-nginx-defaultbackend-6c49c45c8b-kfhcb 0/1 Pending    0          8d
mrs-agent-consolidation-9d45bfbf5-qzzhh             2/2     Running            4          8d
mrs-agent-inventory-79767b4f89-zf675                2/2     Running            3          8d
```

Figura 5.1: Resultado da aplicação dos ficheiros Terraform.

5.2 Prometheus e Grafana

Para se conseguir obter as métricas fez-se a instalação do Prometheus utilizando o helm Chart https://github.com/bitnami/bitnami-docker-prometheus-operator e a instalaçao do Grafana ultizando o Helm Chart https://github.com/grafana/grafana. Para a instalação ser possível criou-se os ficheiros Terraform da mesma forma que o MRS e para se conseguir obter os valores das métricas do RabbitMQ e do Nginx foram alterado os valores dos Charts para fazer a ativação do serviceMonitor que já o chart tem esta opção. A fim de se obter as métricas de cada nó do cluster o Helm chart já vinha com o node exporter que é reconhecido automaticamente pelo Prometheus.

Para se conseguir visualizar estas métricas foram configuradas datasources do Prometheus e do Gragafana Loki no ficheiro values.yaml do Grafana de modo a receber a informação esta configuração é demonstrada por 5.7.

Listing 5.7: Código de configuração de datasources no Grafana

