FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Near real-time analytics engine for vitals and environmental monitoring of first responders

José Pedro Machado

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: João Paulo Cunha

July 27, 2020

Near real-time analytics engine for vitals and environmental monitoring of first responders

José Pedro Machado

Mestrado Integrado em Engenharia Informática e Computação

Aprovado em provas públicas pelo Júri:

Presidente: Cristina Ribeiro
Arguente: Ilído Oliveira
Vogal: João Paulo Cunha

July 27, 2020

Abstract

First Responders (FR) are exposed on a daily basis to extreme conditions not only because of their duties but also due to the type of environment in which they perform. If the exposure to these conditions is not monitored, it may cause several health problems in long and short-term. This exposure can be controlled through vitals and environmental sensors placed in these professionals. However, the collected data is not relevant if it is not analyzed and processed on useful time. This analysis has the most benefits the closer it is to the moment the data is collected, allowing a more informed decision making.

Although there are numerous data processing systems in *IoT*, these systems do not have the necessary requirements for what is intended with this engine, either because they are proprietary solutions or because they are limited solutions in terms of supported applications and capacity processing.

The purpose of this thesis is to study and implement a service-based analytics engine in the cloud enabling near real-time processing of the data that was collected by FR wearable sensors through an IoT system called "WeSENSS". This engine is called VRAnalytics (VitalResponder Analytics).

For the intended purpose, it was necessary to ensure the encapsulation of applications and the creation of an infrastructure that allows the import of data, its processing and its exportation. To ensure the encapsulation was used virtualization using containers. Thus, if there is a faulty application, it will not compromise the functioning of the system. This entire system was assembled in Kubernetes, taking advantage of the benefits of this technology: self-regeneration of faulting containers, horizontal and vertical scalability of all components, and the integrated system monitoring tools.

The resulting analytics engine is a modular and scalable system that allows the processing of data provided by any Web server and exporting them to the same server they were provided from or to another one.

The engine was tested using different applications and configurations, testing its capabilities in terms of scalability, interoperation with *WeSENSS* and the usage of different programming languages, showing that the VRAnalytics engine fulfilled all the initial requirements and that it could be used in real emergency situations.

Keywords:First Responders, IoT, Cloud Computing, Healthcare, Analytics, Kubernetes,Docker.

Resumo

Os socorristas e os bombeiros (FR) estão diariamente sujeitos a condições extremas, não só devido ao tipo de funções que desempenham mas também ao tipo de ambiente em que o fazem. Se a exposição a estas condições não for controlada pode, a longo e curto prazo, causar diversos problemas de saúde. Esta exposição pode ser controlada através de sensores biométricos e ambientais colocados nestes profissionais. No entanto, os dados recolhidos não são relevantes se não forem analisados e processados, análise esta que tem mais benefícios quanto mais próxima for do momento em que os dados forem recolhidos, permitindo a tomada de decisões mais informada.

Apesar de existirem inúmeros sistemas de processamento de dados em *IoT*, estes sistemas não possuem os requisitos necessários para o que se pretende com este motor, ou porque são soluções proprietárias ou porque são soluções limitadas em termos de aplicações suportadas e de capacidade de processamento.

O objetivo desta dissertação é o estudo e implementação de um motor de serviços na cloud que permita o processamento em tempo quase real dos dados recolhidos pelos sensores *wearable* através de um servidor de *IoT* chamado *WeSSENS*. Este motor é chamado VRAnalytics (VitalResponder Analytics).

Para o efeito pretendido teve que se garantir o encapsulamento das aplicações e a criação de toda uma infraestrutura que permita a importação de dados, o seu processamento e a exportação de dados. Para garantir o encapsulamento foi utilizada a virtualização por contentores. Assim, se existir uma aplicação faltosa, esta não vai comprometer o funcionamento de todo o sistema. Todo este sistema foi montado em *Kubernetes*, tirando partido das vantagens desta tecnologia: autoregeneração de contentores faltosos, escalabilidade horizontal e vertical de todos os componentes, e as ferramentas integradas de monitorização do sistema.

O resultado obtido é um sistema modular e escalável que permite o processamento de dados proveninetes de qualquer servidor *Web*, processar esses dados utilizando qualquer linguagem de programação que seja suportada em *Docker* e exportar os mesmos para o mesmo servidor de onde provieram ou para um outro.

O motor foi testado utilizando diferentes aplicações e configurações, testando as suas capacidades no que toca a escalabilidade, interoperação com o *WeSENSS* e a utilização de diferentes linguagens de programação, demonstrando que o *VRAnalytics* cumpre todos os requsitos iniciais e que poderá ser utilizado em situações de emergência reais.

Keywords:Socorristas, IoT, Computaçao na Cloud, Healthcare, Analytics, Kubernetes,Docker.

Agradecimentos

Gostaria de em primeiro lugar agradecer ao meu orientador, Professor João Paulo Cunha não só pela orientação e apoios prestados mas também por me ter dado a oportunidade de desenvolver este projeto que alia duas grandes paixões da minha vida, Informática e Bombeiros, agradecer ao Duarte Dias por ter sido incansável durante todo o desenvolvimento e escrita da tese e também pela amizade e apoio ao longo deste tempo.

Agradecer também à equipa do *BRAIN* do INESC TEC que em todas as reuniões semanais proporcionaram uma troca de conhecimentos e de ideias, em especial ao Vitor Minhoto que sempre se mostrou disponível para tudo que precisasse.

Aos meus colegas de curso com quem partilhei e cresci estes cinco anos, sem eles não estaria aqui hoje.

Aos meus colegas Bombeiros com quem estive na linha da frente no combate à pandemia do COVID-19.

À Maria Viana que me acompanhou ao longo destes últimos três anos e me apoiou sempre que precisei, e durante o desenvolvimento e escrita desta dissertação não foi exceção. Agradecer-lhe por ter tido a paciência de lidar comigo nos momentos menos bons e por ter partilhado comigo os melhores momentos.

Aos meus avós que acreditaram em mim e me apoiaram durante todo o meu percurso académico.

E um agradecimento especial aos meus pais e irmã que sem eles nada disto era possível, que ao longo destes anos deram-me todas as ferramentas para que eu pudesse estar aqui hoje e por todo o amor e carinho que me deram.

A todos o meu sincero obrigado.

Josè Pedro Dias de Almeida Machado

"The greatest enemy of knowledge is not ignorance,
it is the illusion of knowledge."

Stephen William Hawking

Conteúdo

Lista de Figuras

Lista de Tabelas

Acrónimos

API	Application Programming Interface
IOT	Internet Of Things
BI	Business Intelligence
ECG	Eletrocardiograma
HTTP	Hypertext Transfer Protocol
URL	Uniform Resource Locator
LAN	Local Area Network
HTML	HyperText Markup Language
XML	Extensible Markup Language
NPM	Node Package Manager
MIT	Massachusetts Institute of Technology
FR	First Responders
PGG	Photoplethysmogram
STDOUT	Standard Output

Capítulo 1

Introdução

1.1 Contexto

Este trabalho foi desenvolvido no laboratório BRAIN/C - BER do INESC TEC no âmbito do projeto *Vital Responder*, que foi elaborado pelo INESC TEC. O *Vital Responder* é um sistema de monitorização de dados fisiológicos e ambientais direcionado para socorristas, de forma a endereçar as condições adversas que estes profissionais estão sujeitos quer pelo tipo de tarefas que realizam, quer pelo ambiente em que o fazem.

O sistema consiste em dois *wearables* equipados nos socorristas. Um recolhe dados fisiológicos como temperatura corporal, batimentos cardíacos e ECG (eletrocardiograma), e o segundo recolhe dados como a concentração de gases tóxicos, temperatura, pressão, humidade, etc. Os dados recolhidos por estes sensores são enviados para um dispositivo móvel por *bluetooth* que por sua vez envia os dados para um servidor central, onde são armazenados. Este servidor possui uma *API* que serve já múltiplos clientes, nomeadamente para visualização dos dados recolhidos em tempo real.

Apesar do sistema ir ao encontro do melhoramento da gestão de meios em situações de emergência e no melhoramento da qualidade de vida dos profissionais que a elas atendem, ainda há a necessidade de uma ferramenta que permita automatizar e tornar mais célere o processamento dos dados. O sistema existente apenas faz agregação dos dados recolhidos e qualquer processamento tem que ser feito de forma manual e pós-situação de emergência.O motor desenvolvido como base deste documento tem como fim funcionar juntamente com o sistema já desenvolvido para importação e exportação de dados por via da *API* do servidor do *Vital Responder*. O servidor do *Vital Responder* tem por nome *WeSENSS* e funciona com base em eventos que correspodem às situações de emergência e os dados recolhidos pelos sensores do *Vital Responder* estão organizados por fatores e literais, sendo que os fatores representam as medições e os literais repesentam as unidades das mesmas medições.

1.2 Motivação

A existência de um sistema que permite, a quem gere a situação de socorro, ter não só acesso aos sinais vitais dos operacionais e dados ambientais mas também a métricas e indicadores sobre o estado físico e psicológico ajudaria a melhorar não só a gestão de meios humanos no terreno, mas também no que diz respeito à saúde destes operacionais.

A análise de métricas como o stress e fadiga, que no sistema atual são calculadas de forma *offline*, podem ajudar a substituir elementos no terreno e evitar acidentes. Estudos mostram que a fadiga diminui a capacidade de resposta de profissionais como os bombeiros, estando diretamente relacionada com alguns acidentes [20]. A análise em tempo real destas métricas ajudaria numa tomada de decisões mais informada por parte de quem comanda a situação de emergência e eventualmente a prevenção de certos acidentes.

Em termos de motivação pessoal, uma vez que sou bombeiro voluntário, acredito que um sistema como estes possa resolver alguns problemas e lacunas existentes.

Apesar de esta dissertação ter surgido no âmbito do projeto *Vital Responder* e tem como fim colmatar algumas das suas lacunas, pretendemos também que esta solução possa ser aplicada a qualquer outro sistema de agregação de dados em *IoT*, definindo para isso um protocolo específico para integração.

1.3 Objetivos

Esta dissertação tem como objetivo o estudo e implementação de um sistema de processamento de dados distribuído na *Cloud* para *IoT*. O intuito do sistema é automatizar o processamento de dados permitindo a instalação de novos algoritmos de processamento através de um cliente na *Web*. Este sistema visa ainda permitir o controlo de todo o fluxo de dados a processar permitindo que estes venham de serviços externos e que o resultado do processamento possa ser também disponilizado para serviços externos ao sistema. O sistema deve permitir que sejam utilizadas várias linguagens de programação para os algoritmos de processamento e disponiblizar uma interface gráfica para permitir a gestão do sistema e a visualização do resultado do processamento dos dados.

1.4 Estrutura do Documento

Este documento está dividido em 6 capítulos. O capítulo 1 introduz a motivação e contexto desta dissertação e os objetivos da mesma. O capítulo 2 começa por fazer a análise do estado da arte onde primeiro são analisados os conceitos envolvidos nesta dissertação, de seguida são analisadas as soluções semelhantes à solução proposta. No capítulo 3 é feita a análise detalhada do problema que esta dissertação visa resolver, a abordagem para o resolver e a arquitetura da solução proposta e por último, é feita a análise das tecnologias que vão ser utilizadas. No capítulo 4 é descrita a implementação do sistema detalhadamente e é feita uma análise de todos os componentes do sistema. O capítulo 5 fala sobre os testes realizados à plataforma, detalhando as configurações utilizadas cada um e identificando que requisitos se pretende provar com cada teste. Por fim, no capítulo 6 são analisados os resultados obtidos e é feita uma conclusão sobre se o sistema cumpriu ou não os requisitos e se o funcionamento é o esperado.

Capítulo 2

Estado da Arte

2.1 Sistemas Cíber-Físicos

Os Sistemas Cíber-Físicos referem-se a um conjunto de sistemas computacionais feitos com vista a interagir com o mundo físico. Estes sistemas são constituídos por redes de objetos com software embebido que captam alterações do mundo físico utilizando sensores e por sua vez interagem com o mesmo através de atuadores [11]. A internet transformou a forma como o Homem interage e comunica entre si e os Sistemas Cíber-Físicos, por sua vez, vão alterar a forma como o Homem interage e controla o mundo em seu redor [28]. Podem ser aplicados numa grande quantidade de situações reais, quer sejam elas comerciais ou industriais, desde dispositivos e veículos a eletrodomésticos e wearables.

2.2 *IoT* e aplicações na área da Saúde

IoT foi definido primeiramente por Kevin Ashton como sendo uma rede de objetos inteligentes que partilham informação e dados reagindo às alterações do meio que os envolve [22]. A sua utilização tem vindo a aumentar nos últimos anos, assim como tem vindo a aumentar o número de objetos inteligentes conectados à Internet e estima-se que até 2025 esse número mais que duplique passando de 30.73 milhares de milhão para 75.44 milhares de milhão, como se observa na figura 2.1.

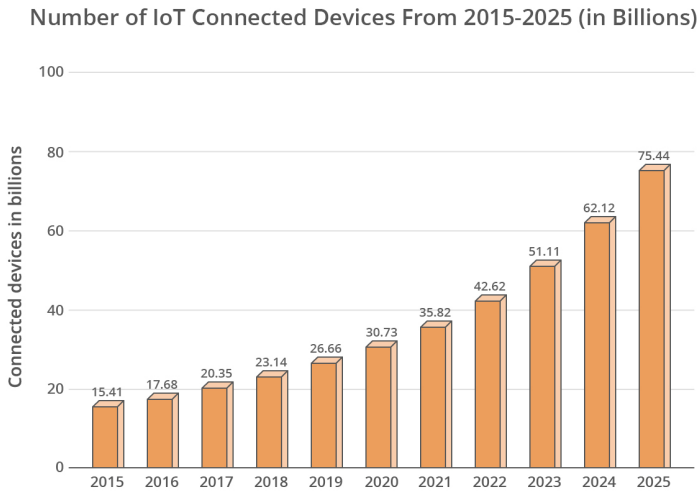

Figura 2.1: Evolução do número de dipositivos *IoT* conectados entre 2015-2025 [2].

Também na Saúde a sua utilização tem vindo a aumentar e cada vez são mais as aplicações com vista a melhorar a qualidade de vida e saúde das pessoas. Os sistemas utilizados na saúde passam principalmente pela recolha de dados relativos ao utilizador através de sensores e da análise de comportamentos do doente. Estes dados recolhidos são muito úteis para os médicos porque através da análise dos mesmos é possível fazer melhores prognósticos e recomendar tratamentos mais eficazes para a situação específica. O uso generalizado de sistemas de IoT na Saúde pode reduzir drasticamente os custos e melhorar não só a rapidez dos diagnósticos bem como aumentar a sua eficácia [17].

Outro conceito introduzido pelo *IoT* na Saúde é o conceito de Telemedicina em que não só a comunicação Médico-Paciente pode ser feita remotamente, mas também a monitorização e diagnóstico o podem. Este conceito destina-se não só a sistemas montados em hospitais que recolhem os dados e os armazenam para posterior análise mas também a sensores que os pacientes podem utilizar no dia-a-dia, sendo que esses dados são depois disponibilizados para análise pelos profissionais de saúde [29].

2.3 Análise de Dados em *IoT*

Todos os dados recolhidos por dispositivos necessitam de ser processados para poderem ser interpretados e para permitir uma tomada de decisões fundamentada. O processamento destes dados pode ser feito a diferentes níveis, tendo implicações no tempo de processamento assim como na capacidade computacional dos sistemas.

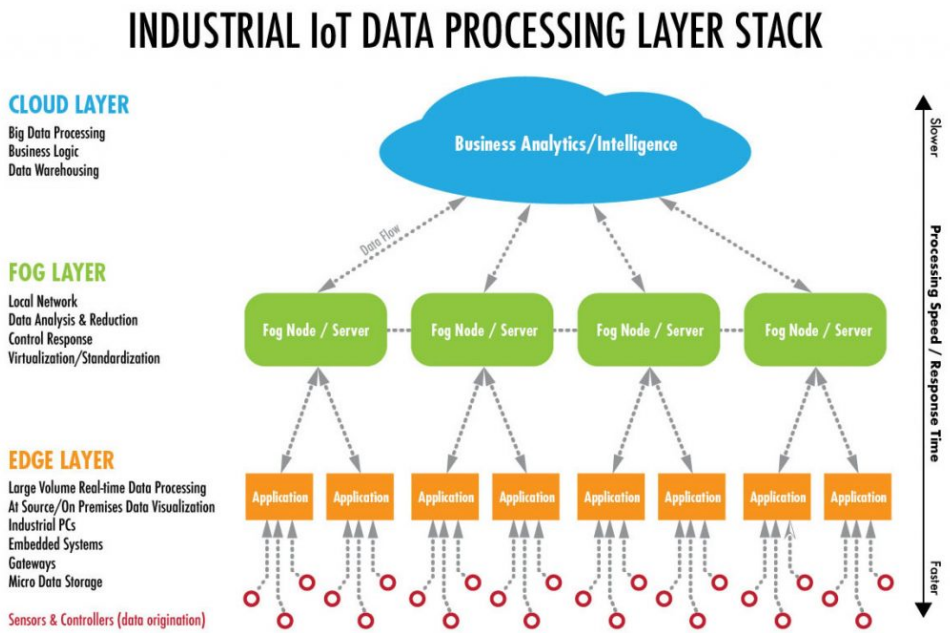

Figura 2.2: Camadas de Processamento de Dados *IoT* [1]

A literatura define que podemos dividir o processamento de dados em três camadas dando origem a três diferentes tipos de computação:

- *Cloud Computing* é o conjunto de aplicações disponilibizadas como serviços através da Internet assim como os recursos adjacentes a estes serviços (tanto de *hardware* como de *software*) [10]. O processamento de dados a este nível é mais lento, mas por se tratar de um sistema computacional centralizado é possível este ter mais poder computacional em relação às outras duas camadas, útil para operações mais pesadas.

- *Fog Computing* é uma plataforma altamente virtualizada, que consegue disponilizar os mesmos serviços que o *Cloud Computing* consegue mas de forma descentralizada, com baixa latência e apresenta um elevado número de nós que pode estar distribuído geograficamente. Apresenta uma latência inferior em relação ao *Cloud Computing*, contudo tem uma maior complexidade de implementação associada [13].

- *Edge Computing* é um paradgima que consiste em passar o processamento para um mais baixo nível, ao nível dos dispositivos que recolhem os dados, porque quando os dados são

produzidos pelos dipositivos torna-se mais eficiente fazer o seu processamento nos mesmos. Em termos de velocidade de processamento esta solução apresenta a maior velocidade das três camadas analisadas no entanto os dispositivos têm que possuir um maior poder de processamento o que também aumenta o custo do sistema [31].

2.3.1 Tipos de Sistemas

Alguns autores em [24] fazem ainda uma divisão dos tipos de sistemas de análise de dados de *IoT* em cinco diferentes tipos:

- **Análise de dados em tempo real** é tipicamente realizada quando há dados a ser recolhidos de sensores (em tempo real) e como tal, os dados estão em constante mudança e são necessárias técnicas de processamento que permitam obter resultados num curto espaço de tempo.
- **Análise de dados off-line** é utilizada quando não há necessidade de obter os dados processados rapidamente, contrariamente ao tipo de sistema visto no ponto anterior.
- **Análise de dados ao nível da memória** é aplicada quando os dados a analisar são inferiores em grandeza ao TB (Terabyte). Para melhorar a performance deste tipo de sistema são utilizadas normalmente diferentes tecnologias de base de dados. Este sistema é normalmente aplicado juntamente com a análise de dados em tempo real.
- **Análise de dados BI (Business Intelligence)** é usada quando os dados ultrapassam a grandeza do TB. Neste caso, os dados são processados num sistema de BI, onde são tratados como volumes.
- **Análise de dados Massiva** utiliza-se quando a quantidade de dados ultrapassa a capacidade de processamento dos sistemas de BI e das bases de dados tradicionais e tem que se utilizar sistemas de ficheiros especializados para mapear e reduzir os dados.

2.4 Arquitetura para sistemas de processamento de dados *IoT* em tempo real

Existem múltiplos sistemas de *IoT* em tempo real, cada um com a sua arquitetura e implementação, utilizando tecnologias proprietárias em vez de *standards*. Para endereçar os problemas de heterogeneidade, falta de interoperabilidade e portabilidade destes sistemas é definida [16] uma arquitetura genérica de alto nível. Esta arquitetura pretende cumprir os seguinte objetivos:

- Fácil configuração e portabildade.
- Formatos dos dados independentes da plataforma em que é aplicada.
- Interoperabilidade entre diferentes plataformas da *Cloud*.
- Mecanismo de comunicação eficiente.

Para isso, propõem um sistema distribuído que contempla comunicação bidirecional ente o servidor e os sensores ou atuadores através de um *middleware*, como podemos ver na figura 2.3. Os dados recolhidos são enviados pelo *middleware* para uma Interface genérica que vai fazer uma verificação de autenticidade dos dados. Para garantir a interoperabilidade foi definido um formato entre o auto-descritivo como o *JSON* ou *XML* e o binário para ter vantagens de ambos os formatos.

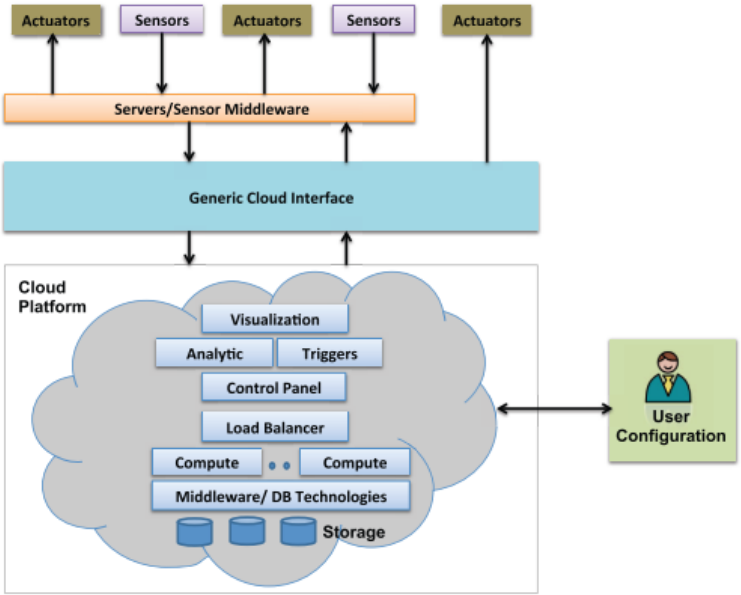

Figura 2.3: Arquitetura proposta em [16].

A arquitetura proposta contempla ainda a existência de um sistema de configuração de *triggers* e um sistema de processamento de dados para tomada de decisões informada, ambos controláveis por um painel de controlo desenhado em *HTML*, painel este que permite ainda a configuração da gestão e do *load balancer* para permitir distribuir a computação necessária para processamento dos dados pelos diferentes nós.

