INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Área Departamental de Engenharia de Eletrónica e Telecomunicações e de Computadores

Profiling KPIs in Highly Scalable Networks

JOÃO PEDRO CRISTO RIBEIRO

Licenciatura Engenharia Informática e de Computadores

Dissertação para obtenção do Grau de Mestre
em Engenharia Informática e de Computadores

Orientadores : Professor Doutor José Manuel de Campos Lages Garcia Simão
Professor Doutor Nuno Miguel Soares Datia

Júri:

Presidente: Professor Doutor Nuno Miguel Machado Cruz

Vogais: Professor Doutor Luís Manuel da Costa Assunção
Professor Doutor José Manuel de Campos Lages Garcia Simão

Dezembro, 2020

INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Área Departamental de Engenharia de Eletrónica e Telecomunicações e de Computadores

Profiling KPIs in Highly Scalable Networks

JOÃO PEDRO CRISTO RIBEIRO

Licenciatura Engenharia Informática e de Computadores

Dissertação para obtenção do Grau de Mestre
em Engenharia Informática e de Computadores

Orientadores : Professor Doutor José Manuel de Campos Lages Garcia Simão
Professor Doutor Nuno Miguel Soares Datia

Júri:

Presidente: Professor Doutor Nuno Miguel Machado Cruz

Vogais: Professor Doutor Luís Manuel da Costa Assunção
Professor Doutor José Manuel de Campos Lages Garcia Simão

Dezembro, 2020

Aos meus Pais e à Mónica.

Agradecimentos

Tenho muitas pessoas a quem devo agradecer, tanto no meu percurso académico como na formulação da dissertação. Quero agradecer aos meus orientadores Professor Doutor José Simão e Professor Doutor Nuno Datia por me darem esta oportunidade e por me terem acompanhado em todo este trajeto. Aos meus companheiros de curso Rui, Rafael e Marco que nunca me deixaram ficar mal. À Mónica a minha namorada que me deu força e que se não fosse por ela, esta tese não existiria. Aos meus pais que me suportaram em todos os momentos do meu percurso académico.

Resumo

As redes de nova geração, como as redes Software Defined Networks (SDN) [55], são redes de larga escala e diversidade no fluxo de dados que geram. Os dados são provenientes dos inúmeros dispositivos que constituem as redes e, como tal, a monitorização destes dispositivos e do fluxo de dados que por eles passa, é uma necessidade cada vez mais presente, levando a que sejam criados sistemas de monitorização e controlo dos mesmos. A monitorização dos dados é essencial para garantir a boa qualidade do serviço de rede, ajudando a detetar falhas, e identificando padrões e métricas úteis para o negócio.

Uma solução que visa fornecer estes tipos de serviços é o produto *Service Quality Manager* (SQM) da empresa Nokia. Este *software* tem a capacidade de agregar e avaliar a informação fornecida por dispositivos de rede, disponibilizando indicadores de desempenho, ou *Key Performance Indicators* (KPIs), configuráveis e úteis a outros sistemas, administradores de rede e outros utilizadores. A arquitetura atual do SQM usa uma abordagem monolítica que não tem a flexibilidade e a escalabilidade que se pretende. A sua evolução passa por definir uma arquitetura que permita responder aos desafios dos dias de hoje, transformando a presente solução num sistema distribuído, escalável e que consiga processar um grande volume de dados em tempo útil.

A solução apresentada nesta tese baseia-se na arquitetura *SMACK Stack* (Spark, Mesos, Akka, Cassandra, Kafka)[40]. Na solução proposta, tal como na arquitetura *SMACK Stack*, pretende-se dividir as múltiplas responsabilidades da aplicação em diversas camadas. O objetivo é englobar várias tecnologias que cooperam entre si para criar uma solução distribuída e escalável em tempo real.

Palavras-chave: Arquiteturas distribuídas, *Cloud*, Escalabilidade, Elasticidade, Indicadores de desempenho, Kubernetes, Virtualização baseada em contentores, Ambientes computacionais de alto desempenho

Abstract

New-generation networks, such as Software Defined Networks (SDN) networks[55], are networks that have a large scale and diversity in the data flow they generate.

The data comes from the numerous devices that constitute the networks and as such the monitoring of these devices and the data flow that passes through them is an increasingly needed, leading to the creation of monitoring and control software. Data monitoring is essencial to ensure good service quality, easily detecting failures, patterns and metrics that are useful for the business.

One solution that provide these types of services is the Service Quality Manager (SQM) solution, presented by Nokia. This software has the ability to aggregate and evaluate the information provided by network devices, providing configurable and useful *Key Performance Indicators* (KPIs) to other systems, network administrators and other users. Since the present solution is a monolithic architecture solution it does not have the flexibility and scalability that is intended.

It is intended to evolve the solution to an architecture that allows us to respond to the today's challenges, transforming the present solution into a distributed system, easily scalable and able to process a large volume of data in real time.

The solution presented in this thesis is based on the SMACK Stack [40] architecture. In the proposed solution, as in the SMACK Stack architecture, is to divide the multiple responsibilities of the application into several layers. The goal is to embrace a number of collaborative technologies to create a scalable, real-time solution.

Keywords: Distributed architectures, Cloud, Scalability, Elasticity, Key Performance Indicators, Kubernetes, Container-based virtualization, High-performance computing environments

Índice

Lista de Figuras **xvii**

1 Introdução **1**

1.1 Problema 2

1.2 Objetivo 3

1.3 Requisitos do Sistema 3

1.4 Resumo da solução 4

1.5 Organização do Documento 4

2 Estado da Arte e Trabalho Relacionado **5**

2.1 Introdução 5

2.2 Base de dados 6

2.2.1 Cassandra 6

2.2.2 HBase 7

2.2.3 MongoDB 8

2.2.4 Análise comparativa 8

2.3 Micro-serviços e Contentores 10

2.3.1 Docker 11

2.4 Ambiente de Execução 12

2.4.1 Kubernetes 13

2.4.2 Mesos 13
2.5 Arquiteturas 14
2.5.1 LAMP *stack* 14
2.5.2 Computação em Nuvem 15
2.5.3 Arquitetura de Micro-serviços 16
2.6 Sumário 17

3 Arquitetura Proposta **19**
3.1 Requisitos funcionais 20
3.2 Arquitetura 22
3.2.1 Camada de armazenamento 24
3.2.2 Camada de processamento 24
3.2.2.1 Serviço de *Report* 25
3.2.2.2 Serviço de *Profile* 25
3.3 Ambiente de execução 26
3.3.1 Cenário de execução 26
3.4 Interface gráfica 27
3.5 Resumo 27

4 Implementação **29**
4.1 Implementação de serviços 29
4.1.1 Criar imagens Docker 30
4.1.2 Utilização da biblioteca 31
4.1.3 Exposição de serviços no Kubernetes 32
4.2 Objectos chave no ambiente de execução 33
4.2.1 *Deployments* 33
4.2.2 *Services* 34
4.2.3 *CronJob* 34
4.2.3.1 *CronJob Raw* 35
4.2.3.2 *CronJob Reporter* 35

4.2.3.3 *CronJob Profile* 35
4.2.4 *StatefulSet* 36
4.2.4.1 *StatefulSet* Cassandra 36
4.2.4.2 *StatefulSet* MySql 38
4.2.5 Interface gráfica 39
4.3 Interação dos módulos 41
4.3.1 Adição de novos módulos 41
4.3.2 Novos KPIs 42
4.4 Deploy da aplicação na cloud 42
4.5 Sumário 43

5 Análise de resultados 45
5.1 Aspetos relevantes 45
5.1.1 Disponibilização da solução 46
5.1.2 Robustez da solução 47
5.1.2.1 Falha na execução de um *Pod* 47
5.1.2.2 Falha de um *Pod* Cassandra 47
5.1.2.3 Falha de um *Pod* de MySQL 47
5.1.2.4 Falha na execução de uma *query* 47
5.1.3 Atualização da solução 48
5.1.3.1 Impacto da atualização no sistema 48
5.1.4 Deploy da aplicação 48
5.1.5 Flexibilidade 49
5.1.5.1 Atualização do sistema 49
5.1.6 Escalabilidade 49
5.2 Análise de desempenho 50
5.2.1 Testes locais 50
5.2.2 Testes na cloud 51
5.2.3 Conclusões da análise 53

6 Conclusão e Trabalho Futuro **57**

6.1 Conclusão 57

6.2 Dificuldades 59

6.3 Trabalho Futuro 60

Referências **61**

Lista de Figuras

2.1 Tabela comparativa das diferentes Base de dados 10
2.2 Diferenças entre contentores Docker e VMs 11
2.3 Estrutura de um contentor Docker 12
2.4 Arquitetura Smack stack 14
2.5 Comparação da arquitetura LAMP stack e SMACK stack 15

3.1 Linha temporal do ciclo de geração de KPIs 21
3.2 Proposta de Arquitetura 23
3.3 Diagrama de sequência de um pedido do utilizador para obter um KPI 27

4.1 Comparação de YAML para java usando fabric8io 32
4.2 Formulário de parametrização do *Report* 39
4.3 Formulário de parametrização do *Profile* 39
4.4 Dashboards para o KPI BitRate 40

5.1 Cenário 1 - Tempo de processamento por blocos de 100 registos por 80 workers num total de 8000 registos 50
5.2 Cenário 2 -Tempo de processamento por blocos de 1000 registos por 8 *workers* num total de 8000 registos 51
5.3 Cenário 3 - Tempo de processamento por blocos de 500 registos por 16 *workers* num total de 8000 registos 52

5.4 *Dashboard* Kubernetes de execução do projeto na Google Cloud . . 52

5.5 Comparação do processamento sequencial e paralelo de *workers* a processar 1000 registos com tempo de execução de 3 minutos, num total de 8000 registos 53

5.6 Tempo médio de processamento da execução de cada *Pod* nos três testes dos cenários locais 54

5.7 Comparação do tempo de execução do processamento de forma paralela e sequencial 55

1 Introdução

O mercado das tecnologias de informação está mais exigente, sendo necessário um processo contínuo de evolução e inovação. Reflexo disso são as redes de nova geração, que apresentam maiores larguras de banda, disseminação mais alargada, ligando diferentes dispositivos que produzem um crescente fluxo de dados [47]. Como tal, é importante conceber e desenvolver soluções que consigam fornecer métricas em tempo real, para que seja possível detetar rapidamente falhas, congestão e degradação na rede. Na verdade, a célere deteção de problemas garante a estabilidade, desempenho e a qualidade de serviço em toda a rede. É possível fazer essa medição através de *key performance indicator* (KPI), pois mede o desempenho de um resultado. Um KPI pode indicar eficiência, eficácia, qualidade, oportunidade, observância, comportamento, economia e utilização de recursos.

Uma aplicação de monitorização deve ser flexível e escalável, pois o fluxo de dados numa rede não é constante, tendo momentos de elevado *stress* e outros de pouca utilização. As soluções com arquitetura monolítica não conseguem, muitas vezes, ter a capacidade de acompanhar as necessidades anteriormente referidas, pois não possuem a flexibilidade e a elasticidade sobre os recursos utilizados. Estes sistemas apenas têm a capacidade de escalar verticalmente através da adição de mais CPU, memória ou disco, na máquina que suporta a aplicação. Tendo em consideração as limitações anteriores, é necessário planear as alterações a efetuar nas aplicações para responsabilizar cada componente pela sua função numa

arquitetura distribuída.

Uma solução distribuída é constituída por vários módulos que contribuem com responsabilidades singulares para um objetivo comum. A sua flexibilidade não é apenas limitada aos recursos, pois é possível adicionar novos módulos de forma a facilmente incrementar o número de funcionalidades de uma aplicação. Devendo ter sempre em conta os impactos no desempenho e/ou na escalabilidade.

As características de um sistema distribuído fazem com que seja mais facilmente escalável, heterogéneo quanto aos tipos de rede e hardware envolvidos, e que consiga paralelizar algumas das tarefas complexas em várias tarefas simples, de forma a que a obtenção de resultado em tempo útil seja possível.

A definição da melhor abordagem na modularização de uma aplicação monolítica é uma tarefa importante e essencial para que seja possível obter uma arquitetura que conjugue as múltiplas responsabilidades da aplicação existente , tendo em vista a criação de módulos constituintes de uma aplicação distribuída.

1.1 Problema

A aplicação SQM da Nokia, como acontece em tantas outras aplicações monolíticas, já não serve os requisitos impostos pelas redes de nova geração. Na verdade, encontra-se limitada a uma aplicação com uma base de dados centralizada e que processa um número limitado de pedidos.

A aplicação de momento não tem uma arquitetura modular e encontra-se por isso restringida a dependências entre as diferentes classes que a constituem. Como consequência do anteriormente relatado, devido ao fluxo de informação recebido, é expectável que a aplicação seja incapaz de dar os resultados em tempo útil. Estes resultados são utilizados para notificar os operadores que têm a responsabilidade de reagir atempadamente, a fim de evitar problemas devido a eventos na rede.

Dada a complexidade da presente solução (SQM), é indispensável que seja efetuada uma interpretação detalhada das suas características.

Adicionalmente é imprescindível definir responsabilidades por forma a criar módulos independentes e efetuar uma seleção dos melhores produtos que respondam às necessidades da arquitetura e do cenário.

1.2 Objetivo

O objetivo da presente tese é propor e demonstrar uma arquitetura, que utiliza tecnologias atuais para produzir em tempo real KPIs sobre dados disponibilizados por dispositivos de rede. A arquitetura proposta deve ter em conta o futuro do produto quanto à sua elasticidade, desempenho e escalabilidade. A solução final deve estar preparada para funcionar no ambiente *cloud*, utilizando os seus recursos de forma dinâmica. Devido à necessidade da elasticidade, o sistema deve ter a capacidade aumentar ou diminuir os recursos e a capacidade de resposta dos pedidos, baseando-se na carga de trabalho a que está submetido. Dado que já existe uma aplicação com arquitetura monolítica que agrega os dados faz parte do objetivo encontrar uma forma de reutilizar a solução anterior na nova arquitetura.

O produto final deve ter em conta a robustez, utilizando mecanismos de recuperação a falhas no armazenamento e no processamento de dados.

1.3 Requisitos do Sistema

Para interpertar os dados não tratados e extrair KPIs, a solução deverá ter em conta os seguintes requisitos funcionais:

- Os KPIs são retirados de uma amostra que tem em conta uma janela temporal e uma periodicidade;
- Capacidade de detetar e notificar a ocorrência de valores atípicos (*outliers values*);
- Interface REST para acesso remoto;
- A arquitetura deve envolver tecnologias *open-source*;
- Manipulação de dados não estruturados.

Para além disso a arquitetura deve utilizar recursos presentes da *cloud* para o armazenamento e processamento com orquestração de um ambiente de execução.

1.4 Resumo da solução

O funcionamento da presente solução inicia-se no consumo de uma base dados que contém informação não processada, fornecida por múltiplos equipamentos de rede. A entidade que consome e agrega essa informação tem o nome de *Reporter*. Este cria relatórios, que representam uma extração de informação numa periodicidade configurada por um utilizador. Depois de o *Report* ser gerado é guardado numa base de dados, para ser consumido pelo módulo de classificação denominado T&P ou *Profiler* que calcula, avalia e fornece KPIs. Consoante os valores dos KPIs, pode existir a necessidade de notificar o utilizador através de vários meios, como E-Mail e SMS. Uma estratégia de paralelização dos módulos de *Reporter* e *Profiler* vão ser essenciais para o desempenho do sistema, conseguindo fazer agregações e cálculos de diferentes KPIs em simultâneo.

1.5 Organização do Documento

O documento encontra-se organizado da seguinte forma:

No Capítulo 1 introduz o cenário inicial e a sua problemática. São também apresentados os objetivos que a dissertação se compromete a cumprir na solução proposta.

No Capítulo 2 é efetuada uma comparação e uma análise das tecnologias candidatas à arquitetura. É também neste capítulo que é apresentado o trabalho relacionado.

Segue-se a descrição da arquitetura proposta cuja a apresentação é efetuada no Capítulo 3, onde são descritos os componentes que constituem a arquitetura e a sua responsabilidade no sistema.

A apresentação de uma proposta de solução na forma de prova de conceito é apresentada e aprofundada no Capítulo 4.

No Capítulo 5 são avaliados os resultados da implementação da arquitetura proposta e o cumprimento dos objetivos propostos.

No Capítulo 6, é levada a cabo uma reflexão sobre o trabalho realizado, sendo apresentados os pontos a desenvolver no futuro, e que não foram abordados nesta dissertação.

Estado da Arte e Trabalho Relacionado

2.1 Introdução

Existe uma grande variedade de ferramentas disponíveis no mercado que desempenham as funções necessárias numa arquitetura. Dado não existir uma ferramenta ideal que funcione em todos os cenários, é necessário escolher segundo os requisitos da atual aplicação. São critérios para a escolha das ferramentas os seguintes: a quantidade de acessos à aplicação; a otimização de escritas em deterioramento das leituras ou a necessidade de utilizar *frameworks* para o processamento de dados.