```
datasources:
  datasources.yml:
    apiVersion: 1
    datasources:
    - name: Prometheus
      type: prometheus
      url: http://kube-kube-prometheus-prometheus:9090
      access: proxy
      isDefault: true
    - name: Loki
      type: loki
      access: proxy
      url: http://loki:3100
      basicAuth: true
      basicAuthUser: my_user
      basicAuthPassword: test_password
```

Em seguida foram criadas dashboards para o RabbitMQ, para o Nginx e para as métricas do Cluster. Estas foram configuradas de modo a que quando se inicializa o Grafana sejam instaladas automaticamente. Como a comunidade do Grafana é aberta estas dashboards foram conseguidas via externa. Estas dashboards foram obtidas pelos URLs https://github.com/dotdc/grafana-dashboards-kubernetes, https://grafana.com/grafana/dashboards/14900-nginx/ e https://grafana.com/grafana/dashboards/10991-rabbitmq-overview/ Os resultados podem observados pelas figuras 5.2, 5.3, 5.4, 5.5, 5.6.

As figuras 5.2, 5.3, 5.4, mostram dashboards com as métricas recolhidas pelo node exporter do cluster e mostram uma dashboard com métricas em relação a um nó, ao sistema completo e a um pod, respetivamente.

As figuras 5.5, 5.6 mostram dashboards com as métricas recolhidas com o service monitor do RabbitMQ e do Nginx,respetivamente.

Figura 5.2: Dashboard criada para a visualização de métricas do nó do cluster no Grafana.

Figura 5.3: Dashboard criada para a visualização de métricas do cluster no Grafana.

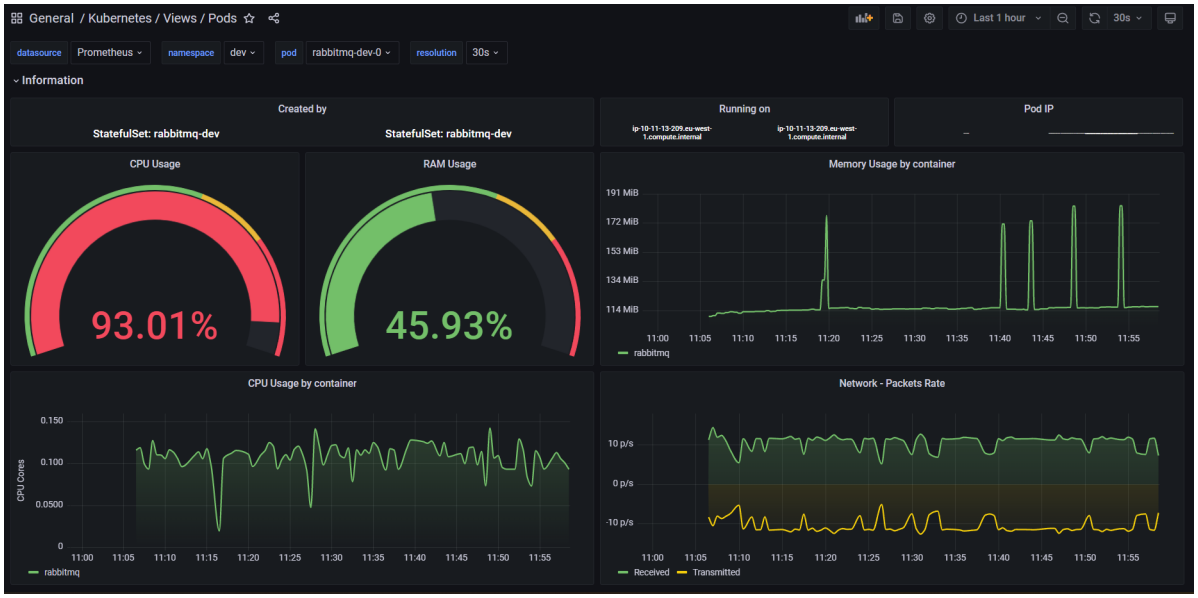

Figura 5.4: Dashboard criada para a visualização de métricas do pod RabbitMQ no Grafana.

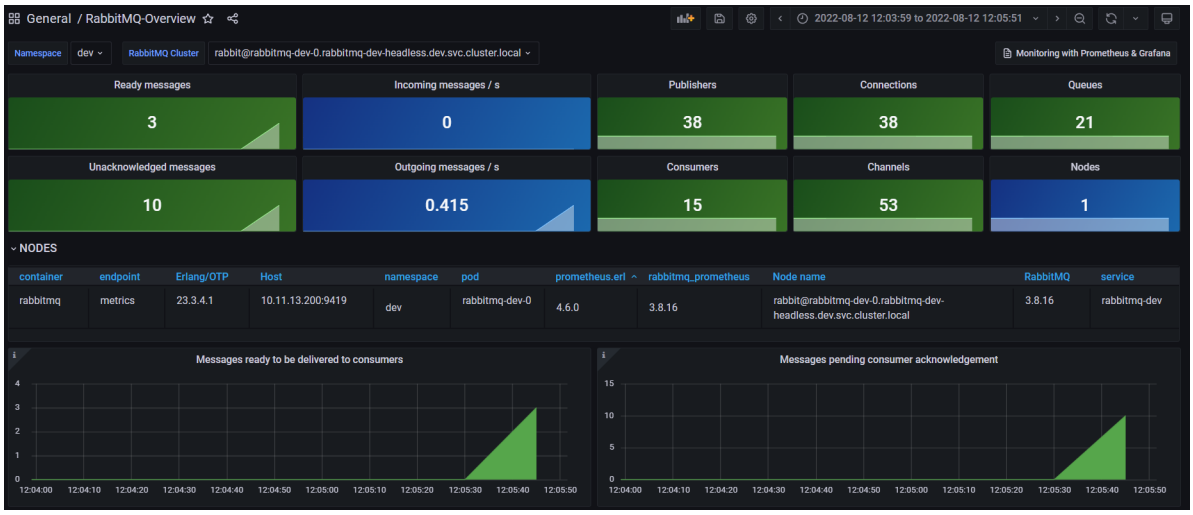

Figura 5.5: Dashboard criada para a visualização de métricas da aplicaçao RabbitMQ no Grafana.

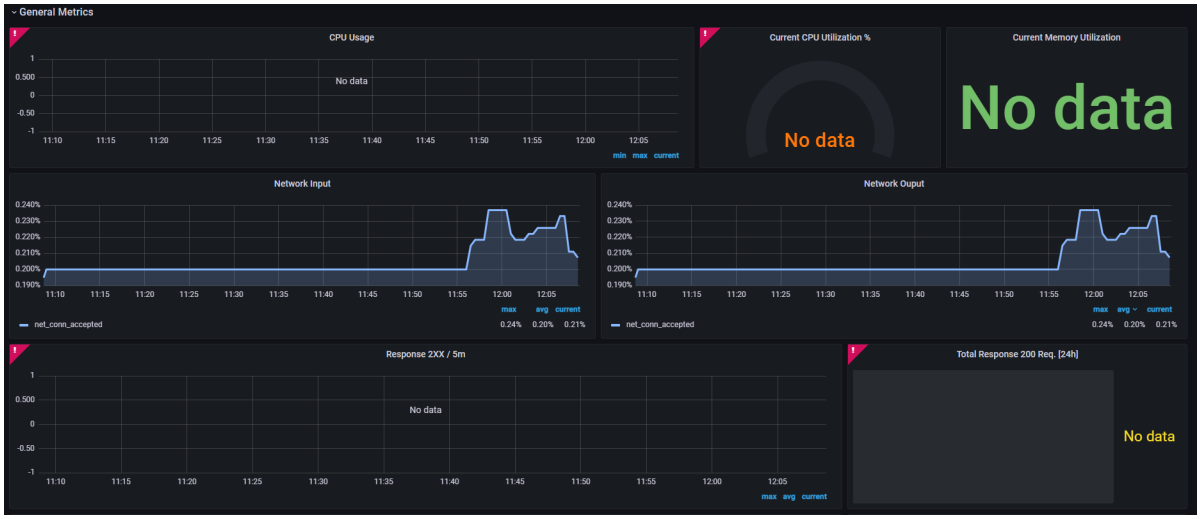

Figura 5.6: Dashboard criada para a visualização de métricas da aplicaçao Nginx no Grafana.

5.2.1 Alertmanager

Para conseguir obter notificações foi necessário fazer a configuração do Alertmanager. Para isto foi necessário criar um Webhook no servidor do Slack para o Alertmanager conseguir fazer a comunicação. Esta configuração é mostrada pelo código A.2.

Para conseguir efetuar notificações foram criadas várias regras uma delas foi a Regra "number_of_queues"que verifica o número de mensagens em todas as filas de espera e estas não podem ultrapassar 6 ,este número foi definido devido a facilidade para testar a regra. Para conseguir este teste foram colocadas mensagens no RabbitMQ de modo a que o limite da regra seja ultrapassado e o Alertmanager ative e por consequência o Slack receba uma notificação.

Foram criadas outras regras, como a regra "KubernetesPodCrashLooping", que faz a alarmística se algum pod entrar em crashloopbackoff, ou seja, caso o pod tenha restarts sem sucesso, o