2.5 Virtualização

Virtualização é a abstração de recursos computacionais, ou seja, permite a separação entre o sistema operativo e o *hardware* em que este corre. Virtualização tem a capacidade de disponibilizar a partir de um conjunto de recursos físicos, diversos conjuntos de recursos virtualizados para diferentes sistemas ou operações, permitindo que múltiplos servidores estejam centralizados na mesma máquina física.

2.5.1 Papel da virtualização nos sistemas de computação na *Cloud*

A virtualização de servidores é muito utilizada por empresas e organizações, porque permite que seja feito um investimento reduzido em recursos e infraestruturas e mesmo assim conseguir boas prestações. Cada servidor virtual comporta-se como uma máquina física com os seus recursos físicos e sistemas operativos próprios [19]. As vantagens de usar servidores virtualizados são as seguintes:

- Redução de custos de consumo de energia, refrigeração e manutenção.
- Menos área física necessária para os *data centers*, devido à redução do número de máquinas necessárias.
- Servidores virtuais permitem que o ambiente de testes seja o mesmo do ambiente de produção.
- Facilita a recuperação de dados.

Por outro lado, além destas vantagens apresentadas, a virtualização de servidores também apresenta as vantagens genéricas da virtualização tais como: isolamento, no sentido que diversas máquinas virtuais ainda que correndo na mesma máquina e partilhando recursos físicos, são independentes umas das outras, sendo que se houver algum problema com alguma das máquinas virtuais as outras continuam a correr sem qualquer problema, e também encapsulamento porque cada máquina possui todas as dependências e software necessários aos seus serviços, facilitando também a portabilidade do sistema para outra máquina física [19].

2.5.2 Arquiteturas de virtualização

Nos sistemas da *Cloud* são utilizadas duas arquiteturas de virtualização, uma delas baseada em máquinas virtuais, representada em (a) da figura 2.4, ou também designada hipervisionada, que é ideal para sistemas que necessitam de aplicações que na mesma *cloud* utilizem diferentes sistemas operativos. A segunda arquitetura, representada em (b) da figura 2.4, chamada de virtualização baseada em contentores, onde as diferentes aplicações num mesmo servidor partilham o sistema operativo e, se for oportuno, as dependências de bibliotecas [12].

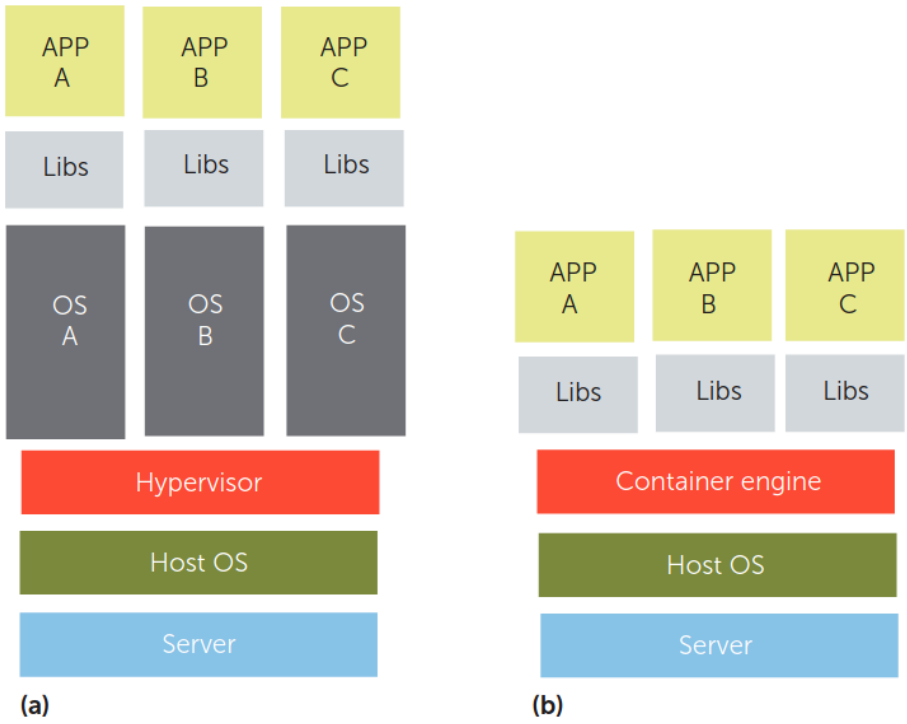

Figura 2.4: Arquiteturas de Virtualização utilizadas na *Cloud* [12].

Na arquitetura hipervisionada por haver uma grande abstração em relação ao *hardware* por utilizar *hardware* virtualizado, permite que seja fácil criar e manter uma imagem do sistema. Se houver algum problema com a máquina física, facilmente e com o mínimo esforço, essa imagem é colocada numa outra máquina física. Outra vantagem é que através da camada chamada *hypervisor* este tipo de arquitetura consegue tirar máximo partido do CPU da máquina física, permitindo um elevado desempenho. Por outro lado, a arquitetura baseada em contentores utiliza *hardware* da máquina física e como tal comunica diretamente com o *kernel* do sistema, o que implica que as aplicações que correm no container têm que ser compatíveis com o sistema operativo da máquina física e com a arquitetura do processador. As imagens dos contentores são normalmente muito mais pequenas que as imagens de máquinas virtuais porque não incluem todas as ferramentas e serviços de um sistema operativo inteiro [14].

2.6 Tecnologias

2.6.1 *Docker*

Docker é um projeto código aberto, que implementa a arquitetura de virtualização baseada em contentores vista em 2.5.2, e que providencia uma forma de automatizar o *deployment* de aplicações em ambiente *Linux* através de contentores. Cada contentor de docker é baseado numa das muitas imagens de *Linux* disponíveis, e é configurado utilizando um ficheiro chamado *Dockerfile* através de instruções, no qual é possível adicionar novas dependências ao contentor. Cada entrada no *Dockerfile* adiciona uma nova camada por cima da existente no contentor [12].

2.6.2 *Node.js*

Node.js é um ambiente de execução de *JavaScript* desenvolvido com o Chrome's V8 JavaScript engine. Apesar de não apresentar um modelo de concorrência permite desenvolver servidores altamente escaláveis através da sua arquitetura baseada em eventos, funcionando de forma assíncrona com uma *API* de *IO* não bloqueante. Com esta arquietura, esta *framework* permite lidar com milhares de eventos em simultâneo numa só máquina física. Esta plataforma além de ser muito leve, consegue valores de latência muito reduzidos e disponibiliza um dos maiores gerenciadores de pacotes de software, o *NPM*. Destes pacotes o *Express.js* é dos mais conhecidos porque facilita a utilização de *HTTP*. Estas caraterísticas tornam esta plataforma em uma das mais utilizadas em servidores na *Cloud* [7].

2.6.3 Comparação de Tecnlogias de Orquestração

2.6.3.1 *Docker Swarm*

É uma plataforma de orquestração de contentores de código aberto. A partir da versão 1.12 de *Docker* passou a ser nativo a esta plataforma. Tem a vantagem de que todo o software, serviços e ferramentas que podem ser utilizados em *Docker* também são compatíveis com esta plataforma. As máquinas, sejam elas físicas ou virtuais são organizadas num *swarm*, ou seja um conjunto de contentores fica a funcionar como apenas um *host*. *Swarn directors* são as máquinas principais num *swarm* que recebem as tarefas a realizar, também cabe a estes distribuir as tarefas pelos *workers* podendo utilizar diversas estratégias para esse efeito, por exemplo atribuir uma tarefa às máquinas com menos tarefas atribuídas ou tentando equilibrar a carga dos nós.

2.6.3.2 *Apache Mesos*

Mesos é uma plataforma de código aberto que, ao contrário das duas soluções apresentadas anteriormente, permite a utilização combinada de múltiplas plataformas de computação por *clusters*, permitindo a partilha de recursos computacionais entre elas [26]. Esta plataforma não possui

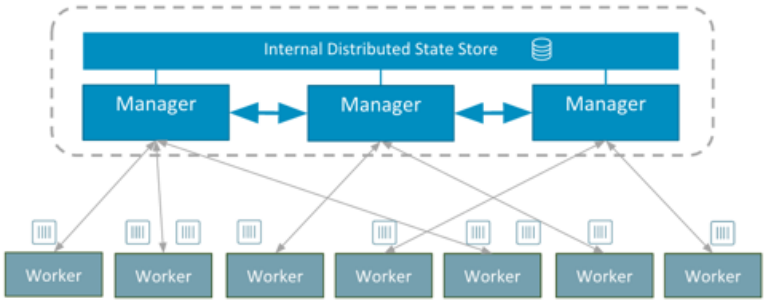

Figura 2.5: Arquitetura do Docker Swarm [23].

um sistema de escalonamento de recursos e tarefas semelhantes às outras, deixando esse escalonamento para as plataformas que integrem com esta mesma, escalonando apenas recursos computacionais entre as diversas plataformas em utilização e, por sua vez, cada plataforma decide quais recursos necessita daqueles que lhes são atribuídos pelo *scheduler* [18]. O *Mesos* é constituído por um processo principal chamado *Mesos master* que é responsável por gerir os *Slave Daemons*, que existem um por cada nó de computação quer seja ele físico ou virtual. Por sua vez estes são responsáveis pelas tarefas de cada framework usada. Esta plataforma pode ser usada juntamente com as outras duas plataformas aqui analisadas [6].

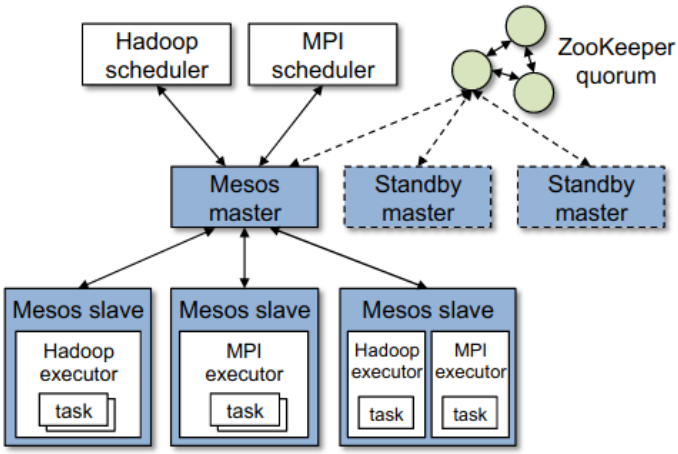

Figura 2.6: Arquitetura do Apache Mesos [18].

2.6.3.3 Kubernetes

Kubernetes é uma plataforma criada pela *Google* para orquestração de contentores, suportando outros tipos de contentores além de *Docker*. Ao contrário de *Docker Swarm* em 2.6.3.1, em que todos os contentores estão agrupados no mesmo grupo, nesta plataforma os contentores estão organizados por *pods*. *Pods* são conjuntos de contentores organizados pela tipologia de serviços que fornecem [30][23]. Os principais componentes de *Kubernetes* são:

- *Kubelet* - presente um por *worker* e é responsável pela comunicação com outros *workers* e o *master*.
- *Kube-proxy* - presente em todos os *workers* e é responsável pela *proxy*, conectando-se à *API* do servidor.
- *Workers* - constituídos por múltipos *pods*. Além destes também possuem *kubelets* e *kube-proxys*.
- *API* do Servidor - é a *API* que pode ser utizada pelos utilizadores para configuração e utilização do sistema.
- *Controller* - responsável por gerir e controlar os processos no *Kubernetes Master*.
- *Scheduler* - é responsável pela distribuíção de tarefas pelos *workers*.

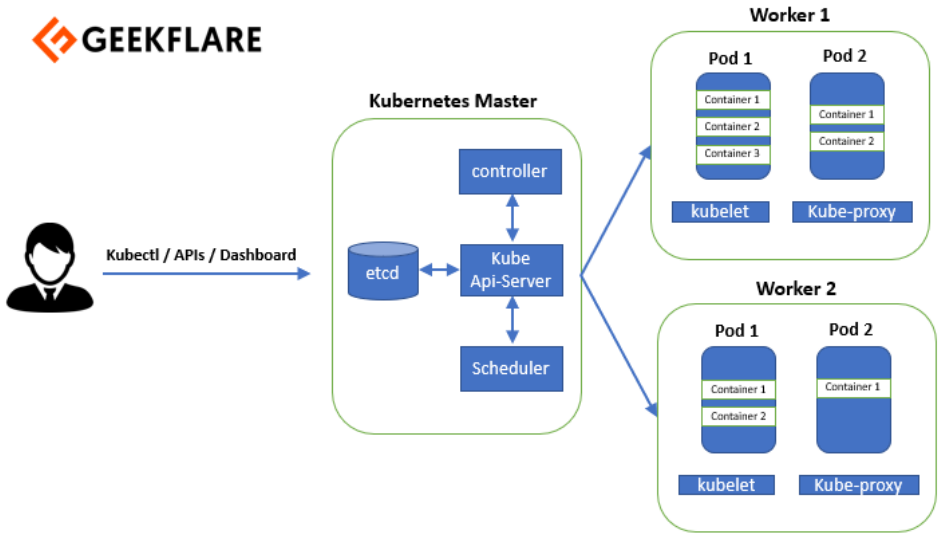

Figura 2.7: Arquitetura utilizada em *Kubernetes*[4].

2.6.3.4 Comparação entre as plataformas

Para comparar estas duas paltaformas de orquestração de contentores foram utilizados os seguintes critérios: escalabilidade, disponibilidade, setup de contentores, GUI, distribuição de tarefas pelos nós, volumes de dados, logs e monitorização. Para fazer esta comparação foram consideradas como referências as comparações feitas em [30] e [5].

Tabela 2.1: Comparação entre *Kubernetes*, *Docker Swarm* e *Apache Mesos*

Parâmetros	Kubernetes	Docker Swarm	Apache Mesos
Escalabilidade	Alta escalabilidade e auto-escalabilidade	Escala 5 vezes mais rápido que *Kubernetes* mas de forma manual	Alta escalabilidade em termos de *Mesos Slaves* mas dependente da escalabilidade de outras plataformas
Disponibilidade	O servidor é capaz de perceber quando um *pod* não responde e cria outro com a mesma configuração	Alta disponibilidade através de réplicas de nós	Replicação de nós e permite atualizações não disruptivas
Setup	Difícil configuração mas maior leque de opções	Fácil configuração mas opções mais limitadas	Requer a utilização de uma outra *framework* em simultâneo o que dificulta o *setup*
GUI	*Kubernetes Dashboard*	N/A	Cliente Web para visualizar o estado do sistema
Load Balancing	*Load Balancing* entre nós mas não entre containers do mesmo *pod*	*Load Balancing* automático entre contentores	N/A
Volumes	Apenas podem partilhar volumes entre containers dentro do mesmo *pod*	Pode partilhar volumes entre todos os contentores	N/A
Logs e Monitorização	Ferramentas integradas	Apenas com a instalação de ferramentas externas	Ferramentas integradas

2.7 Trabalho Relacionado

Existem já no mercado múltiplas plataformas de processamento de dados em tempo real em *IoT* inclusive algumas com aplicações práticas na área da saúde, nomeadamente no que toca à monitorização de pessoas em tempo real. Em [15] é feita a análise de catorze soluções de plataformas de monitorização de dados em *IoT*, em que apenas algumas delas contam com motor de processamento de dados.

Capability/ Features	IBM Mote Runner	SensorCloud	Portus	TempoIQ	FreshTemp	SensaTrack	Bluwired S-Cloud	Xively	Nimbits	ThingSpeak	MS Intelligent Service	Paho	SicsthSense	Relayr
API/Library	SDK for implementations	OpenData API, REST API	No	REST API, client API	No	No	No	RESTful API, client libraries	REST API, libraries	Open API and libraries	No	Open API, client libraries	RESTful and machine to machine API	RESTful API, SDK for Android and iOS
Data Formats	N/A	XDR, CSV	JSON, XML, IDoc	JSON	N/A	N/A	N/A	JSON, XML, CSV	Textual, JSON, XML	JSON, XML, CSV	N/A	N/A	JSON	JSON
Programming Language	Java, C#	Python, Java, C#, C++	C, C++, Java, PHP	Java, .Net, Ruby, Python	N/A	N/A	N/A	Objective C, C, Java, JavaScript, Ruby	Java, JavaScript	Java, JavaScript, Python, Ruby, .Net, node.js	.Net	Python, Java, JavaScript, C, C++	JAVA	N/A
Open Source / Commercial	Commercial with open SDK	Commercial solution	Commercial Solution	Commercial with open source client	Commercial Solution	Commercial Solution	Commercial Solution	Commercial with Open API libraries	Open source solution	Open source with hosted version	Commercial solution	Open source solution	Open source solution	Commercial solution
Visualisation	No	Yes and graphing tool	No, but with external web clients	Yes	Online dashboard	Web enabled	Visual management interface	Management console	Graphic user interface	Web capable devices	Yes	N/A	Yes, graphical interface	Yes, dashboard
Analytic Tool	No	MathEngine	No	Yes, for time series data	No	No	Blu Automation Studio	Yes	No	Yes	HD Insight, Power BI	No	Big data analytics	No
Messaging Protocol	N/A	No	RabbitMQ, JMS, TCP/IP, IBM MQ	No	No	No	No	MQTT	N/A	N/A	N/A	MQTT	N/A	MQTT
Notification / Alert	No	SMS and email	No	Yes	SMS, email, phone	Text message, email	Custom messages	Yes	Yes	Yes	N/A	N/A	Basic actuation	Custom messages
Energy Efficiency	Low power, harvest solar power	No	No	No	No	No	No	No	No	No	No	N/A	No	No
Connection Type	wireless	Http	Http	Http	Wireless	Wireless gateway, Zigbee, Wired	Wireless	N/A	Http	Http, wireless, Zigbee	N/A	Http,	Http, CoAP	Http
Platform Resource Requirement	Resource constrained devices, embedded controllers	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	Resource constrained devices, embedded platforms	N/A	Wunderbar sensors
Software Development Tool	Eclipse, Visual Studio, MonoDevelop	N/A	Eclipse	N/A	No	No	No	Developer workbench	Arduino, Java compatible tools	Arduino, Java compatible tools	N/A	Eclipse	Dropwizard	Postman console
Security	N/A	Https, SSL	SSL over http	SSL encryption	N/A	Secure data servers	N/A	Encryption with TLS and SSL	Keys, oAuth	Write API Keys	Enterprise-grade security	Authentication, SSL	DTLS	OAuth, SSL
Database Type	N/A	N/A	Relational databases	Relational databases	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	Relational databases	N/A
Storage Space	Yes, limited	Seemingly unlimited with triple redundant	Yes	Yes, with 3 times replications	Yes	Yes	Yes	Yes, time series archiving	Yes	Yes	Yes	Yes, limited	Yes, masterless replication	Yes, limited

Figura 2.8: Comparação de sistemas de monitorização em *IoT* feito em [15].

Com a análise destes sistemas, concluíram em [15] que apesar de existirem múltiplas soluções, ainda há problemas em geral que precisam de ser endereçados. As plataformas comerciais apesar de possuírem muitas funcionalidades são normalmente de código fechado, não disponibilizam *API* open-source e implementam tecnologias proprietárias ao invés de utilizar *standards*, problemas estes que são entraves à interoperabilidade e facilidade de adoção. Por outro lado as soluções de código aberto apesar de apresentarem boas bases para o propósito são soluções no geral básicas e com poucas funcionalidades. No entanto, esta análise incorpora soluções que não possuem motor de processamento de dados que é o foco desta dissertação. Então será feita agora uma análise detalhada de duas soluções de sistemas que incorporem motores de processamento de dados.

2.7.1 *ThingSpeak*

ThingSpeak é uma plataforma de análise e processamento de dados para *IoT* [3], é uma solução de código aberto que disponibliza também uma *API* de código aberto para desenvolvedores. No entanto, apesar de ser uma solução de código aberto apresenta apenas uma versão gratuita que limita a quantidade de dados que pode ser recebida por dia. Esta solução permite receber dados a processar via *HTTP* ou via *LAN*, e para processar os dados permite criar facilmente algoritmos em *MATLAB*. Possui um sistema de alertas e permite o agrupamento de dados por geolocalização.

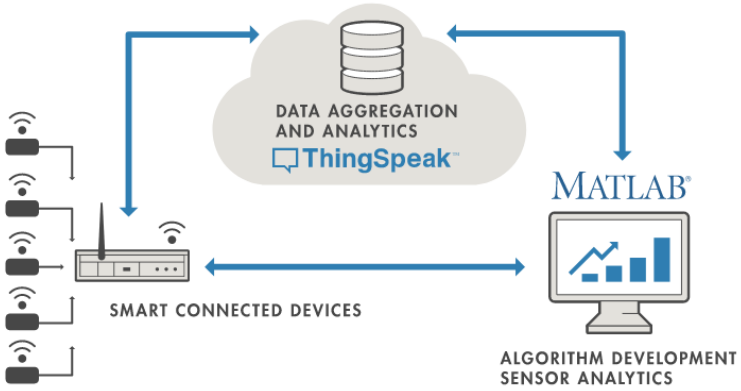

Figura 2.9: Infraestrutura *ThingSpeak* [3].

2.7.1.1 Aplicação em sistemas de monitorização na área da saúde

Em [21] é proposta uma solução de monitorização de doentes utilizando um *smartwatch* para medição do batimento cardíaco, ECG e PGG. Posteriormente os dados são enviados para um *smartphone* que os encapsula e via *HTTP* são enviados para o motor de análise do ThingSpeak. Neste motor os dados são analisados com algoritmos pré-carregados no sistema, e se detetar alguma anomalia nos dados é enviado um alerta via mensagem de texto para o telemóvel do doente para o alertar. Esta arquitetura está descrita na figura 2.10.

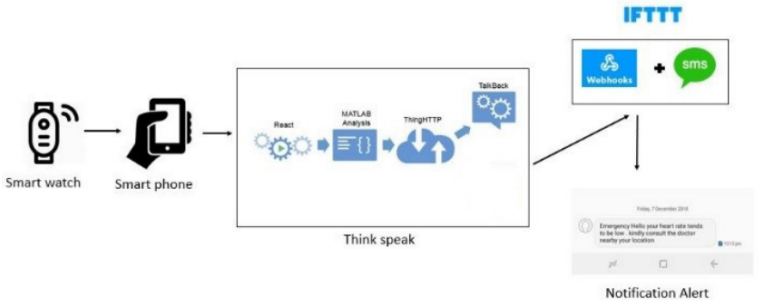

Figura 2.10: Arquitetura proposta em [21] utilizando o *ThingSpeak*.

2.7.2 *SicsthSense*

SicsthSense é uma plataforma de código aberto para a *Cloud* para reunir e partilhar com outros utilizadores dados recolhidos por dispositivos de *IoT* e *smartphones*. Permite que os dispositivos sejam adicionados automaticamente via *API* e que os dados recolhidos dos dispositivos sejam visualizados em tempo real assim como a tomada de decisões utilizando técnicas de análise de *Big Data*. Permite ainda a integração de alguns atuadores e definições de algumas regras simples [15].