A decisão da escolha das tecnologias utilizadas na arquitetura não deve ser influenciada, por outras razões que não estejam relacionadas com características da própria tecnologia ou do cenário em que se enquadra a arquitetura. A escolha deve ter em consideração todas as responsabilidades e funcionamento do sistema de forma isolada. Se assim não for, haverá impacto durante todo o processo de implementação da arquitetura bem como no futuro da aplicação.

Pretende-se encontrar as ferramentas *open-source* que tenham as características ideais para criar uma solução distribuída na monitorização, processamento e disponibilização de indicadores para redes de alto desempenho.

Para a nova arquitetura pretende-se uma base de dados distribuída capaz de guardar a informação para que a mesma seja consumida pelos diferentes módulos, num ambiente de execução onde vai existir o processamento de dados do sistema de *Report* e *Profile*, disponíveis através de micro-serviços [38]. Os módulos constituintes da arquitetura baseiam-se na Arquitetura *SMACK Stack* [40].

2.2 Base de dados

O armazenamento de dados para este cenário, requer uma base de dados distribuída, sobre dados não estruturados. Estes são provenientes de dispositivos de rede que criam registos que não se adequam a uma base de dados relacional.

A utilização de uma base de dados NoSQL [52] é, neste caso, uma vantagem, pois possui maior capacidade de escalabilidade face a um esquema relacional, devido à sua estrutura e dados que manipula. Por seu lado, a sua capacidade de distribuição contribui para um menor tempo de resposta nas consultas, ainda que com consistência eventual, pois as leituras têm de ser propagadas. Desta forma, o NoSQL permite um alto desempenho com elevada disponibilidade.

A informação das características dos dados na solução existente não é clara. Tendo apenas acesso a alguns exemplos de utilização, em que os dados estão em formato numérico, assume-se que exista outros tipos de dados envolvidos, como *timestamps* que identificam o momento em que foi feita a leitura, uso de memória ou CPU, número de erros no processamento de pacotes e largura banda [16].

2.2.1 Cassandra

A base de dados *Apache Cassandra* [42] trata-se das melhores opções na área das bases de dados não relacionais. Esta é uma base de dados *open-source* disponibilizada pelo fornecedor de soluções Apache, que disponibiliza uma solução de base de dados num sistema distribuído em NoSQL que se destaca pelo seu desempenho [1]. Uma das suas características é o mecanismo de tolerância a falhas, que resulta da disponibilização de múltiplas réplicas atualizadas, permitindo que um nó com problemas seja rapidamente substituído por outro, sem comprometer a disponibilidade para o cliente. Esta flexibilidade é possível graças à permanência de todos os nós no mesmo estado e com a mesma informação, o que faz com que não haja um ponto de falha.

Esta solução é também reconhecida pela sua escalabilidade, que ocorre de forma elástica e que permite aumentar e diminuir os recursos que sustentam a base de dados sem *downtime* [24]. O processo de replicação que ocorre entre os nós tem a opção de ser assíncrono ou síncrono, dado uma possibilidade mais concreta de um maior controlo na forma e na escolha de como os mesmos são atualizados.

O Cassandra funciona com uma linguagem semelhante ao SQL, utilizando instruções de *Data Manipulation Language* (DML) e *Data Definition Language* (DDL), denominada *Cassandra Query Language* (CQL). São já inúmeros os sistemas que utilizam atualmente a solução Cassandra, destacam-se: o eBay, com mais de 100 nós e 250 TB (terabyte); a Netflix, com 2,500 nós, 420 TB e um 1 trilião de pedidos por dia; e ainda a Apple, com mais de 75,000 nós e com um armazenamento de mais de 10 PB de dados [27].

2.2.2 HBase

O HBase [32] trata-se de outra base de dados *open-source* do fornecedor de soluções Apache, que tal como a solução apresentada anteriormente, trabalha com um grande volume de dados e com NoSQL. O HBase partilha muitas características do Cassandra, tal como a escalabilidade e a capacidade de trabalhar com um grande volume de dados. Apesar de, à primeira vista, parecer que tem características semelhantes à solução Cassandra, o mesmo não acontece, pois cada uma delas tem os seus pontos fortes e as suas falhas, sendo também por isso que não existe uma ferramenta que se adeque a todos os cenários.

No cenário a que o HBase se adequa, a informação é guardada em tuplos de dados não tipificados, em que não são necessárias ferramentas que as base de dados relacionais providenciam como índices, transações e a necessidade de uma linguagem para interrogações complexas. Um requisito também importante para a utilização do HBase, é ter número mínimo de nós para replicação.

Tendo em consideração a análise efetuada e o conhecimento à priori da característica dos dados de rede (não estruturados), considera-se que o HBase vai de encontro aos requisitos que o sistema necessita.

O ponto que diferencia o HBase [43] de outras bases de dados NoSQL é a sua consistência na leitura e escrita de dados. Este ponto de destaque é conseguido como consequência de uma arquitetura onde são utilizados *Region Servers*. Estes são responsáveis pelo processamento de partes das tabelas, sendo que o serviço

que gere todos estes servidores é o Zookeeper [2], que não é mais que um serviço distribuído que acede às configurações de todos os servidores presentes na *Meta Table* (também ela um *Region Server*), que é responsável pela sincronização distribuída e pela disponibilização de ferramentas que possibilitam gerir o sistema com um todo. Por outro lado, a manipulação dos dados pelo HBase pode ser realizada através de linguagem Java e/ou de uma API, em alternativa a uma linguagem semelhante a SQL.

2.2.3 MongoDB

A solução MongoDB [35] oferece uma alternativa aos produtos do fornecedor de soluções da Apache. A solução MongoDB também ela *open-source* é bem conhecida no mercado, possui uma grande e ativa comunidade [14].

A solução MongoDB trata-se de uma base de dados que guarda estes últimos em documentos, com um formato muito semelhante a objetos JSON. É uma abordagem diferente das clássicas bases de dados que utilizam tabelas, o que permite adicionar, em campos de cada tuplo, valores complexos como outros documentos, *arrays* e *arrays* de documentos. É também possível mapear diretamente este documento para uma instância de objeto, constituída por tipos nativos a muitas linguagens de programação. Esta capacidade faz com que MongoDB seja compatível com grande parte das linguagens utilizadas no mercado. Além das características já expostas, a solução MongoDB vem com um sistema de replicação com auto eleição. Significa isto que caso a base de dados principal falhe, outra base de dados secundária (réplica) pode ser eleita e assumir a base de dados principal.

Uma vez que MongoDB funciona com um único *master*, enquanto o processo de auto eleição está em execução só são permitidas operações de leitura. É importante referir que a configuração deste sistema de replicação não é trivial, requerendo, como tal, competências técnicas avançadas sobre a aplicação, o que dificulta o acesso e capacidade de utilização a utilizadores com pouca experiência. Os tipos de aplicação em que se verifica uma maior aderência da solução MongoDB são aplicações como *Internet of Things* (IoT)[10] e aplicações para dispositivos móveis [15].

2.2.4 Análise comparativa

Todas as tecnologias de base de dados apresentadas são baseadas nos mesmo requisitos. Efetivamente, são bases de dados *open source*, distribuídas, escaláveis,

com dados não estruturados e que funcionam com uma grande quantidade de dados. As bases de dados apresentadas têm todos os requisitos, como o facto de serem distribuídas, ser *open-source* e utilizarem NoSQL. Assim sendo, os pontos fortes e fracos de cada uma (figura 2.1) no cenário apresentado vai ser determinante na escolha da base de dados utilizada. O Cassandra oferece uma solução com escalabilidade com mínima administração e grande grau de fiabilidade. Ao contrário de MongoDB, é possível ter a solução instalada em múltiplos servidores sem grande complexidade, nem necessidade de configurações que são automatizadas pelo *NoSQL Data Management System* (NDBMS) Cassandra. Utiliza CQL, que é mais semelhante a SQL que a linguagem utilizada por MongoDB. Paralelamente, é uma melhor escolha quando existe uma grande quantidade dados não estruturados e temos como previsão o rápido crescimento dos mesmos. Visto que os dados são fornecidos por equipamentos de rede em grande quantidade, a capacidade de crescimento do Cassandra sobrepõe-se à vantagem de armazenamento em formato de documentos do MongoDB [39]. Desta forma, para o cenário apresentado, o Cassandra é a opção mais indicada. A decisão entre o Cassandra e o HBase trata-se de uma escolha mais difícil, devido à sua semelhança nos elementos chave. Posto isto, torna-se necessária uma análise mais detalhada das suas características. O Cassandra tem uma arquitetura *masterless*, e o Hbase tem uma arquitetura baseada num *master*. Isso significa que o Cassandra não tem um único ponto de falha, apesar de no HBase o cliente poder comunicar diretamente com um *slave*, mesmo que o *master* esteja em baixo. Assemelha-se, pois, a uma solução de recurso, comparado com a solução sem interrupções do Cassandra. Para que o NDBMS se mantenha sem interrupções, o Cassandra replica e duplica os dados, sacrificando a consistência e, assim, trazendo alguns problemas. No Hbase, as escritas são feitas num único sítio, nomeadamente no *master*. Desta forma, sabe-se facilmente onde os dados se encontram, não possuindo os problemas de consistência do Cassandra.

	Cassandra	MongoDB	Hbase
Desenvolvido Por	Apache Software Foundation	Mongo DB	Apache Software Foundation
Licença	*Open Source*	*Open Source*	*Open Source*
Linguagem de implementação	Java	C++	Java
Modelo	Orientado a colunas	Orientado a Documentos	Orientado a colunas
Triggers	Sim	Não	Sim
Concorrência	Sim	Sim	Sim
Indexes Secundários	Sim, com restrições	Sim	Não
Consistência	Consistência eventual e imediata	Consistência eventual e imediata	Consistência imediata
Replicação	*Replica Selection*	*Replica Master-Slave*	*Replica Selection*
Funcionalidades Chave	Escalabilidade incremental; Administração e instalação com pouca complexidade; Sem ponto de falha única; Linguagem semelhante a SQL; Disponibilidade elevada.	Multiplataforma; Replicação com auto eleição (Master-Slave); alta performance devido ao modelo orientado a ojetos e indexes.	Consistência imediata de leituras e escritas; Escalabilidade linear; Utilização de *Region Servers* e *Zookeeper*

Figura 2.1: Tabela comparativa das diferentes Base de dados

2.3 Micro-serviços e Contentores

O desafio de adaptar uma solução monolítica a uma solução distribuída obriga a uma modularização dos componentes em diferentes responsabilidades. Uma das formas de separar os diferentes módulos é recorrer a uma estratégia de micro-serviços [38], que consiste em utilizar um conjunto de serviços autónomos, que executam, de forma independente,um único objetivo [41, 53]. Estes serviços podem utilizar linguagens de programação e tecnologias de armazenamento diferentes. Uma das principais diferenças entre um micro-serviços e um serviço "clássico" é a sua granularidade. Os micro-serviços existem em grande quantidade, pois são utilizados para realizar um único propósito de forma muito eficiente. Um serviço por sua vez agrega conjuntos de funções muitas delas complexas, existindo em menor quantidade comparativamente a micro-serviços.

No cenário apresentado, os módulos que criam *Reports* e os módulos que realizam os *Profiles* dos dados são funcionalidades candidatas a micro-serviços. Uma forma de disponibilizar as funcionalidades de código já existente de forma isolada é utilizar a abordagem de contentores. Os Contentores são uma abstração que isola uma aplicação e as suas dependências, de forma a poder executá-las de forma independente.

2.3.1 Docker

Uma das primeiras implementações de contentor com impacto no mercado foi o Docker [23]. Com um contentor Docker, é possível manter a consistência na execução independentemente do ambiente. Paralelamente, obtém-se controlo sobre a quantidade de recursos disponíveis por contentor e a utilização de contentores Docker com tecnologias, para mais facilmente integrar em soluções já existentes. São mais favoráveis ao cenário apresentado as características dos contentores Docker do que a alternativa de utilizar VMs (*Virtual Machines*). As VMs são uma tecnologia muito utilizada no mercado. No entanto, são componentes mais pesados comparativamente aos contentores. Esta diferença ocorre, pois os contentores só incluem o essencial para executar como: o código-fonte, *runtime*, as bibliotecas e configurações de execução. Para além disso, o sistema operativo que corre num contentor partilha o Kernel com o *host OS*. Por conseguinte, ocupa menos espaço e facilita a de informação, existindo menos níveis de abstração do que numa VM [51]. Cada VM vem com um *Guest OS*, que se traduz num sistema operativo que corre sobre o sistema operativo *host*. Na Figura 2.2, é possível verificar a diferença entre as duas estruturas.

Não só a dimensão de cada VM, mas também o tempo que cada uma leva a iniciar torna a opção de contentores Docker muito mais indicada para o cenário apresentado [50, 54].

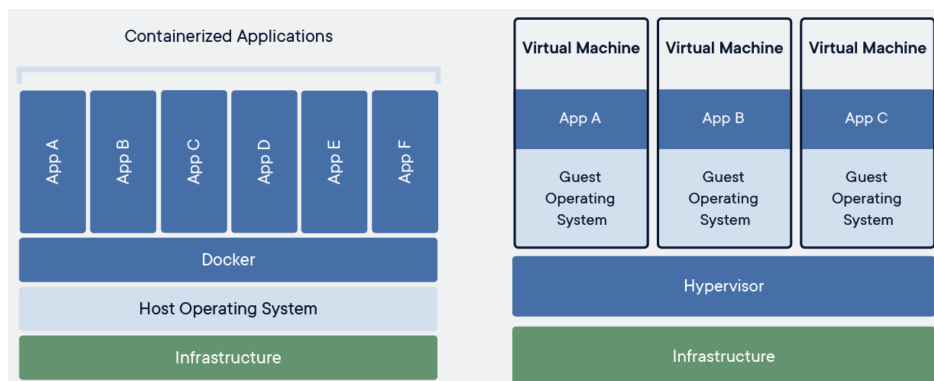

Figura 2.2: Diferenças entre contentores Docker e VMs (imagem obtida de [23])

Os contentores são considerados uma alternativa às VMs [51], isolando várias funcionalidades e as suas dependências num módulo que pode ser reutilizado de forma independente para um objetivo comum, como demonstrado na Figura 2.3. Não precisa, pois, de um Sistema Operativo (SO) embutido para executar, sendo um modulo mais pequeno e que ocupa menos recursos de execução do que as VMs.

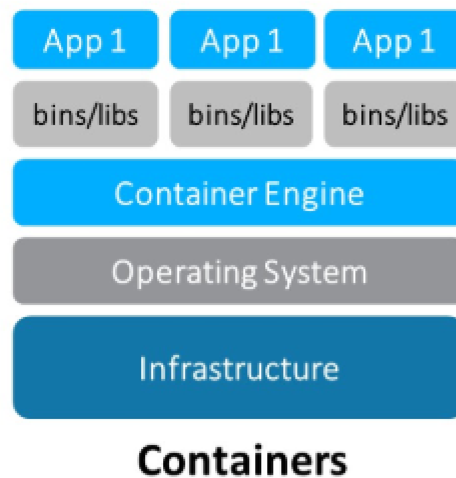

Figura 2.3: Estrutura de um contentor Docker (imagem obtida de [23])

As imagens Docker são os *building blocks* desta solução, que poderão ser atualizados ou alterados se necessário. Neste caso concreto, temos uma imagem que extrai dados e outra que os analisa, mas podemos ter outros módulos que contribuem para uma solução mais elaborada deste cenário. Cada imagem tem um conjunto de parametrizações que são especificadas no formato YAML, na fase de instanciação no ambiente de execução. A linguagem YAML (*Yet Another Markup Language*) é um formato de serialização utilizado em ficheiros de configuração e com o objetivo de ser claro à interpretação humana.

As imagens Docker vão ser essenciais para isolar os serviços que constituem a aplicação. Cada imagem corresponde a um serviço independente, que efetua uma funcionalidade. As imagens comunicam assim entre as suas execuções independentes, para que cumpram os objetivos parciais e, no fim o objetivo principal: a execução com sucesso.

2.4 Ambiente de Execução

Tendo em consideração toda a análise efetuada e detalhada no ponto anterior, considera-se uma boa solução a utilização de contentores. Esta consideração ocorre se tivermos como premissa a necessidade de dar resposta a um dos principais requisitos da arquitetura: a escalabilidade da infraestrutura consoante a necessidade do sistema.

A necessidade de uma grande quantidade de contentores cria uma necessidade adicional na coordenação dos mesmos, sendo que a orquestração dos contentores

é sempre necessária quando são múltiplos [48].

Um ambiente de execução permite aumentar ou diminuir a quantidade de contentores, possibilita a gestão dos recursos disponíveis, permite a definição da quantidade de contentores que são necessários, automatiza novas atualizações em todos os contentores e permite ligar vários contentores entre si, se necessário.

2.4.1 Kubernetes

Um dos mais conhecidos ambientes de execução para orquestração de contentores é o Kubernetes [36], que consiste num sistema que possui todas as funcionalidades essenciais para gerir um sistema escalável, que pretenda utilizar contentores.

