Faculdade de Ciências e Tecnologias
Departamento de Engenharia Informática

Plataforma Escalável de Monitorização para Ambientes Cloud

Rui Filipe Rama e Silva

Dissertação no contexto do Mestrado em Engenharia Informática, Especialização em Comunicações, Serviços e Infraestruturas orientada por Prof. Dr. Nuno Antunes e apresentada à
Faculdade de Ciências e Tecnologias / Departamento de Engenharia Informática.

Janeiro de 2019

Esta página foi propositadamente deixada em branco.

This work is within the software engineering specialization area and was carried out in the Software and Systems Engineering (SSE) Group of the Centre for Informatics and Systems of the University of Coimbra (CISUC).

This work was partially supported by the project ATMOSPHERE, funded by the Brazilian Ministry of Science, Technology and Innovation (51119 - MCTI/RNP 4th Coordinated Call) and by the European Commission under the Cooperation Programme, H2020 grant agreement no 777154.

It is also partially supported by the project METRICS (POCI-01-0145-FEDER-032504), co-funded by the Portuguese Foundation for Science and Technology (FCT) and by the *Fundo Europeu de Desenvolvimento Regional* (FEDER) through *Portugal 2020 - Programa Operacional Competitividade e Internacionalização (POCI)*.

This work has been supervised by Professor Nuno Manuel dos Santos Antunes, Assistant Professor at the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

Esta página foi propositadamente deixada em branco.

Agradecimentos

A realização desta tese de mestrado contou com apoios e incentivos importantes que sem eles não se teria tornado realidade e aos quais estarei para sempre grato.

Ao Prof. Nuno Antunes, pela sua orientação, pela transmissão de conhecimento, pelas opiniões e críticas, colaboração total no solucionar de dúvidas e problemas que surgiram ao longo da realização deste estágio e pelo incentivo incondicional.

Aos membros do júri, o Prof. Carlos Bento e o Prof. Mario Rela, por todo o *feedback* dado na defesa intermédia que foi extremamente útil na orientação do trabalho para este segundo semestre e também pelas indicações dadas para melhorar este documento.

Ao José D'Abruzzo Pereira, com quem eu trabalhei mais de perto no contexto do projeto `ATMOSPHERE`, por toda a paciência e por todas as indicações que contribuíram para o melhoramento da plataforma desenvolvida.

Ao Nuno Cardoso, José Flora, João Campos, Cristiano Lemes, Charles Gonçalves, Inês Valentim e Matheus Torquato por toda a ajuda prestada e pelos bons momentos passados.

Ao David Canoso, por toda a ajuda na revisão deste documento.

Por último, agradeço especialmente aos meus pais, Vasco Cardoso Silva e Maria de Fátima Rama Lopes Azedo, por todo o incentivo e coragem, pelo seu apoio incondicional, amizade e paciência demonstrados ao longo desta caminhada.

A todos que referi anteriormente dedico este trabalho!

Esta página foi propositadamente deixada em branco.

Abstract

Cloud is an emerging technology and already widely used in almost every type of system. This technology can use containers to make the management of services and computational resources of Cloud easier. Due to its adoption in business-critical scenarios, problems related to the reliability, security and privacy in Cloud arise. In such dynamic systems, it becomes necessary to have continuous monitoring, assessment and improvement, but currently there are no efficient solutions for this effect for environments as dynamic as the Cloud. ATMOSPHERE is an international project with the objective of monitoring and improving the trustworthiness of Cloud applications.

The goal of this work is the design and implementation of a scalable monitoring platform that is part of `ATMOSPHERE` project. This platform consists of three key components: `TMA_Monitor`, which is responsible for checking the format of the data collected; a `FaultTolerantQueue`, which is responsible for forwarding the data; and `TMA_Knowledge`, which stores all valid information collected. In order to be easy to deploy and scale, this platform is designed to be deployed in containers. To find the best technologies to use in the implementation of this platform, we analyzed the most popular tools for container building and management, for fault tolerant message queuing and forwarding and for Python frameworks for API REST applications. After the platform development, in order to validate this platform, structural tests, performance tests and scalability tests were performed. The results of the tests showed that platform developed works well even under stress conditions.

Keywords

Cloud, containers, Docker, Kubernetes, scalability, monitoring.

Esta página foi propositadamente deixada em branco.

Resumo

A *Cloud* é uma tecnologia emergente e já amplamente usada em quase todo o tipo de sistemas. Esta tecnologia pode usar *containers* para fazer a gestão dos serviços e recursos computacionais da *Cloud* mais fácil. Devido à sua adopção em cenários críticos para o negócio, problemas relacionados com a confiabilidade, segurança e privacidade na *Cloud* aumentam. Em sistemas dinâmicos como a *Cloud*, torna-se necessário que estes sejam monitorizados, avaliados e melhorados continuamente, mas, atualmente, não existem soluções eficientes para este efeito para ambientes tão dinâmicos quanto a *Cloud*. O ATMOSPHERE é um projeto internacional com o objetivo de monitorizar e melhorar a confiança das applicações em *Cloud*

O objetivo deste trabalho é o desenho e implementação de uma plataforma de monitorização escalável, que é parte do projeto `ATMOSPHERE`. Esta plataforma, consistem e três componentes: o `TMA_Monitor`, que é responsável pela verificação do formato dos dados recolhidos; `FaultTolerantQueue`, que é responsável por reencaminhar dados; e o `TMA_Knowledge`, que armazena toda a informação válida recolhida. De maneira a ser facilmente implmentável e escalável, esta plataforma é desenhada para ser implementada em *containers*. Para encontrar as melhores tecnologias para usar na implementação desta plataforma, foram analisadas as ferramentas mais populares para virtualizadores e gestores de *containers*, para a gestão e encaminhamento de mensagens e para aplicações API REST implementadas em *Python*. Depois da implementação, para validar a plataforma, testes estruturais e testes de desempenho e escalabilidade foram executados. Os resultados mostram que a plataforma desenvolvida funciona bem mesmo sob condições de stress.

Palavras-Chave

Cloud, containers, Docker, Kubernetes, escalabilidade, monitorização.

Esta página foi propositadamente deixada em branco.

Conteúdo

1 Introdução **1**
1.1 Objectivos e Visão Geral da Abordagem . . . 2
1.2 Estrutura do Documento . . . 3

2 Conceitos Teóricos e Estado da Arte **5**
2.1 Ferramentas de Monitorização . . . 5
2.1.1 Ferramentas de Monitorização Genéricas . . . 5
2.1.2 Ferramentas de Monitorização de Infraestrutura . . . 10
2.1.3 Ferramentas de Monitorização de *Containers* . . . 12
2.1.4 Resumo . . . 16
2.2 Microserviços . . . 17
2.2.1 Implementação de microservicos . . . 17
2.2.2 Escalabilidade de microserviços . . . 19
2.3 Ambientes de *Containers* . . . 20
2.3.1 Virtualizadores de *containers* . . . 20
2.3.2 Gestores de *containers* . . . 24
2.4 Processadores de Fluxo de Mensagens . . . 29
2.4.1 Apache Kafka . . . 30
2.4.2 ZeroMQ . . . 31
2.4.3 RabbitMQ . . . 31
2.4.4 Nats.io . . . 32
2.4.5 Resumo . . . 33
2.5 Flafka . . . 34
2.6 Armazenamento Persistente de Dados . . . 35

3 Análise de Requisitos **39**

4 Plataforma Escalável de Monitorização **41**
4.1 Arquitetura Geral da Plataforma . . . 41
4.2 Arquitetura Detalhada da Plataforma . . . 42
4.2.1 Detalhes de Implementação . . . 43

5 Validação e Experimentação **53**
5.1 Plano de Validação da Plataforma . . . 53
5.2 Testes Estruturais . . . 53
5.2.1 Critérios de Cobertura . . . 54
5.2.2 Grafos de *control-flow* . . . 54
5.2.3 Casos de teste . . . 64
5.2.4 Execução dos testes e análise dos resultados . . . 68
5.3 Validação de requisitos não funcionais . . . 69
5.3.1 Ambiente Experimental . . . 69
5.3.2 Metodologia Experimental . . . 69

5.3.3 Análise de Resultados 72

6 Planeamento **79**
6.1 Metodologia de Desenvolvimento 79
6.2 Esforço primeiro semestre 79
6.3 Esforço do Segundo Semestre 80

7 Conclusões e Trabalho Futuro **83**

Acrónimos

API Application Programming Interface. xviii, 2, 7, 17–19, 22, 24, 25, 27, 35, 45, 50, 51

CPU Central Processing Unit. 10, 22, 27, 28, 50, 56, 69, 71, 75

DBMS Database Management System. 43, 47

DDoS Distributed Denial-of-Service Attack. 20

DLL Dynamic-link library. 49, 50

DNS Domain Name Server. 27

ETL extract,transform,load. 41, 46

FIFO *First In First Out*. 24

HDFS Hadoop Distributed File System. 35

HPA Horizontal Pod Autoscaler. 71

HTTP Hypertext Transfer Protocol. 6, 7, 11, 18, 32, 41, 43, 44, 55, 68, 69, 71

HTTPS Hyper Text Transfer Protocol Secure. 44, 49

ICMP Internet Control Message Protocol. 10, 11

IP Internet Protocol. 27, 44, 48

JDBC Java Database Connectivity. 46

JSON Javascript Object. 26, 41, 43, 55, 64

KPI Key Performance Indicator. 12–14, 39

LXC Linux Containers. 20, 22, 24

RAM Random Access Memory. 69

REST Representational state transfer. xviii, 2, 17, 19, 22, 24

RSS Ritch Site Summary. 18

SLA Service Level Agreement. 2

SNMP Simple Network Management Protocol. 10, 11

SQL Standard Query Language. 46–48, 58

TCP Transmission Control Protocol. 10

TMA Trustworthiness Monitoring and Assessment Platform. 41, 45, 48, 81

WSGI Web Server Gateway Interface. 44

XML eXtensible Markup Language. 10

Esta página foi propositadamente deixada em branco.

Lista de Figuras

2.1 Arquitetura Prometheus (fonte [42]). 6
2.2 Arquitetura Graphite (fonte [14]). 8
2.3 Arquitetura Hyperic. 9
2.4 Arquitetura Zabbix (fonte [29]). 11
2.5 Arquitetura Nagios (fonte [25]). 11
2.6 Arquitetura CloudHealth (fonte [48]). 12
2.7 Arquitetura LOUD (fonte [35]). 14
2.8 Arquitetura TICK Stack (fonte [27]). 15
2.9 Arquitetura cAdvisor. 16
2.10 Arquitetura Docker (fonte [5]). 21
2.11 Arquitetura LXC (fonte [46]). 22
2.12 Funcionamento do CoreOS Rocket (fonte [12]). 23
2.13 Arquitetura Docker Swarm (fonte [38]). 25
2.14 Arquitetura Kubernetes (fonte [37]). 26
2.15 Arquitetura Apache Mesos (fonte [4]). 28
2.16 Arquitetura Apache Kafka (fonte [54]). 30
2.17 Exemplos de sockets do ZeroMQ (fonte [33]). 31
2.18 Arquitetura RabbitMQ (fonte [30]). 32
2.19 Arquitetura Nats.io (fonte [41]). 33
2.20 Arquitetura Apache Flume (fonte [53]). 34

4.1 Arquitetura geral da plataforma a desenvolver. 42
4.2 Arquitetura detalhada da solução. 43

5.1 Grafo *control-flow* do método *process_message*. 55
5.2 Grafo *control-flow* do método *validate_schema*. 56
5.3 Grafo *control-flow* do método *main* da *probe_python_demo*. 57
5.4 Grafo *control-flow* do método *get_container_stats*. 58
5.5 Grafo *control-flow* do método *format*. 59
5.6 Grafo *control-flow* do método *main* da *probe_k8s_network*. 60
5.7 Grafo *control-flow* do método *main* da *probe_cs_demo*. 61
5.8 Grafo *control-flow* do método *parse_events* do modo normal. 62
5.9 Grafo *control-flow* do método *parse_events* modo de teste. 63
5.10 Gráfico com *throughput* e número de réplicas para testes com dois minutos . 76
5.11 Gráfico com *throughput* e número de réplicas para testes com um minuto . . 76

6.1 Fases da metodologia Waterfall (fonte [16]). 79
6.2 Diagrama de Gantt com as tarefas reais do primeiro semestre. 80
6.3 Diagrama de Gantt com as tarefas do segundo semestre 81

Esta página foi propositadamente deixada em branco.

Lista de Tabelas

2.1 Tablela comparativa das microframeworks para Representational state transfer (REST) Application Programming Interface (API) 19
2.2 Tablela comparativa dos construtores de *containers* 24
2.3 Tablela comparativa dos gestores de *containers* estudados 29
2.4 Tablela comparativa dos processadores de fluxos de mensagens estudados . . 34

5.1 Casos de teste para o método *process_message* 64
5.2 Casos de teste para o método *validate_schema* 65
5.3 Casos de teste para o método *main* da *probe_python_demo* 65
5.4 Casos de teste para o método *get_stats* da *probe_k8s_docker* 65
5.5 Casos de teste para o método *format* da *probe_k8s_docker* 66
5.6 Casos de teste para o método *main* da *probe_k8s_network* 66
5.7 Casos de teste para o método *main* da *probe_cs_demo* 66
5.8 Casos de teste para o método *parse_events* do modo normal da plataforma 67
5.9 Casos de teste para o método *parse_events* do modo de teste da plataforma 67
5.10 Perfis dos testes utilizados na validação de desempenho e escalabilidade da plataforma. 70
5.11 Perfis dos testes da primeira bateria de testes utilizados na validação da elasticidade da plataforma. 71
5.12 Perfis dos testes da segunda bateria de testes utilizados na validação da elasticidade da plataforma. 71
5.13 Resultados da validação de desempenho da plataforma 72
5.14 Resultados da validação de escalabilidade da plataforma 74

Esta página foi propositadamente deixada em branco.

Lista de Publicações

Os resultados deste trabalho contribuiram para um *deliverable* do projeto `ATMOSPHERE`

- J. Pereira, **Rui Silva**, N. Antunes, B. França (UNICAMP). "Monitoring Instruments and Platform Design", ATMOSPHERE Deliverable 3.2.
  - ***Abstract:*** *This deliverable describes the design of the monitoring platform framework and of the instruments needed to collect the data for the calculation of the trustworthiness score. Moreover, it will define the interfaces for the integration with the trustworthiness assessment facilities of the different layers of ATMOSPHERE.*

Esta dissertação também contribuiu para publicações que estão atualmente submetidas para revisão em conferências:

- J. Pereira, **Rui Silva**, N. Antunes, M. Vieira, B. França (UNICAMP), J. Luiz (UNICAMP). "Artifact for Self-Adaptive Cloud Applications using Trustworthiness Properties", *14th Symposium on Software Engineering for Adaptive and Self-Managing Systems 2019 (SEAMS 2019)*, Montréal, Canada, Maio 25-26, 2019. ***(em revisão)***
  - ***Abstract:*** *Self-adaptive Systems (SaSs) are capable of reflecting on both their own state and the environment, and change their behavior to satisfy the expected objectives. Satisfying properties such as self-configuring, self-optimizing, self-protecting and self-healing is usually the goal of such adaptations. Cloud systems need to be self-adaptive by nature, especially considering that resources are used in a pay-as-you-go manner. Thus, more allocated resources represent a higher cost and leaving resources idle is wasting the customer budget. However, satisfying other trustworthiness properties also demands self-adaptation capabilities, but developers lack an easy-to-use platform to support the assessment of such properties and to execute the required adaptions. This paper presents the TMA-Platform, an artifact that implements a MAPE-K control loop for cloud systems. The platform implements a distributed monitoring system based on probes, which is highly flexible and supports the analysis. The analysis is based on quality models, which aggregates different observations to calculate scores. These scores are later used to plan adaptations by evaluating rules, which are supported by a business rule management system. TMA is a container-based framework that is deployed using Kubernetes, a container orchestrator framework. Although not restricted to the cloud, the platform is more suitable to them due to its architecture.*

Esta página foi propositadamente deixada em branco.

Capítulo 1

Introdução

A *Cloud* é um conjunto de recursos partilhados com alta flexibilidade de configuração alojados na Internet que podem ser utilizados pelo mais variado tipo de utilizador sem preocupações de gestão[32]. Tendo em conta toda a flexibilidade que está inerente ao conceito de *Cloud*, torna-se necessário que os recursos que a compõe possam escalar para que o desempenho não sofra alterações. A escalabilidade neste tipo de ambientes levanta grandes desafios no ponto de vista do *hardware* e da manutenção da infraestrutura. Para resolver estes desafios, foi introduzido o conceito de virtualização.

Nos últimos anos, a *Cloud* tem tido um desenvolvimento bastante acelerado e, o que fez com que haja uma necessidade cada vez maior de existir recursos escaláveis e elásticos[55]. Sem a virtualização, as empresas prestadoras deste serviço teriam de gastar enormes quantidades de fundos em *hardware*, o que implicaria uma gestão de recursos muito mais precisa com custos monetários elevados.

No entanto, no contexto da virtualização, surgiu o conceito de *containers* que fez com que, por um lado, não fosse necessário investir fundos em *hardware* redundante e, por outro lado, veio facilitar bastante a gestão de recursos computacionais[26]. Por consequência, as ferramentas que fornecem uma plataforma baseada em *containers* subiram bastante em popularidade nos tempos mais recentes[31].

O conceito de *container*, apesar de já existir há vários anos, só recentemente é que se tornou popular, porque ferramentas como o *Docker* conseguiram abstrair o utilizador da complexidade de gestão e de execução de *containers*. *Containers* são uma unidade única de *software* que aglomera o código e as dependências de uma aplicação de maneira a que esta seja executada facil e eficazmente [20]. Estando a aplicação a ser executada dentro de um *container*, esta irá ser executada da mesma maneira, independentemente da infraestrutura que a hospeda. As aplicações que são executadas em *containers* são isoladas umas das outras, sendo para cada uma reservada uma quota de recursos computacionais.

Estas características dos *containers* vieram possibilitar o aparecimento de microserviços, que são pequenos serviços independentes uns dos outros que compõem uma determinada aplicação. Como uma aplicação pode ser constituída por vários microserviços, isto torna a gestão do funcionamento dos mesmos difícil de gerir.

Para lidar com a gestão deste tipo de serviços, existem os gestores de *containers*. Estas ferramentas conseguem gerir todos os *containers* que estão a executar num determinado *cluster*, permitindo a escalabilidade e disponibilidade dos serviços prestados pelos mesmos. As ferramentas mais populares neste contexto são o *Kubernetes* e o *Docker Swarm*.

Fazendo uso desta tecnologia dos *containers*, a *Cloud* passou a ser aplicada em vários campos da sociedade, chegando mesmo a armazenar informação crítica de negócios de extrema importância. Além disso, grandes empresas de comércio, nomeadamente, comércio eletrónico, têm os seus serviços a serem executados em servidores que estão alojados na *Cloud*. Portanto, a probabilidade de aparecimento do mais variado tipo de falhas é alta, o que faz com que a constante verificação dos serviços se torne um aspeto muito importante por várias razões, como, por exemplo, gestão de Service Level Agreement (SLA), faturação, resolução de problemas, gestão de desempenho e segurança [1].

Para auxiliar na mitigação da ocorrência de falhas e na manutenção do normal funcionamento dos serviços, a existência de mecanismos de monitorização eficaz torna-se um requisito de alta prioridade de forma a detetar a possível causa da ocorrências de falhas e também para avaliar a confiança do sistema.

Atualmente, existem várias ferramentas de monitorização para ambientes de *Cloud*, mas não são flexíveis ao ponto de monitorizar várias tecnologias. Um exemplo de uma ferramenta que se enquadra nestas necessidades é o `ATMOSPHERE`, que é constituída por micro-serviços que fazem, entre outras funções, uma monitorização eficaz dos sistemas, independentemente das tecnologias envolvidas, e fornece aos seus utilizadores um coeficiente de confiança em relação ao sistema alvo de monitorização. Para conseguir uma monitorização eficaz de um ambiente *Cloud* que faça de uso de diversos serviços que utilizem várias tecnologias, torna-se, então, necessário uma ferramenta que consiga ser flexível o suficiente para monitorizar todas as tecnologias para todos os níveis de granularidade.

1.1 Objectivos e Visão Geral da Abordagem

O principal objetivo deste trabalho é o desenho e implementação de uma plataforma de monitorização de serviços que estejam alojados num ambiente *Cloud*. Para que a monitorização seja eficiente neste tipo de ambientes, esta plataforma necessita de ser flexível e escalável.

Os componentes desenvolvidos são uma parte chave não só da contribuição da Universidade de Coimbra para o projecto *ATMOSPHERE*, mas também do funcionamento do projecto como um todo. Assim, a concepção destes componentes exigiu especial atenção para permitir a integração com os restantes parceiros do projecto. Apesar dos trabalhos ainda continuarem, o componentes já se encontram a ser usados e a demonstrar a sua flexibilidade no seio do projecto. Foram também demonstrados com sucesso durante a avaliação intermédia do projecto.

Para que isto seja possível, a plataforma é composta por três componentes importantes que são o `TMA_Monitor`, o `FaullTolerantQueue` e o `TMA_Knowledge`.

O `TMA_Monitor` fornece uma API REST que é responsável pela receção, validação e envio dos dados recebidos para o segundo componente, `FaultTolerantQueue`. Esta file `queue` é responsável pelo emcaminhamento da informação entre todos os componentes, não só da plataforma apresentada neste documento, mas também de outros componentes que fazem parte do projeto `ATMOSPHERE`. Para armazenar os dados, esta plataforma tem um componente chamado `TMA_Knowledge`, que é responsável por armazenar todos os dados validados pelo `TMA_Monitor`.

Esta plataforma é ainda constituída por pequenas aplicações de *software* chamadas *probes*, que são instaladas nos serviços a monitorizar e que são responsáveis pela recolha dos dados e posterior envio para o *endpoint* do `TMA_Monitor`.

Os componentes `TMA_Monitor`, o `FaullTolerantQueue` e o `TMA_Knowledge` foram implementados sob a forma de microserviços. Por esta razão, todos os componentes desta plataforma vão ser executados em *containers*. Para gerir todos estes componentes, foi usado um gestor de *containers* de modo a que seja mais fácil gerir aspetos relacionados com a escalabilidade e disponibilidade de todos os componentes desta plataforma.

1.2 Estrutura do Documento

O resto do documento está organizado da seguinte forma:

- **Capítulo 2 – Conceitos Teóricos e Estado da Arte**. Descreve com detalhe conceitos como monitorização, microserviços e balanceamento de carga. Neste capítulo, também são apresentadas as soluções existentes para monitorização de *containers*, assim como a descrição e comparação de todos os tipos de tecnologias que foram estudadas para o desenvolvimento de todos os componentes da plataforma.

- **Capítulo 3 – Análise de Requisitos**. Descreve os requisitos funcionais, requisitos não funcionais e algumas restrições que se tiveram de ter em conta na escolha de algumas tecnologias.

- **Capítulo 4 – Solução Proposta**. Começa por apresentar e descrever a arquitetura alto nível da solução implementada, a seleção das tecnologias que mais se adequaram consoante as necessidades existentes e, depois, é apresentada uma arquitetura mais detalhada da aplicação. Neste capítulo é também apresentado pormenorizadamente todo o desenvolvimento de cada um dos componentes, nomeadamente, o `TMA_Monitor`, o `FaullTolerantQueue`, o `TMA_Knowledge` e ainda descritas algumas *probes* que foram desenvolvidas para validar o funcionamento de todos os componentes.

- **Capítulo 5 – Validação da Solução**. Explica todo o processo de validação da plataforma, começando pelo plano de validação, onde é descrito o processo de validação quer dos requisitos funcionais, quer dos requisitos não funcionais. Em relação aos requisitos funcionais, são apresentados os critérios de cobertura usados; em seguida, os gráficos de *control-flow* dos métodos mais complexos que fazem parte de todos os componentes; depois, são apresentados os casos de teste usados para cada um desses métodos, tendo por base os grafos de *control-flow*; finalmente, é descrito o processo de execução dos casos de teste e os respetivos resultados. Na secção dos requisitos não funcionais, são descritas as características do ambiente de testes, é apresentada a metodologia de teste e, por fim, é apresentada os resutados e análise dos testes executados.

- **Capítulo 6 - Planeamento**. Contém a descrição da metodologia de desenvolvimento usada neste projeto e também todas as tarefas planeadas para este estágio.

- **Capítulo 7 – Conclusões e Trabalho Futuro**. Contém as principais conclusões e o trabalho futuro. A conclusão é uma reflexão sobre tudo o que aconteceu durante este projeto. O trabalho futuro apresenta as atividades planeadas que não foram concluídas durante este ano, bem como outras funcionalidades para o melhoramento da plataforma.

Esta página foi propositadamente deixada em branco.

Capítulo 2

Conceitos Teóricos e Estado da Arte

A *Cloud* veio introduzir a ideia de que a computação é recurso que apenas se utiliza quando é necessário, fornecendo serviços que podem ser geridos a partir de qualquer lado desde que haja uma ligação à internet. Este aspeto torna-se ainda mais evidente quando se analisa o setor industrial. As empresas investem cada vez menos em recursos computacionais e em mão de obra humana para os gerir, pois optam por ter todos os serviços que necessitam a nível de tecnologia de informação alojados *Cloud*. Todos estes factos fazem com que a utilização da *Cloud* aumentasse exponencialmente nos últimos anos. No entanto, a *Cloud* é composta por sistemas computacionais de grande dimensão e complexos, o que faz com que a monitorização seja um requisito crítico neste contexto [50].

Nesta secção, irá ser apresentada uma comparação entre todas as ferramentas experimentadas durante este projeto, sendo que essa comparação está dividida de acordo com as funções das ferramentas em questão, bem como alguns conceitos importantes no âmbito deste estágio.

2.1 Ferramentas de Monitorização

A monitorização em *Cloud* define-se como o processo que compreende a gestão, monitorização e avaliação de seviços ou aplicações que estajam alojados nesta tecnologia, mas também a sua própria infraestrutura [2].

Com o objetivo de manter os serviços *Cloud* a funcionar corretamente e resolver rapidamente perturbações que possam ocorrer na sua infraestrutura, existem várias ferramentas monitorizam serviços, aplicações e a infraestrutura da *Cloud*. Nas próximas subsecções vão ser descritas e analisadas não só as ferramentas mais usadas, como também alguns sistemas que são propostos na literatura.

2.1.1 Ferramentas de Monitorização Genéricas

Prometheus

Prometheus é uma ferramenta de monitorização de sistemas e serviços. O Prometheus recolhe métricas a partir dos alvos de monitorização que são configurados para o efeito em intervalos de tempo configuráveis e mostra os valores numa *dashboard*.

O Prometheus ainda possui outras características que outras soluções existentes para o

mesmo propósito não possuem que são:

- A dimensão temporal dos resultados pode ser definida para cada métrica;
- Uma linguagem de queries flexível para suportar a funcionalidade anterior;
- Os servidores de recolha de métricas são autónomos, não existem problemas de dependências;
- Os sistemas a monitorizar são descobertos dinâmicamente ou de configuração estática;
- Permite fazer gráficos de métricas recolhidas de diferentes alvos de monitorização;
- Permite a recolha de métricas armazenadas noutro servidor Prometheus (*federation*).

A arquitetura do Prometheus é ilustrada pela Figura 2.1.

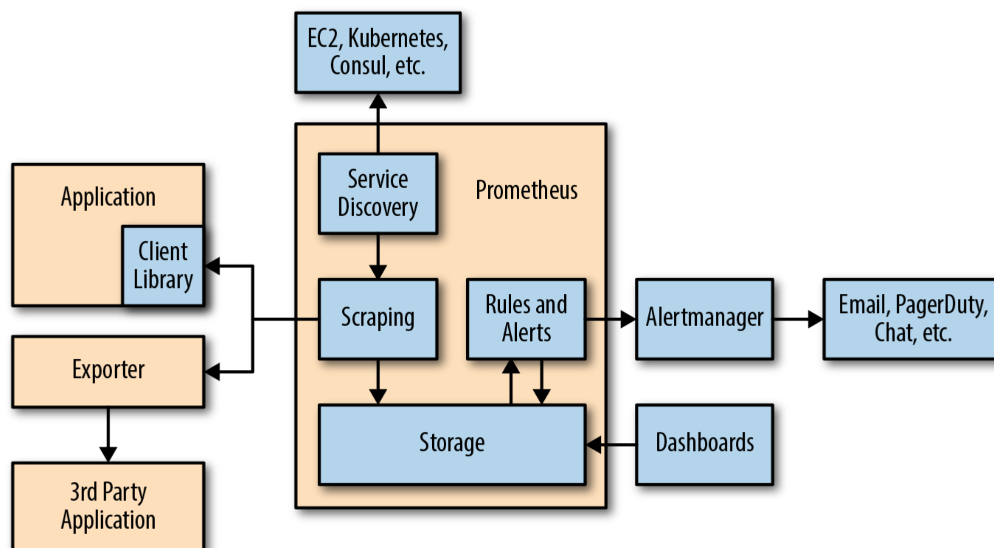

Figura 2.1: Arquitetura Prometheus (fonte [42]).

O Prometheus descobre alvos de monitorização para recolher as métricas através de um serviço de descoberta. Os alvos podem ser configurados diretamente pelo utilizador ou podem estar a ser executados em plataformas externas em que aí, para recolher as métricas, torna-se necessário utilizar uma pequena aplicação para a recolha de métricas. Essa aplicação chama-se *exporter*.

Os dados recolhidos são armazenados e podem ser usados numa *dashboard* usando o *PromQL* ou, com base, esses dados podem ser enviados para o *AlertManager* que os converte em notificações que podem ser apresentadas sob a forma de *pop-ups* ou enviadas através de um serviço de email.

O processo de recolha de métricas, como podemos ver pela Figura 2.1, pode ser feito de várias maneiras. Uma delas são as bibliotecas, que não são mais poucas linhas de código que definem as métricas a recolher, e implicitamente convertem os dados recolhidos para o formato de um Hypertext Transfer Protocol (HTTP) *requests* e enviam para o servidor do Prometheus.

Esta adição de código implica um aumento de memória consoante o número de métricas a recolher. A este processo chama-se recolha direta. Estas bibliotecas estão disponíveis em todas as principais linguagens de programação. Existem bibliotecas oficiais do Prometheus para *Go*, *Python*, *Java* e *Ruby*, mas também já existem bibliotecas desenvolvidas por outras entidades para *C#*, *Node.js*, *Haskell*, *Erlang* e *Rust*.

Por outro lado, também é possível monitorizar aplicações através de pequenas aplicações chamadas *exporters*. Os *exporters* são inicializados no mesmo ambiente em que está a executar a sistema a monitorizar.

Os *exporters* recolhem as métricas, formata-as e envia-as sob a forma de HTTP *request* para o Prometheus. A este método de recolha de métricas dá-se o nome de *custom collectors*.

A componente *Service Discovery* é responsável por descobrir os alvos a monitorizar de uma forma dinâmica. O Prometheus suporta vários tipos de plataformas onde esses alvos poderão estar a ser executados, desde logo, *Chef's database*, ficheiros *Ansible*, instâncias EC2 e *pods* no *Kubernetes*.

O módulo *Scraping* é responsável por enviar pedidos aos agentes instalados nos alvos que estão a ser monitorizados para que estes retornem as métricas que recolheram. O Prometheus faz isto através de um HTTP *request*.

A resposta por parte dos agentes instalados nos alvos é processada e armazenada no componente *Storage*. O módulo *Scraping* adiciona ainda alguma informação como por exemplo, se o processamento dos pacotes HTTP provindos dos agentes foi bem sucedido ou não. Todo este processo de recolha e processamento de métricas repete-se ao longo do tempo, sendo configurável um intervalo de tempo entre cada repetição desde 10 até 60 segundos.

O componente *Strorage* é responsável pelo armazenamento de toda as métricas recolhidas de uma forma centralizada. O mecanismo de armazenamento do Prometheus consegue lidar com milhões de exemplos por segundo, tornando possível monitorizar milhares de máquinas com apenas uma instância do Prometheus.

O Prometheus tem inúmeras Web Application Programming Interface (API) que permitem ao administrador não só, ver a informação em bruto, mas também fazer consultas mais complexas a essa mesma informação. *Recording Rules* permitem executar várias operações matemáticas com base nas métricas recolhidas e guardá-las de modo a que o resultado dessa expressão matemática apareça na *dashboard*.

Alerting Rules permitem a definição de alertas baseados na linguagem do Prometheus e permite também enviar as notificações desses alertas para um outro serviço externo.

O *Alertmanager* tem como função receber os alertas vindos do Prometheus e transformá-los em notificações. As notificações podem ser num formato de email, aplicações de chat como o *Slack* e serviços externos como o *PagerDuty*.

Para não sobrecarregar os serviços para os quais as notificações são enviadas, este componente agrega alertas que estejam de alguma maneira relacionados. Pode-se configurar diferentes formatos de notificações consoante a razão pela qual o alerta foi gerado. Adicionalmente, a geração de alertas pode também ser silenciada.

Graphite

Graphite é uma ferramenta de monitorização que tem um bom desempenho tanto em *hardware* barato como num ambiente na *Cloud*.

Graphite monitoriza o desempenho de sites web, aplicações, serviços e equipamentos de rede. A arquitetura do Graphite otimiza o armazenamento, partilha e visualização dos dados recolhidos.

O Graphite foca-se em duas funções:

- Armazenamento dos dados recolhidos;
- Visualização dos dados graficamente em tempo-real.

O Graphite é escrito em Python e é constituído por três componentes principais:

- Um *driver* de base de dados chamado *whisper*;
- Um *daemon* chamado *carbon*;
- Uma aplicação web que é responsável pela geração dos gráficos e pelo fornecimento de uma interface gráfica para o utilizador.

A arquitetura do Graphite é ilustrada na Figura 2.2:

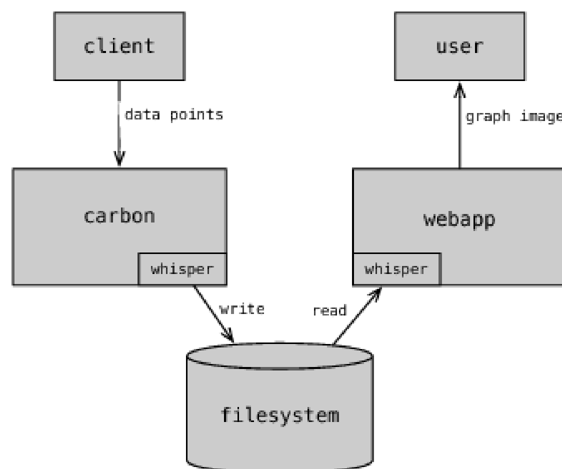

Figura 2.2: Arquitetura Graphite (fonte [14]).