2.7.2.1 Arquitetura

Esta plataforma tem por base um servidor *Java* capaz de correr em diversos dispositivos. Conta na sua arquitetura com uma *API* para comunicação *machine-to-machine*, para que os dispositivos possam de forma automatizada ligar-se à plataforma, mas também o permite fazer de forma mais agradável ao utilizador via cliente *web*. Para armazenamento de dados possui uma base de dados altamente escalável, assíncrona e com replicação dos dados [25].

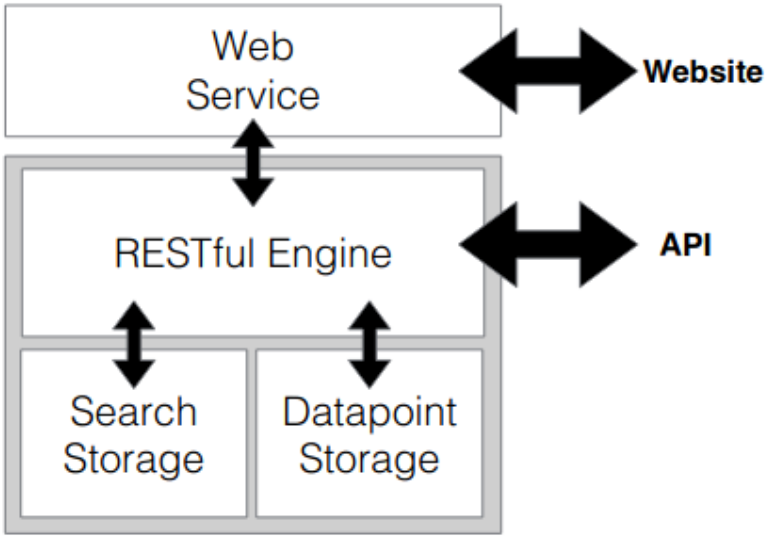

Figura 2.11: Arquitetura utilizada no *SicsthSense* [25].

2.7.3 Sumário

Apesar dos vários sistemas existentes apenas foram analisadas em detalhe soluções de código aberto e que possuem motor de processamento de dados. Existem soluções comerciais com mais capacidades do que estas soluções analisadas, mas por serem soluções proprietárias não são revelados pormenores sobre a sua arquitetura nem sobre detalhes de implementação.

Capítulo 3

Definição do Problema e Abordagem

Este capítulo começa por apresentar o problema de forma detalhada, depois em 3.2.1 fala sobre a abordagem ao problema no que toca à arquitetura do sistema. De seguida, é feita uma análise às tecnológias escolhidas e são fundamentadas as suas escolhas em 3.2.2. Por último, são apresentados os principais requisistos do sistema em 3.2.3.

3.1 Visão geral do problema

Os sistemas atuais de processamento de dados em *IoT* têm as suas limitações 2.7, por isso e com vista a abordar os problemas analisados, o sistema visa implementar *standards* em vez de tecnologias proprietárias e disponibilizar uma *API* para permitir interoperabilidade com outros sistemas de modo a que possam, de forma simples, processar os dados por eles recolhidos e obter o output desse processamento. Um outro problema que esta proposta visa endereçar é o facto das soluções existentes terem motores de *analytics* limitantes, quer seja pela limitação computacional do motor em si, quer pela limitação das linguagens para algoritmos disponíveis. Para endereçar este problema pretende-se que esta plataforma suporte todo o tipo de linguagens de programação e *scripting*. Para isso, pretendemos que o código fonte seja compilado utilizando compiladores nativos de cada linguagem (ou interpretados) existente para sistemas operativos *Linux*. Para permitir esta abordagem, o utilizador deve fornecer a imagem de *docker* em que o algoritmo/programa deve ser executado, com as dependências necessárias já instaladas nessa imagem.

3.2 Plataforma

O objetivo desta plataforma é permitir facilmente a integração dos algoritmos de análise já existentes, interoperar com a plataforma que agrega os dados do *Vital Responder*, a plataforma *WeSENSS*, mas também ser modular o suficiente para permitir a integração de novos algoritmos/aplicações e a integração com diferentes plataformas. O sistema deve funcionar de forma isolada e

modular de modo a que um algoritmo faltoso não ponha em causa o funcionamento do resto dos algoritmos nem do sistema em si. Permitindo assim que sejam processados e analisados em tempo real os dados recolhidos e agregados pelo *Vital Responder*.

3.2.1 Arquitetura

A arquitetura proposta para a plataforma segue a arquitetura proposta em [16] em que definem standards para sistemas deste tipo. A plataforma faz uso de um sistema distribuído de processamento, constituído por múltiplos nós utilizando contentores de *Docker*, mas neste caso cada nó é criado apenas para a execução do algoritmo e é destruído quando este retorna. Este tipo de arquitetura permite o isolamento de cada algoritmo que está a correr num ambiente isolado como visto em 2.5.2. Este isolamento é importante porque tendo uma configuração a executar com múltiplos algoritmos para múltiplos dados de entrada, a probabilidade de falha é alta, e como tal se cada algoritmo estiver a ser executado no seu contentor os outros contentores não serão afetados. Outra vantagem desta arquitetetura é que é simples recomeçar um contentor que por algum motivo deixou de responder, já que cada contentor garante o encapsulamento.

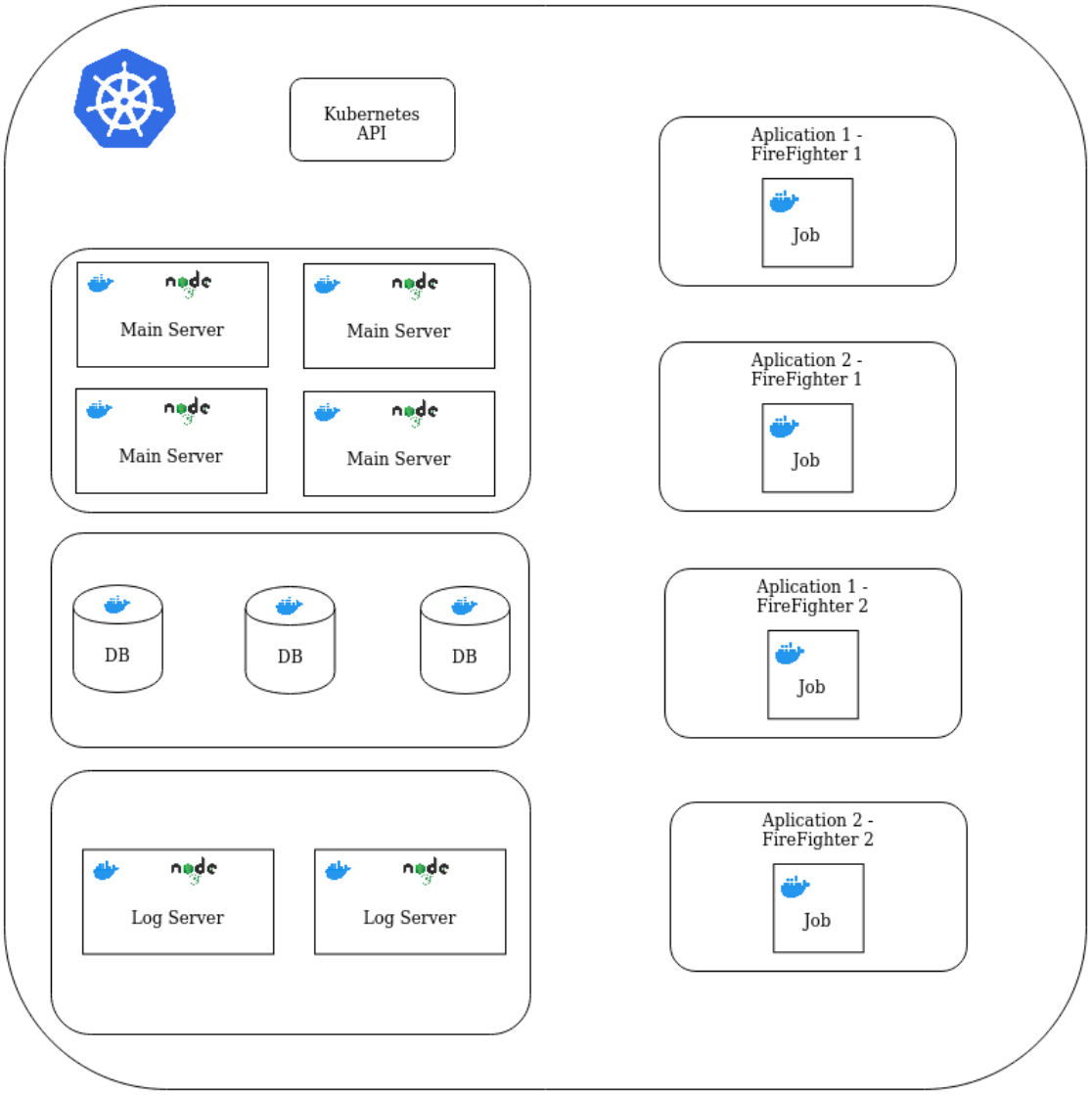

Figura 3.1: Arquitetura genérica proposta para a plataforma.

A arquitetura proposta é uma arquitetura modular orquestrada toda ela sobre um cluster de *Kubernetes*. O facto de toda a arquitetura ser assente sobre esta tecnologia permite que o servidor de configuração tenha um maior controlo sobre todos os outros contentores de processamento de dados e permite que todos os componentes possam tirar partido das vantagens já analisadas desta tecnologia 2.6.3.4. Neste caso o sistema é formado por componentes fixos, sendos eles o servidor principal ou de configuração, a base de dados que serve o sistema, o servidor de logs e os componentes mutáveis que são os algoritmos que tiverem a ser executados num dado momento. Esta arquitetura permite que todo o sistema escale vertical e horizontalmente em qualquer momento ainda que de forma manual.

3.2.2 Tecnologias

Nesta secção vão ser abordadas quais as tecnologias escolhidas para o sistema e quais os motivos para a escolha das mesmas. A tecnlogia utilizada para o servidor e construção da *API* e principais pacotes utilizados juntamente com esta tecnologia são tratados em 3.2.2.1, a tecnologia escolhida para orquestração do sistema em 3.2.2.2 e a tecnologia utilizada para a base de dados que serve o sistema em 3.2.2.4.

3.2.2.1 Tecnologia do servidor de configuração

A tecnologia para o *backend* do servidor teve que ser escolhida tendo em conta que além do desenvolvimento de uma *API*, também era necessário desenvolver um motor para a lógica do servidor baseado no tempo, uma vez que o servidor tem que se basear em unidades de tempo para as diferentes etapas de execução de uma aplicação, desde as tentativas de obtenção dos dados para *input* até ao resultado do processamento ser enviado para um servidor externo. A tecnologia escolhida foi *Node.js* 2.6.2 pelos motivos descritos abaixo.

- **Tempo de Aprendizagem**: Tendo em consideração que a tecnologia escolhida para a tecnologia de orquestração 3.2.2.2 tem uma curva de aprendizagem relativamente alta, por ter sido utilizada em projetos anteriores não requer qualquer tempo para aprendizagem.

- **Performance**: O facto de *Node.js* 2.6.2 ser uma tecnologia que funciona de forma assíncrona (apesar de funcionar com apenas uma *thread*), revela-se muito eficaz a lidar com um número elevado de pedidos. Apesar de esta tecnologia se revelar pouco eficaz em tarefas mais pesadas e que exijam mais do processador, neste caso esta desvantagem nao é notabilizada, já que as tarefas pesadas são executadas em contentores isolados. Esta tecnologia é muito utilizada para o desenvolvimento de sistemas deste tipo, o que reduz também o tempo de desenvolvimento porque muitos dos problemas encontrados são comuns a muitas soluções e foram já ultrapassados.

- **Pacotes Existentes e Gestor de Pacotes**: Pelo facto de *Node.js* 2.6.2 ser amplamente usado para o desenvolvimento de servidores *Web* leva a que existam muitos pacotes já existentes e licensa aberta que facilitam em muito o uso desta tecnologia. Estes pacotes/bibliotecas são facilmente instalados devido à existência do *Node Package Manager* (NPM), que é um gerenciador de pacotes que vem instalado de raiz com esta tecnologia.

- **Compatibilidade**: Apesar de hoje em dia grande parte das tecnologias de desenvolvimento de servidores *web* serem compatíveis com grande parte dos servidores, esta também é uma das vantagens desta tecnologia permitindo que o servidor possa correr em todos os principais sistemas operativos utilizados.

3.2.2.2 Tecnologia de orquestração do sistema

Para tomar uma decisão sobre que tecnologia utilizar para orquestrar o sistema foram a analisadas três das mais utilizadas tecnologias para este efeito 2.6.3.4. A tecnologia selecionada para este efeito foi *Kubernetes* pelas seguintes razões:

- **Facilidade de Utilização**: Quando comparado com o *Apache Mesos*, *Kubernetes* é mais fácil de utilizar e configurar visto que esta solução funciona sozinha sem necessitar de outras tecnologias, o que não se verifica com o *Mesos* que para funcionar necessita de uma das outras duas tecnologias analisadas ou de uma outra tecnologia de orquestração.

- **Suporte e Documentação**: Esta solução comparativamente com *Docker Swarm*, que está a cair em desuso, tem vindo a ser cada vez mais utilizada por grandes empresas e portanto o suporte tem vindo a crescer. Das três soluções analisadas, esta solução também é a que apresenta melhor documentação e mais completa, o que facilita a sua adoção.

- **API de configuração**: Entre as três soluções analisadas, *Kubernetes* é a única que permite configurar todo o sistema via *API*. Isto permite que um serviço que esteja a ser usado dentro desta tecnologia possa controlar o sistema programaticamente e em *run-time*, o que é um dos requisitos do sistema.

3.2.2.3 Pacotes Utilizados

Como foi explicado, para tirar partido de todas as vantagens de *Node.js*, foram instalados alguns pacotes via *NPM*. Todos os pacotes utilizados não têm quaisquer restrições de utilização e são de código aberto com licença *MIT* ou equivalente. Abaixo vão ser enumerados os principais pacotes utilizados e explicadas quais as suas funções/vantagens.

- **Express**: Este pacote facilita o desenvolvimento de uma *API* pois processa e reencaminha os pedidos *HTTP* recebidos e permite, de forma simples, a criação de *middlewares* utilizados por exemplo para autenticação.

- **Axios**: O intuito deste pacote é simplificar pedidos *HTTP* a servidores externos retornando uma promessa com o resultado do pedido, fazendo proveito do assincronismo da tecnologia escolhida.

- **Mysql2**: O pacote *Mysql2* abstrai a ligação à base de dados, funcionando como conector, o que facilita os pedidos à base de dados.

- **TypeScript**: *TypeScript* não pode ser considerado um pacote visto que se trata de um superconjunto de *JavaScript* que adiciona tipagem e algumas funcionalidades a esta linguagem [9]. Depois, os ficheiros desenvolvidos são convertidos para *JavaScript* puro. Foi utilizada esta ferramenta por facilitar o desenvolvimento no que toca a *debug* de código e pela redução do número de problemas, por introduzir tipos nas variáveis.

- **Moment**: Este pacote facilita a utilização e conversão de tempos e datas para *timestamps* e o inverso.

3.2.2.4 Tecnologia da base de dados

O sistema tem necessidade de uma base de dados persistente para guardar dados de configuração e resultados de processamento de dados como visto em 4.1.3. Uma vez que o modelo do sistema é um sistema relacional 4.2, apenas foram consideradas tecnologias de base de dados que implementem modelos deste tipo. Neste caso a tecnologia escolhida foi *MySql* pelos seguintes motivos:

- **Tempo de Aprendizagem** - Uma vez que o tempo para desenvolvimento não era muito extenso, foi escolhida uma tecnologia que não é nova e com a qual nao foi necessário perder tempo para aprendizagem.
- **Facilidade de Adoção** - A tecnologia para o servidor como visto em 3.2.2.1 foi *Node.js* e estas duas tecnologias são muitas vezes utilizadas em conjunto pois o *npm* possui bibliotecas que facilitam a conexão entre o servidor de *backend* e o servidor de base de dados.

3.2.2.5 Orquestração do Sistema

Para orquestrar o sistema de contentores, vai ser usado *Kubernetes*, que permite gerir a quantidade de processamento atribuído a cada nó e permite definir quantas réplicas são necessárias para cada nó. Se um dos nós deixar de responder de forma automática é criado outro com as mesmas especificações do que o que deixou de responder. Em *Kubernetes* o sistema está organizado em vários *pods*. Para esta plataforma, as diferentes linguagens de programação disponíveis estarão separadas em *pods* distintos, permitindo ter o compilador mais eficiente para cada uma delas sem tornar as imagens dos contentores muito pesadas.

3.2.3 Requisitos do Sistema

Para endereçar os problemas encontrados nos sistemas de processamento de dados em *IoT* existentes e para colmatar as necessidades existentes no sistema do *Vital Responder*, o sistema necessita de ser modular, escalável e versátil. Este tem não só que ser interoperável com a plataforma do *Vital Responder* mas também com outros sistemas, ainda que com regras definidas. Os algoritmos e aplicações a serem utilizados são muito distintos entre si pelas diferentes complexidades assim como pelas diferentes linguagens de programação utilizadas para os desenvolver. Como o desenvolvimento e integração de novas aplicações poderá ser feita em qualquer momento por diferentes investigadores, é necessário garantir que caso alguma das aplicações tenha algum problema o servidor e restantes aplicações não serão afetados. Uma vez que em situações reais de emergência existem muitos socorristas envolvidos e para cada um dos socorristas poderá ser necessário processar mais que uma métrica por operacional, é necessário que o sistema consiga ser escalável horizontalmente e verticalmente, de modo a atender as necessidades.

3.2.3.1 Requisitos

Os principais requisitos identificados para o sistema são os seguintes:

- Ser escalável, suportando múltiplas aplicações em simultâneo.
- Suportar múltiplas linguagens de programação para desenvolvimento dos algoritmos.
- Disponibilização de uma *API* de código aberto para permitir a configuração de todo o sistema.
- Permitir que os dados a processar provenham de diversas fontes, nomeadamente via pedidos *Http*.
- O resultado do processamento deve poder ser exportado para qualquer servidor via pedidos *Http*.
- Perceber quando um algoritmo não dá resposta e evitar que tome demasiado tempo do processador.
- Controlar e configurar todos os nós a partir do servidor principal.
- Interoperabilidade com o mínimo de configuração com a plataforma do *WeSENSS*.
- Processar situações de emergência inteiras em modo *offline* da plataforma *WeSENSS*.

Capítulo 4

VRAnalytics: A near real-time analytics engine for vitals and environmental monitoring of first responders

Neste capítulo vão ser abordados os detalhes do motor de análise *VRAnalytics* e detalhes da sua implementação. Este capítulo está dividido em quatro secções, na secção 4.2 é abordado o modelo de dados do sistema e explicada cada uma das principais classes do modelo, na secção 4.3 é explicado como está implementada ao detalhe a infraestutura do sistema utilizando *Kubernetes*. Na secção 4.1 são descritos ao pormenor todos os componentes do sistema, qual a sua função e explicado o seu funcionamento e por último na secção 4.4 é descrito o protocolo implementado para a criação de novas aplicações e de importação e exportação dos dados.

4.1 Arquitetura do Sistema

Nesta secção vai ser apresentada a arquitetura do sistema e vão ser descritos detalhadamente os componentes do sistema. No caso do Servidor de Configuração 4.1.1 por se tratar do principal componente do sistema e do componente com maior complexidade, é analisado em maior detalhe cada componente do mesmo e detalhes de implementação.

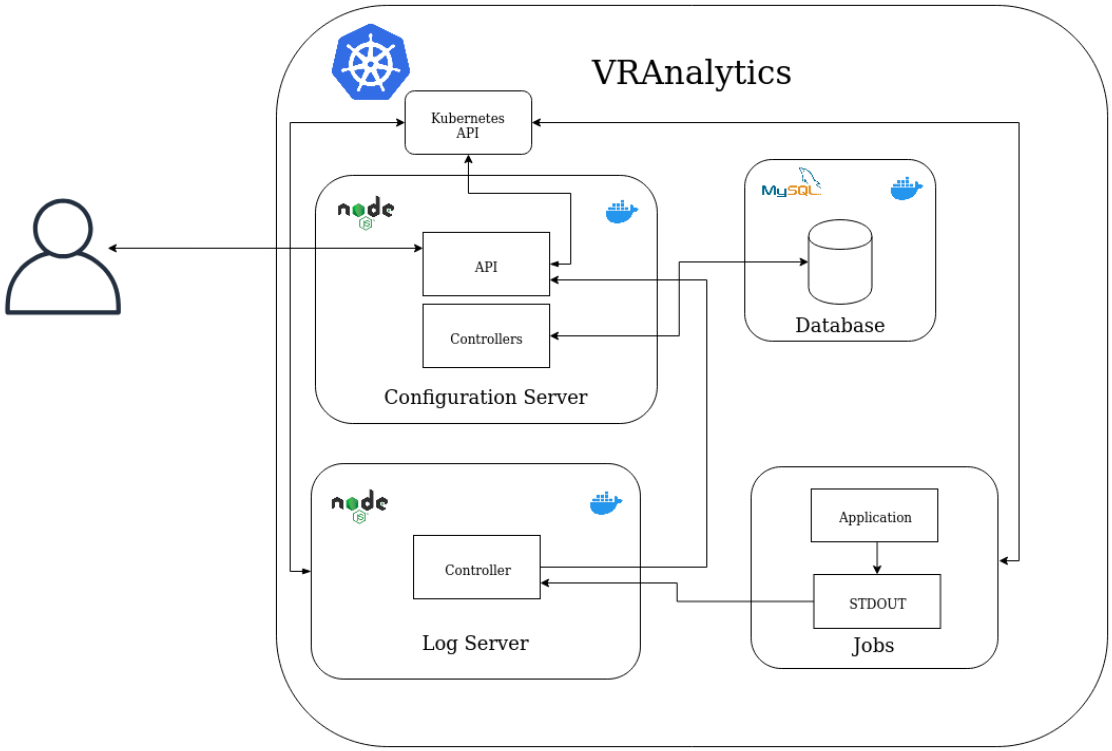

Figura 4.1: Arquitetura utilizada no VRAnalytics.

4.1.1 Servidor de Configuração

Este servidor é o principal componente do sistema, conectando todos os outros componentes. Está implementado em *NodeJS*, todo ele utilizando funções assíncronas para tirar o máximo partido desta tecnologia. O servidor tem múltiplas funções sendo elas as principais:

- Expor uma *HTTP REST API* para que possa ser acedida por outros servidores ou via *browser* para configuração do sistema e para obter informação acerca do estado do mesmo.
- Responsável pela comunicação com o *Kubernetes Master* para lançamento de novas aplicações de processamento de dados sobre a forma de *Jobs*.
- Faz o processamento dos *outputs* enviados via *API* pelo servidor de *logs* 4.1.2.
- Responsável pela obtenção e envio dos inputs e dos outputs respetivamente.