O Kubernetes permite uma gestão ao nível do contentor no qual, permite balancear a carga entre contentores, executar a recuperação de contentores em caso de erro e priorizar recursos para operações mais críticas. A gestão do sistema como um todo também é possível neste ambiente de execução, quando é necessário fazer operações sobre todos os contentores. Exemplo disso são alterações a configurações globais, o *rollout* de uma nova versão ou o *rollback*, se algo correu mal. Por sua vez, a escalabilidade horizontal é feita manualmente, através de comandos, ou automaticamente, baseado na utilização do CPU.

2.4.2 Mesos

O Mesos é um gestor de *clusters*, capaz de distribuir recursos por *frameworks* independentes. Assim sendo, além de utilizar contentores, é possível correr ambientes de execução como o Kubernetes. O objetivo do Mesos, com a sua solução Mesosphere [12], consiste numa tentativa de o tornar mais ambicioso que o Kubernetes, pois tenta agregar todos os módulos de uma arquitetura. Os módulos constituintes do Mesosphere são base de dados, ferramentas de processamento de dados e outros ambientes de execução, para além de contentores. Com esta propriedade, é possível utilizar o melhor dos dois ambientes. Por um lado, é viável utilizar o poder de alocação de recursos do Mesos; por outro lado, a orquestração de contentores do Kubernetes torna-se possível.

O Mesosphere agrupa múltiplas tecnologias na sua arquitetura SMACK [13] Stack (Spark, Mesos, Akka, Cassandra e Kafka), fornecendo uma solução completa e facilmente implementável, que oferece uma forma de orquestrar os contentores de

várias tecnologias como as que são propostas na arquitetura SMACK. A Figura 2.4 apresenta a arquitetura proposta.

Figura 2.4: Arquitetura SMACK *stack*, imagem obtida em [40]

2.5 Arquiteturas

Na seguinte secção, serão apresentadas as arquiteturas que inspiraram e tornaram possível a arquitetura proposta. Espera-se reunir o conhecimento suficiente para permitir uma mais fácil compreensão da arquitetura proposta a partir da sua apresentação, pois são a base dos aspetos desenvolvidos no âmbito desta dissertação.

2.5.1 LAMP *stack*

O aparecimento da arquitetura SMACK stack por si só trata-se de uma evolução no paradigma de desenvolvimento de aplicações, dado o consumo elevado de dados das aplicações e informação que as mesmas disponibilizam hoje em dia. Na versão da *Web* 1.0, aquando da utilização de páginas estáticas, foi criada uma arquitetura com o objetivo de criar dinamismo nas aplicações *web*, através de tecnologias *open-source*. Trata-se da arquitetura LAMP (um acrónimo das tecnologias que utiliza - Linux, Apache Server, MySQL e PHP). O aumento do consumo e produção de informação das aplicações, tal como as constantes evoluções tecnológicas, principalmente motivados pela competitividade das organizações, têm impacto no desenvolvimento de aplicações. A arquitetura LAMP stack não consegue corresponder a todas as necessidades das aplicações mais modernas, que requerem uma grande flexibilidade e adaptabilidade. Dessa necessidade,

apareceu a arquitetura SMACK *stack*, que dá a possibilidade de suportar escalabilidade de processamento em tempo real, para aplicações que utilizam dados em massa. Como se pode verificar através da análise da Figura 2.5 a arquitetura LAMP funciona melhor numa aplicação *web* simples, e a SMACK em aplicações direcionadas a dados com escalabilidade em tempo real.

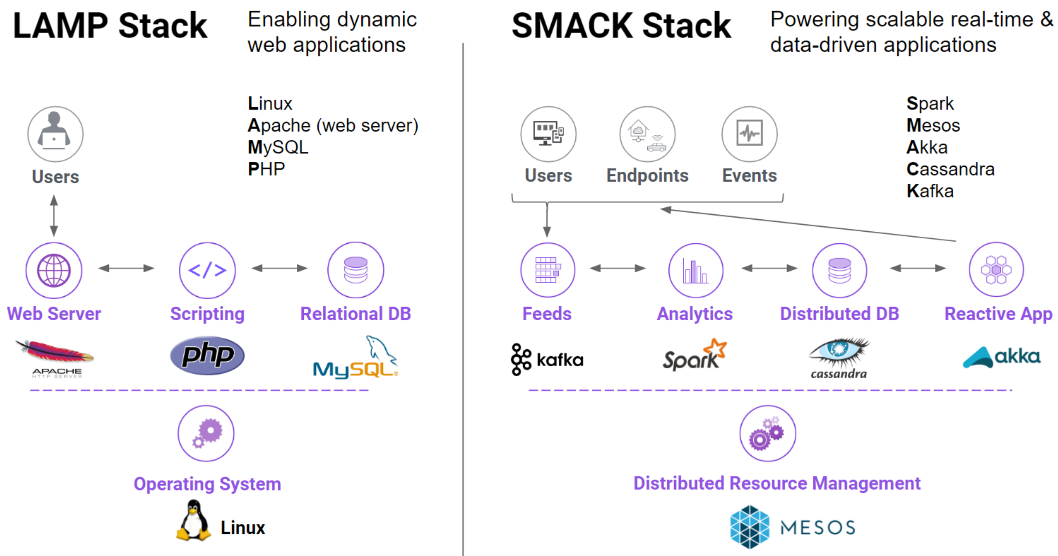

Figura 2.5: Comparação da arquitetura LAMP *stack* e SMACK *stack*, imagem obtida de [22]

2.5.2 Computação em Nuvem

O termo computação em nuvem, *cloud computing*, refere as aplicações disponibilizadas como serviços através da Internet, bem como do *software* e *hardware* nos centros de dados que suportam esses serviços [37]. O *National Institute of Standards and Technology* (NIST) apresenta a seguinte definição: *"Cloud computing is model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage applications and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."*[45].

É uma tendência no mundo das tecnologias da informação, sendo que, na sua essência, trata-se de uma forma de combinar uma grande escala de poder computacional, de forma a obter recursos como armazenamento e processamento, em função da necessidade das aplicações, garantindo a sua disponibilidade e escalabilidade [46].

Segundo a NIST, o modelo *Cloud* é composto por 5 características essenciais que podem resumir-se da seguinte forma:

- ***On-demand self-service.*** Um consumidor deve ter capacidade de aprovisionamento das capacidades da *cloud*, como servidores e armazenamento, de forma automática, sem necessidade de intervenção humana, com o fornecedor do serviço de *cloud*;

- ***Broad network access.*** Os serviços fornecidos pela *cloud* devem promover a heterogeneidade das plataformas clientes (telemóveis, tablets, laptops e desktops);

- ***Resource pooling.*** Os recursos disponibilizados pela *cloud* devem ser alocados ou realocados de forma dinâmica a cada utilizador conforme a necessidade, através um modelo de múltiplos inquilinos. A localização dos recursos deve ser desconhecida pelo utilizador. O utilizador não deverá ter conhecimento nem controlo dos recursos dentro do *data center* que constitui a *cloud*. Apenas deverá ser disponibilizada uma abstração da região geográfica onde se encontra o *data center*, como o continente ou o país;

- ***Rapid elasticity.*** Os recursos da *cloud* poderão ser rapidamente adquiridos ou libertados consoante a necessidade do consumidor. Para o consumidor do serviço, as capacidades deverão estar disponíveis em qualquer quantidade e a qualquer momento;

- ***Measured service.*** Os serviços consumidos pelo cliente devem ser automaticamente medidos por tipo de serviço. Estas medições devem ser monitorizadas, controladas e reportadas de forma transparente, tanto para o fornecedor como para o cliente da *cloud*.

2.5.3 Arquitetura de Micro-serviços

Uma arquitetura de micro-serviços é uma abordagem no desenvolvimento de software que tem vindo a ganhar popularidade devido à sua capacidade de solucionar cenários com grande complexidade. Um micro-serviços corresponde a funcionalidades simples e que comunicam entre si para um objetivo complexo. Estes serviços podem ser monitorizados a partir de um serviço central que acompanha os pedidos e os resultados dos micro-serviços.

Existe enumeras vantagens fase à alternativa, sendo que temos uma arquitetura monolítica, onde a aplicação é constituída por uma grande módulo que centraliza todas as funcionalidades. Encontra-se a primeira vantagem numa necessidade de

qualquer aplicação: manutenção e atualização. Numa aplicação monolítica, uma atualização corresponde a uma instalação nova de toda a aplicação. No entanto, na arquitetura de micro-serviços, é apenas necessário envolver os módulos correspondentes à atualização ou manutenção requerida. Este aspeto é relevante, uma vez que terá uma correlação com a quantidade de intervenções a que a aplicação será submetida, com a estabilidade da aplicação, e, consequentemente, com a experiência de utilização da mesma.

2.6 Sumário

A análise detalhada de cada uma das soluções bem como a análise comparativa efetuada ao longo do documento permitiram definir os módulos constituintes para a arquitetura do SQM. Todas as soluções apresentadas apresentavam-se possíveis de utilização, mas para o cenário apresentado apenas algumas possuem as características que acarretam mais valias para o sistema como um todo. À semelhança da arquitetura SMACK, pretende-se que as tecnologias sejam compatíveis na sua integração e tragam benefícios para o funcionamento e resposta do sistema.

A opção que se considerou dar melhor resposta às necessidades do armazenamento da aplicação foi a base de dados Cassandra. Como visto na secção 2.2.2, trata-se de um produto completo, com pouca necessidade de manutenção, que escala facilmente e que garante uma grande fiabilidade, devido à sua tolerância a falhas.

A base de dados do SQM recebe uma grande quantidade dados de dispositivos de rede, o que se converterá num grande fluxo escritas e leituras na base de dados, sendo que essa informação será armazenada e consumida pela aplicação. A solução monolítica existente não suporta o consumo da grande quantidade de dados que estará disponível. Optou-se, assim, por uma abordagem em micro-serviços [41], isolando os módulos que agregam e analisam os dados em contentores Docker. A utilização de contentores irá permitir paralelizar e distribuir as operações por últimos contentores, permitindo, deste modo, um consumo de informação em tempo útil. A presença de múltiplos contentores cria a necessidade da existência de uma entidade que orquestra, monitoriza e atualiza as instâncias em execução. Com os recursos da *cloud* e utilizando um ambiente de execução Kubernetes para orquestrar os contentores Docker em execução é possível colmatar essa necessidade.

3

Arquitetura Proposta

Um dos conceitos mais relevantes da presente dissertação é o conceito de *Key Performance Indicator* (KPI), pois trata-se do objectivo principal da arquitetura proposta, obter este tipo de indicador através de um grande fluxo de dados não processados, provenientes de redes de larga escala. Um *Key Performance Indicator* é, por definição, um indicador crítico (*key*), que mede o desempenho numa determinada área do negócio ou num determinado aspecto do negócio. Apresenta valores que dão informações essenciais para tomadas de decisão, com relevância estratégica para curto ou longo prazo. Um KPI pode indicar eficiência, eficácia, qualidade, oportunidade, observância, comportamento, economia e utilização de recursos.

A gestão de uma aplicação com o uso de KPIs permite definir objetivos claros ao nível de desempenho desejável. Um KPI com qualidade deve ser claro e objetivo, de forma a auxiliar a tomada decisões e a detetar problemas atempadamente. Um bom indicador oferece valores comparáveis de forma a registar um progresso que pode auxiliar na análise de desempenho da aplicação ao longo do tempo.

A periodicidade de atualização do indicador é um aspeto relevante e que depende de vários fatores. Um exemplo de dois extremos desta situação seria um indicador com o número de vendas que só é relevante no final cada mês e um indicador financeiro cotado na bolsa de valores, recalculado ao segundo.

Podem existir razões para que os indicadores não sejam atualizados em tempo

útil, como, por exemplo, não existirem dados suficientes para que seja processado, ou razões técnicas, em que os dados para os calcular podem ser demasiados, para que um cálculo seja possível em tempo útil, com os recursos existentes. A arquitetura proposta vem resolver este último problema, sendo que utiliza os recursos da *cloud* para processar dados.

3.1 Requisitos funcionais

Na presente tese foi desenvolvida uma prova de conceito que implementa a arquitetura, que utiliza tecnologias atuais para produzir em tempo real KPIs sobre dados, demonstrando, assim, os elementos-chave da arquitetura.

A proposta teve por base os requisitos do enunciado da tese e um manual fornecido pela empresa, que apresenta as características do módulo de análise e os dados que a aplicação SQM atualmente utiliza.

A arquitetura tem como requisito principal apresentar uma solução para a agregação e processamento periódico de dados oriundos de diversos equipamentos de rede. O processamento dos dados dará origem a KPIs que informam o utilizador, para que este possa tomar decisões de forma consciente sobre o sistema. Durante o cálculo de novos valores do KPI deverá ser possível detetar e notificar o utilizador da ocorrência de valores fora do limite definido por KPI.

O processo de geração de novos KPIs estará dividido em duas fases distintas: o *Report* e o *Profile*. Na fase *Report* é feita a obtenção e preparação dos dados para o processamento. A preparação consiste na obtenção de dados de uma janela temporal em blocos, extraindo os valores gerados por dispositivos de rede para futuro processamento. Na fase de *Profile* através dos blocos de dados obtidos na fase de *Report* é feito o processamento e armazenamento dos valores dos KPIs, para uma futura consulta.

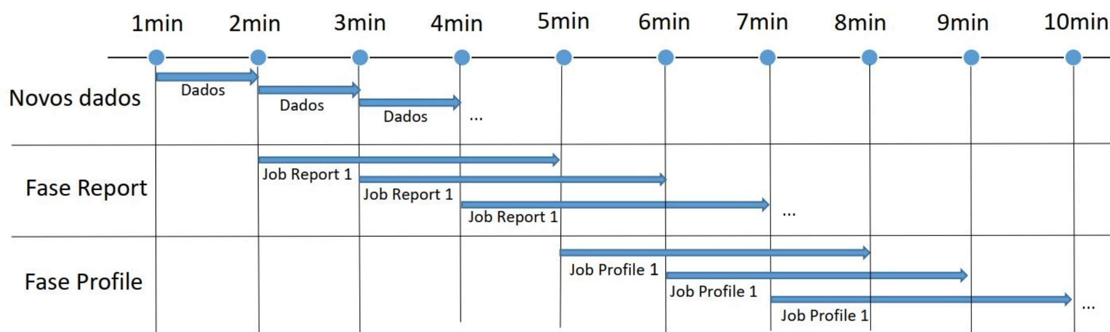

Figura 3.1: Linha temporal do ciclo de geração de KPIs

Na imagem 3.1 é demonstrada a execução no processamento para obter valores de KPIs, durante as fases de *Report* e *Profile*. Na demonstração o carregamento dos dados, a execução do *Report* e o processamento no *Profile* são feitos com uma periodicidade de 1 minuto. Como a execução de cada *Report* e *Profile* demora mais que um minuto, existe uma sobreposição da execução, criando um processamento paralelo do KPI, entre o minuto 3 e 6 no caso do *Report* e entre o minuto 5 e 9 no caso do *Profile*. Também é possível verificar que a execução do *Report* apenas é feita depois dos novos dados estarem disponíveis e a execução do *Profile* é feita quando os *reports* são disponibilizados para processamento.

O utilizador terá à sua disponibilização uma interface para interagir com o sistema através para introduzir as suas parametrizações para a análise dos dados, e irá dispor de *dashboards* para leitura dos KPIs produzidos em tempo de execução. Deverá haver uma área de avisos com as ocorrências de valores fora dos valores limite .

As parametrizações deverão ser específicas por cada tipo de KPI, permitindo o cálculo de múltiplos KPIs em simultâneo. As parametrizações disponíveis devem ser as seguintes:

- A dimensão da amostra de dados não processados através de um janela temporal definida em minutos;
- A periodicidade, em minutos, da criação da amostra;
- A periodicidade, em minutos, do cálculo do KPI através das amostras já obtidas;
- Os valores limite de operação do KPI a partir de um valor máximo e mínimo.

A solução proposta deve ter em conta o futuro do produto quanto à sua elasticidade, desempenho e escalabilidade. Estes requisitos vão de encontro às características presentes na *cloud*, apresentadas na secção 2.5.2. Com a *cloud*, o sistema terá a capacidade de aumentar ou diminuir os recursos e a capacidade de resposta dos pedidos, baseando-se na carga de trabalho a que está submetido.

Por fim, o sistema deverá ser constituído por tecnologias *open-source* (abordado no capítulo 2) e expor as suas funcionalidades, como a sua parametrização, através interface REST . A solução deverá ter uma interface REST e dar acesso a todas as funcionalidades que permitirão interagir com o sistema. Nas funcionalidades de escrita disponíveis, será possível parametrizar o sistema, nomeadamente a introdução de um novo KPI ou alteração de periodicidade e janela temporal de um KPI já existente. O utilizador deverá ter à sua disposição as parametrizações em vigor, obter os KPIs calculados e listar as ocorrências de valores fora do limite definido.

3.2 Arquitetura

A arquitetura proposta abarca a separação de responsabilidades em camadas e define a interação entre os múltiplos módulos que a constituem. As tecnologias envolvidas na proposta da arquitetura foram apresentadas e discutidas no capítulo anterior, Capítulo 2.