Um *whisper* é um *driver* de base de dados usado por aplicações para consultar e manipular dados armazenados numa base de dados. As operações básicas que este *driver* são criar um *whisper*, atualizar e consultar os dados armazenados na base de dados.

O componente *carbon* foi desenvolvido em *Twisted*, uma *framework* para I/O em Python. Esta *framework* permite com que este componente consiga comunicar com um número alargado de clientes e lidar com uma quantidade significativa de tráfego de rede.

As aplicações chamadas *Client* é responsável pela recolha dos dados e enviá-los para o *carbon*, que, depois vai armazenar os dados usando o *whisper*. Esses dados, depois vão ser consultados pela aplicação web para gerar os gráficos.

Hyperic

VMWARE VREALIZE HYPERIC é uma ferramenta de monitorização do desempenho e disponibilidade de um conjunto alargado dos sistemas operativos ou aplicações , quer estejam a executar numa interface física, quer em ambientes *Cloud* [23].

O *Hyperic* é constituído por um servidor, por um agente, por uma base de dados e por uma interface gráfica como ilustra a Figura 2.3 [23].

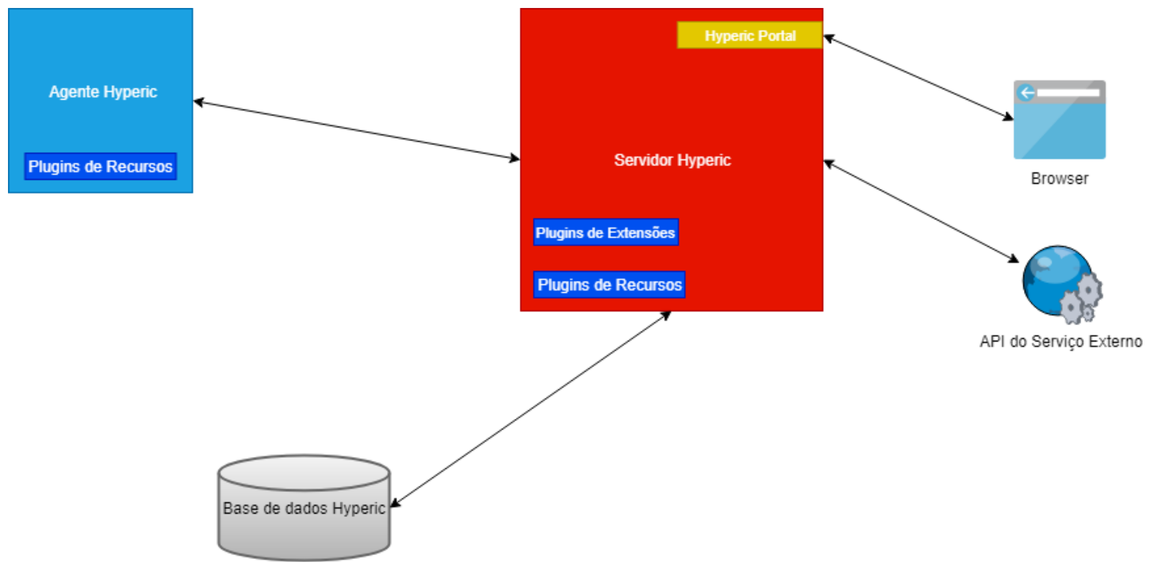

Figura 2.3: Arquitetura Hyperic.

O agente deve ser executado no sistema a monitorizar, quer este seja uma máquina física ou uma máquina virtual.

Estes agentes procuram automaticamente e periodicamente os componentes de *software* a executar nas máquinas que estão a ser monitorizadas. As métricas recolhidas referem-se ao desempenho e disponibilidade.

Como funcionalidades adicionais, este agente é capaz de desligar ou iniciar serviços ou máquinas. Depois de recolher as métricas, o agente envia os dados para o servidor.

O servidor é responsável por receber todos os valores das métricas provindos do agente e armazená-los na base de dados. Este componente agrupa as métricas relacionadas para que seja mais fácil para o administrador de rede detetar possíveis perturbações no sistema. O servidor também é responsável por disparar eventos e gerar as respetivas notificações.

Por fim, o servidor também é responsável por processar ações, devidamente autenticadas, tomadas pelo administrador de rede através da interface gráfica desta ferramenta.

Por sua vez, a interface gráfica do *Hyperic* é uma página web em que se pode visualizar várias informações sobre os sistemas que estão a ser monitorizados.

A página principal desta interface é a *dashboard* que depois tem ligações para outras páginas como o histórico de alertas ou disponibilidade dos recursos monitorizados. No entanto, também é possível a configuração de novas páginas ou remoção de páginas existentes.

Dentro da *dashboard* existem as páginas:

- Recursos - onde podem ser visualizados as propriedades dos recursos, consultar os dados em bruto ou graficamente e tomar uma ação em relação a qualquer serviço monitorizado. É nesta página também que são ligados os alertas.

- Centro de operações - é apresentada o estado da implementação dos alvos monitorizados, incluindo o histórico de alertas, eventos e os serviços que não estão a executar.

- Existem ainda páginas específicas para determinados serviços como o *VSphere* ou *GemFire*.

Esta plataforma precisa de *software* adicional da *VMware* para ser inicializada.

2.1.2 Ferramentas de Monitorização de Infraestrutura

Zabbix

Zabbix é uma ferramenta de monitorização *open-source* recursos computacionais incluindo, rede, servidores, máquinas virtuais e serviços *Cloud*. Zabbix recolhe métricas sobre utilização de rede,carga de Central Processing Unit (CPU) ou consumo de disco. Os elementos alvos de monitorização são configurados no Zabbix através de um ficheiro eXtensible Markup Language (XML) [52].

O Zabbix consegue monitorizar qualquer tipo de família de sistemas operativos, tais como Linux, HP-UX, Mac OS X ou Solaris. Para monitorizar ambientes Windows são necessários agentes.

O armazenamento dos dados recolhidos pode ser feito com recurso aos mais populares gestores de base de dados, como o MySQL, MariaDB, PostgreSQL, SQLite, Oracle ou IBM DB2.

O Zabbix é escrito em C e a sua interface gráfica foi desenvolvida em PHP. A monitorização no Zabbix pode ser feita de vários tipos:

- Verificação simples - Apenas verifica a disponibilidade dos sistemas sem instalação de *software* adicional.
- Monitorização com base em agentes - Agentes instalados no sistema a monitorizar. No caso desse sistema ser Windows, a utilização deste tipo de monitorização torna-se obrigatória. Este tipo de monitorização fornece estatísticas como a carga de CPU, utilização de rede ou espaço disponível em disco.
- Monitorização com base em protocolos - Em alternativa ao tipo de monitorização anterior, o Zabbix suporta a monitorização através de protocolos como o Simple Network Management Protocol (SNMP), Transmission Control Protocol (TCP) ou Internet Control Message Protocol (ICMP).

A Figura 2.4 , ilustra a arquitetura do Zabbix.

O Zabbix tem uma interface (Zabbix *front-end*) que é constituída por páginas PHP, as páginas acedem a uma base de dados que contém os dados para posterior elaboração dos gráficos.

O servidor do Zabbix é responsável pela recolha de informação dos agentes que estão instalados nas máquinas e, por sua vez, envia essa informação para a base de dados.

Existem dois modelos de comunicação entre o agente e o servidor, no modelo *push* o agente envia informações para o servidor, no modelo *pull* o servidor decide que serviços o agente monitoriza. No modelo *push* a recolha de informação é feita pelo *zapcat trapper* que envia por sua vez esta informação ao servidor Zabbix [29].

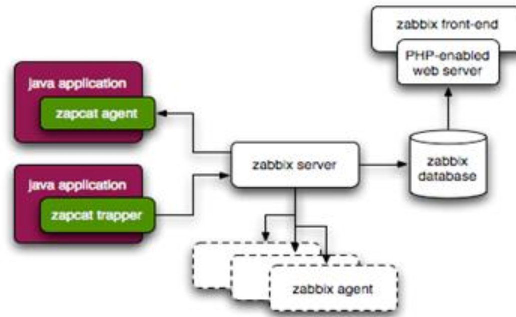

Figura 2.4: Arquitetura Zabbix (fonte [29]).

Nagios

O Nagios é uma aplicação *open source* que foi desenvolvida por Ethan Galstad [3]. Esta aplicação monitoriza protocolos como SNMP, HTTP, ICMP, etc. Para além disso, monitoriza máquinas onde recolhe essencialmente informação do processador, espaço disponível em disco e memória. Para fazer essa recolha de informação, o Nagios recorre a *plugins* que retornam um resultado do estado de cada máquina. O Nagios vai atuar conforme o resultado que recebe desses *plugins.* Se existir alguma perturbação, o Nagios irá notificar o gestor de rede [40].

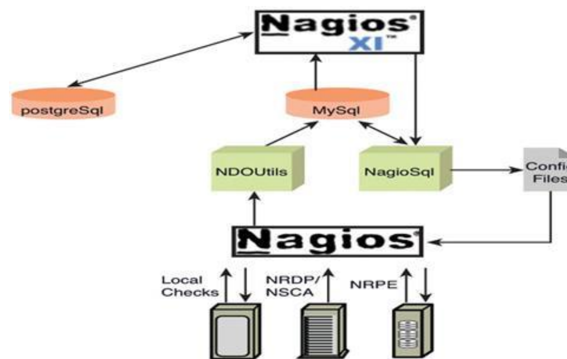

Figura 2.5: Arquitetura Nagios (fonte [25]).

O Nagios é constituído por plugins que estão representados na Figura 2.5 como o NDOUtils. Este *plugin* recolhe informações do Nagios Core, componente que representa o núcleo de processamento, e coloca-as numa base de dados MySql. A interface do Nagios (Nagios XI) para consulta essa base de dados de modo a obter informação sobre o estado das máquinas.

O componente NagiosQL permite, através da interface do Nagios, fazer alterações no seu núcleo. Estas alterações são escritas num ficheiro de configuração que por sua vez irão ser enviadas para o núcleo do Nagios.

Por fim, a interface do Nagios possui uma base de dados que está representada na Figura 2.5 como estando alojada no PostgreSQL, que guarda informações do utilizador e informações de autenticação [25].

2.1.3 Ferramentas de Monitorização de *Containers*

CloudHealth

CloudHealth Monitoring divide a monitorização em diferentes atributos de qualidade repartindo-os em subatributos. Os atributos de qualidade são atributos de alto nível que pode ser selecionados por operadores de *Cloud* como por exemplo o desempenho.

Por cada subatributo, que é um atributos de qualidade de alto nível decomposto, existem métricas. Para a recolha das métricas são usadas *probes* que são instaladas nos sistemas a monitorizar. Os dados são visualizados numa *dashboard*.

CloudHealth opera em três fases [48]:

- Fase de Configuração: Seleção dos atributos de qualidade que devem ser computados e observados durante a operação. Estes atributos são automaticamente mapeados para métricas a recolher;
- Fase de implementação: As métricas a recolher são mapeadas para um conjunto de *probes*;
- Fase de operação: A *dashboard* é automaticamente configurado de acordo com os atributos de qualidade que foram selecionados na fase de configuração.

A identificação dos atributos de qualidade a monitorizar foi feita através do estudo de três modelos de qualidade definidos por *standards ISO* que são *IT service quality*, *product quality* e os modelos de *quality in use*.

Depois deste estudo, foram estudados a confiabilidade, responsividade, adaptatibilidade, efetividade, eficiência, compatibilidade e desempenho.

Cada subatributo é depois mapeado para os recursos *Cloud* a serem monitorizados. A Figura ilustra o modo de funcionamento do *CloudHealth*.

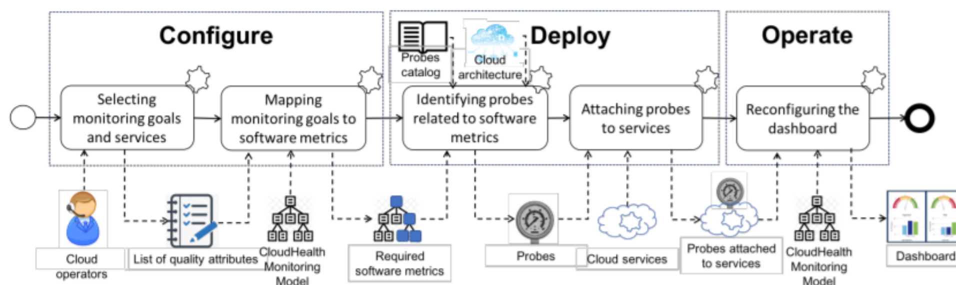

Figura 2.6: Arquitetura CloudHealth (fonte [48]).

A seleção de atributos de qualidade a monitorizar e de serviços é feita pelo utilizador, que levam à recolha de diferentes Key Performance Indicator (KPI). Depois da escolha dos atributos de qualidade, o utilizador pode escolher quais das propriedades que quer monitorizar, podendo optar por dar mais ênfase a uma em relação às restantes. O modelo é flexível ao ponto de se poder criar novos atributos de qualidade e subatributos.

Na fase de identificação das *probes*, o *CloudHealth* identifica automaticamente as *probes* que devem ser usadas para monitorizar as métricas identificadas anteriormente nos serviços selecionados.

Este processo é feito devido ao facto de o *CloudHealth* ter em conta a arquitetura da *Cloud* a monitorizar. Consoante essa arquitetura, o *CloudHealth* possui um catálogo de *probes* que podem ser usadas para recolher as métricas. A implementação das *probes* é feita através de gestores de configuração como o *Ansible*, *Chef* ou o *Puppet*.

Na implementação das *probes* nos serviços a monitorizar, podem existir duas situações:

- O serviço a monitorizar e a *probe* são implementados ao mesmo tempo;
- O serviço a monitorizar já está a ser executado quando a *probe* é implementada.

Quando o serviço e a *probe* vão ser implementados ao mesmo tempo, existem diferentes maneiras de implementação do serviço e da *probe* consoante a tecnologia.

Por exemplo, no caso do ambiente ser o de um operador de *Cloud*, é usado um *Cloud Package*, no caso do Microsoft *Azure*, um *Template* no caso do Google *Cloud*, no caso do ambiente ser um ambiente constituído por máquinas virtuais, é criada uma máquina virtual com o serviço e a *probe* já implementados.

Finalmente, no caso de *containers*, é criado um *container* novo com a *probe* a ser executada.

Quando o serviço a monitorizar já está a ser executado, existem várias opções:

- O serviço é desligado e, tanto a *probe* como o serviço são implementados ao mesmo tempo, ou seja, usando a estratégia anterior;
- Uma nova instância do serviço juntamente com a *probe* é implementada, para isso é necessário um balanceador de carga para que faça a distribuição do tráfego até que serviço original não servir nenhum *request* e seja parado;
- É possível implementar a *probe* com o serviço a executar, mas, para isso, é necessário apenas injetar a *probe* no sistema que está a correr o serviço. Para a injeção da *probe* é necessário ter acesso em *runtime* ao sistema onde o serviço está a executar. Esta ação pode ser executada por uma espécie de agente.

Cada *probe* deve ser configurada adequadamente de modo a que esta seja capaz de reportar os dados recolhidos à ferramenta de monitorização.

A *dashboard* irá sempre reportar apenas um *score* por cada atributo de qualidade monitorizado. A *dashboard* é dinamicamente definida com base no modelo CHHM que relaciona o estado do sistema com os valores das métricas recolhidas.

LOUD

LOUD é um ferramenta online que combina aprendizagem máquina com algoritmos de centralidade de grafos. LOUD monitoriza KPI's.

A aprendizagem máquina é usada não só para detetar anomalias, mas também serve para revelar a possibilidade de relação entre anomalias. Para complementar os algoritmos de aprendizagem máquina, LOUD usa algoritmos baseados em índices centralizados para localizar os recursos com algum tipo de perturbação.

Ao tentar encontrar a relação entre anomalias, LOUD também consegue controlar os falsos positivos.

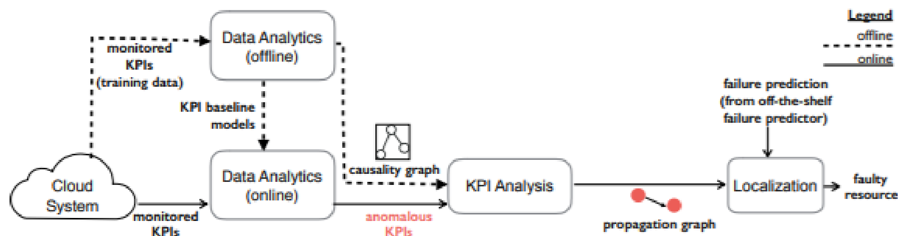

Figura 2.7: Arquitetura LOUD (fonte [35]).

A Figura 3 representa o modo de funcionamento do LOUD.

A fase LOUD *Data Analytics offline* monitoriza o sistema *Cloud* que opera no sistema a monitorizar para modelar a relação entre KPIs quando o sistema está a funcionar normalmente [35]. Os valores para os KPIs são recolhidos por *probes*.

Esta fase produz um conjunto de modelos de KPIs e gráficos de causalidade. Os modelos representam os valores de KPI que caracterizam comportamentos normais.

Os gráficos de causalidade modelam as relações de causalidade entre KPIs.

A fase LOUD *Data Analytics online* identifica as anomalias analisando os KPIs se estes violarem os valores base [35].

A fase LOUD *Localization* combina os conjuntos de KPIs anómalos com os gráficos de causalidade para gerar gráficos de propagação que apenas têm os KPIs anómalos [35].

Um KPI é um par composto pela métrica e pelo recurso a monitorizar. A métrica recolhe aspetos que podem ser medidos do comportamento do sistema a monitorizar e o recurso pode ser qualquer elemento do sistema.

TICK Stack

TICK Stack é uma plataforma que contém várias ferramentas *open source* com o objetivo de recolher, armazenar, mostrar e alertar dados provindos de *containers*. Todos os componentes do TICK Stack são dependentes uns dos outros. A Figura 2.8 ilusta a arquitetura desta ferramenta:

Esta plataforma é consituída pelas seguintes ferramentas [10]:

- InfluxDB - que é responsável pela armazenamento dos dados, pois trata-se de um base de dados de alto desempenho que consegue armazenar dezenas de milhares registos por segundo. O InfluxDB é uma base de dados SQL especialmente desenvolvida para o armazenamento da informação baseada no tempo;

- Telegraf - Agente responsável pela recolha das métricas através de *plugins* instalados nos sistemas a monitorizar. Nesta plataforma é usado não só para fazer a recolha das métricas, mas também para enviar essa informação para a base de dados InfluxDB. Este componente suporta a recolha de métricas de mais de 100 dos serviços mais usados;

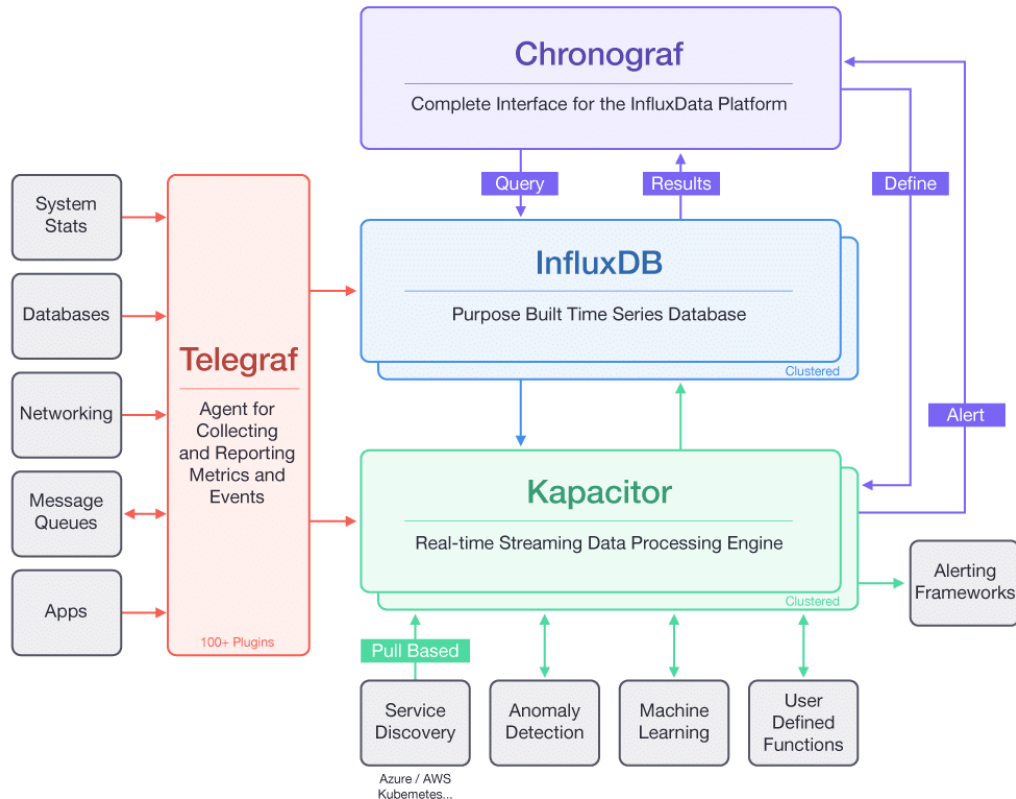

Figura 2.8: Arquitetura TICK Stack (fonte [27]).

- Chronograf - é o componente responsável pelo fornecimento de interface gráfica para o utilizador. Este componente constrói os gráficos com base na informação que está armazenada no componente InfluxDB e dispara os alertas para serem processados no Kapacitor;

- Kapacitor - é o processador de métricas e eventos e gestor de alertas. É responsável por transformar os dados em alertas e enviá-los para a maioria dos serviços mais populares, tais como o *Slack* ou o *PagerDuty*. Para além disso, pode ser integrado com ferramentas de deteção de anomalias, aprendizagem máquina, ou mesmo uma configuração manual. Possui ainda o mecanismo de descoberta de serviços que é útil quando o número de sistemas a monitorizar se torna elevado.

cAdvisor

cAdvisor é uma ferramenta de monitorização que fornece aos administradores de *containers* informações sobre o consumo de recursos e desempenho dos *containers* a executar. Esta ferramenta recolhe, agrega, processa e exporta informações sobre os *containers* que estão em execução [11].

Apenas suporta a monitorização de *containers Docker*.

O cAdvisor é uma ferramenta bastante simples, pois não tem uma solução de armazenamento prolongado dos dados recolhidos. Para além disso, tem um serviço web que pode ser acedido pelo *browser* para ver os dados recolhidos.

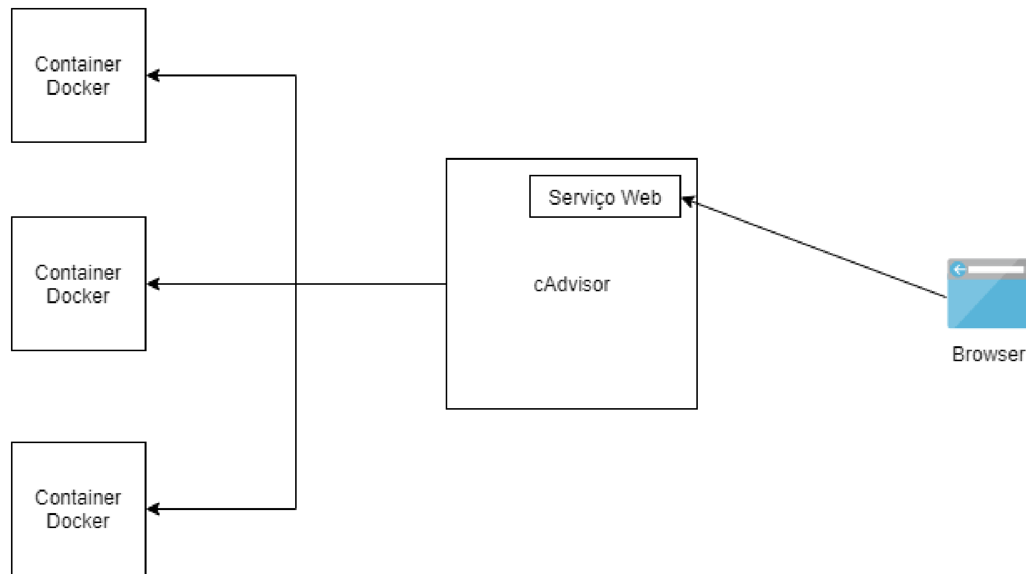

Figura 2.9: Arquitetura cAdvisor.

2.1.4 Resumo

Analisando todas as ferramentas descritas anteriormente, vemos que algumas não possuem características importantes que a plataforma desenvolvida neste estágio possui.

Começando pelo Prometheus, esta ferramenta não suporta *logs*, que são um componente de extrema importância para ter uma perceção total do que se passa no sistema.

Outra desvantagem do uso do Prometheus é o facto de não armazenar os dados durante muito tempo e também não é escalável, que é, talvez, o requisito não funcional mais importante no que diz respeito a ambientes *Cloud*.

Em relação ao *CloudHealth*, o funcionamento do sistema de monitorização é muito semelhante ao funcionamento desta plataforma, mas não armazena o histórico de valores recolhidos para os serviços a monitorizar.

Outra desvantagem deste sistema em relação à plataforma desenvolvida é facto de que quando se pretende monitorizar um serviço que já está a executar, as opções são ou desligar o serviço ou injetar o software da *probe* no *container* do serviço, o que faz com que o desempenho desça. Na plataforma desenvolvida, as *probes* são inicializadas num outro *container* podendo o serviço a monitorizar já estar em execução.

As ferramentas *LOUD* e *TICK Stack* têm como desvantagem principal o facto de que a *probe* tem de estar a executar na mesma máquina do serviço.

Ferramentas como o *Nagios* e o *Zabbix* são as mais populares no âmbito de monitorização de recursos computacionais, mas não são flexíveis o suficiente para que se possa criar *probes* para monitorizarem diferentes métricas.

Ferramentas como o *Graphite* e o *Hyperic* são mais flexíveis, mas necessitam que se altere o código da aplicação para inserir bibliotecas para que a comunicação com o servidor principal da ferramenta.

Por fim, o cAdvisor apenas tem como desvantagem o facto de apenas monitorizar *containers Docker* o que faz com que não seja compatível com outras tecnologias de construção de *containers*, sendo que a plataforma que se desenvolveu é flexível o suficiente para suporta

outros tipos de tecnologia para além do *Docker*.

2.2 Microserviços

Microserviços são serviços leves computacionalmente e de complexidade reduzida que podem ser executados, escalados e testados de forma independente. Cada microserviço tem as suas interfaces explícitas e bem definidas. Devido a esta simplicidade, a utilização deste tipo de serviços pressupõem a utilização de protocolos de comunicação entre eles igualmente simples. Este tipo de serviços são executados recorrendo, normalmente, a *framework* de orquestração, possibilitanto a sua inicialização em qualquer altura sem que haja um processo. No contexto da *Cloud*, os microserviços são executados em *containers* [43].

As vantagens de desenvolver este tipo de serviço são: qualquer alteração na implementação de um serviço não tem qualquer impacto na arquitetura geral da aplicação e na interação com os outros serviços, pois são executados de forma independente; a perceção de todo o sistema fica mais facilitada; o seu desenvolvimento e teste também são mais fáceis [43].

Com base no conceito de microserviços, foi criada uma arquitetura baseada neste tipo de serviços que possibilita a formação de equipas de desenvolvimento que podem desenvolver, executar e escalar os seus serviços em paralelo.

Para ajudar na projeção de uma aquitetura baseada em microserviços e no posterior desenvolvimento dos mesmos, existem alguns padrões de microserviços que não são mais do que as melhores práticas para o desenolvimento de uma aplicação baseada em microserviços. Entre os padrões de microserviços existentes, podem-se encontrar, por exemplo, a agregação de *logs*, *tokens* de acesso e *health check* API.

2.2.1 Implementação de microservicos

Os microserviços são pequenas aplicações que estão interligadas por API Representational state transfer (REST). No contexto deste estágio, o componente `TMA_Monitor` é uma API REST que disponibilizará um *endpoint* pelo qual as *probes* comunicam com a plataforma [49].

Como este componente foi desenvolvido em Python foi necessário escolher qual a *micro-framework* que se adequa aos requisitos da plataforma. Nas próximas subsecções vão ser descritas as *micro-frameworks* estudadas. No fim é feita uma comparação com o objetivo de eleger a melhor.

Falcon

Falcon é uma *microframework* web confiável escrita na linguagem de programação Python para desenvolver microserviços. Esta microframework utiliza o modelo REST e disponibiliza um número considerável de complementos e *templates* para usar em projetos de desenvolvimento [18].

O Falcon tem algumas características interessantes, tais como:

- Rapidez – O Falcon consegue processar os pedidos rapidamente;
- Confiável – Todo o código desta *framework* é testado rigorosamente com vários inputs;

- Flexível – Falcon dá ao programador a oportunidade de tomar quase todas as decisões, sendo altamente personalizável;
- Debuggable – O Falcon é uma *framework* bem documentada e mostra sempre uma descrição completa de erros de compilação.

O Falcon possui ainda algumas funcionalidades, como o fácil acesso aos cabeçalhos e corpos de um pacote HTTP ou respostas de erro HTTP bem descritas.

Flask

Flask é uma microframework web simples, confiável e leve computacionalmente. É desenhado para que seja o mais fácil possível começar a desenvolver aplicações escaláveis com alguma complexidade [21].

O Flask mostra sugestões, mas não é obrigatório adotá-las, pois é o programador que toma todas as decisões. À semelhança do Falcon, existem uma grande quantidade de complementos fornecidos pela comunidade, o que faz com que a adição de novas funcionalidades mais fácil. O Flask tem compatibilidade com qualquer servidor web.

Bottle

O Bottle é uma microframework para aplicações web leve e rápida. É constituída por apenas um ficheiro, tendo só as dependências standard do Python [9].

De entre outras características que o Bottle tem, destacam-se as seguintes:

- Encaminhamento dinâmico de entre todos os endpoints da API;
- Compatibilidade com vários *templates*;
- Acesso fácil a dados de um formulário, upload de ficheiros, *cookies*, cabeçalhos e outros campos de um pacote HTTP.

O Bottle também é compatível com todos os servidores web mais usados atualmente.

Django

Django é uma *microframework open-source* e confiável de alto nível que proporciona o rápido desenvolvimento e uma estruturação limpa e pragmática. Permite ao programador concentrar-se apenas no desenvolvimento da aplicação em vez de preocupar com problemas de rede [15].

O Django é uma *framework* rápida, pois auxilia os programadores a concluírem as suas aplicações o mais rápido possível.

Esta *framework* também traz bastantes complementos extras que podem ser usados nas tarefas de desenvolvimento Web, como por exemplo, inclui *add-ons* para autenticação, *feeds* Ritch Site Summary (RSS), entre outros.

O Django facilita a escalabilidade da aplicação web e é bastante versátil pois pode ser aplicado para sistemas de informação, redes sociais e plataformas de investigação científica.

O aspeto mais importante do Django é a segurança a ataques informáticos, pois esta *framework* ajuda os programadores a evitar ataques como *SQL Injection, XSS*, entre outros, vem também com um sistema de autenticação que permite gerir contas de utilizadores e respetivas palavras-passe.

Resumo

Nesta subsecção, vai ser mostrada uma tabela comparativa com as principais características que uma *microframework* deve possuir de modo a que se possa eleger a melhor neste âmbito.

A Tabela 2.1 faz uma comparação das *microframeworks* que foram anteriormente descritas.

Tabela 2.1: Tablela comparativa das microframeworks para REST API

	Falcon	Flask	Bottle	Django
Computacionalmente leve	X	X	X	
Apenas funcionalidades necessárias		X	X	
Suporte a funções	X	X		X
Confiabilidade do código	X	X		X

Na Tabela 2.1, estão presentes as principais características que uma *microframework* deve possuir. Através da análise desta tabela, podemos concluir que o Flask é a melhor ferramenta, porque tem todas as características que estão presentes nesta tabela.

O Falcon é uma boa *framework*, mas vem com muitas funcionalidades que se pode não necessitar e, consequentemente, fará com que o desempenho seja menor em relação ao Flask, por exemplo.

O Bottle não suporta funções, pois só suporta a compilação de um ficheiro, o que impõe graves problemas na estruturação de código. Adicionalmente, o seu código não é muito confiável.

Por fim, o Django é a *framework* mais pesada computacionalmente de todas as estudadas, o que faz com que o seu desempenho seja significativamente pior que os restantes.

2.2.2 Escalabilidade de microserviços

Para manter o desempenho e a disponibilidade de uma aplicação, a escalabilidade torna-se um requisito não funcional importante. A escalibilidade pode-se dividir em dois tipos [28]:

- Escalbilidade vertical - Neste tipo de escalabilidade é aumentado a capacidade de uma aplicação adicionando mais *hardware* ao ambiente onde ela está a ser executada.
- Escalabilidade horizontal - Ao contrário da escalabilidade vertical, neste tipo de escalabilidade, são criadas mais instâncias da aplicação.

Os microserviços são habitualmente desenvolvidos recorrendo a tecnolgias elásticas, mas isso não significa que ele sejam automaticamente escalados, pois existem alguns aspetos dos microserviços que têm de ser pensados para que estes possam ser escalados eficientemente.

A escalbilidade de uma aplicação baseada em microserviços difere de uma aplicação baseada num outro tipo de conceito ou arquitetura. No contexto de microserviços a escalabilidade

aplica-se ao nível do serviço não da aplicação em geral, ou seja, quando se escala um serviço, os outros não sofrem qualquer tipo de alteração [22].

Com base no que foi dito anteriormente, quando se trata de aplicações baseadas em microserviços, o tipo de escalabilidade mais vantajoso é a escalablidade horizontal.

No entanto, escalar um microserviço não é sufuciente para que o desempenho desse mesmo serviço sofra alterações. Para que o desempenho seja maior torna-se necessário conjugar a escalabilidade com o balanceamento de carga, de modo a que o serviço lide corretamente com a carga, com a segurança e se mantenha disponível, mitigando, assim, alguns ataques informáticos como Distributed Denial-of-Service Attack (DDoS).

2.3 Ambientes de *Containers*

Os *containers* fornecem a possibilidade de empacotar todas as dependências e código de aplicações de modo a que seja possível executar qualquer aplicação em qualquer ambiente de uma forma leve computacionalmente. Cada aplicação que é executada dentro de um *container* tem um ambiente completamente isolado das restantes.