4.1.1.1 API

A *API* do servidor de configuração está desenhada utilizando a arquitetura *REST* e todas as suas especificações. Para facilitar esta implementação foi utilizada a *framework* de *NodeJs*, *ExpressJs*, que permite abstrair da complexidade adjacente a pedidos *HTTP* através de uma série de métodos e *middlewares*. Neste caso, a *API* foi dividida em dois módulos, separando os pedidos referentes às aplicações e os pedidos referentes à configuração das aplicações. Esta decisão foi

tomada para facilitar o desenvolvimento e para tornar o código mais modular. Nos próximos parágrafos serão descritos todos os pedidos de cada módulo e será explicada a estrutura do código de cada pedido.

- **Modulo da Aplicação**
  - **GET /user/:userId/applications** - Este *endpoint* permite obter todas as aplicações de um dado utilizador.
  - **GET /user/:userId/application/:applicationId** - Através deste *endpoint* é possível obter a aplicação de id de valor *applicationId*.
  - **GET /user/:userId/application/:applicationId/start** - Através deste *endpoint* é iniciar uma aplicação de id de valor *applicationId* ou terminar a sua execução se ela já estiver iniciada.
  - **POST /user/:userId/application** - Este *endpoint* permite criar uma nova aplicação.
  - **DELETE /user/:userId/application/:applicationId** - Através deste *endpoint* é possível eliminar a aplicação de id de valor *applicationId*.
  - **PUT /user/:userId/application/:applicationId** - Através deste *endpoint* é possível editar a aplicação de id de valor *applicationId*.
  - **GET /user/:userId/environments** - Este *endpoint* permite obter todas os *environments* de um dado utilizador.
  - **GET /user/:userId/environment/:environmentId** - Este *endpoint* permite obter todas os *environments* de um dado utilizador.
  - **POST /user/:userId/environment** - Este *endpoint* permite criar um novo *environment*.
  - **PUT /user/:userId/environment/:environmentId** - Através deste *endpoint* é possível editar o *environment* de id de valor *environmentId*.
  - **DELETE /user/:userId/environment/:environmentId** - Através deste *endpoint* é possível eliminar o *environment* de id de valor *environmentId*.
  - **GET /user/:userId/environment/:environmentId/env** - Através deste *endpoint* é possível obter todas as variáveis de ambiente para o *environment* de id de valor *environmentId*.
  - **POST /user/:userId/environment/:environmentId/env** - Através deste *endpoint* é possível criar uma nova variável de ambiente para o *environment* de id de valor *environmentId*.
  - **GET /user/:userId/environment/:environmentId/podLabel** - Este *endpoint* permite obter todas as*pod labels* para o *environment* de id de valor *environmentId*.

  - **POST /user/:userId/environment/:environmentId/podLabel** - Através deste *endpoint* é possível criar uma nova *pod label* para o *environment* de id de valor *environmentId*.

- **Modulo da Configuração**
  - **GET /user/:userId/configurations** - Este *endpoint* permite obter todas as configurações de um dado utilizador.
  - **GET /user/:userId/configuration/:configurationId** - Através deste *endpoint* é possível obter a configuração de id de valor *configurationId*.
  - **PUT /user/:userId/configuration/:configurationId** - Com este *endpoint* é possível editar a configuração de id de valor *configurationId*.
  - **DELETE /user/:userId/configuration/:configurationId** - Com este *endpoint* é possível eliminar a configuração de id de valor *configurationId*.
  - **GET /user/:userId/application/:applicationId/configuration** - Por meio deste *endpoint* é possível obter a configuração de uma aplicação id de valor *applicationId*.
  - **POST /output/:jobName** - Por meio deste *endpoint* o servidor de *logs* 4.1.2 envia para o servidor de configuração o resultado da aplicação de processamento.

- **Exemplo de Endpoint** Como vemos na primeira linha de 4.1.1.1, este exemplo de *endpoint* faz parte do módulo da aplicação 4.1.1.1, sendo que em seguida temos definida a expressão regular a ser aceite por este pedido, aceitando um número como parâmetro. Neste caso, trata-se de um pedido *GET* e tal como todas os métodos neste servidor, é um método assíncrono. Na linha 10 são verificados os parâmetros necessários, sendo neste caso apenas o *userId*. O método do controlador da aplicação é invocado com os devidos parâmetros através de uma chamada assíncrona e posteriormente retornada uma resposta para o utilizador.

```
1  applicationModule.addRoute({
2
3      path: /^\/user\/\d+\/applications\/?$/,
4      method: 'get',
5      withAuthentication: false,
6      mountRoute: (api: Api) => {
7          api.app.get('/user/:userId/application/', async (req: FRequest, res:
               express.Response) => {
8
9              try {
10                 let result : RequestResponse = await checkParameters(['userId
                       '],req.params)
```

```
11                  await applicationController.getApplications(req.params.userId
                        ,result)
12                  res.json(result.get())
13              }
14
15              catch(err) {
16                  console.error(err)
17                  res.status(400)
18                  res.json(err)
19              }
20
21
22          })
23
24      }
25  })
```

Listing 4.1: Exemplo de endpoind da *API*.

4.1.1.2 Controladores

Foram desenvolvidos três controladores para o servidor de modo a facilitar o desenvolvimento e aumentar a modularidade da aplicação. Sendo dois deles correspondentes a cada um dos módulos da *API* descritos em 4.1.1.1 e um terceiro controlador chamado *kubernetesController*, que é responsável pela comunicação com o *Kubernetes Master* através da *Kubernetes API*. Todos os controladores foram implementados em classes de *TypeScript* utilizando o padrão de desenvolvimento *Singleton*, para que outras classes que necessitem de invocar métodos do controlador o possam fazer sem ter que declarar um objeto da mesma e desta forma todas têm acesso à mesma instância do controlador.

- **Application Controller** - este controlador é responsável pelas operações CRUD referentes ao módulo da *API* 4.1.1.1. Todas as funções desta classe são assíncronas para tirar máximo partido da *framework* utilizada.

- **Configuration Controller** - este controlador é responsável pelas operações CRUD referentes ao módulo da *API* 4.1.1.1. Este controlador é também responsável por processar os *logs*, resultado do processamento dos *jobs* enviado pelo servidor de *logs* 4.1.2. Após processar os *logs*, o controlador extrai o *output* ou *outputs* e é responsável por enviar estes mesmos *outputs* para os *endpoints* de exportação definidos, com um máximo de três tentativas, guardando o *status* da resposta do pedido. Da mesma forma, também é responsável por fazer os pedidos *HTTP* para requerer os *inputs* necessários para a execução da aplicação, também com um máximo de três tentativas e guardando o *status* da resposta. Quando todos os *inputs* estiverem prontos é responsável por retornar os mesmos, já no formato especificado pelo utilizador.

- **Kubernetes Controller** - este controlador é unicamente responsável pela comunicação com o *Kubernetes*, nomeadamente com a criação de *Jobs* com as especificações da aplicação configurada pelo utilizador.

4.1.1.3 Configuração

Para o bom funcionamento do sistema, nomeadamente do servidor de configuração, é necessária ser feita alguma configuração. Neste caso, para que o servidor se possa conectar à base de dados, esta configuração é feita utilizando variáveis de ambiente especificadas no ficheiro de configuração do *Deployment* 4.3.1.

- **DB_HOST** - refere-se ao endereço da base de dados. Neste caso o endereço é definido pelo nome do serviço *ClusterIp*, sendo este endereço posteriormente traduzido pelo *Kubernetes Master* por um outro endereço que nos é abstraído.
- **DB_NAME** - refere-se ao nome da base de dados dentro do contentor de *MySql*.
- **DB_PWD** - define qual a password de acesso à base de dados.
- **DB_USER** - define qual o nome do utilizador com permissões de *root* de acesso à base de dados.

Estas configurações são usadas na interface de acesso à base de dados para configurar a ligacão entre o servidor e a mesma.

```
1       DatabaseUtils.connectionProps = {
2     host: process.env.DB_HOST,
3     password: process.env.DB_PWD,
4     database: process.env.DB_NAME,
5     user: process.env.DB_USER,
6     charset: 'utf8'
7 };
```

Listing 4.2: Configurações utilizadas para acesso à base de dados.

Uma vez que o código apesar de ser desenvolvido em *TypeScript* foi transpilado para *JavaScript*, para o servidor iniciar basta correr o comando *npm start* definido no ficheiro de configuração *package.json* para executar o comando *"nodemon dist/index.js"*.

4.1.2 Servidor de *Logs*

O resultado do processamento de cada aplicação é enviado para o *STDOUT* do contentor onde foi executado, sendo que também é guardado em volumes em todos os contentores no caminho '/var/log/containers' com o formato "2020-04-23 01:01:05 3,56"em que o nome do ficheiro é correspondente ao nome do contentor. Para isso, foi necessário desenvolver um servidor, neste caso implementado em *Nodejs*, que esteja sempre a observar estes ficheiros com os *logs* dos contentores. Quando houver uma nova entrada, caso seja de um *Job* de processamento, é enviada via *API* por um pedido *HTTP* para o servidor de configuração que identifica de que aplicação se trata, processando o *log* e guardando informação sobre o momento em que o *log* foi emitido sobre a forma de *timestamp*. Depois disso, o *log* é processado e dividido nos *outputs* convenientes definidos pela *string* de configuração. Este servidor foi implementado usando a estrutura de *Kubernetes* chamada *DaemonSet* 4.3.4, pois neste caso como podem existir vários noś, o sistema não partilha volumes de ficheiros com os *logs* entre nós e assim é necessário ter para cada nó um servidor em execução a observar os ficheiros de *logs* e reencaminhar a informação para o servidor de configuração. Uma vez que é guardado no *log* o momento em que ele foi escrito, é possível saber o momento exato em que a aplicação terminou a sua execução e não introduzir tempo adicional nos dados a guardar, devido ao tempo que demora depois para o *log* ser enviado para o servidor de configuração e a ser processado.

4.1.3 Base de Dados

O sistema dispõe de uma base de dados persistente implementada em *MySql* utilizada pelo servidor de configuração para armazenar dados sobre as aplicações e configurações mas também dos resultado do processamento de dados. A base de dados foi implementada utilizando o esquema A.1 no apêndice. Neste caso, além das tabelas também foi criado um *trigger* que verifica se as aplicações estão prontas a ser executadas, analisando para cada aplicação se todos os inputs de uma dada ordem já estão prontos, ou seja, se já foram importados de servidores externos ou inseridos manualmente e se isso se verificar é colocada uma *flag* a *true*,intitulada de *configurationReady*, e assim o sistema sabe que pode iniciar a configuração em causa.

4.1.4 Cliente de Administração do Sistema

Para facilitar a administração de todo o sistema foi utilizada a interface *web* de *Kubernetes*. Esta interface requer apenas alguma configuração para funcionar, nomeadamente criar uma *service account* com as permissões necessárias para aceder aos recursos do *cluster* e lançar um serviço com a interface propriamente dita. Esta interface permite monitorizar de forma eficaz o funcionamento de todo o sistema, permitindo consultar, editar e apagar todos os recursos do sistema. Nas subsecções abaixo apresentadas foram identificadas algumas das principais funcionalidades da interface acompanhadas com a respetiva interface.

4.1.4.1 *Overview do Sistema*

Nesta secção da interface é possível ter uma vista geral sobre todo o sistema e perceber quantos recursos existem de cada tipo e perceber qual o estado desses recursos.

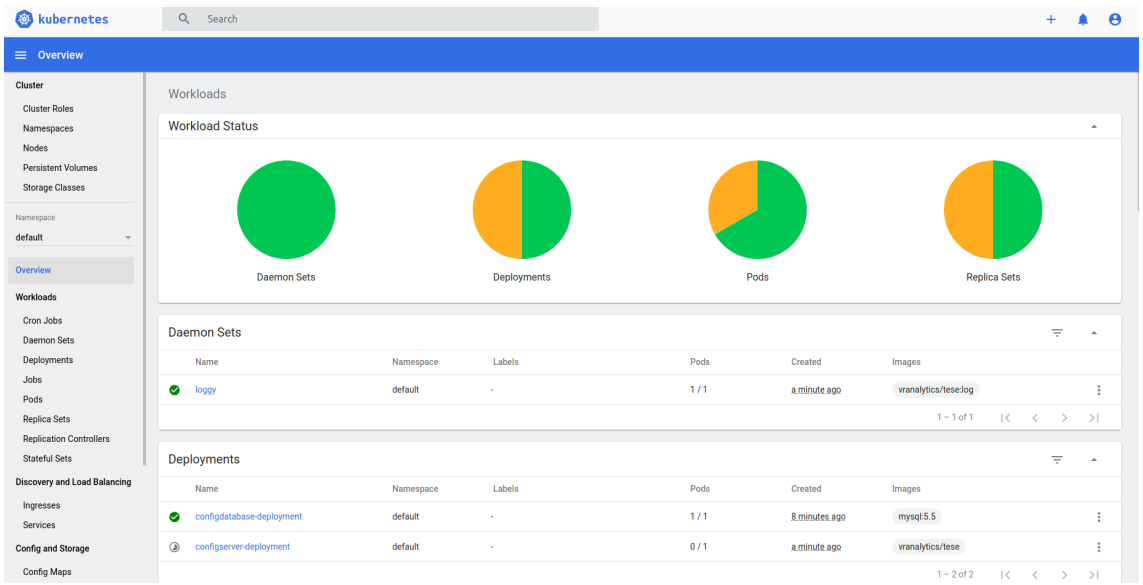

Figura 4.2: Overview do Sistema.

Nos gráficos circulares em 4.2 é possível visualizar a verde quantos de cada recurso já se encontram disponíveis e a amarelo quais dos recursos ainda estão a ser criados. Nas secções da interface imediatamente abaixo podemos ver listados por tipo todos os recursos 4.3.

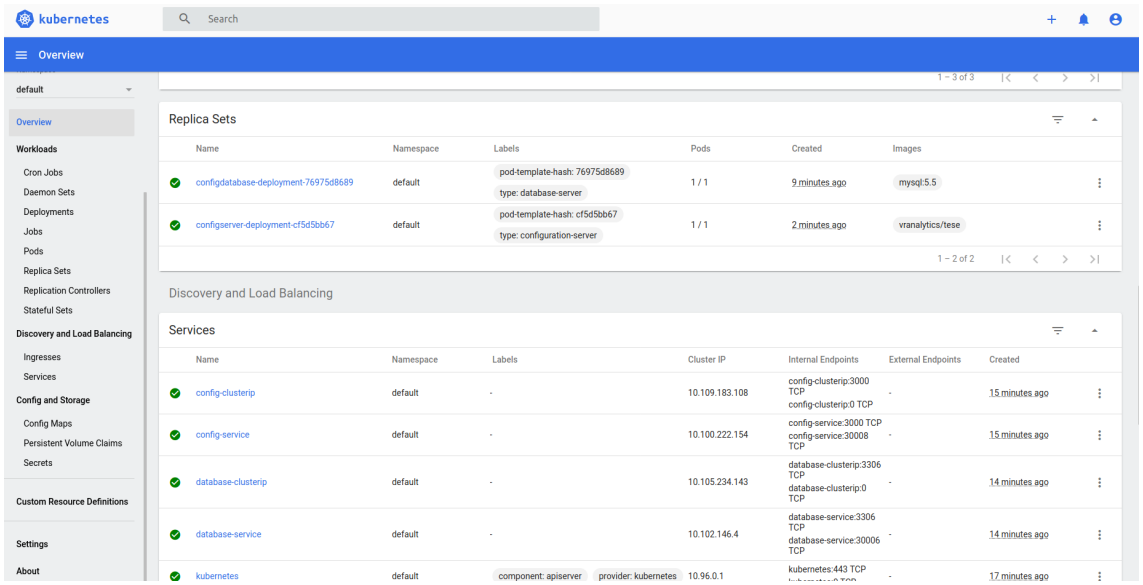

Figura 4.3: Recursos do Sistema.

4.1.4.2 Escalar Recursos

Através da interface mostrada em 4.4 vemos que é extremamente simples alterar a quantidade de réplicas que estão a ser executadas no *cluster* permitindo assim escalar horizontalmente o sistema. O processo inverso é de igual forma simples.

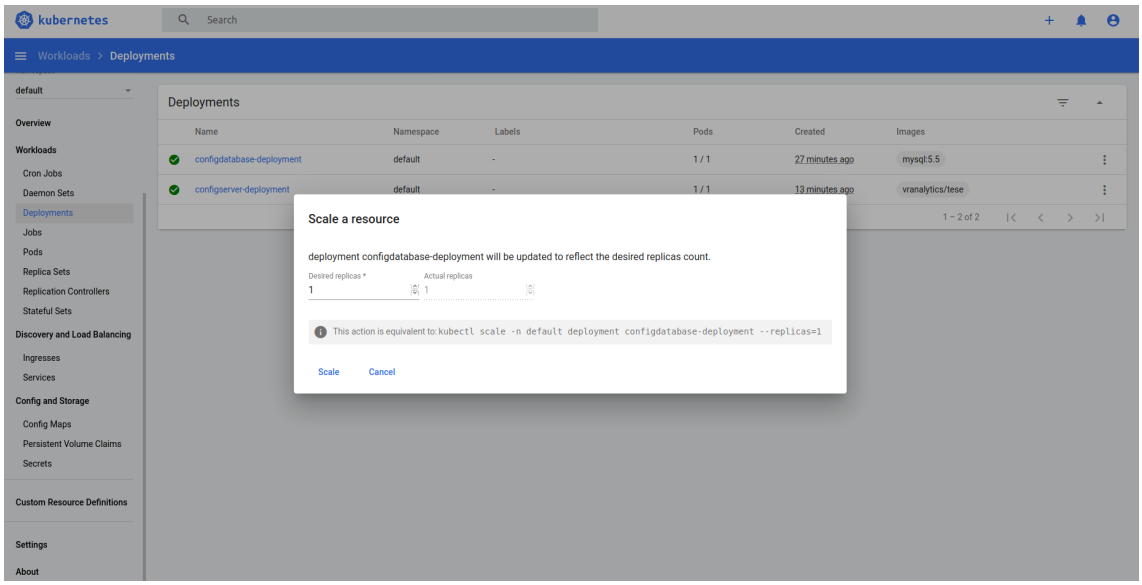

Figura 4.4: Escalar Recursos do Sistema.

4.1.4.3 Logs dos Recursos

Na interface 4.5 podemos ver todos os *logs* de um dado *pod* do sistema, neste caso do contentor da base de dados de *MySql*.

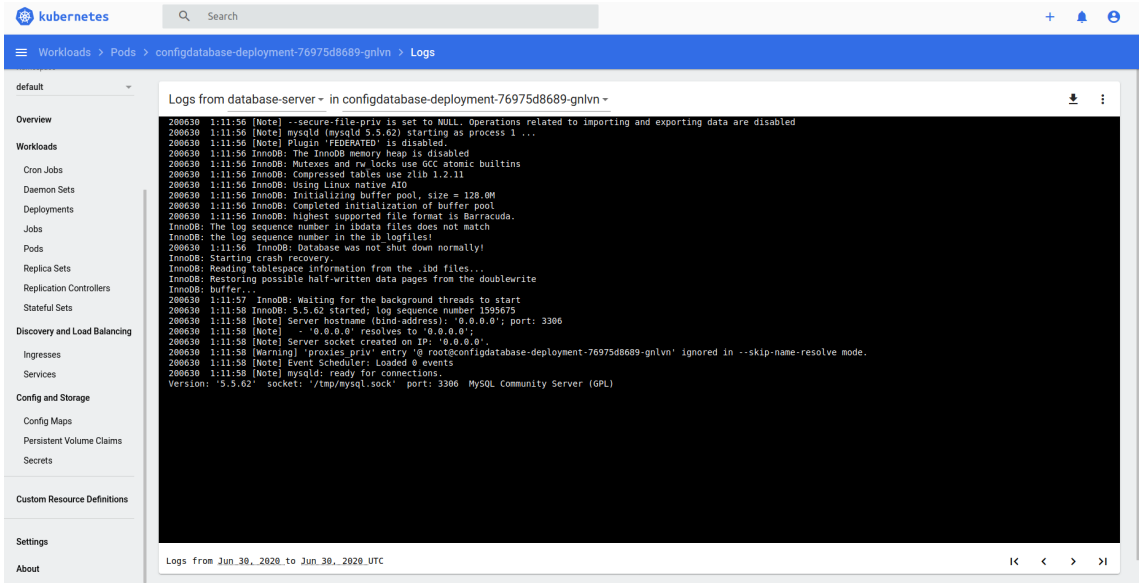

Figura 4.5: Logs do Sistema.

4.2 Modelo de dados

Na figura 4.6 está representado o modelo de dados do sistema, que neste caso se trata de um modelo relacional. Neste modelo de dados são armazenadas não só as configurações do sistema mas também o resultado do processamento dos dados. Apesar de o objetivo ser enviar os dados para um outro servidor para agregação e visualização dos mesmos, o facto de serem mantidos de forma persistente permite que cada vez que uma dada aplicação seja executada, seja possível manter um histórico dos dados de entrada e dos dados de saída. Outra razão é que os servidores para exportação de dados podem não estar disponíveis e assim os dados não são perdidos. Cada aplicação representa uma métrica que está a ser calculada através de um algoritmo ou aplicação num dado ambiente.

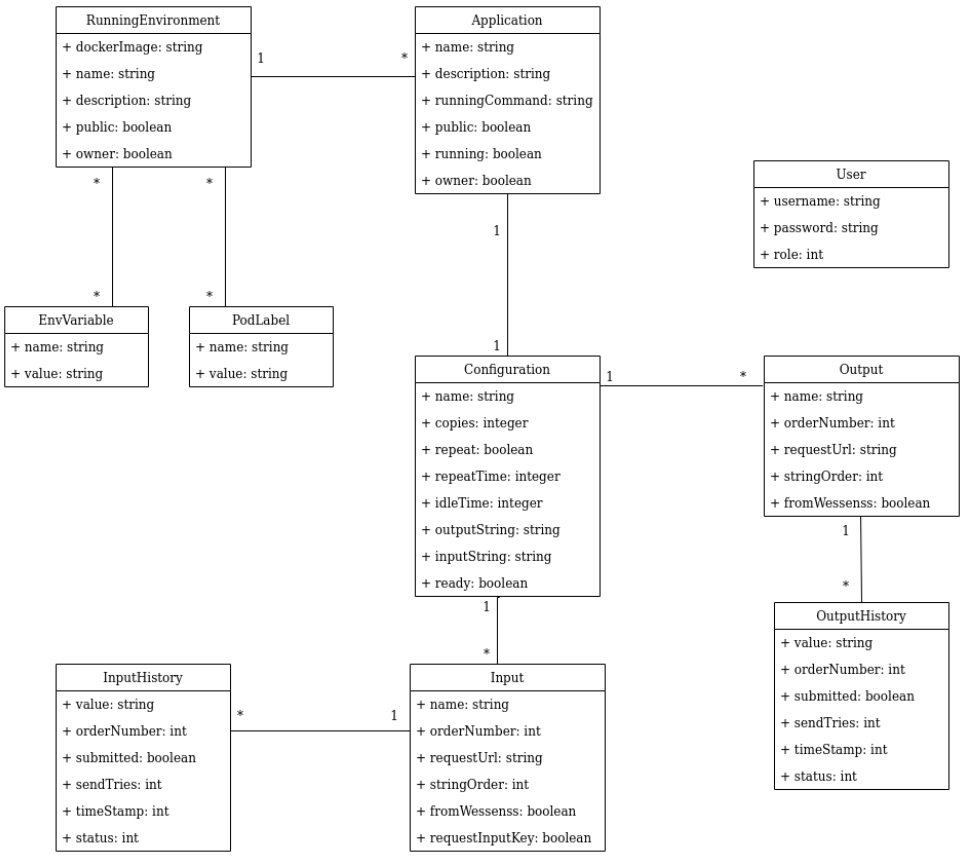

Figura 4.6: Modelo de Dados do *VRAnalytics*.