A Figura 3.2 apresenta as diferentes camadas e módulos que constituem a arquitetura.

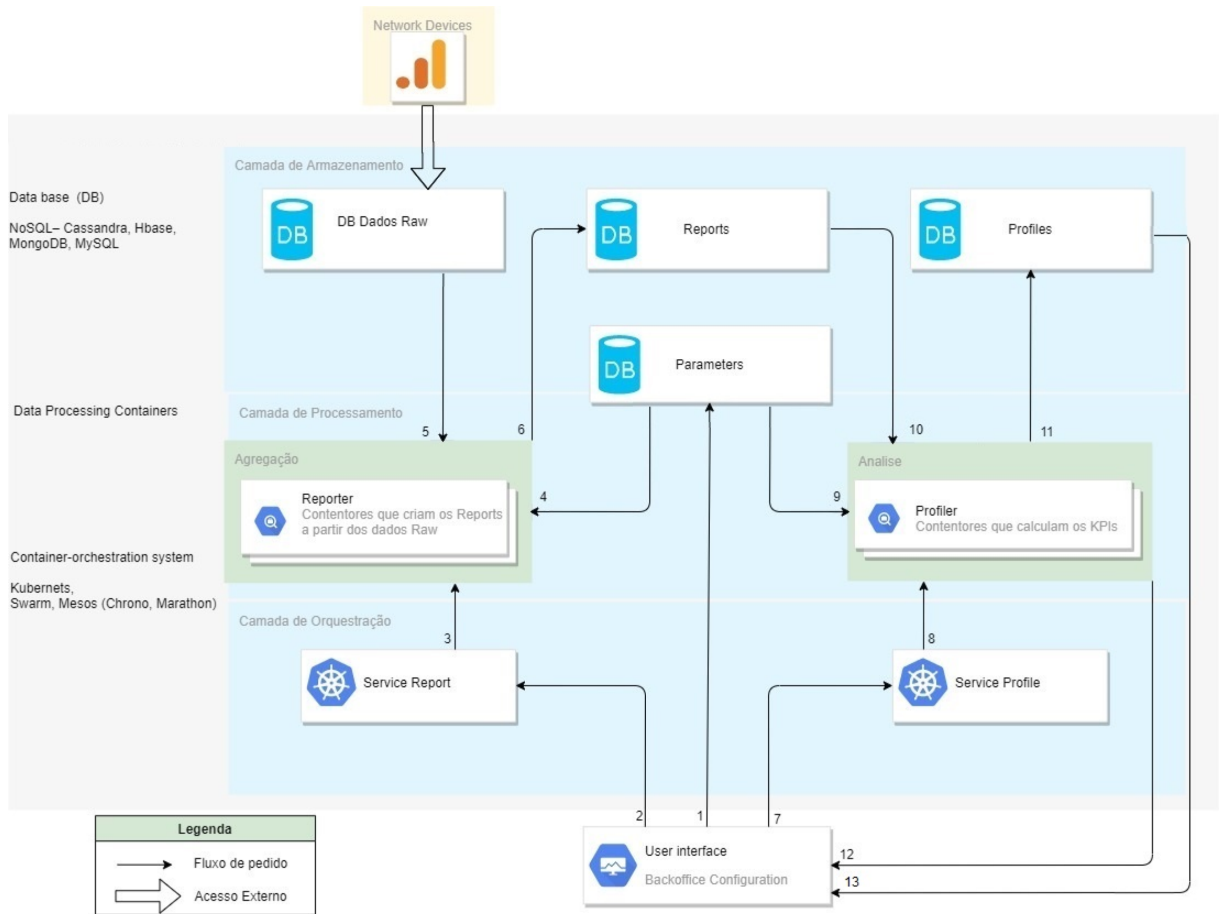

Figura 3.2: Proposta de Arquitetura

Na arquitetura proposta, é possível verificar a semelhança com a arquitetura SMACK, pois existe a mesma definição de responsabilidades. Neste caso, é constituída por uma camada de armazenamento, utilizando o Cassandra e o mySQL, uma camada de processamento que é constituída por uma fase de agregação (*Reporter*) e um tratamento dos dados (Profiler) e, por fim, por uma camada de orquestração que suporta a execução dos componentes anteriores, através da tecnologia Kubernetes.

Na imagem 3.2 é demonstrada a posposta da arquitetura e a interação entre os múltiplos módulos. A primeira interação com o sistema é a configuração da parametrizações do KPI através de um formulário na *User Interface* (UI) que será utilizado durante a agregação dos dados e o cálculo do KPI (1). A fase de agregação dos dados inicia-se no pedido de criação através do serviço *Report* (2) que é utilizado para lançar instâncias de *Report* (3) que utilizam as parametrizações configuradas para um KPI (4). Cada instância de *Reporter* obtém os dados *Raw*

(5) e disponibiliza-os no formato *Report* (6). O cálculo dos KPIs inicializado com o pedido de processamento através do serviço *Profile* (7). O *Profile* cria instâncias (8) que utiliza as parametrizações configuradas (9) para obter *reports* (10) de forma a calcular e armazenar o *Profile* do KPI (11). Depois do cálculo de cada KPI se o valor calculado for atípico segundo os valores calculados, será enviado para a UI um aviso da sua ocorrência (12). Os últimos indicadores calculados são mostrados na UI (13).

3.2.1 Camada de armazenamento

A camada de armazenamento, além de ter a responsabilidade de armazenar os dados que serão processados, deve também guardar os dados dos *Reports*, que serão consumidos na camada de processamento, disponibilizar os mesmos para que sejam analisados na interface gráfica e guardar as parametrizações definidas pelo utilizador.

A primeira base de dados funcionará como uma memória persistente, que acumulará os dados e funcionará no *background* do sistema, fornecendo e guardando dados da camada de processamento. A base de dados que mais se adequa a esta necessidade é a base de dados Cassandra. O armazenamento de dados na presente arquitetura requer, tal como foi apresentado no capítulo anterior, uma base de dados que permita uma capacidade flexível no número de acessos, que tenha mecanismos que garantam a disponibilidade e garanta robustez tanto nas escritas como nas leituras.

Nesta camada também estará presente uma segunda base de dados que terá as configurações e os dados que são apresentados na interface gráfica. Esta base de dados funcionará como uma *cache*, guardando os últimos dados processados com a informação dos KPI. Também é nesta base de dados que vão ser armazenados os parâmetros introduzidos pelo utilizador para interagir com o sistema em tempo de execução. Devido à quantidade de dados utilizada, ao número de atualizações e de acesso a que esta base de dados está sujeita, não existe restrição ao tipo de base de dados a utilizar.

3.2.2 Camada de processamento

Na camada de processamento de dados, é feita a agregação dos dados provenientes de dispositivos de rede e a transformação e disponibilização dos dados

em KPIs úteis para os operadores do sistema. Todos esses dados são provenientes e armazenados na base de dados Cassandra da camada anterior 3.2.1. O tratamento feito nesta camada é específico para os requisitos desta aplicação e reaproveita grande parte dos algoritmos já utilizados na solução SQM.

O processamento de cada indicador é feito de forma especifica. Cada indicador tem configurações, parametrizações, *jobs* específicos e diferentes formas de calcular cada tipo de indicador. O utilizador poderá, através de um *frontend*, alterar a quantidade de dados que irá ser processada; a periodicidade em que é feita a extração/processamento e existe processamento e parametrizações distintas por cada KPI. Este cenário só é possível devido à transformação em imagens Docker dos módulos de agregação (*Reporter*) e análise (*Profiler*). Cada processamento traduz-se, na prática, na execução de uma imagem Docker, contextualizada no indicador pedido. A partir dessa contextualização, é possível obter da camada de armazenamento todos os dados e configurações para esse mesmo indicador.

3.2.2.1 Serviço de *Report*

O serviço de *Report* tem as seguintes funções: a criação e a interação dos *workers* de *reporting* associado a um KPI. O serviço *Report* será criado no Kubernetes e irá lançar os *workers* na periodicidade configurada pelo utilizador. A periodicidade pode ser reconfigurada através do serviço, fazendo *refresh* do valor da periodicidade em tempo de execução, com o valor presente na base de dados mySQL.

Um dos valores relevantes para um *Report* de um KPI é a janela temporal em minutos de dados não, processados que é utilizado para a criação de um *Report*. Este valor, tal como as parametrizações anteriores, é um valor específico e configurável por KPI .

3.2.2.2 Serviço de *Profile*

Este é um serviço semelhante ao de *Report*, mas direccionado para fase de *Profile*. O serviço tem como objetivo criar um *worker* para um processar um *report* disponível com as parametrizações definidas pelo utilizador, para um KPI específico. As parametrizações podem ser alteradas diretamente a partir das funções disponibilizadas pelo serviço. As parametrizações que estão disponíveis para alteração, são as seguintes: periodicidade da execução do *Report* de um KPI, valor máximo/mínimo permitido por KPI.

3.3 Ambiente de execução

A escalabilidade da arquitetura é um aspecto importante e por isso deve existir uma infraestrutura que suporta essa escalabilidade consoante a necessidade do sistema. A solução proposta é a utilização de uma implementação de contentores, que utiliza um ambiente de execução onde é possível aumentar ou diminuir a quantidade de instâncias e recursos *on demand*. Para que os múltiplos contentores persistentes da camadas de armazenamento e os *workers* que processam a informação interajam entre si, é necessária uma camada de orquestração.

O Kubernetes oferece as ferramentas necessárias para a coordenação dos múltiplos módulos e da criação/monitorização de instâncias de execução de imagens Docker, através de uma API específica.

O Ambiente de execução também tem um papel fundamental na robustez da solução, devido ao seu mecanismo de *auto-healing*. Se existir um processamento com problemas ou que terminou abruptamente, é possível recuperar o mesmo, relançando-o automaticamente.

A solução apresentada também tem como objetivo ter camadas independentes e isoladas das outras. A ideia principal é ter uma arquitetura dinâmica , podendo substituir-se numa das camadas por outra tecnologia semelhante, mas que seja uma mais valia para outro cenário com a garantia que as funcionalidades são as mesmas e cuja interface é compatível. Isto é possível visto que cada componente tem uma responsabilidade isolada. O ambiente de execução serve, assim, como base que dá suporte às camadas anteriores, garantido a execução de todos os componentes do sistema.

3.3.1 Cenário de execução

Um cenário de execução poderia ser o que se explica de seguida (imagem 3.3). O operador faz um pedido, através de uma interface gráfica, para obter os KPIs de uma semana com uma periodicidade diária (1). A interface, por sua vez, utiliza o serviço *Reporter*, exposto no Kubernetes através de uma API REST (2). O serviço lançará as instâncias de *workers* no Kubernetes, utilizando uma API específica, para interagir diretamente com o ambiente de execução (3). Os *workers* de *reporting* irão realizar extrações todos os dias é gerado um com os últimos 7 dias, criando e armazenando as amostras (*reports*)(4). Quando um *Report* tiver disponível, o serviço *Profile* lançara *workers* para calcular o indicador a partir da

amostra presente do *Report* (5) e disponibilizará uma nova leitura do indicador diariamente através do *dashboard* (6). Será disponibilizada a leitura ao utilizador (7), juntamente a uma notificação de aviso se o valor do KPI calculado, passar algum *threshold* definido.

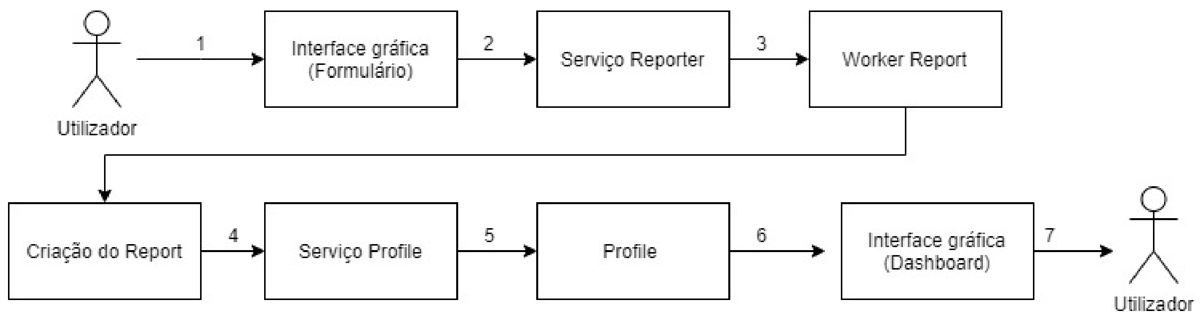

Figura 3.3: Diagrama de sequência de um pedido do utilizador para obter um KPI

3.4 Interface gráfica

A *user interface* tem como objetivo permitir ao utilizador interagir com o sistema. Na interface o utilizador tem acesso a *dashboards*, onde são apresentados diferentes os valores dos KPIs. Tem também acesso a notificações de aviso quando os *boundary values* forem ultrapassados. Na interface são apresentados os valores dos parâmetros que o sistema está a utilizar. Alem dos *dashboards*, o utilizador terá formulários para interagir com o sistema. Nos formulários, depois de ser contextualizar um KPI é possível iniciar o processamento e alterar os parâmetros em vigor.

3.5 Resumo

A arquitetura apresentada é baseada na arquitetura SMACK, diferindo nas tecnologias, mas semelhante quanto à sua abordagem de definição das responsabilidades das camadas para uma arquitetura modular. As responsabilidades podem-se dividir nas seguintes camadas: armazenamento, processamento e execução. A camada de armazenamento tem responsabilidade disponibilizar e guardar os dados das diferentes fases durante o cálculo dos KPI's. O processamento dos dados

onde existe a agregação dos dados e cálculo dos KPIs é feito na camada de processamento. Por fim a camada de execução realiza a orquestração das camadas anteriores para comunicarem entre si de forma a responderem aos pedidos do utilizador.

4

Implementação

Para demonstrar uma possível abordagem ao problema com a utilização da arquitetura proposta, foi desenvolvida uma prova de conceito. Esta terá os mesmos requisitos funcionais abordados na secção 3.1. Será possível, posteriormente, tirar conclusões de forma a perceber a viabilidade da solução proposta.

4.1 Implementação de serviços

Os primeiros passos da implementação da prova de conceito foram efetuados na estruturação dos serviços, que expõem as funcionalidades do sistema através de uma *inteface* REST. Estes serviços têm como principal responsabilidade criar um *Report* (*Reporter*) e calcular indicadores a partir dos *reports* criados (*Profiler*).

Desenvolvidos na *framework* Spring, os serviços fornecem as funcionalidades a que o utilizador recorre para interagir com o ambiente de execução. O utilizador tem disponível a possibilidade de criar um novo processamento e alterar configurações de um processamento já existente (como a periodicidade de um indicador).

As entidades que se pretende paralelizar e executar em grande escala são as de agregação e processamento dos dados. Por essa razão, cada pedido será executado através de múltiplos *Pods*, que utilizam os recursos do ambiente de execução. O serviço e o processamento do pedido são disponibilizados no ambiente de execução através de imagens Docker. Para isso, é necessário criar as imagens e colocar as mesmas acessíveis ao ambiente de execução.

Após a criação das imagens do serviço, é necessário expor a funcionalidades do serviço no ambiente de execução, através do objeto *Service*. O Serviço irá expor as suas funções para tráfego interno, através de um identificador único atribuído no Kubernets, e irá expor as suas funções ao tráfego externo, através de um endereço IP.

Depois de terem sido feitas as configurações anteriores, os serviços ficam disponíveis para tráfego interno e externo, para que sejam usados pelos outros *Pods* e pelo *frontend*, respetivamente.

4.1.1 Criar imagens Docker

A criação de imagens Docker corresponde ao processo de conversão de um software em contentores Docker, de forma a ser executado de forma independente, utilizando todas as dependências necessárias. As imagens Docker nesta demonstração foram criadas a partir de um ficheiro de *build* para esse efeito, chamado *Dockerfile* [6].

O Dockerfile contém os comandos que o Docker tem para executar para criar a imagem na sequência do comando *docker build* [4]. No exemplo 4.1.1, foi usado para transformar o .jar que realiza o *Profile* dos dados numa imagem Docker:

```
FROM java:8
ARG JAR_FILE
COPY ./profiler.jar /profiler.jar
ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","profiler.jar"]
```

Deste exemplo, destaca-se a instrução *FROM*, que indica que a nova imagem Docker gerada se baseia uma imagem Docker já existente. Neste caso, a imagem usada contém o java 8, que é uma dependência necessária para executar o *Profiler*. Salienta-se também a instrução *ENTRYPOINT*, que indica as instruções que o Docker executa no arranque da imagem. Para criar a imagem é utilizada a seguinte instrução:

```
Docker build . -t tese/Profiler:latest
```

Para que a imagem gerada fique acessível ao Kubernetes, a mesma é colocada num repositório para o efeito, seja este público ou privado[8]. O repositório usado neste protótipo é o Docker Hub [7] na sua versão pública. A versão privada do repositório tem custos e deve ser usada em contextos de organizações ou equipas fechadas [20].