Tudo isto faz com que os *containers* ofereçam um ambiente em que o tempo de execução da aplicação é consistente, ocupem pouco espaço em disco, pois apenas é armazenado o que é estritamente necessário para a execução da aplicação e que haja um sobrecarga baixa no sistema que aloja os *containers* devido ao isolamento de recursos.

Por todas estas razões, o projeto `ATMOSPHERE` tem a maior parte dos seus serviços implementados sob a forma de *containers*, o que fez com que a plataforma desenvolvida neste estágio também assim o tivesse de ser.

No âmbito da tecnologia dos *containers* existem ferramentas que são responsáveis pela virtualização e construção dos *containers*. Por outro lado, uma aplicação baseada em microserviços pode ser composta por vários serviços, o que leva ao aumento da complexidade na gestão de todos esses serviços. No entanto existem também ferramentas que fazem com que toda esta gestão fique mais facilitada.

Devido às necessidades da plataforma foi necessário estudar qual o virtualizador e o gestor de *containers* mais adequado.

2.3.1 Virtualizadores de *containers*

Um *container* é um pacote executável de uma aplicação que contém tudo o que é necessário para executá-la como, por exemplo, código, ferramentas, bibliotecas e definições de sistema. As aplicações que estão dentro de *containers* executarão sempre da mesma forma independentemente do sistema em que são executadas, já que os *containers* isolam o *software* que possuem de tudo o que o rodeia, reduzindo, assim, possíveis conflitos.

Os virtualizadores de *containers* são responsáveis por construir e executar *containers*. Neste contexto foram estudados o *Docker*, Linux Containers (LXC) e *CoreOS Rocket*.

Docker

Docker é uma ferramenta *open-source* que constrói e executa *containers* fazendo com que desenvolver e distribuir aplicações seja o mais simples possível. Docker torna mais fácil a

automatização a execução de aplicações em *containers*. Numa infraestrutura virtualizada em *containers*, Docker cria um ambiente rápido e leve para que o código de uma aplicação possa ser testado antes de ser executado em produção. A arquitetura do Docker é constituída por quatro componentes: Cliente Docker e Docker Daemon, imagens Docker, Docker Registries e Docker *containers*. [8] A arquitetura do Docker é ilustrada na Figura 2.10.

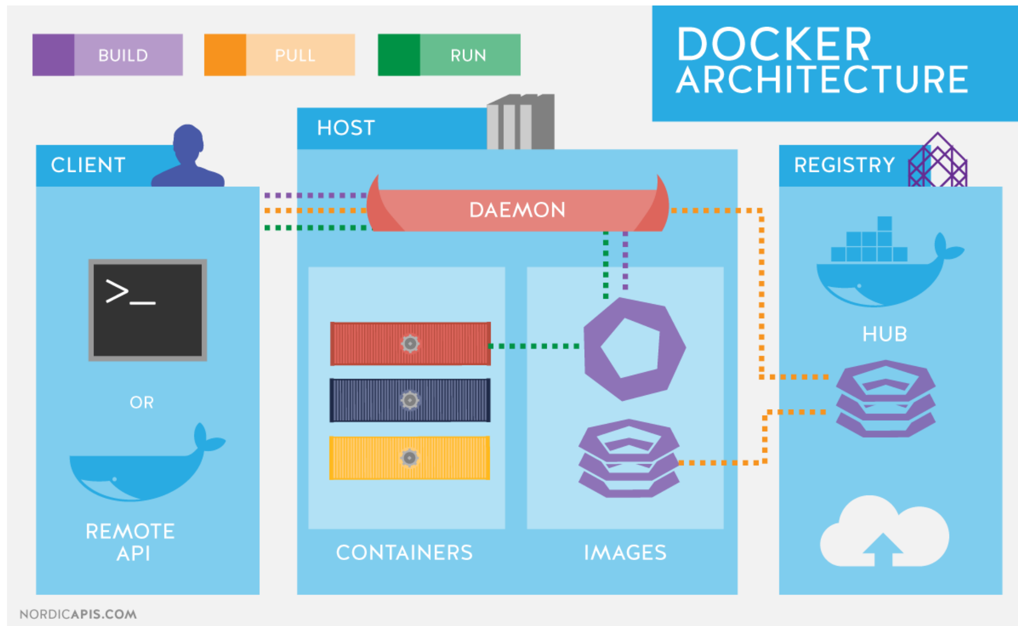

Figura 2.10: Arquitetura Docker (fonte [5]).

Docker Client é responsável por fazer pedidos para o Docker Daemon que os processa. Normalmente, o Docker Client e o Docker Daemon são executados na mesma máquina, mas também podem ser instalados em máquinas distintas.

As imagens Docker servem para criar e executar os *containers*. Existem, essencialmente, dois métodos para as criar. Um dos métodos consiste em utilizar uma imagem de um sistema operativo e, a partir daí, ir adicionando ficheiros a esse sistema operativo até que esteja devidamente configurado para a função desejada. Essa adição de ficheiros são guardadas na imagem sob a forma de *commits*.

O outro método é criar um Dockerfile que contém uma lista de instruções que são executadas quando o utilizador constrói a imagem, automatizando, assim, a adição de aplicações necessárias.

Docker Registries são repositórios públicos de imagens Docker. Estes repositórios funcionam da mesma forma que os repositórios de *software*, onde se pode fazer *push* ou *pull* de imagens. Existem repositórios públicos e privados. O repositório grátis mais conhecido e onde se encontra a grande maioria das imagens Docker chama-se Docker Hub.

Finalmente, Docker *containers* são o resultado da execução das imagens Docker. Os *containers* têm todos os pré-requisitos para que uma aplicação possa ser executada de maneira isolada.

Como principais funcionalidades, o Docker garante o isolamento e constução leve computacionalmente dos *containers*. O Docker é compatível com todos os sistemas operativos mais utilizados. As imagens criadas pelo Docker permitem que se façam alterações e permitem ainda que sejam reutilizadas como imagens de base para outros *containers*.

LXC e LXD

LXC é construtor de *containers* onde os recursos do sistema operativo são virtualizados, o que permite ter vários sistemas operativos na mesma máquina sob a forma de *containers*, aumentando, assim, o desempenho comparativamente à utilização de máquinas virtuais [45].

A arquitetura do LXC é ilustrada na Figura 2.11.

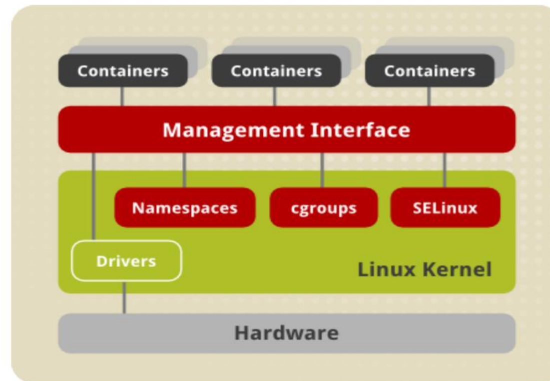

Figura 2.11: Arquitetura LXC (fonte [46]).

O LXC tem a funcionalidade de *namespaces* do *kernel* Linux para virtualizar e isolar recursos como configurações de rede, utilizadores, processos, sistema de ficheiros, entre outros. Este isolamento e virtualização permite que o *container* possa ser independente do sistema em que está a ser executado. Esta independência possibilita que o *container* possa ser migrado para outro sistema sem conflitos, que a gestão do *container* possa ser feita de uma forma mais completa, como se de um sistema independente se tratasse, e que possa também ter os seus próprios serviços de rede [17].

O LXC possibilita que os recursos do sistema, como o CPU, memória e ficheiros abertos que são consumidos pelo *container* sejam limitados através da utilização de mecanismos baseados em *cgroups*.

Os recursos dos *containers* são alocados segundo o algoritmo *Completely Fair Scheduler* (CFS), que determina que cada *container* tem uma prioridade. Por exemplo, a largura de banda é atribuída com base na sua prioridade. SELinux é usado para tornar o *container* independente quer do sistema em que está a executar, quer dos outros *containers* que possam estar a executar nesse mesmo sistema.

Semelhante a esta ferramenta, mas com o objetivo de fornecer ao utilizador uma nova experiência de utilização, existe o LXD. Esta ferramenta fornece uma única linha de comandos que permite criar e executar *containes*. Para aumentar a qualidade de utulização do utilizar, os *containers LXD* podem ser geridos a partir de qualquer lugar com conexão à internet através uma API REST [39].

CoreOS Rocket

CoreOS Rocket é um construtor de *containers* computacionalmente leve desenvolvido para ambientes cloud, o ambiente criado pelo Rocket foi desenvolvido e definido de modo a que possa ser integrado com outros sistemas.

A unidade básica do Rocket é o *pod* que não é mais do que um conjunto de aplicações que

são executadas num determinado ambiente. Esta definição é semelhante à do Kubernetes, descrita na próxima subsecção. O Rocket permite aos utilizadores isolarem o ambiente que criaram, quer ao nível do *pod*, quer a um nível mais pormenorizado, como, por exemplo, ao nível da aplicação que o *container* está a executar.

Segundo a arquitetura do Rocket, cada *container* é executado como se fosse um processo Unix isolado dos outros processos. O Rocket tem os seus próprios métodos de criação das imagens dos *containers*, mas também é compatível com a execução de imagens criadas pelo Docker.

Desde que foi introduzido pelo CoreOS em 2014, esta ferramenta teve bastantes atualizações, nomeadamente, a capacidade de guardar as alterações feitas dentro de um *container*. Desde aí passou a ser mais utilizada, o que fez com que se esteja a desenvolver pacotes de atualizações para que possa ser integrada com o Kubernetes.

A Figura 2.12 ilustra a arquitetura do CoreOS Rocket.

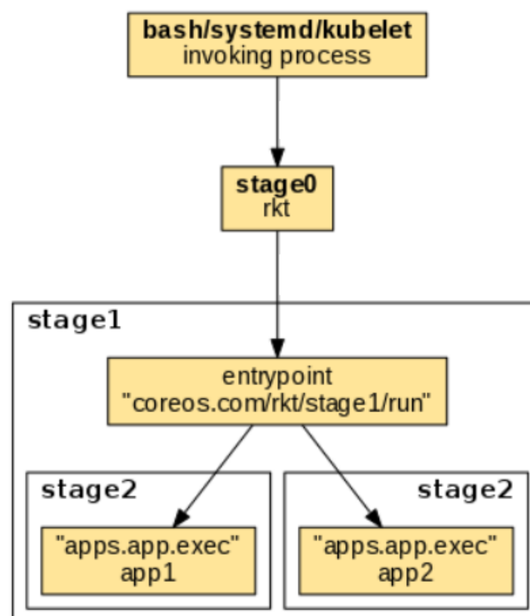

Figura 2.12: Funcionamento do CoreOS Rocket (fonte [12]).

Como podemos observar pela Figura 2.12, o funcionamento do Rocket consiste, basicamente, em três etapas. A etapa 0 consiste em executar o processo rkt do Rocket que lança uma instância da ferramenta. Na etapa 1, é executada a instrução para lançar um ou vários *pods* e, por fim, na etapa 2, são lançados os *containers* que pertencem a esse *pod* e, por consequência, as aplicações que se encontram dentro desses *containers* começam também a executar.

Resumo

Nesta subsecção, vai ser mostrada uma tabela comparativa com as principais características que um construtor de *containers* deve possuir de modo a que se possa eleger a melhor ferramenta neste âmbito

A Tabela 2.2 faz uma pequena comparação dos construtores de *containers* que foram anteriormente descritos.

Tabela 2.2: Tablela comparativa dos construtores de *containers*

	Docker	LXC/LXD	CoreOS Rocker
Isolamento do *containers*	X	X	X
Construção de *containers* leves computacionalmente	X	X	X
Compatibilidade com outros sistemas operativos	X		
Guardar alterações num *container*	X		X
Reutilização de imagens	X		
Suporte a partilha de imagens	X		
Gestão a partir de API REST		X	

Como podemos verificar nesta comparação, o Docker possui quase na totalidade todas as características principais que uma ferramenta deste âmbito deve possuir, apenas não fornecendo uma API REST para a gestão dos seus *containers*.

O LXC /LXD assegura o isolamento dos *containers* e a sua construção de modo a que estes fiquem o mais leves computacionalmente possível. Fornece ainda uma API REST para uma maior facilidade de gestão. O Rocket é melhor que o LXC, apenas por possibilitar a gravação de alterações nos *containers*, mas fica muito aquém do Docker.

2.3.2 Gestores de *containers*

Gestores de *containers* são responsáveis pela organização da execução dos *containers* e alocação de recursos para otimizar a eficiência e o balanceamento das cargas do sistema. Gestores de *containers* permitem o escalonamento dos *containers*, o que aumenta em larga escala o desempenho de aplicações distribuídas executadas neste tipo de ambiente.

Em suma, um gestor de *containers* deve ser capaz de:

- Organizar os *containers* em *clusters*;
- Automatizar ou agendar a execução de *containers*;
- Replicar *containers* que tenham a mesma função.

Nesta categoria foram estudadas as ferramentas Docker Swarm, Kubernetes e Apache Mesos.

Docker Swarm

Docker Swarm é uma ferramenta de agrupamento e escalonamento de *containers* Docker. A principal função do Docker Swarm é escolher qual máquina do *cluster* deverá executar o *container* em questão. O Docker Swarm suporta a replicação de *containers* com a mesma função. Este processo tem duas fases, em que uma é fazer o atendimento do pedido e a outra é escolher a máquina do *cluster* para executar o *container*.

Na fase de atendimento do pedido, o Docker Swarm utiliza o algoritmo *First In First Out* (FIFO), ou seja, o primeiro pedido a chegar é o primeiro a ser atendido. Na fase de escolha da máquina, o Docker Swarm pode ser configurado de acordo com várias estratégias.

A estratégia por omissão é a estratégia *spread* que escolhe a máquina que tiver menor número de *containers* a serem executados para executar o *container* em questão. Outra estratégia que o Docker Swarm utiliza é a estratégia *Binpack*, que segue o princípio oposto da anterior, ou seja, escolhe a máquina que tem mais *containers* a serem executados. Por fim, a última estratégia chama-se estratégia aleatória, que, como o próprio nome indica, o Docker Swarm escolhe uma máquina aleatoriamente.

A Figura 2.13 ilustra a arquitetura do Docker Swarm.

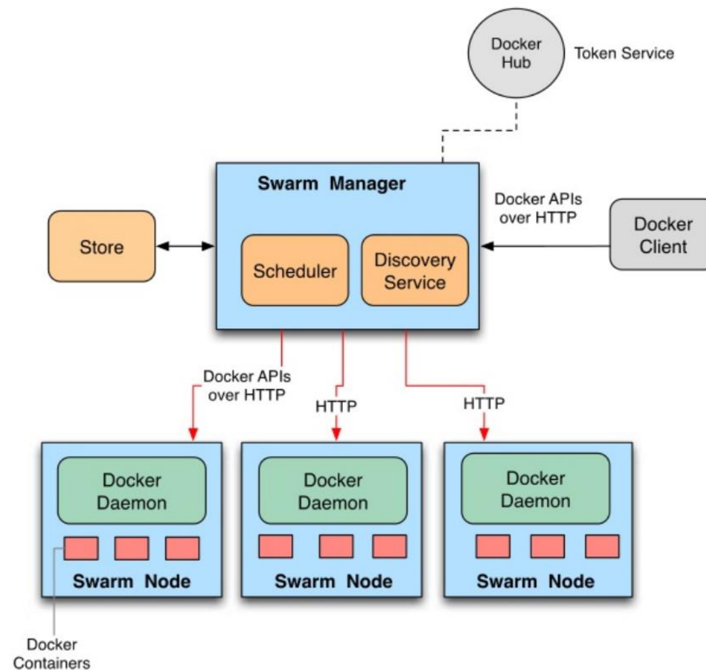

Figura 2.13: Arquitetura Docker Swarm (fonte [38]).

O Swarm Manager é o componente responsável por gerir os nós no Swarm *cluster* e acede ao Docker através da sua API. O Swarm Node é uma máquina que pertence ao *cluster* Swarm que tem:

- ID;
- Endereço IP;
- Lista de *Containers*;
- Lista de Imagens;
- Estado de *health*;
- Total de CPUs;
- CPUs Utilizados;
- Memória total;
- Memória usada.

O *Scheduler* é responsável por distribuir os *containers* pelos nós usando as três estratégias descritas anteriormente. O componente *Store* armazena o estado do *cluster* sob a forma

de um ficheiro Javascript Object (JSON) que é carregado em memória quando o *cluster* começa a ser executado [24].

O funcionamento do *Scheduler* baseia-se na seguinte sequência de ações: em primeiro lugar, obtém o estado de um *container* e armazena-o, depois carrega o ficheiro JSON novamente para memória, substitui o estado anterior com o novo e, por fim, apaga o estado anterior.

Finalmente, o Serviço de Descoberta tem como principais funções registar um novo nó e atualizar a lista do Swarm Manager.

Kubernetes

Kubernetes é gestor de *containers open-source* que automatiza a execução de aplicações em recursos computacionais. Para isso, o Kubernetes utiliza *containers* de modo a gerir esses recursos. O Kubernetes tem como principais funcionalidades o agrupamento de *containers* em *pods*, bem como o auto escalonamento dos mesmos.

A Figura 2.14 ilustra a arquitetura do Kubernetes.

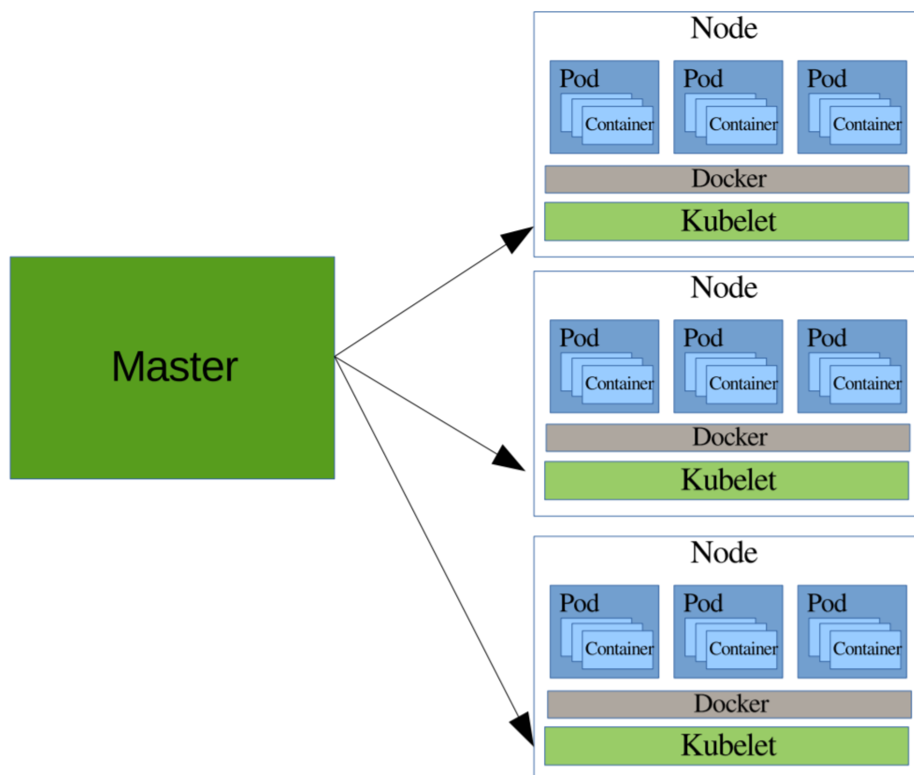

Figura 2.14: Arquitetura Kubernetes (fonte [37]).

Kubernetes é baseado numa arquitetura *Master-Node*. As aplicações são submetidas na máquina *Master*, sendo que depois, é esta que as distribuí por todos os nós do *cluster*. O *Master* também é capaz de executar *containers* desde que seja configurado para isso. A comunicação entre o *Master* e os nós do *cluster* é assegurada pelo serviço kubelet. O serviço kubelet é executado em todas as máquinas que pertencem ao *cluster* Kubernetes.

O Kubernetes utiliza o Docker para executar os *containers*, por isso, existe uma instância do Docker *Daemon* em cada máquina do *cluster*. Para coordenar os recursos para cada

container e partilhar informação entre as máquinas do *cluster*, o Kubernetes usa o projeto etcd. No Master existe ainda um API *server* com uma interface RESTful que permite que os serviços dentro do *cluster* possam ser acedidos externamente.

Para executar uma aplicação, esta tem de estar contido num ou em vários *containers* sob a forma de imagens. Por omissão, o Kubernetes tenta fazer download das imagens a partir de repositórios públicos. O Kubernetes executa as aplicações em forma de *pods*. Um *pod* é a unidade básica do Kubernetes, que representa um ou vários *containers*. Quando vários *containers* pertencem ao mesmo *pod*, estes partilham a mesma interface de rede e sistema de ficheiros.

Um *pod* tem duas características principais: é executado numa só máquina, mesmo que no mesmo *pod* estejam vários *containers*, e um *pod* tem um endereço Internet Protocol (IP) dentro da rede do *cluster* e todos os *containers* dentro desse *pod* partilham os mesmos portos.

Portanto, um *pod* tem um único endereço IP para comunicações entre todos os outros *pods* e restantes nós no *cluster* Kubernetes, impossibilitando, assim, que dentro de mesmo *pod* serem executados *containers* que tenham serviços no mesmo porto.

Um *pod* pode ser replicado por várias máquinas do *cluster* o que faz com que o serviço desse mesmo *pod* seja escalável e também tolerante a falhas. Quando existe a replicação de um serviço mais complexo, como, por exemplo, uma aplicação web com ligação a uma base de dados no mesmo *pod*, o Kubernetes apenas se foca no isolamento de recursos e não na coordenação do próprio serviço, por isso o Kubernetes não garante que o serviço distribuído esteja completamente funcional ao nível da aplicação. O Kubernetes apenas consegue saber quais os serviços a serem executados noutros *pods* através do serviço de Domain Name Server (DNS).

Quando assim configurado, o Kubernetes tem como funcionalidade principal o auto escalonamento dos seus *pods*. Cada *pod* é automaticamente escalado de acordo se a percentagem de CPU ativo ultrapassar o limite configurado.

No entanto, o algoritmo de auto escalonamento do Kubernetes tem várias condicionantes. Este algoritmo começa com a recolha periódica e configurável dos valores de CPU por partes dos *pods* alvos deste tipo de escalabilidade. Numa segunda fase, é comparada a média de amostras recolhidas durante um minuto com o limite de utilização de CPU configurado pelo administrador e, se se justificar, é ajustado o número de réplicas desse *pod* preservando o limite mínimo e máximo de réplicas também configurado pelo administrador.

A utilização de CPU por cada *pod* é dada pela divisão da média das amostras recolhidas no último minuto sobre o total de *cpu* que esse *pod* tem disponível. Por fim o número de *pods* é dado pela fórmula:

`Número de Pods = ceil(sum(Utilização de CPU)/ limite CPU`

Em que *ceil* é o inteiro mais pequeno maior ou igual ao somatório da divisão da utilização de CPU pelo limite definido pelo administrador.

Para evitar *outliers* provocados por qualquer tipo de perturbação esporádica, este algoritmo tem as seguintes restrições:

- O aumento das réplicas só é executado quando não houve algum escalonamento do *pod* nos últimos três minutos.
- A redução do número de réplicas só é feita se nenhum escalanomento tiver sido

efetuado nos últimos 5 minutos.

- O escalonamento só é efetuado se a média da divisão do consumo de CPU a dividir pelo limite configurado pelo administrador baixar de 0,9 ou estiver acima de 1,1, ou seja existe 10% de tolerância,

Com base nas restrições anteriormente referidas, pode-se afirmar que este algoritmo é um algoritmo conservativo no sentido em que se se verificar um aumento de carga no *pod*, é aumentado rapidamente o núemro de réplicas desse *pod* para que não haja perca de dados. Por outro lado, a redução do número de réplicas não é encarada como urgente.

Concluindo, este algoritmo foi desenhado para prevenir a execução rápida de ações de escalonamento se a carga no *pod* for instável.

Apache Mesos

Apache Mesos é uma ferramenta *open-source* para gerir *containers*, permitindo a replicação e auto escalonamento dos mesmos.

A arquitetura do Apache Mesos é ilustrada na Figura 2.15.

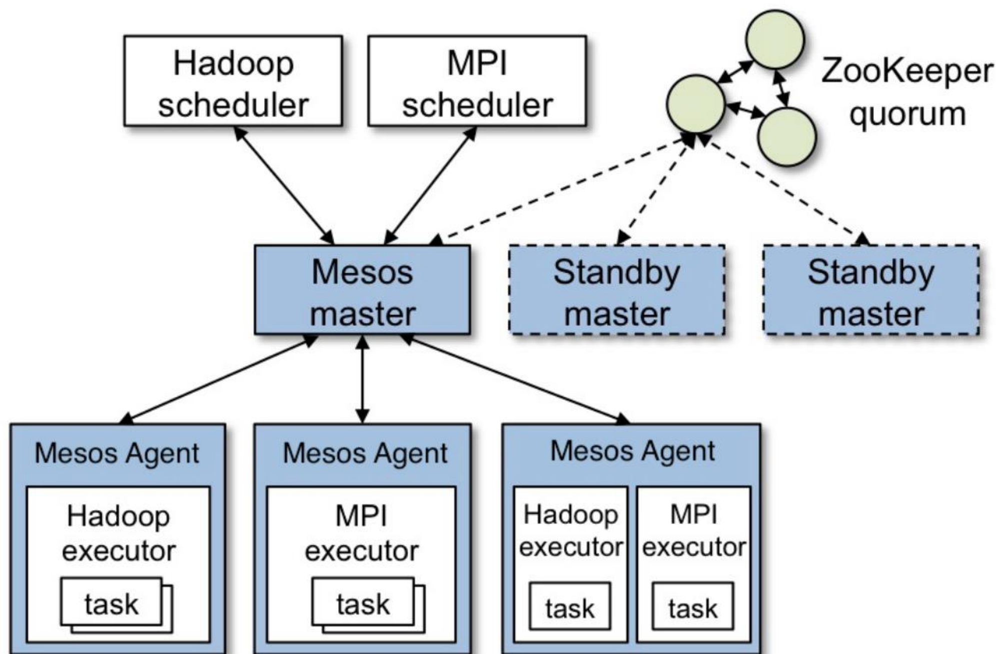

Figura 2.15: Arquitetura Apache Mesos (fonte [4]).

As unidades principais do funcionamento do Apache Mesos são o Mesos *Master* e o Mesos *Agent*. O Mesos *Master* é responsável por gerir os agentes que estão a ser executados em todos os nós do *cluster*. Existe ainda a Mesos *Framework* que tem como função executar as tarefas nos agentes.

Nas suas funções de gestão, o Mesos *Master* partilha os recursos computacionais do *cluster* por cada tarefa/*framework* de acordo com determinados algoritmos, tal como *fair sharing* ou *strict priority*. Podem adicionar-se mais políticas através de *plugins*.

Existem dois componentes que são executados em no *Master* Mesos que são o *scheduler* e

o *executer*. O *scheduler* é responsável por determinar que recursos dar a cada tarefa. O *executer* é o processo responsável por executar as tarefas em cada nó.

Basicamente, o processo de executar uma tarefa no Apache Mesos consiste em três passos. O primeiro passo é definir os recursos a atribuir a essa tarefa, o que é feito pelo Mesos *Master*, que cria uma descrição desses mesmos recursos. Este processo designa-se por *resource offer*. Depois, o *scheduler* atribui a *resource offer* a essa tarefa, e, por fim, o Mesos *Master* atribui a tarefa a um agente.

Resumo

Nesta subsecção, vai ser mostrada uma tabela comparativa com as principais características que um gestor de *containers* deve possuir de modo a que se possa eleger a melhor ferramenta neste âmbito.

A Tabela 2.3 faz uma pequena comparação dos gestores de *containers* que foram anteriormente descritos.

Tabela 2.3: Tablela comparativa dos gestores de *containers* estudados

	Docker Swarm	Kubernetes	Apache Mesos
Organização em *clusters*	X	X	X
Automatização de execução de *containers*	X	X	X
Replicação de *containers*	X	X	X
Auto-escalonamento dos *containers*		X	X
Suporte a *containers* Docker	X	X	X
Suporte a volumes persistentes remotos	X	X	

Como se pode observar na Tabela 2.3, todos os gestores de *containers* estudados cumprem com todos os requisitos enumerados no início desta subsecção. No entanto, existem outros dois critérios essenciais no contexto deste projeto que são o auto-escalonamento e o suporte a *containers* Docker que importa considerar.

Em relação ao suporte ao auto escalonamento, o Kubernetes e o Apache Mesos suportam esta funcionalidade enquanto que o Docker Swarm não, logo, o Docker Swarm não será o gestor de *containers* que mais se adequará a este contexto.

Por outro lado, em relação à possibilidade de configuração de volumes persistentes alojados remotamente, o Apache Mesos não cumpre com este requisito, e sendo esta um dos requisitos mais importantes desta plataforma para diminuir a possibilidade de perdas de dados, o Apache Mesos não será utilizado neste contexto.

Como o Kubernetes é o único gestor de *containers* estudado que possui todas as características principais de uma ferramenta deste tipo, neste projeto, irá utilizar-se o Kubernetes como gestor dos *containers* Docker nesta solução.

2.4 Processadores de Fluxo de Mensagens

Neste projeto é necessário encaminhar mensagens entre todos os componentes, de uma maneira segura, e tolerante a falhas. Todas estas características estão presentes em ferramentas de processamento de fluxos de mensagens. Por isso, no âmbito deste estágio, foram estudadas as ferramentas Apache Kafka, ZeroMQ, RabbitMQ e Nats.io.

2.4.1 Apache Kafka

Apache Kafka é uma ferramenta distribuída *open-source* de mensagens que assenta numa arquitetura de produtor-consumidor permitindo que, quando um produtor envia uma mensagem, todos os consumidores conseguem recebe-la.

A arquitetura do Apache Kafka é ilustrada na Figura 2.16.

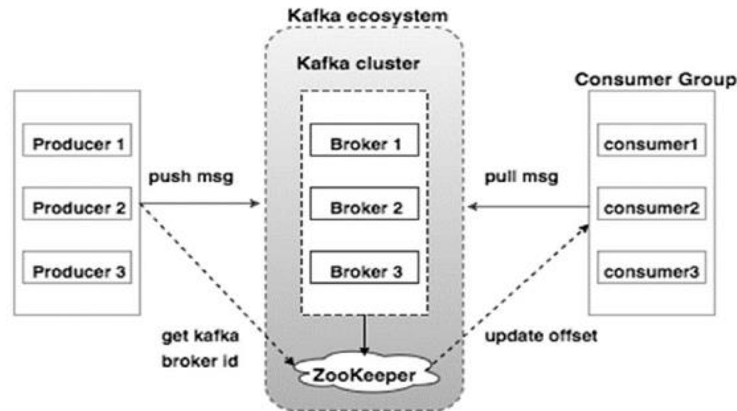

Figura 2.16: Arquitetura Apache Kafka (fonte [54]).

Os componentes principais do modo de funcionamento do Apache Kafka são os *Brokers*, o *Zookeeper*, os produtores e os consumidores.

Um *cluster* Kafka, normalmente, contém vários *Brokers* de modo a garantir balanceamento de carga. Os *Brokers* do Apache Kafka não guardam o seu estado, daí a necessidade de existência do Zookeeper para monitorizar o estado do *cluster* do Apache Kafka.

Um *Broker* é capaz de receber e enviar milhares de mensagens por segundo, sendo que um *cluster*, tal como a Figura 2.16 ilustra, é capaz de lidar com terabytes de dados sem qualquer impacto no seu desempenho. O Zookeeper também tem a função de escolher o *Broker* que irá reencaminhar a mensagem.

Como já foi dito, o Zookeeper é o responsável por gerir e coordenar todas as operações nos *Brokers* do Apache Kafka, mas a sua função principal é notificar os produtores e os consumidores de que novos *Brokers* estão disponíveis ou algum dos existentes deixou de funcionar.

Os produtores são os elementos que enviam os dados para os *Brokers*. Quando um novo *Broker* é iniciado, os produtores imediatamente começam a enviar mensagens para esse novo *Broker* até chegar ao seu limite da capacidade de processamento.

Por outro lado, o consumidor recebe as mensagens vindas dos produtores. Como os *Brokers* do Apache Kafka não guardam o seu estado, é o consumidor que tem a tarefa de contar quantas mensagens já recebeu para evitar receber mensagens em duplicado. Este controlo é feito através do campo *partition offset*, que pode ser manipulado pelo consumidor para voltar a receber ou rejeitar mensagens [6].

O valor de cada *offset* do consumidor é mantido no Zookeeper. Para receber mensagens, os consumidores fazem *pulls* assíncronos ao *Broker*.

O envio de mensagens do produtor para o consumidor é feito através de um tópico. Um tópico é um canal de comunicação onde vários produtores conseguem enviar mensagens e

vários consumidores são capazes de as receber.

Algumas das características que fazem com que o Apache Kafka seja uma das ferramentas mais populares neste contexto são a garantia de entrega de mensagens, o armazenamento persistente das mesmas e o facto de ser estável e escalável.

O Apache Kafka tem a capacidade de replicar estes canais e de fazer a sua partição, o que faz com que seja possível balanceamento de carga entre vários *Brokers* que possuam o mesmo tópico. Um tópico também pode ser dividido em várias partições, permitindo deste forma categorizar as mensagens para o mesmo tópico, possibilitando também que os consumidores recebem mensagens que são publicadas numa determinada partição de um tópico em específico.

2.4.2 ZeroMQ

O ZeroMQ não tem uma arquitetura definida tal como o Apache Kafka descrito anteriormente, mas é sim, uma biblioteca de processamento de mensagens que pode ser integrada em aplicações distribuídas para fornecer um serviço de troca de mensagens com elevada taxa de processamento e baixa latência.

O modelo que é utilizado no ZeroMQ é também um modelo de produtor-consumidor sem quaisquer sistemas intermediários. Para troca de mensagens, o ZeroMQ implementa um novo conceito de socket, como ilustra a Figura 2.17.

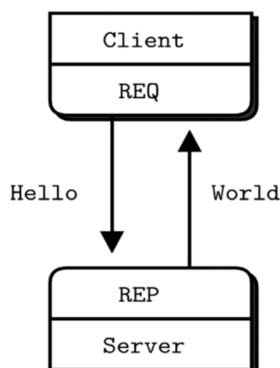

Figura 2.17: Exemplos de sockets do ZeroMQ (fonte [33]).