4.2.1 *Application*

Esta classe do modelo diz respeito à unidade principal de todo o sistema onde são guardados os dados gerais da aplicação e a associação para todas as outras classes do modelo de dados. Nesta classe são guardadas informações descritivas da aplicação, o utilizador que criou a aplicação (pois só ele tem permissões de edição sobre a mesma) e a indicação sobre se num dado momento a aplicação está a ser executada ou não.

4.2.2 *Running Environment*

A classe *Running Environment* diz respeito ao ambiente do container em que o algoritmo ou tarefa vai ser executado, contendo informações genéricas para diferenciar cada um dos ambientes e informação acerca da imagem de docker que deve ser utilizada quando for criado o *Job* para executar a tarefa.

4.2.2.1 *Pod Label* e *Environment Variable*

Ambas as classes do modelo de dados estão intimamente ligadas ao *Running Environment* 4.2.2, sendo que a classe *Environment Variable* diz respeito às variáveis de ambiente a serem criadas no ambiente de execução da tarefa e a classe *Pod Label* diz respeito à informação de mais alto nível em termos de infraestutura, sendo *labels* que ajudam na identificação do *Job* no cluster de *Kubernetes*.

4.2.3 *Configuration*

A classe *Configuration* especifica a configuração de uma dada aplicação e todas as informações necessárias para a execução da mesma, contendo informações sobre o formato do input e do output, permitindo que a aplicação possa obter diversos dados de diferentes servidores, processando o resultado e permitindo que a informação seja armazenada de forma separada. Contém informações também sobre se uma dada tarefa é para ser executada mais que uma vez e com que periodicidade e indica ainda se a configuração está pronta para ser executada.

4.2.4 *Output* e *Output History*

Na classe *Output* são guardadas informações que dizem respeito ao resultado da execução de uma dada tarefa ou algoritmo, em que cada aplicação pode ter um ou mais *outputs* dependendo do que foi configurado na classe 4.2.3. Nesta classe é especificado para cada output qual o *URL* para onde este será enviado quando terminar cada execução e é especificado para cada *output* se vai ser enviado para a *API* da *WeSENSS* (pois esta tem algumas peculiaridades que já estão especificadas no servidor) ou para qualquer outro servidor externo. Quanto à classe *Output History*, esta armazena informações que dizem respeito ao resultado do processamento propriamente dito, ou seja, o valor ou resultado obtido e o *timestamp* de quando o mesmo foi obtido, e informações que dizem respeito ao pedido ao servidor externo para exportar o mesmo. São guardadas informações quanto

ao resultado do pedido, o número de tentativas que foram feitas para exportar o resultado e qual o número de ordem do *output*.

4.2.5 *Input* e *Input History*

As informações armazenadas nas classe *Input* e *Input History* são análogas às informações guardadas em 4.2.4, mas estas dizem respeito aos valores dos argumentos necessários para a aplicação ser executada. Na classe *Input* são armazenadas as mesmas informações que na classe *Output* mas neste caso referentes aos valores de entrada, o *RequestURL* refere-se ao *URL* para o qual deve ser requerido o valor que vai servir de argumento para a aplicação. Na classe *Input History* ficam todos os dados referentes ao resultado do pedido ao servidor externo de forma análoga ao *Output History*.

4.3 Infraestutura do Sistema

Todos os componentes do sistema estão montados num cluster de *Kubernetes* para que o sistema seja escalável e tolerante a falhas. Todos os componentes executam em ambientes diferentes e isolados de forma a que se um componente falhar não interfira com o funcionamento de outros componentes e possa facilmente ser recomeçado. Neste subcapítulo é explicada de que forma está montada a infraestutura de todo o sistema, fazendo a associação entre os componentes do sistema e a infraestrutura.

4.3.1 *Deployments*

Os componentes do sistema, nomedamente o servidor de configuração e a base de dados, estão implementados como *Deployments* de *Kubernetes* permitindo que cada instância esteja num contentor isolado dentro de um *Pod* e que sejam criados múltiplos *Pods*. O facto de os componentes estarem implementados como *Deployments* permite com apenas um comando escalar horizontalmente os serviços para qualquer número de instâncias e o processo inverso também é da mesma forma simples. Outra vantagem que esta infraestrutura tem é auto-regeneração, ou seja, se houver um problema com alguma das instâncias dos serviços esta é apagada e recriada e assim que a instância estiver pronta, o controlador de *Kubernetes* começa a encaminhar o tráfego para a mesma. Para atualizações dos componentes, os *Deployments* permitem que seja feita uma atualização gradual e/ou parcial das instâncias do mesmo serviço para que este nunca fique indisponível. Caso a atualização tenha algum problema é facilmente revertida para a versão prévia.

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4     name: configserver-deployment
5 spec:
6     template:
```

```
7         metadata:
8             name: config-server
9             labels:
10                type: configuration-server
11        spec:
12            serviceAccount: config-server-serviceaccount
13            containers:
14            - name: config-server
15              image: vranalytics/tese
16              imagePullPolicy: OnFailure
17              env:
18                - name: DB_NAME
19                  value: config
20                - name: DB_PWD
21                  value: config
22                - name: DB_USER
23                  value: config
24                - name: DB_HOST
25                  value: database-service
26              ports:
27                - containerPort: 3000
28              command: ["npm", "start"]
29    replicas: 3
30    selector:
31        matchLabels:
32            type: configuration-server
```

Listing 4.3: Ficheiro de configuração do *Deployment* do servidor de configuração

Como podemos ver no ficheiro de configuração 4.3.1 do *deployment* do servidor de configuração do sistema, é especificada qual a imagem a ser utilizada e são definidas neste ficheiro as variáveis de ambiente a serem declaradas nos contentores dos *Pods*. É também definido qual o comando para iniciar o servidor e o porto em que este deve estar à escuta de pedidos. Neste caso é associada a *Service Account* 4.3.5 para que o *Deployment* do servidor de configuração tenha as permissões necessárias para utilizar a *API* de *Kubernetes* para criação de Jobs e obter informações sobre o estado do sistema. Nesta circunstância, estão especificadas três réplicas de modo a que quando o cluster é iniciado sejam criadas três instâncias de *Pods* com a imagem referente ao servidor de configuração.

4.3.2 *ClusterIp*

Para todos os componentes do sistema foi necessário definir um serviço *ClusterIp* para permitir criar um endereço *IP* virtual, que permite aceder a um porto dos componentes para o qual é definido, de modo a que os componentes estejam acessíveis entre si. Visto que existem várias instâncias de cada componente e que ao longo do tempo essas instâncias podem ser apagadas e recriadas, não seria possível saber qual o endereço *IP* de cada uma. Este serviço funciona também como *load balancer* sendo que reencaminha o tráfego de forma aleatória para uma das instâncias do componente. Qualquer componente do sistema dentro do *Cluster* para aceder a um outro componente pode fazê-lo, pelo nome do serviço definido no ficheiro de configuração do serviço.

4.3.3 *Node Port*

Este serviço, por sua vez, permite que os componentes do sistema sejam acedidos de fora do *Cluster*, como por exemplo através de um navegador de internet. Para isso, este serviço reencaminha o tráfego que chega a um porto especificada do cluster para uma das instâncias de um dado *Deployment* apontado por este serviço, funcionando tal como o elemento 4.3.2: como um *load balancer* que, de forma aleatória, reencaminha o tráfego para um dos contentores.

4.3.4 *Daemon Set*

Para o servidor de controlo de *logs* 4.1.2 foi um utilizada uma estrutura de *Kubernetes*, o *Daemon Set*, que funciona de forma semelhante a um *Deployment*, mas assegura-se que há uma e uma só instância do *Pod* especificado a correr em cada nó de computação do *Cluster*, pois em cada nó existe um serviço de *logs* isolado. Este serviço garante também que caso seja adicionado ou removido um novo nó de computação, automaticamente também seja criado ou removido nesse nó um *Pod* com a respetiva imagem.

4.3.5 *Service Account*

O servidor de configuração 4.1.1 necessita de comunicar com o *Kubernetes Master* utilizando a sua *API*, sendo para isso necessário definir todas as permissões que o componente tem em relação à *API*. Isto é feito utilizando uma *Service Account* e um *Role*, em que são definidas as permissões de utilização da *API* através da especificação de recursos a que se pode aceder e a que verbos referentes aos mesmo recursos.

```
1  apiVersion: v1
2  kind: ServiceAccount
3  metadata:
4      namespace: default
5      name: config-server-serviceaccount
6  ---
7
8  apiVersion: rbac.authorization.k8s.io/v1
9  kind: Role
10 metadata:
11   namespace: default
12   name: config-server-authorization
13 rules:
14 - apiGroups: [""]
15   resources: ["pods","services"]
16   verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]
17 - apiGroups: ["batch", "extensions"]
18   resources: ["jobs","cronjobs"]
19   verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]
20 ---
21
22 apiVersion: rbac.authorization.k8s.io/v1
23 kind: RoleBinding
24 metadata:
25   namespace: default
26   name: config-server-role
27 roleRef:
28   kind: Role
29   name: config-server-authorization
30   apiGroup: rbac.authorization.k8s.io
31 subjects:
32 - kind: ServiceAccount
33   name: config-server-serviceaccount
34   namespace: default
```

Listing 4.4: Ficheiro de configuração da *Service Account*

4.4 Protocolo de Integração de Aplicações

O sistema foi desenhado para estar integrado automaticamente com a plataforma que agrega os dados do *Vital Responder*, o *WeSENSS*, daí no modelo de dados nas tabelas *Output* e *Input* especificar se se trata de uma integração com esta plataforma ou não. Mas para facilitar a integração de outras plataformas e de novas aplicações, foi definido um conjunto de regras e standards que vão ser explicados nesta secção.

4.4.1 Aplicação de Processamento de dados

As aplicações de processamento de dados têm que ser aplicações que corram em containers de *Docker* ou seja têm que ser independentes entre si e têm que funcionar em ambientes *Linux*. Neste caso, como é de grande complexidade a criação de volumes de ficheiros persistentes em *Kubernetes*, as imagens das aplicações devem ser criadas com esta limitação em vista, ou seja, se for necessário um ficheiro com o código a correr este deve ser colocado à priori na imagem. A imagem deve ser publicada como pública para o *DockerHub*, após ser construída pelo utilizador utilizando um Dockerfile e serem copiados para a mesma os ficheiros necessários para a execução da aplicação/algoritmo. O comando para a execução do algoritmo deve ser um comando único da linha de comandos seguido imediatamente pelos *inputs* no formato especificado. Para o bom funcionamento do sistema é requirido que o único *log* que seja feito seja o retorno da aplicação.

```
1  apiVersion: stringbatch/v1
2  kind: Job
3  metadata:
4      name: test
5  spec:
6      template:
7          spec:
8              containers:
9                  - name: example-test
10                 image: vranalytics/tese
11                 env:
12                 - name: env_test
13                   value: test
14                 command: ["java start", "[1,2,3,4,5]"]
15             restartPolicy: OnFailure
```

Listing 4.5: Exemplo de ficheiro de configuração de Job.

Cada job é lançado com as informações da aplicação configurada. Na secção *metadata* podemos encontrar os *pod labels* que o utilizador especificou, no campo *image* temos o nome do repositório público de *DockerHub*, neste caso já com o executável java chamado *start*. Temos no *container* especificadas variáveis de ambiente que podem ser utilizadas no ambiente onde vai

correr o algoritmo/aplicação e, por fim, temos o comando de execução da aplicação no *container*, constituído pelo comando definido pelo utilizador, neste caso "java start", seguido pelo *input* já construído pelo *Configuration Controller*.

4.4.2 Inputs

No caso dos *Inputs*, caso sejam valores provenientes do *WeSENSS*, os pedidos e formato do pedido já está configurado por defeito, sendo apenas necessário de quanto em quanto tempo será para repetir o pedido e qual o endpoint em específico. No caso de um pedido para outro servidor, este tem que ser um pedido GET e é necessário que a resposta do servidor inclua um *status* entre 200 e 300 para que o servidor de configuração possa saber que o pedido foi bem sucedido e visto que a resposta pode ter vários campos, é também preciso especificar qual o nome do campo da resposta que é necessário guardar. É possível que uma aplicação necessite de mais dados dos que são obtidos de apenas um pedido a um servidor ou que a forma como os dados estão apresentados esteja alterada. Para isso, foi especificado um molde de configuração em que cada *input* é identificado por um caracter '?'.

```
1 input 1 = ? -> 3.45
2 input 2 = {value: ?} -> {value: 3.45}
3 input 3 = [?,?,?] -> [3.56 ,9 ,7.55]
```

Listing 4.6: Exemplos de *Inputs* permitidos

Vemos no exemplo acima três configurações de *inputs* diferentes que o protocolo suporta. No primeiro exemplo é o caso base em que apenas temos um *input* que é pedido a um dado endpoint e sem qualquer processamento, e é usado como argumento para a aplicação. No segundo exemplo temos também apenas um *input*, mas neste caso o valor obtido de um servidor externo tem que ser processado e é injetado no local onde se encontrava o caracter '?'. No último exemplo, a aplicação requer como parâmetro três inputs e o formato em que eles vão ser passados está especificado na *string* de configuração, sendo que a ordem pela qual aparecem na *string* é definida pelo número de ordem definido no modelo de dados do *input*.

4.4.3 Outputs

No caso dos *outputs*, estes funcionam de forma exatamente igual aos *inputs*, mas em vez de os valores serem injetados numa *string* de configuração, eles são removidos do local onde na *string* se encontra o caracter '?'. Também é possível especificar uma *key* para cada output para fazer par *key/value* no corpo do pedido POST para exportar cada um dos *outputs*. Como resposta do pedido de exportação, o servidor espera uma resposta com um *status* situado entre 200 e 300, indicando que o pedido foi bem sucedido e os dados exportados.

4.4.4 Outputs para o *WeSENSS*

Ao contrário dos *inputs* que se trata de um simples pedido *GET* tanto para qualquer sistema como para o caso do *WeSENSS*, no caso dos *outputs* estes são muito mais específicos. No caso da exportação de dados para este sistema os pedidos requerem a identificação do evento, a identificação do agente , a identificação do fator e do literal à qual os dados dizem respeito, o *timestamp* dos dados e os dados propriamente ditos. Para isso quando o utilizador especifica que a exportação de dados é para ser feita para o *WeSENSS* o *url* de configuração aceite deverá ser do tipo :

```
1 url|event_id|agent_id|factor_id|literal_id
```

Listing 4.7: Definição de *output* para o *WeSENSS*.

Depois disso o sistema processa a configuração e tudo que está depois do *url* são parâmetros para o corpo do pedido *POST*.