Para realizar o *push* da imagem para o Docker Hub, depois de autenticado, é necessário utilizar o comando Docker, indicando o nome do repositório, o nome da imagem e a sua versão. No seguinte exemplo 4.1.1, a instrução usada para colocar no repositório a imagem *Profiler*:

```
sudo Docker push tese/Profiler:latest
```

4.1.2 Utilização da biblioteca

De forma a executar um pedido de execução no Kubernetes, é necessário utilizar a entidade *Job* [11]. Um *Job* trata-se de uma execução supervisionada de um ou mais *Pod*, que executam um processamento no Kubernetes até à sua conclusão. Um *Pod* [18], por sua vez, é a unidade mais básica na definição da arquitetura e os *workers* do Kubernetes [25]. Cada um corresponde a uma abstração que representa um ou vários contentores Docker que trabalham em conjunto para atingirem um objetivo. Quando esse objetivo é atingido, o *Pod* desaparece. Um *Pod* que falhe a sua execução pode ser recuperado e novamente executado de forma transparente ao seu solicitador. O objeto Kubernetes que permite este comportamento denomina-se *ReplicaSet*[19] e garante que os *Pods* sejam automaticamente repostos, se necessário. Assim sendo, com a utilização de *Jobs*, é possível utilizar as potencialidades de escalabilidade com robustez e lançar múltiplas instâncias de imagens Docker do *Report* e do *Profile*.

Para utilizar esta solução, deve ser definida uma estratégia para executar *jobs* de imagens Docker no Kubernetes, através de pedidos ao serviço permanente. Para o serviço ter a capacidade de lançar *jobs* de imagens Docker que executam no Kubernetes, é necessário utilizar uma biblioteca específica para o efeito.

Estão disponíveis pela equipa de Kubernetes as bibliotecas clientes em Java para o Kubernetes [33].

Para esta demonstração foi usada uma extensão da biblioteca oficial: Fabric8io Java client for Kubernetes [9]. Esta extensão do cliente java para Kubernetes tem mais funcionalidades que a versão oficial, incluindo as funcionalidades essenciais para a solução (como a criação de *jobs* e parameterizações dos mesmo) e possui uma melhor documentação, com exemplos práticos que suportam a biblioteca. Para além disso, a biblioteca cliente fornecida pela Fabric8io permite a utilização de uma linguagem fluente, com uma estrutura semelhante aos ficheiros de configuração (YAML) dos serviços, como demonstra a Figura 4.1, e, por isso, é muito fácil de utilizar, porque se baseia em regras já conhecidas.

Figura 4.1: Conversão de YAML para java usando fabric8io para criar um serviço

4.1.3 Exposição de serviços no Kubernetes

Após a criação das imagens dos serviços, será necessário colocá-las no ambiente de execução. O *kubectl* [17] é o comando usado para executar comandos sobre o Kubernetes e, neste caso, é usado para criar os objetos no Kubernetes a partir de um ficheiro YAML [26].

O YAML tem todas as parametrizações necessárias para criar um *objeto service* [21], desde *metadata*, que identifica o serviço através do nome e o *namespace*, até configurações de rede no *cluster*, como *ports* e o protocolo de rede utilizado. No exemplo seguinte, é mostrada a criação do serviço:

```
kubectl create -f ProfilerService.yml
```

Em Kubernetes, os objetos *service* são abstrações responsáveis pela definição e distribuição balanceada dos pedidos por múltiplos *Pods*. Também é no serviço que se define a política de acesso, para que os *Pods* possam ser acedidos por tráfego interno e externo. Estas funcionalidades são fornecidas pela entidade `kubeproxy` [34]. A exposição do serviço é feita através de um IP, de um porto e de um nome. A descoberta destes serviços é feita ou através da configuração de variáveis de ambiente na criação do serviço, ou através de *plugins* que funcionam como servidores DNS, associando o nome do serviço ao endereço IP [3, 5]. Quando os serviços estão disponíveis no Kubernetes, têm a possibilidade de ser replicados e

geridos através de um balanceador de carga, para que este esteja sempre disponível.

Os processamentos que os serviços permanentes fazem podem ser divididos em dois passos distintos: *reporting* e *profiling*. Para realizar estas operações, o serviço cria, no ambiente de execução, *Jobs* que executam as imagens de *Report* ou *Profile*. É da responsabilidade do serviço acompanhar os *Jobs* lançados e de dar informação da execução dos mesmo quando for solicitada. Devido à sua natureza, o ciclo de vida de um *Job* termina com a sua execução; portanto, é necessário um serviço permanente que faça a gestão do pedido. No caso de se tratar de uma execução de um *Report*, o serviço informa o *Profile* de que a extração foi feita e que a mesma pode fazer a análise dos dados. No caso da execução *Profile*, esta informa o utilizador que os KPI se encontram disponíveis e testa indicadores de forma a detetar valores fora do normal. Os valores esperados são *thresholds* definidos pelo operador na IU, que são guardados na base de dados. Se algum dos valores dos KPI sai do intervalo esperado, o operador é notificado. A notificação será feita através de *dashboard* na interface de utilizador, mas com a possibilidade no futuro de notificar o utilizador através de E-Mail, *short message service* (SMS).

A atualização dos contentores *Profile*, *Report* e dos serviços permanentes é gerida automaticamente pelo Kubernetes. Depois de ser feita a atualização da imagem do *worker*, é feita o *rollout* da nova versão para os *Pods* em execução, um a um, até que a versão da imagem anterior deixe de ser utilizada no ambiente. Esta atualização é feita sem *downtime* e sem os utilizadores se aperceberem de que estão a usar a nova versão.

4.2 Objectos chave no ambiente de execução

4.2.1 *Deployments*

Pods são abstrações que representam um grupo ou mais contentores que executam em conjunto. No contexto desta implementação, tratam-se dos *workers* que executam as imagens Docker. Durante a execução dos *Pods*, estes estão sujeitos a erros e falhas. Para isso, existem *ReplicaSet*, que mantêm um número específico de réplicas que repõem automaticamente *Pods* em erro. Os *Pods* de *ReplicaSet* permanecem estáticos durante o seu ciclo de vida.

O *Deployment* tem sob a sua tutela grupos de *Pods* e *ReplicaSet*, permitindo, assim, fazer alterações e atualizações transversais a todas as entidade que gere. Desta

forma, se for efetuada uma atualização ao sistema, é possível fazer o *rollout* da mesma, sem existir indisponibilidade.

A implementação da solução abordada nesta dissertação tem dois *Deployments*: um dedicado aos *Pods* de agregação de dados (*reports*) e outro ao *Pods* de processamento dos dados (*profiles*).

4.2.2 *Services*

Os *Services* constituem a camada de abstração que define um conjunto de *Pods* que fornecem um serviço. Esta camada mantém *Pods* disponíveis para atender pedidos através de tráfego externo ou interno ao *cluster*. Os pedidos são balanceados pelas réplicas disponíveis.

Os *Pods* são identificados e agregados através de um identificador chamado Label e associados a um endereço IP. Dessa forma, é possível que outros *Pods*, durante a sua execução, consigam encontrar e comunicar com os *Pods* do tipo *service*.

Os componentes da arquitetura que são executados como serviços são: *Report* e *Profile*.

4.2.3 *CronJob*

O *CronJob* é um objeto Kubernetes que cria *Jobs* periodicamente, em horários ou intervalos específicos. Durante a implementação surgiu a questão: será a utilização do *CronJob* a mais indicada?

No contexto do presente cenário as extrações e os KPIs calculados, são feitos numa dada periodicidade. Sendo uma funcionalidade requerida do sistema, caso a opção fosse a utilização *Job*, seria necessário implementar nos serviços um mecanismo de temporização que iria despoletar e mapear o processamento de *reporting* e *profiling* para cada KPI. Em alternativa, o *CronJob* tem como funcionalidade base de executar um *Job* com dada periodicidade não sendo necessário implementação adicional.

A decisão final de utilizar o *CronJob* deveu-se ao facto de se adaptar perfeitamente às necessidades deste cenário. Apesar de a utilização do Job se tratar de uma solução mais genérica, dando alguma flexibilidade, o *CronJob* confere a possibilidade de utilizar um objecto base no Kubernetes para corresponder a um requisito sem ser necessário implementar lógica adicional.

4.2.3.1 *CronJob Raw*

Para que o sistema funcione, é necessário que dados sejam disponibilizados, dados não processados, e que se simule os outputs de dispositivos de rede. O seguinte *CronJob* irá criar um *Job* que adiciona valores à tabela *Raw* para que estes sejam agregados pelo *Job Report*. Desta forma, é o *Job Raw* que "alimenta" o sistema com dados a serem agregados e processados, de forma a obter o resultado em forma de KPI. O *Cronjob Raw* é apenas utilizado no prototipo para simular a geração de dados de dispositivos de rede, não se trata num aspeto relevante de implementação.

4.2.3.2 *CronJob Reporter*

O *CronJob Reporter* lançará o *worker* que fará uma leitura da base de dados *Raw*, numa janela de tempo configurada pelo utilizador. Dos dados recolhidos, são extraídos os dados constituintes de um Report, criado e associado a um *reportID*. Por fim, o *reportID* é colocado na fila de trabalho do *Profile*.

Um exemplo prático do funcionamento deste *CronJob* seria o que se explica de seguida. Assim, ter-se-ia um *CronJob ReportBitRate1* que se trata, neste exemplo, de um *Job* de *reports* referente ao *Bitrate*, obtendo a configuração definida do utilizador. O *Job* é lançado de hora em hora, obtendo os valores do *Bitrate* nos últimos 30 minutos. Esses dados são adicionados à tabela *Report*, com um identificador único que identifica o *Report* e também o coloca na fila, para o mesmo ser processado pelo *Job* de *Profile*.

4.2.3.3 *CronJob Profile*

O *CronJob Profile* que cria o *worker* que vai buscar o primeiro *Report* de um KPI que está na fila de trabalho. Depois, calcula o KPI e coloca-o na base de dados Cassandra. Para efeitos de demonstração, o cálculo que está a ser feito no *worker* é uma média, podendo ser implementadas novas formas de cálculo por KPI. Posteriormente, avalia o valor calculado, verificando se está dentro de um valor válido. Em caso negativo, adiciona-o à tabela de notificações, informando o utilizador da ocorrência.

4.2.4 *StatefulSet*

O *StatefulSet* é um objeto que permite controlar a ordem de *deployment* de um grupo de *Pods*, atribuindo identificadores únicos a esses *Pods*. Tal como o *Deployment*, o *StatefulSet* gere *Pods* idênticos, mas mantém uma ligação aos mesmos através de identificadores persistentes que são substituídos rapidamente em caso de erro.

O objeto é usado para manter estado nas aplicações dentro do ambiente de execução. Devido a estas características, este objeto vai ser usado para suportar os *Pods* com as imagens de armazenamento de dados.

4.2.4.1 *StatefulSet* Cassandra

É o StatefulSet onde está a ser executado um cliente Cassandra. Na base de dados Cassandra, são armazenados os dados mais volumosos da aplicação. Trata-se dos dados de dispositivos de rede e os dados estruturados, como *reports* e *profiles*.

Tabela Raw A Tabela *Raw* contém dados que simulam as leituras de um dispositivo de rede. A tabela é preenchida pelo *CronJob Raw*, sendo que cada linha representa uma leitura de um dispositivo de rede. Os dados inseridos nesta tabela alimentam o sistema, pois, a partir deles, são criados *reports* e, consequentemente, *profiles*. Os campos de cada leitura são:

- ID - Identificador único da leitura de um dispositivo de rede;
- DeviceName - Nome do dispositivo que fez a leitura;
- DeviceType - Tipo de dispositivo que fez a leitura. Os dispositivos poderão ser *routers*, *switches* ou qualquer outra entidade que pertença a rede que envie dados para serem processados;
- Timestamp - data e hora em que foi feita a leitura;
- Bitrate / ImpactingEvents / TCPLoss

  Conjunto de valores inteiros que serão usados para calcular o KPI. Para adicionar mais indicadores, é necessário adicionar novas colunas e outro tipo de valores. Os valores utilizados neste exemplo são:

  - *Bit rate*: que significa taxa ou fluxo de bits, ou fluxo de transferência de bits;

- *Impacting Event*, também conhecido por *Performance Impacting Event* (PIE): número de problemas de desempenho que existem num nó, em dois nós ou na comunicação entre dois nós;

- *TCP Loss*: média de pacotes (por cada 1000) que foram enviados e nunca recebidos no destino.

Tabela Report Esta é a tabela onde são agrupadas as extrações dos dados *Raw*. Cada conjunto de leituras constituem um *Report*. Os campos de cada *Report* são os que se elenca de seguida:

- ID - Identificador único de um *Report*;
- DeviceName - Nome do dispositivo que gerou esta leitura no Report;
- DeviceType - Tipo de dispositivo que gerou esta leitura no Report;
- KpiReport - Corresponde KPI que está a ser tratado no Report. Por exemplo: *Bit rate*, *Impacting Events* e *TCP Loss*;
- Value - Valor lido associado ao KPI;
- Timestamp.

Tabela WorkQueue Tabela na qual são disponibilizados os *Report* para serem alvo de cálculo. Cada linha é um *report* pronto a ser transformado num *profile*. Os campos da tabela são os seguintes:

- Reportid - Identificador de *report* a ser tratado pelo *profile*;
- KpiReport - KPI a ser tratado no contexto do *report*.

Tabela Profile Tabela que representa os KPIs calculados através dos *reports* disponibilizados. Cada linha é um *profile* de KPI calculado. Os campos de um *profile* são os apresentados de seguida:

- ID - identificador único do *profile*;
- Reportid - Identificador único do *report* que deu origem ao *profile*;
- KPI - Nome do KPI associado ao *profile*;
- Value - Valor calculado do KPI;
- Timestamp.

4.2.4.2 *StatefulSet* MySql

As configurações dos utilizadores são armazenadas numa base de dados MySQL, acedida ao longo da execução do pedido, para que este tenha em conta as configurações de um dado KPI. As configurações disponíveis para o utilizador, tal com a periodicidade por KPI, podem ser alteradas a qualquer momento, através do *frontend*. Também nesta base de dados são disponibilizadas as mensagens que informam os utilizadores se um indicador está fora do intervalo considerado válido pelo utilizador.

Tabela Parâmetros Tabela que contém os parâmetros de um dado KPI. Cada linha representa uma configuração de um dado indicador. Desta forma, por cada coluna de valores a processar da tabela *Raw* deve existir uma linha de configuração na tabela de parâmetros, com as parametrizações específicas para esse KPI. Cada configuração tem os seguintes campos:

- KPI - Nome do KPI que vai ser parametrizado;
- TimeReport - Tempo em minutos para realizar novos *reports*;
- TimewindowReport - Tempo em minutos que define a janela temporal dos valores raw a serem obtidos;
- TimeProfile - Tempo em minutos para criar *profiles*;
- ValMax - Valor máximo que o KPI pode ter;
- ValMin - Valor mínimo que o KPI pode ter;
- ReportSleep - Tempo de execução do *Report* em minutos (apenas utilizado para testes);
- ProfileSleep - Tempo de execução do *Profile* em minutos (apenas utilizado para testes).

Tabela Notificação Tabela que contém as notificações para os utilizadores quando os valores calculados são superiores ou inferiores aos valores parametrizados. Cada notificação tem os seguintes campos:

- Error - Mensagem de notificação;
- TimestampError - Tempo em que a mensagem de notificação foi gerada.

4.2.5 Interface gráfica

Para interagir com o sistema, o utilizador final terá acesso a uma interface gráfica. A interface será uma abstração para a chamadas às funções disponibilizadas nos serviços implementados. Optou-se usar o Grafana [31] para apresentar os formulários de parametrização e os gráficos de consulta. Assim sendo, o utilizador terá acesso a formulários de parametrização do processo de *Report* e *Profile*.

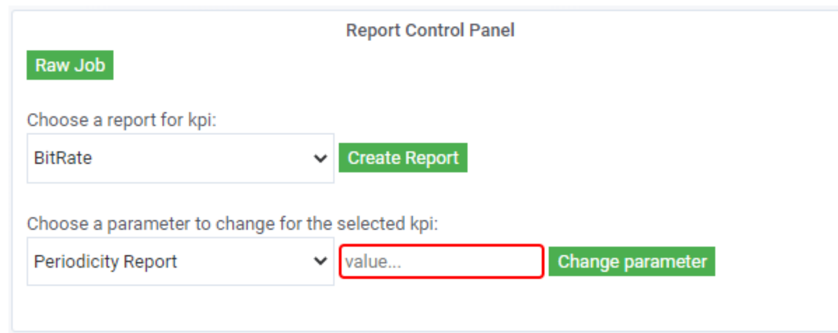

Figura 4.2: Formulário de parametrização do *Report*

O formulário apresentado na imagem 4.2 trata-se do formulário reservado ao *Report*. Neste formulário é possível:

- Iniciar o *Job Raw* que alimentará a BD de dados por processar;
- Iniciar o processo de *reporting* de um KPI;
- Alterar a parametrização de um Report: Periodicidade e janela temporal.

Semelhante ao formulário anterior, o seguinte formulário 4.3 está reservado à parametrização do *Profiler*.