Os sockets do ZeroMQ vêm com um mecanismo que permite comunicação assíncrona, como se observa na Figura 2.17, que ilustra um sistema de pedidos e respostas. O ZeroMQ, quando o consumidor não estiver a funcionar, guarda as mensagens sem que seja necessária carga adicional para o consumidor [33].

Uma das desvantagens do ZeroMQ é o facto de ter de ser integrado na aplicação, o que adiciona complexidade à própria lógica da aplicação em questão, principalmente, para aplicações simples de troca de mensagens. No entanto, o ZeroMQ é uma ferramenta bastante estável, fazendo a sua própria monitorização.

2.4.3 RabbitMQ

RabbitMQ é um *Broker* de mensagens *open-source* computacionalmente leve escrito em *Erlang*. A arquitetura do RabbitMQ é bastante simples e modular, sendo que só suporta

dois protocolos nativamente, que são o AMQP e STOMP, embora protocolos como o HTTP possam ser adicionados através de *plug-ins*.

O RabbitMQ suporta todas as principais funcionalidades que um sistema de mensagens deve possuir, como persistência de mensagens, alta disponibilidade, garantia de entrega de mensagens, fazer a sua própria monitorização e suportar a replicação e particionamento dos tópicos.

A Figura 2.18 ilustra a arquitetura do RabbitMQ.

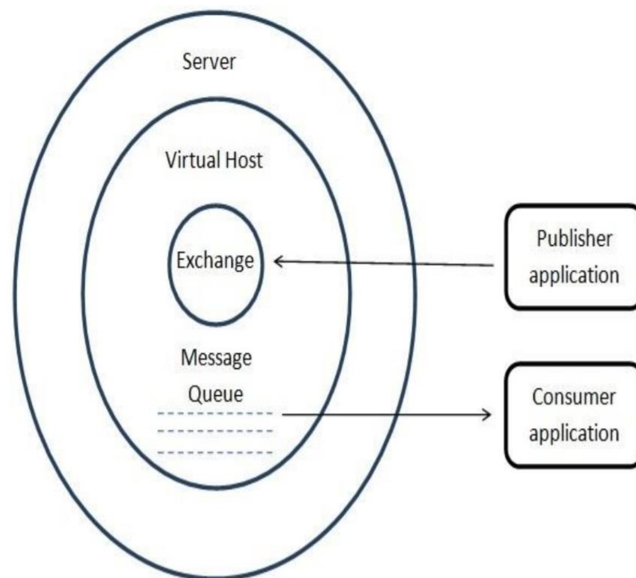

Figura 2.18: Arquitetura RabbitMQ (fonte [30]).

A arquitetura do RabbitMQ tem como componente principal o servidor que tem duas funções principais, que são aceitar as mensagens do produtor e depois reencaminhá-las para os vários consumidores. Quando os consumidores não estão ligados ao servidor, este guarda as mensagens em memória para que, quando um novo consumidor se ligar, imediatamente receba todas as mensagens que perdeu e que estão armazenadas no servidor [36].

Como foi dito anteriormente, toda a comunicação entre o servidor, consumidores e produtores é feita através do protocolo AMQ por omissão. Este protocolo divide as mensagens recebidas em pequenos blocos e depois combina-as de uma maneira mais consistente e robusta.

Depois, o módulo *Exchange* manda as mensagens para as respetivas filas de espera para depois serem enviadas para o consumidor.

2.4.4 Nats.io

O Nats.io é um serviço de mensagens *open-source* e é escrito na linguagem de programação *Go*. Existem muitas bibliotecas de clientes escritas na maior parte das linguagens existentes para interagir com o *Broker* de mensagens.

O Nats.io tem como principais características ter um bom desempenho, escalável, fácil de usar, garantir a entrega das mensagens e incluir funcionalidade de monitorização do seu estado.

A Figura 2.19 ilustra a arquitetura interna do Nats.io.

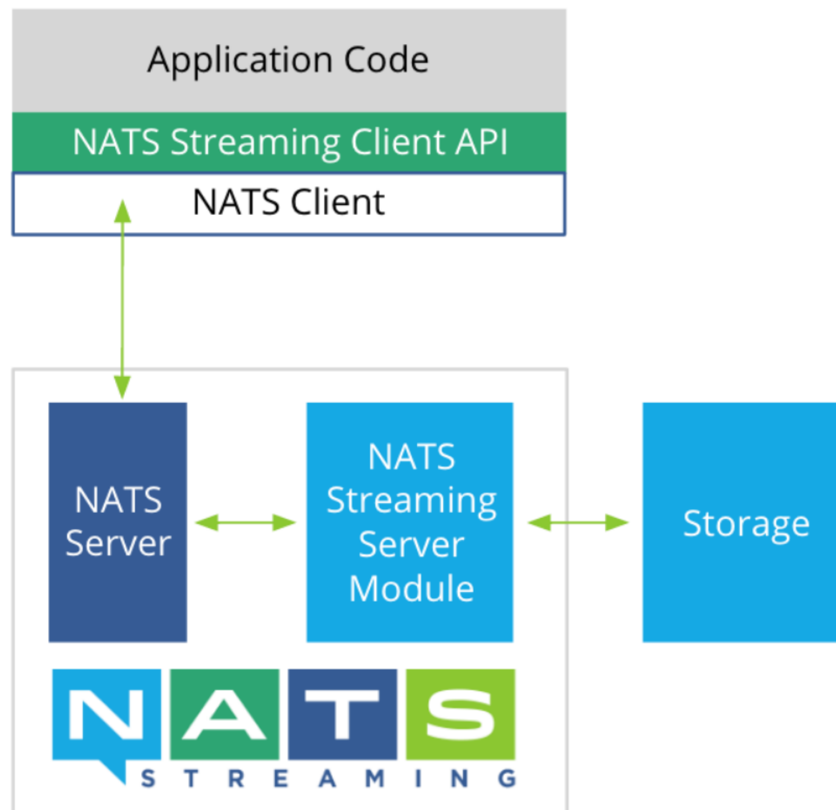

Figura 2.19: Arquitetura Nats.io (fonte [41]).

O cliente NATS é responsável por enviar eventos da aplicação em que está inserido para o servidor NATS, que pode ser executado numa máquina separada ou pode estar junto com o NATS *Streaming Server Module*, que depois armazena as mensagens. As mensagens podem ser armazenadas em memória ou em ficheiros.

2.4.5 Resumo

Nesta subsecção, vai ser mostrada uma tabela comparativa com as principais características que um processador de fluxo de mensagens deve possuir de modo a que se possa eleger a melhor ferramenta neste âmbito.

A Tabela 2.4 faz uma comparação dos processadores de fluxos de mensagens que foram anteriormente descritos.

Através da análise da Tabela 2.4 pode-se concluir que o Apache Kafka é a melhor ferramenta de todas as que foram estudadas, pois só não suporta monitorização por omissão.

A ferramenta ZeroMQ é estável, já vem com monitorização por omissão e suporta *clustering*, mas não tem funcionalidades críticas como a garantia de entrega ou armazenamento de mensagens.

RabbitMQ não suporta *clustering*, que é uma função importante neste tipo de aplicações.

Tabela 2.4: Tablela comparativa dos processadores de fluxos de mensagens estudados

	Apache Kafka	ZeroMQ	RabbitMQ	Nats.io
Garantia de Entrega	X		X	X
Armazenamento persistente de mensagens	X		X	
Estabilidade	X	X		X
Monitorização por omissão		X	X	X
Suporte a Clustering	X	X		
Suporte a particionamento e replicação de tópicos	X		X	

Por fim, o Nats.io suporta garantia de entrega, mas guarda as mensagens em memória, e como também não suporta *clustering*, se a máquina em que está a ser executado tiver uma falha de memória, as mensagens são perdidas.

Concluindo, o Apache Kafka é a ferramenta que reúne todas as principais funcionalidades que uma ferramenta de processamento de mensagens deve ter.

2.5 Flafka

Um dos requisitos da plataforma de monitorização é a inserção dos dados recolhidos do sistema monitorizado numa base de dados que faz parte do componente `TMA_Knowledge`. Esta necessidade fez com que houvesse a necessidade de haver um serviço que cumprisse com essa função.

Esse serviço foi implementado integrando o *Apache Flume* com o *Apache Kafka*. O *Apache Flume* é uma ferramenta confiável que coleta, agrega e transfere eficiente grandes volumes de dados para um armazenamento central [7]. A sua arquitetura é simples e flexível, o que confere ao *Apache Flume* robustez e tolerância a falhas. A Figura 2.20 ilustra a arquitetura do *Apache Flume*.

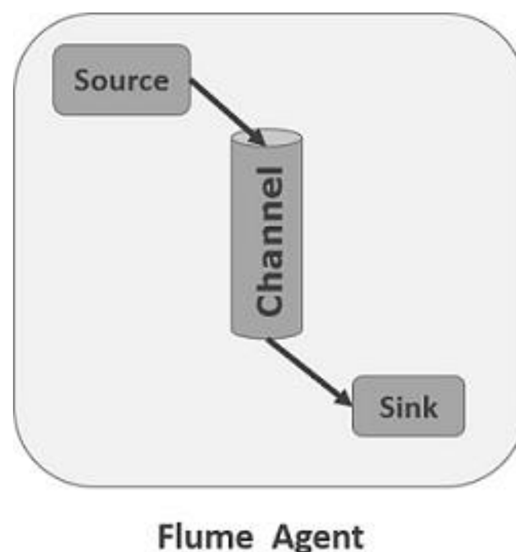

Figura 2.20: Arquitetura Apache Flume (fonte [53]).

Como ilustrado na Figura 2.20, o *Apache Flume* é composto por três componentes principais que são a *source*, o *channel* e o *sink*.

A *source* é o componente que é responsável por receber os dados e transfere-os para um ou

mais *channels* sob a forma de eventos. O *Apache Flume* suporta várias *sources* incluindo *sourcers* para recolher dados do *Twitter* ou do *Apache Kafka*.

O *channel* é um armaznamento transitório que recebe os eventos recolhidos pela *source* e faz *buffer* deles até serem processados pelo *sink*. Resumidamente, o *channel* tem a função de "ponte" entre a *source* e o *sink* do *Apache Flume*. Existem *channels* que conseguem suportar varios tipos de *sources* ao mesmo tempo.

O *Apache Flume* suporta vários tipos de *channels* entre os quais a utilização de um tópico do *Apache Kafka* ou o armazenamento da informação num ficheiro de texto.

Por fim, o *sink* armazena os dados em sistema de armazenamento tais como uma base de dados ou Hadoop Distributed File System (HDFS). Este componente processa os eventos que estão armazenados nos *channels* e armazena-os no *sink* configurado.

A integração do *Apache Kafka* com o *Apache Flume*, a *Cloudera* chamou de Flafka.

Esta integração traz muitas vantagens em relação à API do *Apache Kafka* [47]:

- Integração das *sources* do *Apache Flume* como produtores ou produtores do *Apache Kafka*;
- Integração dos *sinks* do *Apache Flume* como consumidores ou produtores do *Apache Kafka*;
- Processamento e transformação de dados em tempo-real.

Na implementação da plataforma, o *Apache Flume* a *source* do *Apache Flume* foi usada como consumidor de um tópico do *Apache Kafka* e o *sink* insere os dados na base de dados.

2.6 Armazenamento Persistente de Dados

Os *containers* não guardam o seu estado quando param de serem executados, o que faz com que toda os dados que estejam guardados no sistema de ficheiros do *containers* sejam perdidos. Quando esse *container* volta a ser executado, este inicia completamente "limpo".

Os volumes resolvem este problema. Um volume é uma diretoria, que pode conter dados de antemão que é acessível pelos *containers*. Este tipo de recurso tem um tempo de vida maior que o *container* sendo que conserva toda a informação armazenada nessa diretoria mesmo que o *container* seja reiniciado.

Este tipo de recurso foi necessário utiliza na plataforma no componente `TMA_Knowledge` já que é necessário preservar os dados armazenados na base de dados mesmo que, por alguma razão, o *pod* do MySQL fique indisponível.

No Kubernetes, os volumes podem ser mapeados nos *containers* segundo três modos de acesso:

- *ReadWriteOnce* - o volume é montado para leitura e escrita apenas para um *pod*.
- *ReadOnlyMany* - o volume é montado apenas para leitura para vários *pods*.
- *ReadWriteMany* - o volume é montado para leitura e escrita por vários *pods*.

Nem todos os tipos de volumes que o Kubernetes suporta podem ser configurados com os modos acessos descritos anteriormente. Outra aspeto a considerar é que um volume do Kubernetes só pode configurado com apenas um dos três modos de acesso.

O Kubernetes suporta vários tipos de volumes cerca de vinte e sete tipos diferentes de volumes, em que cada um tem o seu tipo de armazenamento. Neste plataforma optou-se por utilizar *block storage*, pois obtém um melhor desempenho para grandes quantidades de dados, os dados podem ser guardados em múltiplas localizações e escala em maior número do que as restantes tecnologias como o *file storage*.

De todos os volumes que o Kubernetes os volumes que suportam apenas o *Ceph* e o *GlusterFS* suportam o tipo de armazenamento escolhido. No entanto, o *GlusterFS* foi desenhado para fornecer *file storage* e não está otimizado para fornecer *block storage*, por isso o seu desempenho no que diz respeita ao armazenamento de base de dados fica aquém do *Ceph*.

O *Ceph* é um sistema de armazenamento desenhado para ser distribuído fornecendo bom desempenho, confiabilidade e escalabilidade, O *Ceph* não tem pontos únicos de falhas, é escalável até ao Exabyte e tem um bom nível de auto-gestão e de monitorização [44].

O *Ceph* é constituído por quatro componentes, cada um com a sua função:

- Monitores de *Cluster* que tratam da monitorização dos restantes componentes;
- Servidores de Metadata que armazenam os ficheiros em *cache*;
- Sistemas de armazenamento de objetos (OSD) que armazenam os dados no sistem de ficheiros;
- *Gateways RESTFul* que expõe a camada de armazenamento através de uma interface.

O método de armazenamento usado pelo *Ceph* é semelhante ao utilizado pelo *RAID0* em que é usado o algoritmo *Round-Robin* para armazenar os blocos de dados. Os blocos de dados com maior frequência de alterações são replicados por mais OSDs se existirem, o que possibilita o balanceamento entre carga entre eles.

BlueStore é o tipo de armazenamento recomendado, pois é o mais personalizável e é o que fornece menor latência, por isso, é o usado por omissão. Quando uma aplicação escreve um bloco de dados, esse bloco é replicado por todos os OSDs.

Os servidores de metadata podem ser dinamicamente escaláveis para que haja balanceamento de carga, o que reduz a latência na atualização dos dados armazenados.

O processo de controlo de acesso aos dados armazenados no *Ceph* utiliza o mecanismo de exclusão mútua.

Quando um ficheiro é acedido por múltiplos clientes com múltiplos escritores ou um conjunto de escritores e leitores, o Servidor de Metadata nega o acesso a qualquer operação em *cache*, forçando que apenas um tipo de operação seja executada, o que faz com que as ações sejam sincronizadas.

Quando um cliente está a aceder a um bloco de dados para executar qualquer operação, esse cliente adquire um acesso exclusivo a esse bloco. Finalizadas as operações, é desbloqueado esse bloco de dados para futuras ações. Todo o controlo de concorrência no armazenamento dos blocos de dados é feito pelo OSD.

Para além disso, a utilização do *Ceph* para armazenamento de dados que estão armazenados na base de dados tem as seguintes vantagens:

- O *Ceph* tem facilidade em mover os dados entre as suas réplicas mesmo que estejam a executar em máquinas diferentes.
- O *Ceph* paraleliza as operações de consulta, inserção e remoção de dados entre as várias réplicas desde que não sejam referentes ao mesmo bloco de dados.
- O *Ceph* replica automaticamente os dados fornecendo assim redundância de armazenamento.
- O *Ceph* suporta *snapshots* que são rápidos de se gerarem e não afetam o desempenho da ferramenta.

Esta página foi propositadamente deixada em branco.

Capítulo 3

Análise de Requisitos

.

Neste capítulo vão ser descritos de forma resumida os requisitos funcionais e não funcionais da plataforma e também algumas restrições que condicionaram o processo de desenvolvimento.

Os requisitos vão ser mostrados formalmente sob a forma de casos de uso, pois é o tipo de definição de requisitos mais popular por várias razões. Algumas destas são a sua flexibilidade de aplicação a sistemas com as mais variadas funções e a simplicidade e clareza.

Outras razões importantes para esta escolha de definição de requisitos é o destaque da identificação de problemas e incoerências, bem como a definição de pré-condições e pós-condições que auxiliam, mais tarde, na fase de testes, a testar as funcionalidades do sistema.

Os requisitos da plataforma de monitorização a desenvolver podem dividir-se em três grandes tarefas: monitorização em tempo-real, avaliar e verificar a configuração dos sistemas monitorizados e armazenar o histórico de eventos.

De forma a não sobrecarregar este capítulo neste documento, o leitor poderá consultar a especificação completa, como casos de uso, no Documento de Requisitos em anexo. As informações a seguir pretendem apresentar uma descrição resumida dos requisitos, fornecendo apenas a informação essencial ao leitor.

Na categoria da monitorização em tempo real foram especificados catorze requisitos funcionais. Na categoria de avaliar e verificar a configuração dos sistemas monitorizados foram especificados três requisitos principais e na categoria de armazenamento do histórico de eventos foram especificados quatro requisitos. Com base no número de requisitos acima descritos, foram especificados vinte e um requisitos funcionais para implementar no próximo semestre.

Os requisitos não funcionais, também conhecidos como atributos de qualidade, servem para definir um limite mínimo de um determinado parâmetro, sobre o qual o sistema pode ser avaliado. A definição desse limite pode ter impacto no sistema todo ou apenas numa parte dele.

Os requisitos não funcionais no contexto deste projeto são: desempenho, escalabilidade e elasticidade. Os *KPI* de todos estes requisitos não funcionais estão descritos em mais detalhe no Documento de Requisitos que se encontra no Anexo A deste relatório.

Por fim, este projeto também tem algumas restrições que são divididas em restrições do projeto e restrições do produto. Uma vez mais, está feita uma descrição detalhada de

ambos os tipos de restrições no Documento de Requisitos que se encontra no Anexo A deste relatório.

Capítulo 4

Plataforma Escalável de Monitorização

Nesta capítulo, irá ser apresentado, em primeiro lugar, a arquitetura geral da plataforma. Depois é apresentada e justificada a escolha das ferramentas para cada necessidade da plataforma e, por fim, é apresentada e explicada a arquitetura detalhada da plataforma.

4.1 Arquitetura Geral da Plataforma

A plataforma de monitorização desenvolvida neste estágio pertence a uma plataforma chamada `Trustworthiness Monitoring and Assessment Platform (TMA)` que tem como principal objetivo a execução de adaptações automáticas ao ambiente que está a monitorizar. Todos os componentes desta plataforma foram desenhados para serem desenvolvidos sob a forma de microserviços o que faz com que cada componente possa ser desenvolvido, testado e escalado de forma independente. A Figura 4.1 ilustra a arquitetura da plataforma TMA.

O perímetro a tracejado delimita os componentes desenvolvidos neste estágio. O componente `TMA_Monitor` fornece uma interface genérica que pode ser usada de duas maneiras: pelas *probes* que estão instaladas em diferentes tecnologias que enviam os dados sobre os eventos e medidas recolhidas desses mesmos sistemas ou pelo utilizador, ao inserir dados manualmente.

O `TMA_Monitor` consiste numa interface *Restful* que recebe pacotes HTTP com o comando *POST*. Cada mensagem *POST*. Cada mensagem deve seguir o formato *JSON* de acordo com o *template* definido para este projeto.

Se os dados provindos das *probes* não estiverem no formato do JSON *schema* defenido pelo projeto, estes são descartados pelo `TMA_Monitor`. Por outro lado, se os dados forem validados pelo `TMA_Monitor`, este encaminha os dados para o componente *FaultTolerantQueue* que é responsável pela comunicação entre todos os componentes do TMA.

Dentro do componente *FaultTolerantQueue* existe um serviço que é responsável por fazer operações de extract,transform,load (ETL) ficando, assim, os dados devidamente formatados para serem armazenados no componente `TMA_Knowledge`.

O componente `TMA_Knowledge` é constituído por um gestor de base de dados e por um volume de armazenamento por blocos onde é guardada, de forma persistente, toda a informação gerada pelos componentes do TMA. Por se tratar de *containers*, todos os dados

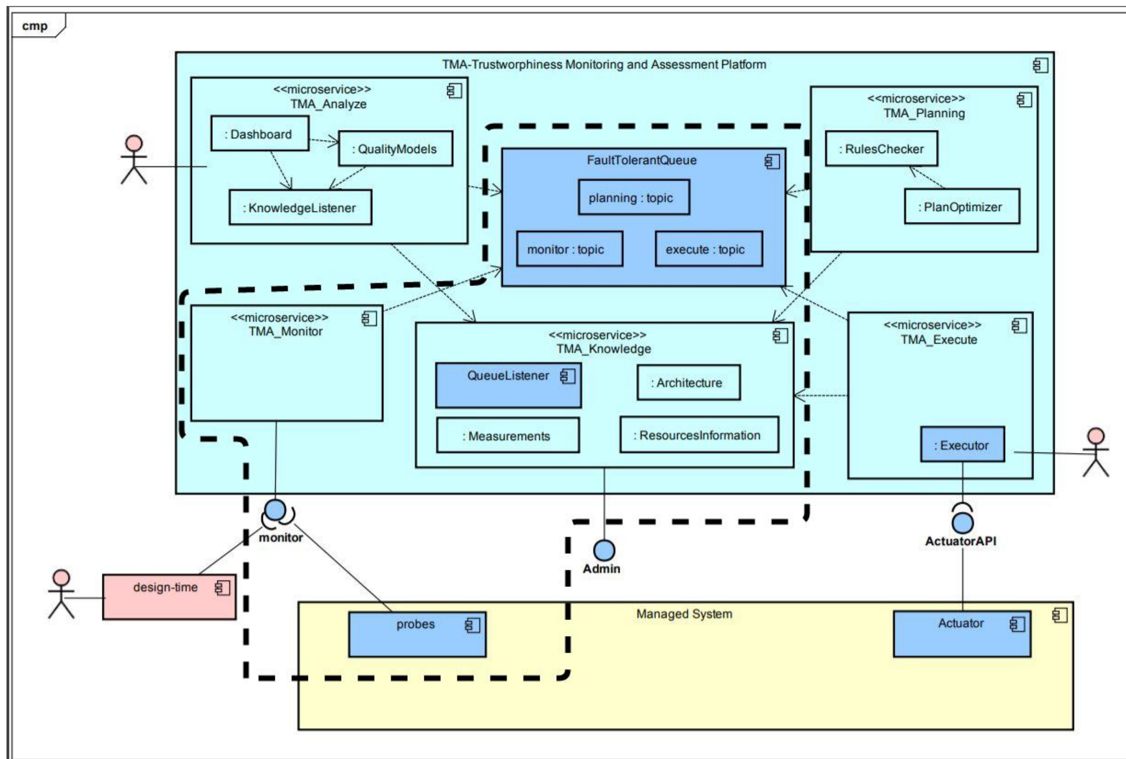

Figura 4.1: Arquitetura geral da plataforma a desenvolver.

precisam de ser armazenados de forma persistente para que não sejam apagados ou ficarem indisponíveis quando o *container* for apagado devido a qualquer perturbação.

4.2 Arquitetura Detalhada da Plataforma

As escolha das ferramentas a usar é uma etapa muito importante que deve ser ponderada cuidadosamente de modo a que seja possível implementar todos os requisitos. Para este projeto, foi necessário tomar várias decisões para as várias necessidades da plataforma:

- Construtor de *containers* – Para que seja possível executar todos os componentes da arquitetura em *containers*;
- Gestor de *containers* – Para tratar da escalabilidade e da comunicação dos *containers* onde estão a ser executados os componentes da arquitetura;
- Processadores de fluxos de mensagens – Para enviar os dados normalizados pelo `TMA_Monitor` para o `TMA_Knowledge`;
- Microframework para desenvolvimento do `TMA_Monitor` em Python;

As ferramentas adotadas para cada um destes aspetos foram as melhores ferramentas que foram comparadas no Capítulo 2. Por isso, para construtor de *containers* foi escolhido o Docker, para gestor de *containers* foi escolhido o Kubernetes, para Processador de fluxos de mensagens foi escolhido o Apache Kafka e o `TMA_Monitor` vai ser desenvolvido em Flask.

Esta visão mais detalhada da arquitetura serve para explicar em detalhe como vai funcionar a solução proposta. A Figura 4.2 ilustra a arquitetura detalhada da solução.

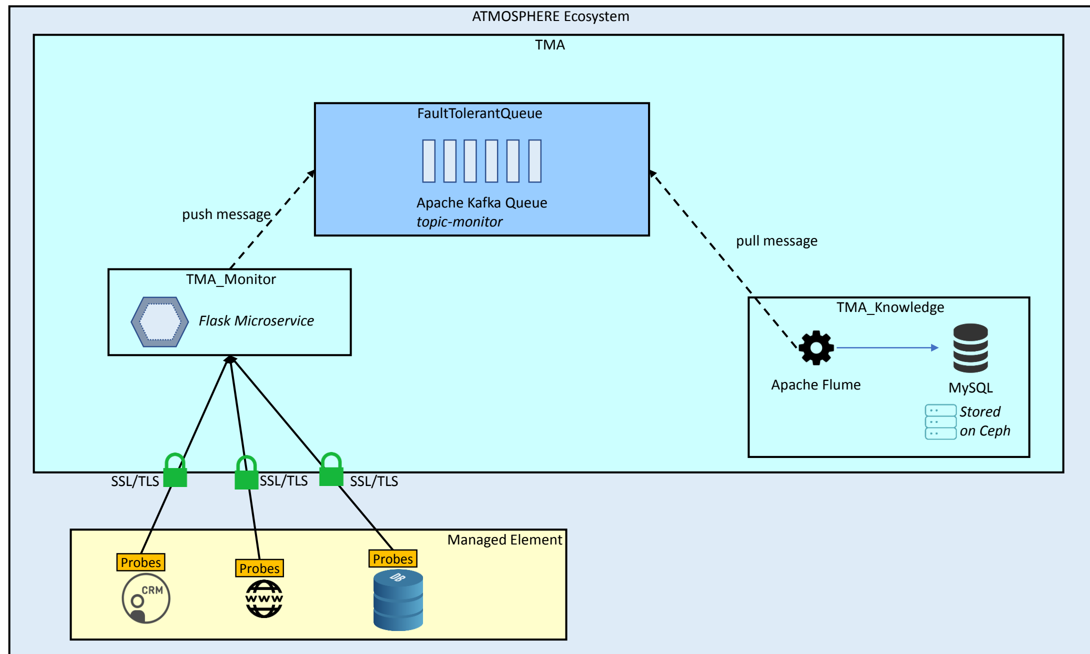

Figura 4.2: Arquitetura detalhada da solução.

O funcionamento da solução começa pela recolha por parte das *probes* dos dados pretendidos que são enviados num pacote HTTP *POST* num formato *JSON* de acordo com o *template* do projeto para o *endpoint* do `TMA_Monitor`. Quando recebe os dados, o `TMA_Monitor` verifica se a estrutura e os dados da mensagem recebida são válidos, se forem válidos, esses mesmos dados são enviados por um tópico do Apache Kafka chamado topic-monitor. Por outro lado, se os dados não forem válidos, são descartados pelo `TMA_Monitor`.

Quando a mensagem enviada pelo tópico chega ao *Broker* do Apache Kafka, este envia os dados normalizados para o Apache Flume. O Apache Flume é responsável por guardar os dados normalizados no `TMA_Knowledge`.

Por fim, o `TMA_Knowledge` é constituído pelo Database Management System (DBMS) MySQL e pelo um volume persistente Ceph, onde é guardada toda a informação da base de dados.

De salientar que todas as ferramentas que foram anteriormente referidas são executadas em *containers* Docker que, por sua vez, constituem um *pod* no Kubernetes, de modo a que seja possível escalar os componentes independentemente, respeitando o conceito de microserviços.

4.2.1 Detalhes de Implementação

Os detalhes de implementação de todos os componentes desenvolvidos neste estágio vão ser descritos de seguida.

Antes de começar o desenvolvimento dos componetes que fazem parte desta plataforma foi necessário perceber todo o processo de criação de um *cluster* Kubernetes. Um *cluster* Kubernetes, como já foi referido na subsecção 2.3.2 do Capítulo 2, foi necessário configurar uma máquina para ser a *Master* e dois *Workers*.

Na máquina *Master* é necessário executar o serviço *kubeadm* para inicializar todos os serviços necessários que dizem respeito à máquina *Master* do *cluster*. No fim da sua execução, o serviço *kubeadm* retorna o comando que é necessário executar em cada um dos nós *workers* para que estes passem a pertencer ao *cluster*.

O último passo da incialização do *cluster* prende-se com o facto de ser necessário instalar um *plug-in* de rede para que os *pods* sejam capazes de comunicar entre si. O *plug-in* utilizado para esse efeito foi o *Flannel*. O *Flannel* é um dos *plug-ins* mais fáceis de configurar, pois basta executar apenas um ficheiro *yaml* para esse efeito, mas também não afeta de forma significativa a comunicação entre os *pods* do *cluster*[51].

`TMA_Monitor`

O `TMA_Monitor` foi desenhado e desenvolvido para ser executado num *pod* num *cluster* Kubernetes. Este componente é responsável por fornecer um *endpoint* para onde as *probes* enviam os dados recolhidos do sistema que estão a monitorizar.

Para que o `TMA_Monitor`, tal como todos os outros componentes da plataforma desenvolvida neste estágio, sejam executados corretamente num *cluster* Kubernetes foi necessário gerar uma imagem Docker e um ficheiro *yaml*.

A criação da imagem Docker é feita a partir de um ficheiro chamado *Dockerfile*, onde são colocadas todas as instruções necessárias para que o `TMA_Monitor` seja executado de forma correta, incluindo os comandos para a instalação de todas as dependências e a própria inicialização do `TMA_Monitor`. Como toda a informação enviada das *probes* para o `TMA_Monitor` é feita através do protocolo Hyper Text Transfer Protocol Secure (HTTPS), neste *Dockerfile* é também copiado o certificado digital gerado para o `TMA_Monitor`. Para a criação da imagem Docker, foi feito um *script* em *bash* para agilizar todo este processo.

O ficheiro *yaml* contém todas as especificações de um objeto do Kubernetes a incializar. Este ficheiro *yaml* inclui instruções como, por exemplo, a imagem Docker do `TMA_Monitor` descrita anteriormente e também a indicação dos portos necessários para a comunicação com o *pod* do `TMA_Monitor`.

O `TMA_Monitor` é incializado no Kubernetes como sendo um *StatefulSet*, o que faz com que, sempre que seja inicializado no Kubernetes, o respetivo *pod* tenha sempre o mesmo nome. Para que serviços que não estejam a executar no *cluster* do Kubernetes possam aceder ao *endpoint* do `TMA_Monitor`, o serviço deste componente é configurado no Kubernetes como *NodePort*, o que faz com que este microserviço seja acedido através do IP do *Master* do *cluster* Kubernetes.

Dentro do *container*, de modo a aumentar o seu desempenho, o `TMA_Monitor` é executado num servidor Web Server Gateway Interface (WSGI) chamado *Gunicorn*. Em relação ao código do `TMA_Monitor`, este foi desenvolvido usando a *micro-framework Flask* de modo a que seja possível implementar um serviço *web* e a disponibilização de um *endpoint*.

O `TMA_Monitor` apenas recebe *requests* HTTP com o comando *POST*. Se receber um *request* com o comando *GET*, este retorna uma mensagem a informar que não suporta tal comando HTTP.

Quando recebe um *request* com o comando HTTP correto no seu *endpoint*, o `TMA_Monitor` começa por fazer a validação dos dados recebidos com o *schema* do projeto. Se os dados forem inválidos, o `TMA_Monitor` descarta os dados e retorna uma mensagem com o número de erros encontrados e a respetiva descrição.

Por fim, se os dados forem válidos, é inicializado um produtor recorrendo à API do *Apache Kafka* para *Python* para que se possa encaminhar os dados devidamente validados para o tópico do *Apacha Kafka* chamado *topic-monitor*.

Por último, todas as operações do `TMA_Monitor` são devidamente registadas em ficheiros de *log*.

`FaultTolerantQueue`

O componente `FaultTolerantQueue` é constituído por três ferramentas que são o o *Apache Zookeeper*, o *Apache Kafka*, que torna-se o componente central do toda a plataforma TMA, e o *Apache Flume*. Para todos estes componentes, à semelhança do `TMA_Monitor`, foram geradas imagens Docker recorrendo a *Dockerfiles* e a ficheiros *yaml* para que também estes componentes sejam executados num *cluster* Kubernetes.

O *Apache Zookeeper* é um componente que não foi alvo de grandes alterações, pois, neste caso, é apenas uma solução de gestão de serviços do *Apache Kafka*. Sendo assim, o *Dockerfile* do *Apache Zookeeper* tem as instruções para o seu download e instalação.

Para este componente, foram necessários criar dois ficheiros *yaml*. Num dos ficheiros *yaml*, é dada a indicação de como este componente é implementado. O *Apache Zookeeper* é implementado no *cluster* Kubernetes sob a forma de um *StatefulSet* para que o nome do respetivo *pod* seja estático. Neste caso, para o *Apache Zookeeper*, torna-se obrigatório que seja mesmo um *StatefulSet*, pois torna muito mais simples a automatização da comunicação com o *pod* do *Apache Kafka*. Ainda neste ficheiro *yaml*, é indicada a imagem Docker gerada com o *Dockerfile*, algumas configurações de inicialização do serviço do *Apache Zookeeper* e, por fim, também a indicação da diretoria onde será mapeado um volume persistente.

O outro ficheiro *yaml* diz respeito, justamente, à inicialização de um volume persistente do tipo *hostpath*. Este tipo de volume faz com que o conteúdo da diretoria dentro do *pod* seja mapeado para uma diretoria na máquina onde esse *pod* está a executar. Neste ficheiro *yaml*, foi declarado este tipo de volume do Kubernetes para que, caso o *pod* do *Apache Zookeeper* pare a sua execução, os seus dados mais importantes não sejam perdidos.

O *Apache Kafka* é o componente central, não só da plataforma de monitorização, mas também de toda a plataforma TMA. Os ficheiros necessários para a implementação deste componente são semelhantes aos do *Apache Zookeeper*.

Para o *Apache Kafka*, foi necessário criar uma imagem Docker recorrendo a um ficheiro *Dockerfile*, onde estão presentes as instruções para o download e a instalação desta ferramenta. Foram ainda criados dois ficheiros *yaml* para iniciar o *Apache Kafka* no *cluster* Kubernetes. Estes dois ficheiros têm funções semelhantes aos criados no caso do *Apache Zookeeper*.