```
1 "http://vr2marketdb.inesctec.pt||506||15||62|58"
2
3 body:{
4     "event_id": 506,
5     "agent_id": 15,
6     "factor_id": 62,
7     "literal_id": 58,
8     "data": "146.6666"
9 }
```

Listing 4.8: Exemplo configuração de *output* para o *WeSENSS*.

4.5 Funcionamento do Sistema

Na última secção deste capítulo será analisado o funcionamento do sistema, vendo em detalhe cada fase do ciclo de execução do mesmo. Para cada fase será explicado o significado da mesma e o que dispoleta o sistema a transitar para a fase seguinte. Por se tratar num sistema para processamento de dados em tempo real, ou seja, que depende de tempo, toda a lógica do sistema foi implementada sobre um motor de tempo que é acionado de minuto a minuto e todas as tarefas cíclicas são realizadas nesse espaço de tempo.

4.5.1 Etapa 1: Aplicação em Execução

Para passar da etapa 1 a aplicação tem que estar em execução. Para isso, o utilizador pode alterar o estado da aplicação via *API*. Ainda que não exista nenhuma aplicação em execução, a cada minuto é verificado o estado de todas as aplicações.

4.5.2 Etapa 2: Aplicação em Pausa por tempo definido

É possível definir para cada aplicação de quanto em quanto tempo ela deve ser executada novamente. Este tempo é definido em minutos, pelo utilizador na configuração da mesma. Nesta etapa o tempo é reduzido a cada minuto até chegar ao minuto zero em que a aplicação pode passar para a etapa seguinte.

4.5.3 Etapa 3 e 4: Pedido dos Novos Inputs e Configuração Pronta

Assim que o tempo definido para a configuração termina, são feitos pedidos externos aos *endpoints* configurados para obtenção dos dados de entrada para as aplicações. A aplicação encontra-se neste estado até ao final de três tentativas sem conseguir obter sem sucesso um dos dados de entrada ou assim que a aplicação fique pronta. A aplicação é considerada pronta quando todos os dados de entrada tiverem sido obtidos com sucesso, estando este sucesso dependente do *status* da resposta do servidor a que foi pedido, em menos de três tentativas.

4.5.4 Etapa 5: Lançamento da Aplicação em *Kubernetes*

Nesta etapa, os dados de entrada são transformados no formato requerido pelo utilizador e é lançado um *Job* no cluster de *Kubernetes* com as especificações da aplicação. No final, a aplicação itera dando indicação de que não está pronta a ser executada e o processo repete-se voltando à etapa 2.

4.5.5 Etapa 6: Exportação dos Resultados Obtidos

Apesar de não ser de caráter obrigatório, o conceito do sistema assenta no princípio da exportação dos dado. Assim que a aplicação termina e retorna, o servidor de *logs* trata de enviar os dados para o servidor principal e os dados são processados conforme a configuração que tinha

sido definida pelo utilizador. Como as aplicações podem ter tempo de execução muito diferentes e esse tempo ser superior a um minuto, a etapa 6 ocorre em simultâneo com todas as outras. Todos os minutos o servidor verifica se existe algum resultado que ainda não tenha sido exportado, até um máximo de três tentativas.

4.6 Funcionamento em modo *offline* com importação de dados

A plataforma também possui um modo de funcionamento desenhado principalmente para a plataforma *WeSENSS* que permite, após um evento em que os dados não tenham sido processados, ou ainda que o tenham e seja necessário calcular outras métricas, o processamento dos mesmos. Para tal efeito, a plataforma permite tratar os dados requeridos de um *url* espeficico de uma só vez, como se de uma só execução de uma dada aplicação se tratasse ou permite dividir os dados no número de parcelas escolhido pelo utilizador e assim simular para o sistema uma simulação em tempo real. A única diferença é nao existir importação de dados de um servidor externo, porque os dados já se encontram armazenados na base de dados do *VRAnalytics*. Este modo também permite definir, se o utilizador desejar, que existe sobreposição de dados e com que percentagem, ou seja, se o utilizador quiser dividir um evento em dez partições com 20% de sobreposição, implica que na verdade serão doze partições porque 20% dos primeiros dados de cada partição são iguais aos últimos 20% da partição anterior.

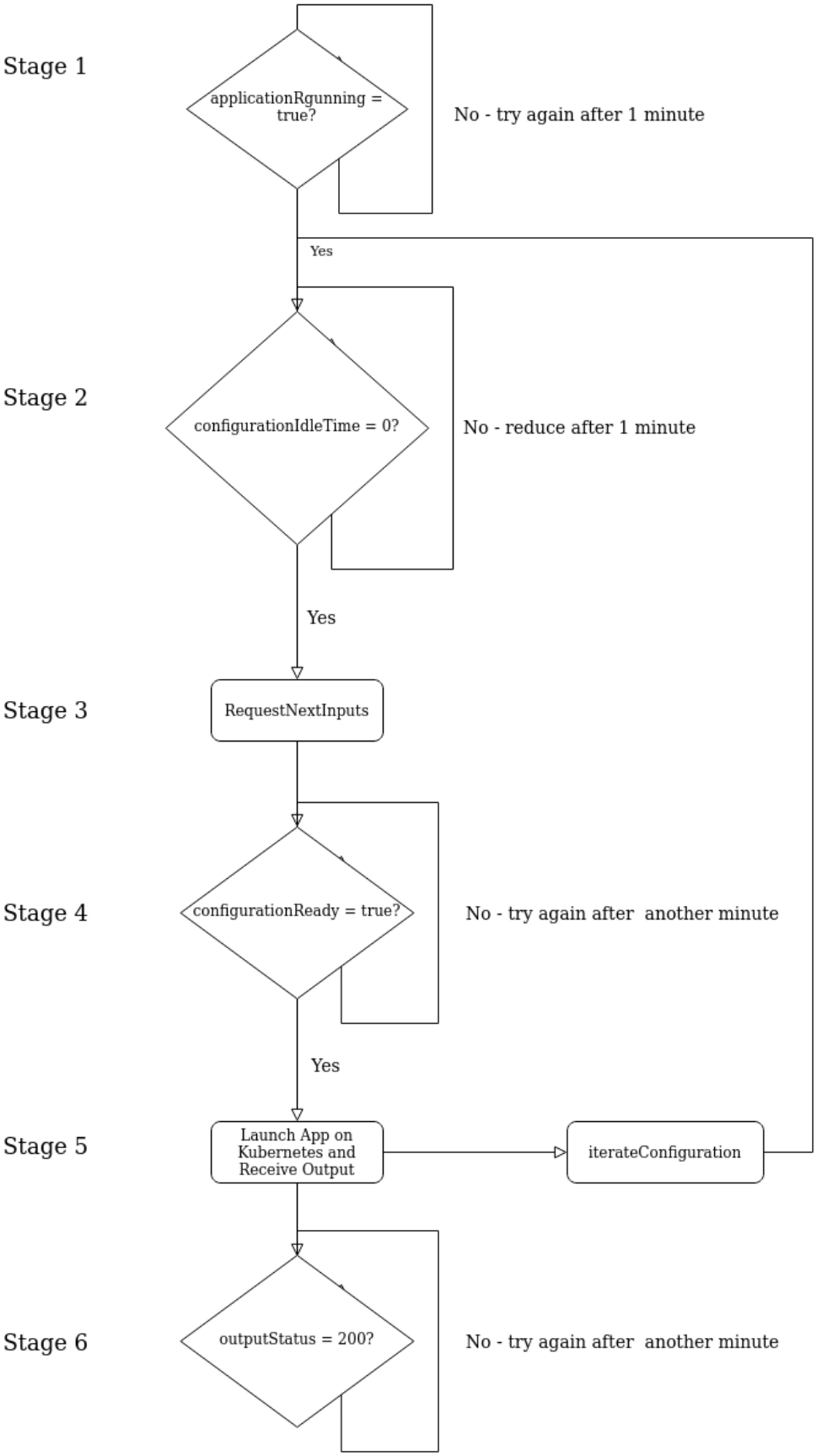

Figura 4.7: Diagrama de funcionamento do VRAnalytics.

Capítulo 5

Resultados

Neste capítulo vai ser descrito com maior detalhe o funcionamento do sistema, sendo analizado passo a passo o funcionamento deste. Vão ser enumerados e descritos os testes feitos à plataforma e quais os resultados, testando os diferentes requisitos do sistema, partindo de uma prova de conceito do sistema com a execução de uma simples aplicação em modo *offline* até uma situação em tempo real de análise de dados recolhidos em dispositivos reais.

5.1 Condições de Teste

Todos os testes e experiências realizadas foram feitas utilizando apenas um nó de computação de *kubernetes* usando *minikube*, que permite criar um *cluster* de *kubernetes* localmente. O *cluster* foi criado num computador com sistema operativo *ubuntu 18.04* uma vez que os contentores dependem do *kernel* de *Linux*. O computador utilizado para as experiências possui um processador de oito núcleos Intel Core i7-6700HQ CPU @ 2.60GHz e 32GB de memória *RAM*. Não foram impostas quaisquer limitações na percentagem de memória e processador que poderia ser utilizado pelo *cluster* ou por qualquer *pod*, sendo que essa gestão foi feita livremente pelo *kube master*. No início de cada experiência foram apagados todos os componentes do *cluster* e recriados através dos ficheiros de configuração, para que os teste existissem nas mesmas condições. Nos testes em que era necessário interagir com o servidor do *WeSENSS* foi utilizada uma ligação *VPN* para a rede interna do *INESC TEC*, uma vez que este servidor está montado num servidor interno.

5.2 Situações de Teste

Com os testes realizados pretende-se mostrar a efetividade do sistema e demonstrar que cumpre os requisitos propostos para o mesmo, descritos em 3.2.3. Para cada situação de teste vai ser descrita toda a sua configuração, qual o propósito da realização do teste e que requisitos são demonstrados com o teste em questão.

5.2.1 Teste 1 - Aplicação *Offline*

Nesta primeira situação pretende-se apenas demonstrar o funcionamento geral do sistema com uma aplicação simples que foi executada apenas uma vez, sem qualquer importação ou exportação de dados. Este primeiro teste serve apenas para demonstrar o funcionamento da infraestrutura e do modelo de dados, assim como a criação do contentor para execução da aplicação e a respetiva configuração dos dados de entrada e o processamento do resultado de saída. Para isso, foram utilizados cinco *inputs* que devem ser convertidos em um único formato de entrada e o resultado deve ser processado de forma a extrair apenas o valor que é necessário.

5.2.1.1 Aplicação de Teste Utilizada

Para este primeiro teste foi desenvolvido um pequeno algoritmo em *Python* que calcula a média dos valores fornecidos num *array*. O algoritmo utilizado encontra-se no anexo A.2. Este algoritmo segue o protocolo apresentado pois é executado com apenas um comando seguido dos dados de entrada já formatados segundo a *string* de configuração e o único dado que é enviado para o *STDOUT* é o resultado final.

5.2.1.2 Configuração utilizada

Neste primeiro teste foi utilizado a imagem de *Python* de *Docker* e foi copiado o ficheiro com o código para a imagem. A imagem foi publicada num repositório público de *DockerHub* para ser utilizada na criação do *Job*. Este teste consiste em uma única aplicação com a seguinte configuração:

Tabela 5.1: Configuração utilizada para o teste 1.

Configuração de Input	'?,?,?,?,?'
Configuração de Output	'Average of all entered numbers = ?'
Número de Inputs	5
Url de Importação de Dados	N/A
Url de Exportação de Dados	N/A
Tempo de Repetição	N/A
Número de Outputs	1
Comando de Execução da Aplicação	python ./alg/avg.py INPUT

5.2.1.3 Resultados Obtidos

Tal como era esperado um minuto após o começo do servidor, foi verificado que existia uma aplicação pronta para ser executada uma vez que não era necessário importar dados para *input*. Os cinco *inputs* foram configurados com os valores de 100, 150, 200, 175 e 130. O sistema configurou os dados consoante a configuração de *input* definida ficando, neste caso, como "100,150,200,175,130"e estes dados são agregados ao comando definido pelo utilizador e criado um *job* no *cluster* com as configurações definidas 5.1.

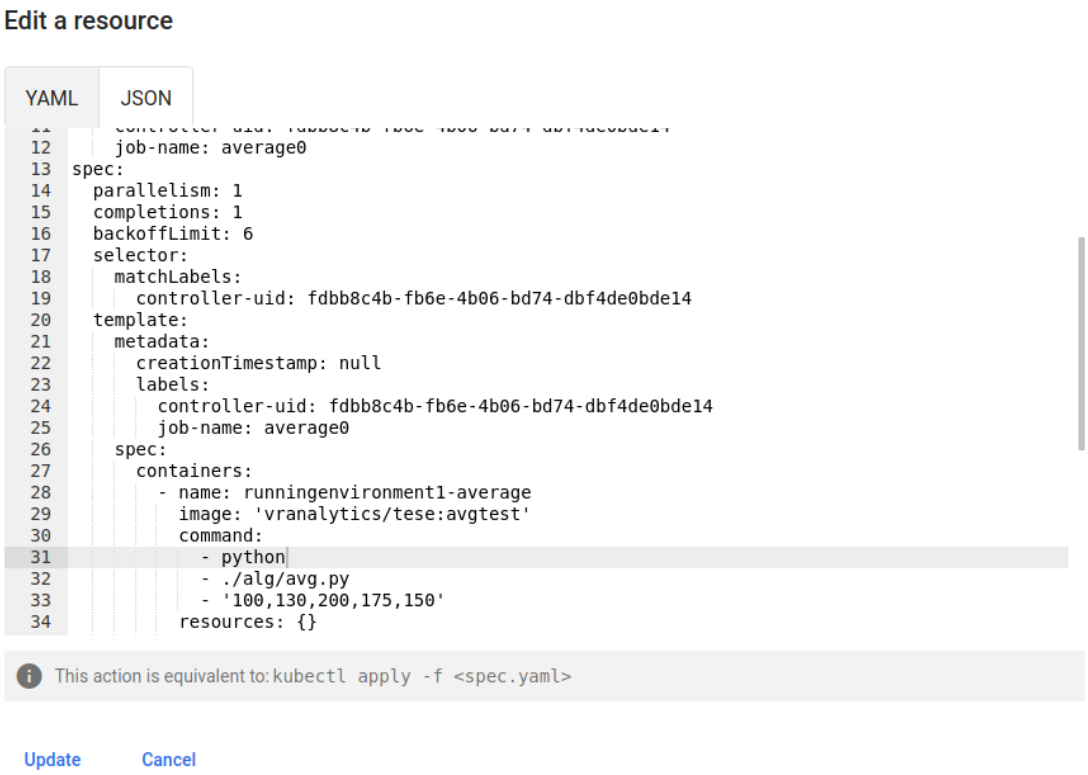

Figura 5.1: Configuração da Experiência 1.

O resultado da aplicação está no formato que foi configurado ’Average of all entered numbers = ?’ e com o resultado propriamente dito inserido no lugar do caracter ’?’. Para efeitos de persistência de dados, apenas o valor obtido "151"é armazenado 5.2.

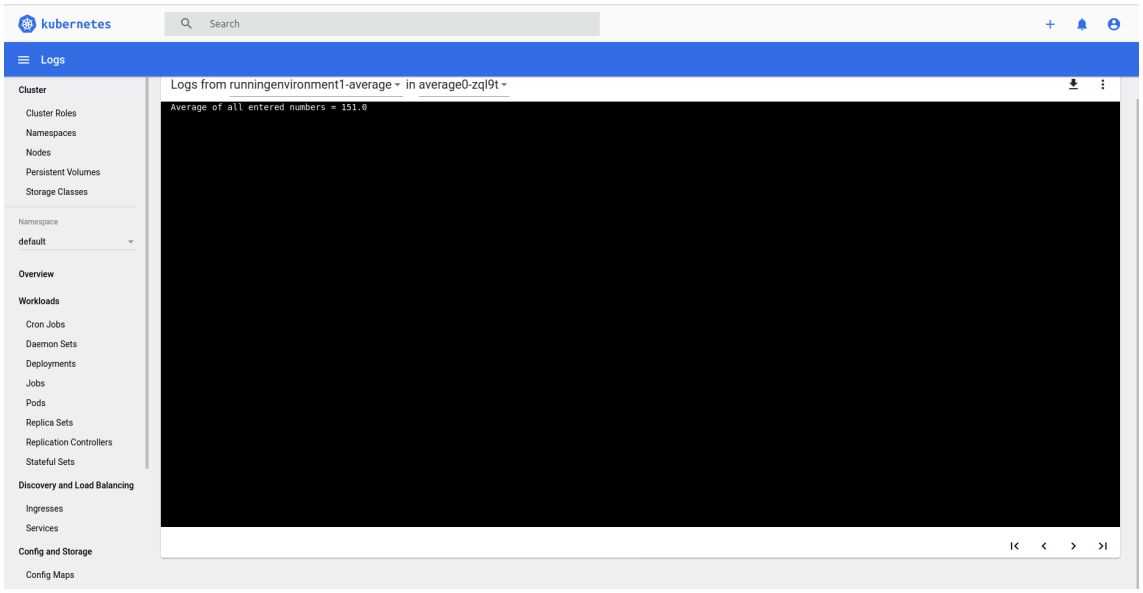

Figura 5.2: Logs da Experiência 1.

5.2.2 Teste 2 - Aplicação em Tempo Real

Nesta segunda experiência pretende-se demonstrar o sistema a funcionar no seu principal propósito, ou seja, numa situação em tempo real e interoperando com a plataforma do *WeSENSS* para importação e exportação de dados. Neste teste pretende-se provar que a plataforma consegue importar e exportar dados de um servidor, que o motor temporal funciona e que quando chega ao fim do período estabelecido pelo utilizador os dados são importados, o alogoritmo é executado, e os dados são exportados. Finalmente, neste teste também se pretende demonstrar a interoperabilidade com o sistema *WeSENSS* que tem certas peculiaridades que não funcionariam apenas com o protocolo implementado para outros sistemas. Para esta experiência foi utilizado um dispositivo móvel com um simulador que gera dados de batimento cardíaco aleatórios, e por conseguinte os valores obtidos pelo algoritmo poderão não fazer sentido por se tratarem de dados não reais.

5.2.2.1 Aplicação de Teste Utilizada

Para este teste foi usada uma aplicação real que já é atualmente utilizada com este sistema, ainda que em modo *offline*. Esta aplicação faz a análise do ritmo cardíaco e retorna diversas métricas que permitem analisar se existe algum problema com o operacional em termos cardíacos, além de medidas como o batimento cardíaco médio e batimento cardíaco máximo e mínimo. Retorna também medidas como:

- *mean_nni* - Corresponde à média dos intervalos *NN* que indicam o número de ciclos cardíacos observados por intervalo de tempo [27].
- *pnni_20* - Corresponde à percentagem do número de *NN* consecutivos que variam mais de 20ms entre eles. [27].
- *pnni_50* - Corresponde à percentagem do número de *NN* consecutivos que variam mais de 50ms entre eles. [27].

Este algoritmo foi desenvolvido em *Python*, utilizando duas bibliotecas externas e encontra-se em A.3.

5.2.2.2 Configuração utilizada

Neste teste, à semelhança do primeiro, foi criada uma imagem tendo como base a imagem de *Python* e no qual foram instalados os pacotes necessários à sua execução. A configuração foi feita de forma a existir 20% de sobreposição de dados, ou seja, à plataforma do *WeSENSS* foi feito o pedido de dados de oito em oito minutos dos dados que tinham sido recolhidos nos últimos dez minutos. Neste caso por a plataforma utilizada ser *WeSENSS* o sistema está preparado para fazer a agregação das cerca de duzentas entradas de ritmo cardíaco recebidas num pedido num só *input*. A exportação dos dados foi feita apenas para o *mean_nni* demonstrar que o resultado pode ser processado e extraído.

Tabela 5.2: Configuração utilizada para o teste 1.

Configuração de Input	'?'
Configuração de Output	"{'mean_nni': ?, 'sdnn': ?, 'sdsd': ?, 'nni_50': ?, 'pnni_50': ?, 'nni_20': ?, 'pnni_20': ?, 'rmssd': ?, 'median_nni': ?, 'range_nni': ?, 'cvsd': ?, 'cvnni': ?, 'mean_hr': ?, 'max_hr': ?, 'min_hr': ?, 'std_hr': ?}"
Número de Inputs	1
Url de Importação de Dados	http://vr2marketdb.inesctec.pt/api/v1/event_product/export/506?factor_id=10&agent_id=15&max_time=15
Url de Exportação de Dados	http://vr2marketdb.inesctec.pt/api/v1/event_product/data
Número de Outputs	16
Comando de Execução da Aplicação	python ./alg/hrv.py INPUT

5.2.2.3 Resultados Obtidos

Neste teste, e como era esperado, o sistema cumpriu o seu propósito e, de oito em oito minutos, foram requeridos os dados ao servidor do *WeSENSS* e foi iniciado um novo *job* com os dados requeridos e a aplicação configurada. Como podemos ver na imagem 5.3 apesar de só termos um *input* configurado, existem vários dados de entrada devido à agregação de dados que é feita. Quanto aos dados, podemos ver que existem *outliers* como os valores com valor nulo, que são dados erroneamente introduzidos pelo simulador, todavia o algoritmo já está preparado para eliminar estes *outliers*. Na imagem 5.4 podemos ver o estado dos sistema ao fim de três ciclos de execução. Temos três *jobs* que já terminaram a sua execução cada um correspondente a um ciclo de execução.

Podemos ver o resultado do processamento da primeira iteração do algoritmo que está de acordo com a *string* de configuração.

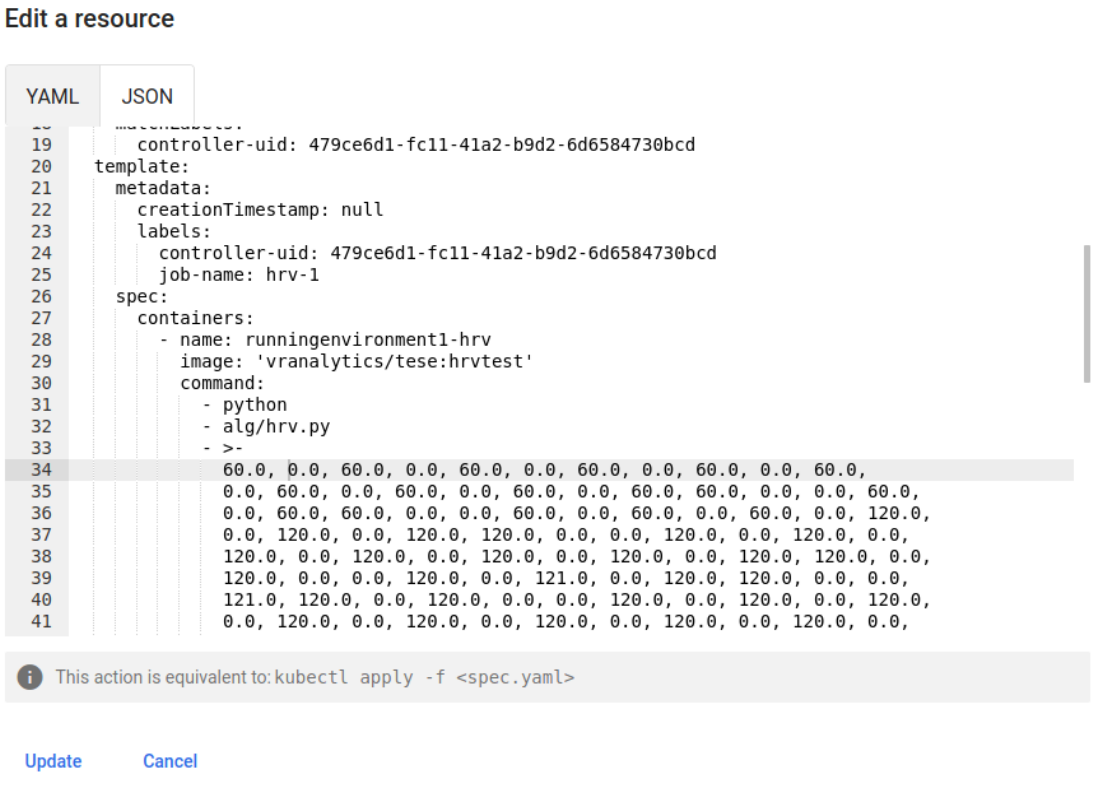

Figura 5.3: Inputs da Experiência 2.

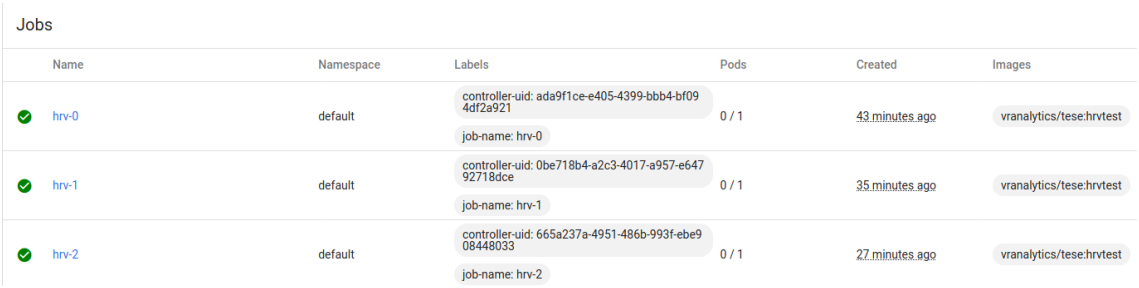

Figura 5.4: Jobs da Experiência 2.

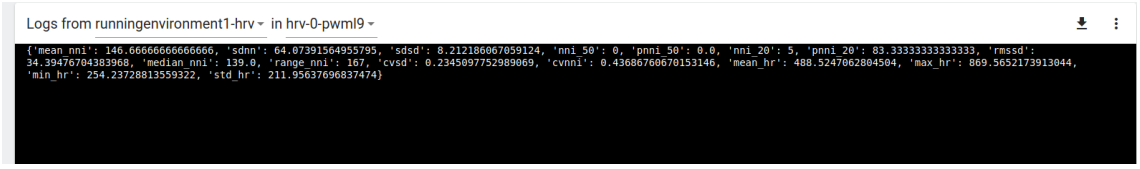

Figura 5.5: Logs da Experiência 2.

Com esta experiência demonstrou-se que o sistema é capaz de interagir com a platforma do *WeSENSS* em tempo real e processar os dados recebidos que posteriormente são devolvidos à plataforma. Também se comprovou que o sistema não apresenta dificuldade com algoritmos computacionalmente mais exigentes.

5.2.3 Teste 3 - Aplicação em Tempo Real com Teste de Escalabilidade

O objetivo deste teste é testar a escalabilidade do sistema, simulando uma situação real em que para todos os socorristas estaria a ser analisado o seu ritmo cardíaco. Os testes serão incrementalmente realizados com mais réplicas da mesma configuração nomeadamente: 2, 5, 10, 30 e 50 réplicas.

Para este teste será utilizada a mesma configuração de 5.2.2 para todas as réplicas de configuração mas com tempo de repetição reduzido para um quarto do tempo desta (dois minutos). Apesar de se tratarem dos mesmos dados é indiferente para o sistema, visto que é como se tratassem de dados diferentes e portanto simula uma situação real com o número definido de operacionais. Cada situação foi executada durante cinco ciclos. É de salientar que todos os testes foram realizados no mesmo *cluster* com apenas 2 *cores* e 4 *GB* de memória disponíveis. Para todas as experiências foi medido, ao longo do período de dez minutos, a percentagem de utilização da memória disponível em relação à memória disponível para o nó e a percentagem de tempo de utilização do processador em relação ao número total de *cores* existentes na máquina física, ou seja esta percentagem nunca poderá passar os 25% por serem utilizados apenas 2*cores*. Para monitorização do sistema foi utilizada uma ferramenta diretamente integrada no *Cluster*, o *Prometheus* [8].

5.2.3.1 Primeira Experiência - Controlo

Para controlo foi feita uma primeira experiência com apenas uma réplica. Podemos observar nos gráficos de controlo que os picos de utilização de memória e processador são relativamente no mesmo período de tempo e que estes picos se verificam sensivelmente de dois em dois minutos, que é o período estipulado para a aplicação ser executada e como tal, existe um ligeiro aumento de consumo de recursos, sendo que estes picos são mais visíveis no gráfico de utilização do processador 5.7. Podemos ver que a percentagem de memória utilizada já é por defeito superior a 20% devido aos outros recursos presentes no *cluster* necessários para o funcionamento do sistema.

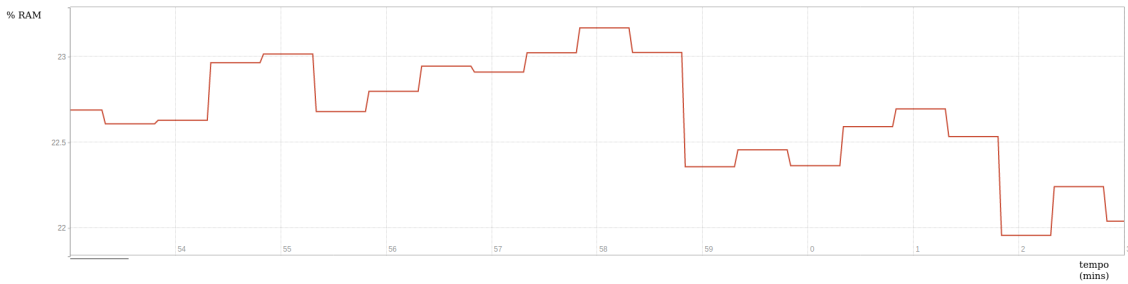

Figura 5.6: Gráfico da Percentagem de Memória Utilizada ao longo de dez minutos na primeira experiência.

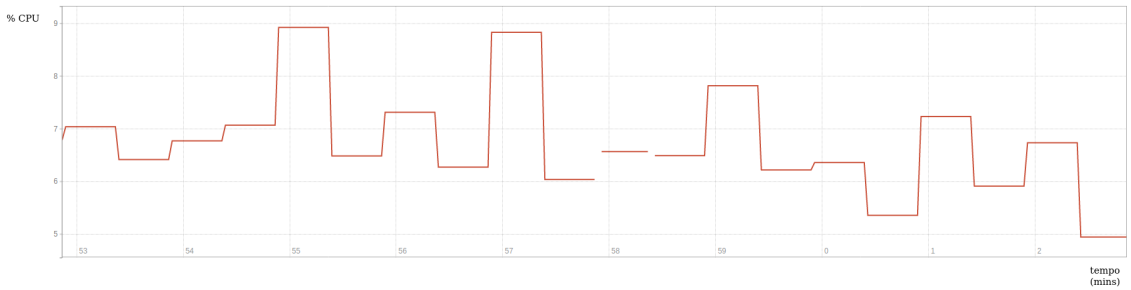

Figura 5.7: Gráfico da Percentagem de Tempo de Processador Utilizado ao longo de dez minutos na primeira experiência.

5.2.3.2 Segunda Experiência - Duas Réplicas

No caso da segunda experiência com duas réplicas, os resultados foram muito semelhantes aos da primeira só que neste caso os valores são ligeiramente mais elevados por serem necessários mais recursos computacionais na execução de duas aplicações. Nesta experiência é possível observar mais nitidamente os picos de utilização de recursos de dois em dois minutos.

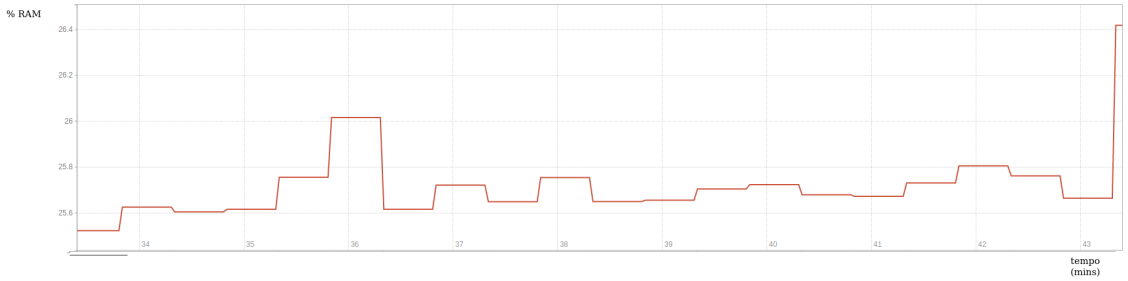

Figura 5.8: Gráfico da Percentagem de Memória Utilizada ao longo de dez minutos na segunda experiência.

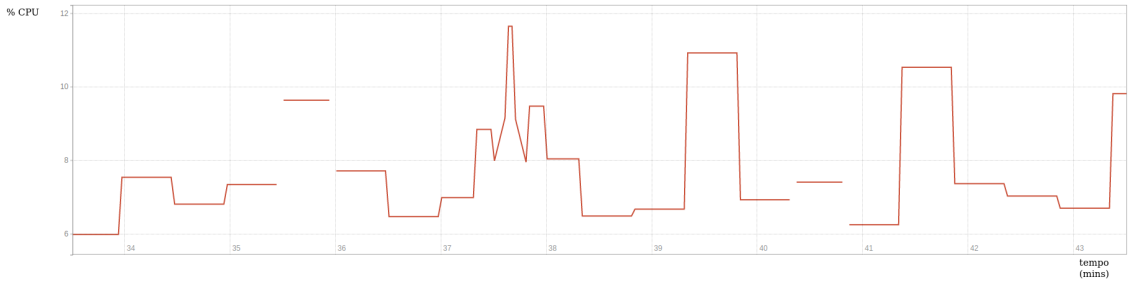

Figura 5.9: Gráfico da Percentagem de Tempo de Processador Utilizado ao longo de dez minutos na segunda experiência.

5.2.3.3 Terceira Experiência - Cinco Réplicas

No terceira experiência verifica-se uma tendência crescente já verificada na experiência anterior que é uma maior distinção dos picos de utilização nos intervalos de tempo definidos. Este

fenómeno explica-se pela criação dos *jobs* para processamento do batimento cardíaco ao fim de cada intervalo de tempo. Os picos vão ser cada vez mais claros devido ao crescente número de réplicas que exige mais recursos. Observa-se também que as percentagens utilizadas pelo sistema são identicamente superiores às utilizadas pelo sistema na experiência anterior.

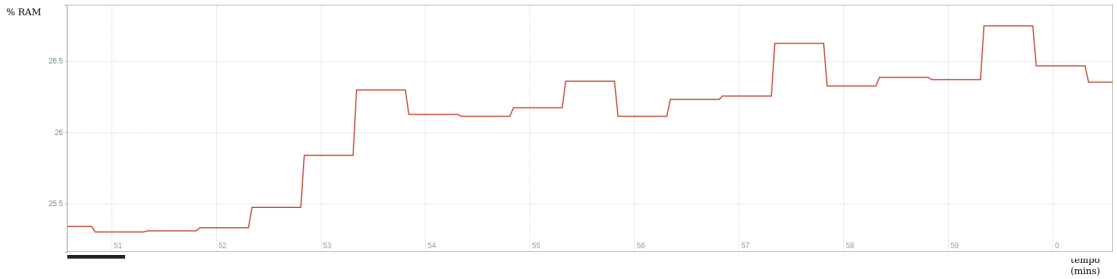

Figura 5.10: Gráfico da Percentagem de Memória Utilizada ao longo de dez minutos na terceira experiência.

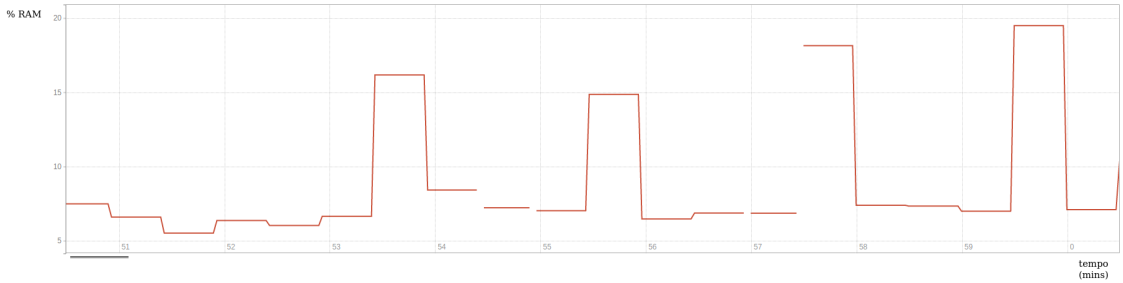

Figura 5.11: Gráfico da Percentagem de Tempo de Processador Utilizado ao longo de dez minutos na terceira experiência.

5.2.3.4 Quarta Experiência - Dez Réplicas

Além das observações feitas entre as experiências anteriores, que se mantêm de igual forma, nesta quarta experiência observa-se que os valores de utilização atingiram pela primeira vez o limite de 25% (2 *cores* do processador) e que os picos máximos de utilização são mais irregulares.

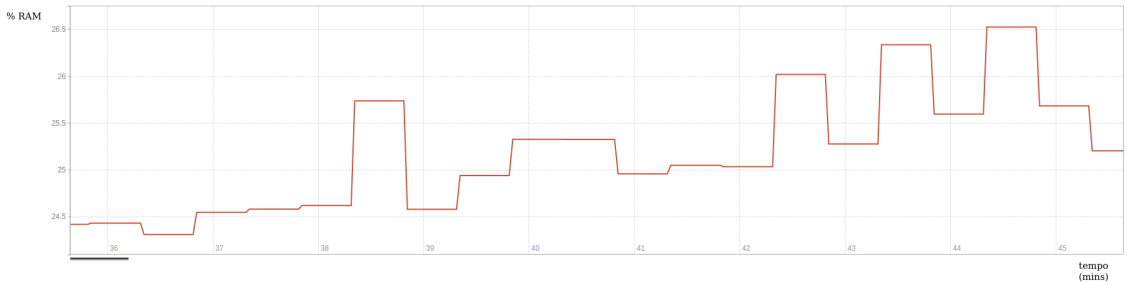

Figura 5.12: Gráfico da Percentagem de Memória Utilizada ao longo de dez minutos na quarta experiência.

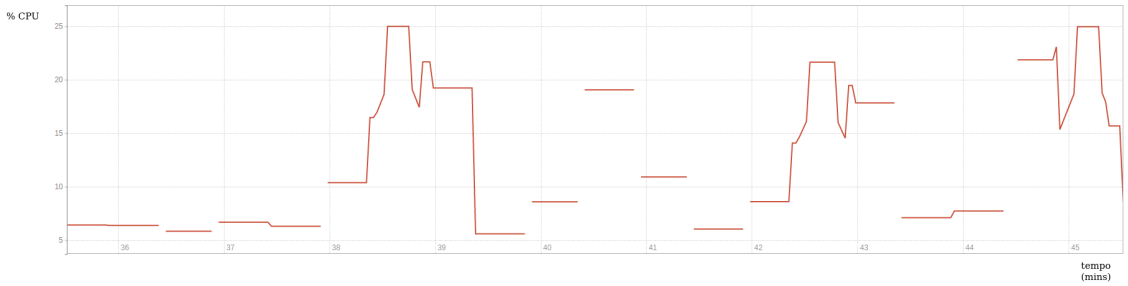

Figura 5.13: Gráfico da Percentagem de Tempo de Processador Utilizado ao longo de dez minutos na quarta experiência.

5.2.3.5 Quinta Experiência - Trinta Réplicas

Nesta quinta experiência observamos que a utilização de memória já não segue o padrão visto nas outras experiências, existe uma maior utilização ao longo do tempo e os picos já não são tão visíveis. Isto é explicado devido ao facto de que no final do período de dois minutos quando o sistema já está a iniciar o seguinte ciclo de execução, algumas das trinta aplicações do ciclo anterior ainda não terminaram a sua execução. Apesar disso, não existe qualquer perda de dados e todos os resultados de processamento são guardados no sistema prontos para serem exportados. No que diz respeito ao gráfico de utilização do processador, apesar de a experiência ter sido repetida múltiplas vezes e os resultados serem sempre semelhantes, o traçado de utilização não é constante. Segundo relatos no repositório de código da ferramenta utilizada, este problema deve-se ao facto de que quando a utilização do processador é muito elevada durante um longo período de tempo a ferramenta não tem resposta do processador no que toca à percentagem de utilização e como tal será considerado que nos locais onde não existe traçado do gráfico o valor seja sempre 25% ou muito próximo disso. Esta suposição foi verificada pela consulta do monitor de sistema da máquina física utilizada.

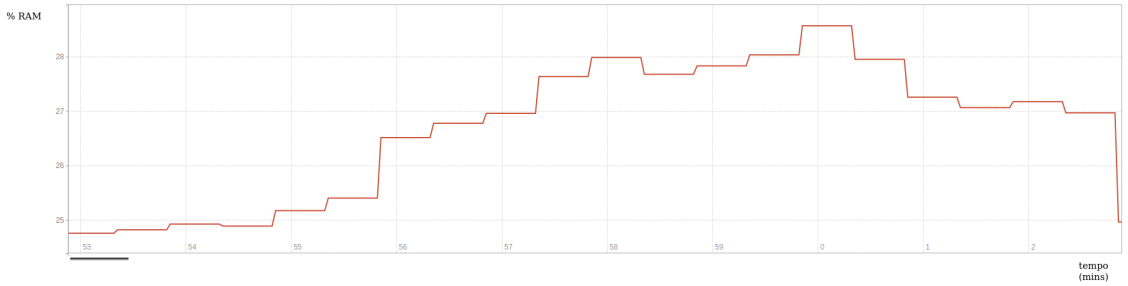

Figura 5.14: Gráfico da Percentagem de Memória Utilizada ao longo de dez minutos na quinta experiência.

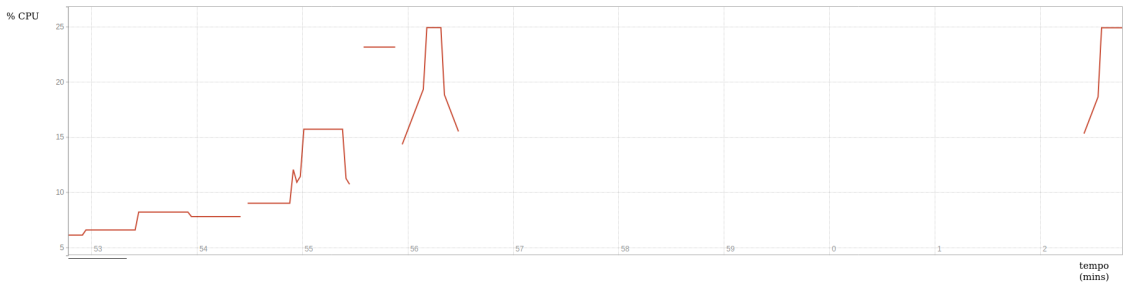

Figura 5.15: Gráfico da Percentagem de Tempo de Processador Utilizado ao longo de dez minutos na quinta experiência.

5.2.3.6 Sexta Experiência - Cinquenta Réplicas

Nesta sexta experiência as observações são as mesmas que na experiência 5.2.3.5. Apenas se verifica que a percentagem de memória utilizada é ligeiramente superior.

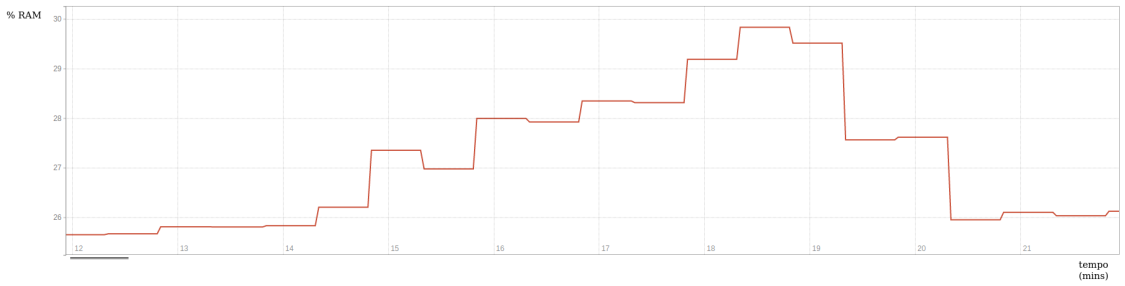

Figura 5.16: Gráfico da Percentagem de Memória Utilizada ao longo de dez minutos na sexta experiência.

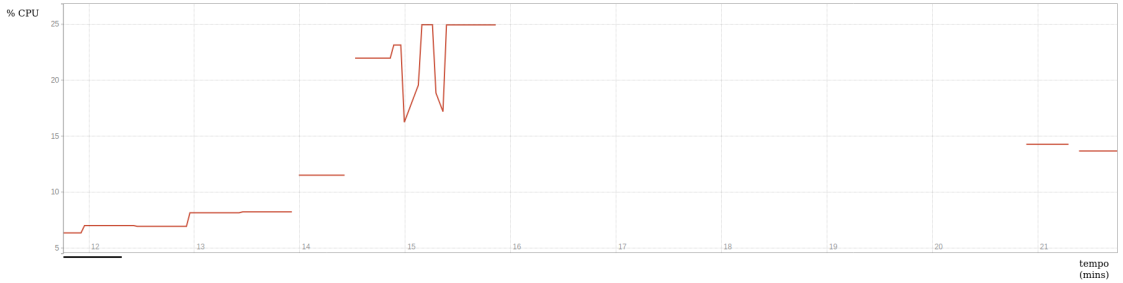

Figura 5.17: Gráfico da Percentagem de Tempo de Processador Utilizado ao longo de dez minutos na sexta experiência.

5.2.4 Resultados Obtidos

Os resultados neste teste mostraram a escalabilidade do sistema. Apesar de só ter sido utilizado um nó no *cluster* com recursos limitados, a única limitação observada foi o processador para um número de réplicas elevado. Este facto é explicado com a complexidade computacional do algoritmo utilizado. O facto de todo o sistema estar montado num *cluster* de *kubernetes*, permite que o sistema facilmente escale horizontalmente aumentando os recursos disponíveis e permite que existam milhares de nós como o utilizado nestes testes. Nos testes com maiores números de réplicas, apesar de os recursos terem sido em certos casos insuficientes, não existiu qualquer perda de dados e todos os resultados de processamento foram enviados para o servidor e exportados para o *url* configurado, sendo que o único constrangimento foi o tempo, uma vez que eram muitos *jobs* para serem criados num único nó. Como tal, verificava-se um atraso de tempo desde o pedido de criação do *job* até o *kuber master* conseguir atender o pedido e ser efetivamente criado.

Numa situação real, o algoritmo em questão utilizado não seria executado em períodos tão curtos de tempo porque os dados obtidos não seriam suficientes para verificar anomalias no ritmo cardíaco.

5.2.5 Teste 4 - Linguagens de Programação

Neste teste pretende-se demonstrar o funcionamento de duas linguagens de programação em simultâneo. Para isso, foi utilizado o mesmo algoritmo utilizado na 5.2.1 desenvolvido em *Python* que é uma linguagem interpretada e foi desenvolvido um algoritmo com o mesmo propósito de calcular a média de um conjunto de números passado como argumento. Foi escolhida como linguagem de programação para este algoritmo *Java*, por se tratar de uma linguagem compilada, ao contrário de *Python*. O algoritmo desenvolvido encontra-se em A.4 e foi criada uma imagem com o algoritmo, tendo como base a imagem *openjdk*.

5.2.5.1 Configuração utilizada

Apenas vai ser descrita a configuração utilizada para a aplicação de *Java* uma vez que para a segunda aplicação foi utilizada a mesma configuração de 5.2.1.

Tabela 5.3: Configuração utilizada para o teste 4.

Configuração de Input	'?,?,?,?,?'
Configuração de Output	'Average=?'
Número de Inputs	5
Url de Importação de Dados	N/A
Url de Exportação de Dados	N/A
Tempo de Repetição	N/A
Número de Outputs	1
Comando de Execução da Aplicação	java Average INPUT

5.2.5.2 Resultados Obtidos

Como podemos ver nas imagens 5.21 e 5.22, os *jobs* foram iniciados com as configurações acima explicadas. Os resultados em 5.19 e 5.20 mostram que o resultado está conforme a configuração definida e o valor obtido em ambas é o mesmo. Com esta simples experiência, mostrou-se que não importa qual o tipo de linguagem que esteja inerente ao algoritmo a executar, desde que a imagem provida tenha os recursos necessários para executar o algoritmo, o sistema consegue lidar com qualquer tipo de algoritmo/aplicação.

Figura 5.18: Jobs da Experiência 4.

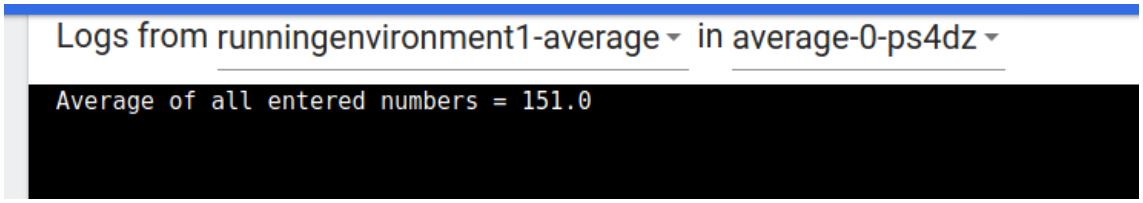

Figura 5.19: Logs do Algoritmo de *Python*.

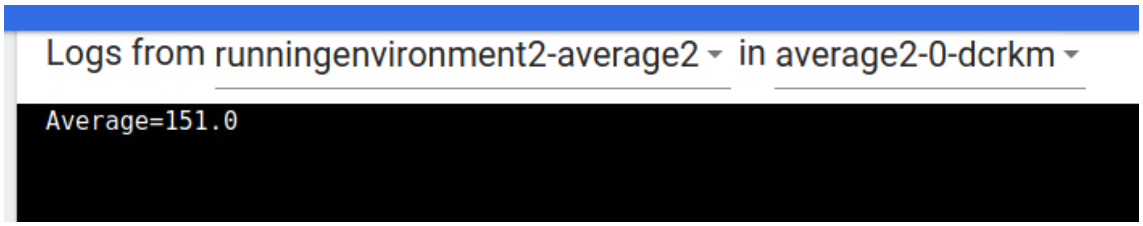

Figura 5.20: Logs do Algoritmo de *Java*.

Figura 5.21: Configuração do Job de *Python*.

Edit a resource

YAML JSON

