UNIVERSIDADE BEIRA INTERIOR
Engenharia

Comparação do Desempenho da Plataforma Apache OpenWhisk em Ambiente Local Usando Kubernetes e na IBM Cloud

Versão Definitiva Após Defesa Pública

Gilberto Maloco Mpaca Gimbi

Dissertação para obtenção do Grau de Mestrado em
Engenharia Informática
(2º Ciclo de Estudos)

Orientador: Prof. Doutor Mário Freire

Covilhã, Setembro de 2021

Projeto de Dissertação elaborado no Instituto de Telecomunicações - Delegação da Covilhã e no Departamento de Informática da Universidade da Beira Interior e submetido à Universidade da Beira Interior para discussão em provas públicas.

Este trabalho foi financiado pela FCT/MCTES através de fundos nacionais e quando aplicável cofinanciado por fundos comunitários no âmbito do projeto UIDB/50008/2020 e foi suportado pela operação Centro-01-0145-FEDER-000019 - C4 - Centro de Competências em Cloud Computing, cofinanciada pelo Fundo Europeu de Desenvolvimento Regional (FEDER) através do Programa Operacional Regional do Centro (Centro 2020), no âmbito do Sistema de Apoio à Investigação Científica e Tecnológica - Programas Integrados de ICDT.

Cofinanciado por:

Dedicatória

Em primeiro lugar dedico este trabalho a Deus por ser essencial na minha vida, meu destino e guia e aos meus pais e irmãos pelo apoio incondicional.

Agradecimentos

Gostaria de expressar a minha profunda gratidão ao meu professor e orientador Mário Marques Freire, por me dar a oportunidade de trabalhar nesta dissertação e fornecer valiosos conhecimentos e diretrizes para elaboração deste trabalho. Também aos professores do Departamento de Informática da Faculdade de Engenharia da UBI pelo ensino de qualidade. Sem esquecer os meus pais e irmãos pelo apoio incondicional e motivação constantes durante a formação.

Resumo

Os recentes avanços nas tecnologias de virtualização e computação em nuvem levaram o surgimento da computação sem servidor, uma tecnologia também conhecida como Function-as-a-Service. É um modelo de computação em nuvem que visa abstrair a gestão de servidores e as decisões de infraestrutura de baixo nível dos utilizadores, sendo que o utilizador cria, desenvolve e implanta funções e aplicações e a gestão do servidor fica a cargo de provedor de serviço de nuvem. Atualmente, os principais provedores de serviços em nuvem pública oferecem plataformas de computação sem servidor. No entanto, tais plataformas requerem que as funções sejam escritas ou implantadas de uma determinada maneira, o que resulta em vendor lock-in (dependência do fornecedor). Várias plataformas sem servidor open source foram propostas para permitir a execução de computação sem servidor em infraestruturas privadas, de maneira a evitar, assim, qualquer forma de dependência de fornecedores.

Esta dissertação pretende comparar o desempenho da plataforma sem servidor open source Apache OpenWhisk numa implementação local, e a plataforma sem servidor da IBM Cloud, denominado de IBM Cloud Functions que usa Apache OpenWhisk como gestão para funções como serviço. No ambiente local, Apache Openwhisk foi implementada no cluster do Kubernetes, as configurações, execuções e invocações de funções foi feita usando a ferramenta de linha de comando da Apache OpenWhisk, conhecido como CLI OpenWhisk (wsk). Na IBM Cloud, as mesmas foram feitas na interface de utilizador baseada em Web da IBM Cloud Functions. Os experimentos envolveram tempo de respostas de invocações de funções usando a linguagens PHP. Para avaliação de tempo de respostas foi usada a ferramenta de monitorização Prometheus e Grafana no ambiente local, na IBM foi através IBM Cloud Functions Dashboad.

Palavras-chave

Apache OpenWhisk, Kubernetes, IBM Cloud, Desempenho.

Abstract

Recent advances in virtualization and cloud computing technologies have led to the emergence of serverless computing, a technology also known as Function-as-a-Service, a cloud computing model that aims to abstract server management and low-level infrastructure decisions away from developers, where the user creates, develops and deploys functions and applications and server management is the responsibility of the cloud service provider. Today, leading public cloud service providers offer serverless computing platforms. However, such platforms require functions to be written or implemented in a certain way, which results in a vendor lock-in (vendor dependency). Several open source serverless platforms have been proposed to allow the execution of serverless computing in private infrastructures, thus avoiding any form of dependence on suppliers.

This dissertation aims to compare the performance of the open source Apache OpenWhisk serverless platform in a local implementation, and the IBM Cloud serverless platform, called IBM Cloud Functions which uses Apache OpenWhisk as management for functions as a service. In the local environment, Apache Openwhisk was implemented in the Kubernetes cluster, the configurations, executions and function invocations were done using the Apache OpenWhisk command line tool, known as CLI OpenWhisk (wsk). In the IBM Cloud, they were done in the IBM Cloud Functions web-based user interface. The experiments involved function invocation response times using the PHP languages. To evaluate the response time, the monitoring tool Prometheus and Grafana was used in the local environment, at IBM it was through IBM Cloud Functions Dashboad.

Keywords

Apache OpenWhisk, Kubernetes, IBM Cloud, Performance.

Índice

Lista de Figuras

Lista de Tabelas

Lista de Acrónimos

API	Application Programming Interface
AWS	Amazon Web Services
BaaS	Backend-as-a-Service
CLI	Command-line interface
CNCF	Cloud Native Comptuting Foundation
CPU	Central Processing Unit
CRDs	Custom Resource definitions
DNS	Domain Name System
DSRM	Design Science Research Methodology
EKS	Elastic Kubernetes Service
FaaS	Functions-as-a-Service
GCP	Google Cloud Platform
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure-as-a-Service
IBM	International Business Machines Corporation
IoT	Internet of Things
IP	Internet Protocol
LXC	Linux Containers
OSI	Open System Interconnection
PaaS	Platform-as-a-service
PHP	Hypertext Preprocessor
REST	Representational State Transfer
SaaS	Software-as-a-Service
SO	Sistema Operativo
SSH	Secure Socket Shell
TI	Tecnologia de Informação
VM	Virtual Machine
VMM	Virtual Machine Manager

Capítulo 1

Introdução

1.1 Âmbito e Foco da Dissertação

Os recentes avanços nas tecnologias de virtualização e computação em nuvem levaram ao surgimento da computação sem servidor, uma tecnologia que alavanca a virtualização baseada em containers para implementar aplicações e serviços. [DKP20]. A computação sem servidor é uma arquitetura de implantação de aplicação que visa fornecer funcionalidade baseada em eventos pré-pagos. As aplicações são desenvolvidas de acordo com cada processo desejado e o evento que o invoca. As plataformas sem servidor fornecem a infraestrutura para implantar código para execução em nuvem e definir a lógica de processamento de eventos que solicita que as funções sejam executadas usando o modelo: evento, trigger e action (ação) . A computação sem servidor abstrai a gestão da infraestrutura subjacente aos utilizadores, permitindo apenas o acesso mínimo em alguns parâmetros básicos, como alocação de memória de função e tempo limite de execução de função [KGB21]. A computação sem servidor permite-nos construir e executar aplicações e serviços sem nos preocupar com o provisionamento do servidor, dimensionamento e como geri-lo. A infraestrutura pode ser dimensionada em qualquer extensão com provisionamento manual ou automático de novos servidores em caso de aumento repentino no uso. Isto economiza tempo de desenvolvimento e ajuda os programadores a se concentrarem na programação da lógica de negócio principal, em vez de se preocuparem com a gestão e operação de servidores ou tempos de execução [SS18].

Atualmente, os principais provedores de serviços em nuvem pública oferecem plataformas de computação sem servidor. No entanto, tais plataformas requerem que as funções sejam escritas ou implantadas de uma determinada maneira, o que resulta em vendor lock-in(dependência do fornecedor). Várias plataformas sem servidor open source foram propostos para permitir a execução de computação sem servidor em infraestruturas privadas, de maneira a evitar, assim, qualquer forma de dependência de fornecedores [PKC19]. O foco desta dissertação consiste em comparar o desempenho da plataforma open source sem servidor Apache OpenWhisk no ambiente local, usando Kubernetes e a plataforma sem servidor da nuvem pública da IBM Cloud, denominado por IBM Cloud Functions/ Apache OpenWhisk.

1.2 Definição de Problemas e Objetivo

A plataforma Apache Openwhisk é uma plataforma sem servidor open source que pode ser implementada em ambiente local e tem sido usado por alguns provedores de serviços de

nuvens públicas como a base de gestão de FaaS. As organizações podem usar as plataformas open source em vez das plataformas comercias, de maneira a evitar dependência de fornecedor de oferta sem servidor e a diminuição de custos. Com esta dissertação pretende-se investigar o desempenho que a plataforma Apache OpenWhisk pode apresentar localmente em comparação com a plataforma sem servidor da IBM Cloud, denominado de IBM CLoud Functions.

1.3 Estratégia para Resolver o Problema

A metodologia de investigação fornece uma forma de resolver sistematicamente um problema de investigação; isto pode ser entendido como uma ciência que estuda como a investigação é feita cientificamente. Dentro dela são formuladas as várias etapas que devem ser adotadas no estudo do problema de investigação [Nal14]. Para a realização do trabalho proposto , foi adotada a metodologia investigação designada por Design Science Research Methodology (DSRM) [PTRC07]. Foram definidas várias fases conforme se ilustra na figura 1.1.

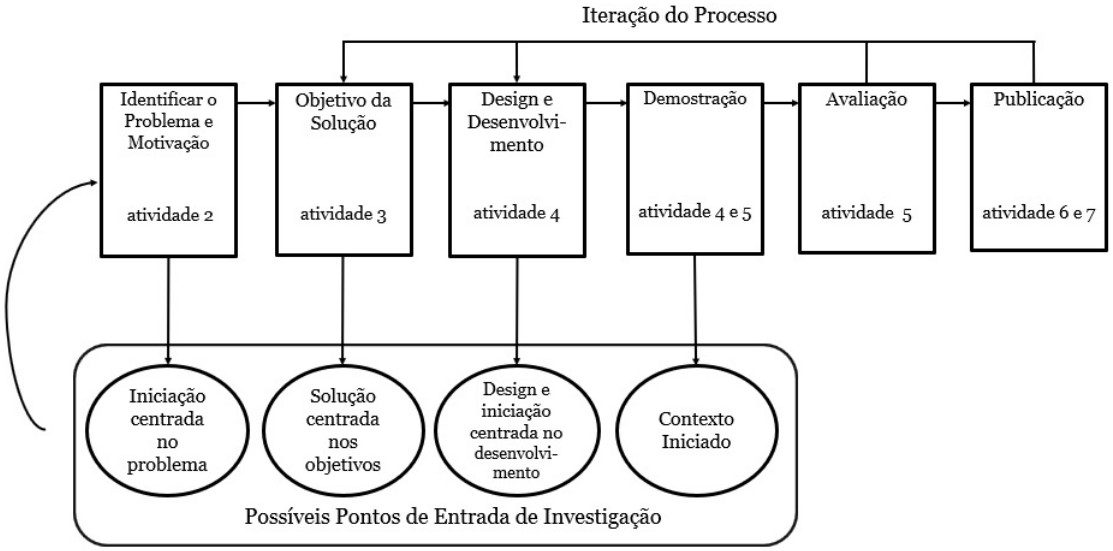

Figura 1.1: Fases de processo de elaboração do trabalho (Redesenhado e adaptado a partir de [PTRC07]

A metodologia usada permitiu a realização das atividades, que estão sequencialmente em iteração com cada atividade subsequente. Podemos descrever cada uma destas atividades da seguinte forma:

1. Background e Estado da Arte: Nesta fase foram investigados e apresentados alguns conceitos fundamentais e tecnologias de maneira a permitir uma melhor compreensão do trabalho proposto. Foram abordados conceitos sobre virtualização, tecnologias de container Docker, Kubernetes, Computação na nuvem, plataformas sem servidor, incluindo as plataformas comercias e open source, e com maior detalhe sobre Apache OpenWhisk por ser uns dos conceitos chave do trabalho. Esta fase foi

feita através de pesquisas bibliográficas, baseadas em fundamentos da literatura e trabalhos relacionados.

2. Especificação do Problema e Motivação: A plataforma Apache Openwhisk é uma plataforma sem servidor open source que pode ser implementada no ambiente local e tem sido usado por alguns provedores de serviços de nuvens públicas como a base de gestão de FaaS. As organizações podem usá-la localmente, de maneira a evitar dependência de fornecedor de oferta sem servidor e a diminuição de custos. Sendo assim, pretende-se perceber como apresenta o desempenho desta plataforma open source Apache OpenWhisk e comparar com a plataforma sem servidor comercial da IBM Cloud, nomeadamente IBM Cloud Functions.

3. Objetivo da Investigação: O objetivo principal consiste em comparar o desempenho da plataforma Apache OpenWhisk em ambiente local usando Kubernetes e a plataforma sem servidor da nuvem pública da IBM Cloud, IBM Cloud Functions. A partir do objetivo principal, definiu-se os seguintes objetivos específicos:

   (a) Instalar e configurar a plataforma Apache OpenWhisk junto com a plataforma Kubernetes em ambiente local;

   (b) Criar e comparar funções em diferentes configurações, envolvendo os dois ambientes distintos, local e na nuvem pública da IBM Cloud. Focando-se na latência de tempo de resposta (response time) de invocação de funções, envolvendo a geração de números primos e Rest API de Países.

4. Especificação da Arquitetura do Sistema e Implementação do Ambiente Experimental: Nesta atividade dedicou-se a implementação do ambiente experimental proposto que envolveu em dois ambientes distintos. Na implementação local foi criada um ambiente sem servidor com a plataforma Apache OpenWhisk e Kubernetes, onde o Apache OpenWhisk foi implantado no cluster do Kubernetes. Na nuvem pública da IBM Cloud as configurações foram feitas na Interface de Utilizador da IBM Cloud Functions baseada em Web.

5. Avaliação da Solução a Implementar e Resultados Esperados: As métricas para avaliação deste trabalho, focou-se no tempo de resposta de invocação de funções. Foi usada a ferramenta *Prometheus* e *Grafana* para avaliação e análise do desempenho. As referidas ferramentas foram integradas no cluster Kubernetes em com junto com Apache OpenWhisk. Na nuvem pública foi usada o *Dashboard da IBM Cloud Functions* que permite monotorizar invocações de funções, desempenho e erros.

6. Escrita da Dissertação de Mestrado: Está fase foi dedicada à escrita da Dissertação de Mestrado que que foi elaborada em conformidade com background e estado da arte e em conjunto com o trabalho prático.

7. Disseminação dos Resultados: O resultado obtido neste trabalho proporcionará um artigo.

Ainda assim, foi criado um diagrama de Gannt conforme se ilustra a figura 1.2, de modo a perceber os avanços das execuções das atividades mencionadas anteriormente, incluindo os dois semestres do ano letivo 2020/2021.

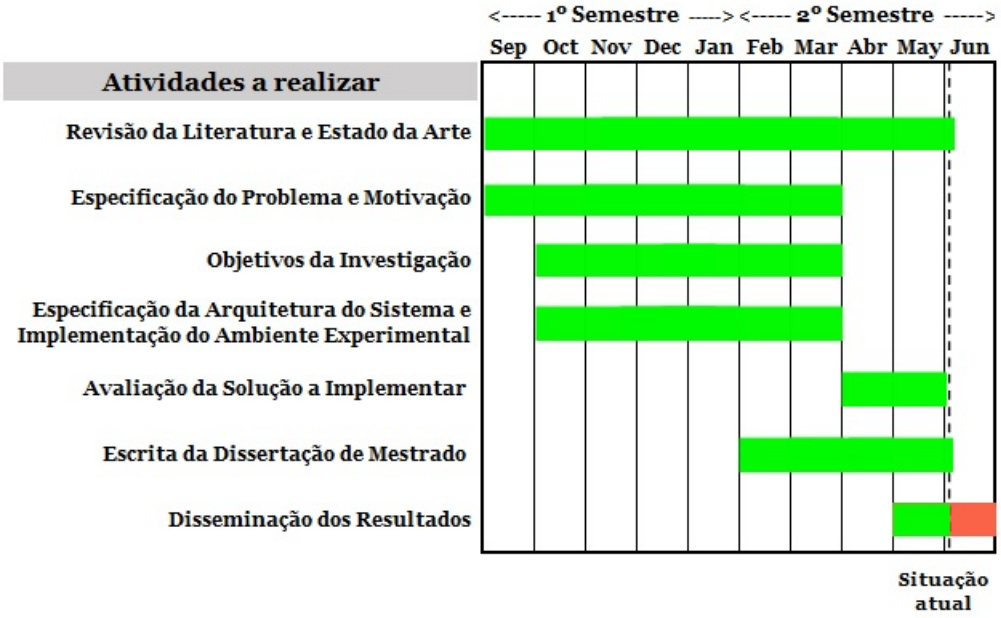

Figura 1.2: Diagrama de Gantt.

A atividade de revisão de literatura, especificação de problemas e motivação e Especificação da Arquitetura do Sistema e Implementação do Ambiente Experimental foram iniciadas no primeiro semestre e concluídas no segundo semestre. As atividades referentes à avaliação da solução implementada, os resultados esperados, a escrita da dissertação e a disseminação dos resultados foram iniciadas no segundo semestre e concluídas no mesmo semestre.

1.4 Limitações do Trabalho Realizado

O número limitado de máquinas para implementação e avaliação do ambiente experimental referente ao ambiente local, envolvendo Apache OpenWhisk e Kubernetes. Apenas foi disponibilizada uma máquina física que serviu de host para as máquinas virtualizadas que foram criadas para os testes.

1.5 Organização da Dissertação

A dissertação está dividida em cinco capítulos principais organizados da seguinte forma:

- Capítulo 1 - Introdução: Descreve o âmbito e foco da dissertação, definição do problema e objetivo, estratégia para resolver o problema, limitação do trabalho realizado e a organização da dissertação;

- Capítulo 2 - Background e Estado da Arte: Apresenta os fundamentos teóricos para compreensão do trabalho, sendo que nelas foram abordados assuntos sobre Virtualização, tecnologia de container Docker, ferramenta de orquestração de container Kubernetes, computação na nuvem, computação sem servidor, tipos de plataformas sem servidor, uma descrição mais abrangente sobre a Apache OpenWhisk por ser umas das chaves do trabalho, e por fim, trabalhos relacionados;
- Capítulo 3 - Implementação do Ambiente Experimental e Análise dos Resultados Experimentais: Descreve o ambiente experimental, a qual inclui a sua implementação, caraterização, especificações de hardware e software. Ainda assim, apresenta as ferramentas para análise de desempenho usadas, análise dos resultados experimentais e a comparação experimental do desempenho das duas plataformas em estudo;
- Capítulo 4 - Conclusão: Apresenta as principais conclusões e sugestões para o trabalho futuro.