Um dos ficheiros *yaml* tem como função criar o *StatefulSet* do *Apache Kafka* no *cluster* Kubernetes. Neste caso, também se torna obrigatório utilizar este recurso do Kubernetes para que o endereço do *broker* do *Apache Kafka* seja estático, pois é a ferramenta central de toda a plataforma TMA. Se o *Apache Kafka* não tivesse um endereço estático, era necessário alterar o código de todos os componentes da plataforma TMA. Ainda neste ficheiro, está presente a declaração da imagem Docker referente ao *Apache Kafka*, todas as configurações de inicialização deste serviço, incluindo o endereço do serviço do *Apache Zookeeer*, e, por fim, a declaração do volume persistente para este *pod*.

No segundo ficheiro *yaml*, é definido o volume persistente no qual está mapeada a dire-

toria que contém os ficheiros com a informação mais importante do Apache Kafka. Este volume é útil para evitar a perda de informação, caso o *pod* do *Apache Kafka* sofra alguma perturbação que leva à indisponibilidade do serviço.

Por fim, a última ferramenta que faz parte do componente `FaultTolerantQueue` é o *Apache Flume*. Para esta ferramenta, não foi necessário gerar tantos ficheiros como nas ferramentas anteriores, devido ao facto de ser a ferramenta mais personalizável pelo utilizador desta plataforma, já que tem dois modos de operação, nomeadamente, o modo normal e modo de teste.

O modo normal de operação do *Apache Flume* consiste em recolher as mensagens escritas no tópico *topic-monitor* do *Apache Kafka*, fazer todo o processo de ETL dos dados e gerar e executar as *queries* Standard Query Language (SQL) na base de dados presente no componente `TMA_Knowledge`.

Para implementar este modo de operação, foi necessário desenvolver uma *source* para o *Apache Flume* e criar um ficheiro de configuração para implementar as funcionalidades enunciadas anteriormente. A *source* é onde se faz as operações de ETL. Esta *source* foi desenvolvida em Java e por sua vez compilada para um ficheiro *jar* de forma a que possa ser incluída nas bibliotecas do *Apache Flume*.

O código desta biblioteca começa por recolher as mensagens do tópico *topic-monitor* do *Apache Kafka* e, por cada observação, é gerado um evento do *Apache Flume* novo. Esse evento possui os campos *probeID*, *resourceID*, *descriptionId*, *time* e *value*. No fim, o novo evento com os campos descritos anteriormente é adicionado a uma lista de eventos para serem consumidos pelo *sink* do *Apache Flume*.

Para incializar o *Apache Flume* neste modo de operação, foi necessário também criar um ficheiro de configuração, onde são configurados a *source*, o *channel* e o *sink*.

Na configuração da *source*, é indicada a biblioteca desenvolvida e descrita anteriormente, assim como o endereço do *Apache Zookeeper* e respetivo porto, o tópico no qual se deve consumir as mensagens (neste caso, o tópico *topic-monitor*) e o nome do *channel*.

Por sua vez, na configuração do *channel*, é necessário indicar o tipo de *channel*. Neste caso, optou-se por um tópico do *Apache Kafka* chamado *queue-listener* e também pelo endereço do *broker* do *Apache Kafka*.

Por fim, para configurar o *sink* do *Apache Flume* neste modo, é necessário escolher o tipo (neste caso, é do tipo Java Database Connectivity (JDBC)), em seguida, deve-se inserir a *connection string*, que é constituída pelo endereço do *pod* do MySQL pertencente ao componente `TMA_Knowledge`, seguido do respetivo porto e do nome da base de dados onde vão ser inseridos os dados. Neste caso, a base de dados chama-se *knowledge*.

Para que o *Apache Flume* consiga executar as *queries* SQL com sucesso, é necessário também configurar o *sink* com o nome e a palavra-passe de um utilizador criado no MySQL. Neste caso, é usado o utilizador *root*. Por fim, resta definir a *query* SQL para inserir os dados dos novos eventos na base de dados.

No modo de teste, o *Apache Flume* faz as mesmas operações de ETL que no modo normal, a diferença prende-se com o facto de que as *queries* SQL são escritas num ficheiro. Devido a esta diferença, foi necessário criar uma biblioteca nova para o *Apache Flume*.

O código é semelhante ao desenvolvido para o modo normal, já que, por cada observação de uma mensagem recebida através do tópico *topic-monitor* do *Apache Kafka*, é gerado um novo evento no *Apache Flume*. Este é constituído por uma *string* que contém uma

query SQL com o comando *INSERT* onde são inseridos os valores dos campos *probeID*, *resourceID*, *descriptionId*, *time* e *value*.

Estando o ficheiro *jar* gerado da biblioteca do modo de teste do *Apache Flume*, foi necessário criar um ficheiro de configuração para que o *Apache Flume* pudesse ser iniciado neste modo de operação. Nesse ficheiro de configuração, também foi necessário configurar a *source*, o *channel* e o *sink*.

Na configuração da *source*, foi necessário especificar que o tipo era o nome da classe da biblioteca que foi descrita anteriormente, assim como o nome do *channel*, o endereço do *Apache Zookeeper* e o tópico de onde vão ser lidas as mensagens, ou seja, o tópico *topic-monitor*.

A configuração do *channel* é exatamente igual ao que foi configurado para o modo de teste, por isso, também foi necessário indicar o tipo de *channel* (neste caso optou-se por um tópico do *Apache Kafka* chamado *queue-listener*) e o endereço do *broker* do *Apache Kafka*.

Na configuração do *sink*, foi necessário indicar também o seu tipo. Como o objetivo é escrever as *queries* SQL num ficheiro, esse tipo é *FILLE_ROLL*. Também foi necessário indicar a diretoria onde irá ser guardado esse ficheiro de texto e o nome do *channel*.

Depois do desenvolvimento dos módulos, foi necessário criar todos os ficheiros necessários para executar o *Apache Flume* no *cluster* Kubernetes. Para isso, começou-se por criar uma imagem base recorrendo a um *Dockerfile* que faz o download do *Apache Flume*. Esta imagem foi disponibilizada no repositório público *Docker Hub* para ser reutilizada na construção de outras imagens Docker.

Com a imagem base criada e publicada no *Docker Hub*, foi criada uma imagem do *Apache Flume* em que o *Dockerfile* apenas tem a referência para utilizar a imagem base. O comando para a construção da imagem Docker está contido num *script bash* para facilitar a sua execução. Para executar o *Apache Flume* num *cluster* Kubernetes, foi necessário também criar um ficheiro *yaml* em que é referenciada a imagem Docker do *Apache Flume* entre outros parâmetros obrigatórios. Neste ficheiro *yaml*, também foi necessário indicar o porto pelo qual o *Apache Flume* comunica com o exterior. Neste caso, é o 9100.

Por fim, ainda no contexto do ficheiro *yaml* do *Apache Flume*, foi declarado um volume persistente que apenas é necessário caso esta ferramenta seja iniciada em modo de teste. Este volume persistente mapeia na máquina onde o *pod* do *Apache Flume* está a executar a diretoria que irá conter o ficheiro cujo conteúdo será as *queries* SQL geradas a partir das mensagens recebidas no tópico *topic-monitor*.

`TMA_Knowledge`

O componente `TMA_Knowledge` é constituído por um DBMS *MySQL* e uma solução de armazenamento persistente por blocos *Ceph*. Para fazer a implementação do *Ceph*, usou-se uma imagem pública *Docker* chamada *docker.avvo.com/ceph-demo:luminous*.

A execução do *Ceph* deverá ser feita numa máquina fora do *cluster* Kubernetes de modo a que a informação armazenada no *TMA_Knowledge* não se perca. Para fazer o *download* e iniciar o *Ceph*, é necessário executar alguns comandos, que estão automatizados num *script bash* para que seja mais prática a sua execução. Esse *script* tem os comandos para instalação do *Ceph* na máquina, o *pull* da imagem Docker do *Ceph* do *Docker Hub* e a sua execução.

Antes da inicialização do MySQL, é necessário criar e configurar a imagem do *Ceph* onde vai ser mapeada a diretoria com todos os dados armazenados na base de dados *knowledge*. Para tornar mais prática a execução de todos os comandos necessários, foi também criado um *script* em *bash* para automatizar este processo. Este *script* tem o comando de criação da imagem e os comandos de extração da chave de autenticação que vai ser necessária para o *pod* do MySQL conectar-se com sucesso ao *container* do *Ceph*.

Com o *Ceph* a executar e com a chave de autenticação gerada, o último passo de configuração do *Ceph* é criar um recurso do Kubernetes chamado *secret*, que garante que a chave é privada para quem aceda ao *cluster* externamente.

Estando o *Ceph* a funcionar corretamente na sua máquina, pode-se iniciar o *MySQL* no *cluster* do Kubernetes. Para isto, foi necessário gerar a imagem Docker através de um ficheiro *Dockerfile*. No ficheiro *Dockerfile*, está definido que a imagem Docker vai ter como base a última imagem referente à última versão do MySQL presente no *Docker Hub*. Para além disto, também estão presentes instruções para a cópia dos *scripts* SQL para dentro do *container*. Esses *scripts* criam a base de dados e também populam as tabelas com alguma informação de exemplo. Para a construção da imagem, existe um *script* em *bash* para que seja mais prático a execução desta operação.

Com a imagem Docker do MySQL gerada, o próximo passo é iniciar o *pod* no *cluster* do Kubernetes. Para isto, foi desenvolvido também um *script* em *bash* que contém todos os comandos necessários para iniciar e configurar a base de dados no MySQL.

O primeiro comando deste *script* prende-se com a criação de um *secret* no Kubernetes que corresponde à palavra-passe de *root* no MySQL. Este tipo de recurso garante que a palavra-passe não é mostrada a quem acede ao *cluster* externamente.

De seguida, é executado o ficheiro *yaml* de criação e configuração do *pod* do MySQL. Neste ficheiro, é indicada a imagem Docker a utilizar, assim como o porto pelo qual deverá ser acedido o serviço MySQL. Neste caso, é o 3306. Ainda neste ficheiro *yaml*, é também necessário definir o valor da palavra-passe de *root* em que, neste caso, é referenciado o nome do *secret* criado anteriormente. Por fim, é definido o tipo de volume persistente, que, neste caso, é o *Ceph*, indicando o IP e o porto da máquina onde o *Ceph* está a executar, assim como a imagem onde os dados vão ser guardados e, por fim, a indicação do nome do *secret* onde está codificada a chave de autenticação para aceder ao *Ceph*.

Os *secrets* no Kubernetes utilizam o algoritmo base64 para fazer a codificação dos dados.

Os últimos dois comandos são executados assim que o *pod* do MySQL esteja no estado "*Running*" e servem para criar a base de dados e respetivas tabelas, assim como inserir alguns dados de exemplo para que essas tabelas fiquem com alguns registos.

Probes

Neste estágio, foram desenvolvidos quatro exemplos de *probes* para validar não só a plataforma de monitorização, como também a plataforma TMA, fornecendo, também, as bibliotecas necessárias para que qualquer utilizador da plataforma consiga desenvolver as suas próprias *probes*.

Para isso, foram desenvolvidas bibliotecas em Python e em C# para lidar com a comunicação com o componente `TMA_Monitor`. Para reduzir a necessidade de configuração do ambiente de desenvolvimento, foram criadas imagens Docker com estas bibliotecas, fazendo que seja fácil de desenvolver novas *probes* que sejam executadas em *containers* Docker. A

seguir, serão descritas em detalhe as duas bibliotecas desenvolvidas e as quatro *probes* desenvolvidas neste contexto.

Começando pela biblioteca para *probes* desenvolvida em Python, esta é constituída por quatro ficheiros, cada um contendo uma classe. Existe um ficheiro para a comunicação com o `TMA_Monitor` que contém o endereço do *endpoint* do `TMA_Monitor` e é responsável pelo envio das mensages para esse componente.

Existe outro ficheiro com a classe `Data` que possui as variáveis que representam o tipo de dados recolhidos e também um *array* de objetos da classe `observations`. Para as observações existe, justamente, a classe `observations`, que contém as variáveis para armazenar o tempo em que a observação foi recolhida e o respetivo valor.

Por fim, o último ficheiro contém a classe `message`, onde existem as variáveis para armazenar os valores de *probeID*, *resourceID*, *messageID*, *sentTime* e um *array* de objetos da classe `Data`.

Como foi dito anteriormente, todos estes ficheiros podem ser incluídos numa imagem Docker para que não haja necessidade de fazer configurações. Para isso, foi criado um *Dockerfile* que copia todos os ficheiros descritos anteriormente e também o certificado digital necessário para a comunicação por HTTPS com o `TMA_Monitor`. Para construir esta imagem, existe um *script* em *bash* para ser mais prático todo este processo.

A biblioteca para *probes* em C# tem mais ficheiros, apesar de seguir a mesma estrutura da anterior. Esta biblioteca tem os mesmos ficheiros do que a bibliteca para *probes* em Python com as mesmas funções, ou seja, existe um ficheiro para a comunicação com o `TMA_Monitor`, que contém o endereço do *endpoint* do `TMA_Monitor`, e um objeto é criado quando se envia mensagens para o `TMA_Monitor`.

Existe outro ficheiro com a classe `Data` que possui as variáveis que representam o tipo de dados recolhidos e também um *array* de objetos da classe `observations`. Para as observações, existe, também, a classe `observations`, que contém as variáveis para armazenar o tempo em que a observação foi recolhida e o respetivo valor. Por fim, a classe `message` contém as variáveis para armazenar os valores de *probeID*, *resourceID*, *messageID*, *sentTime* e um *array* de objetos da classe `Data`.

Para além destes ficheiros, foi necessário incluir algumas bibliotecas externas sob forma de uma Dynamic-link library (DLL) para construir a mensagem e para fazer o *logging* do funcionamento da biblioteca. Para o *logging*, foi necessário também criar um ficheiro de configuração para definir a forma como os *logs* são armazenados.

Todos os ficheiros desta biblioteca podem ser compilados na forma de uma DLL para que esta biblioteca possa ser facilmente incluída em qualquer projeto. No entanto, também há a possibilidade de criar uma imagem Docker onde são incluídos todos os ficheiros descritos anteriormente e também o certificado digital necessário para comunicar com o `TMA_Monitor`. Para isso, existe um ficheiro *Dockerfile* que copia e compila todos os ficheiros sob a forma de uma DLL e adiciona o certificado digital do `TMA_Monitor` ao conjunto de certificados de confiança do sistema operativo que está executar dentro do *container*.

Qualquer uma destas imagens Docker pode ser usada como base para outras imagens que contenham *probes* desenvolvidas por qualquer utilizador da plataforma.

Usando estas bibliotecas como base, foram desenvolvidas quatro *probes* com diferentes funções:

- `probe-cs-demo` - Esta *probe* foi desenvolvida em C# para gerar dados aleatórios no

formato do *schema* do projeto.

- `probe-k8s-docker` - Esta *probe* foi desenvolvida em Python para recolher dados de métricas de desempenho de *containers* Docker geridos pelo Kubernetes. É capaz de recolher dados de métricas relacionados com a utilização de CPU, memória e disco.

- `probe-k8s-network` - Esta *probe* foi desenvolvida em Python para recolher dados de métricas de rede dos *pods* a executar num *cluster* Kubernetes. É capaz de recolher valores de métricas, como o *rate* de pacotes de rede recebidos, transmitidos ou descartados por cada *interface* de rede de cada *pod*.

- `probe-python-demo` - Esta *probe* foi desenvolvida em Python para gerar dados aleatórios no formato do *schema* do projeto.

O código da *probe* `probe-cs-demo` é simples, sendo constituído apenas por dois métodos: `main` e `addValues`. No método `main`, são declarados e iniciados os objetos para o envio da mensagem para o `TMA_Monitor` e para a mensagem propriamente dita. No método `addValues`, são gerados valores para as observações e adicionados ao *array* de objetos `Data`. Para executar esta *probe*, basta passar como argumento o endereço do *endpoint* do `TMA_Monitor`.

Para além do código, também foram criados todos os ficheiros necessários para executar esta *probe* num *cluster* Kubernetes. Para isso, o *Dockerfile* desta *probe* tem como imagem base a imagem Docker da biblioteca C# descrita anteriormente. Por isso, este *Dockerfile* apenas copia todas as *DLLs* necessárias para fazer o *logging* e a construção da mensagem e também o ficheiro de código descrito anteriormente. Depois, são executados os comandos para a compilação desta aplicação usando a DLL da biblioteca.

Por fim, para iniciar esta *probe* no *cluster* Kubernetes, foi necessário criar um ficheiro *yaml* no qual é necessário indicar a imagem Docker criada anteriormente, para além de outros campos obrigatórios.

A *probe* `probe-k8s-docker` também é constituída por um ficheiro Python, no qual é iniciado um objeto da classe que é responsável pela comunicação com o `TMA_Monitor` e são chamados os métodos para recolher e formatar os dados recolhidos. Nesses métodos, são recolhidos os valores referentes à utilização de CPU, memória e disco. Depois, os valores são formatados e enviados para o *endpoint* do `TMA_Monitor`.

Tal como a *probe* anterior e como as restantes que vão ser descritas a seguir, esta *probe* pode ser executada num *cluster* Kubernetes. Para isso, foi criado um ficheiro *Dockerfile* que tem como base a imagem da biblioteca Python descrita anteriormente e que possui todos os comandos para copiar o ficheiro da *probe* descrita anteriormente. Possui ainda os comandos para a instalação do Docker dentro do *container* que é necessário para que o *container* consiga aceder à API do Docker da máquina local. O último comando deste ficheiro tem como função a execução desta *probe* passando como argumentos o nome do *container* a monitorizar e o *endpoint* do `TMA_Monitor`.

A *probe* onde existem mais pré-requisitos para a sua inicialização é a *probe* `probe-k8s-network`. Devido a todos estes pré-requisitos, foi criado um *script* em *bash* que automatiza a implementação desta *probe* num *cluster* Kubernetes.

De forma a recolher as métricas de rede dos *pods* que estão a executar num determinado *cluster* do Kubernetes, é necessário fazer a implementação do Prometheus, ferramenta analisada no capítulo 2, nesse mesmo *cluster*. Como foi dito a respeito desta ferramenta, é necessário executar um *exporter* no sistema a monitorizar. Ora, essa é a primeira ação

que é feita por este *script*. Depois, é implementado e executado o Prometheus no *cluster* Kubernetes. Para isso, foi necessário criar um *Dockerfile* e um ficheiro *yaml*. O *Dockerfile* apenas copia o ficheiro de configuração do Prometheus para dentro da imagem base. Essa imagem é disponibilizada no *Docker Hub* oficial do Prometheus. O ficheiro *yaml* tem apenas a indicação da imagem Docker a executar, para além dos campos obrigatórios necessários.

Assim que o Prometheus estiver a recolher as métricas de rede, pode-se iniciar a *probe* no *cluster* Kubernetes. Essa inicialização é feita através de um ficheiro *yaml* que tem a imagem Docker da *probe* assim como todos os restantes campos que são obrigatórios.

A imagem Docker desta *probe* é criada através de um *Dockerfile* em que apenas é copiado o ficheiro Python da *probe* e a inicialização da mesma. Para iniciar a *probe*, é passado como argumento de entrada o *endpoint* do `TMA_Monitor`. A imagem Docker desta *probe* tem como base a imagem que contém os ficheiros da biblioteca Python.

O código desta *probe* começa com a inicialização do objeto que é responsável por enviar a mensagem para o `TMA_Monitor`. Depois, são feitos pedidos à API do Prometheus para recolher os valores do *rate* de pacotes de rede recebidos, transmitidos ou descartados por cada *interface* de rede de cada *pod*. Com os dados recolhidos, é criada e devidamente formatada a mensagem e é enviada para o *endpoint* do `TMA_Monitor`.

Por fim, a *probe* `probe-phython-demo` é uma *probe* que serve apenas para demonstração de utilização da biblioteca Python. Para além do código da *probe* propriamente dita, foi criado também um *Dockerfile* e um ficheiro *yaml* para que esta *probe* possa ser executada num *cluster* do Kubernetes.

O ficheiro *Dockerfile* apenas copia para dentro do *container* o ficheiro de código da *probe* e executa o comando de execução da mesma, onde é preciso inserir como argumento de entrada o *endpoint* do `TMA_Monitor`. Esta imagem tem como imagem base a imagem Docker criada com a biblioteca Python. Para fazer a construção da imagem, existe um *script* em *bash* que torna mais prática a criação desta imagem. O ficheiro *yaml* criado apenas tem o nome da imagem gerada desta *probe*, para além dos parâmetros obrigatórios que têm de estar presentes nesse ficheiro.

Em relação ao código propriamente dito desta *probe*, sendo uma *probe* de demonstração, o código é simples. Em primeiro lugar, é criado um objeto da classe `Communication` que é responsável por enviar as mensagens para o *endpoint* do `TMA_Monitor`. De seguida, é chamado o método que cria a mensagem e preenche os *arrays* de `observation` com dados sintéticos. No fim, é enviada a mensagem para o *endpoint* do `TMA_Monitor`.

Esta página foi propositadamente deixada em branco.

Capítulo 5

Validação e Experimentação

Neste capítulo vai ser descrito todo o processo de validação e teste da plataforma desenvolvida neste estágio.

5.1 Plano de Validação da Plataforma

A arquitetura da plataforma e a própria plataforma foram validadas de diferentes formas. No que diz respeito à arquitetura da solução ilustrada pela Figura 4.2, esta foi validada pelos elementos responsáveis pelo desenvolvimento dos restantes componentes da plataforma, bem como por todos os elementos do *consortium* do ATMOSPHERE.

No que diz respeito à implementação da solução propriamente dita, foram planeados e executados alguns testes estruturais aos diferentes componentes desenvolvidos neste estágio. Os testes estruturais não serão muito complexos, pois o código desenvolvido executa operações relativamente simples e, por consequência, fáceis de testar. Por isso, o desafio maior foi a validação dos requisitos não funcionais.

Os requisitos não funcionais foram validados recorrendo aos seguintes métodos:

- Teste de desempenho;
- Teste de escalabilidade;
- Teste de elasticidade;

A plataforma ainda foi validada, através da integração com os outros componentes do ATMOSPHERE desenvolvidos pelos restantes membros do projeto.

Toda a descrição do processo de validação da plataforma irá ser descrita nas próximas secções.

5.2 Testes Estruturais

Testes estruturais são um tipo de teste de software que se centra na avaliação da estrutura do código fonte da aplicação a testar. A utilização deste tipo de teste pressupõe que haja conhecimento desse mesmo código, daí ser feito, essencialmente, pela equipa de desenvolvimento e não pela equipa de testes.

Tendo em conta o tipo de teste considerado, o método de teste de software que foi utilizado neste contexto foi o teste *white-box*. O método *white-box* testa o software de modo garantir que todas as partes do seu código são alcançáveis e executáveis. Existem, essencialmente, duas técnicas para aplicar este método [19]:

- *Control-Flow testing* - Esta técnica utiliza um grafo de *control-flow* como modelo. Neste tipo de teste é feita uma conversão do código do software a testar para um grafo, onde cada nó representa um bloco de código sequencial e as arestas representam a mudança de fluxo entre os nós.

- *Data flow testing* - Esta técnica centra-se nos dados que cada variável armazena utilizado, à semelhança da técnica anterior, utiiliza um grafo como modelo que se chama *data-flow graph*. Neste grafo, cada nó representa o valor que cada variável armazena num determinado bloco sequencial de código e cada aresta representa uma operação que leva a uma alteração no valor da variável.

Neste estágio teste foi usada a técnica de *Control-Flow testing*, pois, como o código executa operações relativamente simples, as variáveis presentes no código não sofrem grandes alterações, pelo que se torna mais importante validar corretamente a lógica da aplicação.

O planeamento e execução deste tipo de testes passou pelas seguintes etapas:

- Definição dos critérios de cobertura;

- Geração dos grafos de *control-flow*;

- Geração dos casos de teste.

- Execução dos testes e recolha dos resultados.

Nas próximas subsecções, todos os passos enunciados anteriormente irão ser descritos mais detalhadamente.

5.2.1 Critérios de Cobertura

Para além da validação da arquitetura e da solução com os restantes componentes do projeto, foram feitos alguns testes funcionais. A primeira fase deste tipo de testes é a definição dos critérios de cobertura.

Os critérios de cobertura são usados para medir qual a percentagem de código que foi executada para um determinado conjunto de testes.

Os critérios de cobertura que foi usado é a cobertura de caminhos que consite em todos os caminhos gerados pelo grafo de *control-flow* são executados, no mínimo uma vez [34].

5.2.2 Grafos de *control-flow*

Nesta subsecção são apresentados os grafos de *control-flow* dos principais métodos dos componentes desenvolvidos neste estágio. Uma descrição mais detalhada da implementação de todos os componentes encontra-se no capítulo anterior.

Os grafos ilustrados pela Figura 5.1 e pela Figura 5.2 representam os métodos presentes no componente `TMA_Monitor`.

O grafo da Figura 5.1 reprensenta o método *process_ message* é responsável filtrar os tipos de HTTP *requests* que chegam ao `TMA_Monitor`. Este método é simples, pelo que tem como número de *McCabe* 2, o que significa que este método tem dois caminhos independentes. Por isso, é basta gerar dois casos de teste, em que cada um percorra um caminho independente para que se possa testar 100% do código deste método.

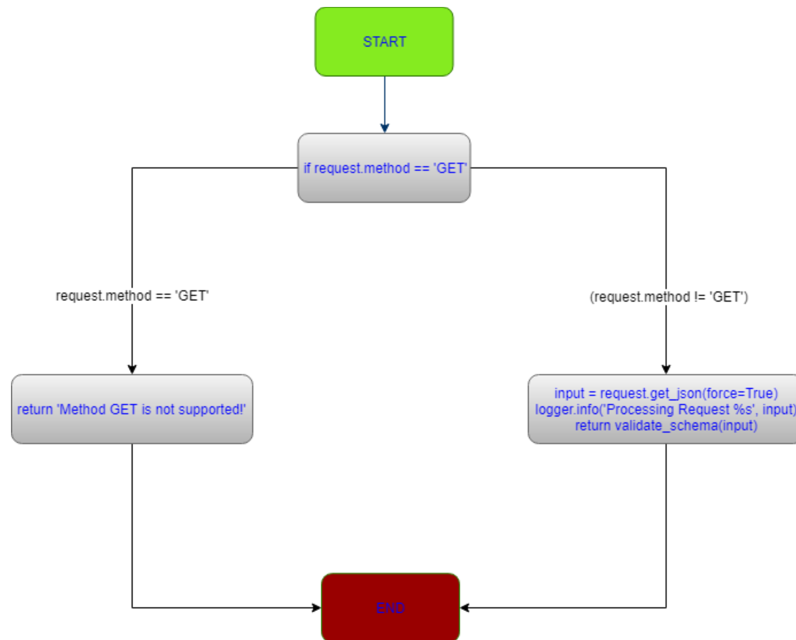

Figura 5.1: Grafo *control-flow* do método *process_ message*.

Por outro lado, o outro método do `TMA_Monitor` é mais complexo do que o anterior. O método *validate_ schema* é responsável por verficar se a mensagem contida no HTTP *request* enviado para `TMA_Monitor` está de acordo com o formato do JSON *schema* definido no início deste estágio.

Este método começa com um tratamento de exceções, o que incrementa a complexidade do método em uma unidade. O que torna este método ainda mais complexo é também o ciclo que é responsável pela verificação da existência de erros na comparação com o JSON *schema* do projeto.

Depois desta comparação, existe uma estrutura de *if/else* que é reponsável pelo seguimeto do *request* consoante este tenha erros ou não. Este tipo de estutura também incrementa a complexidade do método em uma unidade. Posto isto, a complexidade deste método é de 4, o que significa que serão precisos 4 casos de teste para que 100% do código seja testado.

Passando para a implementação das *probes*, só foram gerados os grafos de *control-flow* para o método *main* das mesmas, pois, quer os restantes métodos, quer as bibliotecas desenvolvidas, são constituídos por métodos extremamente simples, pelo que a sua complexidade seria apenas de uma unidade. O que significa que qualquer caso de teste gerado para esses métodos cobriria automaticamente todo o código.

Mais especificamente, o método *main* da *probe* de demonstração escrita em *Python*, representado pela Figura 5.3, não tem uma complexidade muito grande, pois é constituída apenas por um mecanismo de deteção de exceções e um ciclo em que é executado o processo de criar o *request* e enviá-lo para o *endpoint* do `TMA_Monitor`.

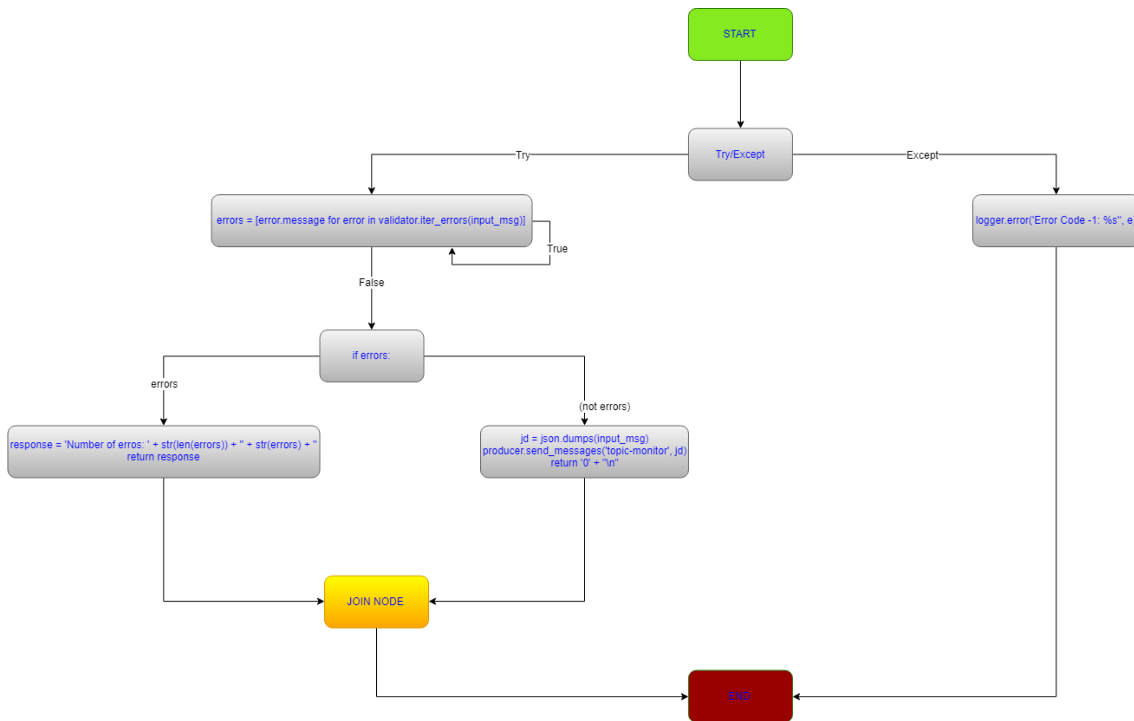

Figura 5.2: Grafo *control-flow* do método *validate_ schema*.

Como tanto a estrutura de captura de exceções como o ciclo incrementam em uma unidade a complexidade do método, este método tem de complexidade 3 unidades, ou seja, será necessário gerar 3 casos de teste, um para cada caminho independente, para testar 100% do código.

Na *probe* de monitorização de CPU, memória e disco de *pods*, escrita igualmente em *Python*, existem dois métodos que são passíveis de serem testados. Esses métodos os grafos que representam esses métodos estão ilustrados na Figura 5.4 e na Figura 5.5.

O método *get_ container_ stats* é o menos complexo dos dois que são alvo de testes, pois apenas tem um ciclo que aumenta a complexidade em uma unidade, o que faz com que a sua complexidade seja de 2 unidades, bastando gerar apenas dois casos de teste para que todo o código deste método seja coberto.

O método mais complexo de todos é claramente o método *format* da *probe* de monitorização de *pods* do *Kubernetes*. Este método é reponsável por formatar todas as métricas, caso algum valor das mesmas tenha sido obtido, de acordo com o *schema* do projeto, por isso, torna-se bastante complexo.

A complexidade deste método deve-se à verificação da existência de valores para as mais variadas métricas. Se houverem valores para essas métricas, é executado um ciclo que são armazenados esses valores num *array*, que no final, contém toda os valores das métricas recolhidas.

Esta sequências de estruturas do tipo *if/else* e de ciclos faz com que este método tenha uma complexidade de 3 unidades, pelo que será necessário gerar 3 casos de testes que percorram, cada um, um caminho diferente para que se possa cobrir 100% do código deste método.

Em relação a outra *probe* desenvolvida no âmbito deste estágio, a *probe* de monitorização de rede para *pods* do *Kubernetes* é semelhante à *probe* de demonstração anteriormente

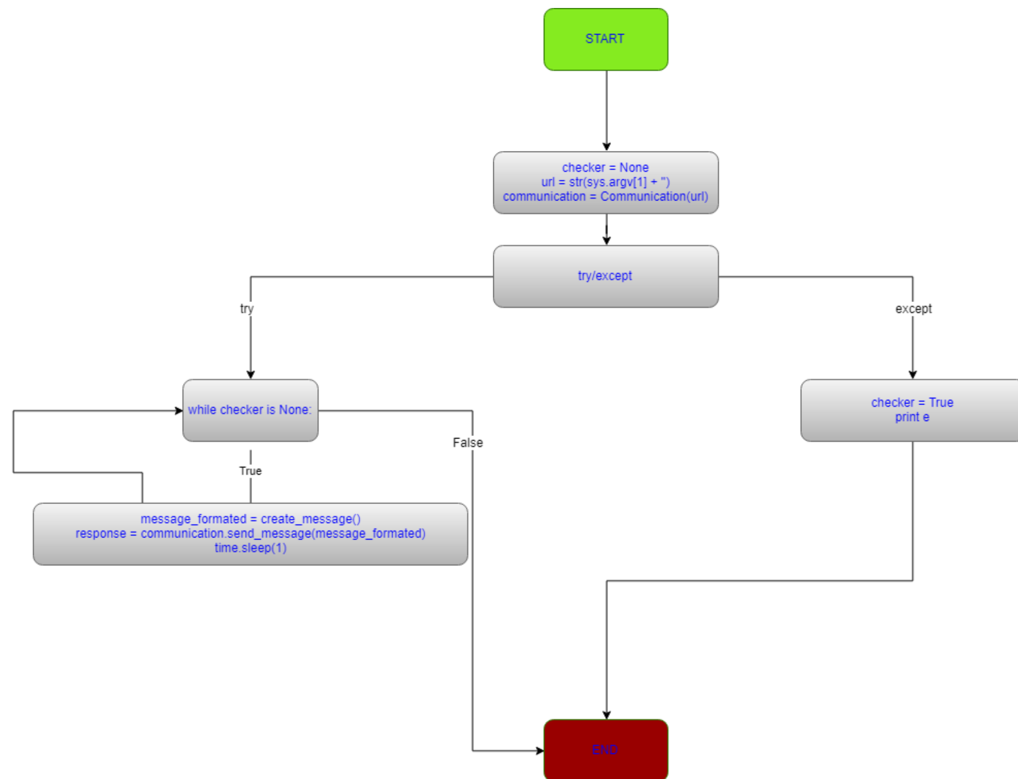

Figura 5.3: Grafo *control-flow* do método *main* da *probe_python_demo*.

descrita.