Alertmanager envia notificação ao Slack caso ocorra mais de 5 restarts num minuto. O código para implementação destas regras encontra-se em A.1 , esta mostra o código usado para o Terraform criar o recurso customizado PrometheusRules com a regras anteriormente explicadas.

A figura 5.7 demonstra a interface do Slack da notificação feita pelo Alertmanager na regra "number_of_queues".

Figura 5.7: Notificação do Slack pela Regra "number_of_queues".

5.3 Grafana Loki

Para conseguir fazer a visualização dos logs no Grafana foi instalado a partir de um ficheiro Terraform, tal como sucedeu com o Prometheus e o Grafana, um Helm Chart https://github.com/grafana/helm-charts/tree/main/charts/loki-stack.

Foi configurado a datasource para fazer com que Grafana mostrasse os logs como se apresenta no código 5.7. Para confirmar que o Grafana consegue obter os logs do sistema foi feita uma query ao pod do MRS "mrs-agent-notification". Esta confirmação pode ser confirmado pela figura 5.8

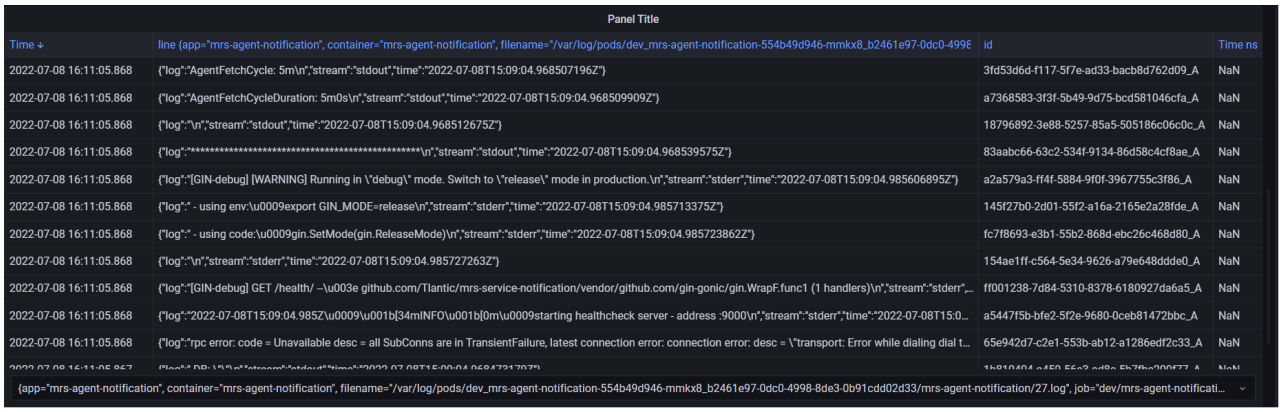

Figura 5.8: Exemplo de uma query de logs a um componente da aplicação da Tlantic.

5.4 Jaeger

Para fazer a instalação do Jaeger procedeu-se ao mesmo procedimento das outras tecnologias, utilizando o Helm chart https://github.com/jaegertracing/jaeger-kubernetes.

Para conseguir fazer a visualização do traces foi escrito um programa em GOlang usando as bibliotecas do opentelemetry. O código A.3 é a parte mais importante de um destes programa que foram feitos para a visualização de tracers. Este programa faz um loop infinito que em cada ciclo gera 5 tracers. Cada um destes enviam um pedido HTTP para a base de dados para confirmar a comunicação com esta, outro para confirmar a existência de conteúdo na base de dados, utilizando querys, e os outros três tracers criam um item na base de dados, atualizam-no e apagam-no.

As figuras 5.9, 5.10 demonstram a interface gráfica do Jaeger e a visualização de traces. A primeira figura uma janela de visualização mais genérica podendo observar-se quando os tracers chegam e a sua duração. E a segunda figura, uma abordagem mais especifica sendo possível analisar com mais detalhe cada tracer.

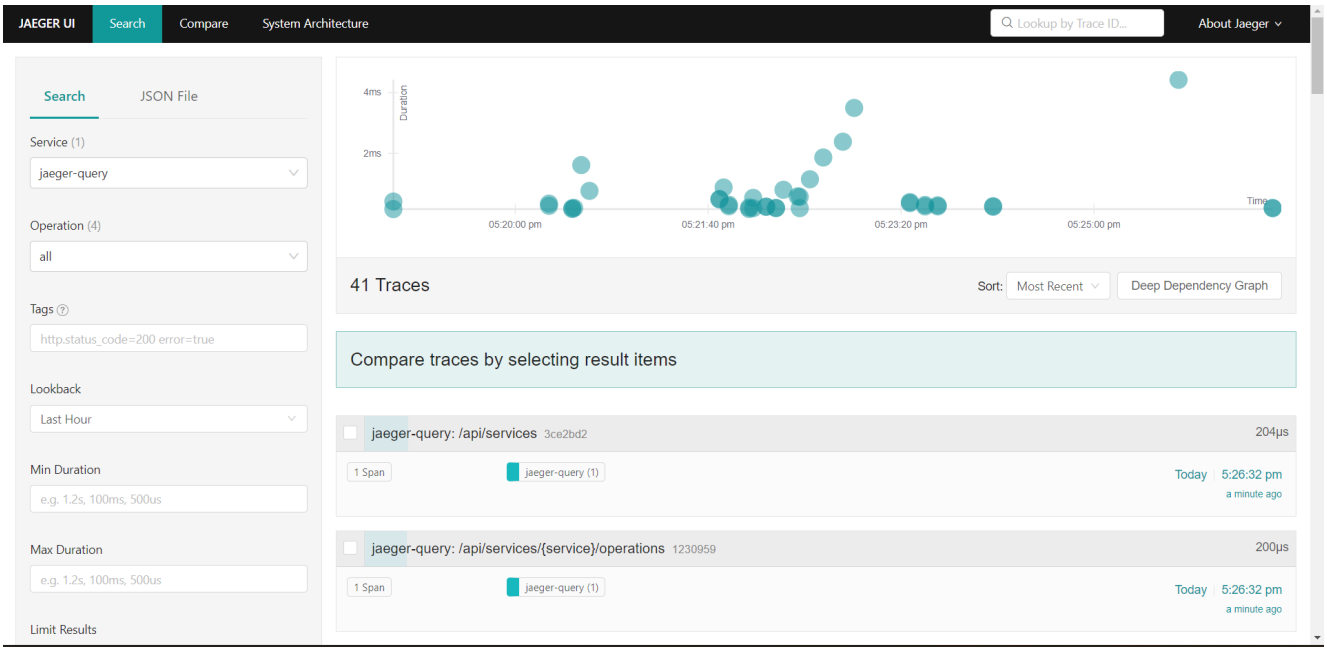

Figura 5.9: Interface do Jaeger com querys com vista geral sobre o sistema.

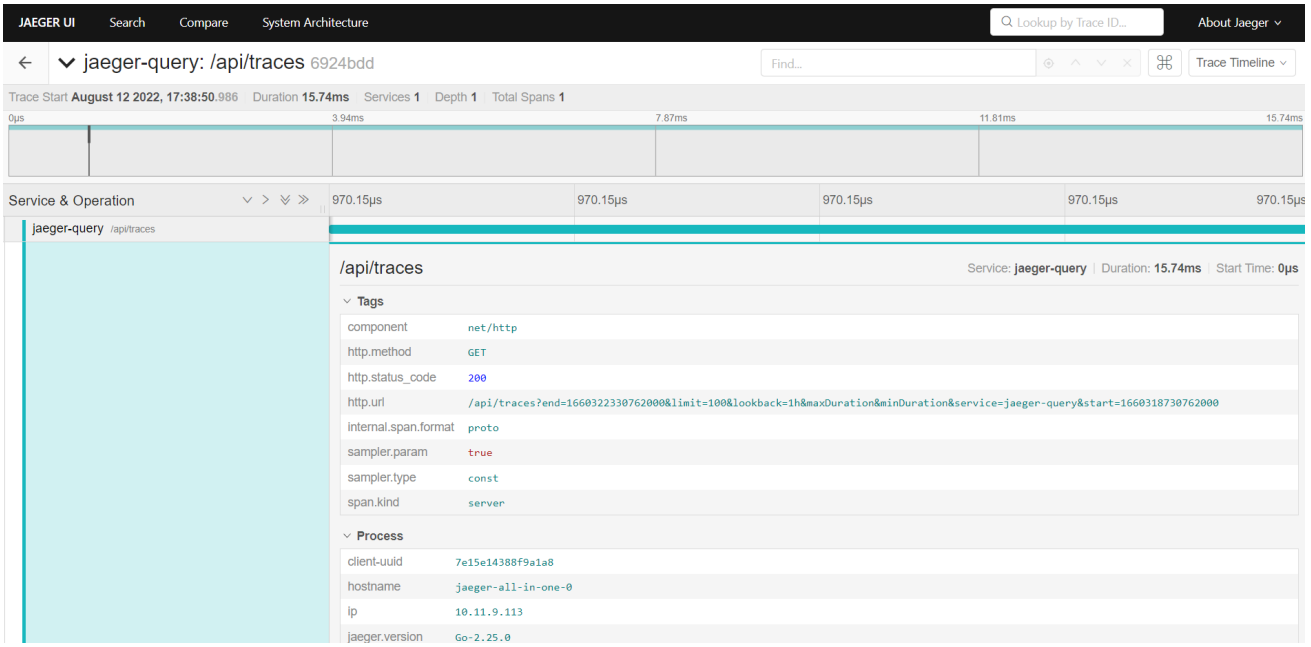

Figura 5.10: Interface do Jaeger com querys com vista sobre uma query.

5.5 Análise de Resultados

Com o desenvolvimento deste sistema de monitorização podemos concluir que os objetivos deste projeto foram completados. O Teraform automatizou a criação dos namespaces e todos os

módulos do MRS, RabbitMQ e Nginx. No entanto, houve problemas no cluster, mais concretamente problemas com a rede interna do cluster. Estes problemas resultaram que MRS não ficou complemente funcional e que o Nginx não funcionasse na sua plenitude. Por isso, não foi possível executar todas as funcionalidades do MRS e nenhuma do Nginx. Independentemente destes problemas o sistema de monitorização ficou funcional e automatizado.