Capítulo 2

Background e Estado da Arte

2.1 Introdução

Este capítulo tem como objetivo apresentar os fundamentos teóricos associados aos elementos que formam a base necessária para o melhor entendimento do trabalho. É apresentado um breve conceito sobre virtualização, aborda sobre tecnologia de container, nomeadamente Docker, plataforma de orquestração container Kubernetes, computação na nuvem, fundamento sobre computação sem servidor. Além dos tipos de plataformas, incluiu-se detalhes sobre a plataforma Apache OpenWhisk e apresentam-se alguns trabalhos relacionados.

2.2 Conceitos Básicos Sobre Virtualização

A virtualização é a tecnologia fundamental que impulsiona a computação em nuvem. Virtualização é uma técnica para abstrair as características físicas de recursos computacionais de forma a que outros sistemas, aplicações e utilizadores finais possam interagir com esses recursos [Lou18]. A virtualização usa software para criar uma camada de abstração que permite que os elementos de hardware de uma máquina como processadores, memória, armazenamento, entre outros, sejam divididos em várias máquinas virtuais. Cada máquina virtual executa o seu próprio sistema operativo (SO) e comporta-se como uma máquina independente [IBM19].

Atualmente esta tecnologia tem sido padrão na arquitetura de TI organizacionais, impulsionando também a economia da computação em nuvem. Os principais benefícios da virtualização incluem independência de hardware, alta disponibilidade, isolamento, eficiência de recursos, segurança e a facilidade de gestão. Portanto, é considerado como a base da computação em nuvem [ZLP16].

2.2.1 Tipos de Virtualização

Nesta subsecção aborda-se de forma breve os dois tipos de virtualização mais populares, sendo elas a virtualização baseada em hypervisor e a virtualização baseada em container.

Virtualização baseada em hypervisor: Atende aos requisitos de alta funcionalidade e confiabilidade de sistemas embarcados complexos. Este tipo de virtualização é possível através de um software denominado de hypervisor, também conhecido como monitor de máquina

virtual (VMM) que cria e executa máquinas virtuais (também conhecido de guest). O hypervisor permite que uma máquina host suporte várias máquinas virtuais guests, partilhando virtualmente seus recursos, como memória e processamento, separa o sistema operativo e o hardware para partilhar o hardware entre várias máquinas virtuais (VM), cada uma executando uma instância isolada de um sistema operativo que atende à execução de um subconjunto de aplicações[Vmw20], [GADP14]. Os hypervisors tornam a virtualização possível ao traduzir solicitações entre os recursos físicos e virtuais. Existem dois tipos de virtualização baseadas em hypervisor a saber:

1. Hypervisor do tipo 1: São executados diretamente no hardware host e podem monitorizar os sistemas operativos. Estes hypervisors são independentes dos sistemas operativos, qualquer problema que houver nas máquinas virtuais ou nos sistemas operativos guests não afeta outra máquina virtual [Ans16]. Exemplos de esses são *VMWare ESXi*, *Microsoft Hyper-V*, *VMware ESX/ESXi*, *KVM*, *Oracle VM Server*, *Citrix Hypervisor* [Yfa20].

2. Hypervisor do tipo 2: Estes são também chamados de hypervisors hospedados e são instalados na parte superior de um sistema operativo. Oferecem suporte a vários sistemas operativos guests. O hypervisor tipo 2 depende do sistema operativo host [Ans16], exemplos destes são *Oracle Solaris Zones*, *Oracle VM Server para x86*, *Oracle VM Virtual Box*, *VMWare Workstation*, *VMware Fusion* [Yfa20].

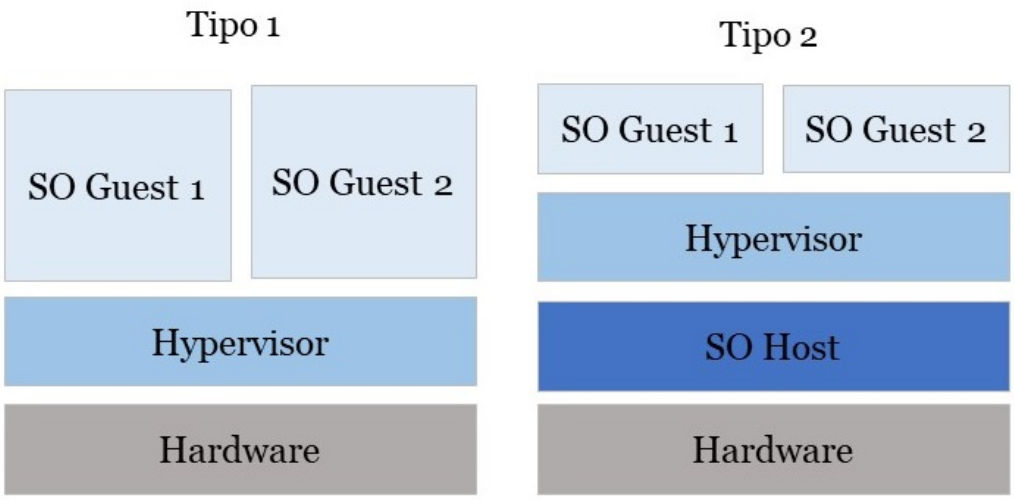

Figura 2.1: Virtualização baseada em hypervisor (figura redesenhada a partir de [Res19])

Conforme apresenta a figura 2.1, é possível diferenciar os dois tipos de hypervisors. Os hypervisors tipo 1 estão em execução diretamente no hardware onde o sistema operativo é instalado, por isso muitas vezes é referido como Bare metal, por este não requerer um sistema operativo e tem os seus próprios drivers, enquanto os hypervisors tipo 2 exigem um sistema operativo host cujos recursos são usados em ordem para realizar as suas operações [EK16].

Enquanto os hypervisors do tipo 1 ou bare-metal são executados diretamente no hardware de computação, os hypervisors hospedados ou tipo 2 são executados no sistema operativo da máquina host. Embora os hypervisors hospedados sejam executados no sistema operativo, sistemas operativos adicionais (e diferentes) podem ser instalados na parte superior do hypervisors. A desvantagem dos hypervisors hospedados é que a latência é maior do que os hypervisors bare-metal. Isso ocorre porque a comunicação entre o hardware e o hypervisor deve passar pela camada extra do sistema operativo. Os hypervisor hospedados às vezes são conhecidos como hypervisors de cliente porque são usados com mais frequência com utilizadores finais e testes de software [Vmw20]. É de referir que o tipo de hypervisor mais implementado é o tipo 1, estes são isolados do sistema operativo, torna-os mais seguros e menos sujeito a ataques. Além disso, apresentam um desempenho melhor e mais eficiente do que os hypervisors hospedados ou do tipo 2. [Vmw20].

Virtualização Baseada em Containers: A Virtualização baseada em container, também conhecida como virtualização no nível do sistema operativo, utiliza os recursos do kernel para criar um ambiente isolado para processos. A virtualização baseada em container fornece um nível diferente de abstração em termos de virtualização e isolamento quando comparada com hypervisors. Os hypervisors consomem muito hardware, o que resulta na sobrecarga em termos de virtualização de hardware e drivers de dispositivos virtuais. Em contraste, os containers implementam o isolamento dos processos no nível do sistema operativo, evitando assim a sobrecarga. Os containers partilham o mesmo kernel do sistema operativo da máquina host subjacente e um ou mais processos podem ser executados em cada container [EK16]. Os containers contêm um microsserviços ou aplicação e tudo que que precisam para serem executados. Tudo dentro de um container é preservado por imagem, um arquivo baseado em código que inclui todas as bibliotecas [Red20], a figura 2.2 ilustra virtualização baseada em container.

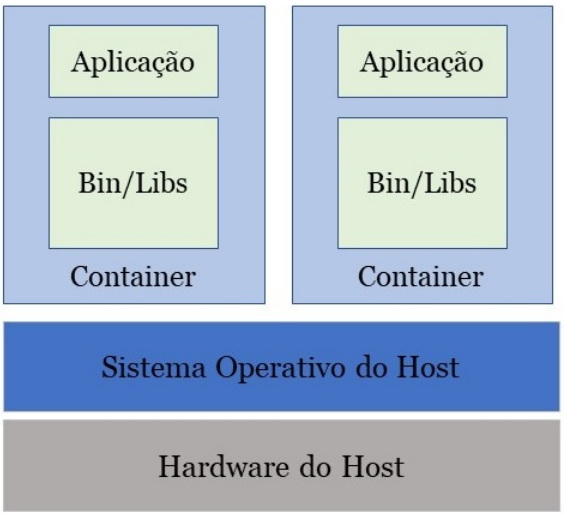

Figura 2.2: Virtualização baseada em Containers

Conforme ilustra a figura 2.2, a virtualização baseada em container é feita a nível do sistema operativo host, e não a nível de hardware. cada container partilha o kernel do sistema operativo host, este processo torna a virtualização baseada em container mais flexível e consume menos recursos. Várias soluções baseadas em container foram desenvolvidas, como *LXC*, *Googles lmctfy*, *Docker*. Um container é um sistema leve em execução dentro do sistema host. Os containers virtualizam o sistema operativo enquanto os hypervisores virtualizam os recursos de hardware. Portanto, a virtualização baseada em container é bem conhecida por consumir menos recursos e reduzir a sobrecarga. [SLC20], [ZLP16].

2.2.2 Docker

Docker é uma plataforma open source baseada em container para desenvolvimento, envio e execução de aplicações. A plataforma permite que o programador efetue a gestão da sua infraestrutura da mesma forma que pode ser feita em outros ambientes, mas tirando proveito das metodologias do Docker para enviar, testar e implantar código de forma rápida. Assim, reduzindo significativamente o atraso entre escrever o código e executá-lo na produção [Doc20a], [LBG18]. A plataforma permite que as aplicações sejam empacotadas como uma imagem Docker, um arquivo que inclui a aplicação, suas dependências e fundamentos de um sistema operativo para aplicações em execução [Li20].

O Docker oferece a capacidade de empacotar e executar aplicações num ambiente livremente isolado denominado container. O isolamento e a segurança permitem que o programador execute vários containers simultaneamente em um determinado host. Os containers são leves e contêm tudo o que é necessário para executar uma aplicação, sem precisar de outros softwares instalados no host [Doc20a]. Os containers Docker têm se mostrado bastante benéficos para a execução de aplicações. Embora não seja uma alternativa completa à máquina virtual, ainda fornecem isolamento para outras aplicações de container [LBG18].

A plataforma baseada em container do Docker permite cargas de trabalho altamente portáteis. Os containers do Docker podem ser executados em diferentes ambientes, incluindo em ambiente local, numa *data center*, em provedores de nuvem ou num ambiente mista. A portabilidade e a leveza do Docker também facilitam a gestão dinâmica de cargas de trabalho, aumentando ou diminuindo aplicações e serviços de acordo com as necessidades de negócio, quase em tempo real. A plataforma fornece uma alternativa viável e económica às máquinas virtuais baseadas em hypervisor, para que o programador possa usar mais de sua capacidade de computação para atingir os seus objetivos de negócio. O Docker é perfeito para ambientes de alta densidade e para implantação de pequeno e médio porte, onde se precisa fazer mais com menos recursos [Doc20a].

2.2.2.1 Arquitetura e Componentes

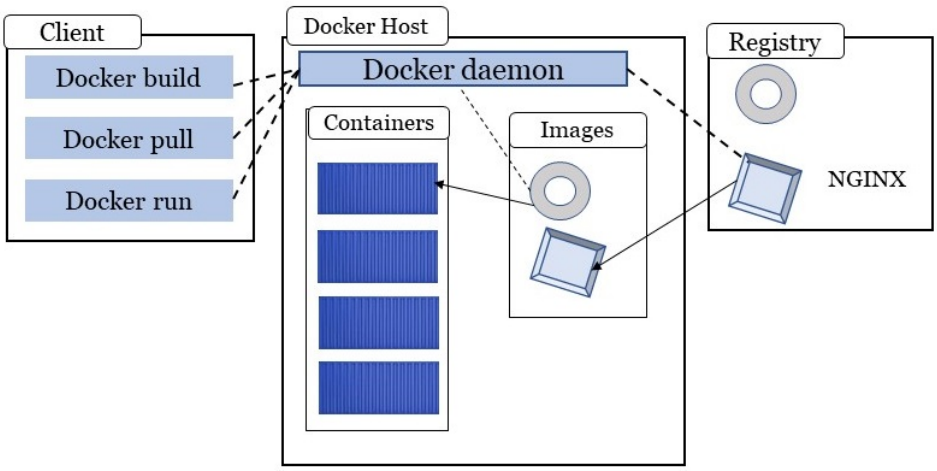

Figura 2.3: Arquitetura Docker (figura redesenhada a partir de [Doc20a])

O Docker segue uma arquitetura cliente-servidor conforme ilustra a figura 2.3, o Docker *client* comunica-se com o Docker *daemon*, que é responsável por construir, executar e distribuir os containers Docker. Os Docker *client* e o *daemon* podem ser executados no mesmo sistema, ainda assim, Docker *client* também pode ser usado para conetar um Docker daemon remoto do Docker. O Docker *client* se comunicam com *daemon* por meio de sockts ou usando REST API. Outro Docker client é o Docker Compose, que permite trabalhar com aplicações que consistem num conjunto de containers. [Doc20a]. A seguir a descrição das suas componentes:

- Docker Engine: É a parte central de todo o sistema Docker, esta componente é uma aplicação que segue uma arquitetura cliente-servidor, e é instalado na máquina host. Existem três componentes no Docker Engine [Doc20a]:
  - Servidor: É Docker daemon chamado *dockerd*, pode criar e gerir imagens Docker, containers, redes, entre outros.
  - API Rest: É usada para instruir o Docker *daemon* sobre a tarefa a efetuar.
  - Interface de linha de comando (CLI) : É um cliente usado para inserir comandos do Docker.
- Docker Client: É através desta componente que os utilizadores podem interagir com o Docker. Quando qualquer comando Docker é executado, o *client* os envia para o Docker *daemon*, que os executa. O Docker Client pode se comunicar com mais de um daemon [Doc20a].
- Docker Registries: É o local onde as imagens do Docker são armazenadas. *Docker Hub* é um registo público que pode ser usado, por padrão Docker está configurada para procurar as imagens neste registo. Também pode ser criada e executada imagem no registo privado [Doc20a].

- Docker objects: São várias entidades usadas para criar aplicações no Docker, a seguir a descrição de alguns destes objetos [Doc20a]:
  - Images: As imagens são o modelo instantâneo somente leitura usado para criar um container do Docker e essas imagens podem ser enviadas e retiradas de repositórios públicos ou privados. As imagens são leves, pequenas e rápidas em comparação com as imagens de máquina virtual. Também podem criar-se as próprias imagens usando o *Dockfile*, através de uma sintaxe que define as etapas necessárias para criar a imagem e executá-la. Cada instrução em um *Dockerfile* cria uma camada na imagem.
  - Containers: é uma instância executável de uma imagem. O utilizador pode criar, iniciar, parar, mover ou excluir um container usando a Docker API ou através da interface da linha de comando. Pode também conetar-se um container a uma ou mais redes, anexar armazenamento a ele ou até mesmo criar uma nova imagem com base no seu estado atual. O container é definido pela sua imagem, bem como por quaisquer opções de configuração fornecidas ao criá-lo ou iniciá-lo.
  - Rede: É uma passagem pela qual todos os containers isolados se comunicam. Entre as principais driver estão :
    - Bridge: Driver de rede padrão para um container.
    - Host: Este driver remove o isolamento de rede entre os containers do docker e o host do docker. É usado quando não se precisa de nenhum isolamento de rede entre o host e o container.
    - overlay: Este conecta vários Docker daemons e permite que os serviços de swarm se comuniquem entre si.
    - macvlan: Permite atribuir um endereço MAC *(Media Access Control)* a um container, fazendo com que ele apareça como um dispositivo físico numa rede.
    - none: Para este container, desative todas as redes. Normalmente é usado em conjunto com um driver de rede personalizado.
  - Volume: São mecanismos preferenciais para persistir os dados gerados e usados por containers Docker. Eles são completamente geridos pelo Docker por meio do Docker CLI ou Docker API e funcionam em containers Linux e Windows.

2.2.3 Kubernetes

Kubernetes, também conhecido como K8s, é uma plataforma open source para automatizar implantação, escalonamento e gestão de aplicações em containers [Kub20b]. O objetivo do seu desenvolvimento é projetar um mecanismo para configurar, implantar, gerir e manter containers de forma a tornar implantação das aplicações em containers mais fácil e eficiente. Atualmente, o Kubernetes tem sido amplamente usado na nuvem

pública, nuvem privada, nuvem híbrida e outros cenários devido à sua boa portabilidade e escalabilidade [SGKH20].

O Kubernetes permite a orquestração e a gestão de containers, utilizando uma unidade de programação básica chamada *Pods* (grupo de containers) que auxilia na implantação, dimensionamento e operação automatizada dos containers em clusters de hosts [SB19]. O Kubernetes resolve o problema de proliferação de containers agrupando-os em um pod, onde cada pod recebe o seu próprio endereço IP exclusivo nos clusters. Cada pod pode ser acedido de qualquer outro pod no cluster e cada container em um pod partilha recursos com outros containers no pod. No entanto, cada container em um pod não pode aceder a recursos diretamente ou um container de outro pod. Se um container em um pod exigir a capacidade de outro container em outro pod, o pod do solicitante precisará do endereço IP do pod do container que está a procurar. Portanto, via pods, o K8s facilita um nível mais alto de abstração ao agrupar como serviços em containers em um host local dentro de um pod, auxilia no agendamento de cargas de trabalho e fornece os serviços de rede e armazenamento necessários aos containers [SB19]. O Kubernetes tem sido bastante utilizado porque reduz os processos manuais repetitivos envolvidos na instalação e orquestração de container [IAR20]

2.2.3.1 Recursos do Kubernetes

O Kubernetes tornou-se a plataforma de orquestração padrão para containers. Os principais serviços de provedores de nuvem oferecem suporte, tornando a escolha lógica para organizações que buscam mover mais aplicações para a nuvem [Doc20b]. O Kubernetes fornece uma estrutura comum para executar sistemas distribuídos para que as equipas de desenvolvimento tenham uma infraestrutura consistente e imutável do desenvolvimento à produção para cada projeto. O Kubernetes pode gerir os requisitos de escalonamento, disponibilidade, failover, padrões de implantação e muito mais [Doc20b]. Alguns dos recursos do Kubernetes incluem[Kub20c]:

- Descoberta de serviço e balanceamento de carga: O Kubernetes pode expor um container usando o nome DNS ou seu próprio endereço IP. Se o tráfego para um container for alto, o Kubernetes pode balancear a carga e distribuir o tráfego de rede para que a implantação seja estável.