No método *main* desta *probe*, ilustrado pela Figura 5.6, começa-se com um mecanismo *try/except* para tratamento de exceções, depois entra-se num ciclo de valores de métricas de rede. Dentro desse ciclo, é criado ainda outro ciclo que é responsável por adicionar esses dados ao *request* a enviar para o *endpoint* do `TMA_Monitor`.

Tanto o mecanismo de tratamento de exceções como os dois ciclos incrementam a complexidade deste método em uma unidade, o que faz com que este método tenha 4 unidades de complexidade, pelo que vai ser preciso gerar 4 casos de teste, um para cada caminho independente.

Durante este estágio, foi também desenvolvida uma *probe* na linguagem C# para um dos parceiros do projeto do `ATMOSPHERE`. O grafo de *control-flow* do método *main* desta *probe* é ilustrado pela Figura 5.7.

O método *main* desta *probe* não é muito complexo, pois apenas tem um mecanismo de tratamento de exceções e um ciclo de construção e envio do *request* para o *endpoint* do `TMA_Monitor`.

Devido a este facto, este método tem uma complexidade de 3 unidades, por isso torna-se necessário gerar 3 casos de teste, um para cada caminho independente deste método.

Por fim, os últimos dois métodos alvos de testes dizem respeito aos módulos desenvolvidos para o Apache Flume para os dois modos de operação da plataforma que são, o modo normal e o modo de teste. Os grafos de *control-flow* ilustram o método extração de informação provinda do `TMA_Monitor` de modo a que esta seja inserida na base de dados corretamente.

Os módulos desenvolvidos para o *Apache Flume* têm mais métodos, mas, à semelhança,

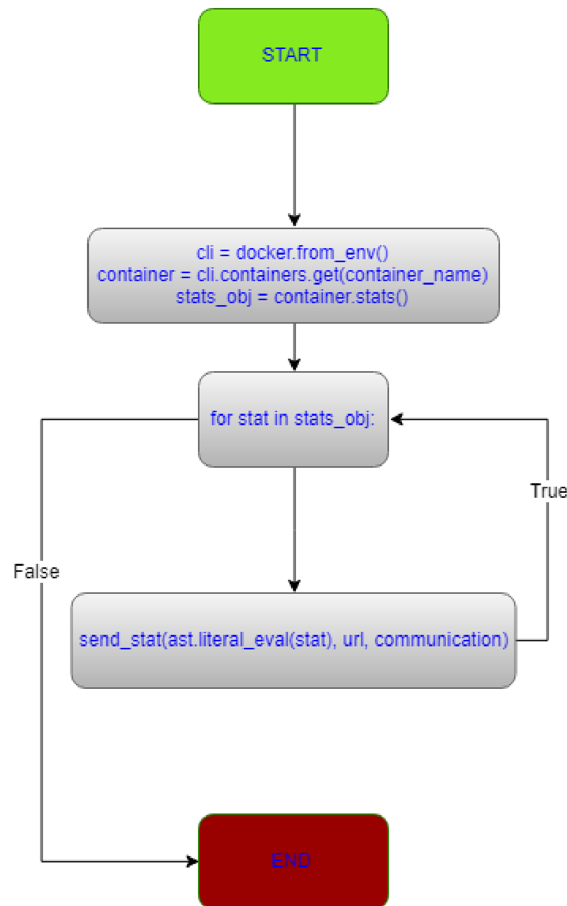

Figura 5.4: Grafo *control-flow* do método *get_ container_ stats*.

do que se passou no código do `TMA_Monitor` e nas bibliotecas desenvolvidas para as *probes*, têm apenas uma unidade de complexidade, o que faz com que qualquer caso de teste passe por todo o código desses mesmos métodos. Daí não serem ilustrados nestes grafos de *control-flow*.

Com base no que foi dito anteriormentem, os grafos de *control-flow* ilustrados pela Figura 5.8 e pela Figura 5.9 representam o método com mais complexidade. Começando pelo grafo de *control-flow* da Figura 5.8, este é composto por 3 ciclos encadeados que servem para retirar todos os campos das mensagens provindas do `TMA_Monitor` e formatar corretamente o evento de saída para que as queries SQL possam inserir corretamente a informação na base de dados.

Devido à sua estrutura, este método tem de complexidade 4 unidades, pelo que será necessário 4 casos de testes, um para cada caminho independente, para que consiga cobrir 100% do código.

Finalmente, o método *parse_ events* ilustrado pela Figura 5.9 tem a estrura igual ao método equivalente para o modo normal, pelo que o número de complexidade e os casos de testes a gerar seja igual ao método anterior, ou seja, 4.

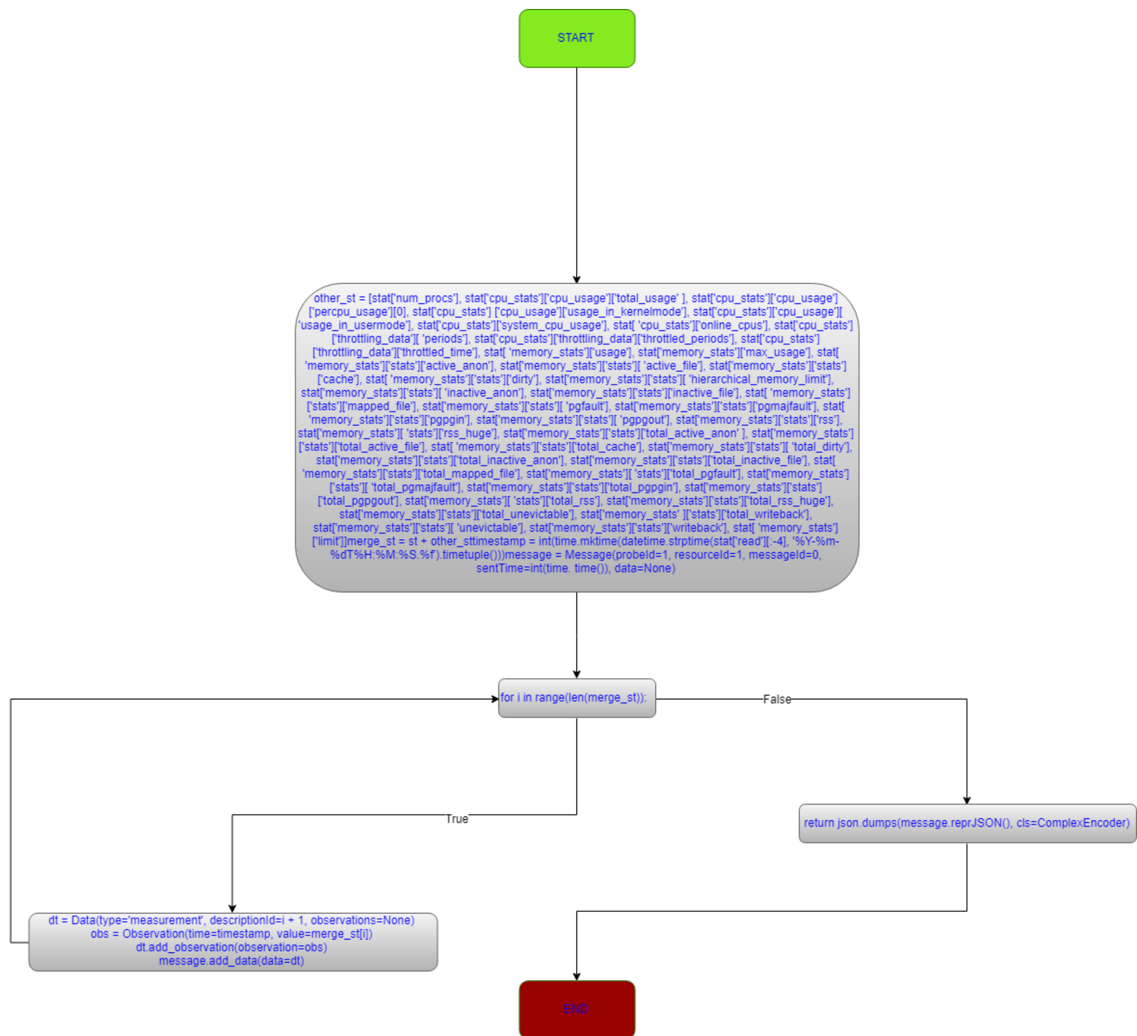

Figura 5.5: Grafo *control-flow* do método *format*.

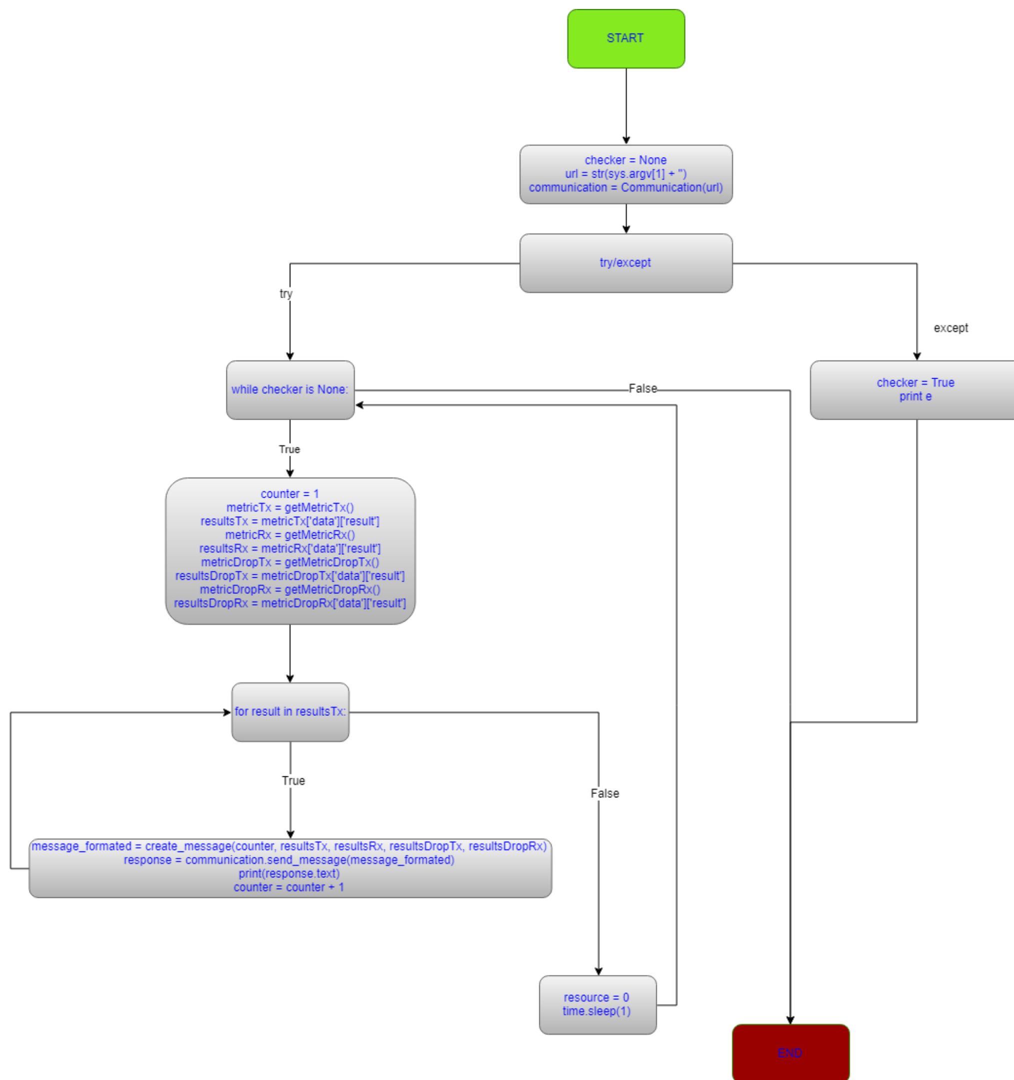

Figura 5.6: Grafo *control-flow* do método *main* da *probe_ k8s_ network*.

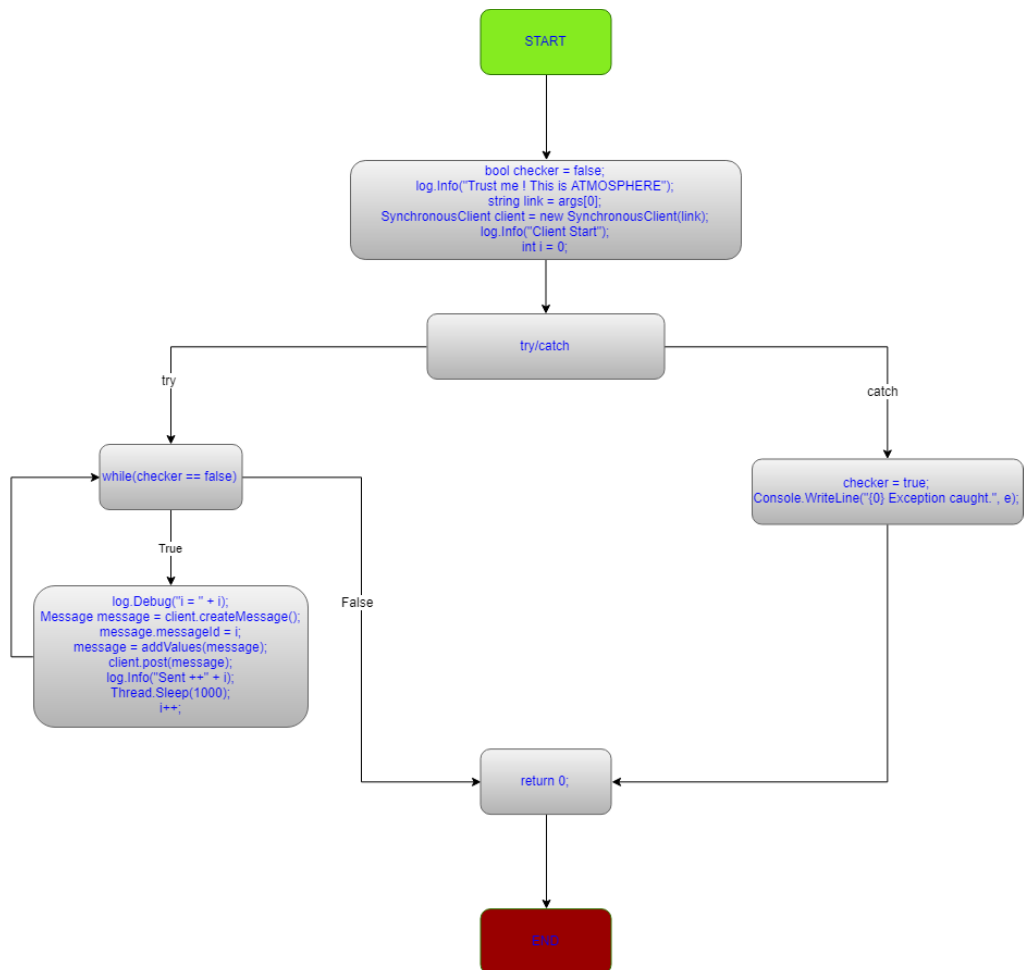

Figura 5.7: Grafo *control-flow* do método *main* da *probe_ cs_ demo*.

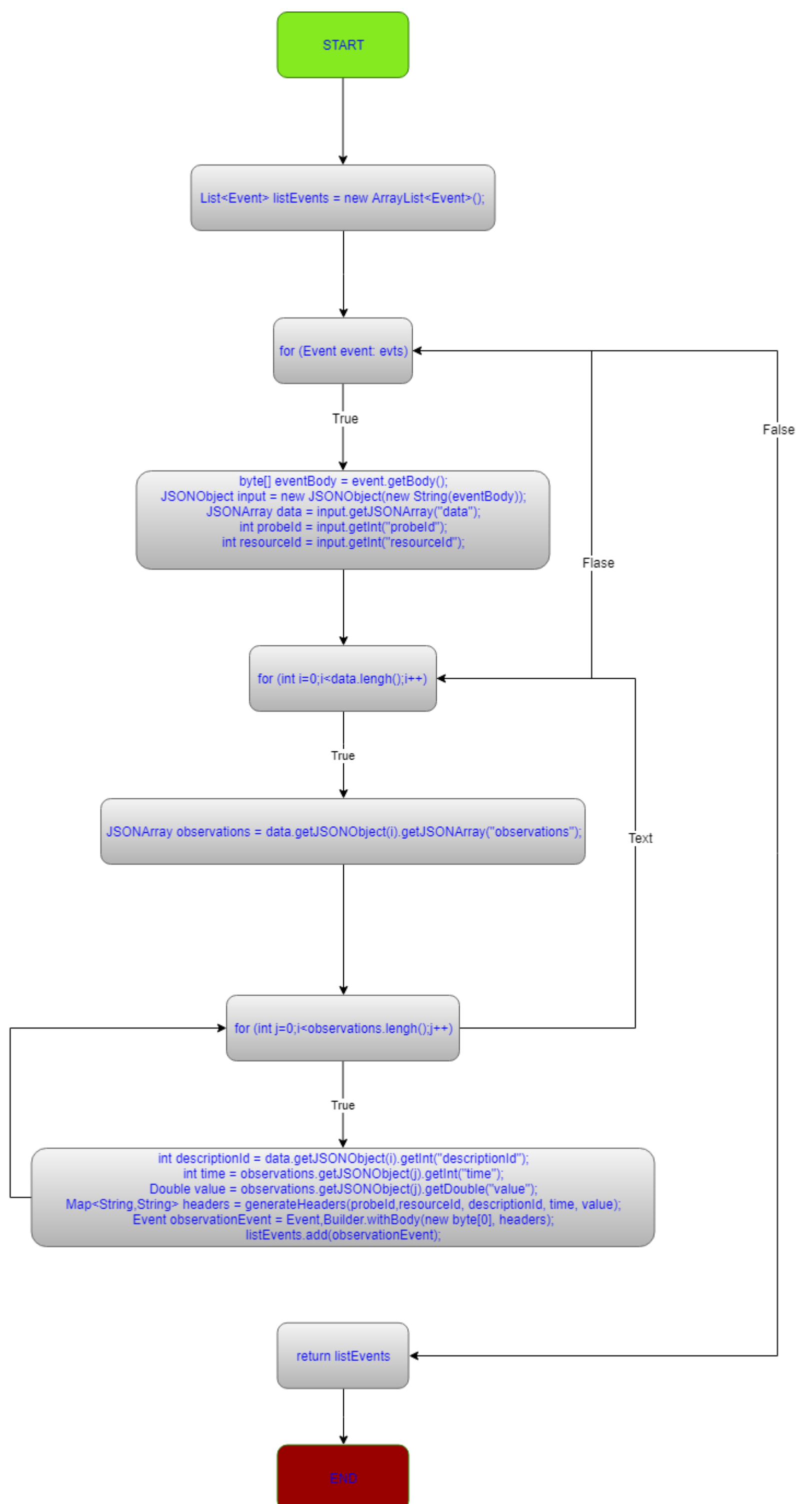

Figura 5.8: Grafo *control-flow* do método *parse_ events* do modo normal.

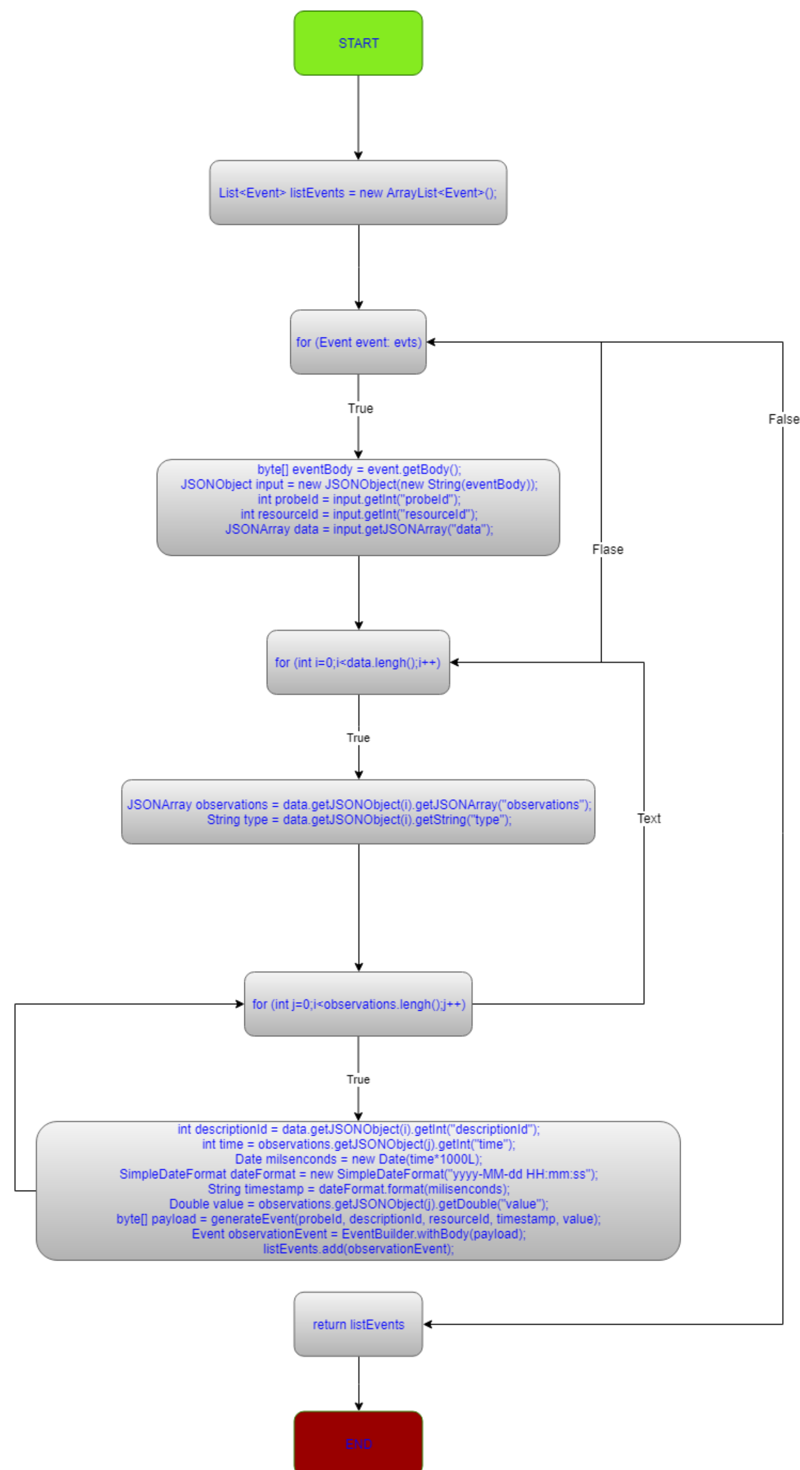

Figura 5.9: Grafo *control-flow* do método *parse_ events* modo de teste.

5.2.3 Casos de teste

Nesta subsecção vão ser descritos os casos de teste usados para testar todos os métodos que foram ilustrados na subsecção anterior pelos grafos de *control-flow*. Na descrição dos grafos de *control-flow* na subsecção anterior foi, para cada um, indicado o número de complexidade. Este número determina o número de caminhos independentes que se podem percorrer para cobrit 100% do código desses mesmos métodos.

Por isso, o número de casos de teste gerados para cada método é igual à sua complexidade com a garantia de que cada caso de teste percorrerá um caminho independente diferente dos restantes.

Com base no que foi dito anteriormente, os casos de teste gerados para o grafo de *control-flow* do método *process_ message* do `TMA_Monitor` ilustrado pela Figura 5.1 são apresentados na Tabela 5.1

O grafo de *control-flow* deste método é bastante simples, por isso, não foi necessário gerar muitos casos de teste. Como este método apenas assenta numa declaração de *if/else*, foi gerado um caso de teste que forçasse o método a executar as instruções do bloco que está dentro do *if* e outro caso de teste em que fossem executadas as intruções que estão dentro do bloco *else*.

Tabela 5.1: Casos de teste para o método *process_ message*

Caso de Teste	Input	Outcome Esperado
1	HTTP Request com o comando GET	Retorno da mensagem "Method GET is not supported"
2	HTTP Request com o comando POST vazio	Log da mensagem "Processing Request"e chama do método validate_schema

O método *validate_ schema* é o método mais complexo do `TMA_Monitor` com uma complexidade de 4 unidades, por isso foi necessário gerar 4 casos de teste. Todos os casos de teste gerados para este método são apresentados na Tabela 5.2

Como este método é o responsável pela validação das mensagens enviadas pelas *probes*, dois *inputs* de dois casos de teste são um *input* de uma mensagem com o formato JSON correta e um *input* com uma mensagem num formato JSON que não é correto.

Os outros dois casos de testes prendem-se com a interrupção da executação do `TMA_Monitor` causada por qualquer sinal, e, por fim, no quarto caso de teste, o *input* é uma mensagem que contém um conteúdo aleatório sem estar no formato de *JSON*.

Passando para os métodos das *probes* e começando pelo método *main* da *probe probe_ python_ demo*, esta *probe* foi desenvolvida com o intuito de gerar dados aleatórios para testar o funcionamento do `TMA_Monitor`, daí ser pouco complexa. Por tudo o que foi dito anteriormente, a complexidade deste método é de 3 unidades, por isso foram gerados 3 casos de testes que estão apresentados na Tabela 5.3.

Como todas as *probes* desenvolvidas neste estágio recebem os *inputs* por argumentos no comando de inicialização das mesmas, todos os casos de teste relacionados com as *probes* prendem-se com *inputs* enviados como argumentos no terminal. Neste caso, os casos de teste também foram facilmente gerados, bastando apenas, iniciar esta *probe* sem parâmetros de entrada, com a indicação do *endpoint* do `TMA_Monitor` errado e, por fim, com o *endpoint* do `TMA_Monitor` correto.

Tabela 5.2: Casos de teste para o método *validate_schema*

Caso de Teste	Input	Outcome Esperado
1	Sinal de interrupção do funcionamento da aplicação	Log da mensagem "Error Code -1"seguida da descrição da exceção
2	HTTP Request com a mensagem { "probeId": 10, "resourceId": 11, "messageId": 1000, "sentTime": 1524780010, "data": [{"type": "measurement", "descriptionId": 2019, "observations": [{ "time": 1524782445,"value": -1}] }] }	Mensagem enviada para o Apache Kafka
3	HTTP Request com a mensagem {"resourceId": 11, "messageId": 1000, "sentTime": 1524780010, "data": [{"type": "measurement", "descriptionId": 2019, "observations": [{ "time": 1524782445,"value": -1}] }] }	Retorno do número de erros e respetiva descrição
4	HTTP Request com uma mensagem "hello"	Exceção na validação da mensagem

Tabela 5.3: Casos de teste para o método *main* da *probe_python_demo*

Caso de Teste	Input	Outcome Esperado
1	Iniciar aplicação sem parâmetros de entrada	Impressão da exceção
2	Endpoint do Monitor errado	Impressão da exceção
3	Endpoint do Monitor certo	Envio de mensagens para o Monitor

A *probe probe_k8s_docker* tem dois métodos que foram alvo de teste. O primeiro, que é o menos complexo, é o método *get_stats* que tem uma complexidade de 2 unidades, pelo que foi necessário gerar dois casos de teste cobrir todo o código deste método. OS casos de teste gerados está presentes na Tabela 5.4.

Tabela 5.4: Casos de teste para o método *get_stats* da *probe_k8s_docker*

Caso de Teste	Input	Outcome Esperado
1	Nome de um *container* que está executar	Receção de mensagens no Monitor
2	Interrupção da execução do *container* com a aplicação a executar	Saída do método

O método *format* da *probe probe_k8s_docker* tem de complexidade 2, pelo que foi necessário gerar 2 casos de teste para cobrir 100% do seu código. Os casos de teste são ilustrados na Tabela 5.5

Para o método *main* da *probe probe_k8s_network* foram gerados 4 casos de teste que estão apresentados na Tabela 5.6. Este método tem mais uma unidade de complexidade em relação aos restantos métodos *main* das restantes *probes*, depende de uma aplicação externa que faz com que o número de complexidade suba em uma unidade em relação

Tabela 5.5: Casos de teste para o método *format* da *probe_k8s_docker*

Caso de Teste	Input	Outcome Esperado
1	Introdução do nome de um *container* que esteja em execução	retorno de um JSON com os valores de todas as métricas recolhidas
2	Interrupção da execução do *container* durante a execução desta aplicação	O JSON retornado está vazio

aos restantes métodos das outras *probes*. Essa aplicação chama-se *Prometheus* que foi abordada em capítulos anteriores deste relatório.

Tendo em conta esta dependência, os *inputs* dos casos de teste que foram gerados para este método dependem diretamente da execução ou não do *Prometheus*, da existência ou não de passagem de argumentos pelo terminal quando se inicia a aplicação e, se forem passados esses argumentos, se estes estão corretos ou não.

Tabela 5.6: Casos de teste para o método *main* da *probe_k8s_network*

Caso de Teste	Input	Outcome Esperado
1	Iniciar aplicação sem parâmetros de entrada	Impressão da exceção gerada
2	Endpoint do Monitor errado	Impressão da exceção gerada
3	Prometheus não está a ser executado	Impressão da exceção gerada
4	Endpoint do Monitor correr e Prometheus a executar	Envio de mensagens para o Monitor

A *probe probe_cs_demo*, tal como para a *probe probe_python_demo*, tem de complexidade 3 unidades, o que faz com que tenham sido gerados 3 casos de teste para testar todos os caminhos independentes que existem neste método. Os casos de teste gerados são apresentados na Tabela 5.7.

Posto isto, e tal como os casos de teste apresentados na Tabela 5.3, os casos de teste usados para testar este método prendem-se com a inicialização desta *probe* sem parâmetros de entrada, com a indicação do *endpoint* do `TMA_Monitor` errado e, por fim, com o *endpoint* do `TMA_Monitor` correto.

Tabela 5.7: Casos de teste para o método *main* da *probe_cs_demo*

Caso de Teste	Input	Outcome Esperado
1	Iniciar aplicação sem parâmetros de entrada	Impressão da exceção gerada
2	Endpoint do Monitor errado	Impressão da exceção gerada
3	Endpoint do Monitor corretamente inserido	Envio de mensagens para o Monitor

Passando para os módulos desenvolvidos para o *Apache Flume* e começando pelo módulo desenvolvido para o modo normal da plataforma, este módulo é relativamente simples, tendo de complexidade 4 unidades, pelo que foram necessários gerar 4 casos de teste para cobrir 100% do seu código. Os casos de teste gerados para este método estão apresentados na Tabela 5.8.

Os casos de teste gerados são relativamente simples, pois este módulo está a consumir eventos vindos do tópico *topic-monitor* do *Apache Kafka*, daí que um dos *inputs* dos casos de teste seja a não receção desses mesmos eventos por parte deste módulo.

Os casos de teste nº 2 e 3 foram gerados tendo em conta os ciclos que percorrem o *array* data e o *array observations*. Se no evento recebido não existir nenhum dos *arrays* enumerados anteriormente, o módulo apenas retorna o conteúdo da mensagem recebida.

Por outro lado, o último caso de teste prevê que se a mensagem chegar no formato correto, o módulo irá funcionar normalmente e os dados serão guardados na base de dados.

Tabela 5.8: Casos de teste para o método *parse_events* do modo normal da plataforma

Caso de Teste	Input	Outcome Esperado
1	Não existência de eventos	Retorno da lista de eventos vazia
2	Mensagem enviada pelo Monitor com o array de data vazio	Retorno da mensagem com o array de data vazio
3	Mensagem enviada pelo Monitor com o array de observations vazio	Retorno da mensagem com o array de observations vazio
4	Mensagem enviada pelo Monitor de acordo com o formato correto	Retorno da lista de eventos

O outro módulo desenvolvido para o *Apache Flume* para possibilitar a implementação da plataforma no modo teste também foi alvo de testes. O código deste módulo tem exatamente a mesma estrutura do módulo desenvolvido para modo normal, daí ter exatemente a mesma complexidade, 4 unidades, e, por consequência o mesmo número de casos de teste.

Como o código em si deste método é muito semelhante ao código do módulo para o modo normal, o casos de teste gerados são iguais, por isso, os *inputs* dos casos de testes são, a não receção desses mesmos eventos por parte deste módulo, a não existência dos *arrays* data e *observations* e, por fim, a receção da mensagem no formato correto.

Tabela 5.9: Casos de teste para o método *parse_events* do modo de teste da plataforma

Caso de Teste	Input	Outcome Esperado
1	Não existência de eventos	Retorno da lista de eventos vazia
2	Mensagem enviada pelo Monitor com o array de data vazio	Retorno da mensagem com o array de data vazio
3	Mensagem enviada pelo Monitor com o array de observations vazio	Retorno da mensagem com o array de observations vazio
4	Mensagem enviada pelo Monitor de acordo com o formato correto	Retorno da lista de eventos

5.2.4 Execução dos testes e análise dos resultados

Todos os casos de teste apresentados e descritos na subsecção anterior foram injetados nos métodos para os quais foram gerados. Para injetar os casos de teste presentes na Tabela 5.1 e na Tabela 5.2 foi usada a ferramenta *curl* para gerar e injetar os HTTP *requests* com o respetivo comando HTTP no *endpoint* do `TMA_Monitor` [13].

Para injetar os *inputs* da Tabela 5.3 apenas basta iniciar a *probe probe_ python_ demo* com os *inputs* dos casos de teste presentes nesta tabela.

Os casos de teste da Tabela 5.4 exigem se inicialize um *container Docker* atribuindo-lhe um nome. Depois executou-se os casos de teste conforme os *inputs* presentes nesta tabela.