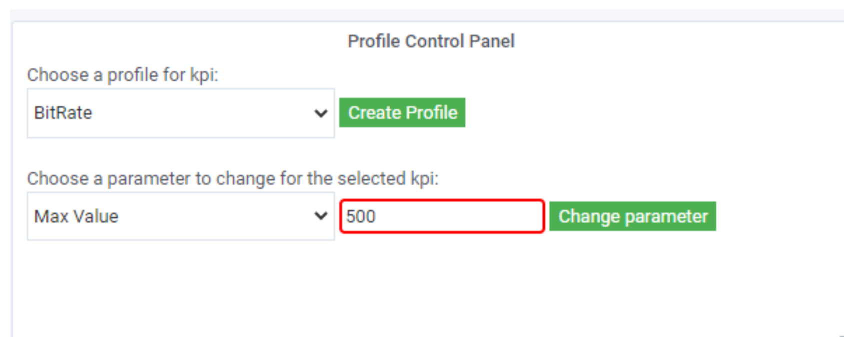

Figura 4.3: Formulário de parametrização do *Profile*

Neste formulário será possível:

- Iniciar o *profiling* de um KPI;
- Alterar a parametrização de um Report: Periodicidade, valor máximo e valor mínimo por KPI.

Para que o utilizador tenha acesso aos dados dos KPIs, parametrizações em vigor e notificações de ocorrências de valores fora dos limites definidos, acede à área de *dashboards*.

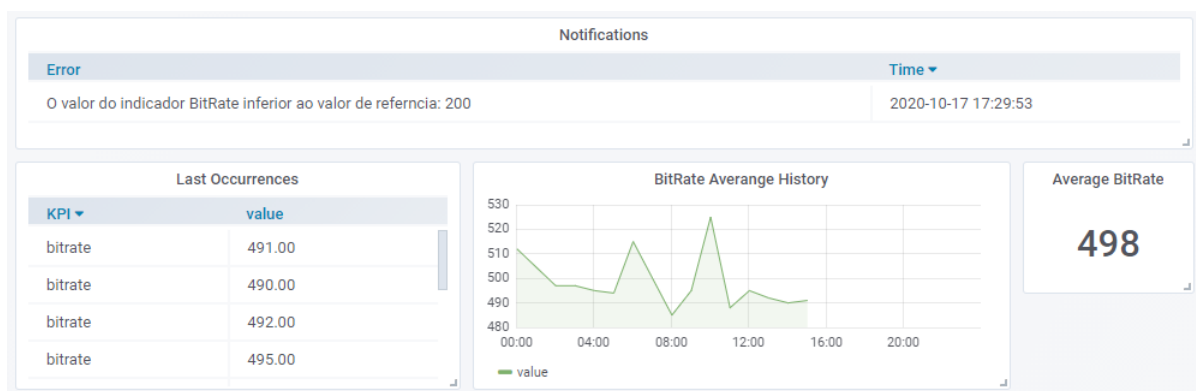

Figura 4.4: Dashboards para o KPI BitRate

Na imagem 4.4 é demonstrada uma possibilidade de dispor a informação no *dashboard* de um KPI. Neste exemplo é possível verificar as seguintes funcionalidades para o KPI BitRate:

- Lista de parâmetros em vigor do KPI;
- Últimos valores de KPI gerados;
- Média dos valores gerados;
- Gráfico de histórico das atualizações do KPI;
- Lista de ocorrência de valores fora do limite definido.

4.3 Interação dos módulos

Para o funcionamento da presente arquitetura, é necessária a comunicação entre os diferentes módulos, para que o seu objetivo individual contribua para um objetivo global. Os módulos que se encontram dentro do ambiente de execução comunicam diretamente com a infraestrutura Kubernetes. Os serviços, por sua vez, como comunicam através de uma interface REST, recebem os pedidos internos e vindos do exterior, como a interface gráfica.

O facto de existir uma divisão dos módulos por responsabilidade permite isolar rapidamente um problema e aplicar uma solução ao módulo com o erro, sem ser necessário causar indisponibilidade ou fazer um novo *deploy* dos outros módulos que constituem o sistema. Outra vantagem na manutenção e evolução dos módulos isolados é o facto de que torna possível abstrair do funcionamento de todo o sistema e focar nas funcionalidades do módulo em que foi feita a intervenção, permitindo, assim, que a manutenção tenha um menor risco, conhecimento necessário e impacto. Um exemplo das vantagens apresentadas é a atualização de um cálculo do KPI, em que seria apenas necessário atualizar a imagem de *Profile* com o novo cálculo. Neste exemplo, o código ficaria disponível sem indisponibilidade da módulo, tornando o sistema mais robusto e com fácil manutenção.

4.3.1 Adição de novos módulos

Ao longo do tempo, irão surgir novos desafios e novas funcionalidades que resultam da experiência do utilizador e das suas necessidades. Efetivamente, novas implementações que interagem diretamente com o sistema poderão ser adicionadas através de atualizações às imagens Docker existentes. As imagens Docker atualizadas, graças aos mecanismos do ambiente de execução, são disponibilizadas nos novos pedidos.

A adição de novos módulos à solução é possível, mas devem ser tidos em consideração os módulos existentes e as suas interfaces. Será também necessário alterar as interfaces existentes para que um novo modulo interaja com os módulos e o fluxo de tratamento de dados que existe na solução. Um exemplo de uma ideia para um novo modulo seria, um modulo de pré-processamento dos dados *Raw*. Na prova de conceito apresentada, os dados de testes são gerados e, por isso, estão num formato que o sistema consegue facilmente interpretar. Num cenário real, nem sempre é possível que os dados estejam completos e bem

formatados. Uma possibilidade de um novo módulo, na arquitetura proposta, seria o pré-processamento onde os dados pudessem ser transformados de forma a estarem prontos a alimentar o sistema. Este novo módulo poderia ser introduzido no sistema apenas com pequenas alterações à imagem de *Report* que faz a extração dos dados, para obter os dados transformados pelo novo módulo e não diretamente dos dados *Raw*.

4.3.2 Novos KPIs

Os indicadores presentes nesta prova de conceito são apenas exemplos. Num cenário real, existirão múltiplos indicadores relevantes que serão diferentes dos exemplos apresentados nesta solução. Para isso, o desenvolvimento da solução tem em conta um mecanismo dinâmico, de modo a adicionar novos indicadores ao sistema.

Para adicionar um novo KPI ao sistema, é necessário introduzir uma parametrização do novo KPI na tabela Parâmetros da BD MySQL, com o mesmo nome da coluna dos dados a processar na tabela Raw da BD Cassandra.

Por fim, para observar os valores do novo indicador, é necessário adicionar um *dashboard* na IU Grafana.

4.4 Deploy da aplicação na cloud

Para disponibilização da aplicação na *cloud*, foi utilizado a plataforma Google Cloud [30], utilizando os mesmos passos e ficheiros de configuração que foram usados no Kubernetes em ambiente local, através da aplicação Docker Desktop [29].

Depois da criação do *cluster*, através da Cloud Shell [28] foi utilizado o comando *kubectl* para fazer *deployment* dos objectos Kubernetes através dos ficheiros de configuração YAML.

O *deployment* da aplicação pode ser feito pela seguinte ordem de acções:

- Disponibilização da camada de armazenamento através da criação dos *statefull sets* de MySql e Cassandra;
- Criação do modelo de dados em cada uma das base de dados;

- Alojamento no Docker Hub das imagens Docker dos *workers* e dos serviços *Profile* e *Report*;
- Criação e exposição através de serviço do *Deployer* de *Profile* e *Report*;
- Disponibilização da imagem do Grafana [31] no kubernetes.

4.5 Sumário

A implementação apresentada é um protótipo de uma aplicação da arquitetura para um cenário específico. Contudo contém aspectos que são considerados essenciais para a aplicação da estratégia proposta.

Foram implementados serviços orquestradores (*Reporter* e *Profiler*) que têm a função de receber os pedidos dos utilizadores para interagir com o sistema e *workers* que executam diretamente no ambiente de execução para fazer a agregação dos dados. Para disponibilizar os módulos implementados em objectos Kubernetes foi necessário transformar os mesmos em imagens Docker através do processo de *conteinerização*. No Kubernetes foram disponibilizados componentes, que dão suporte ao sistema, como os *Cronjobs* que lançam as instâncias de *workers* com uma periodicidade definida pelo utilizador ou os *StatefulSet* que permitem manter estado durante a execução do sistema e são usados para alojar as base de dados Cassandra e o MySql. A base de dados por sua vez, contém o modelo que dá suporte a todas as fases de processamento descritas no capítulo. Por fim foi desenvolvida uma interface gráfica no Grafana, que disponibiliza formulários para parametrização do sistema e *dashboards* para disponibilizar resultados dos dados agregados.

5

Análise de resultados

Neste capítulo, será feita a análise da experiência da aplicação da arquitetura proposta, tendo por base a implementação apresentada no capítulo anterior.

5.1 Aspetos relevantes

Existem inúmeros aspetos que determinam a viabilidade de uma solução, sendo que serão abordados alguns dos mais relevantes, de forma a se obter um ponto de comparação com outras arquiteturas. Cada arquitetura tem as suas vantagens e desvantagens, equilibrando a sua flexibilidade, complexidade e desempenho, pois uma arquitetura que esteja focada na flexibilidade e no desempenho resultará numa solução complexa. Uma solução simples e flexível terá, provavelmente, problemas de desempenho, e uma arquitetura com baixa complexidade e alto desempenho resultará numa solução rígida e inflexível [44].

Os aspetos abordados para avaliar a solução são as seguintes:

- Disponibilização da solução - Avaliar a complexidade de disponibilizar o sistema do zero ao utilizador;
- Robustez da solução - Todas as soluções estão sujeitas a falhas e a indisponibilidade. Por isso, é necessário ter em conta os mecanismos de recuperação a falhas, de forma a se alcançar uma solução mais robusta;

- Atualizações à solução – Eventualmente, qualquer aplicação necessita de atualizações ou manutenção. A capacidade de efetuar e disponibilizar alterações de forma eficiente num produto, durante a sua utilização, constitui um aspeto muito relevante, tanto na viabilidade como na resposta às necessidades do utilizador.

- Flexibilidade - A solução deve estar preparada para sofrer alterações de requisitos, adaptando-se às necessidades do utilizador, sem envolver mudanças estruturais. Este é um aspeto que tem reflexo direto na flexibilidade da solução, uma vez que, com a adição de parametrizações, permite que o sistema responda de forma eficaz à mudança de requisitos. Também é necessário ter em conta que o excesso de configurações e parametrizações pode ter impacto no desempenho e, principalmente, na complexidade;

- Escalabilidade - Como uma solução está sempre a evoluir, é necessário que a mesma esteja preparada para escalar horizontalmente e verticalmente.

- Elasticidade - A elasticidade é o ponto em que um sistema é capaz de se adaptar às mudanças da carga de trabalho por prover ou desprover recursos de forma automática, de tal forma, que em cada momento os recursos disponibilizados correspondem à demanda atual o mais próximo possível das necessidades [49].

5.1.1 Disponibilização da solução

A solução está dividida em camadas de responsabilidade, sendo que cada uma delas é constituída por módulos, e para cada um dos últimos existe uma imagem Docker que contém tudo o que é necessário para executar a sua função no sistema. A disponibilização da solução passa por colocar todas as imagens Docker a executar e expostas no ambiente Kubernetes. Para isso, é necessário criar as imagens Docker dos desenvolvimentos que se pretende disponibilizar ou obter as imagens oficiais de software de terceiros a utilizar na solução final. Depois, é utilizado o comando *kubectl*, que utiliza ficheiros de configuração escritos em formato YAML ou JSON. Os ficheiros contêm todas as configurações necessárias para o Kubernetes disponibilizar a imagem Docker dentro do ambiente de execução e a entidades externas.

5.1.2 Robustez da solução

Durante a execução da aplicação, o ambiente de execução está sujeito a erros e indisponibilidade. Deste modo, pretende-se avaliar a forma como o Kubernetes recupera desses mesmos erros

5.1.2.1 Falha na execução de um *Pod*

Processo de auto-healing do *Pod*, fornecida pelo Kubernetes. Um worker (*Pod*) é rapidamente substituído, se falhar, por uma réplica exata. A substituição é feita de forma transparente para o utilizador e administrador do sistema.

5.1.2.2 Falha de um *Pod* Cassandra

A base de dados Cassandra oferece réplicas de um *Pod*, gerida por um *Pod* coordenador, que tem em conta todas as réplicas. O coordenador mantém todas as réplicas atualizadas e o número de réplicas parametrizadas ativas.

5.1.2.3 Falha de um *Pod* de MySQL

Em caso de falha, o *Pod* de MySql é recuperado, como qualquer outro *Pod* no sistema de execução. No entanto é perdido o estado da base de dados MySql no processo de recuperação que substitui o *Pod* em questão. Isto acontece porque, ao contrário dos *Pods* de Cassandra, que têm um sistema de replicação, a base de dados MySql que contém parametrizações e dados temporários não requer o mesmo nível de robustez.

5.1.2.4 Falha na execução de uma *query*

Para colmatar falhas pontuais dos *workers*, o ambiente de execução realiza tentativas das transações de forma atómica. O software faz uso dos sistemas de gestão de base dados presentes no Cassandra e MySQL, para que as leituras e escritas sejam feitas de forma correta.

5.1.3 Atualização da solução

Os conteúdos programáveis da solução estão disponíveis em imagens Docker. Para introduzir uma nova atualização no sistema, é necessário colocar-se a imagem Docker atualizada no Docker Hub [7]. Depois da imagem estar disponível a atualização estará também disponível ao sistema.

5.1.3.1 Impacto da atualização no sistema

Depois da imagem Docker estar atualizada, é realizado o *rollout* da atualização pelos *Pods*, através do seu *deployer*, que tem a responsabilidade de substituir os *Pods* em execução por *Pods* com a imagem atualizada. À medida que os *Pods* antigos vão terminando o seu ciclo de vida, novos *Pods* com a atualização são criados e, eventualmente, substituem na totalidade a versão antiga, sem que o utilizador sinta indisponibilidade da aplicação.

5.1.4 Deploy da aplicação

Em primeiro lugar, para se colocar a solução a funcionar, é necessária uma instância Kubernetes disponível, com os recursos mínimos para poder alojar todas as imagens dos componentes. A versão de Kubernetes utilizada durante o desenvolvimento da tese, em ambiente local, foi a versão 1.15.5, que está alojada na versão 2.2.04 da aplicação Docker desktop. O ambiente local foi utilizado para desenvolver e testar localmente antes de ser alojada a solução na *cloud*. Na *cloud* foi utilizada a versão 1.15.12-gke.20 do Kubernetes.

Para que a solução esteja disponível, têm de existir os recursos recomendados para suportar os componentes da arquitetura. Para alojamento e disponibilização da solução na *cloud* foi criado um *cluster* Kubernetes, que é constituído por 3 VM's com o total de 6 CPU *cores* e 12 GB's de memoria no total.

Para criar os objetos dos componentes da aplicação, é apenas necessário executar os ficheiros de configuração YAML. Cada YAML é diferente para cada tipo de objeto, sendo que cada um tem uma referência para a imagem Docker do componente, no servidor Docker Hub [7].

Para a base de dados foram disponibilizados ficheiros executáveis, que utilizam o comando *kubectl* para executar os *scripts* de criação do modelo de dados na BD Cassandra e na BD MySQL.

5.1.5 Flexibilidade

5.1.5.1 Atualização do sistema

A arquitetura proposta tem uma base modular que separa os módulos em responsabilidades individuais e independentes dos outros módulos, mas que interagem entre si para um fim comum. Como todos os módulos são disponibilizados em imagens Docker que interagem através de serviços REST e a API java, existe facilidade de se adicionar e remover módulos, pois apenas é necessário alterar sua interface.

A execução da aplicação tem em conta parametrizações que são atualizadas em tempo de execução pelo utilizador. As parametrizações disponíveis são disponibilizadas por KPI:

- Tempo em minutos para realizar novos *reports*;
- Tempo em minutos que define a janela temporal dos valores *raw* a serem obtidos;
- Tempo em minutos para criar *profiles*;
- Valor máximo do KPI;
- Valor mínimo do KPI.

5.1.6 Escalabilidade

A arquitetura tem como objectivo oferecer dinamismo tanto na escalabilidade vertical como na horizontal.

A escalabilidade horizontal é a capacidade de adicionar (ou remover) nós a um sistema, como adicionar um novo computador para uma aplicação de software distribuída. A verticalidade da arquitetura é garantida graças à utilização da *cloud* que permite adicionar o número desejado de nós através de uma parametrização. A escalabilidade vertical também é garantida através da *cloud*, pois cada nó adicionado na *cloud* pode ser atualizado com os recursos necessários. Todas estas parametrizações reflectem-se através de custos, mas garante flexibilidade nos recursos consoante a sua necessidade.