- Orquestração de armazenamento: O Kubernetes permite que o programador monte automaticamente um sistema de armazenamento de sua escolha, como armazenamentos locais, provedores de nuvem pública e outros.

- Lançamentos e reversões automatizadas: Permite que o programador descreve o estado desejado para seus containers implantados usando o Kubernetes, e a plataforma pode alterar o estado real para o estado desejado num ritmo controlado. Ainda assim, o programador pode automatizar o Kubernetes para criar novos containers para a sua implantação, remover os containers existentes e adotar todos os

seus recursos para o novo container.

- Autocorreção: O Kubernetes reinicia os containers que falham, substitui os containers, elimina os containers que não respondem à verificação de integridade definida pelo utilizador e não os anuncia aos clientes até que estejam prontos para servir.

- Gestão de configuração e de segredos: O Kubernetes permite armazenar e gerir informações confidenciais, como senhas, *tokens OAuth* e *chaves SSH*. O programador pode implantar e atualizar segredos e configuração de aplicações sem reconstruir as imagens de container e sem expor segredos na pilha de configuração.

2.2.3.2 Arquitetura

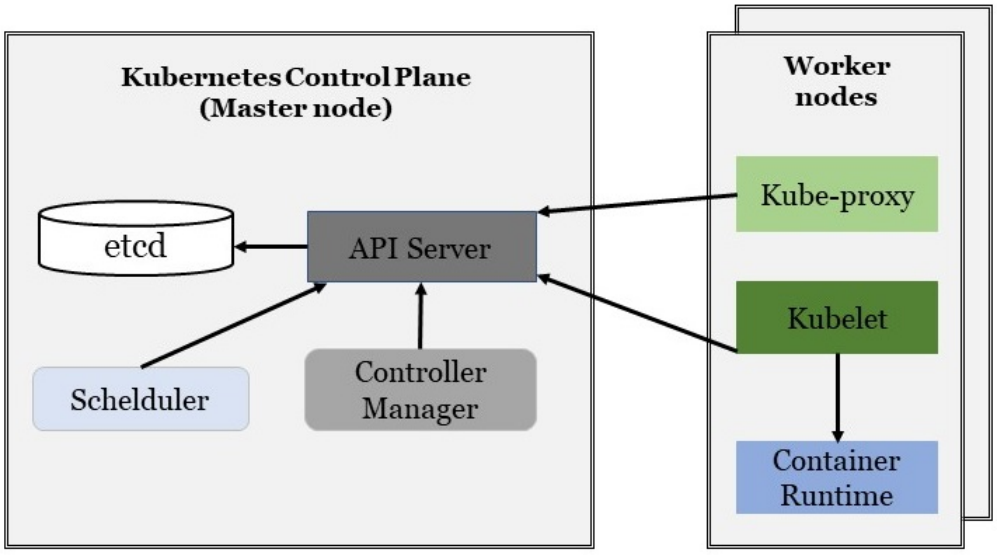

Figura 2.4: Arquitetura de Kubernetes (figura redesenhada a partir de [BIID20])

O Kubernetes oferece uma plataforma para programar e executar containers em clusters de máquinas físicas, virtuais ou na nuvem. A arquitetura do Kubernetes divide um cluster em componentes que trabalham em conjunto para manter o estado definido do cluster. Cada cluster Kubernetes consiste em apenas um master node, atuando como um plano de controlo e um ou mais worker node [FJFCSG+20]. A figura 2.4 apresenta arquitetura que consiste em um master node e um worker node. A seguir os detalhes de cada parte que constituem o cluster e com as suas respetivas componentes:

- Kubernetes Master nodes: Constituem essencialmente o cérebro do cluster, eles são responsáveis na gestão do cluster geral. Kubernetes nodes contêm os serviços necessários para executar aplicações em componentes chamados de pods. Os principais componentes do master node são categorizados da seguinte forma [KPR20]:

  - Etcd Storage: É um meio de armazenamento de valor-chave usado para armazenar dados de pods existentes no cluster Kubernetes. Em sua implementação, o etcd pode ser dividido em dois, ou seja, etcd interno e etcd externo. A implementação do etcd externo requer a criação de um cluster etcd que ofereça suporte à criação de clusters Kubernetes de alta disponibilidade. Uma das vantagens de implementar o etcd externo é que, embora o cluster Kubernetes tenha sido redefinido, os dados do cluster Kubernetes anterior ainda podem ser reutilizados.
  - API Server: É um componente que expõe a API Kubernetes e é um front-end do master node. Um dos usos do Kube-APIserver é aumentar horizontalmente os recursos necessários ao serviço Kubernetes. Além disso, a API Kubernetes também funciona para ver uma lista de serviços compatíveis com o cluster Kubernetes e gerir o serviço.
  - Scheduler: É uma componente que funciona para observar pods, criando no cluster Kubernetes e não foi atribuído ao worker node e, em seguida, seleciona um worker node para executar serviços de pod. O Scheduler presta atenção a vários fatores ao colocar pods em worker nodes, como requisitos de recursos, especificações de afinidade, localização de dados e carga de trabalho dos worker nodes.
  - Controller Manager: Executa vários processos de controlador distintos como controlador de replicação, controlador de terminal, controlador de nó, conta de serviço e controlador de token. As funções desses nós são:
    * controlador de nó: Fornecer resposta se o número e a prontidão do nó forem reduzidos.
    * Controlador de replicação: Mantém o número de pods para corresponderem ao número de necessidades de cada objeto controlador de replicação que está no sistema.
    * Conta de serviço e controlador de token: Carregar contas e acesso de token de API em cada namespace criado.

- Worker node: Worker Node também conhecido como Kubernetes node contém as informações para gestão da rede entre containers, como Docker, e a comunicação entre o master node, atribuindo os recursos aos containers de acordo com a programação, os seus componentes são [BIID20], [Mar19a]:

  - Kubelet: Esta componente funciona como principal agente e mecanismo de comunicação dos nodes no cluster. Controla o estado de execução e a verificação de integridade dos containers do pod no node [SGKH20].
  - Kube-proxy: Tem como tarefa manter regras de rede no cluster e garantir a disponibilidade dos serviços de container em execução no cluster [SGKH20].
  - Container: Os containers são o nível mais baixo de microsserviços colocados dentro do pod e precisam de endereço IP externo para visualizar o processo externo.

2.3 Computação na Nuvem

Com o desenvolvimento das tecnologias de processamento e armazenamento e o sucesso da Internet, os recursos de computação tornaram-se mais baratos, mais poderosos e mais onipresentes do que nunca. Esta tendência tecnológica permitiu a realização de um modelo de computação denominado computação em nuvem, no qual os recursos como CPU e armazenamento são fornecidos como utilitários gerais que podem ser facilitados e libertados pelos utilizadores através da Internet de forma sob procura. Num ambiente de computação em nuvem, a função tradicional de provedor de serviços é dividida em duas: os provedores de infraestrutura que efetuam a gestão de plataformas de nuvem e alugam recursos de acordo com um modelo de preços baseado no uso, e provedores de serviços, que alugam recursos de um ou vários provedores de infraestrutura para servir aos utilizadores finais. O surgimento da computação em nuvem teve um impacto tremendo no setor de Tecnologia da Informação (TI) nos últimos anos, onde grandes empresas como *Google*, *Amazon* e *Microsoft* se esforçam para fornecer plataformas de nuvem mais poderosas, confiáveis e económicas, e as empresas buscam reformular os seus modelos de negócio para obter benefícios a partir deste paradigma [ZCB10].

A computação em nuvem pode ser entendida como um modelo que permite acesso ubíquo, conveniente e sob procura à rede a um pool partilhado de recursos de computação configuráveis como redes, servidores, armazenamento, aplicações e serviços, que podem ser rapidamente provisionados e libertados com mínimo esforço de gestão ou interação do provedor de serviços. Este modelo de nuvem é composto por cinco características essenciais, três modelos de serviço e quatro modelos de implantação [Pet11] que serão descritas mais em diante.

2.3.1 Arquitetura da Computação em Nuvem

De um modo geral, a arquitetura de uma computação em nuvem ambiente pode ser dividida em 4 camadas: camada de hardware ou datacenter, a camada de infraestrutura, camada de plataforma e a camada de aplicação conforme ilustra a figura 2.5. A seguir a descrição de cada uma delas[ZCB10]:

- Camada de hardware: É responsável por gerir os recursos físicos da nuvem, incluindo físicos servidores, router, switches, sistemas de energia entre outros. Na prática, a camada de hardware é normalmente implementada em data centers. Um data center geralmente contém milhares de servidores que são organizados em racks e interligados por meio de switches, roteadores ou outras malhas. Problemas típicos na camada de hardware incluem configuração de hardware, tolerância a falhas, gestão de tráfego e recursos de energia.
- Camada de infraestrutura: Também conhecida como camada de virtualização, a camada de infraestrutura cria um pool de armazenamento e recursos de computação

particionando os recursos físicos usando tecnologias de virtualização como *KVM* e *VMware*. A camada de infraestrutura é um componente essencial da computação em nuvem, uma vez que muitos recursos, como atribuição dinâmica de recursos, são apenas disponibilizados por meio de tecnologias de virtualização.

- Camada de plataforma: construída sobre a camada de infraestrutura, a camada de plataforma consiste em sistemas operativos e frameworks de aplicações. O objetivo da camada de plataforma é minimizar a carga de implantação de aplicações diretamente em containers de VM. Por exemplo, o *Google App Engine* opera na camada da plataforma para fornecer suporte de API para a implementação de armazenamento, base de dados e lógica de negócios de aplicações da web típicos.

- A camada de aplicação: No nível mais alto da hierarquia, a camada de aplicação consiste nas aplicações em nuvem reais. Diferente de aplicações tradicionais, as aplicações em nuvem pode aproveitar o recurso de escalonamento automático para alcançar melhor desempenho, disponibilidade e menor custo operacional. Em comparação com os ambientes tradicionais de hospedagem de serviços como farms de servidores dedicados, a arquitetura da nuvem a computação é mais modular. Cada camada é fracamente acoplada com as camadas acima e abaixo, permitindo que cada camada evoluir separadamente. Isso é semelhante ao design do modelo OSI para protocolos de rede. A modularidade arquitetónica permite que a computação em nuvem suporte uma ampla gama de aplicações de requisitos enquanto reduz a gestão e a manutenção a sobrecarga.

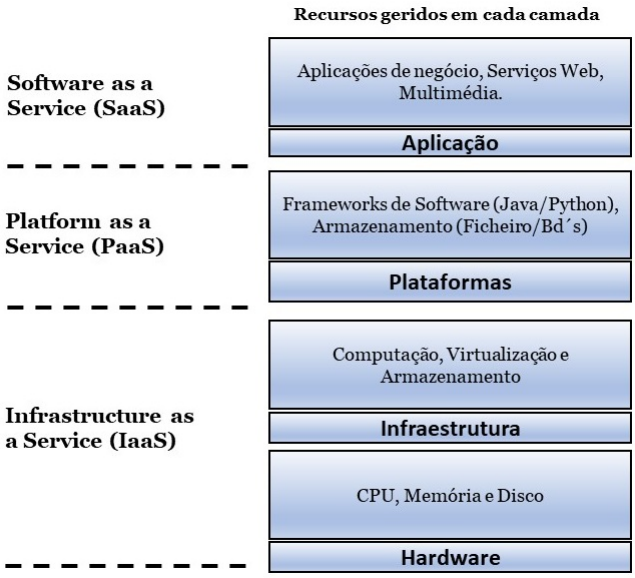

Figura 2.5: Arquitetura por Camada de Computação na Nuvem e Modelo de Serviço (figura redesenhada a partir de [ZCB10])

2.3.2 Caraterísticas

As caraterísticas da Computação em Nuvem definem a sua usabilidade e eficiência, abaixo são apresentadas as caraterísticas essências desta tecnologia:

- Auto-serviço sob procura: O cliente pode provisionar recursos de computação unilateralmente. Esses recursos incluem hora do servidor e armazenamento em rede. Eles podem ser provisionados sempre que necessário, sem interação humana com os provedores de serviços[Pet11].
- Amplo acesso à rede: Existem recursos disponíveis na rede e podem ser acedidos por meio de várias plataformas, como telemóveis, laptops, PDAs e outros dispositivos eletrónicos.
- Pooling de recursos: Os recursos de computação do provedor são agrupados para atender a múltiplos clientes usando um modelo multi-tenant, com diferentes recursos físicos e virtuais atribuídos e reatribuídos dinamicamente de acordo com a procura do cliente.
- Elasticidade: Os recursos computacionais podem ser provisionados e libertados de forma elástica, em alguns casos de forma automática, para escalar rapidamente para fora e dentro de acordo com a procura. Para o cliente, os recursos disponíveis para provisionamento muitas vezes parecem ser ilimitados e podem ser apropriados em qualquer quantidade e momento.
- Serviço controlado: Os sistemas em nuvem controlam e otimizam automaticamente o uso de recursos, potencializando uma capacidade de medição em algum nível de abstração apropriado para o tipo de serviço (por exemplo, armazenamento, processamento, largura de banda e contas de utilizadores ativas). O uso de recursos pode ser monitorizado, controlado e relatado, fornecendo transparência tanto para o provedor quanto para o cliente do serviço utilizado.

2.3.3 Modelos de Serviços de Computação em Nuvem

A computação em nuvem emprega um modelo de negócios orientado a serviços. Em outras palavras, os recursos de hardware e de nível de plataforma são fornecidos como serviços sob procura. Conceitualmente, cada camada da arquitetura descrita na secção anterior pode ser implementada como um serviço. Estes modelos de serviços podem ser classificados em três categorias: Software-as-a-Service (SaaS), Platform-as-a-Service (PaaS) e Infrastructure-as-a-Service (IaaS). A seguir a descrição de cada uma delas:

- Software-as-a-Service(SaaS) - O recurso fornecido ao consumidor é usar as aplicações do provedor em execução numa infraestrutura em nuvem. As aplicações são acedidas a partir de vários dispositivos clientes por meio de uma interface de utilizador, como um navegador da web (por exemplo, e-mail baseado na web), ou uma interface de programa. O cliente não gere ou controla a infraestrutura de nuvem subjacente, incluindo rede, servidores, sistemas operativos, armazenamento ou

mesmo recursos de aplicações individuais, com a possível exceção de definições de configuração de aplicações específicas do utilizador [Pet11].

- Platform-as-a-Service (PaaS): Refere-se ao fornecimento de ferramentas sob procura para desenvolver, testar, entregar e gerir aplicações de software. PaaS oferece uma estrutura para os programadores e arquitetos de TI criarem aplicações web ou móveis escaláveis [Kal21]. O cliente não efetua a gestão da infraestrutura de nuvem subjacente, incluindo rede, servidores, sistemas operativos ou armazenamento, mas tem controlo sobre as aplicações implantadas e possivelmente as definições de configuração para o ambiente de hospedagem de aplicações [Pet11].

- Infrastructure-as-a-Service (IaaS) - O recurso fornecido ao cliente é provisionar processamento, armazenamento, redes e outros recursos de computação fundamentais onde o cliente é capaz de implantar e executar software arbitrário, que pode incluir sistemas operativos e aplicações. O cliente não gere nem controla a infraestrutura de nuvem subjacente, mas tem controle sobre os sistemas operativos, armazenamento e aplicações implantados [Pet11].

2.3.4 Modelo de Deploy da Computação na Nuvem

Os modelos de implantação define o tipo de acesso à nuvem e todos os modelos de implantação de nuvem são baseados no mesmo princípio de virtualização (abstração de recursos de hardware bare metal), mas diferem em termos de localização, capacidade de armazenamento, acessibilidade e entre outros. Estes modelos podem ser classificados em nuvem privada, nuvem pública, nuvem comunitária e nuvem híbrida, a seguir a descrição de cada uma delas:

- Nuvem privada: A infraestrutura de nuvem é fornecida para uso exclusivo por uma única organização composta por vários clientes. Pode ser de propriedade, gerida e operada pela organização, um terceiro ou alguma combinação destas, e pode existir dentro ou fora das instalações [Pet11].

- Nuvem Pública: A infraestrutura da nuvem é fornecida para uso aberto pelo público em geral. Pode ser de propriedade, gerida e operada por uma organização empresarial, académica ou governamental, ou alguma combinação destas. Existe nas instalações do provedor de nuvem [Pet11]. Num ambiente de nuvem pública, os recursos são partilhados entre vários utilizadores. O custo de uso dos serviços de nuvem é determinado pelo uso dos recursos de TI consumidos [Kal21].

- Nuvem Comunitária: A infraestrutura da nuvem é fornecida para uso exclusivo por uma comunidade específica de clientes de organizações que partilhem ou tenham o mesmo objetivo comum [Pet11]. Esta infraestrutura de nuvem é construída após a compreensão das necessidades de computação de uma comunidade, pois há muitos fatores, incluindo conformidade e políticas de segurança, que precisam ser incluídos na infraestrutura de nuvem da comunidade [Kal21]. Esta pode ser de propriedade,

administrado e operado por uma ou mais organizações na comunidade, um terceiro, ou alguma combinação deles, e pode existir dentro ou fora das instalações [Pet11].

- Nuvem Híbrida: A infraestrutura deste tipo é composta por duas ou mais infraestruturas de nuvem distintas (privada, comunidade ou pública) que permanecem entidades únicas, mas são unidas por tecnologia padronizada ou proprietária que permite a portabilidade de dados e aplicações [Pet11]. As organizações usam principalmente a nuvem híbrida quando a sua infraestrutura local precisa de mais escalabilidade, então elas fazem uso da escalabilidade na nuvem pública para atender às procuras de negócios flutuantes. As organizações podem manter seus dados confidenciais na sua nuvem privada ao colher o poder da nuvem pública [Kal21].