À semelhança da execução dos casos de teste para o método *get_ stats* da *probe probe_ k8s_ docker*, os casos de teste do método *format*, ilustrados pela Tabela 5.5 da mesma *probe* necessitam do mesmo requisito. A diferença encontra-se no facto de que para cada caso de teste ter sido necessário iniciar um determinado *container* de acordo com o *input* desse mesmo caso de teste.

De modo a injetar os casos de teste da Tabela 5.6 na *probe probe_ k8s_ network*, foi necessário lançar dois *pods* do *Kubernetes*. Num dos *pods* está a executar o *container* que é o alvo da monitorização e no outro *pod* está a ser executado o *Prometheus*.

Com estes dois pré-requisitos cumpridos, iniciou-se a *probe probe_ k8s_ network* e injetou-se os *inputs* presentes na tabela.

Em relação aos casos de teste da Tabela 5.7, estes foram executados quando foi feita a inicialização da *probe probe_ cs_ demo* conforme os *inputs* de cada caso de teste desta tabela.

Os últimos dois métodos que foram alvos de testes foram as módulos desenvolvidos para o *Apache Flume* e como ambos têm o código com a mesma estrutura, os casos de teste gerados foram os mesmos e, por isso, os testes também foram executados de forma idêntica.

Sendo assim, para executar os casos de teste presentes na Tabela 5.8 foi necessário iniciar o `TMA_Monitor`, o *Apache Kafka* e o *Apache Flume* em diferentes *pods* num *cluster Kubernetes*. No *script* de inicialização do *Apache Flume* escolheu-se a opção 1, ou seja, a inicialização em modo normal. Posto isto, foi apenas necessário enviar o HTTP *request* de acordo com os *inputs* dos casos de teste para o *endpoint* do `TMA_Monitor`.

Por fim, para executar os casos de teste apresentados na Tabela 5.9 para o módulo do *Apache Flume* para o modo de teste da plataforma foi necessário também a inicialização do `TMA_Monitor`, o *Apache Kafka* e o *Apache Flume* em *pods* num *cluster Kubernetes*. A diferença está na inicialização do *Apache Flume*, que, para este teste, foi necessário iniciá-lo no modo de teste.

Posto isto, foi apenas necessário enviar os HTTP *requests* para o *endpoint* do `TMA_Monitor` de acordo com os *inputs* dos casos de teste.

Como todos os métodos testados têm uma complexidade, relativamente, pequena, a execução dos testes foi rápida e os resultados desses testes foram os esperados. Sendo assim, todos os casos de teste apresentados na subsecção anterior foram executados e o *outcome* dos mais variados métodos correspondeu ao *outcome* esperado, por isso, consegui-se concluir que os testes não detetaram nenhum *bug* e que o código foi 100% coberto segundo o critério de cobertura de caminhos.

5.3 Validação de requisitos não funcionais

Nesta secção vai ser descrito todo o processo de valiação de requisitos não funcionais como o desempenho, a escalabilidade e a elasticidade da plataforma de monitorização. Em primeiro lugar será descrito o ambiente de testes incluindo as especificações da máquina usada para gerar a carga e também das máquinas que constituem o *cluster* Kubernetes. Em seguida é explicado todo o processo de execução de cada tipo de teste e, por fim, são apresentados os resultados e respetiva análise.

Todos os testes de validação acima descritos têm como principal objetivo testar o funcionamento do `TMA_Monitor`, pois é o que fornece o único *endpoint* de entrada de dados de toda a plataforma, por isso, pretende-se que seja capaz de processar volumes de dados relativamente grandes sem que toda a plataforma sofra perturbações.

5.3.1 Ambiente Experimental

A validação dos três requisitos não funcionais foram validados usando o mesmo ambiente de testes. Este ambiente é consituído por uma máquina cujas especificações são as seguintes:

- **Processador**: AMD Ryzen 7 1700 Eight-Core Processor x 16;
- **Memória**: 32GB DDR4 RAM;
- **Disco**: Samsung SSD 850 EVO 500GB (EMT02B6Q).

Dentro desta máquina foram virtualizadas recorrendo ao programa Virtual Machine Manager todas as quatro máquinas que foram necessárias para hospedar a plataforma, onde se incluem uma máquina *Master* do *cluster*, dois *Workers* e uma outra máquina onde está a executar o *Ceph*.

Para validar quer o desempenho, quer a escalabilidade, que a elasticidade foi criada uma aplicação em Java que tem como função enviar HTTP *requests* para o *endpoint* do `TMA_Monitor`. Esta aplicação foi desenvolvida de forma a que seja possível personalizar os tipos de dados que constituem a mensagem a enviar (eventos ou medidas ou ambos), o tamanho dos mesmos e a velocidade a que são enviados para o `TMA_Monitor`.

As máquinas virtuais onde está hospedado o *Master* do *cluster* e o *Ceph* têm as seguintes especificações: 4096 MB de memória Random Access Memory (RAM), 4 CPUs e 40GB de espaço em disco. Por outro lado, os dois *workers* têm as seguintes especificações: memória RAM: 8192MB; 4 *cores* CPU e 40GB de espaço em disco.

5.3.2 Metodologia Experimental

O desempenho e a escalabilidade da plataforma foram validados recorrendo a dez perfis de testes onde se foram variando variáveis como a diversidade e número de dados da mensagem a enviar, assim como a frequência com que estes são enviados para o *endpoint* do `TMA_Monitor`.

Os dez perfis de testes estão representados na Tabela 5.10.

Os dez perfis de teste foram gerados combinando vários valores de parâmetros como a diversidade e o tamanho dos dados e a frequência com que estes são enviados para o

Tabela 5.10: Perfis dos testes utilizados na validação de desempenho e escalabilidade da plataforma.

	Diversidade			Tamanho				Frequência	
	Tipo 1	Tipo 2	Tipo 3	Pequeno	Médio	Grande	Muito Grande	Baixa	Alta
#	eventos	medidas	ambos	1 Obs	500 Obs	5000 Obs	50000 Obs	500 rps	5000 rps
1	X			X				X	
2		X		X					X
3		X			X				X
4			X		X				X
5	X					X		X	
6		X				X			X
7			X			X			X
8	X						X	X	
9		X					X		X
10			X				X		X

TMA_Monitor. Dentro da diversidade dos dados, as observações podem ser constituídas por eventos, medidas ou ambas, pois são os únicos tipos de dados que o TMA_Monitor é capaz de validar.

A variação do tamanho das mensagens também foi considera nesta validação, por isso foram considerados quatro tamanhos diferentes que foram uma mensagem pequena com apenas com uma observação, uma mensagem maior com 500 observações e depois, dois tamanhos que já são considerados grandes que são 5000 e 50000 observações. Estes dois últimos tamanhos foram incluídos nesta validação com o objetivo de verificar o limite do tamanho de mensagens que o TMA_Monitor.

Em relação ao tamanho da mensagem, os valores utilizados foram de 1, 500, 5000 e 50000 observações. Tratando-se de um teste de desempenho torna-se importante a existência de pacotes de tamanho que vão desde o mais pequeno possível até um tamanho exageradamente grande, pois só assim é que se pode tirar conclusões concretas acerca do tamanho máximo de uma mensagem que o TMA_Monitor consegue processar.

O último parâmetro considerado a frequência de envio de mensagens onde é definido quantidade de *requests* que são enviados para o *endpoint* do TMA_Monitor. Para esta parâmetro foram usados valores de 500 *requests* por segundo, pois é a frequência de *requests* que se espera que o TMA_Monitor seja capaz de processar e, depois, também foi usada uma frequência mais elevada, de 5000 *requests* por segundo que tem como objetivo tentar concluir a frequência máxima de *requests* por segundo que o TMA_Monitor consegue lidar.

Antes da execução de cada perfil de teste, todos os componentes da plataforma foram implementados no *cluster* Kubernetes, em que foi garantido que os tópicos do *Apache Kafka* se encontram vazios, toda a informação da base de dados foi apagada e, posteriormente, foram inseridos exatamente os mesmos dados de exemplo nas tabelas para evitar que houvesse erros devido à existência de chaves estrangeiras entre as tabelas populadas e para garantir que a base de dados é iniciada sempre no mesmo estado.

Todos os perfis de teste foram executados com a duração de dez minutos. No fim da execução de cada teste, foram recolhidos os valores das métricas de tempo de resposta e de *throughput*, assim como o respetivo desvio padrão.

O desempenho foi validado recorrendo à execução de todos estes perfis de teste usando a aplicação de geração de carga descrita anteriormente com apenas uma réplica do TMA_Monitor. Enquanto que para validar a escalabilidade, o foram repetidos exatamente os mesmos testes, mas desta vez com duas réplicas do TMA_Monitor inicializadas no *cluster* Kubernetes.

Por sua vez, para validar a elasticidade do `TMA_Monitor` foram gerados 8 perfis de teste onde só é alterada a frequência de envio dos HTTP *requests*. Com esta validação pretendeu-se demonstrar a capacidade de auto-escalonamento do `TMA_Monitor` face a diferentes volumes de carga.

Todo o processo de auto-escalonamento do `TMA_Monitor` foi configurado com base no recurso Horizontal Pod Autoscaler (HPA), onde se define o *deployment* ao qual pertence o *pod* a escalar, o número mínimo e máximo de réplicas que esse *pod* pode ter e, por fim, a percentagem de CPU limite que, quando atingido, são criadas mais réplicas desse mesmo *pod*.

Neste caso, o HPA utilizado foi configurado para escalar o *pod* do `TMA_Monitor` com um número mínimo de uma réplica e um máximo de 5 réplicas. O limite deCPU foi configurado para 80%.

A validação da elasticidade foi feita recorrendo duas baterias de testes. Em cada uma dessas baterias de testes apenas foi alterado duração de cada perfil de teste a frequência de envio dos dados.

A Tabela 5.11 e a Tabela 5.12 ilustram, respetivamente, as configurações dos perfis de teste das primeira e segunda baterias de testes.

Tabela 5.11: Perfis dos testes da primeira bateria de testes utilizados na validação da elasticidade da plataforma.

#	Duração (min)	nº de obs	Diversidade de dados	Frequência (req/seg.)
1	2	500	ambos	500
2	2	500	ambos	1000
3	2	500	ambos	1500
4	2	500	ambos	500
5	2	500	ambos	2000
6	2	500	ambos	1500
7	2	500	ambos	100
8	2	500	ambos	500

Tabela 5.12: Perfis dos testes da segunda bateria de testes utilizados na validação da elasticidade da plataforma.

#	Duração (min)	nº de obs	Diversidade de dados	Frequência (req/seg.)
1	1	500	ambos	500
2	1	500	ambos	1500
3	1	500	ambos	3000
4	1	500	ambos	1000
5	1	500	ambos	5000
6	1	500	ambos	1000
7	1	500	ambos	5000
8	1	500	ambos	1000

A execução destes teste levam menos tempo do que os anteriores e a variação da frequência de envio de dados é grande, pois o objetivo desta validação foi verificar a rápida adaptação do `TMA_Monitor` face a mudanças abruptas de carga.

O `TMA_Monitor` foi sujeito a estas baterias de testes usando a mesma aplicação usada nos testes de desempenho e de escalabilidade. Para executar estas baterias foi usado um *script bash* em que essa aplicação é executada oito vezes recebendo por argumento a frequência de envio de dados e o tempo de duração de cada perfil de teste.

5.3.3 Análise de Resultados

Nesta subseção vão ser apresentados os resultados e respetiva análise da validação do desempenho, escalabilidade e elasticidade da plataforma desenvolvida neste estágio.

Os resultados dos testes de validação de desempenho são apresentados na Tabela 5.13. Nestes testes apenas foi usada uma réplica do `TMA_Monitor`.

Tabela 5.13: Resultados da validação de desempenho da plataforma

	Tempo de resposta (ms)		Throughput (req/seg.)
Teste	Média	Desvio Padrão	Valor
1	7.567	54.977	499.983
2	29.83	101.623	1552.078
3	193.787	208.972	253.954
4	206.767	248.528	237.77
5	1643.977	1614.874	29.174
6	1631.489	1210.343	29.569
7	1634.992	1556.437	29.269
8	15917.105	8264.458	2.576
9	15971.455	8811.39	2.568
10	16432.478	11206.677	2.175

O número dos testes presentes na Tabela 5.13 correspondem aos que foram apresentados na Tabela 5.10. Em relação ao primeiro perfil de teste, a frequência definida nesse teste em específico é de 500 *requests* por segundo e o `TMA_Monitor` conseguiu um *throughput* de praticamente esse valor, o que indica que a plataforma conseguiu suportar com a carga produzida por este perfil de teste. Neste teste, as mensagens apenas possuem uma observação, o que torna simples o processamento. O tempo de resposta foi de 7,567 milisegundos, o que é um valor bastante aceitável.

Aumentando o tipo de dados e a frequência de 500 *requests* por segundo para 5000 *requests* por segundo, o *TMA_Monitor* consegue processar cerca de 1550 *requests* por segundo com um tempo de resposta de 29 milisegundos aproximadamente. Neste caso, o `TMA_Monitor` já não é capaz de acompanhar a frequência de envio por parte do cliente, o que permite concluir que para uma réplica do `TMA_Monitor`, para mensagens com apenas uma observação, a capacidade de processamento deste componente é cerca de 1550 *requests* por segundo.

Passando para cargas mais exigentes, nomeadamente para mensagens 500 obsevações, o que configura um aumento significativo da quantidade de dados a validar, conclui-se que o *throughput* do *TMA_Monitor* baixa significativamente acompanhando o aumento do número de observações. Neste caso, o tempo de resposta também aumenta significativamente

devido ao facto de aumentar a quantidade de dados que são necessários validar.

Neste perfil de teste em específico, a frequência de envio também foi de 5000 *requests* por segundo, o que contribui também bastante para a degradação da taxa de processamento do `TMA_Monitor`.

Os resultados da execução da configuração do Teste 4 são semelhantes aos obtidos pela configuração do Teste 3. A diferença entre a configuração do Teste 3 e a do Teste 4 encontra-se na diversidade dos dados, já que no Teste 3 apenas foram enviadas medições, enquanto que no Teste 4 foram enviados tanto eventos como medições. Através dos resultados podemos concluir que esse facto tem um ligeiro impacto no desempenho do `TMA_Monitor`.

Os resultados obtidos executando as configurações dos testes 5, 6 e 7 são muito semelhantes. Nas configurações destes três perfis de testes cada mensagem tem 5000 observações, dez vezes mais do que as configurações dos últimos testes. De notar que uma mensagem com este tamanho num cenário de produção já se pode considerar uma mensagem de grande dimensão.

Esse aumento substâncial, cerca de dez vezes, no tamanho da mensagem e, por sua vez, no número de dados que é necessário processar fez com que o *throughput* do `TMA_Monitor` baixasse, sensivelmente na mesma ordem de grandeza. Apesar dos resultados serem sensilvente os mesmos entre estas três configurações de testes, existem algumas diferenças entre eles.

No Teste 5, a frequência de envio das mensagens assume o valor de 500 *requests* por segundo, enquanto que nas restantes duas configurações, a frequência situa-se nos 5000 *requests* por segundo. Tendo em conta a diferença mínima de resultados, quer a nível de tempo de resposta, quer a nível de *throughput*, pode-se concluir que o `TMA_Monitor`, para este tamanho específco de mensagem não consegue processar mais de cerca de 30 *requests* por segundo com apenas uma réplica do seu serviço.

Nestes três testes ainda se variou a diversidade dos dados com o objetivo de confirmar se esta variável tinha algum impacto no desempenho do `TMA_Monitor`. Mais especificamente no Teste 6 e 7, todas as outras variáveis têm o mesmo valor e apenas é variada a diversidade das mensagens em que no Teste 6 só são enviadas medidas e no Teste 7 são enviados eventos e medidas.

Como o valor do tempo de resposta e de *throughput* obitdos executando estas duas configurações são praticamente os mesmos, existindo diferenças residuais, pode-se concluir que a diversidade das mensagens não tem impacto no desempenho do `TMA_Monitor` para tamanhos de mensagens na ordem das 5000 observações.

Por fim, o mesmo raciocínio também se pode aplicar aos resultados obtidos pelas configurações dos Testes 8, 9 e 10. Nestas três últimas configurações, os resultados também são bastante idênticos entre si. No caso destas três configurações em específico, o tamanho da mensagem foi aumentado também em cerca de dez vezes, em relação aos três testes anteriores. O tamanho da mensagem nestas três últimas configurações é de 50000 observações.

Uma mensagem com 50000 observações não é praticável. Este tipo de mensagens só é passível de serem processadas quando se trata de migração de dados e, quando assim, é as mensagens são enviadas a velocidades bastantes menores do que as que são consideradas nesta bateria de testes e com a existência de várias réplicas do mesmo serviço.

Nestas três últimas configurações vemos que o valor do *throughput* também sofreu uma diminuição para dez vezes menos, no entanto o tempo de resposta manteve-se, o que leva a concluir que que 16 segundos é o tempo de resposta mínimo garantido pelo `TMA_Monitor`

para esta ordem de grandeza de mensagens.

Nestes três últimos testes, também se variam outras variáveis como a frequência de envio das mensages e a diversidade das mesmas, tal como nas configurações anteriores, no entanto não alteraram significativamente os resultados. O que leva a concluir que para um tamanho de 50000 observações, o desempenho do `TMA_Monitor` não fugirá muito destes valores de tempo de resposta e de *throughput*.

A análise do desempenho do `TMA_Monitor` permitiu alcançar os limites de desempenho deste componente e da plataforma de monitorização em geral face às diversas configurações de perfis de testes apresentadas anteriormente.

Analisados os resultados referentes ao desempenho do `TMA_Monitor`, a próxima validação diz respeito à escalabilidade deste componente.

A Tabela 5.14 representa os resultados desta validação. Os resultados desta tabela foram recolhidos com duas réplicas do `TMA_Monitor` a executar no *cluster* Kubernetes.

Tabela 5.14: Resultados da validação de escalabilidade da plataforma

	Tempo de resposta (ms)		Throughput (req/seg)
Teste	Média	Desvio Padrão	Valor
1	5.179	49.756	497.651
2	26.02	98.474	1627.981
3	137.111	134.906	358.549
4	138.39	92.743	354.483
5	1149.66	608.848	41.568
6	1140.972	703.903	41.925
7	1137.579	461.835	42.368
8	11830.432	4776.223	3.512
9	10731.468	6339.000	3.942
10	12256.007	6744.051	3.058

Este conjunto de testes foram executados com o objetivo de verficar se duas réplicas do `TMA_Monitor` conseguem obter melhores resultados de tempo de resposta e de *throughput* do que apenas só uma réplica. Olhando para a Tabela 5.14 percebe-se que os valores, para as mesmas configurações de testes, subiram de uma forma significativa na maioria dos casos. Analisando o Teste 1, vemos que o resultado do *throughput* diminui ligeiramente em relação ao mesmo valor na Tabela 5.13. No entanto, o tempo de resposta diminui cerca de 2 milisegundos. Este teste é configurado para mandar apenas uma observação por mensagem, o faz com que as duas réplicas do `TMA_Monitor` consigam processar quase na totalidade todas as mensagens que lhe chegam com estas características.

Mantendo o tamanho da mensagem e aumentando a frequência de envio, uma maior percentagem de mensagens são processados pelo *TMA_ Monitor* com mais um réplica. A diferença é de cerca de 5%.

Passando para tamanhos de mensagens maiores com 500 observações, o *throughput* diminui e o tempo de resposta aumenta, não variando muito entre o Teste 3 e o Teste 4, pois no

caso destas duas configurações, apenas é mudado o conteúdo da mensagem, o que faz com que se possa concluir que o conteúdo da mensagem não influencia significativamente o desempenho do `TMA_Monitor` com duas réplicas. No entanto houve uma melhoria de cerca de 45% dos valores em relação aos obitos com apenas uma métrica deste componente a ser executada.

Tal como nos testes de desempenho, os valores obtidos pelos testes 5, 6 e 7 são bastante semelhantes. Nestes testes foi aumentado o tamanho da mensagem para 5000 observações, o que fez, como era esperado, que o *throughput* das duas réplicas diminuisse e o tempo de resposta aumentasse significativamente. No entanto, o *throughput* aumentou cerca de 43% em relação aos valores com apenas uma réplica deste serviço.

Os testes 8, 9 e 10 são os testes cujos resultados obtidos foram piores, pois trata-se de mensagens com 50000 observações. Este tamanho é propositadamente grande de forma a verficar se mesmo com duas réplicas, o `TMA_Monitor` conseguia processar alguns dados. De acordo com os dados ilustrados na Tabela 5.14, duas réplicas deste componente conseguem processar três *requests* por segundo com mensagens deste tamanho.

Esta resultado parece ser razoável, pois mensagens com este tamanho apenas serão geradas e processadas quando existir uma migração de uma quantidade elevada de dados e, nesse caso, os dados não serão enviados a 500 *requests* por segundo nem tão pouco a uma frequência de 5000 *requests*. No entanto, com duas réplicas houve uma melhoria de cerca de 43% face ao desempenho obtido com uma apenas uma réplica do `TMA_Monitor`.

Resumindo, os valores para o tempo de resposta e *throughput* com duas réplicas a executar do `TMA_Monitor` aumentaram significativamente em relação aos valores obtidos utilizando apenas uma réplica deste componente. As melhorias rondaram os 28% no caso do tempo de resposta e 35% no caso do *throughput*. Face a estes aumentos, consegue-se concluir que o componente `TMA_Monitor` consegue escalar eficazmente de modo a aumentar a capacidade de processamento de toda a plataforma.

Por fim, para testar a elasticidade do `TMA_Monitor` utilizou-se o mecanismo de auto-escalonamento automático do Kubernetes para verificar a adaptação deste componente face a diferentes volumes de dados que está sujeito. O auto-escalonador do Kubernetes, foi configurado para replicar o *pod* do `TMA_Monitor` quando este chegar aos 80% de utilização de CPU até um máximo de 5 réplicas.

Para testar este requisito foram geradas duas baterias de testes que estão representadas pela Tabela 5.11 e pela Tabela 5.12. Em ambas as baterias de testes a única variável que se altera de teste para teste é a frequência de envio de mensagens.

Os resultados da primeira bateria de teste são ilustrados pela Figura 5.10

Os resultados ilustrados pela Figura 5.10 mostram que no início ou seja, quando a frequência não é muito elevada, o número de réplicas mantém-se. Aos 6 minutos é criada um nova réplica, referente ao aumento do *throughput* do teste 4.

No entanto, nos próximos testes, as frequências já são mais elevadas, o que faz com que novas réplicas sejam criadas para manter ou aumentar o *throughput*. No teste 8, como a descida da frequência no envio dos dados é muito abrupta, o Kubernetes não diminui o número de réplicas do `TMA_Monitor`. No minuto 10, dá-se início a configuração com mais frequência, com cerca 2000 *requests* por segundo, sendo que o número de réplicas do `TMA_Monitor` chega ao máximo, a 5 réplicas ao início do minuto 12.

Depois deste pico, nos próximos três testes, a frequência de envio de mensagens diminui lentamente e o número de réplicas do `TMA_Monitor` vai-se adaptando para que haja o

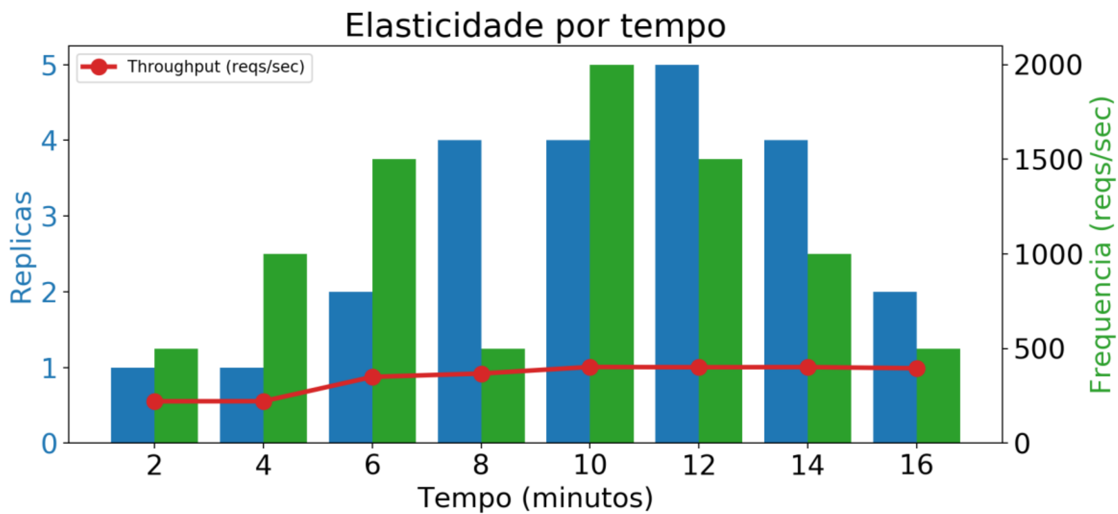

Figura 5.10: Gráfico com *throughput* e número de réplicas para testes com dois minutos

menor número de réplicas possível. Se a experiência continuasse, exatamente com a mesma frequência com que terminou, era de espera que o *throughput* descesse para valores próximos do minuto 2 ou do minuto 4.

A média do *throughput* sofre um variação aos seis minutos, pois a frequência no envio dos dados também aumenta. A partir desse momento, este valor mantém-se até ao fim da execução da bateria de testes, apesar de todas as adaptações feitas pelo Kubernetes.

Os resultados da segunda bateria de testes é ilustrado pelo gráfico da Figura 5.11.

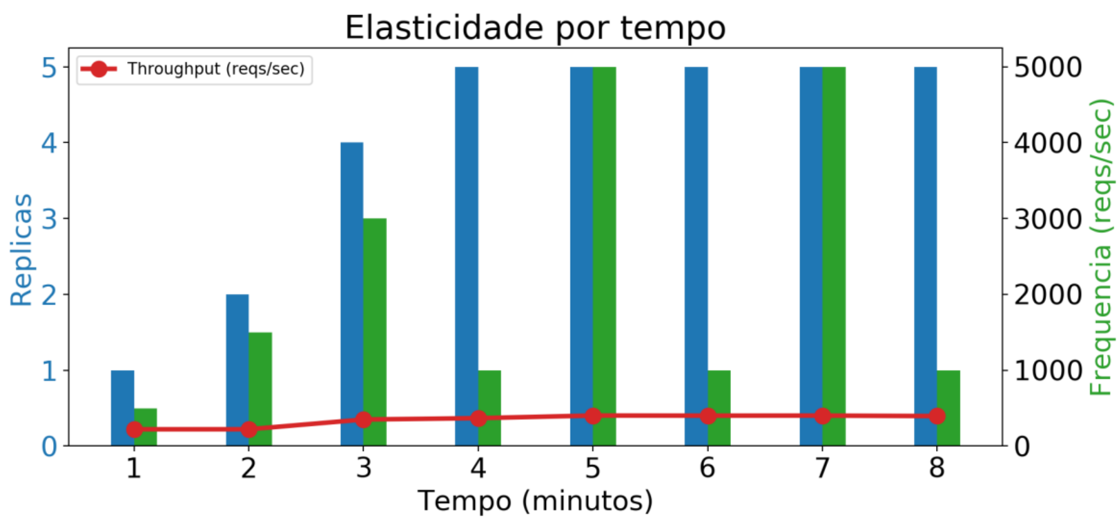

Figura 5.11: Gráfico com *throughput* e número de réplicas para testes com um minuto

A frequência dos perfis dos primeros 3 testes vai aumentando na ordem dos 1000 *requests* por segundo, o que faz com o núemro de réplicas também vá aumentando para se ajustar à frequência desses mesmos perfis. Ao minuto 4, o número de réplicas atinje o seu máximo devido ao aumento da frequência dos primeiros três testes, como foi referido anteriormente.

No teste 4, existe uma descida abrupta de *throughput*, no entanto, mais uma vez o Kubernetes mantém as 5 réplicas por se tratar de uma descida brusca na frequência, cerca de 5 vezes menos, o que faz com que o Kubernetes opte por manter o número de réplicas. A

partir do minuto 5, a frequência de envio de mensagens varia bruscamente entre os 5000 e o 1000 *requests* por segundo, pelo que é mantido o número máximo de réplicas.

A média do *throughput* sofre poucas alterações, o que permite concluir que apesar do aumento e diminuição da frequência de envio de mensagens, o `TMA_Monitor` adapta-se com sucesso de modo a manter o seu desempenho.

Esta página foi propositadamente deixada em branco.

Capítulo 6

Planeamento

Neste capítulo, irão ser mostradas as tarefas que foram realizadas no primeiro e segundo semestre deste estágio assim como a metodologia de desenvolvimento a adotar no desenvolvimento do projeto.

6.1 Metodologia de Desenvolvimento

A metodologia que foi usada neste projeto foi o *Waterfall*. O *Waterfall* é uma metodologia tradicional cujo processo se baseia numa sequência de cinco fases essenciais à conceção de software: análise de requisitos, arquitetura, implementação, testes; e manutenção (ver Figura 6.1). É sempre assumido que a transição para a fase seguinte é feita apenas quando a atual se encontra finalizada e revista, daí o conceito de cascata.

O processo é muito rígido, o que obriga a um esforço bastante grande quando é feita uma alteração por ser necessário corrigir e rever todas as fases. No entanto, é bastante útil quando se pretende concluir totalmente uma fase antes de se avançar para as seguintes.

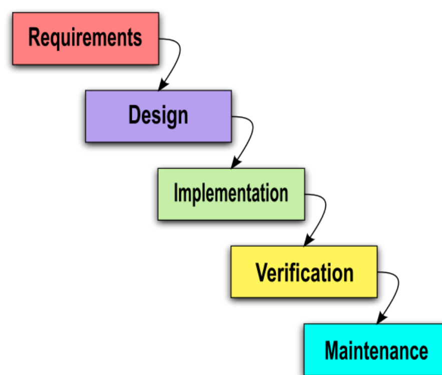

Figura 6.1: Fases da metodologia Waterfall (fonte [16]).

6.2 Esforço primeiro semestre

Este estágio foi planeado em duas partes em que cada uma delas corresponde a um semestre. A planificação irá ser ilustrada com o auxílio de diagramas de *Gantt*. Nesta secção, irão

ser mostrados dois diagramas de *Gantt* correspondendo, respetivamente, ao planeamento do primeiro semestre e ao planeamento previsto para o segundo semestre.

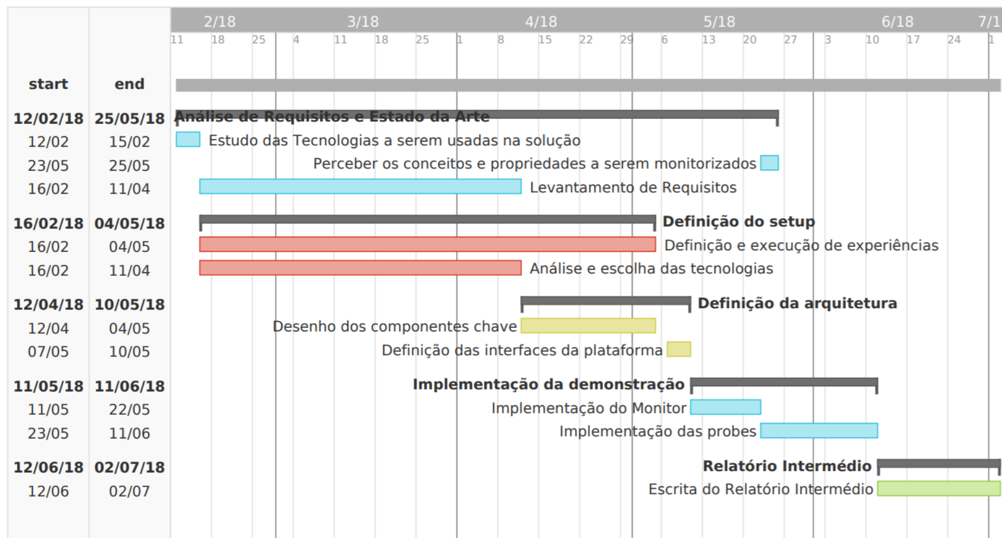

Figura 6.2: Diagrama de Gantt com as tarefas reais do primeiro semestre.

A Figura 6.2 ilustra o diagrama de *Gantt* com as tarefas, e respetiva duração, que efetivamente ocorreram na primeira parte deste estágio.

Através da observação do gráfico da Figura 6.2, vemos que a tarefa do estudo das ferramentas demorou pouco, pois já se tinha adquirido alguma experiência com a ferramenta essencial neste projeto, Docker, em outras unidades curriculares deste mestrado.

Sendo assim, investiu-se mais tempo nas tarefas de definição de *setup*, pois foi feita uma análise mais detalhada de todas as ferramentas, as quais foram quase todas experimentadas na prática, como, por exemplo, o Swarm e o Kubernetes, que são ferramentas equivalentes no que diz respeito à sua função, em que as experiências foram exatamente replicadas para os dois, o que acabou por consumir bastante tempo.

A definição da arquitetura decorreu como esperado. Duas tarefas que não estavam planeadas no início foram a implementação de uma demonstração do Monitor e de uma *probe*, acabando por demorar sensivelmente um mês a ser implementada.

Neste intervalo de um mês também se efetuou uma tarefa do primeiro conjunto de tarefas que foi a definição das métricas a monitorizar pela plataforma. Esta tarefa só foi feita nesta altura, pois foi necessária para o desenvolvimento das *probes*.

Estas duas tarefas adicionais também fizeram com que o tempo para a escrita do relatório passasse de um mês para sensivelmente três semanas.

6.3 Esforço do Segundo Semestre

Na Figura 6.3 são ilustradas as tarefas realizadas no segundo semestre.

As tarefas do segundo semestre consitiram no desenvolvimento da plataforma e respetiva validação, a escrita de um artigo científico e a escrita da tese final. A plataforma foi desenvolvida a partir do protótipo feito no primeiro semestre, onde foi continuada a imoplementação do `TMA_Monitor` e, a seguir, foram desenvolvidos os componentes `FaultTolerantQueue`

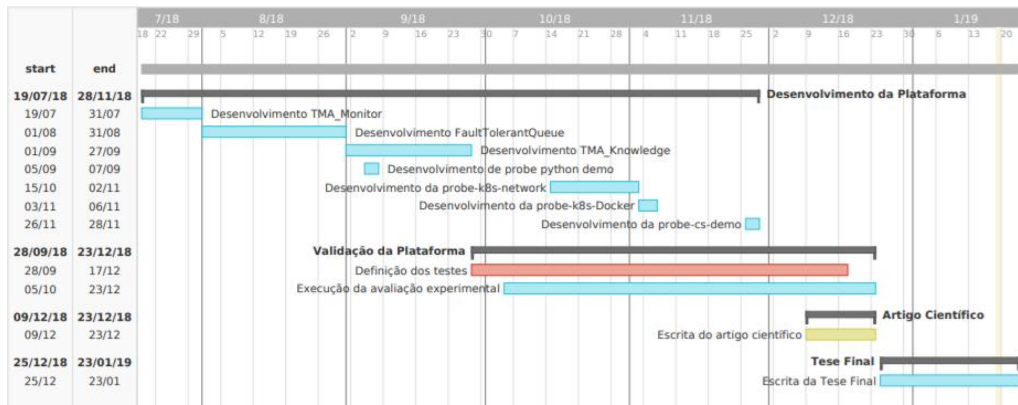

Figura 6.3: Diagrama de Gantt com as tarefas do segundo semestre

e o *TMA_ Knowledge*.