Foi possível obter as métricas no cluster e do RabbitMQ mas não se conseguiu obter as métricas do Nginx devido ao problema acima referido, como se pode observar pela figura 5.6. Utilizando estas métricas foi feito um sistema de alarmistica que envia notificações para o Slack.

Como esperado foi possível recolher os logs do MRS com o Grafana Loki sem apresentar qualquer problema, como mostra a figura 5.8. E para finalizar, também foi realizada a produção de tracers no ambiente. Estes tracers não são distribuídos devido ao fato de ser necessário fazer alteração no código do MRS. Assim, foi decido em conjunto a empresa não fazer esta mudança. Pois, para realizar tracers distribuídos é necessário que pods do MRS consigam realizar troca de informação o que complica a formação destes tracers.

Capítulo 6

Conclusão

Atualmente, com a proliferação da utilização da computação em *cloud*, os sistemas ficaram muito mais dinâmicos e dependem de outros serviços para completarem funcionalidades totais. Para haver uma certeza absoluta que todos os processos estão a funcionar de forma adequada foi realizado um sistema de monitorização. Este sistema tem de atingir todos os aspetos de observabilidade, ou seja, colecionar métricas, logs e tracers e ainda conseguir realizar a alarmística.

Para isso, foi arquitetado um sistema de monitorização. O Grafana Loki foi usado para recolha e coleção de logs. O Prometheus foi utilizado para recolha de métricas e concretizar a alarmística através do AlertManager. O Grafana foi empregado para visualização dos logs e das métricas. Por último, o Jaeger fez a recolha de tracers e a sua visualização.

Assim, este sistema foi implementado sem grandes problemas e chegou-se à conclusão que num sistema kubernetes os aspetos mais importantes a monitorizar são os elementos da rede, tais como o número de mensagens perdidas no RabbitMQ e o número de erros no Nginx. Este sistema está automatizado podendo ser feita a sua instalação num outro cluster de kubernetes apenas utilizando um comando do Terraform.

Para trabalho futuro será necessário corrigir os erros de rede no cluster, bem como a escrita de código para a formação de tracers distribuídos. Outro aspeto a melhorar são as dashboards do Grafana que podem ficar mais especificas e completas para ambiente da presente aplicação. Os logs recolhidos pelo Grafana Loki podem também ser aperfeiçoados e assim beneficiar a equipa na leitura dos mesmos.

Para concluir, este projeto foi realizado com sucesso, tendo sido criado um sistema de monitorização sem custos e com a possibilidade de implementação automática.

Anexo A

Código utilizado

Listing A.1: Código para implementação das regras de Prometheus.