```
completions: 1
backoffLimit: 6
selector:
  matchLabels:
    controller-uid: 0dd2c7e3-ea11-4024-bf96-c676513e4d25
template:
  metadata:
    creationTimestamp: null
    labels:
      controller-uid: 0dd2c7e3-ea11-4024-bf96-c676513e4d25
      job-name: average2-0
  spec:
    containers:
      - name: runningenvironment2-average2
        image: 'vranalytics/tese:java'
        command:
          - java
          - Average
          - 100 150 200 175 130
        resources: {}
        terminationMessagePath: /dev/termination-log
        terminationMessagePolicy: File
        imagePullPolicy: Always
```

This action is equivalent to: `kubectl apply -f <spec.yaml>`

Figura 5.22: Configuração do Job de *Java*.

Capítulo 6

Conclusão e Trabalho Futuro

Neste último capítulo serão tiradas conclusões sobre os resultados obtidos e será analisado se o sistema resultante cumpre os requisitos inicais. Numa segunda parte será abordado o trabalho futuro e possíveis melhorias para o sistema.

6.1 Conclusão

O objetivo desta dissertação era construir um sistema de processamento de dados na *cloud* como foi mencionado no capítulo 1. Este sistema teria não só que funcionar com a platforma do *Wessenss*, mas também ser facilmente integrado com qualquer outro sistema através da definição de um protocolo.

Apesar de existirem múltiplos sistemas de processamento de dados em tempo real ligados à área de *IoT* como foi analisado no capítulo 2, os sistemas existentes possuem limitações que tornariam a interoperabilidade com o *Wessenss* impraticável, quer por uma questão de limitações de arquitetura, quer por limitações de configuração para o que é pretendido. O sistema desenvolvido é diferente de todos os existentes no sentido em que permite a interoperabilidade de um grande número de sistemas e permite também uma grande liberdade na configuração do processamento de dados, não limitando o tipo de aplicações e algoritmos utilizados. Além da interoperabilidade com outros sistemas, o *VRAnalytics* está especialmente preparado para interoperar com o *Wessenss*, que possui certas especificidades para com o qual o sistema já se encontra configurado, tanto no que toca à importação de dados, como no que toca ao processamento de uma sessão em modo *offline*, e à exportação dos dados.

No capítulo 3 são apresentados os requisitos do sistema, sendo que podemos afirmar que os três principais requisitos deste sistema são:

- **Importação/Exportação de dados ao longo do tempo, para permitir a análise destes em tempo útil.**

- **Suporte para diversas linguagens de programação para desenvolvimento dos algoritmos.**
- **Escalabilidade do Sistema.**

Como podemos ver pelos resultados, foi demonstrado que os dois primeiros requisistos foram cumpridos. Quanto ao terceiro requisito, como se pôde concluir no terceiro teste 5.2.3, a escalabilidade do sistema apenas está limitada pelos recursos físicos atribuídos ao *cluster*. Também foi possível observar na subseção 4.1.4.2 que é muito simples aumentar o número de instâncias do servidor para poder aumentar o número de pedidos recebidos por unidade de tempo. No que toca aos algoritmos, visto que correm de forma isolada num *pod* só estão limitados pelos recursos disponíveis no *cluster* de *Kubernetes*.

Pelos testes realizados concluiu-se que mesmo com recursos computacionais o sistema teve uma boa performance e não houve qualquer perda de dados. Todos os dados foram requeridos, processados e exportados. Apesar de o sistema já estar na iteração seguinte quando recebia os dados processados da iteração anterior, o facto de todo o sistema funcionar de forma assíncrona permite que possam ser recebidos os dados de processamento após um longo período de espera. Concluiu-se também no teste 5.2.5 que o motor funciona com diferentes linguagens em simultâneo e que as performances para as diferentes linguagens são muito semelhantes.

Como tal a plataforma satisfaz todos os requisitos propostos e a utilização do *VRAnalytics* aliado ao *Vital Responder* numa situação real com socorristas poderá não só ajudar a prevenir acidentes com a monitorização constante dos intervenientes, mas ajudar também ao melhor planeamento e gestão de situações de socorro através do fornecimentos de dados e métricas acerca dos socorristas envolvidos, permitindo a tomada de decisões mais informada.

6.2 Trabalho Futuro

Como em todos os sistemas, existe espaço para melhorias e novas funcionalidades. Algumas das novas funcionalidades já tinham sido identificadas como opcionais, no entanto não foram implementadas por não serem prioritárias e a curva de aprendizagem das tecnologias novas ter sido superior ao esperado. Outras funcionalidades serão identificadas quando o sistema começar a ser utilizado em cenários reais.

- **Sistema de Autenticação** - Apesar do modelo de dados ter implementado o conceito de utilizadores e de dono de uma aplicação, não está implementado qualquer sistema de autenticação, permitindo que qualquer utilizador aceda a todas as aplicações através do identificador de utilizador. Esta questão será pertinente se o sistema for utilizado em muitas situações de emergência em simultâneo.
- **Cliente de Configuração** - Um cliente de configuração do sistema estava nos requisitos iniciais ainda que com uma baixa prioridade e como tal, não foi implementado. Um cliente de configuração facilitaria na visualização e configuração do sistema no que toca ao modelo

de dados uma vez que se trata de um sistema complexo. A baixa prioridade foi atribuída uma vez que no futuro se pretende que a configuração seja feita de forma automática pelo servidor *Wessenss* utilizando a *API* do *VRAnalytics*.

- **Interligação de Aplicações** - O sistema no seu estado atual não suporta interligação de aplicações, ou seja, não é possível utilizar diretamente o resultado de uma aplicação como dados de entrada de outra. Na versão atual do sistema tal seria possível exportando os dados obtidos para um servidor e posteriormente uma segunda aplicação importar os dados desse mesmo servidor. Em termos práticos, seria muito mais simples uma abordagem sem ser necessária qualquer importação ou exportação de dados, no entanto a implementação desta abordagem foi considerada de baixa prioridade uma vez que nenhum dos algoritmos atualmente utlizados para processamento dos dados no *Wessenss* requer uma configuração deste tipo.

Além das novas funcionalidades indentificadas com a evolução do *Vital Responder* e adaptação a diferentes contextos, aparecerão novos requisitos e serão identificadas novas melhorias, como acontece em todos os sistemas de informação que continuam a evoluir com o aumento da sua utilização. No entanto, o sistema foi desenvolvido de forma modular, com vista a suportar atualizações e novas funcionalidades de forma mais simples, o que facilita todo este processo. Apesar do trabalho futuro apresentado, o sistema cumpre de forma satisfatória os principais requisistos inicialmente propostos.

Anexo A

Apêndice

A.1 Esquema da Base de Dados