Ao longo do processo de desenvolvimento dos componentes principais foram desenvolvidas as quatro *probes* que eram necessárias para atestar o funcionamento da plataforma e também para *delivrables* do `ATMOSPHERE`.

Assim que acabou o desenvolvimento dos componentes, começaram a definição em pormenor dos testes de validação da plataforma. Esses testes incluíram testes estruturais e testes de desempenho, escalabilidade e de elasticidade.

Dado que houve uma *review* do projeto em Novembro, foi necessário demostrar o funcionamento do TMA. Dos resultados dessa demonstração, foi escrito um artigo científico que se encontra submetido e a agurdar a decisão por parte dos revisores. Para a escrita da tese final foi reservado cerca de um mês.

Esta página foi propositadamente deixada em branco.

Capítulo 7

Conclusões e Trabalho Futuro

Nos últimos anos, a utilização da *Cloud* aumentou significativamente, passando a ser adotada pela maior parte das grandes organizações para alojar os seus serviços ou informação que fazem parte do *core* de negócio. Tendo estes serviços ou informações uma importância crítica para as organizações, a monitorização torna-se um requisito crítico para perceber as causas das perturbações antes destas influenciarem a experiência do utilizador.

Neste estágio, foi desenvolvida uma plataforma escalável e flexível de monitorização para ambientes *Cloud*. A arquitetura desta plataforma foi desenhada para que os seus componentes sejam executados sob a forma de microserviços, por isso, são executados dentro de *containers* Docker geridos pelo Kubernetes.

Esta plataforma é constituída por três componentes principais: o `TMA_Monitor`, que recebe e valida os dados; `FaullTolerantQueue`, que é o componente principal da plataforma e é responsável por encaminhar toda a informação entre os componentes da plataforma; por fim, o `TMA_Knowledge`, que é o componente onde é armazenada toda a informação válida recolhida dos sistemas monitorizados. Juntamente com estes componentes, também foram desenvolvidas quatro *probes* que recolhem ou geram diferentes tipos de dados.

A validação do funcionamento da plataforma de monitorização foi feita através de testes de *white-box* de desempenho e de escalabilidade.

O trabalho futuro inclui o desenvolvimento da ferramenta de *design-time*, que fornece métricas relacionadas com o desenvolvimento dos componentes, a implementação da autenticação das *probes* no `TMA_Monitor` através de *tokens* e a implementação de mecanismos de segurança que garantam a confidencialidade e integridade das mensagens recebidas e encaminhadas pela ferramenta *Apache Kafka*, que pertence ao componente `FaultTolerantQueue`.

Este estagio foi também importante porque permitiu a participação em um projecto europeu onde tive a oportunidade de contribuir para o desenvolvimento de uma plataforma internacional e de interagir com parceiros da Europa e Brasil. Ainda tive a oportunidade de participar na reunião do *ATMOSPHERE* realizada em Universidade de Coimbra durante três dias, onde pude conhecer pessoalmente as pessoas que trabalham no projeto e de participar nas discussões técnicas.

Esta página foi propositadamente deixada em branco.

Referências

[1] Giuseppe Aceto, Alessio Botta, Walter de Donato, and Antonio Pescapè. Cloud monitoring: Definitions, issues and future directions. *2012 IEEE 1st International Conference on Cloud Networking (CLOUDNET)*, pages 63–67, 2012.

[2] Mehmet S. Aktas. Hybrid cloud computing monitoring software architecture. *Concurrency and Computation: Practice and Experience*, 30(21):e4694, 2018.

[3] Hirohisa Aman, Akiko Yamashita, Takashi Sasaki, and Minoru Kawahara. Multistage growth model for code change events in open source software development: An example using development of nagios. In *Proceedings - 40th Euromicro Conference Series on Software Engineering and Advanced Applications, SEAA 2014*, pages 207–212, 08 2014.

[4] Apache mesos - architecture. `http://mesos.apache.org/documentation/latest/architecture/`. Accessed: 2018-12-01.

[5] Docker architecture. `https://www.aquasec.com/wiki/display/containers/Docker+Architecture`. Accessed: 2018-12-01.

[6] Jiwon Bang, Siwoon Son, Hajin Kim, Yang-Sae Moon, and Mi-Jung Choi. Design and implementation of a load shedding engine for solving starvation problems in apache kafka. In *2018 IEEE/IFIP Network Operations and Management Symposium, NOMS 2018, Taipei, Taiwan, April 23-27, 2018*, pages 1–4, 2018.

[7] Marouane Birjali, Abderrahim Beni-Hssane, and Mohammed Erritali. Analyzing social media through big data using infosphere biginsights and apache flume. *Procedia Computer Science*, 113:280 – 285, 2017. The 8th International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN 2017) / The 7th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare (ICTH-2017) / Affiliated Workshops.

[8] Carl Boettiger. An introduction to docker for reproducible research. *SIGOPS Oper. Syst. Rev.*, 49(1):71–79, January 2015.

[9] Bottle: Python web framework — bottle 0.13-dev documentation. `https://bottlepy.org/docs/dev/`. Accessed: 2018-12-01.

[10] Morgan Brattstrom and Patricia Morreale. Scalable agentless cloud network monitoring. In *Cyber Security and Cloud Computing (CSCloud), 2017 IEEE 4th International Conference on*, pages 171–176. IEEE, 2017.

[11] Emiliano Casalicchio and Vanessa Perciballi. Measuring docker performance: What a mess!!! In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, pages 11–16. ACM, 2017.

[12] Architecture. https://coreos.com/rkt/docs/latest/devel/architecture.html. Accessed: 2018-12-01.

[13] curl. https://curl.haxx.se/. Accessed: 2019-01-04.

[14] Chris Davis. The architecture of open source applications: Graphite. https://www.aosabook.org/en/graphite.html. Accessed: 2019-01-07.

[15] Django: The web framework for perfectionists with deadlines. https://www.djangoproject.com/. Accessed: 2018-12-01.

[16] Comparing traditional systems analysis and design with agile methodologies. http://www.umsl.edu/~hugheyd/is6840/waterfall.html. Accessed: 2018-12-01.

[17] Michael Eder. Hypervisor-vs. container-based virtualization. *Future Internet (FI) and Innovative Internet Technologies and Mobile Communications (IITM)*, 1, 2016.

[18] Falcon - bare-metal web api framework for python. https://falconframework.org/. Accessed: 2018-12-01.

[19] Azadeh Farzan, Andreas Holzer, and Helmut Veith. Perspectives on white-box testing: Coverage, concurrency, and concolic execution. *2015 IEEE 8th International Conference on Software Testing, Verification and Validation, ICST 2015 - Proceedings*, 05 2015.

[20] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*, pages 171–172. IEEE, 2015.

[21] Welcome | flask (a python microframework). http://flask.pocoo.org//. Accessed: 2018-12-01.

[22] Wilhelm Hasselbring. Microservices for scalability: Keynote talk abstract. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, ICPE '16, pages 133–134, New York, NY, USA, 2016. ACM.

[23] Josune Hernantes, Gorka Gallardo, and Nicolas Serrano. It infrastructure-monitoring tools. *IEEE Software*, 32(4):88–93, 2015.

[24] Cheng-Hao Huang and Che-Rung Lee. Enhancing the availability of docker swarm using checkpoint-and-restore. In *Pervasive Systems, Algorithms and Networks & 2017 11th International Conference on Frontier of Computer Science and Technology & 2017 Third International Symposium of Creative Computing (ISPAN-FCST-ISCC), 2017 14th International Symposium on*, pages 357–362. IEEE, 2017.

[25] How does it work. http://www.informit.com/articles/article.aspx?p=2033339&seqNum=2. Accessed: 2019-01-07.

[26] Isam Mashhour Al Jawarneh, Paolo Bellavista, Luca Foschini, Giuseppe Martuscelli, Rebecca Montanari, Amedeo Palopoli, and Filippo Bosi. Qos and performance metrics for container-based virtualization in cloud environments. In *Proceedings of the 20th International Conference on Distributed Computing and Networking*, ICDCN '19, pages 178–182, New York, NY, USA, 2019. ACM.

[27] Luc Juggery. The tick stack as a docker application package. https://medium.com/lucjuggery/the-tick-stack-as-a-docker-application-package-1d0d6b869211. Accessed: 2019-01-07.

[28] Murat Karakus and Arjan Durresi. A survey: Control plane scalability issues and approaches in software-defined networking (sdn). *Computer Networks*, 112:279–293, 2017.

[29] Architecture. `http://www.kjkoster.org/zapcat/Architecture.html`. Accessed: 2019-01-07.

[30] Charalampos Gavriil Kominos. Performance analysis of different virtualization architectures using openstack, 2017.

[31] Andrew Leung, Andrew Spyker, and Tim Bozarth. Titus: Introducing containers to the netflix cloud. *Commun. ACM*, 61(2):38–45, January 2018.

[32] Keqin Li. Optimal power and performance management for heterogeneous and arbitrary cloud servers. *IEEE Access*, PP:1–1, 12 2018.

[33] L Magnoni. Modern messaging for distributed sytems. *Journal of Physics: Conference Series*, 608(1):012038, 2015.

[34] Michaël Marcozzi, Sébastien Bardin, Mickaël Delahaye, Nikolai Kosmatov, and Virgile Prevosto. Taming coverage criteria heterogeneity with ltest. *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 500–507, 2017.

[35] Leonardo Mariani, Cristina Monni, Mauro Pezzè, Oliviero Riganelli, and Rui Xin. Localizing faults in cloud systems. *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 262–273, 2018.

[36] A. Martin, S. Raponi, T. Combe, and R. Di Pietro. Docker ecosystem – vulnerability analysis. *Computer Communications*, 122:30 – 43, 2018.

[37] Víctor Medel, Rafael Tolosana-Calasanz, José Bañares, Unai Arronategui, and Omer Rana. Characterising resource management performance in kubernetes. *Computers & Electrical Engineering*, 68:286–297, 05 2018.

[38] Kubernetes vs docker swarm. `https://mindmajix.com/kubernetes-vs-docker-swarm`. Accessed: 2018-12-01.

[39] Roberto Morabito, Jimmy Kjällman, and Miika Komu. Hypervisors vs. lightweight virtualization: A performance comparison. *2015 IEEE International Conference on Cloud Engineering*, pages 386–393, 2015.

[40] Nagios plugins. `https://assets.nagios.com/downloads/nagioscore/docs/nagioscore/4/en/plugins.html`. Accessed: 2019-01-07.

[41] Introducing nats streaming. `https://nats.io/blog/introducing-nats-streaming/`. Accessed: 2018-12-01.

[42] 1.what is prometheus? `https://www.oreilly.com/library/view/prometheus-up/9781492034131/ch01.html`. Accessed: 2019-01-06.

[43] Claus Pahl and Pooyan Jamshidi. Microservices: A systematic mapping study. In *Proceedings of the 6th International Conference on Cloud Computing and Services Science - Volume 1 and 2*, CLOSER 2016, pages 137–146, Portugal, 2016. SCITEPRESS - Science and Technology Publications, Lda.

[44] Moo-Ryong Ra. Understanding the performance of ceph block storage for hyperconverged cloud with all flash storage. *CoRR*, abs/1802.08102, 2018.

[45] Muhammad Rathore. Kvm vs. lxc: Comparing performance and isolation of hardware-assisted virtual routers. *American Journal of Networks and Communications*, 2:88, 01 2013.

[46] Chapter 1. introduction to linux containers. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_atomic_host/7/html/overview_of_containers_in_red_hat_systems/introduction_to_linux_containers. Accessed: 2018-12-01.

[47] Gwen Shapira and Jeff Holoman. Flafka: Apache flume meets apache kafka for event processing. https://blog.cloudera.com/blog/2014/11/flafka-apache-flume-meets-apache-kafka-for-event-processing/. Accessed: 2019-01-17.

[48] Anas Shatnawi, Matteo Orrù, Marco Mobilio, Oliviero Riganelli, and Leonardo Mariani. Cloudhealth: A model-driven approach to watch the health of cloud services. *CoRR*, abs/1803.05233, 2018.

[49] Jacopo Soldani, Damian Andrew Tamburri, and Willem-Jan Van Den Heuvel. The pains and gains of microservices: A systematic grey literature review. *Journal of Systems and Software*, 146:215 – 232, 2018.

[50] Hassan Syed, Abdullah Gani, Fariza Nasaruddin, Anjum Naveed, Abdelmuttlib Ibrahim Abdalla Ahmed, and Khurram Khan. Cloudprocmon: A non-intrusive cloud monitoring framework. *IEEE Access*, PP:1–1, 08 2018.

[51] Kimitoshi Takahashi, Kento Aida, Tomoya Tanjo, and Jingtao Sun. A portable load balancer for kubernetes cluster. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, pages 222–231. ACM, 2018.

[52] Adriana Telesca, F Carena, W Carena, S Chapeland, V Chibante Barroso, F Costa, E Dénes, R Divià, U Fuchs, A Grigore, et al. System performance monitoring of the alice data acquisition system with zabbix. In *Journal of Physics: Conference Series*, volume 513, page 062046. IOP Publishing, 2014.

[53] Apache flume architecture. https://www.tutorialspoint.com/apache_flume/apache_flume_architecture.htm. Accessed: 2019-01-17.

[54] Apache kafka cluster architecture. https://www.tutorialspoint.com/apache_kafka/apache_kafka_cluster_architecture.htm. Accessed: 2018-12-01.

[55] Jun Yang, Wenjing Xiao, Jiang Chun, M. Shamim Hossain, Ghulam Muhammad, and Syed Amin. Ai powered green cloud and data center. *IEEE Access*, PP:1–1, 12 2018.

Anexos

Esta página foi propositadamente deixada em branco.

Anexo A

UNIVERSIDADE DE COIMBRA
FACULDADE CIÊNCIAS E TECNOLOGIAS
DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

MESTRADO EM ENGENHARIA INFORMÁTICA

FCTUC FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

Documento de Requisitos

Autor
Rui Silva

Tabela de Revisões dos Requisitos de Plataforma Escalável de Monitorização de *Cloud*

Número de Revisão	Data	Autores	Descrição
1	01/07/2018	Rui Silva	Elaboração

Índice

Índice 3
1.0 Informação Geral 4
- 1.1 Propósito 4
- 1.2 Âmbito 4
- 1.3 Acrónimos e abreviações 4

2.0 Atores 5
3.0 Requisitos funcionais 6
- 3.1 Integração de *probes* 8
- 3.2 Permissão de acesso aos dados recolhidos 9
- 3.3 Monitorização Contínua dos dados 10
- 3.4 Monitorização dos dados em várias tecnologias 10
- 3.5 Retenção e Recuperação de *Logs* 11
- 3.6 Autorização de recolha de dados 11
- 3.7 Inserção manual de eventos e medidas 12
- 3.8 Normalização das medidas e eventos recolhidos 13
- 3.9 Processo de Normalização 14
- 3.10 Regras de agrupamento de dados 15
- 3.11 Filtragem dos recursos 15
- 3.12 Validação dos eventos e medidas recebidos 16
- 3.13 Segurança do Monitor 17
- 3.14 Execução do sistema de monitorização 18
- 3.15 Políticas de verificação para os recursos sob monitorização 18
- 3.16 Armazenamento das medidas e eventos recolhidos 19
- 3.17 Armazenamento dos eventos e medidas normalizados 20
- 3.18 Deslocação dos dados para o *backup* 21

4.0 Requisitos não-funcionais 22
- 4.1 Elasticidade 22
- 4.3 Desempenho 22
- 4.5 Escalabilidade 22

5.0 Restrições 23
- 5.1 Restrições do projeto 23
- 5.2 Restrições do produto 23

1.0 Informação Geral

1.1 Propósito

O propósito deste documento de requisitos é fornecer aos leitores informação sobre os requisitos e contexto da solução idealizada para uma plataforma escalável de monitorização de *Cloud*. Esta solução passará pela implementação de *probes*, que servirão para recolher os dados a monitorizar, um Monitor que é responsável por processar os dados e por um *Knowledge base* chamado *Knowledge base* que irá guardar os dados recolhidos normalizados.

1.2 Âmbito

Este Documento de Requisitos serve para esboçar para que contexto esta solução foi pensada, perceber o problema, analisar todos os requisitos e clarificar o que a solução proposta terá de suportar.

1.3 Acrónimos e abreviações

JSON	Javascript Object Notation
KPI	Key Performance Indicator

2.0 Atores

Os atores da plataforma a desenvolver serão apenas dois, o sistema e o utilizador. A maior parte das funcionalidades da plataforma a desenvolver são executadas sem qualquer interação com o utilizador, por isso, espera-se que todo o funcionamento da plataforma seja, no geral, independente do utilizador.

3.0 Requisitos funcionais

Os requisitos funcionais são as funcionalidades que um sistema deve possuir para ser executado em ambiente de produção. Os requisitos vão ser mostrados formalmente sob a forma de casos de uso, pois é o tipo de definição de requisitos mais popular por várias razões. Uma das quais é a sua flexibilidade, simplicidade e clareza. Outras razões importantes para esta escolhe de definição de requisitos é destaca a identificação de problemas e incoerências, bem como a definição de pré-condições e pós-condições que auxiliam, mais tarde, na fase de testes, a testar as funcionalidades do sistema.

A todos os requisitos a seguir descritos foram-lhes atribuídas uma das seguintes prioridades:

- *Must* – Nesta prioridade encontram-se requisitos que são necessários para o sistema funcionar e sem eles o sistema não se encontra acabado;
- *Should* – Nesta prioridade encontram-se os requisitos que melhoram o sistema significativamente e portanto, são de extrema importância. Os requisitos desta categoria serão implementados depois dos requisitos classificados como *Must* forem implementados.
- *Could* – Os requisitos classificados com esta prioridade apenas adicionarão valor às funcionalidades já implementadas, pois são para ser implementados depois dos requisitos classificados como *Should*.

Os requisitos da plataforma de monitorização a desenvolver podem-se dividir em três grandes tarefas: monitorização em tempo-real, avaliar e verificar a configuração monitorizada e armazenar o histórico de eventos.

Para a representação dos casos de uso vai feita sob a forma de uma tabela, com os dados que se entenderam importantes para a elaboração dos mesmos. De modo a facilitar a compreensão dos casos de uso, de seguida encontra-se o *template* da tabela que serviu de orientação, com uma descrição do que consiste cada um dos campos.

<versão de edição>

Identificador Único	<identificador único, com uma referência textual para o caso de uso>
Prioridade	<prioridade na implementação do requisito>
Descrição	<breve parágrafo que descreve o objetivo que cada ator pretende alcançar de cada caso de uso>
Atores	<ator central das atividades descritas no caso de uso>
Pré-condições	<qualquer facto que o sistema possa assumir como verdadeiro aquando do começo do caso de uso >
Cenário Principal	<conjunto de passos que os atores devem fazer de modo a atingir o objectivo do caso de uso. Deve ser feita uma descrição clara do que o sistema faz em resposta a cada ação do utilizador>
Cenários Alternativos	<descreve as interações menos improváveis do ator com o sistema, como situações que não eram suposto acontecer ou que não levam ao sucesso da execução do caso de uso>
Pós-condições	<qualquer facto que tem de ser verdadeiro quando o caso de uso está completo>

Nas subsecções seguintes, irão ser descritos cada um dos casos de uso.

3.1 Integração de *probes*

v 1.0

ID	1.1
Prioridade	*Must*
Descrição	O sistema de monitorização deve ter *probes* de recolha de eventos e medidas que irão recolher informação sobre os recursos a monitorizar
Atores	Sistema
Pré-condições	As *probes* e o sistema de monitorização devem ter interfaces de rede para comunicarem
Cenário Principal	As *probes* recolhem e mandam os dados para o sistema
Cenários alternativos	Se ocorrer uma falha de rede, as *probes* e o monitor não serão capazes de comunicar.
Pós-condições	O sistema é capaz de receber os dados medidos

Tabela 1: Requisitos Funcionais: Integração das *probes*

3.2 Permissão de acesso aos dados recolhidos

v 1.0

ID	1.2
Prioridade	*Must*
Descrição	O sistema de monitorização deve permitir as ferramentas usadas para avaliação terem acesso aos dados guardados na base de dados. Caso não seja possível, deverá ser mostrada ao utilizador uma interface para adicionar os dados manualmente.
Atores	Sistema
Pré-condições	1. As *probes* recolhem os dados. 2. O Monitor recebe os dados. 3. O sistema de monitorização guarda os dados no *Knowledge base.*
Cenário Principal	As ferramentas acedem aos dados guardados no *Knowledge base*.
Cenário Alternativo	Se ocorrer uma falha de rede, as ferramentas de avaliação não serão capazes de aceder aos dados
Pós-condições	As ferramentas de avaliação terão acesso aos dados recolhidos

Tabela 2: Requisitos Funcionais: Permissão de acesso a dados

3.3 Monitorização Contínua dos dados

v 1.0

ID	1.3
Prioridade	*Must*
Descrição	O sistema de monitorização deve ser implementado de forma a recolher os eventos e monitorizar as medidas de forma contínua.
Atores	Sistema
Pré-condições	As *probes* e o sistema de monitorização devem ter interfaces de rede para comunicarem
Cenário Principal	1. As *probes* recolhem os dados. 2. O Monitor recebe os dados.
Cenário Alternativo	Se ocorrer uma falha de rede, as *probes* e o monitor não serão capazes de comunicar.
Pós-condições	O sistema recolhe os dados e guarda-os na base de dados.

Tabela 3: Requisitos Funcionais: Monitorização Contínua dos dados

3.4 Monitorização dos dados em várias tecnologias

v 1.0

ID	1.4
Prioridade	*Should*
Descrição	O sistema de monitorização deve recolher eventos e medidas a partir de várias tecnologias.
Atores	Sistema
Pré-condições	As *probes* e o sistema de monitorização devem ter interfaces de rede para comunicarem
Cenário Principal	As *probes* conectam-se a sistemas de várias tecnologias.
Cenário Alternativo	Se ocorrer uma falha de rede, as *probes* e o monitor não serão capazes de comunicar.
Pós-condições	O sistema recolhe os dados provindos de várias tecnologias e guarda-os na base de dados.

Tabela 4: Requisitos Funcionais: Monitorização dos dados em várias tecnologias

3.5 Retenção e Recuperação de *Logs*

v 1.0

ID	1.5
Prioridade	*Should*
Descrição	O sistema de monitorização deve reter os *logs* e recuperá-los.
Atores	Sistema
Pré-condições	1. As *probes* recolhem os dados. 2. O Monitor recebe os dados e guarda-os nos ficheiros de log.
Cenário Principal	Os eventos e medidas recolhidos são registados em ficheiros *log*
Cenário Alternativo	Falha no sistema de ficheiros na máquina do sistema de monitorização
Pós-condições	O sistema mantém os *logs* e, se for necessário é capaz de recuperação

Tabela 5: Requisitos Funcionais: Retenção e Recuperação de *Logs*

3.6 Autorização de recolha de dados

v 1.0

ID	1.6
Prioridade	*Must*
Descrição	As *probes* devem ter as autorizações necessárias para recolher os dados
Atores	Sistema
Pré-condições	As *probes* devem comunicar com os sistemas a monitorizar
Cenário Principal	As *probes* conectam-se aos sistemas a monitorizar
Cenário Alternativo	Falha na comunicação das *probes* com o sistema a monitorizar.
Pós-condições	As *probes* são capazes de recolher os dados do sistema e mandam para o Monitor.

Tabela 6: Requisitos Funcionais: Autorização de recolha de dados

3.7 Inserção manual de eventos e medidas

v 1.0

ID	1.7
Prioridade	*Could*
Descrição	O sistema de monitorização deve suportar o carregamento manual de eventos e medidas
Atores	Utilizador
Pré-condições	O Monitor deve estar a executar normalmente.
Cenário Principal	O utilizador deve fazer POST dos ficheiros com os eventos e medidas para o *endpoint* do monitor
Cenário Alternativo	1. Erro de validação do ficheiro JSON. 2. O sistema não está a funcionar corretamente devido a falhas na máquina em que está a ser executado.
Pós-condições	O sistema recebe os dados e guarda-os no *Knowledge base*.

Tabela 7: Requisitos Funcionais: Inserção manual dos dados

3.8 Normalização das medidas e eventos recolhidos

v 1.0

ID	1.8
Prioridade	*Should*
Descrição	O sistema de monitorização deve ser capaz de normalizar as medidas e os eventos de diferentes fontes para um único formato
Atores	Sistema
Pré-condições	1. As *probes* recolhem os dados. 2. O Monitor recebe os eventos e medidas e uniformiza-os
Cenário Principal	O sistema de monitorização normaliza os eventos e medidas recebidos das *probes*
Cenário Alternativo	1. Falha na receção dos eventos e das medidas por parte do Monitor. 2. Falha na comunicação com o *Knowledge base*.
Pós-condições	Os dados recolhidos ficam num formato normalizado

Tabela 8: Requisitos Funcionais: Normalização das medidas e eventos recolhidos

3.9 Processo de Normalização

v 1.0

ID	1.9
Prioridade	*Should*
Descrição	O processo de normalização por parte do sistema de monitorização deve ter um vocabulário claro e objetivo.
Atores	Sistema
Pré-condições	1. As *probes* são capazes de recolher os eventos e medidas. 2. O Sistema de monitorização recebe os dados provindos das *probes*.
Cenário Principal	O sistema de monitorização normaliza os eventos e medidas recebidos das *probes* com vocabulário claro e objetivo.
Cenário Alternativo	Falha da comunicação das *probes* com o Monitor
Pós-condições	Os dados são nomeados com os respetivos nomes e são guardados no *Knowledge base.*

Tabela 9: Requisitos Funcionais: Processo de Normalização

3.10 Regras de agrupamento de dados

v 1.0

ID	1.10
Prioridade	*Should*
Descrição	O agrupamento dos dados deve ser implementado segundo regras específicas.
Atores	Sistema
Pré-condições	1. As *probes* são capazes de recolher os eventos e medidas. 2. O Sistema de monitorização recebe os dados provindos das *probes*.
Cenário Principal	O Monitor agrupa os eventos e medidas recolhidas de acordo com algumas regras.
Cenário Alternativo	Falha da comunicação das *probes* com o Monitor.
Pós-condições	Os dados são agrupados segundo regras específicas e registados como uma única entrada no *Knowledge base*.

Tabela 10: Requisitos Funcionais: Regras de agrupamento de dados

3.11 Filtragem dos recursos

v 1.0

ID	1.11
Prioridade	*Must*
Descrição	O sistema de monitorização deve ser capaz de filtrar os dados de recursos específicos com base em regras definidas.
Atores	Sistema
Pré-condições	As *probes* são capazes de recolher os eventos e medidas.
Cenário Principal	As *probes* filtram os dados mais importantes para mandar para o Monitor.
Cenário Alternativo	Falha da comunicação das *probes* com o Monitor.
Pós-condições	Os eventos e medidas são guardados na *Knowledge base*.

Tabela 11: Requisitos Funcionais: Filtragem de recursos

3.12 Validação dos eventos e medidas recebidos

v 1.0

ID	1.12
Prioridade	*Must*
Descrição	O sistema de monitorização deve ser capaz de validar os eventos e medidas recolhidas pelas *probes* segundo o template JSON do projeto, senão forem válidos, os dados são descartados.
Atores	Sistema
Pré-condições	1. As *probes* são capazes de recolher os eventos e medidas. 2. O Monitor recebe os eventos e medidas recolhidos pelas probes.
Cenário Principal	O Monitor valida os eventos e medidas recebidos das *probes* com o template JSON do projeto.
Cenário Alternativo	Falha da comunicação das *probes* com o Monitor.
Pós-condições	1. Se eventos e medidas forem válidos são guardados na *Knowledge base*. 2. Se não forem válidos são descartados

Tabela 12: Requisitos Funcionais: Validação dos eventos e medidas recolhidos

3.13 Segurança do Monitor

v 1.0

ID	1.13
Prioridade	*Must*
Descrição	Adicionar o protocolo de segurança TLS ao Monitor
Atores	Sistema
Pré-condições	Geração dos certificados para todos os componentes da plataforma
Cenário Principal	Todos os componentes devem comunicar com o Monitor através do protocolo TLS.
Cenário Alternativo	1. Falha na comunicação com o Monitor. 2. Certificados não válidos.
Pós-condições	Todos os componentes comunicam com o Monitor.

Tabela 13: Requisitos Funcionais: Segurança do Monitor

3.14 Execução do sistema de monitorização

v 1.0

ID	2.1
Prioridade	*Must*
Descrição	O sistema de monitorização deve ser executado com base num evento definido
Atores	Sistema
Pré-condições	Definição do evento
Cenário Principal	O sistema de monitorização é lançado quando um determinado *trigger* for disparado.
Cenário Alternativo	Falha na construção do *setup* do sistema de monitorização
Pós-condições	O sistema de monitorização começa a executar.

Tabela 14: Requisitos Funcionais: Execução do sistema de monitorização

3.15 Políticas de verificação para os recursos sob monitorização

v 1.0

ID	2.2
Prioridade	*Must*
Descrição	O sistema de monitorização deve permitir múltiplas políticas de verificação dos recursos sob monitorização
Atores	Sistema
Pré-condições	1. As *probes* são capazes de recolher os eventos e medidas. 2. O Sistema de monitorização recebe os dados provindos das *probes*.
Cenário Principal	São aplicadas várias políticas aos sistemas monitorizados
Cenário Alternativo	1. Falha da comunicação das *probes* com o Monitor.
Pós-condições	Os sistemas monitorizados começam-se a comportar com base nas políticas aplicadas

Tabela 15: Requisitos Funcionais: Políticas de verificação para os recursos sob monitorização

3.16 Armazenamento das medidas e eventos recolhidos

v 1.0

ID	3.1
Prioridade	*Must*
Descrição	O sistema de monitorização deve armazenar todos os eventos e medidas recolhidos num container central chamado *Knowledge base*
Atores	Sistema
Pré-condições	1. As *probes* são capazes de recolher os eventos e medidas. 2. O Sistema de monitorização recebe os dados provindos das *probes*.
Cenário Principal	Os eventos e as medidas são enviados para o *Knowledge base*
Cenário Alternativo	1. Falha da comunicação das *probes* com o Monitor.
Pós-condições	Os eventos e as medidas ficam guardadas numa base de dados.

Tabela 16: Requisitos Funcionais: Armazenamento das medidas e eventos recolhidos

3.17 Armazenamento dos eventos e medidas normalizados

v 1.0

ID	3.2
Prioridade	*Must*
Descrição	As medidas e eventos recolhidos devem ser armazenados num formato normalizado
Atores	Sistema
Pré-condições	1. As *probes* são capazes de recolher os eventos e medidas. 2. O Sistema de monitorização recebe os dados provindos das *probes*.
Cenário Principal	Os eventos e as medidas normalizados são enviados para o *Knowledge base*
Cenário Alternativo	1. Falha da comunicação das *probes* com o Monitor. 2. Falha na comunicação entre o Monitor e o *Knowledge base*.
Pós-condições	Os eventos e as medidas normalizados ficam guardadas numa base de dados.

Tabela 17: Requisitos Funcionais: Armazenamento das medidas e eventos recolhidos

3.18 Deslocação dos dados para o *backup*

v 1.0

ID	3.3
Prioridade	*Should*
Descrição	A retenção dos dados deve ser definida para que mover os eventos e medidas para o backup
Atores	Sistema
Pré-condições	1. As *probes* são capazes de recolher os eventos e medidas. 2. O Sistema de monitorização recebe os dados provindos das *probes*. 3. Os eventos e as medidas são guardados no *Knowledge Base*.
Cenário Principal	Ao fim de algum tempo, os dados são movidos para o *backup*
Cenário Alternativo	1. Falha da comunicação das *probes* com o Monitor. 2. Falha na comunicação entre o Monitor e o *Knowledge base*. 3. Falha de comunicação entre o *Knowledge Base* e o sistema de *backup*
Pós-condições	Os dados são armazenados no sistema de *backup*.

Tabela 18: Requisitos Funcionais: Deslocação dos dados para *backup*

4.0 Requisitos não-funcionais

Os requisitos não-funcionais, também conhecidos como atributos de qualidade, servem para definir um limite mínimo de um determinado parâmetro, sobre o qual o sistema pode ser avaliado. A definição desse limite pode ter impacto no sistema todo ou apenas numa parte dele.

Sendo este projeto uma parte de um projeto com mais componentes do que os desenvolvidos neste projeto, alguns dos requisitos não-funcionais apresentados aplicam-se a todo o projeto, não apenas ao sistema de monitorização. Assim, nos sub-capítulos seguintes irão ser apresentados os requisitos não-funcionais que se consideram pertinentes para uma solução como a deste projeto.

4.1 Elasticidade

O componente TMA_MONITOR deverá ser escalado automaticamente consoante a carga que recebe.

4.3 Desempenho

A plataforma não deverá ter um impacto negativo na performance de outras plataformas que a possam vir a usar aquando das suas funções de monitorização.

4.5 Escalabilidade

Toda os componentes da plataforma deverão ser escaláveis.

5.0 Restrições

Este capítulo servirá para identificar as restrições impostas a este projeto, independentemente da sua natureza. Estas restrições impostas ou identificadas podem ser divididas em duas categorias diferentes: restrições do projeto e restrições do produto, que serão os temas abordados nos sub-capítulos seguintes.

5.1 Restrições do projeto

Estas são as restrições associadas ao projeto em si e não estão relacionadas diretamente à solução a implementar. Em relação às ferramentas utilizadas, todas elas devem ser open-source.

5.2 Restrições do produto

Existem algumas restrições em relação ao produto a ser desenvolvido que vão ser apresentadas a seguir:

- Tecnologia: O sistema vai ser desenvolvido em Python.
  O DBMS usado vai ser MySQL, para ser compatível com outros componentes da plataforma.
- Formato dos dados: Todos os dados recolhidos pelas *probes* deverão estar no formato JSON segundo o template específico do projeto.