```
resource "helm_release" "kube" {
  name       = var.prometheus_name
  depends_on = [kubernetes_namespace.monitoring]
  repository = "./prometheus/chart"
  chart      = "kube-prometheus"

  namespace = "monitoring"
  values = [
    "${file("prometheus/values.yaml")}"]
}

resource "kubernetes_manifest" "prometheusrule0" {

  depends_on = [kubernetes_namespace.monitoring]

  manifest = {
    "apiVersion" = "monitoring.coreos.com/v1"
    "kind" = "PrometheusRule"
    "metadata" = {
      "annotations" = {
        "meta.helm.sh/release-name" = "prometheus"
        "meta.helm.sh/release-namespace" = "default"
      }
      "labels" = {
        "app" = "kube-prometheus-stack"
        "app.kubernetes.io/instance" = "prometheus"
        "app.kubernetes.io/managed-by" = "Helm"
```

```
        "app.kubernetes.io/part-of" = "kube-prometheus-stack"
        "app.kubernetes.io/version" = "33.1.0"
        "chart" = "kube-prometheus-stack-33.1.0"
        "heritage" = "Helm"
        "release" = "kube"
      }
      "name" = "prometheus-kube-prometheus-kube-state-metrics"
      "namespace" = "default"
    }
    "spec" = {
      "groups" = [
        {
          "name" = "rabbitmq_alert"
          "rules" = [
            {
              "alert" = "number_of_queues"
              "expr" = "rabbitmq_queue_messages_ready␣>6"
              "for" = "30s"
              "labels" = {
                "severity" = "slack"
              }
            },
            {
              "alert" = "KubernetesPodCrashLooping"
              "annotations" = {
                "description" = <<-EOT
                "Pod␣{{␣$labels.pod␣}}␣is␣crash␣looping"
                  VALUE = {{ $value }}
                  LABELS = {{ $labels }}""
                EOT
                "summary"="Kubernetespodcrashlooping(instance␣{
{$labels.instance}})"
              }
              "expr" = "increase(kube_pod_container
_status_restarts_total[1m])>5"
              "for" = "30s"
              "labels" = {
                "severity" = "slack"
              }
            },
```

```
          ]
        },
      ]
    }
  }
}
```

Listing A.2: Configuração do AlertManager.

```
global:
  resolve_timeout: 2m

  receivers:
  -
    name: slack_general
    slack_configs:
      -
        api_url: "https://hooks.slack.com/services/***/**/**"
        channel: "#general"
        icon_url: https://avatars3.githubusercontent.com/u/3380462
        title: |-
          [{{ .Status | toUpper }}{{ if eq .Status "firing" }}:
          {{.Alerts.Firing | len}}{{ end }}]{{.CommonLabels.alertname}} for
          {{ .CommonLabels.job }}
          {{- if gt (len .CommonLabels) (len .GroupLabels) -}}
            {{" "}}(
            {{- with .CommonLabels.Remove .GroupLabels.Names }}
              {{- range $index, $label := .SortedPairs -}}
                {{ if $index }}, {{ end }}
                {{- $label.Name }}="{{ $label.Value -}}"
              {{- end }}
            {{- end -}}
            )
          {{- end }}
        text: >-
          {{ range .Alerts -}}
          *Alert:* {{.Annotations.title }}{{ if .Labels.severity }-`{{

          .Labels.severity }}`{{ end }}

          *Description:* {{ .Annotations.description }}
```

```
          *Details:*
            {{ range .Labels.SortedPairs }}.*{{ .Name }}:* `{{ .Value }}`
            {{ end }}
          {{ end }}
  -
    name: slack

  route:
    group_by:
      - cluster
    receiver: slack
    routes:
      -
        group_wait: 1m
        matchers:
          - severity = "slack"
        receiver: slack_general
```

Listing A.3: Parte do Código para a formação de Tracers.