2.4 Computação Sem Servidor

Computação sem servidor (ou simplesmente sem servidor) está a emergir como um novo e atraente paradigma para a implantação de aplicações em nuvem, em grande parte devido à recente mudança de arquiteturas de aplicações organizacionais para containers e microserviços [BCC+17]. A computação sem servidor refere-se ao conceito de construção e execução de aplicações que não requerem a gestão de servidor [CNC]. A computação sem servidor é um modelo de computação em nuvem com o objetivo de abstrair a gestão de servidores e as decisões de infraestrutura de baixo nível dos programadores. Fornece um serviço real com pagamento de acordo com o uso, sem desperdício de recursos e reduz a barreira para os programadores, confiando no provedor de nuvem para lidar com todas as suas complexidades operacionais. Em comparação com outros modelos de computação em nuvem, a computação sem servidor está mais próxima das expectativas originais de que a computação em nuvem seja tratada como um serviço utilitário [ZLL21]. A computação sem servidor destaca-se como um paradigma de computação que tem sido amplamente adotado pela indústria para processamento escalável baseado em eventos em nuvens computacionais abstratas [NRdA+20].

A computação sem servidor não significa que não são usados servidores para hospedar e executar funções; nem significa que os utilizadores de operações deixam de ser necessários. Em vez disso, referem-se à ideia de que os clientes de computação sem servidor já não precisam de perder tempo, em provisionar recursos de servidor, manutenção, atualizações e dimensionamento. Em vez disso, todas essas tarefas e recursos são tratados por uma plataforma sem servidor e são completamente abstraídas dos utilizadores [CNC] .

2.4.1 Modelos De Computação em Nuvem Sem Servidor

Cloud Native Comptuting Foundation (CNCF) inclui as soluções de computação Sem Servidor em duas categorias de modelos de serviços de nuvem: Function-as-a-Service (FaaS) e Backend-as-a-Service (BaaS) [CNC]. A seguir a descrição das duas categorias mencionadas.

Functions-as-a-Service (FaaS): Functions-as-a-Service, que normalmente é baseado em eventos, onde programadores executam e gerem o código de aplicações com funções que são accionados por eventos ou solicitações HTTP. Os programadores implantam pequenas unidades de código para o FaaS, que são executados conforme necessário como ações discretas, a escalonarem sem a necessidade de gerir servidores ou qualquer outra infraestrutura subjacente [CNC]. o modelo FaaS proporciona maior controle aos programadores, possibilitando a criação de aplicações personalizadas, evitando a dependência de uma biblioteca de serviços pré-gravados [Red]. O tempo de execução das funções normalmente é limitado, e as mesmas não estão constantemente ativas. Em vez disso, como plataformas FaaS detetam eventos que instanciam as funções. Portanto, as funções devem ser acionadas por eventos, como solicitações de clientes, eventos necessários por quaisquer sistemas externos, fluxos de dados ou outros. O provedor FaaS é então responsável por escalar horizontalmente as execuções de funções em resposta ao número de eventos de entrada [NT20].

Backend-as-a-Service (BaaS): São serviços baseados em API de terceiros que substituem subconjuntos principais de funcionalidade numa aplicação. Essas APIs são fornecidas como um serviço com escalonamento automático e operação transparente, esta parece ao programador não ter servidor [CNC]. Exemplos são sistemas de autenticação remota, gestão de banco de dados, armazenamento em nuvem e hospedagem. Um exemplo de BaaS pode ser o *Google Firebase*, um base de dados totalmente gerido que pode ser usado diretamente a partir de uma aplicação [NT20].

2.4.2 Arquitetura da Computação Sem Servidor

A arquitetura sem servidor é uma forma de abordar a computação em nuvem na qual os provedores de serviços em nuvem efetuam a gestão de alocação de recursos do servidor de forma dinâmica. Os processos são executados isoladamente e são disparados quando certos triggers ou eventos acontecem; os recursos necessários para executar o processo estão normalmente sob a gestão do provedor de nuvem. Ao combinar serviços em nuvem de terceiros, a lógica do lado do cliente e a capacidade de solicitar serviços baseados em nuvem com uma variedade de triggers, a arquitetura sem servidor oferece o que muitas vezes é chamado de Functions-as-a-Service (FaaS) [Tho18]. A capacidade principal de uma plataforma sem servidor é a de um sistema de processamento de eventos, conforme ilustrado na Figura 2.6. O serviço deve efetuar a gestão de um conjunto de funções definidas pelo utilizador (também chamado de ações), receber um evento enviado por HTTP ou recebido de uma fonte de evento (também chamado de triggers), a determinar qual função para deve lidar com o evento, encontrar uma instância existente da função ou criar uma nova instância, enviar o evento para a instância da função, esperar por uma resposta, recolher logs de execução, disponibiliza a resposta para o utilizador e interrompem a função quando não for mais necessário [BCC+17].

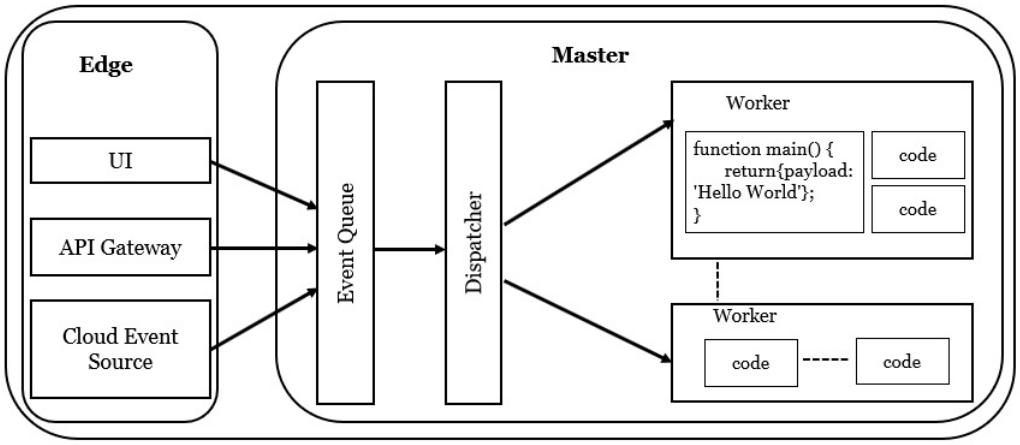

Figura 2.6: Arquitetura Sem Servidor (figura redesenhada a partir de [BCC+17])

2.4.3 Plataformas Sem Servidor

Esta secção descreve de forma breve sobre as plataformas sem servidor, incluindo as plataformas Comercias e as Plataformas Open Source.

2.4.3.1 Plataformas Comerciais

Atualmente os principais provedores de nuvem pública oferecem pelo menos uma solução de serviços sem servidor. Eles incluem a Amazon Web Services (AWS) com o AWS Lambda, o Microsoft Azure com o Azure Functions, o Google Cloud com Google Cloud Functions e a IBM Cloud com o IBM Cloud Functions [Red], a seguir a descrição de cada plataforma:

- AWS Lambda: Uma oferta de FaaS que pertence à Amazon Web Services que foi introduzido em 2014, a plataforma permite executar código em resposta a eventos e efetuar automaticamente a gestão dos recursos computacionais subjacentes [LKW+18]. A AWS Lambda foi a primeira plataforma sem servidor e definiu várias dimensões-chave, incluindo custo, modelo de programação, implantação, limites de recursos, segurança e monitorização [BCC+17]. A plataforma pode ser usada para estender outros serviços da AWS com lógica personalizada ou desenvolvimento de serviços de back-end que operam com a escala, o desempenho e a segurança da AWS. O AWS Lambda pode executar automaticamente código em resposta a vários eventos, como solicitações HTTP por meio do *Amazon API Gateway*, modificações de objetos em *buckets* do *Amazon S3*, atualizações de tabela no *Amazon DynamoDB* e transições de estado no *AWS Step Functions* [AWS]. O Lambda executa o seu código em infraestrutura de computação de alta disponibilidade e realiza toda a gestão dos recursos computacionais, incluindo manutenção de servidor, sistema operativo, provisionamento de capacidade e escalabilidade automática, implantação de código e de patch de segurança e monitorização e registo de funções, o utilizador apenas precisa de implantar funções [AWS]. O código executado no AWS Lambda é chamado de *Lambda function*. Assim que uma função é criada, a mesma

está pronta para execução. Cada função inclui seu código e algumas informações de configuração associadas, incluindo o nome da função e os requisitos do recurso. As funções do Lambda são "stateless", sem nenhuma afinidade com a infraestrutura adjacente [AWS]. A plataforma monitoriza automaticamente as funções do Lambda, recolhendo métricas por meio do Amazon *CloudWatch*. Regista todas as solicitações tratadas por sua função e também armazena automaticamente os logs produzidos por funções por meio do Amazon *CloudWatch Logs*. O Lambda oferece suporte nativamente aos códigos Java, Go, PowerShell, Node.js, C#, Python e Ruby, bem como uma API de tempo de execução que permite usar qualquer linguagem de programação adicional para criar as funções. A AWS Lambda é compatível com funções de empacotamento e implantação como imagens de container, facilitando a criação de aplicações baseadas em Lambda usando ferramentas, fluxos de trabalho e dependências de imagens de container [AWS].

- Google Cloud Functions: A Plataforma foi lançada pela Google em fevereiro de 2016, projetado principalmente para os serviços do Google Cloud. O Google cita uma série de casos de uso específicos para o Google Cloud Functions incluindo back-end móvel, APIs e desenvolvimento de microsserviços, processamento de dados, webhooks (para responder a acionadores de terceiros), IoT, entre outros [LRLE17]. A plataforma Google Cloud Functions permite criar, desenvolver e efetuar o deploy de funções e aplicações, como código-fonte ou containers, simplificando o esforço do programador, sendo que toda gestão da infraestrutura fica a cargo do provedor de serviço da nuvem. O Cloud Functions é executado em um ambiente sem servidor sob a gestão da Google, em que o Google processa totalmente a infraestrutura, sistemas operativos e ambientes de execução. Cada função do Cloud é executada no próprio contexto de execução seguro e isolado, com dimensionamento automático e um ciclo de vida independente de outras funções [Goo]. O Cloud Functions implementa o paradigma sem servidor, onde o utilizador executa o seu código sem se preocupar com a infraestrutura subjacente, como servidores e máquinas virtuais. Para permitir que o Google faça a gestão e escala de forma automática as funções, elas precisam ser sem estado [Goo]. O Cloud Functions pode aceder à maioria dos principais serviços do Google Cloud Platform por meio de bibliotecas de cliente de APIs de linguagens específicas ou através REST APIs. Atualmente, o Cloud Functions oferece suporte a eventos dos seguintes provedores: *HTTP*, *Cloud Storange*, *Cloud Pub/Sub*, *Cloud Firestore*, entre outros. A plataforma suporta várias linguagens com o ambiente de execução, Node.js, Python, Go, C#Java, Ruby [Goo].

- Microsoft Azure Functions: A plataforma Microsoft Azure Functions foi disponibilizada numa versão limitada em março de 2016 e o lançamento completo em novembro de 2016. A plataforma foi projetada para estender a plataforma de aplicação Azure *(Azure Application Platform)* existente com recursos para implementação de funções orientado a eventos que ocorrem no Azure ou serviço de terceiros, bem como em ambiente on-primises [LRLE17]. Azure Function é um serviço de nuvem disponível a pedido, que fornece todos os recursos e infraestrutura continu-

amente atualizados de que precisa para executar as aplicações, onde o programador concentra-se nos pedaços de código que mais o interessam, e as funções tratam o resto. As funções fornecem uma computação sem servidor para o Azure, onde as mesmas podem construir APIs web, responder a alterações de base de dados, processar streams de IoT, gerir filas de mensagens, entre outros [Azu21a]. A plataforma fornece computação sob procura de duas formas significativas: em primeiro lugar, o Azure Functions permite-lhe implementar a lógica do seu sistema em blocos de código facilmente disponíveis. Estes blocos de código são chamados de *funções*. Diferentes funções podem ser executadas sempre que precisar para responder a eventos críticos. Em segundo lugar, à medida que os pedidos aumentam, as *Azure Functions* satisfazem o pedido com o maior número de recursos e instâncias de função que for necessário. À medida que os pedidos caem, quaisquer recursos extras e instâncias de aplicação caem automaticamente [Azu21a]. A plataforma suporta um vasto leque de tipos de triggers. Eis alguns dos tipos mais comuns: *Timer*, *HTTP*, *Blob*, *Fila*, *Azure Cosmos DB* e *Hub de Event*. A plataforma suporta várias linguagens como C#, JavaScript, F#, Java, PowerShell, Python [Mic19], ainda assim, é possível um manipulador personalizado através do qual podem criar-se funções em linguagens que não são oficialmente suportadas. A plataforma oferece integração incorporada com a *Azure Application Insights* para monitorizar funções, a ferramenta recolhe dados de registo, desempenho e erro, e são concebidas para ajudar a melhorar continuamente o desempenho e a usabilidade das funções [Azu21a] .

- IBM Cloud Functions: É uma plataforma sem servidor, orientada a eventos que executa aplicações em resposta a eventos ou chamadas diretas de aplicações web ou móveis. É considerada uma plataforma de programação poliglota de funções como serviço da IBM para o desenvolvimento de código leve que é executado de forma escalável sob procura. IBM Cloud Functions é baseado no OpenWhisk, um projeto open source que combina componentes como NGINX, Kafka, Docker e CouchDB para formar um serviço de programação baseado em eventos sem servidor. A plataforma está sob a orientação da *Apache Foundation*, onde qualquer programador pode contribuir com seu código de ação como blocos de construção para o repositório em expansão. Isso significa que o Apache OpenWhisk está em constante crescimento e aprimoramento. O Apache OpenWhisk acelera o desenvolvimento de aplicações, o que permite que se desenvolve aplicações de forma rápida com sequências de ação. A plataforma inclui acesso a serviços cognitivos como APIs do IBM Watson dentro do fluxo de trabalho de event-trigger-action, torna a análise cognitiva dos dados de aplicação inerentes aos seus fluxos de trabalho [SS18] [IBM20c]. O IBM Cloud Functions acelera o desenvolvimento de aplicações, o que permite que aos programadores desenvolverem aplicações de forma rápida com sequências de ação que são executadas em resposta orientada a eventos. A plataforma é diferente das tecnologias de computação tradicionais, apenas é paga pelo tempo que o código atender às requisições. Isso significa que o utilizador pode ver uma economia de custo considerável em relação a outras tecnologias, como VMs e con-

tainers. Os custos aumentam apenas à medida que utilizador executa mais carga no Cloud Functions, a fatura é baseada no tempo e na memória necessários para executar funções. Uma ação do Cloud Functions é um pedaço de código que executa uma tarefa específica. Uma ação pode ser escrita na linguagem de preferência do programador, pode fornecer-se ação para Cloud Functions como código-fonte ou uma imagem Docker. As ações podem ser acionadas a partir de eventos em serviços próprios ou diretamente por meio da API REST, as ações também podem responder automaticamente a eventos de serviços IBM Cloud e serviços de terceiros usando um trigger. O código poder ser desenvolvido numa interface baseada na web do IBM Cloud ou com seu IDE de desenvolvimento próprio e usando o Cloud Functions CLI para carregá-lo. As funções oferecem suporte a várias linguagens como GO, Node.js PHP, Java, Python, Ruby, Swift, .NET, e fornece container docker para desenvolver em qualquer linguagem compilada [IBM20c].

A tabela 2.1, apresenta algumas comparações das caraterísticas de alguns recursos das plataformas acima mencionadas. Pode notar-se que todas as plataformas suportam várias de linguagens que permitem a escrita das funções. Quanto à alocação de memória que uma função pode utilizar, IBM Cloud Functions apresenta a quantidade de limite inferior em relação às restantes plataformas. O tempo de limite a qual uma função invocada pode estar em execução, Lambda apresenta o limite superior. De referir que relativamente à limitação de memória e tempo de execução, as plataformas mencionadas têm os seus valores padrão.

Table 2.1: Comparação de Alguns Recursos das Plataformas Públicas.

Recursos	**AWS Lambda**	**Google Cloud Functions**	**Microsoft Azure Functions**	**IBM Cloud Functions**
Linguagens Suportadas	Java, Go, PowerShell, Node.js, C, Python e Ruby	Node.js, Python, Go, C, Java, Ruby	C#, JavaScript, F#, Java, PowerShell, Python	GO, Node.js PHP, Java, Python, Ruby, Swift, .NET
Memória máxima e tempo de limite por função	10,240 MB, 900 segundos [AWS21]	4096 MB, 540 segundos [Clo21]	1536 MB, 600 segundos dependendo de tipo de plano [Azu21b]	2048 MB, 600 segundos [IBM21b]
Escalabilidade	Automática	Automática	Manual / Automática	Automática
Monitorização	CloudWacth	Google Stackdriver	Azure Application Insights	IBM Cloud Function Dashboard

2.4.3.2 Plataforma Open Source

As Plataformas sem servidor open source podem ser uma opção caso se pretenda evitar a dependência de serviços de provedor de nuvem pública e diminuição de custos. A seguir a descrição de algumas destas plataformas:

- OpenFaaS : é uma plataforma sem servidor que permite que os programadores implantem funções orientadas a eventos e microservices no Kubernetes sem programação repetitiva e padronizada. A plataforma pode ser implementada em qualquer ambiente, incluindo numa nuvem pública ou privada. O programador empacota o código ou um binário existente numa imagem Docker para obter um endpoint altamente escalonável com escalonamento automático e métricas. O OpenFaaS pode ser implementado em uma variedade de orquestradores de containers: Kubernetes, OpenShift, Docker Swarm ou em um único host com faasd. As funções do OpenFaaS podem ser acionadas facilmente por qualquer tipo de evento, onde um pedaço de código é convertido da origem do evento e acionado a função usando a API Gateway de OpenFaaS, o caso de uso mais comum é o HTTP. Ainda assim há outros triggers: Cron, OpenFaaS CLI, Streaming Async / NATS e outras fontes de eventos [Ope20a]. A plataforma usa a ferramenta OpenFaaS CLI é usada para o programador efetuar a implantação de funções no OpenFaaS. Apenas a função e o manipulador devem ser fornecidos pelo programador, e CLI manipula o empacotamento da função em um container do Docker. O container compreende uma função watchdog, ou seja, um servidor web que atua como um ponto de entrada para chamadas de função dentro da plataforma. Um API Gateway fornece uma interface externa para as funções, recolhe métricas e lida com o dimensionamento interagindo com o plug-in do orquestrador de container [MPD18]. O Watchdog efetua a gestão de encaminhamento das requisições para as funções definidas pelo utilizador e também implementa tarefas de verificação de integridade [BMS20]. O *Prometheus* recolhe as métricas que estão disponíveis por meio de API Gateway e que são usadas para escalonamento automático e essas podem ser visualizadas com o *Grafana*. O Kubernetes é a ferramenta de orquestração de container recomendada para efetuar o deploy com o OpenFaaS , seja um ambiente local, um cluster auto-hospedado ou com um serviço gerido, como o AWS Elastic Kubernetes Service (EKS). Os serviços e funções podem ser escritos em qualquer linguagem com *Template Store* tais como Node.js, Python , Go, Java, incluindo *Dockerfile* [Ope20a].