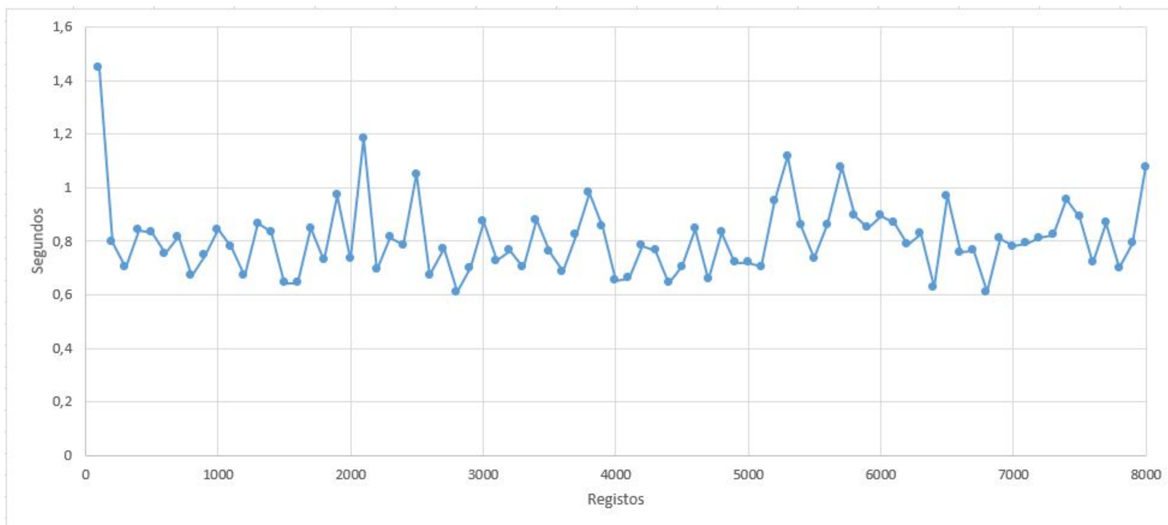

Figura 5.1: Cenário 1 - Tempo de processamento por blocos de 100 registos por 80 workers num total de 8000 registos

5.2 Análise de desempenho

5.2.1 Testes locais

Após o desenvolvimento, foram feitos testes de desempenho à solução. Foram feitos testes com diferentes quantidades de registos, sendo que o número de registos processados aumenta ou diminui, com a variação da periodicidade e da janela temporal definida na parametrização do sistema. Através dos testes, é possível medir o desempenho do sistema a partir do tempo de processamento dos *workers* e do tempo total que um KPI demora a ser calculado.

O cenário da figura 5.1 permite verificar o comportamento do sistema com uma grande quantidade de *workers* e com um pequeno número de dados a processar. Este cenário demonstra um indicador com uma periodicidade e janela temporal pequena, em que existiria a necessidade de uma atualização constante do indicador. Cada *worker* está a processar 100 registos e serão lançados um total de 80 *workers*, em série para processar um total de 8000 registos. No total o processamento levou 54,6 segundos a devolver 80 resultados do indicador.

Na figura 5.2 testa-se o outro extremo do cenário anterior, em que existe uma periodicidade e uma janela temporal maior. Por consequência existe menos *workers* neste cenário, mas cada *worker* tem uma maior quantidade de dados para processar. O indicador neste cenário é atualizado menos vezes, mas tende a ser

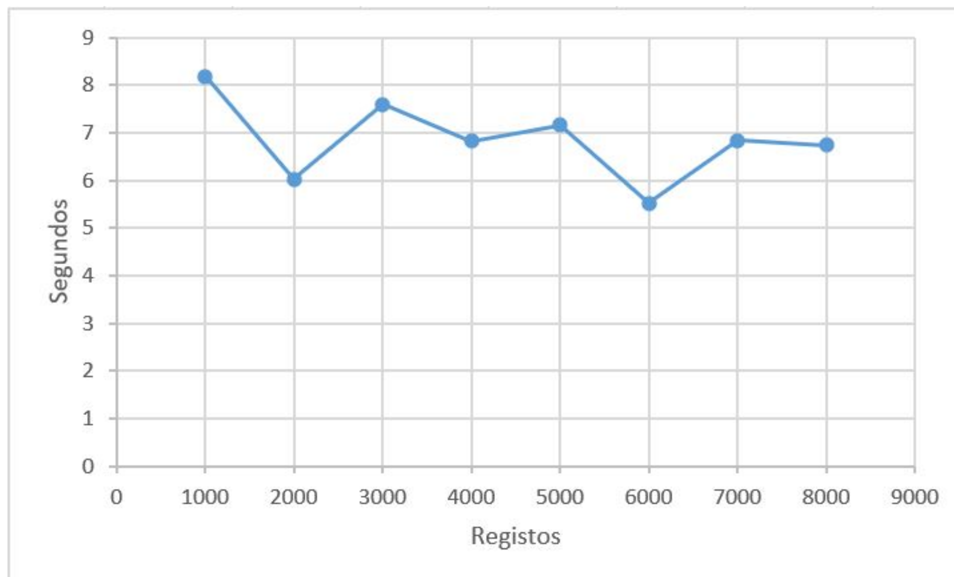

Figura 5.2: Cenário 2 -Tempo de processamento por blocos de 1000 registos por 8 *workers* num total de 8000 registos

mais preciso, pois baseia-se numa amostra maior. Cada *worker* está a processar 1000 registos e com 8 *workers* em série a processar no total 8000 registos. O processamento neste cenário levou 64,67 segundos em série a devolver apenas 8 resultados.

Na figura 5.3 trata-se de um cenário balanceado em relação aos dois cenários apresentados. Cada *worker* está a processar 500 registos e com 16 *workers* a processar os mesmos 8000 registos. No total o processamento levou 34,9 segundos a calcular 16 resultados do KPI.

5.2.2 Testes na cloud

Foi testado o cenário ilustrado na imagem 3.1 no capítulo 3, depois da solução estar disponível na *cloud* através dos passos apresentados na secção 4.4. A imagem 5.4 apresenta a consola de monitorização do serviço Kubernetes na Google Cloud durante a execução dos testes.

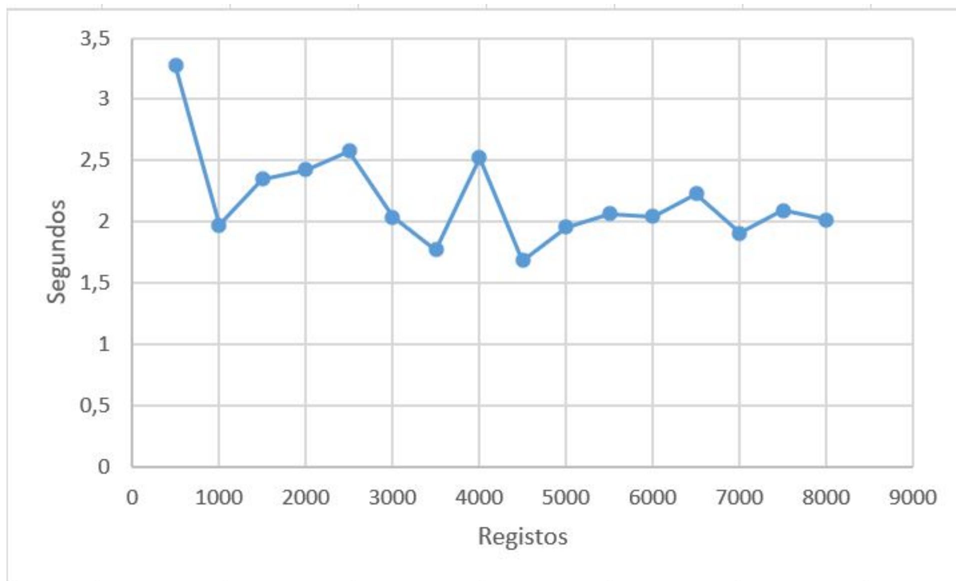

Figura 5.3: Cenário 3 - Tempo de processamento por blocos de 500 registos por 16 *workers* num total de 8000 registos

Google Cloud Platform KPiPorfileTest Search products and resources

Workloads REFRESH DEPLOY DELETE

Cluster cluster-2 Namespace RESET SAVE BETA

Is system object : False Status : OK: OK OR Status : OK: Running Filter workloads

Name	Status	Type	Pods	Namespace
cassandra-operator	OK	Deployment	1/1	default
grafana	OK	Deployment	1/1	default
kube-proxy-gke-cluster-2-default-pool-b357260c-f0q1	Running	Pod	1/1	kube-system
kube-proxy-gke-cluster-2-default-pool-b357260c-kwlm	Running	Pod	1/1	kube-system
kube-proxy-gke-cluster-2-default-pool-b357260c-r44q	Running	Pod	1/1	kube-system
mysql	OK	Deployment	1/1	default
mysql-client	Running	Pod	1/1	default
profilerbitrate	OK	Cron Job	0/2	default
profilerbitrate-manual-j6x	OK	Job	0/1	default
profilerfactory	OK	Deployment	1/1	default
raw	OK	Cron Job	0/2	default
raw-manual-559	OK	Job	0/1	default

Figura 5.4: *Dashboard* Kubernetes de execução do projeto na Google Cloud

O teste foi baseado no cenário 3 onde são processados 1000 registos, mas com o intuito de causar paralelismo no processamento. Neste cenário o processamento de cada *worker* do Report e *Profile* é configurado com uma periodicidade de 1 minuto e com um tempo de execução de 3 minutos cada. O tempo de execução

foi introduzido através de um *sleep* de 3 minutos para cada tipo de *worker*.

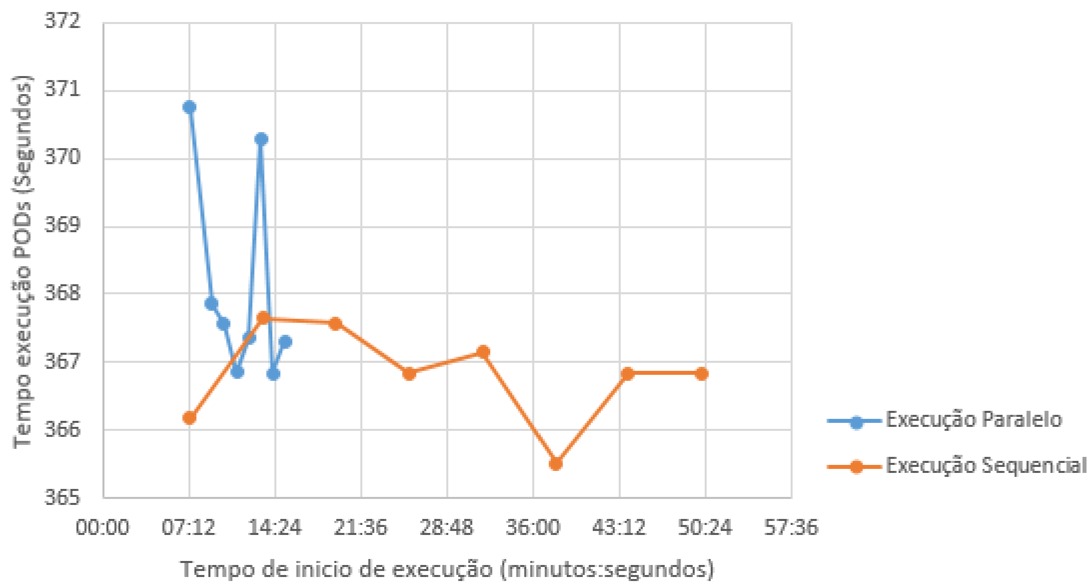

Figura 5.5: Comparação do processamento sequencial e paralelo de *workers* a processar 1000 registos com tempo de execução de 3 minutos, num total de 8000 registos

Nos gráficos da imagem 5.5 é possível comparar a execução do teste de forma sequencial e paralela. Em termos de tempo de execução por *worker* verificou-se que não houve grande variação nos resultados, pois os tempos de execução foram semelhantes nos dois testes. Verificou-se um grande impacto no tempo total de processamento da execução de todos os 8000 registos. O processamento dos 8 *workers* demoraram em sequência 50 minutos para obter os KPIs (aproximadamente 6 minutos de execução por 8 *Pods*) e em paralelo conseguiu-se fazer todo o processamento em apenas 15 minutos.

5.2.3 Conclusões da análise

Através dos testes, foi possível concluir que a quantidade de dados está diretamente relacionada com o tempo de processamento, não parecendo existir uma degradação no tempo de execução em paralelo relacionado com a infra-estrutura e arquitetura do sistema. Como se pode verificar, deve existir sempre uma preocupação na configuração da quantidade de *workers* em cada processamento e a quantidade de dados que cada *worker* está a tratar. Essa configuração é dada através da periodicidade (que se reflete no número de *workers* presente) e na janela

temporal (que se reflete na quantidade de dados por *worker*), presentes na tabela parâmetros. Devido à necessidade do ajuste destes valores para cada KPI, foi introduzida um formulário da interface gráfica, que permite ao utilizador alterar estes valores em tempo de execução. O objetivo é ajustar as parametrizações conforme a necessidade, de forma a obter os melhores resultados em situações com variação na quantidade do fluxo de dados a serem processados e na necessidade de uma atualização dos indicadores.

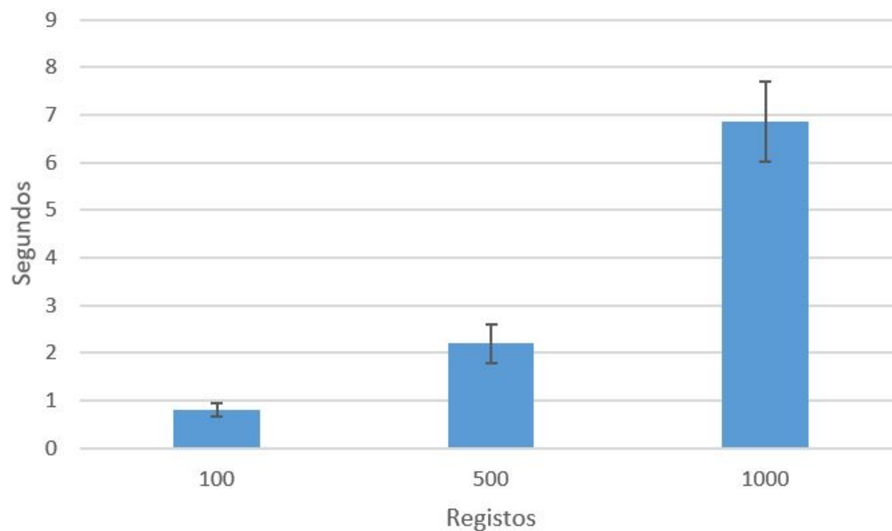

Figura 5.6: Tempo médio de processamento da execução de cada *Pod* nos três testes dos cenários locais

O cenário 1 (100 registos), em termos de atualizações, tem o maior taxa de refrescamento em comparação aos outros cenários, apesar do tempo total de processamento ser muito semelhante ao cenário 2 (1000 registos). Este tipo de cenário estaria a associado a um indicador crítico para o sistema e deve-se manter atualizado o mais possível. No cenário 2 o indicador calculado teve acesso a uma maior quantidade de dados, durante o processamento. Tornando este indicador mais preciso, mas mais facilmente desatualizado, em comparação aos outros dois cenários. O cenário 3 (500 registos) é uma balanceamento da taxa de atualização e a quantidade de dados a processar, o que levou a ser o cenário mais rápido a processar em relação aos cenários anteriores. Isto aconteceu porque, o número de registos a processar por um *worker* não é tão grande como no cenário 2 e o número de *workers* é muito menor que o cenário 1. É possível verificar no gráfico resumo 5.6, a relação dos 3 cenários locais anteriores quanto ao tempo médio do processamento de dados e a quantidade de dados por *worker*.

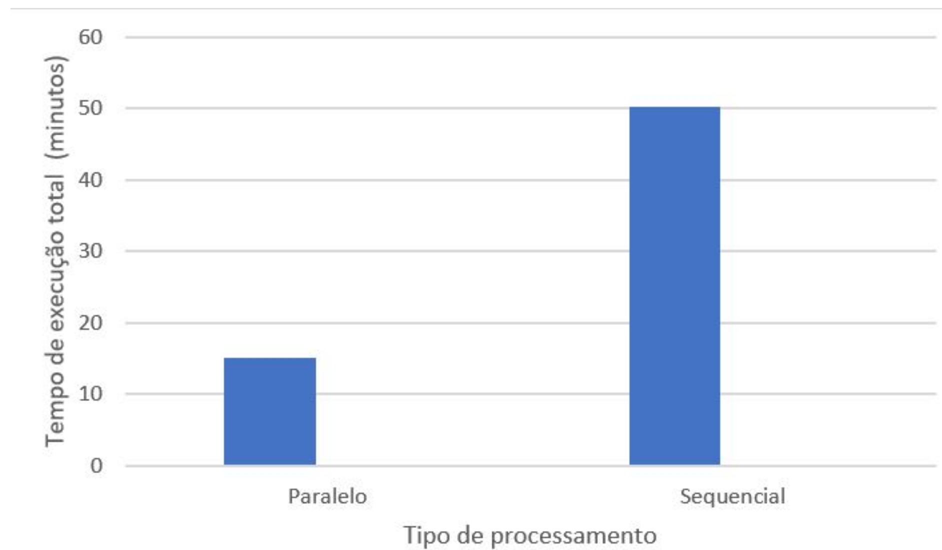

Figura 5.7: Comparação do tempo de execução do processamento de forma paralela e sequencial

Na figura 5.7 é possível ter uma melhor visualização da diferença de tempo, que existe entre o processamento paralelo e o sequencial. Os dados neste cenários foram processados na *cloud* com as características indicadas na secção 5.1.4. Existe uma grande vantagem em utilizar processamento paralelo em comparação a um processamento sequencial, isso é facilmente perceptível pela diferença do tempo total de processamento nos dois testes. Esta vantagem é maior se o tempo de processamento dos *workers* durante as fases de *Report* e *Profile*, for superior ao tempo de periodicidade na criação dos *workers*. Os cenários ideais são, quando existem *workers* que demoram mais tempo e a periodicidade é reduzida, pois neste caso existiria haver mais *workers* a processar dados em paralelo.