```
func initTracer() func(context.Context) error {
        exporter, err := jaeger.New(jaeger.WithCollectorEndpoint
        (jaeger.WithEndpoint(collectorURL)))
        if err != nil {
                log.Fatal(err)
        }
        resources, err := resource.New(
                context.Background(),
                resource.WithAttributes(
                        attribute.String("service.name", "resources"),
                        attribute.String("library.language", "go"),
                ),
        )
        if err != nil {
                log.Printf("Could not set resources: ", err)
        }

        otel.SetTracerProvider(
                sdktrace.NewTracerProvider(
                        sdktrace.WithSampler(sdktrace.AlwaysSample()),
```

```
                        sdktrace.WithBatcher(exporter),
                        sdktrace.WithResource(resources),
                ),
        )
        return exporter.Shutdown
}
func main() {

        cleanup := initTracer()
        defer cleanup(context.Background())

        r := gin.Default()
        r.Use(otelgin.Middleware(serviceName))
        // Connect to database
        models.ConnectDatabase()

        // Routes
        r.GET("/resources", controllers.FindResources)
        r.GET("/tasks/:id", controllers.FindBook)
        r.POST("/tasks", controllers.CreateBook)
        r.PATCH("/tasks/:id", controllers.UpdateBook)
        r.DELETE("/tasks/:id", controllers.DeleteBook)

        // Run the server
        r.Run(":8091")
}



func FindBook(c *gin.Context) {
        // Get model if exist
        var book models.Book
        if err := models.DB.WithContext(c.Request.Context()).
        Where("id=?", c.Param("id")).First(&book).Error; err != nil {
           c.JSON(http.StatusBadRequest,gin.H{"error":"Recordnotfound!"})
           return
        }

        c.JSON(http.StatusOK, gin.H{"data": book})
}
```

```
// POST /books
// Create new book
func CreateBook(c *gin.Context) {
        // Validate input
        var input CreateBookInput
        if err := c.ShouldBindJSON(&input); err != nil {
                c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
                return
        }

        // Create book
        book := models.Book{Name: input.Name, StartDate: time.Now().String()}
        models.DB.WithContext(c.Request.Context()).Create(&book)

        c.JSON(http.StatusOK, gin.H{"data": book})
}

// PATCH /books/:id
// Update a book
func UpdateBook(c *gin.Context) {
        // Get model if exist
        var book models.Book
        if err := models.DB.WithContext(c.Request.Context()).Where("id=?",
        c.Param("id")).First(&book).Error; err != nil {
                c.JSON(http.StatusBadRequest,gin.H{"error":"Recordnotfound!"})
                return
        }

        // Validate input
        var input UpdateBookInput
        if err := c.ShouldBindJSON(&input); err != nil {
                c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
                return
        }

        models.DB.WithContext(c.Request.Context()).Model(&book).Updates(input)

        c.JSON(http.StatusOK, gin.H{"data": book})
}
```

```
// DELETE /books/:id
// Delete a book
func DeleteBook(c *gin.Context) {
        // Get model if exist
        var book models.Book
        if err := models.DB.WithContext(c.Request.Context()).Where("id=?",
        c.Param("id")).First(&book).Error; err != nil {
            c.JSON(http.StatusBadRequest,gin.H{"error":"Recordnotfound!"})
            return
        }

        models.DB.Delete(&book)