- Fission : É uma plataforma open source que fornece uma arquitetura sem servidor sobre o Kubernetes. Uma das vantagens do Fission é que a plataforma cuida da maioria das tarefas de escalonamento automático de recursos no Kubernetes, dispensando o programador sobre a gestão manual dos recursos. A segunda vantagem, o utilizador não fica vinculado a um provedor e pode mover-se livremente de um para outro, desde que eles ofereçam suporte a cluster Kubernetes [Rib20]. O Fission tem três conceitos principais: funções, ambientes e triggers. Uma função de Fission é algo que a Fission executa. Geralmente é um módulo com um ponto

de entrada e esse ponto de entrada é uma função com uma determinada interface. Ambientes são as partes específicas da linguagem do Fission. Um ambiente contém apenas software suficiente para construir e executar uma função de Fission. Como o Fission invoca funções por meio de HTTP, isso significa que o tempo de execução de um ambiente é um container com um servidor HTTP e, geralmente, um carregador dinâmico que pode carregar uma função. Alguns ambientes também contêm container de construtor, que cuidam da compilação e recolhe de dependências. As funções são chamadas na ocorrência de um evento; um Trigger é o que configura a Fission para usar esse evento de forma a invocar uma função. Fission suporta vários triggers, incluem HTTP Triggers, Timer Triggers, Message Queue Triggers e Kubernetes Watch Triggers [Fis20]. O Fission é extensível a qualquer linguagem; o núcleo é escrito em Go e as partes específicas da linguagem são isoladas em algo chamado *ambientes*, onde contém apenas software suficiente para construir e executar uma função de Fission [Fis20]. O Fission atualmente oferece suporte a Node.js, Python, Ruby, Go, PHP, Bash e qualquer executável Linux [Fis20]. A plataforma Fission expõe métricas com Prometheus por padrão, que podem ser prontamente copiadas e usadas com um servidor *Prometheus* e visualizadas com o *Grafana*. Essas métricas ajudam na monitorização do estado das funções, bem como os componentes do Fission [Fis20].

- Kubeless: É uma plataforma sem servidor open source nativa do Kubernetes que permite implantar funções sem a necessidade de se preocupar com a infraestrutura subjacente. A plataforma foi projetada para ser implantada em um cluster do Kubernetes e aproveitar todas as vantagens de todos os recursos primitivos do Kubernetes [Kub20a]. O modelo de programação Kubeless é baseado em três primitivas: funções, trigger e runtime. Uma função é uma representação do código a ser executado e o trigger é uma fonte de evento. O trigger pode ser associado a uma única função ou a um grupo de funções, dependendo do tipo de fonte do evento. O Kubeless garante que as funções associadas sejam invocadas pelo menos uma vez. Runtime representa uma linguagem e um ambiente específico de tempo de execução no qual uma função será executada [AKC19]. A plataforma usa Definições de Recursos Personalizados *(Custom Resource Definitions, CRDs)* para estender o API Kubernetes e cria funções como objetos personalizados. Esta permite que os programadores usem as APIs nativas do Kubernetes para interagir com as funções como se fossem objetos nativos do Kubernetes. O controlador Kubeless observa continuamente as mudanças para funcionar objetos e tomar as medidas necessárias. Por exemplo, se um objeto de função é criado, o controlador cria um pod para a função e limpa recursos quando a função objeto é excluída. Runtime de uma função é empacotado numa imagem do container e *Kubernetes configmaps* são usados para injetar o código de uma função no tempo de execução [MPD18]. A plataforma suporta execuções de funções para Python, GO, Java, Node.js, Ruby, PHP, Golang, .NET, Ballerina e tempos de execução personalizados. Por padrão, o Kubeless oferece suporte para tempos de execução em diferentes estados: *estável* que aten-

dem aos critérios dos requisitos técnicos e *incubador* permite que os tempos de execução sejam partilhados e melhorados até que estejam prontos para serem movidos para a pasta estável [Kub20a]. As funções podem ser implementadas usando vastas opções de triggers: *HTTP Trigger*, *CronJob Trigger* e *PubSub Trigger* [Kub20a]. O Kubeless aproveita Kubernetes *ingress* para fornecer roteamento para funções. Por padrão, uma função implementada é correspondida a um serviço Kubernetes usando *ClusterIP* como o serviço, isso significa que a função não é exposta publicamente [Kub20a]. A monitorização de Kubeless é feita através do software Prometheus para recolha de métricas e o Grafana para visualizar as métricas do Prometheus expostas por Kubeless [Kub20a].

- Apache OpenWhisk: É uma plataforma de computação sem servidor open source inicialmente desenvolvida pela *IBM* e posteriormente parte do projeto *Apache Incubator*. É também a tecnologia subjacente da plataforma IBM Cloud Functions do provedor da nuvem pública IBM Cloud [MPD18]. O modelo de programação OpenWhisk é baseado em três primitivas: ações, triggers e regras. Uma ação é uma função sem estado que executa código, enquanto trigger é uma classe de eventos que podem se originar de várias fontes. Uma regra associa um trigger a uma ação. Várias ações de diferentes linguagens podem ser compostas juntas para criar uma *pipeline de processamento* mais longo, chamada de *sequência*. Os principais componentes desta plataforma são: um servidor da web Nginx, controlador (controller), Apache Kafka, um componente Invoker e uma base de dados CouchDB para armazenar as credenciais do utilizador, metadados de ação, namespaces e as definições de ações, triggers e regras [AKC19]. A plataforma suporta várias opções para implantação, incluindo várias ferramentas de container: Kubernetes e OpenShift, Mesos e Docker Compose [Apa20]. A plataforma oferece suporte a uma lista crescente de linguagens, como Node.js, Go , Java , Scala , PHP , Python , Ruby e Swift , bem como adições recentes para Ballerina, .NET e Rust. recorrendo diretamente como ambiente à criação de containers docker com o ambiente de execução [Ope20b]. Na próxima secção é descrito com mais detalhes sobre esta plataforma por ser uns dos conceitos chave do trabalho.

As plataformas apresentadas na tabela 2.2, foram selecionadas com base no número de pontuação no github, exceto Apache OpenWhisk que é um do conceito chave deste trabalho, mesmo assim, a plataforma consta na lista das plataformas sem servidor open soure com mais estrelas no github. A tabela de resumo apresenta todas as plataformas que podem ser implementadas sobre ferramentas de orquestração de container, Kubeless e Fission são apenas recomendadas a implementação sobre Kubernetes, Apache OpenWhisk e OpenFaaS suportam mais ferramentas de orquestração de container. As mesmas suportam variedades de linguagens comum, o OpenWhisk e e Kubeless suportam listas mais crescente de linguagens. Todas as plataformas podem monitorizar as funções através de Prometheus de forma nativa.

Table 2.2: Comparação de Alguns Recursos das Plataformas Open Source.

Recursos	**OpenFaaS**	**Fission**	**Kubeless**	**Apache OpenWhisk**
Licença Open Source	MIT License	Apache Licence 2.0	Apache Licence 2.0,	Apache Licence 2.0
Linguagens Suportadas	Node.js, Python , Go, Java, incluindo Dockerfile	NodeJS, Python, Ruby, Go, PHP, Bash e qualquer executável Linux	Python, GO, Java, Node.js, Ruby, PHP, Golang, .NET, Ballerina e tempos de execução personalizados.	Node.js, Go , Java , Scala , PHP , Python , Ruby e Swift, Ballerina, .NET e Rust.
Escalabilidade	Automática	Automática	Manual / Automática	Automática
Monitorização	Prometheus e Grafana	Prometheus e Grafana	Prometheus e Grafana	Prometheus e Grafana
Orquestração de Container	Kubernetes, OpenShift e Docker Swarm	Kubernetes	Kubernetes	Kubernetes e OpenShift, Apache Mesos e Docker Compose

2.5 Apache OpenWhisk

Apache OpenWhisk [Apa20] é uma plataforma sem servidor open source que executa funções em resposta a eventos em qualquer escala. A plataforma usa o modelo de Function-as-a-Service (FaaS) para efetuar a gestão da infraestrutura, servidores e da escalabilidade, usando container Docker para que o programador possa concentrar-se apenas no desenvolvimento de aplicações [Apa20].

2.5.1 Resenha Histórica

No início de 2015, um grupo de investigadores da IBM iniciou um novo esforço para levar ao mercado um serviço de programação baseado em eventos distribuído e em nuvem. Em fevereiro de 2016 OpenWhisk foi anunciado publicamente pela primeira vez com disponibilidade no *IBM Cloud Bluemix* e no *GitHub*. Em dezembro de 2016, a plataforma tornou-se amplamente disponível. O Apache OpenWhisk também foi admitido pela *Apache Software Foundation Incubator* e ganhou mais de 110 contribuidores no GitHub, bem como parceiros como *Adobe* e *Red Hat* [IBM20a]. Antes de Apache Software Foundation (ASF) ter aceite o projeto Apache OpenWhisk, a base de código OpenWisk já estava sendo usada como tecnologia subjacente para IBM Cloud Functions [IBM20b].

O Apache OpenWhisk é um projeto popular e tem sido usada por empresas como uma solução para a gestão nas seguintes ofertas de nuvem: *IBM Cloud*, *Adobe I/O*, *Nimbella*

e entre outros [Mar20]. IBM contribui ativamente no projeto Apache OpenWhisk. Existem outras organizações que também contribuem como Adobe, Abilisense, AdvisorConnect, Altoros, Articoolo, BigVu, CloudStation, GreenQ, NASA JPL, KONG, Magentiq Eye, Naver, NeuroApplied, Nimbella, QZCloud, Red Hat, SiteSpirit etc[Apa20]. A plataforma OpenWhisk oferece suporte a um modelo de programação no qual os programadores escrevem lógica funcional (denominada Actions), em qualquer linguagem de programação compatível que pode ser agendada dinamicamente e executada em resposta a eventos associados (via triggers) de fontes externas (Feeds) ou de solicitações HTTP . A plataforma inclui uma interface de linha de comando (CLI) baseada em API REST junto com outras ferramentas para oferecer suporte a pacotes, serviços de catálogo e muitas opções populares de implementação de container. A plataforma constrói seus componentes usando containers, suporta muitas opções de implementação tanto local quanto numa infraestruturas de nuvem pública. As opções para implementação incluem muitos frameworks de container populares, como Kubernetes e OpenShift, Mesos e Docker Compose [Apa20].
O OpenWhisk torna simples para os programadores integrarem suas ações com muitos serviços populares usando Pacotes *(Packages)* que são fornecidos como projetos desenvolvidos de forma independente na família OpenWhisk ou como parte do Catálogo *(Catalog)* padrão da plataforma. Os pacotes oferecem integrações com serviços gerais, como filas de mensagens *Kafka*, base de dados, incluindo *Cloudant*, *Push Notifications* de aplicaç es móveis, *mensagens Slack* e f*eeds RSS*. Os pipelines de desenvolvimento podem aproveitar as integrações com *GitHub*, *JIRA* [Apa20].

2.5.2 Modelo de Programação e Funcionamento da Apache OpenWhisk

O modelo de programação da Apache OpenWhisck é orientado a eventos com três primitivas principais: action, trigger e regra(rule). Os eventos orientam a execução funções funcionais chamadas de ações (actions). Os eventos podem vir de qualquer fonte de evento ou serviço de feed, incluindo: Armazenamento de dados, filas de mensagens, aplicações móveis e web, sensores, chatbots, tarefas agendadas (via alarmes) entre outros. Ações são funções sem estado que são executados na plataforma OpenWhisk. As ações empacotam a lógica da aplicação a ser executada em resposta a eventos. As ações podem ser invocadas manualmente pelo API REST OpenWhisk, OpenWhisk CLI (Command line interface), APIs simples e criadas pelo utilizador ou automatizadas por meio de Triggers. Os triggers são canais nomeados para classes ou tipos de eventos enviados de Fontes de eventos. As regras são usadas para associar um trigger a uma ação. Depois deste tipo de associação ser criado, sempre que um evento de trigger é disparado, a ação é invocada [Ope20b]. As regras permitem que o mesmo Trigger seja associado a várias ações, bem como permitem que a automação específica seja habilitada ou desabilitada dinamicamente sem destruir a definição do Trigger. A plataforma oferece suporte a uma lista crescente de linguagens, como Node.js , Go , Java , Scala , PHP , Python , Ruby e Swift , bem como adições recentes para Ballerina, .NET e Rust. recorrendo diretamente como ambiente à criação de containers docker com o ambiente de execução [Ope20b].

OpenWhisk é baseado em uma arquitetura orientada a eventos, onde a maioria das ações são executadas conforme os eventos acontecem. O próprio Trigger é ”disparado” com um dicionário de pares de valores-chave (ou seja, os parâmetros) provenientes da Fonte de Eventos e permite a configuração de valores padrão opcionais, ajudando a garantir que os dados sejam compatíveis com quaisquer ações associadas[Ope20b]. O OpenWhisk tem alguns limites de sistema, incluindo quanta memória uma ação pode usar e quantas invocações de ação são permitidas por minuto conforme apresenta alguns destes limites de recurso na tabela 2.3. Esses limites padrão são para a distribuição de código aberto [Ope19]; implementações de produção como IBM Cloud Functions têm limites mais altos conforme mencionada anteriormente na tabela 2.1. Estes valores de limite do sistema OpenWhisk estão representados na tabela 2.3, pode notar-se que por padrão o sistema OpenWhisk aloca por ação uma memória de 256 MB e tempo por ação de 60.000 ms.

Table 2.3: Alguns Valores de limites do Sistema OpenWhisk.

Limite	Unidade	Intervalo	Padrão
Memória por ação	MB	128 MB até 512 MB	256 MB
Tempo por ação	milissegundos	100 ms até 300000 ms	60.000 ms

2.5.3 Arquitetura da Apache OpenWhisk

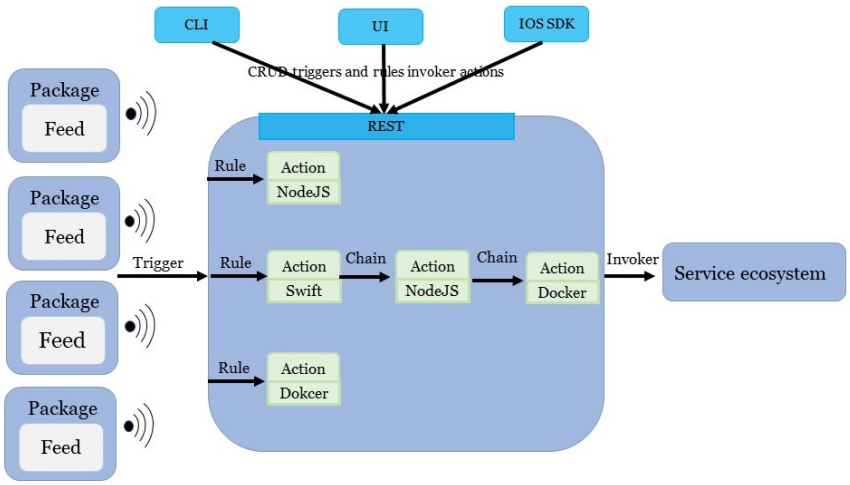

Figura 2.7: Arquitetura OpenWhisk de alto nível (figura redesenhada a partir de [The20])

A figura 2.7 ilustra uma arquitetura OpenWhisk de alto nível. Os eventos de fontes externas e internas são canalizados por meio de um trigger, e as regras permitem que as ações reajam a esses eventos. As ações podem ser pequenos fragmentos de código (NodeJs, Swift e outras linguagens compatíveis) ou código binário personalizado incorporado em um container do Docker. Além de associar ações a trigger, é possível invocar diretamente

uma ação usando a API OpenWhisk, CLI ou iOS SDK [The20].

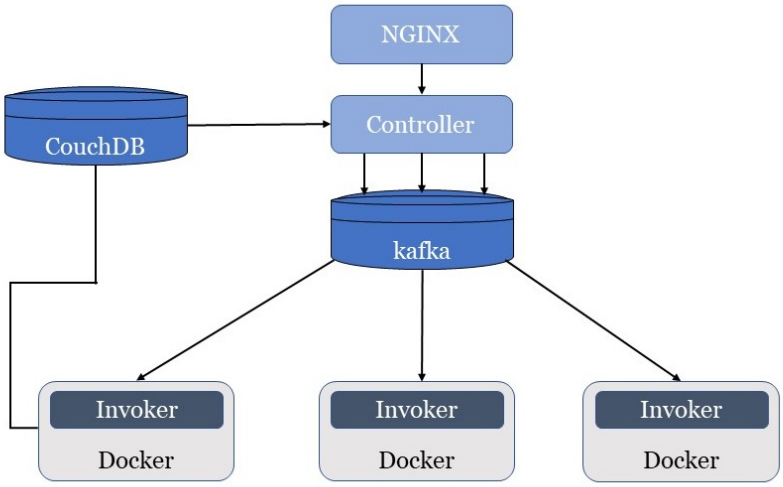

Figura 2.8: Fluxo de processo interno (figura redesenhada a partir de [Apa20])

A figura 2.8 apresenta uma arquitetura de fluxo de processamento interna do Apache OpenWhisk, incluindo vários componentes como Nginx, Kafka, Docker, CouchDB. Todos esses componentes juntam-se para formar um *serviço de programação baseado em eventos sem servidor*. A seguir a descrição de como cada dessas componentes operam no Apache Openwhisk:

- NGINX: Todas as solicitações no OpenWhisk passam pelo Nginx que é um proxy reverso e servidor HTTP. É o primeiro ponto de entrada do sistema, usado para encerramento SSL e encaminhamento de chamadas HTTP apropriadas para o próximo componente [The20].
- Controller: É a componente que permite fazer o controlo do processo de invocação do cluster. Os controller podem funcionar sem ter o API Gateway na frente deles. O controller atende o protocolo HTTP diretamente, de forma insegura, pois a parte HTTPS é a função do API Gateway. Basicamente, uma fonte de controler é uma parte do OpenWhisk. [Kae18].
- CouchDB: É a componente de armazenamento de dados do OpenWhisk baseado em documentos altamente disponíveis. O controller se comunica com o CouchDB para armazenar todas as entidades relacionadas à chamada da função. A entidade mais importante armazenada no CouchDB são os dados de ativação. Os dados de ativação contêm informações de cada processo de invocação. O andamento das ações e seus resultados são armazenados na forma de documentos [Kae18].
- Kafka: Desempenha um papel muito importante no sistema. Por natureza, o Kafka é um corretor de mensagens que armazena todas as mensagens recebidas e as reproduz de maneira confiável. Com o Kafka como *backbone*, as solicitações de invocação de ação serão entregues de forma confiável aos invokers [Kae18].