6

Conclusão e Trabalho Futuro

6.1 Conclusão

No desafio proposto pela Nokia, é apresentado um cenário real de uma aplicação com pouca capacidade de escalabilidade, limitada na capacidade de processamento/armazenamento e na capacidade de se adaptar às mudanças, tendo em conta a carga de trabalho de forma automática. Desta forma, definiu-se como objetivo principal encontrar uma solução para esta situação, através de uma arquitetura modular que utiliza a *cloud*.

No primeiro capítulo, foi feita uma contextualização da aplicação SQM e das características da aplicação para suportar o negócio. Devido à característica da sua arquitetura atual, não é possível dar respostas às necessidades dos operadores. A solução encontra-se na definição da arquitetura, criando camadas de abstração que se focam numa responsabilidade específica e que trabalham entre si para chegar a um objetivo comum. Utilizando a arquitetura na *cloud*, torna-se possível que os módulos presentes em cada camada da arquitetura tenham acesso a recursos de forma dinâmica, conseguindo, assim, abstrair-se de problemas relacionados com o hardware na sua escalabilidade.

Depois da análise do problema e da definição de objetivos é necessário estabelecer as tecnologias que devem ser utilizadas na solução no cenário proposto. Neste sentido, foi efetuada uma comparação de tecnologias candidatas por cada camada. As camadas propostas na arquitetura estão dividas da seguinte forma:

uma camada de armazenamento, onde são colocados os dados que serão consumidos para obter os indicadores; uma camada de processamento, onde é feita a agregação/análise dos dados provenientes de dispositivos de rede; e um ambiente de execução, que fará a orquestração dos pedidos e coordenação dos módulos constituintes. As tecnologias utilizadas têm um grande impacto na robustez e na estabilidade da solução. Por isso, não devem ser escolhidas de ânimo leve.

Para a arquitetura proposta, foi necessário definir como seriam constituídas as diferentes camadas. Com a utilização de imagens Docker num ambiente de execução kubernetes, existe a possibilidade de se reutilizar a arquitetura em outros cenários, tornando esta solução mais flexível. A solução de utilizar imagens Docker também faz com que a mesma seja independente das linguagens de programação e tecnologias utilizadas. Existe também uma API REST exposta, para pedidos provenientes de uma interface com o utilizador. Para este cenário, foram colocados em contentores Docker os módulos já existentes de criação de *reports* e análise de dados, para poderem ser executados de forma paralela e em tempo útil, na cloud.

Por fim, foi levada a cabo uma prova de conceito, que põe em prática a solução proposta, a fim de demonstrar o comportamento esperado. Para isso, foram desenvolvidos os módulos constituintes e configurado um ambiente de execução local. Desta forma, é possível demonstrar a interação entre as camadas da arquitetura, de forma a responder a um pedido de um utilizador e gerar KPIs através dos recursos, num ambiente de execução como o Kubernetes.

O processo de transformação e modularização de uma aplicação é uma tarefa árdua, que envolve várias tecnologias e a aprendizagem de múltiplos conceitos. A adoção desta solução envolve uma mudança de paradigma no desenvolvimento de software e envolverá custos na transformação e criação da infraestrutura. Apesar disso, é previsível que sem a adoção desta solução, a aplicação não conseguirá cumprir de forma eficiente a sua função, tendo um aumento do custo na manutenção/adição de novas funcionalidades até ao ponto de se tornar insuportável e com dificuldade em adaptar alterações tecnológicas, que são cada vez mais frequentes.

No início do documento, mais propriamente na Seção 1.4 foram definidas metas que foram alcançadas da seguinte forma:

- A utilização da *cloud* torna possível a escalabilidade vertical e horizontal, através da utilização e gestão de recursos presentes na mesma. Desta forma, as máquinas físicas deixam de ser necessárias;

- A robustez é garantida a partir das tecnologias utilizadas e mecanismos presentes no kubernetes, como o balanceamento de carga, a replicação e a recuperação automática de *Pods*;
- A capacidade de processamento com a paralelização de pedidos na execução das imagens faz com que os pedidos sejam respondidos em tempo real;
- A utilização de contentores e os mecanismos de *rollout* de atualizações no ambiente de execução permitem que um novo *deploy* esteja presente sem *downtime*.

6.2 Dificuldades

Durante a escrita da dissertação e a implementação da prova de conceito, surgiram dificuldades. Alguns desses obstáculos foram possíveis de ultrapassar; outros, dada a sua complexidade, tiveram impacto no desenho da solução ou na sua implementação.

Como referido anteriormente, as características da aplicação existente, como os dados que a aplicação consome e os KPIs que gera, não foram conhecidas durante o desenvolvimento da prova de conceito. Esse facto dificultou a escolha de tecnologias, pois existe sempre uma correlação entre as tecnologias utilizadas e os requisitos de negócio.

Durante este projeto, foi necessário algum tempo para assimilar conceitos e arquiteturas que tiram partido dos recursos disponíveis em *clouds*. Para além disso, foi necessário utilizar tecnologias na fase de implementação, que não haviam sido abordadas na licenciatura nem no mercado de trabalho, como o Docker e o Kubernetes.

A utilização da biblioteca Java para interagir com o Kubernetes foi também um desafio, pois ainda lhe falta alguma maturidade no ponto de vista de funcionalidades. Tendo a sua primeira versão estável em maio de 2018, ainda se trata de uma realidade relativamente recente.

Também no desenvolvimento da interface gráfica houve dificuldades. A decisão de utilizar o Grafana facilitou a criação dos *dashboards*, que dão visibilidade do funcionamento do sistema ao utilizador. Infelizmente ainda não existem *dashboards* criados a partir de *queries* que utilizam uma *datasource* Cassandra. Na verdade, à data, apenas existe para MySQL.

6.3 Trabalho Futuro

Como a definição da arquitetura e a proposta de uma solução, coloca-se os olhos postos no futuro, numa forma de aplicar a solução num ambiente produtivo. Para isso, é necessário um desenvolvimento futuro que torne esta possibilidade em algo concreto, através das seguintes estratégias:

- Evolução do código utilizado nas imagens, através de ferramenta de integração contínua de modo a criar um ciclo de desenvolvimento;
- Interação entre as imagens que correm no *Pod* e outras imagens no ambiente de execução através de um nome DNS;
- Utilização de uma interface gráfica semelhante ao Grafana, que suporte Cassandra;
- Adição de módulos de cálculo mais complexos no módulo de *profiling*.

Referências

[1] Apache cassandra leads nosql benchmark | datastax. https://www.datastax.com/apache-cassandra-leads-nosql-benchmark, . (Accessed on 06/27/2019). (p. 6)

[2] Apache hbase ™ reference guide. https://hbase.apache.org/book.html{#}zookeeper, . (Accessed on 06/26/2019). (p. 8)

[3] Connecting applications with services - kubernetes. https://kubernetes.io/docs/concepts/services-networking/connect-applications-service/. (Accessed on 05/11/2019). (p. 32)

[4] Create a base image | docker documentation. https://docs.docker.com/develop/develop-images/baseimages/. (Accessed on 05/03/2019). (p. 30)

[5] Dns for services and pods - kubernetes. https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/. (Accessed on 06/16/2019). (p. 32)

[6] | docker documentation. https://docs.docker.com/engine/reference/builder/, . (Accessed on 06/26/2019). (p. 30)

[7] Docker hub. https://hub.docker.com/, . (Accessed on 06/16/2019). (pp. 30 e 48)

[8] Docker registry | docker documentation. https://docs.docker.com/registry/, . (Accessed on 06/16/2019). (p. 30)

[9] Github - fabric8io/kubernetes-client: Java client for kubernetes & openshift. https://github.com/fabric8io/kubernetes-client. (Accessed on 06/23/2019). (p. 31)

[10] Internet of things | mongodb. https://www.mongodb.com/use-cases/internet-of-things. (Accessed on 06/16/2019). (p. 8)

[11] Jobs - run to completion - kubernetes. https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/. (Accessed on 06/16/2019). (p. 31)

[12] Overview - Mesosphere DC/OS Documentation, . URL https://docs.mesosphere.com/1.12/overview/. (p. 13)

[13] The SMACK Stack is the New LAMP Stack - Mesosphere, . URL https://mesosphere.com/blog/smack-stack-new-lamp-stack/. (p. 13)

[14] Introduction to MongoDB — MongoDB Manual, . URL https://docs.mongodb.com/manual/introduction/{#}key-features. (p. 8)

[15] Mongodb and mysql compared | mongodb. https://www.mongodb.com/compare/mongodb-mysql, . (Accessed on 06/16/2019). (p. 8)

[16] Network health monitoring overview. https://www.cisco.com/c/dam/en_us/training-events/product-training/prime-infrastructure-31/ja-nethealth/PI31-NetworkHealthMonitoringOverview-JobAid.pdf. (Accessed on 06/27/2019). (p. 6)

[17] Overview of kubectl - kubernetes. https://kubernetes.io/docs/reference/kubectl/overview/. (Accessed on 06/04/2019). (p. 32)

[18] Pods - kubernetes. https://kubernetes.io/docs/concepts/workloads/pods/pod/. (Accessed on 06/16/2019). (p. 31)

[19] Replicaset - kubernetes. https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/, . (Accessed on 06/16/2019). (p. 31)

[20] Repositories | docker documentation. https://docs.docker.com/docker-hub/repos/, . (Accessed on 06/23/2019). (p. 30)

[21] Service - kubernetes. https://kubernetes.io/docs/concepts/services-networking/service/. (Accessed on 06/16/2019). (p. 32)

[22] The smack stack is the new lamp stack - mesosphere. https://mesosphere.com/blog/smack-stack-new-lamp-stack/. (Accessed on 06/01/2019). (p. 15)

[23] What is a container? | docker. https://www.docker.com/resources/what-container. (Accessed on 05/15/2019). (pp. 11 e 12)

[24] Why does scalability matter, and how does cassandra scale? | datastax. https://www.datastax.com/dev/blog/why-does-scalability-matter-and-how-does-cassandra-scale. (Accessed on 06/27/2019). (p. 7)

[25] Viewing Pods and Nodes | Kubernetes. URL https://kubernetes.io/docs/tutorials/kubernetes-basics/explore/explore-intro/. (p. 31)

[26] Yaml syntax | grav documentation. https://learn.getgrav.org/16/advanced/yaml. (Accessed on 05/03/2019). (p. 32)

[27] Apache Cassandra. URL http://cassandra.apache.org/. (p. 7)

[28] Cloud shell | google cloud. URL https://cloud.google.com/shell. (p. 42)

[29] Docker desktop for mac and windows | docker. URL https://www.docker.com/products/docker-desktop. (p. 42)

[30] Compute engine: máquinas virtuais (vms) | google cloud. URL https://cloud.google.com/compute. (p. 42)

[31] Grafana: The open observability platform | grafana labs. URL https://grafana.com/. (pp. 39 e 43)

[32] Apache HBase ™ Reference Guide. URL https://hbase.apache.org/book.html{#}arch.overview.when. (p. 7)

[33] java/kubernetes at master · kubernetes-client/java · github. https://github.com/kubernetes-client/java/tree/master/kubernetes. (Accessed on 06/23/2019). (p. 31)

[34] kube-proxy - kubernetes. https://kubernetes.io/docs/reference/command-line-tools-reference/kube-proxy/. (Accessed on 05/15/2019). (p. 32)

[35] Red Hat drops MongoDB over concerns related to its Server Side Public License (SSPL) | Packt Hub. URL https://hub.packtpub.com/red-hat-drops-mongodb-over-concerns-related-to-its-server-side-public-license-sspl/. (p. 8)

[36] What is Kubernetes? - Kubernetes, 2018. URL https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/. (p. 13)

[37] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy H Katz, Andrew Konwinski, Gunho Lee, David A Patterson, and Ariel Rabkin. Above the clouds: A berkeley view of cloud computing, 2009. URL http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html. (p. 15)

[38] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. Microservices Architecture Enables DevOps : An Experience Report on Migration to a Cloud-Native Architecture The Architectural Concerns for Microservices Migration The Architecture of Backtory Before the Migration. IEEE Software, 33(3):42–52, 2016. ISSN 07407459. doi: 10.1109/MS.2016.64. (pp. 6 e 10)

[39] Elif Dede, Madhusudhan Govindaraju, Daniel Gunter, Richard Shane Canon, and Lavanya Ramakrishnan. Performance evaluation of a MongoDB and hadoop platform for scientific data analysis. page 13, 2013. doi: 10.1145/2465848.2465849. (p. 9)

[40] Raul Estrada and Isaac Ruiz. Big Data SMACK. 2016. ISBN 978-1-4842-2174-7. doi: 10.1007/978-1-4842-2175-4. URL http://link.springer.com/10.1007/978-1-4842-2175-4. (pp. ix, xi, 6, e 14)

[41] S. Hassan and R. Bahsoon. Microservices and their design trade-offs: A self-adaptive roadmap. In 2016 IEEE International Conference on Services Computing (SCC), pages 813–818, June 2016. doi: 10.1109/SCC.2016.113. (pp. 10 e 17)

[42] Eben Hewitt. Cassandra: The Definitive Guide. O'Reilly Media, Inc., 1st edition, 2010. ISBN 1449390412, 9781449390419. (p. 6)

[43] Revision History. Apache HBase Reference Guide. pages 247–248, 2012. ISSN 0301-0546. URL https://hbase.apache.org/apache_hbase_reference_guide.pdf. (p. 7)

[44] Anders Karlsen. Flexibility — a Software Architecture Principle | by Anders Karlsen | FAUN | Medium, jan 2019. URL https://medium.com/faun/flexibility-a-software-architecture-principle-6eafe045a1d4. (p. 45)

[45] Peter Mell and Timothy Grance. The NIST Definition of Cloud Computing Recommendations of the National Institute of Standards and Technology. Technical report. (p. 15)

[46] Kirit Modi and Yashpalsinh Jadeja. Cloud computing-concepts, architecture and challenges. doi: 10.1109/ICCEET.2012.6203873. URL https://www.researchgate.net/publication/254035330. (p. 15)

[47] Nisha Panwar, Shantanu Sharma, and Awadhesh Kumar Singh. A survey on 5G: The next generation of mobile communication. Physical Communication, 18:64–84, mar 2016. ISSN 18744907. doi: 10.1016/j.phycom.2015.10.006. URL https://www.sciencedirect.com/science/article/pii/S1874490715000531. (p. 1)

[48] Maria A. Rodriguez and Rajkumar Buyya. Container-based cluster orchestration systems: A taxonomy and future directions. Software: Practice and Experience, 0(0). doi: 10.1002/spe.2660. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2660. (p. 13)

[49] Nikolas Roman Herbst, Samuel Kounev, and Ralf Reussner. Elasticity in Cloud Computing: What It Is, and What It Is Not. Technical report, 2013. (p. 46)

[50] Prateek Sharma, Lucas Chaufournier, Prashant Shenoy, and Y. C. Tay. Containers and virtual machines at scale: A comparative study. In Proceedings of the 17th International Middleware Conference, Middleware '16, pages 1:1–1:13, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4300-8. doi: 10.1145/2988336.2988337. URL http://doi.acm.org/10.1145/2988336.2988337. (p. 11)

[51] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry

Peterson. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. SIGOPS Oper. Syst. Rev., 41(3):275–287, March 2007. ISSN 0163-5980. doi: 10.1145/1272998.1273025. URL http://doi.acm.org/10.1145/1272998.1273025. (p. 11)

[52] Dan Sullivan. NoSQL for Mere Mortals. Addison-Wesley Professional, 1st edition, 2015. ISBN 0134023218, 9780134023212. (p. 6)

[53] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. Architectural patterns for microservices: A systematic mapping study. 03 2018. doi: 10.5220/0006798302210232. (p. 10)

[54] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. F. De Rose. Performance evaluation of container-based virtualization for high performance computing environments. In 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, pages 233–240, Feb 2013. doi: 10.1109/PDP.2013.41. (p. 11)

[55] A. Yassine, H. Rahimi, and S. Shirmohammadi. Software defined network traffic measurement: Current trends and challenges. IEEE Instrumentation Measurement Magazine, 18(2):42–50, April 2015. ISSN 1094-6969. doi: 10.1109/MIM.2015.7066685. (pp. ix e xi)