        c.JSON(http.StatusOK, gin.H{"data": true})
}
```

Referências

[1] Mordor Intelligence. Cloud microservices market - growth, trends, covid-19 impact, and forecasts (2021-2026). 2020.

[2] Mainak Chakraborty e Ajit Pratap Kundan. *Monitoring Cloud-NativeApplications: Lead Agile Operations ConfidentlyUsing Open Source Software*. Apress;, 2021.

[3] Martin Fowler e James Lewis. Microservices: Nur ein weiteres konzept in der softwarearchitektur oder mehr. *Objektspektrum*, 1(2015):14–20, 2015.

[4] Davide Taibi, Valentina Lenarduzzi, Claus Pahl, e Andrea Janes. Microservices in agile software development: a workshop-based study into issues, advantages, and disadvantages. Em *Proceedings of the XP2017 Scientific Workshops*, páginas 1–5, 2017.

[5] Yu Gan e Christina Delimitrou. The architectural implications of cloud microservices. *IEEE Computer Architecture Letters*, 17(2):155–158, 2018. doi:10.1109/LCA.2018.2839189.

[6] Roberto Morabito, Jimmy Kjällman, e Miika Komu. Hypervisors vs. lightweight virtualization: A performance comparison. Em *2015 IEEE International Conference on Cloud Engineering*, páginas 386–393, 2015. doi:10.1109/IC2E.2015.74.

[7] Nishant Deepak Keni e Ahan Kak. Adaptive containerization for microservices in distributed cloud systems. Em *2020 IEEE 17th Annual Consumer Communications Networking Conference (CCNC)*, páginas 1–6, 2020. doi:10.1109/CCNC46108.2020.9045634.

[8] Rabindra K. Barik, Rakesh K. Lenka, K. Rahul Rao, e Devam Ghose. Performance analysis of virtual machines and containers in cloud computing. Em *2016 International Conference on Computing, Communication and Automation (ICCCA)*, páginas 1204–1210, 2016. doi:10.1109/CCAA.2016.7813925.

[9] David Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014. doi:10.1109/MCC.2014.51.

[10] Thanh-Tung Nguyen, Yu-Jin Yeom, Taehong Kim, Dae-Heon Park, e Sehan Kim. Horizontal pod autoscaling in kubernetes for elastic container orchestration. *Sensors*, 20(16), 2020. URL: https://www.mdpi.com/1424-8220/20/16/4621, doi:10.3390/s20164621.

[11] Yuqi Fu, Shaolun Zhang, Jose Terrero, Ying Mao, Guangya Liu, Sheng Li, e Dingwen Tao. Progress-based container scheduling for short-lived applications in a kubernetes cluster. Em *2019 IEEE International Conference on Big Data (Big Data)*, páginas 278–287, 2019. doi:10.1109/BigData47090.2019.9006427.

[12] Víctor Medel, Omer Rana, José ángel Bañares, e Unai Arronategui. Modelling performance resource management in kubernetes. Em *Proceedings of the 9th International Conference on Utility and Cloud Computing*, UCC '16, página 257–262, New York, NY, USA, 2016. Association for Computing Machinery. URL: https://doi.org/10.1145/2996890.3007869, doi:10.1145/2996890.3007869.

[13] Jay Shah e Dushyant Dubaria. Building modern clouds: Using docker, kubernetes google cloud platform. Em *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*, páginas 0184–0189, 2019. doi:10.1109/CCWC.2019.8666479.

[14] Babak Bashari Rad, Harrison John Bhatti, e Mohammad Ahmadi. An introduction to docker and analysis of its performance. *International Journal of Computer Science and Network Security (IJCSNS)*, 17(3):228, 2017.

[15] Ashish Sharma, Sarita Yadav, Neha Gupta, Shafali Dhall, e Shikha Rastogi. Proposed model for distributed storage automation system using kubernetes operators. Em Vanita Jain, Gopal Chaudhary, M. Cengiz Taplamacioglu, e M. S. Agarwal, editores, *Advances in Data Sciences, Security and Applications*, páginas 341–351, Singapore, 2020. Springer Singapore.

[16] Jason Dobies e Joshua Wood. *Kubernetes operators: Automating the container orchestration platform*. O'Reilly Media, 2020.

[17] Mario Villamizar, Oscar Garcés, Lina Ochoa, Harold Castro, Lorena Salamanca, Mauricio Verano, Rubby Casallas, Santiago Gil, Carlos Valencia, Angee Zambrano, e Mery Lang. Infrastructure cost comparison of running web applications in the cloud using aws lambda and monolithic and microservice architectures. Em *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, páginas 179–182, 2016. doi:10.1109/CCGrid.2016.37.

[18] Gigi Sayfan. *Mastering kubernetes*. Packt Publishing Ltd, 2017.

[19] Shivani Gokhale, Reetika Poosarla, Sanjeevani Tikar, Swapnali Gunjawate, Aparna Hajare, Shilpa Deshpande, Sourabh Gupta, e Kanchan Karve. Creating helm charts to ease deployment of enterprise application and its related services in kubernetes. Em *2021 International Conference on Computing, Communication and Green Engineering (CCGE)*, páginas 1–5, 2021. doi:10.1109/CCGE50943.2021.9776450.

[20] Dimitrios Ntosas. Running kafka clusters on kubernetes. Tese de mestrado, Πανεπιστήμιο Πειραιώς, 2019.

[21] Esteban Elias Romero, Carlos David Camacho, Carlos Enrique Montenegro, Óscar Esneider Acosta, Rubén González Crespo, Elvis Eduardo Gaona, e Marcelo Herrera Martínez. Integration of devops practices on a noise monitor system with circleci and terraform. *ACM Transactions on Management Information Systems (TMIS)*, 2022.

[22] Michal Orzechowski, Bartosz Balis, Krystian Pawlik, Maciej Pawlik, e Maciej Malawski. Transparent deployment of scientific workflows across clouds - kubernetes approach. Em *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, páginas 9–10, 2018. doi:10.1109/UCC-Companion.2018.00020.

[23] Yevgeniy Brikman. *Terraform: up & running: writing infrastructure as code*. O'Reilly Media, 2019.

[24] Gourav Shah. *Ansible Playbook Essentials*. Packt Publishing Ltd, 2015.

[25] Lorin Hochstein e Rene Moser. *Ansible: Up and Running: Automating configuration management and deployment the easy way*. "O'Reilly Media, Inc.", 2017.