- Invoker: Responsável por receber pedidos de invocação de tópicos Kafka, filas de mensagens que os consumidores podem assinar para receber mensagens. Depois de receber as mensagens, o invocador executa as funções usando um tempo de execução de backend. OpenWhisk suporta tempos de execução nativos e Docker. O tempo de execução do Docker é chamado internamente de caixa preta [Kae18].
- Docker: O Docker é usado para configurar um novo ambiente auto empacotado (chamado container) para cada ação que for invocada de forma rápida, isolada e controlada. Para cada ação invocada, um container do Docker é gerado, o código de ação é injetado e é executado usando os parâmetros passados a ele [The20].

2.6 Trabalhos Relacionados

Há vários estudos realizados por diversos investigadores que envolvem a plataforma Apache OpenWhisk em conjunto com outras plataformas sem servidor, incluindo as plataformas open source e as plataformas comerciais. A maioria de estudos realizados sobre a plataforma Apache OpenWhisk está relacionados com plataforma comercial tal como IBM Cloud Functions que a sua gestão de funções como serviços é baseada em Apache Openwhisk.

Kuntsevich, Nasirifard e Jacobsen et al. [KNJ18] propuseram uma abordagem de análise e benchmarking para investigar potenciais problemas e limitações da plataforma sem servidor Apache OpenWhisk. Propomos várias funções de teste para vários casos de uso sem servidor. Desenvolvemos uma estrutura analítica para executar os testes em uma instalação personalizada do OpenWhisk e também uma aplicação baseada em servidor. Além disso, o sistema reúne as métricas de teste. O ambiente de teste foi implementado em quatro máquinas virtuais Ubuntu 16.04 com 1 VCPU, 2,4 GB de RAM e uma VM com 4 VCPUs, 9,8 GB de RAM, as funções de teste foram implementadas em JavaScript e Java, onde as mesmas foram definidas para teste de execução na plataforma: Cálculo do enésimo número primo, dado o N como a entrada. Desde que N seja grande o suficiente, o cálculo é CPU intensivo, Cálculo da multiplicação da matriz, que é RAM e uso intensivo da CPU, solicitação de um recurso externo da web, o que mostra potencial Gargalos de E / S.

No artigo [BPGLR+19] é apresentado um estudo sobre orquestração de funções sem servidor para execuções paralelas, demostraram um estudo comparativo sobre overhead de orquestrador de funções oferecido por alguns provedores de serviços de nuvens, nomeadamente Step Functions da AWS da AWS, Azure Durable Functions da Microsodft Azure e Apache Openwhisk Composer/IBM, onde os três primeiros incorreram overheads consideráveis, apenas o compose da IBM ofereceu um desempenho adequado. Foi comparada a sobrecarga adicionada por diferentes orquestradores FaaS oferecidos por provedores de nuvem para avaliar o desempenho desses serviços ao gerir grandes quantidades de tarefas paralelas em um fluxo de trabalho. Ainda assim, foi analisada a arquitetura do OpenWhisk como um sistema FaaS de código aberto e os seus recursos de orquestração.

Quevedo e Merchan et al. [QMRD19] apresentam um estudo sobre desempenho da plataforma sem servidor Apache Openwhisk. A plataforma Apache OpenWhisk foi avaliada construindo funções com os tempos de execução Java e JavaScript. Os experimentos foram feitos no Amazon Elastic Compute Cloud (Amazon EC2), em um trecho Debian amd64 do tipo t2.medium. Para testar o desempenho do OpenWhisk com os parâmetros padrão, o utilizador da web é implementado que chama simultaneamente as funções de redimensionamento de imagem criadas com Java e JavaScript. O tempo de execução em Java foi de 18,09ms e em JavaScript de 34,68ms. Verificou-se que, sob certas premissas, obtém-se uma melhoria do desempenho nas latências de inicialização cold-booting de até 38%.

McGrath e Brenner et al. [MB17] abordaram sobre a plataforma de computação sem servidor orientada para o desempenho implementada em .NET, integrada no Microsoft Azure e utilizando containers do Windows como ambientes de execução de funções. Propuseram métricas para avaliar o desempenho de execução de plataformas sem servidor na AWS Lambda, Azure Functions, Google Cloud Functions e Apache OpenWhisk da IBM. Foi projetado um teste de Backoff e teste de concorrência para medir o desempenho de execução de funções. Desenvolveram uma ferramenta de desempenho para realizar esses experimentos, que implementa uma função de teste Node.js para os diferentes serviços usando o Serverless Framework. Também construímos um Plug-in sem servidor para habilitar o suporte do Serverless Framework para a plataforma. As medições mostram o protótipo alcançando maior rendimento do que outras plataformas na maioria dos níveis de concorrência, e examinaram as tendências de escala e expiração de instância nas implementações. Além disso, discutiu-se as lacunas e limitações no design desenvolvido, propondo soluções possíveis.

No artigo de Bila, Dettori, Kanso e Watanabe [BDK+17] é apresentado design de uma arquitetura automatizada de mitigação de ameaças baseadas em OpenWhisk e Kubernetes com o container do Linux. No kubernetes foram implementadas as funcionalidades que permitem ao sistema reagir a ameaças de maneira automatizada de acordo com as suas políticas definidas pelo utilizador, OpenWhisk como um habilitador chave para esta funcionalidade, de maneira a permitir que os utilizadores criem suas próprias políticas de uma forma genérica. No contexto de proteção de um container comprometido, o serviço de aplicação de segurança deve suportar o encerramento a bruto, o encerramento normal ou a quarentena de container. A quarentena é aplicada ao bloquear qualquer comunicação de entrada e saída do container comprometido. Para deteção de ameaças foi usado o scanner de vulnerabilidade e ferramentas de varredura de vulnerabilidades de container, como *Vulnerability Advisor* e *Docker Security Scanning*.

Na dissertação de José [Mar19b] é realizada a identificação e estudo das principais plataformas sem servidores existentes. Sendo estas comparadas em termos de funcionalidades e

características através da utilização de um conjunto de parâmetros apresentados. Foi proposto e desenhando benchmark de utilização genérica de plataforma sem servidor que que permite fazer um grupo de testes e com a vista de automatização da sua execução. Foram testadas as plataformas sem servidor AWS Lambda, Azure Functions, Google Functions, IBM Functions/Openwhisk. O sistema de execução de benchmark implementado permite o deploy de funções e a execução do conjunto de teste propostos nas referidas plataformas.

Lee, Satyam e Fox [LSF18] apresentam um estudo sobre Avaliação de ambientes de computação sem servidor de produção, incluindo as plataformas AWS Lambda, Microsoft Azure Functions, Google Cloud Functions e IBM Cloud Functions. Foram avaliados os ambientes de computação sem servidor quanto ao rendimento de invocação simultânea, CPUs, tempo de resposta para carga de trabalho dinâmica, sobrecarga de tempo de execução. Também comparou-se o custo, a taxa de transferência do acionador de evento e os recursos usando um conjunto de funções escritas por tempos de execução suportados, como nodeJS, Python, Java e C#. Todas as avaliações foram realizadas usando alocação de memória de 1,5 GB, exceto IBM com 512 MB. Foi usada biblioteca Boto3 no Amazon Lambda para especificar o tamanho de uma chamada de função simultânea e API HTTP para Microsoft Azure e IBM OpenWhisk. Os resultados mostram que a elasticidade do Amazon Lambda excede outras em relação ao desempenho da CPU, largura de banda da rede e uma taxa de transferência de E/S de arquivo quando invocações de função simultâneas são feitas para cargas de trabalho dinâmicas.

Djemame et. al. [DKP20] investigam sobre as capacidades e limitações inerentes à arquitetura de computação sem servidor Apache OpenWhisk. Foi realizado por meio de uma extensa avaliação do OpenWhisk, envolvendo uma variedade de experimentos e benchmarks. O plano para a avaliação do desempenho do OpenWhisk foi feita, usando uma única linguagem de programação para implementação de funções separadas, com cada função direcionada a um único recurso de hardware. Os experimentos envolveram a criação e execução de funções do programa, cada uma das quais destinada a consumir um recurso de hardware específico. Duas métricas foram de interesse na experimentação: tempo de execução da função e utilização de recursos. Os resultados de cada experimento mostraram que o OpenWhisk pode superar uma solução que emprega funcionalidade semelhante, por meio do uso de virtualização baseada em container. Também demostrou o quão próximo o Open-Whisk está em termos de desempenho de uma solução mais otimizada que não sofre com as despesas de virtualização.

2.7 Conclusão

Este capítulo forneceu os fundamentos teóricos que serviram de base para tornar claro o tema em estudo, foram apresentados os conceitos básicos sobre virtualização, sobre tecnologia de container Docker, Kubernetes, Computação na Nuvem, plataformas sem servidor, incluindo as plataformas públicas e open source. Apache Openwhisk faz parte

das plataformas open source usado como base para gestão de algumas plataformas sem servidor comercial, nomeadamente a IBM Cloud através da sua plataforma sem servidor IBM Cloud Functions e Adobe com Adobe I/O. Apache Openwhisk pode ser opção para clientes que não pretendam depender de fornecedores de nuvem publica, pode ser instalada localmente ou numa nuvem privada, e é recomendada com Kubernetes para ambiente de produção. IBM Cloud Functions é baseada em OpenWhisk que combina componentes como NGINX, Kafka, Docker e CouchDB para formar um serviço de programação baseado em eventos sem servidor. A plataforma IBM Cloud Functions é diferente das tecnologias de computação tradicionais, apenas é paga pelo tempo que o código atender às requisições. Os custos aumentam apenas à medida que utilizador executa mais carga no Cloud Functions, a fatura é baseada no tempo e na memória necessária para executar funções.

Capítulo 3

Implementação do Ambiente Experimental e Análise dos Resultados Experimentais

3.1 Introdução

Neste capítulo são apresentados os ambientes experimentais propostos no trabalho, com os dois ambientes distintos, localmente usando a plataforma Apache OpenWhisk junto com a ferramenta Kubernetes, e na nuvem pública, envolvendo o provedor de serviço cloud da IBM, concretamente na sua plataforma sem servidor denominado de IBM Cloud Functions. Ainda assim, são apresentadas as caraterísticas de recursos de hardware e software utilizados no ambiente de teste, os métodos de implementação dos ambientes experimentais, ferramentas para avaliação do desempenho, análise dos resultados experimentais e a comparação experimental do desempenho das duas plataformas em estudo.

3.2 Ambiente de Testes Experimental

3.2.1 Caraterização do Ambiente de Teste Experimental

Os ambientes experimentais envolveram dois ambientes distintos, conforme apresenta a figura 3.1, a figura representada por letra A representa o ambiente local, o mesmo é constituído por duas máquinas virtuais que serviram para o cluster do Kubernetes na qual foi implementada a plataforma Apache OpenWhisk, numa configuração que permitiu a integração da ferramenta de métrica *Prometheus* e *Grafana*, detalhadas mais em diante. Isto foi feito através de uma máquina host e foi usada a tecnologia de virtualização *VMware Workstation* para criação de máquinas virtuais. O cluster criado é composto por um master node e um worker node indicados para executar funções de utilizador pelos invocadores *(invokers)* de OpenWhisk. Todas as máquinas foram instaladas o sistema operativo *Ubuntu 18.04 LTS*, a máquina encontra-se na mesma rede, a conexão de rede no VMware foi usada a opção de bridge e as máquinas foram atribuídas IPs estatísticos. A figura representada por B representa provedor de serviço de nuvem púbica, conforme mencionada várias vezes em diferentes pontos anteriores deste trabalho, foi usada a provedor pública IBM Cloud, concretamente a sua plataforma sem servidor denominada de IBM Cloud Functions, a mesma usa OpenWhisk como base de Function-as-a-Service. De maneira a operar nesta plataforma foi criada uma conta do tipo *Lite*, na qual não requere o uso de cartão de crédito, em diante é apresentada com mais detalhes.

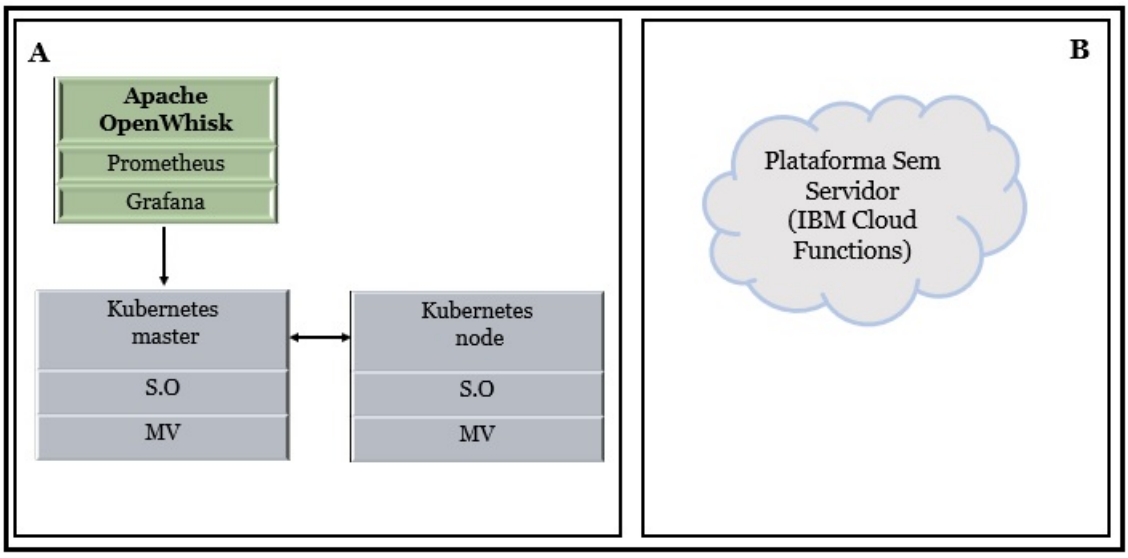

Figura 3.1: Arquitetura: A - Local, B - Nuvem Pública.

3.2.2 Especificações de Hardware e Software do Ambiente de Testes

Indo em conta com o que já havia referenciado na secção anterior, o ambiente experimental local consiste em um host, na qual foi instalada a ferramenta de virtualização *VMware*. A tabela 3.1, apresenta as suas especificações de hardware e software.

Tabela 3.1: Especificações de Hardware e Software da máquina host.

Descrição	**Especificações**
Processador	Intel Core i7-7700 3.60 GHz
Memória RAM	16.0 GB
Armazenamento	930 GB
Sistema Operativo	Windows 10 Pro
Software de Virtualização	VMware Workstation 15 Pro

A tabela 3.2 apresenta as especificações de hardware e software, bem como as versões, do cluster Kubernetes onde foi implementa a plataforma Apache OpenWhisk. É de referir que o Node Master e Worker têm as mesmas especificações e o provedor de serviço nuvem usada é da IBM Cloud, na sua plataforma sem servidor IBM Cloud Functions/OpenWhisk.

Tabela 3.2: Especificações do Hardware e Software do Cluster.

Descrição	**Especificações**
Sistema Operativo	Ubuntu -18.04-desktop-amd64
CPU	2 Cores
Memória RAM	4.5 GB
Sistema Operativo	Ubuntu-18.04-desktop-amd64
Docker	18.06.1.ce
Apache OpenWhisk	V1.0.0
Kubernetes	V1.20.5
Helm	V3.2.4
OpenWhisk CLI	V1.1.0
Rede do Cluster	Calico

3.3 Instalação e a Configuração do Ambiente Experimental

Esta secção descreve os procedimentos de implementação do ambiente experimental, envolvendo os dois ambientes distintos já mencionados. Em primeiro lugar são descritos os procedimentos de ambiente local usando a plataforma Apache OpenWhisk junto com a plataforma Kubernetes, posteriomente, o provedor de serviços da IBM Cloud, nomeadamente a sua plataforma sem servidor denominada de IBM Cloud Funtions, e por último as ferramentas para análise do desempenho.

3.3.1 Instalação e a Configuração do Kubernetes e Apache OpenWhisk

Para implementação do ambiente sem servidor baseando-se na arquitetura proposta, envolvendo a plataforma Apache OpenWhisk com a Plataforma Kubernetes foi necessário primeiramente criar um cluster Kubernetes, usando a ferramenta Kubeadm[1]. Para tal foi necessário primeiro efetuar a instalação de Docker que se encarregou de baixar os pacotes que deram suporte a complementos para o funcionamento do ambiente. Essa configuração envolveu a instalação dos pacotes, foi adicionado a chave GPG e o repositório Docker em cada nodes do cluster. Foi necessário definir systemd como Cgroup driver que é recomendado na criação de cluster Kubernetes. De realçar que quando o Docker é instalado e iniciado, por padrão usa Cgroupfs *(control group)* como cgroup driver. Foram feitas alterações no ficheiro docker.service localizado em */lib/systemd/system/*, substituído o caminho por *ExecStart=/usr/bin/dockerd –exec-opt native.cgroupdriver=systemd*, em seguida recarregado *daemon* e reiniciando o Docker com os comandos *systemctl daemon-reload* e *systemctl restart docker*. A figura 3.2 mostra o resultado de instalação correta do Docker executando o *docker run hello-world*.