```
1  DROP DATABASE IF EXISTS config;
2
3  DROP USER IF EXISTS  'config'@'%';
4  CREATE DATABASE config CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
5
6  CREATE USER config@'%' IDENTIFIED BY 'config';
7  GRANT ALL PRIVILEGES ON config.* TO config@'%';
8  ALTER USER 'config'@'%' IDENTIFIED WITH mysql_native_password BY 'config';
9
10 USE config;
11 SET foreign_key_checks = 1;
12
13
14 CREATE TABLE Role(
15
16     roleId INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
17     roleName  VARCHAR(15) NOT NULL,
18     roleDescription VARCHAR(64)
19 );
20
21 CREATE Table User (
22 userId INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
23 userName VARCHAR(50) NOT NULL UNIQUE,
24 userPassword VARCHAR(128),
25 userRole INT DEFAULT 1, -- NO DEFAULT
26     FOREIGN KEY(userRole)
27         REFERENCES Role(roleId)
28         ON DELETE CASCADE
29         ON UPDATE CASCADE
30
```

```
31 );
32
33
34 CREATE TABLE EnvVariable(
35 envVariableId INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
36 envVariableName VARCHAR(128),
37 envVariableValue VARCHAR(128)
38 );
39
40 CREATE TABLE PodLabel(
41 podLabelId INT  PRIMARY KEY AUTO_INCREMENT,
42 podLabelName VARCHAR(128),
43 podLabelValue VARCHAR(128)
44 );
45
46 CREATE TABLE RunningEnvironment(
47 runningEnvironmentId INT PRIMARY KEY AUTO_INCREMENT,
48 runningEnvironmentName VARCHAR(50) NOT NULL UNIQUE,
49 runningEnvironmentImage VARCHAR(128) NOT NULL,
50 runningEnvironmentDescription VARCHAR(256) NOT NULL,
51 runningEnvironmentPublicFlag BOOLEAN NOT NULL DEFAULT false,
52 runningEnvironmentOwner INT NOT NULL,
53 FOREIGN KEY(runningEnvironmentOwner) REFERENCES User(userId) ON DELETE CASCADE ON
       UPDATE CASCADE
54 );
55
56
57 CREATE TABLE Configuration(
58 configurationId INT PRIMARY KEY AUTO_INCREMENT,
59 configurationName VARCHAR(128) NOT NULL UNIQUE,
60 configurationRepeat BOOLEAN NOT NULL DEFAULT false,
61 configurationRepeatTime INT NOT NULL DEFAULT 0,
62 configurationIdleTime INT NOT NULL DEFAULT 0,
63 configurationOutputString VARCHAR(512) DEFAULT "?",
64 configurationInputString VARCHAR(512) DEFAULT "?",
65 configurationReady BOOLEAN NOT NULL DEFAULT false
66 );
67
68 CREATE TABLE Output(
69 outputId INT PRIMARY KEY AUTO_INCREMENT,
70 outputOrderNumber INT NOT NULL DEFAULT 0,
71 outputRequestUrl VARCHAR(256) DEFAULT "",
72 outputName VARCHAR(256) NOT NULL,
73 outputConfigurationId INT NOT NULL,
74 outputStringOrder INT NOT NULL DEFAULT 0,
75 outputFromWesenss BOOLEAN NOT NULL DEFAULT FALSE,
76 FOREIGN KEY(outputConfigurationId) REFERENCES Configuration(configurationId) ON
       DELETE CASCADE ON UPDATE CASCADE
77 );
```

```
78
79 CREATE TABLE OutputHistory(
80  outputHistoryId INT PRIMARY KEY AUTO_INCREMENT,
81  outputId INT NOT NULL,
82  outputHistoryOrderNumber INT NOT NULL DEFAULT 0,
83  outputHistoryValue VARCHAR(2000),
84  outputHistorySubmitted BOOLEAN DEFAULT false,
85  outputHistoryStatus INT DEFAULT 0,
86  outputSendTries INT DEFAULT 0,
87  outputHistoryTimeStamp INT NOT NULL DEFAULT 0,
88  FOREIGN KEY(outputId) REFERENCES Output(outputId) ON DELETE CASCADE ON UPDATE
        CASCADE
89 );
90
91
92
93 CREATE TABLE Input(
94 inputId INT PRIMARY KEY AUTO_INCREMENT,
95 inputOrderNumber INT NOT NULL DEFAULT 0,
96 inputRequestUrl VARCHAR(256) DEFAULT "",
97 inputResquestInputKey VARCHAR(256) DEFAULT "",
98 inputName VARCHAR(256) NOT NULL,
99 inputConfigurationId INT NOT NULL,
100 inputStringOrder INT NOT NULL DEFAULT 0,
101 inputOfflineMode BOOLEAN DEFAULT FALSE,
102 inputFromWesenss BOOLEAN NOT NULL DEFAULT FALSE,
103 FOREIGN KEY(inputConfigurationId) REFERENCES Configuration(configurationId) ON
        DELETE CASCADE ON UPDATE CASCADE
104 );
105
106
107 CREATE TABLE InputHistory(
108  inputHistoryId INT PRIMARY KEY AUTO_INCREMENT,
109  inputId INT NOT NULL,
110  inputHistoryValue VARCHAR(2000) DEFAULT "",
111  inputHistoryOrderNumber INT NOT NULL DEFAULT 0,
112  inputHistorySubmitted BOOLEAN DEFAULT false,
113  inputHistoryStatus INT DEFAULT 0,
114  inputSendTries INT  DEFAULT 0,
115  inputHistoryTimeStamp INT NOT NULL DEFAULT 0,
116  FOREIGN KEY(inputId) REFERENCES Input(inputId) ON DELETE CASCADE ON UPDATE CASCADE
117 );
118
119
120
121 CREATE TABLE EnvVariableRunningEnvironment(
122 envVariable INT NOT NULL,
123 runningEnvironment INT NOT NULL,
124 PRIMARY KEY(envVariable,runningEnvironment),
```

```
125 FOREIGN KEY(envVariable) REFERENCES EnvVariable(envVariableId) ON DELETE CASCADE ON
        UPDATE CASCADE,
126 FOREIGN KEY(runningEnvironment) REFERENCES RunningEnvironment(runningEnvironmentId)
        ON DELETE CASCADE ON UPDATE CASCADE
127 );
128
129 CREATE TABLE PodLabelRunningEnvironment(
130 podLabel INT NOT NULL,
131 runningEnvironment INT NOT NULL,
132 PRIMARY KEY(podLabel,runningEnvironment),
133 FOREIGN KEY(podLabel) REFERENCES PodLabel(podLabelId) ON DELETE CASCADE ON UPDATE
        CASCADE,
134 FOREIGN KEY(runningEnvironment) REFERENCES RunningEnvironment(runningEnvironmentId)
        ON DELETE CASCADE ON UPDATE CASCADE
135 );
136
137 CREATE TABLE Application (
138 applicationId INT PRIMARY KEY AUTO_INCREMENT,
139 applicationName VARCHAR(50) NOT NULL UNIQUE,
140 applicationDescription VARCHAR(100),
141 applicationRunningCommand VARCHAR(300),
142 applicationPublicFlag BOOLEAN NOT NULL DEFAULT false,
143 applicationOwner INT NOT NULL,
144 applicationRunningEnvironment INT NOT NULL,
145 applicationConfiguration INT NOT NULL,
146 applicationRunning BOOLEAN NOT NULL DEFAULT false,
147 FOREIGN KEY(applicationRunningEnvironment) REFERENCES Application(applicationId) ON
        DELETE CASCADE ON UPDATE CASCADE,
148 FOREIGN KEY(applicationOwner) REFERENCES User(userId) ON DELETE CASCADE ON UPDATE
        CASCADE,
149 FOREIGN KEY(applicationConfiguration) REFERENCES Configuration(configurationId) ON
        DELETE CASCADE ON UPDATE CASCADE
150 );
151
152
153 DROP TRIGGER IF EXISTS input_verification;
154 DELIMITER $$
155 CREATE TRIGGER input_verification
156 AFTER INSERT ON InputHistory FOR EACH ROW
157 BEGIN
158     DECLARE configId integer;
159     SET @configId := (SELECT Input.inputConfigurationId FROM Input JOIN
            InputHistory ON Input.inputId = InputHistory.inputId WHERE NEW.inputId=
            Input.inputId LIMIT 1);
160         IF (SELECT count(*)FROM InputHistory JOIN Input on InputHistory.inputId =
                Input.inputId WHERE InputHistory.inputHistoryOrderNumber=Input.
                inputOrderNumber AND InputHistory.inputHistoryStatus >= 300)<=0 THEN
161         UPDATE Configuration SET configurationReady = true WHERE configurationId=
                @configId;
```

```
162         ELSE
163         UPDATE Configuration SET configurationReady = false WHERE configurationId=
                @configId;
164          END IF;
165 END $$
166 DELIMITER ;
167
168 DROP TRIGGER IF EXISTS input_verification_update;
169 DELIMITER $$
170 CREATE TRIGGER input_verification_update
171 AFTER UPDATE ON InputHistory FOR EACH ROW
172 BEGIN
173     DECLARE configId integer;
174     SET @configId := (SELECT Input.inputConfigurationId FROM Input JOIN
            InputHistory ON Input.inputId = InputHistory.inputId WHERE NEW.inputId=
            Input.inputId LIMIT 1);
175         IF (SELECT count(*) FROM InputHistory JOIN Input on InputHistory.inputId =
                Input.inputId WHERE InputHistory.inputHistoryOrderNumber=Input.
                inputOrderNumber AND InputHistory.inputHistoryStatus >= 300)<=0 THEN
176         UPDATE Configuration SET configurationReady = true WHERE configurationId=
                @configId;
177         ELSE
178         UPDATE Configuration SET configurationReady = false WHERE configurationId=
                @configId;
179          END IF;
180 END $$
181 DELIMITER ;
```

Listing A.1: Esquema da Base de Dados.

A.2 Algoritmo Utilizado para a Experiência 1.

```
1 import sys
2 import numpy as np
3
4 numberList = sys.argv[1]
5 numberList = np.fromstring(numberList, dtype=int, sep=',')
6 average = sum(numberList) / len(numberList)
7 print("Average of all entered numbers = {}".format(average))
```

Listing A.2: Algoritmo Utilizado para a Experiência 1.

A.3 Algoritmo Utilizado para a Experiência 2.

```
import numpy as np
from biosppy.signals import ecg
from hrvanalysis import get_time_domain_features
import sys

# INPUT HANDLING

signal = sys.argv[1]

signal = np.fromstring(signal, dtype=float, sep=',')
temp = signal.tolist()

for i, s in enumerate(temp):
    if temp[i] > 0:
        temp[i] = round(60000/temp[i])
    else:
        temp[i] = 0

signal = np.array(temp)
# process it and plot
out = ecg.ecg(signal=signal, sampling_rate=120., show=False)
rr=out["rpeaks"]
time_domain_features = get_time_domain_features(rr)
print(time_domain_features)
```

Listing A.3: Algoritmo Utilizado para a Experiência 2.

A.4 Algoritmo Utilizado para a Experiência 4.

```
1  import java.util.Scanner;
2  class Average
3  {
4     public static void main(String args[])
5     {
6    double res=0;
7     String[] test =args[0].split(" ");
8
9     long n = test.length;
10   for(int i=0;i<n;i++)
11            res=res+Integer.parseInt(test[i]);
12
13           res= res/ n;
14
15   System.out.println("Average=" +res);
16    }
17
18 }
```

Listing A.4: Algoritmo Utilizado para a Experiência 4.

Referências

[1] Data Processing Layers on IoT. http://www.ccg.pt/cloud-computing-vs-fog-computing-vs-edge-computing-na-internet-das-coisas-i Online; Acedido Fevereiro 2020.

[2] Evolution of the number of IoT connected devices from 2015-2025. https://ipropertymanagement.com/research/iot-statistics. Online; Acedido Fevereiro 2020.

[3] IoT Analytics - ThingSpeak Internet of Things. https://thingspeak.com/. Online; Acedido Fevereiro 2020.

[4] Kubernetes - Architecture. https://geekflare.com/kubernetes-architecture/. Online; Acedido Maio 2020.

[5] Kubernetes vs. Docker Swarm: What's the Difference? - The New Stack. https://thenewstack.io/kubernetes-vs-docker-swarm-whats-the-difference/. Online; Acedido Janeiro 2020.

[6] Learn Apache Mesos: A beginner's guide to scalable cluster management and ... - Manuj Aggarwal - Google Livros. https://books.google.pt/books?id=N-l1DwAAQBAJ&pg=PA6&lpg=PA6&dq=slave+daemons+mesos&source=bl&ots=UTeMeGWWu3&sig=ACfU3U0NV1kLNJVcRFWQg5fHmOrmCF6PqQ&hl=pt-PT&sa=X&ved=2ahUKEwi01ez60uPpAhVkDWMBHds7BIcQ6AEwAHoECAkQAQ#v=onepage&q=slavedaemonsmesos&f=false. Online; Acedido Junho 2020.

[7] Node.js. https://nodejs.org/en/. Online; Acedido Maio 2020.

[8] Prometheus - Monitoring system & time series database. https://prometheus.io/. Online; Acedido Julho 2020.

[9] TypeScript – Wikipédia, a enciclopédia livre. https://pt.wikipedia.org/wiki/TypeScript. Online; Acedido Maio 2020.

[10] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, e Matei Zaharia. A view of cloud computing, apr 2010.

[11] Radhakisan Baheti e Helen Gill. Cyber-physical systems. *The impact of control technology*, 12(1):161–166, 2011.

[12] David Bernstein. Containers and Cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing*, 1(3):81–84, sep 2014.

[13] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, e Sateesh Addepalli. Fog computing and its role in the internet of things. Em *Proceedings of the first edition of the MCC workshop on Mobile cloud computing - MCC '12*, página 13, New York, New York, USA, 2012. ACM Press.

[14] Michael Eder e Holger Kinkelin. Hypervisor- vs. Container-based Virtualization. *Network Architectures and Services*, (July):1–7, 2016.

[15] Vincent Emeakaroha, Kaniz Fatema, Philip Healy, e John Morrison. Sensors & Transducers Contemporary Analysis and Architecture for a Generic Cloud-based Sensor Data Management Platform. *Sensors and transducers*, 185:100–112, 2015.

[16] Vincent C. Emeakaroha, Neil Cafferkey, Philip Healy, e John P. Morrison. A Cloud-Based IoT Data Gathering and Processing Platform. Em *Proceedings - 2015 International Conference on Future Internet of Things and Cloud, FiCloud 2015 and 2015 International Conference on Open and Big Data, OBD 2015*, páginas 50–57. Institute of Electrical and Electronics Engineers Inc., oct 2015.

[17] Moeen Hassanalieragh, Alex Page, Tolga Soyata, Gaurav Sharma, Mehmet Aktas, Gonzalo Mateos, Burak Kantarci, e Silvana Andreescu. Health Monitoring and Management Using Internet-of-Things (IoT) Sensing with Cloud-Based Processing: Opportunities and Challenges. Em *Proceedings - 2015 IEEE International Conference on Services Computing, SCC 2015*, páginas 285–292. Institute of Electrical and Electronics Engineers Inc., aug 2015.

[18] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, e Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. *Proceedings of NSDI 2011: 8th USENIX Symposium on Networked Systems Design and Implementation*, páginas 295–308, 2011.

[19] Nancy Jain e Sakshi Choudhary. Overview of virtualization in cloud computing. Em *2016 Symposium on Colossal Data Analysis and Networking, CDAN 2016*. Institute of Electrical and Electronics Engineers Inc., sep 2016.

[20] Steven A. Kahn, Tina L. Palmieri, Soman Sen, Jason Woods, e Oliver L. Gunter. Factors Implicated in Safety-related Firefighter Fatalities. *Journal of Burn Care & Research*, 38(1):e83–e88, 2017.

[21] S Pradeep Kumar, Vemuri Richard Ranjan Samson, U Bharath Sai, P L S D Malleswara Rao, e K Kedar Eswar. Smart health monitoring system of patient through IoT. Em *2017 International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)*, páginas 551–556. IEEE, feb 2017.

[22] Somayya Madakam, R. Ramaswamy, e Siddharth Tripathi. Internet of Things (IoT): A Literature Review. *Journal of Computer and Communications*, 03(05):164–173, 2015.

[23] Nikhil Marathe, Ankita Gandhi, e Jaimeel M Shah. Docker Swarm and Kubernetes in Cloud Computing Environment. Em *2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI)*, páginas 179–184. IEEE, apr 2019.

[24] Mohsen Marjani, Fariza Nasaruddin, Abdullah Gani, Ahmad Karim, Ibrahim Abaker Targio Hashem, Aisha Siddiqa, e Ibrar Yaqoob. Big IoT Data Analytics: Architecture, Opportunities, and Open Research Challenges. *IEEE Access*, 5:5247–5261, 2017.

[25] Liam McNamara, Beshr Al Nahas, Simon Duquennoy, Joakim Eriksson, e Thiemo Voigt. Demo Abstract: SicsthSense - Dispersing the Cloud. 2014.

[26] Giovanni Morana e Daniele Zito. Coope4M: A deployment framework for communication-intensive applications on mesos. *Proceedings - 2018 IEEE 27th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises, WETICE 2018*, páginas 42–47, 2018.

[27] Gonçalo Pimentel, Susana Rodrigues, Pedro Alberto Silva, António Vilarinho, Rui Vaz, e João Paulo Silva Cunha. A wearable approach for intraoperative physiological stress monitoring of multiple cooperative surgeons. *International Journal of Medical Informatics*, 129:60–68, sep 2019.

[28] Ragunathan Rajkumar, Insup Lee, Lui Sha, e John Stankovic. Cyber-physical systems: The next computing revolution. Em *Proceedings - Design Automation Conference*, páginas 731–736, 2010.

[29] Carlos Oberdan Rolim, Fernando Luiz Koch, Carlos Becker Westphall, Jorge Werner, Armando Fracalossi, e Giovanni Schmitt Salvador. A cloud computing solution for patient's data collection in health care institutions. Em *2nd International Conference on eHealth, Telemedicine, and Social Medicine, eTELEMED 2010, Includes MLMB 2010; BUSMMed 2010*, páginas 95–99, 2010.

[30] Jay Shah e Dushyant Dubaria. Building modern clouds: Using docker, kubernetes google cloud platform. Em *2019 IEEE 9th Annual Computing and Communication Workshop and Conference, CCWC 2019*, páginas 184–189. Institute of Electrical and Electronics Engineers Inc., mar 2019.

[31] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, e Lanyu Xu. Edge Computing: Vision and Challenges. *IEEE Internet of Things Journal*, 3(5):637–646, oct 2016.