[26] Ionut-Catalin Donca, Cosmina Corches, Ovidiu Stan, e Liviu Miclea. Autoscaled rabbitmq kubernetes cluster on single-board computers. Em *2020 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR)*, páginas 1–6, 2020. doi:10.1109/AQTR49680.2020.9129886.

[27] Sricharan Vadapalli. *DevOps: continuous delivery, integration, and deployment with DevOps: dive into the core DevOps strategies*. Packt Publishing Ltd, 2018.

[28] Yevgeniy Brikman. Why we use terraform and not chef, puppet, ansible, saltstack, or cloudformation. *Retrieved April*, 24:2020, 2016.

[29] Nitin Sukhija e Elizabeth Bautista. Towards a framework for monitoring and analyzing high performance computing environments using kubernetes and prometheus. Em *2019 IEEE SmartWorld, Ubiquitous Intelligence Computing, Advanced Trusted Computing, Scalable Computing Communications, Cloud Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*, páginas 257–262, 2019. doi:10.1109/SmartWorld-UIC-ATC-SCALCOM-IOP-SCI.2019.00087.

[30] He Huang e Liqiang Wang. P amp;p: A combined push-pull model for resource monitoring in cloud computing environment. Em *2010 IEEE 3rd International Conference on Cloud Computing*, páginas 260–267, 2010. doi:10.1109/CLOUD.2010.85.

[31] Suman Karumuri, Franco Solleza, Stan Zdonik, e Nesime Tatbul. Towards observability data management at scale. *SIGMOD Rec.*, 49(4):18–23, mar 2021. URL: https://doi.org/10.1145/3456859.3456863, doi:10.1145/3456859.3456863.

[32] Rahul Sharma e Avinash Singh. *Getting Started with Istio Service Mesh: Manage Microservices in Kubernetes*. Apress, 2020.

[33] Austin Parker, Daniel Spoonhower, Jonathan Mace, Ben Sigelman, e Rebecca Isaacs. *Distributed tracing in practice: Instrumenting, analyzing, and debugging microservices*. O'Reilly Media, 2020.

[34] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, e Jason Cong. An efficient design and implementation of lsm-tree based key-value store on open-channel ssd. Em *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, New York, NY, USA, 2014. Association for Computing Machinery. URL: https://doi.org/10.1145/2592798.2592804, doi:10.1145/2592798.2592804.

[35] Tak chung Fu. A review on time series data mining. *Engineering Applications of Artificial Intelligence*, 24(1):164–181, 2011. URL: https://www.sciencedirect.com/science/article/pii/S0952197610001727, doi:https://doi.org/10.1016/j.engappai.2010.09.007.

[36] Divyesh Bhatnagar, R Jaya SubaLakshmi, e C. Vanmathi. Twitter sentiment analysis using elasticsearch, logstash and kibana. Em *2020 International Conference on Emerging Trends in Information Technology and Engineering (ic-ETITE)*, páginas 1–5, 2020. doi:10.1109/ic-ETITE47903.2020.351.

[37] A. Srivastava. *Mastering Kibana 6.x*. Packt Publishing, 2018. URL: https://books.google.pt/books?id=Asr1twEACAAJ.

[38] G.S. Sachdeva. *Practical ELK Stack: Build Actionable Insights and Business Metrics Using the Combined Power of Elasticsearch, Logstash, and Kibana*. Apress, 2017. URL: https://books.google.pt/books?id=q30ZMQAACAAJ.

[39] E. Salituro. *Learn Grafana 7.0: A Beginner's Guide to Getting Well Versed in Analytics, Interactive Dashboards, and Monitoring*. Packt Publishing, 2020. URL: https://books.google.pt/books?id=ewmYzQEACAAJ.

[40] Janne Sirviö. Monitoring of a cloud-based it infrastructure. 2021.

[41] Brian Brazil. *Prometheus: Up & Running: Infrastructure and Application Performance Monitoring*. "O'Reilly Media, Inc.", 2018.

[42] John Arundel e Justin Domingus. *Cloud Native DevOps with Kubernetes: building, deploying, and scaling modern applications in the Cloud*. O'Reilly Media, 2019.

[43] AMEENAH KHALID ABUSINNAIN, THEGA ALAMEEN MOKHTAR, MARSA ABU-TALIB YOUSEF, e MAJEDA TAJELSIR MERGHANY. Web-based application performance monitoring tool. 2017.

[44] Grafana loki. URL: https://grafana.com/oss/loki/.

[45] Ach Izalul Haq e Banu Santoso. Analisis perbandingan performa metode elk stack dan grafana loki pada honeypot server. *Jurnal Sisfokom (Sistem Informasi dan Komputer)*, 10(3):376–385, 2021.

[46] Promtail. URL: https://grafana.com/docs/loki/latest/clients/promtail/#:~:text=Promtail%20is%20an%20agent%20which,Attaches%20labels%20to%20log%20streams.

[47] Kenneth Hitchcock. Logging. Em *Linux System Administration for the 2020s*, páginas 241–257. Springer, 2022.

[48] Ankit Anand. Jaeger vs zipkin - key architecture components, differences and alternatives, 3 2022. URL: https://signoz.io/blog/jaeger-vs-zipkin/.

[49] S Mallanna e M Devika. Distributed request tracing using zipkin and spring boot sleuth. *Int. J. Comput. Appl*, 975:8887, 2020.

[50] Aws x-ray. URL: https://aws.amazon.com/pt/xray/.

[51] Rui Deng. Benchmarking of serverless application performance across cloud providers: An in-depth understanding of reasons for differences. 2022.

[52] Tlantic mobile retail. URL: https://www.tlantic.com/pt/mobile-retail.

[53] TlanticSI. Building high availability architecture in cloud.

[54] David Dossot. *RabbitMQ essentials*. Packt Publishing Ltd, 2014.

[55] Rahul Soni. *Nginx*. Springer, 2016.

[56] What is prometheus? URL: https://prometheus.io/docs/introduction/overview/.

[57] Mainak Chakraborty e Ajit Pratap Kundan. Grafana. Em *Monitoring Cloud-Native Applications*, páginas 187–240. Springer, 2021.

[58] Grafana loki's architecture. URL: https://grafana.com/docs/loki/latest/fundamentals/architecture/.

[59] Tlantic mobile retail. URL: https://www.tlantic.com/pt/mobile-retail.

[60] Architecture. URL: https://www.jaegertracing.io/docs/1.23/architecture/#:~:text=The%20Jaeger%20agent%20is%20a,collectors%20away%20from%20the%20client.