[1] `https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm/`

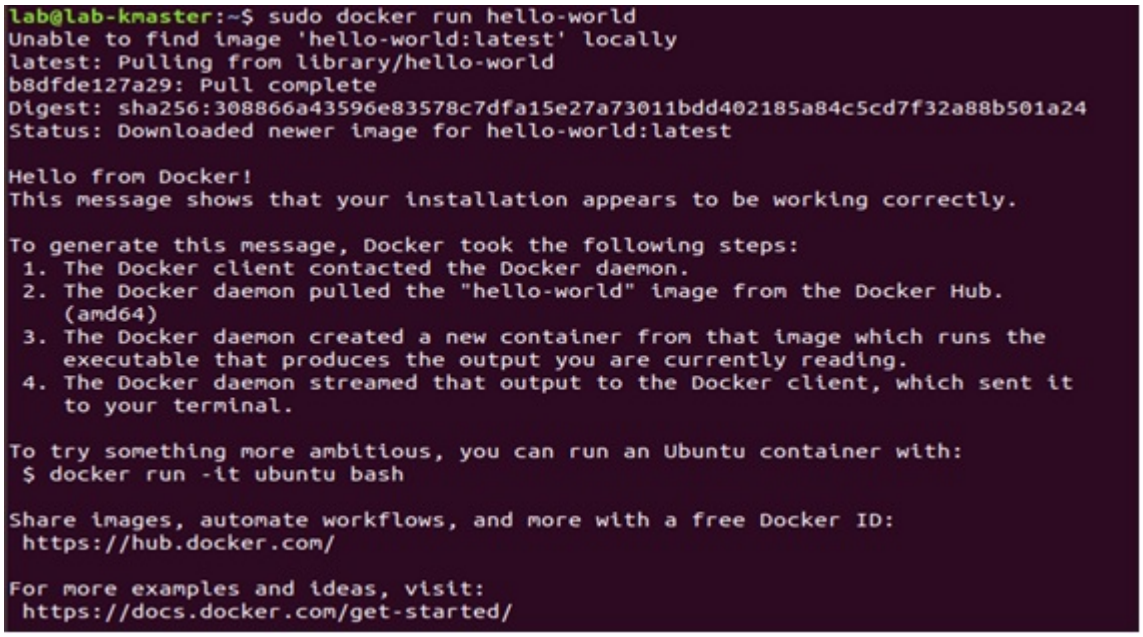

Figura 3.2: Verificação da instalação do Docker.

Para instalação de Kubernetes e os seus componentes principais usando o Kubeadm para criação de cluster, é necessário garantir alguns requisitos basilares para que não se comprometa o funcionamento Kubernetes, tais como [Kub21]:

- O sistema operativo da máquina deve ser compatível ao kubernetes;
- As máquinas devem ter pelo menos 2GB ou mais de memória RAM e 2 CPU (Principalmente a máquina escolhida como Kubernetes master);
- Deve haver a conetividade de rede entre todas máquinas do cluster;
- O nome de host, endereço de MAC (Media Access Control) deve ser exclusivo para cada nodes do cluster;
- A memória SWAP deve estar desabilitada de maneira a permitir que o kubelet funcione corretamente;
- Algumas portas especificas de master node e worker nodes têm de estar abertas.

Tendo cumprido os requisitos mencionados acima, em seguida são instaladas as componentes de suporte HTTPS, adicionando a chave de assinatura e repositório do Kubernetes no sistema. Logo em seguida é feita a instalação dos principais componentes: *kubelet, kubeadm e kubectl* em todos os nodes do cluster. É iniciado o cluster conforme a figura 3.3, exclusivamente na máquina escolhida como node master com o comando *kubeadm init*, incluido os parâmetros - *-pod-networker-cidr* que é usado para configurar rede e definir intervalos CIDR (Classless Inter-Domain Routing) que é o método de endereçamento IP sem classes e *–apiserver-advertise-address* que define o endereço IP que é anunciado pelo Kubernetes como o seu servidor API.

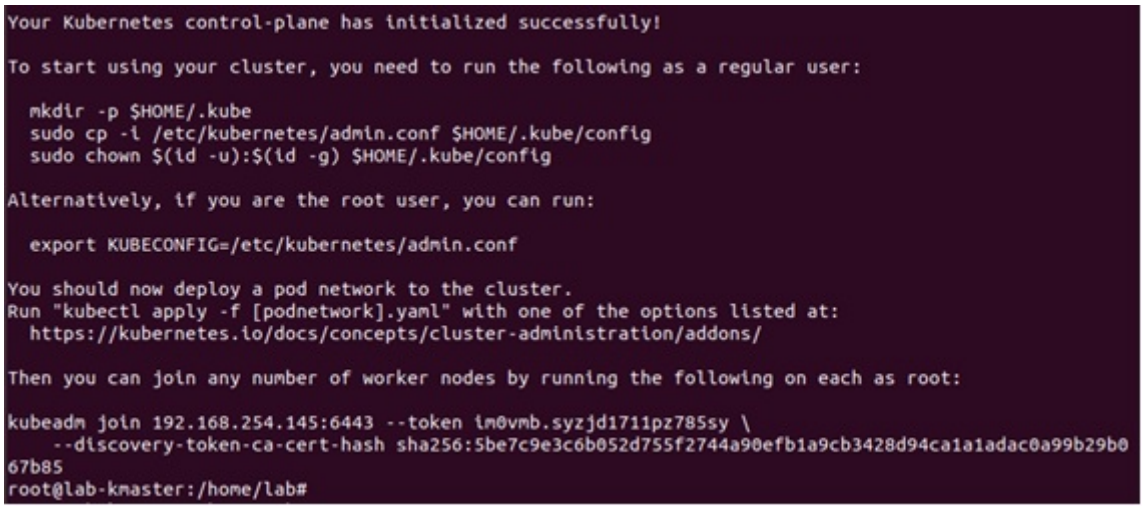

Figura 3.3: Inicialização do Node Master.

A figura 3.3 apresenta a inicialização de node master com duas orientações importantes fornecidas pelo *kubeadm*; a primeiro indique a execução de três comandos como utilizador regular na máquina master que permita a configuração a conexão de *kubectl* e de forma a permitir usar o cluster, e a outra é referente ao *token* que é executado junto com o comando *kubeadm join* que permite unir os nodes worker no cluster. A figura 3.4 apresenta a associação de worker node a cluster depois de ter sido executado o comando kubeadm join. Este procedimento é feito na máquina worker node e, ainda assim, foi necessário fornecer uma rede de pod no cluster, como plugin de rede foi instalada Calico[2]. Este procedimento foi feito na máquina master node.

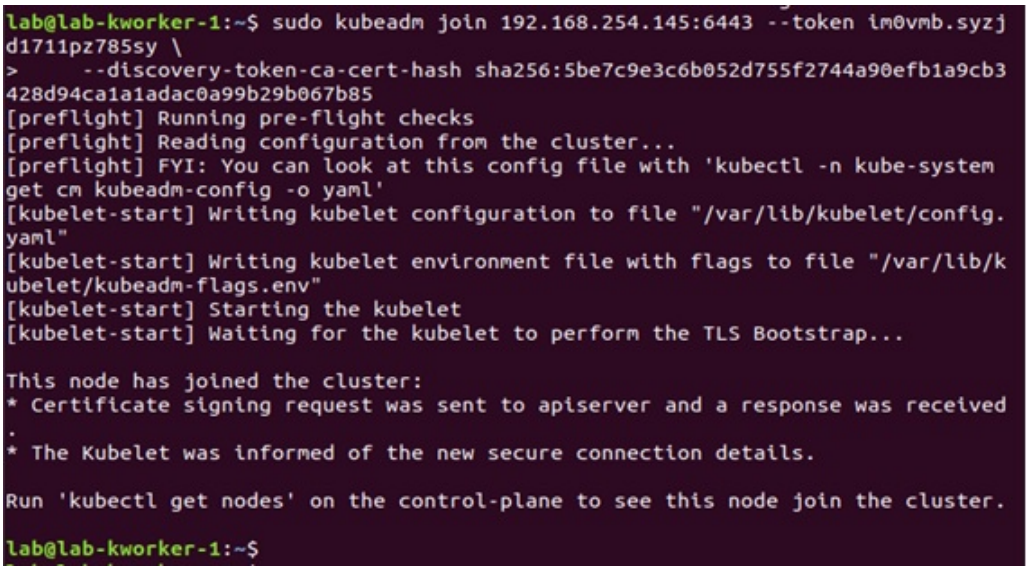

Figura 3.4: Worker node assocociado a cluster.

A figura 3.5 apresenta a lista de nodes associados a cluster, e que estão prontas para executarem a carga de trabalho, isto pode ser verificado executando o comando kubectl get nodes.

[2] https://docs.projectcalico.org/getting-started/kubernetes

```
lab@lab-kmaster:~$ kubectl get nodes
NAME           STATUS   ROLES                  AGE     VERSION
lab-kmaster    Ready    control-plane,master   12m     v1.20.5
lab-kworker-1  Ready    <none>                 2m56s   v1.20.5
lab@lab-kmaster:~$
```

Figura 3.5: Listas de Nodes associados no Cluster.

Tendo sido criado o cluster Kubernetes, para efetuar a implementação da plataforma Apache OpenWhisk no cluster, foi necessário configurar o Helm[3] que é uma ferramenta para simplificar a implementação e a gestão de aplicações em cluster Kubernetes. Foi necessário baixar o pacote Helm e descompactá-lo, em seguida é movido o binário para diretório bin conforme o comando e caminho *sudo mv linux-amd64/helm /usr/local/bin/helm*, foi necessário indicar os nodes worker de Kubernetes que foram usados para executar OpenWhisk, neste caso, foi indicado que todos os nodes no cluster a poderem executar o OpenWhisk através do comando *kubectl label nodes –all openwhisk-role=invoker*. Foi criado um namespace Kubernetes para o OpenWhisk com o comando kubectl create namespace openwhisk. Em seguida é clonado o repositório OpenWhisk com o comando *git https://github.com/apache/openwhisk-deploy-kube.git*, é acedido este repositório e criado o ficheiro *myscluetr.yaml* que regista os principais aspetos do cluster Kubernetes que são necessários para configurar a implementação do OpenWhisk no cluster Kubernetes, conforme a figura 3.6. É importante referir que quando é clonado o Openwhisk, é criado por padrão um ficheiro de denominado por *values.yaml*, localizado na árvore da fonte, *helm/openwhisk/*, que contem toda informação de configuração de Openwhisk, o ficheiro *myscluetr.yaml*, apenas personalize a implementação de OpenWhisk.

```
whisk:
  ingress:
    type: NodePort
    apiHostName: 192.168.254.145
    apiHostPort: 31001

nginx:
  httpsNodePort: 31001

k8s:
  persistence:
    enabled: false

invoker:
  containerFactory:
    impl: "kubernetes"

metrics:
  prometheusEnabled: true

metrics:
  userMetricsEnabled: true
```

Figura 3.6: Ficheiro de configuração OpenWhisk.

[3] https://https://helm.sh/

A figura 3.6 ilustre uma estrutura do ficheiro mycluster.yaml para uma implementação padrão de OpenWhisk, com uma entrada *NodePort* para aceder ao OpenWhisk, o *apiHostname*, neste caso é atribuído o IP de node master, recolha de métricas com duas opções habilitados prometheusEnabled e userMetricsEnabled que permitam instalar inventos de utilizadores, Prometheus e Grafana no cluster Kubernetes, ainda assim, foi desativado o uso de persistência. É instalado o OpenWhisk com o suporte da ferramenta Helm, executando o comando *helm install owdev ./helm/openwhisk -n openwhisk -f mycluster.yaml*. A figura 3.7 apresenta o resultado após a execução do comando referido, apresentado a data de deployment, o owdev como o nome da versão e openwhisk como o nome de namespace.

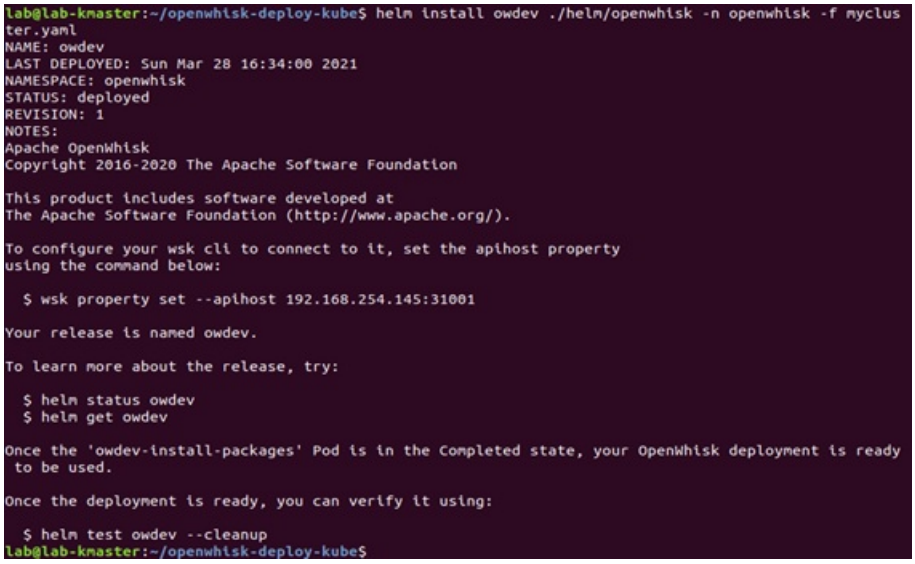

Figura 3.7: Implementação da Apache Openwhisk.

Após a implementação da plataforma Apache OpenWhisk através da ferramenta Helm, é possível observar os pods a serem executados através do comando *kubectl -n openwhisk get pod -w*, conforme apresenta figura a3.8.

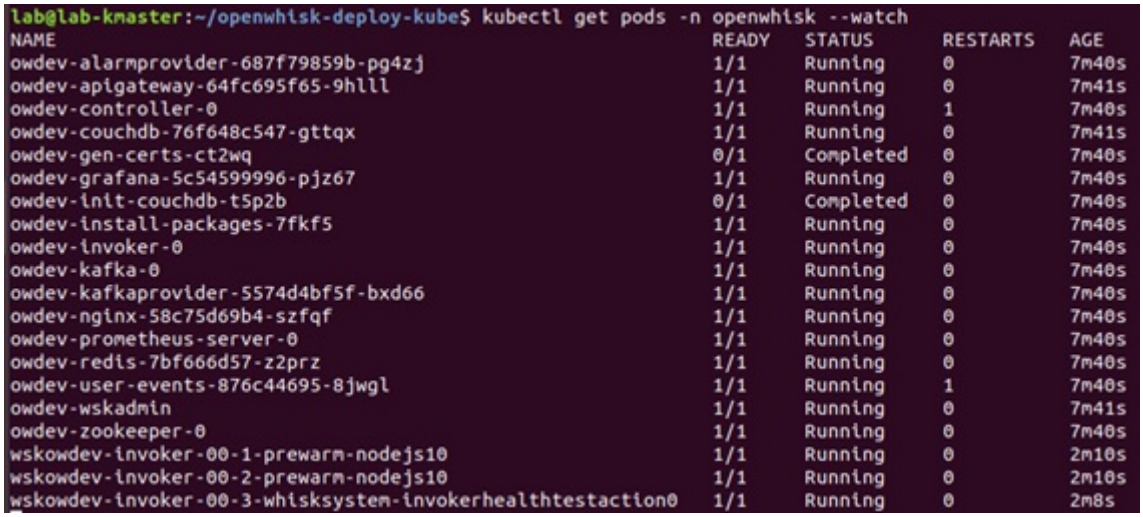

Figura 3.8: Verificação de execução dos pods.

Em seguida é instalado e configurado OpenWhisk CLI[4] que é uma ferramenta unificada que fornece uma interface consistente para interagir com os serviços OpenWhisk, o mesmo é baixado, descompactado e movido o binário para o diretório bin. a figura 3.9 mostra a verificação da instalação através do comando wsk –help.

Figura 3.9: Verificação da instalação do OpenWhisk CLI.

Para que se utilize na integra o OpenWhisk CLI, é necessário configurar duas propriedades essenciais, a primeira é *API host* que se refere o nome ou endereço de IP, neste caso foi atribuído o IP configurado no arquivo mycluster.yaml, e a segunda propriedade é *Authorization key* (chave de autenticação) que se refere ao nome de utilizador e senha que permitem aceder à API OpenWhisk. Essas duas propriedades tem as seguintes chaves [Apa21b]:

- APIHOST – Chave solicitada para o valor de API host;
- AUTH – Chave solicitada para autenticação (Authorization key).

Tendo configurado o OpenWhisk com êxito, executando a linha de comando *kubectl -n openwhisk -ti exec owdev-wskadmin – wskadmin user list guest*, que contem o nome de implantação (owdev) e o namespace (openwhisk), é possível listar utilizador guest (convidado) conforme a figura 3.10. Por padrão o token especificado é o token padrão adicionada no Openwhisk na conta guest no momento da implantação. Em produção é recomendado editar o arquivo *values.yaml*, localizado na arvore da fonte, helm/openwhisk/ de maneira a alterar as credências. As configurações das propriedades OpenWhisk CLI podem ser configuradas através do comando:

[4] https://github.com/apache/openwhisk/blob/master/docs/cli.md#openwhisk-cli

1. wsk -i property set –apihost whisk.ingress.apiHostName;
2. wsk property set –auth.

Onde *whisk.ingress.apiHostName* e *whisk.ingress.apiHostPort* é substituída pelos valores reais do ficheiro mycluster.yaml da figura 3.6, na segunda propriedade com o comando *wsk property set –auth mencionado o token*, conforme a figura 3.10.

```
lab@lab-kmaster:~$ kubectl -n openwhisk -ti exec owdev-wskadmin -- wskadmin user list guest
23bc46b1-71f6-4ed5-8c54-816aa4f8c502:123zO3xZCLrMN6v2BKK1dXYFpXlPkccOFqm12CdAsMgRU4VrNZ9lyGVCGuMDGI
wP      guest
lab@lab-kmaster:~$ wsk -i property set --apihost 192.168.254.145:31001
ok: whisk API host set to 192.168.254.145:31001
lab@lab-kmaster:~$ wsk -i property set --auth 23bc46b1-71f6-4ed5-8c54-816aa4f8c502:123zO3xZCLrMN6v2
BKK1dXYFpXlPkccOFqm12CdAsMgRU4VrNZ9lyGVCGuMDGIwP
ok: whisk auth set. Run 'wsk property get --auth' to see the new value.
lab@lab-kmaster:~$
```

Figura 3.10: Configuração das Propriedades do OpenWhisk CLI.

Tendo sido configuradas as duas propriedades de OpenWhisk CLI com as suas respetivas chaves, é possível verificar se estão corretamente definidas as configurações das referidas propriedades, executando o comando *wsk -i property get* ou listar alguns pacotes instalados através do comando *wsk -i package list /whisk.system*, conforme a figura 3.11.

```
lab@lab-kmaster:~$ wsk -i property get
whisk API host          192.168.254.145:31001
whisk auth              23bc46b1-71f6-4ed5-8c54-816aa4f8c502:123zO3xZCLrMN6v2BKK1dXYFpXlPkccOFqm12C
dAsMgRU4VrNZ9lyGVCGuMDGIwP
whisk namespace         guest
client cert
Client key
whisk API version       v1
whisk CLI version       2020-10-02T00:33:38.010+0000
whisk API build         2020-10-07-10:25:41Z
whisk API build number  20201007a
lab@lab-kmaster:~$ wsk -i package list /whisk.system
packages
/whisk.system/messaging                                                 shared
/whisk.system/alarms                                                    shared
/whisk.system/github                                                    shared
/whisk.system/weather                                                   shared
/whisk.system/websocket                                                 shared
/whisk.system/utils                                                     shared
/whisk.system/slack                                                     shared
/whisk.system/samples                                                   shared
lab@lab-kmaster:~$
```

Figura 3.11: Verificação das Configurações das Propriedades OpenWhisk CLI.

3.3.2 Configuração da IBM Cloud Functions/OpenWhisk

A plataforma sem servidor da IBM é denominada de IBM Cloud Functions, a mesma usa a base de OpenWhisk para Function-as-a-Service conforme mencionado anteriormente em vários pontos neste trabalho.

Para sua utilização foi necessário criar uma conta, a mesma possui três tipos de contas diferente: *Lite*, *Pay-AS-You-GO (pré-pago)* e *Subscription (Assinatura)* [IBM21a]. Para este trabalho foi criada a conta *Lite* que não requer o uso de cartão de crédito, as duas

últimas são faturáveis. O IBM Cloud Functions oferece um *plug-in*[5] para a CLI do IBM Cloud que permite efetuar a gestão completa do sistema Cloud Functions. É possível usar o plug-in da CLI do Cloud Functions para gerir funções em ações, criar triggers e regras para ativar as ações para responder a eventos e empacotar ações. Ainda assim, também possui uma *Interface de Utilizador*[6] baseada em Web da IBM Cloud Functions que permite efetuar a gestão de aplicações sem a necessidade de interface de linha de comando. Para este trabalho foi usada a *Interface de Utilizador* Web da IBM Cloud Functions, para acedê-lo depois de ter a conta criada, é necessário ir para menu de navegação, em seguida clicando na opção functions, logo é apresentada a página inicial da IBM Cloud Functions conforme o endereço `https://cloud.ibm.com/functions/`, clicando na opção start creating, em seguida são apresentadas todas as entidades da IBM Cloud Functions tais como *action (ação)*, *trigger (acionador)*, *sequence (sequência)* entre outros, conforme a página `https://cloud.ibm.com/functions/create`. De referir para aceder esses links é necessário ter a conta criada.

3.3.3 Ferramentas de Análise de Desempenho

Para avaliação do desempenho da arquitetura proposta, na instalação local foram utilizados os softawares open source *Prometheus*[7] e Grafana[8], no serviço de nuvem publica foi usada a ferramenta da *IBM Cloud Function Dashoard*[9] que permite efetuar a monitorização invocações de funções, desempenho e erros.

Prometheus Prometheus é um software open souce de monitorização e serviço que permite recolher métricas de destinos configurados em determinados intervalos, avalia expressões de regras, exibe os resultados e pode acionar alertas quando as condições especificas são observadas. O Prometheus extrai métricas de trabalhos instrumentados, seja diretamente ou por meio de um gateway intermediário para trabalhos de curta duração. A ferramenta armazena todas as amostras recolhidas localmente e executa regras sobre esses dados para agregar e registar novas séries temporais de dados existentes ou gerar alertas [Pro21].. O Prometheus consiste em vários componentes, muitos dos quais são opcionais [Pro21]:

- Servidor Prometheus: O servidor principal que recolhe e armazena as métricas recolhidas em um base de dados de série temporal;
- Scrape: O servidor Prometheus usa um método de extração para recuperar as métricas.
- Target: Os clientes dos servidores Prometheus dos quais recupera informações;

[5] `https://cloud.ibm.com/docs/openwhisk?topic=openwhisk-cli_install`
[6] `https://cloud.ibm.com/functions/`
[7] `https://prometheus.io/`
[8] `https://grafana.com/grafana/`
[9] `https://cloud.ibm.com/functions/dashboard/`

- Explorer: Bibliotecas de target que convertem e exportam as métricas existentes para o formato Prometheus; exclusivo para cada nodes do cluster;

- Alert Manager: Componente responsável pelo tratamento de alerta, entre outras ferramentas de suporte.

O Prometheus geralmente é usado em conjunto com outro software open source denominado de Grafana, é um software de código aberto para visualização e análise. A ferramenta permite que se consulte, visualize, alerte e explore as métricas, não importa onde armazenadas, ou seja oferece ferramentas para transformar os dados de base de dados de série temporal em gráficos e visualizações [Lab21]. o *Promteheus* recolhe as métricas e o *Grafana* pega nelas e exibe-as em forma de dashboard. A plataforma Apache OpenWhisk suporte métricas e *Prometheus*. Essas métricas incluem as métricas do sistema que contém informações sobre o desempenho do sistema e outra refere-se ao utilizador que abrange informações sobre o desempenho da ação que são enviadas ao Kafka na forma de eventos [Apa21a]. Essas opções foram ativadas conforme as configurações feitas no ficheiro myscluster.yaml da figura 3.6.

```
metrics:
  prometheusEnabled: true
```

Figura 3.12: Configuração de Métricas do Sistema.

Conforme a figura 3.12, desta forma as métricas do sistema é recolhida, armazenada e exibida com o Prometheus, ativando automaticamente o Prometheus dentro do Cluster. Para aceder ao Prometheus usando o encaminhamento de porta é necessário executar o comando *kubectl port-forward svc/owdev-prometheus-server 9090:9090 –namespace openwhisk*, conforme apresenta o resultado na figura 3.14. Para Ativação da métrica de utlizador, foi conforme a configuração feita no mesmo ficheiro da figura 3.6:

```
metrics:
  userMetricsEnabled: true
```

Figura 3.13: Configuração de Métricas do Utilizador.

Como se pode ver na figura 3.13, é desta forma que são instalados os eventos do utilizador, Prometheus e Grafana no cluster com painéis Grafana já pré-configurados para visualizar as métricas geradas pelo utilizador. A figura 3.15 apresenta a página inicial de *Grafana*.

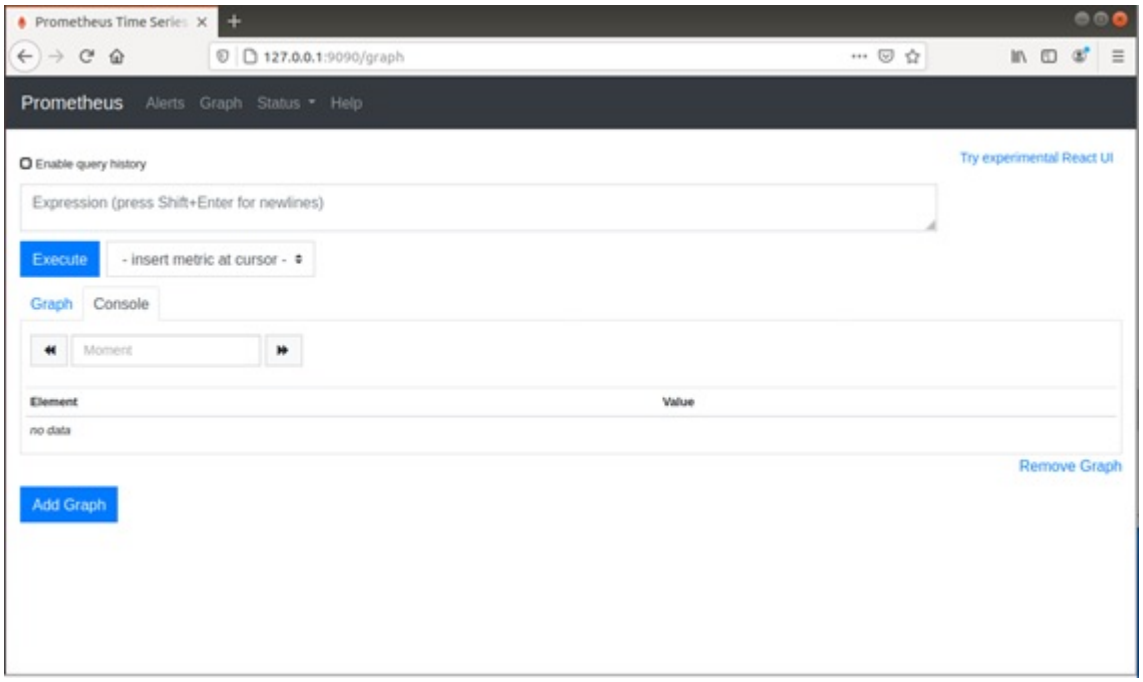

Figura 3.14: Pagina Inicial de Prometheus.

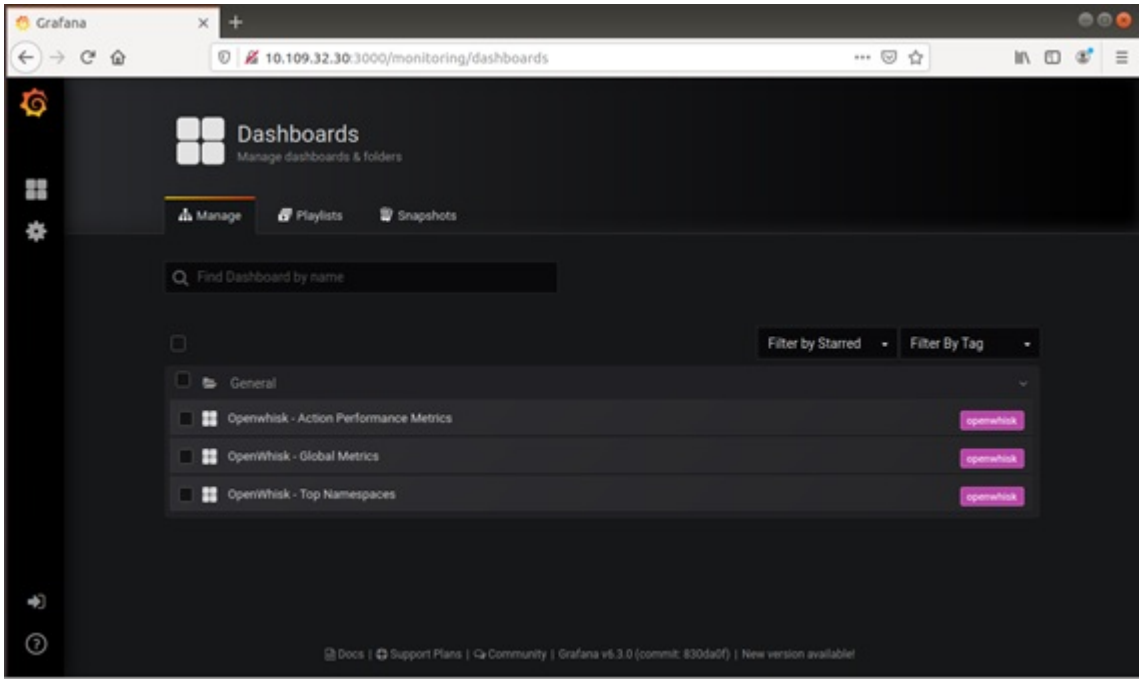

Figura 3.15: Pagina Inicial de Grafana.

3.4 Análise dos Resultados Experimentais

Nesta secção são apresentados os resultados de testes de desempenho experimentais referentes dois ambientes distintos, envolvendo a plataforma sem servidor, Apache OpenWhisk localmente junto com a ferramenta Kubernetes e o provedor de serviço de nuvem pública, a sua plataforma sem servidor da IBM Cloud Functions/OpenWhisk.

3.4.1 Caraterização de Teste

As comparações de desempenho envolveram as duas plataformas sem servidor conforme proposta, ou seja, localmente a plataforma Apache OpenWhisk integrada no cluster de Kubernetes e provedor de serviço pública da IBM, na sua plataforma sem servidor IBM Cloud Functions/OpenWhisk. Na IBM as configurações foram feitas a partir de Interface de utilizador baseada em Web da IBM Cloud Functions que permite efetuar a gestão de aplicações sem a necessidade de interface da linha de comando. No ambiente local todas execuções foram feitas através da interface da linha de comando da Apache OpenWhisk, denominado de OpenWhisk CLI (wsk).

O propósito final foi de ter feita todas as configurações, avaliar e monitorizar o desempenho de tempo de respostas de execuções de funções (actions), localmente usando a ferramenta *Prometheus* e *Grafana*, na Cloud foi através de IBM Cloud Function Dashoard. Tendo em conta algumas limitações da plataforma Apache OpenWhisk relativamente aos detalhes dos sistemas, isto é, as quantidades de memoria que é alocada por ações, e número ações invocadas por minutos, que difere da IBM Cloud Funrions, onde esses valores são mais alto, deste modo, o experimento baseou-se nos valores padrão das duas plataformas, principalmente na alocação de memória (256 MB) e tempo de invocação por sessenta segundos (60s).

Foi criada uma função que pegasse *API Rest de Países* e retornasse ao utilizador dados de cada país, tendo em conta cada continente, cidade, população e língua. A função foi escrita em linguagem PHP e foram executadas várias invocações e repetidos os mesmos testes que foram nomeados como teste 1, teste 2 e teste 3, de maneira a garantir os resultados. Ainda foi escrita uma função que gerasse números primos abaixo de 1000.000 (um milhões), de maneira a fornecer um resultado que não interferisse na latência de rede, também com as respetivas sequências de testes mencionados.

3.4.2 Comparação Experimental do Desempenho

A figura 3.17 e 3.16 apresentam os resultados de invocações de funções, conforme referenciado anteriormente, que foi usada a linguagem PHP, a latência baseada em tempo de respostas e realizado vários testes que foram definidos como teste 1, teste 2 e teste 3.

onde a primeira figura apresenta o resultado 30 invocações e a segunda o resultado de 50 invocações, e os resultados estão apresentados por médias e milissegundos.

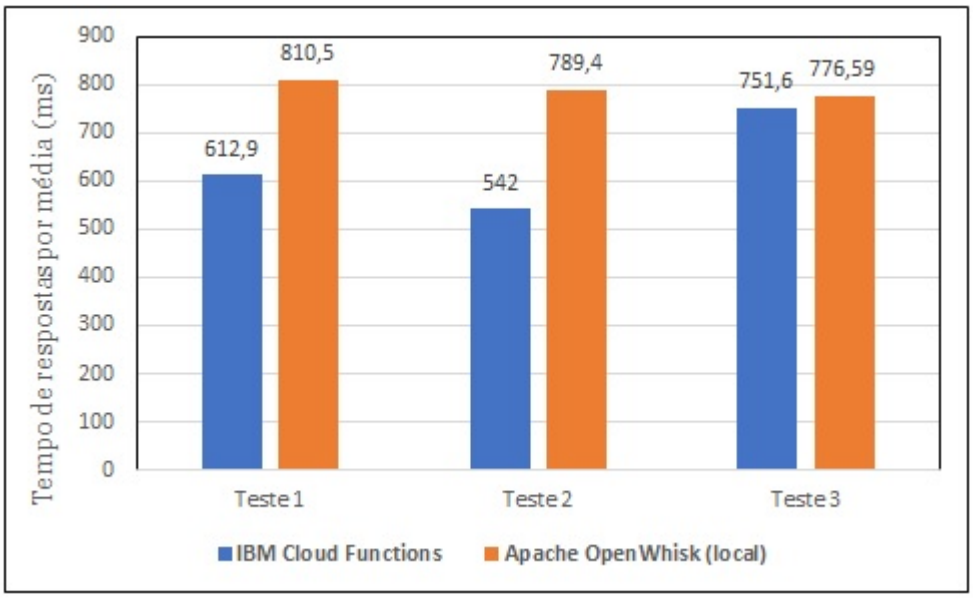

Figura 3.16: Latência por tempo de respostas (30 invocações).

A figura 3.16 mostra-nos que a plataforma sem servidor IBM Cloud Functions apresentou a latência de tempo de resposta inferior à Apache OpenWhisk na instalação local, sendo que no Teste1 apresenta uma diferença de 13,88 %, no Teste 2 de 18,58 % e no Teste 3 de 1,65%.

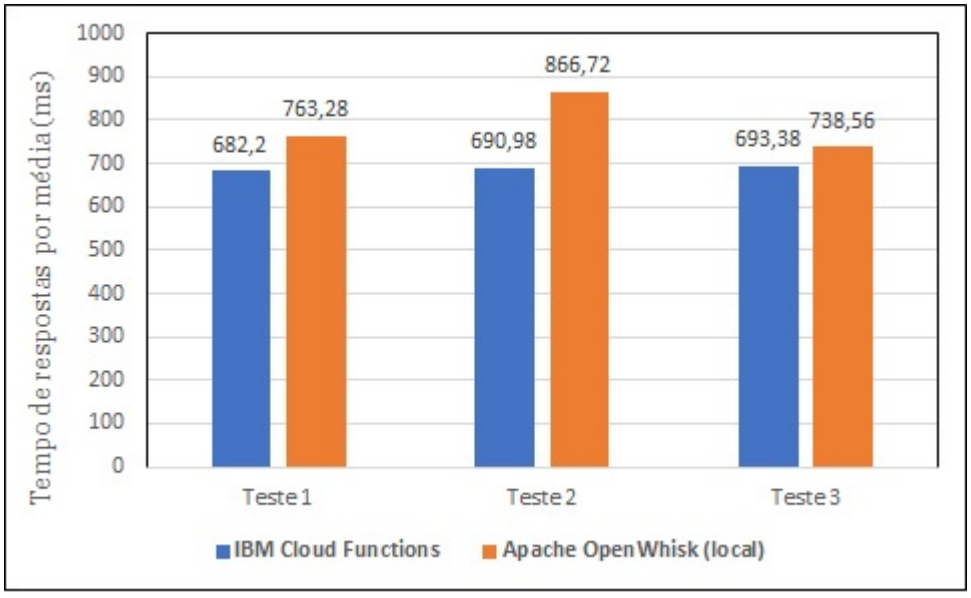

Figura 3.17: Latência por tempo de respostas (50 invocações).

A figura 3.17 envolve 50 invocações, pode ver-se que a plataforma sem servidor IBMCloud-Functions apresentou a latência de tempo de resposta inferior à Apache OpenWhisk na instalação local, sendo que no Teste1 apresenta uma diferença de 5,6%, Teste 2 de 11,28%

e Teste 3 de 3,15%.

A figura 3.18 e 3.19 apresentam a comparação de desempenho em ambas plataformas, conforme já referenciadas anteriormente, que mesmo baseou-se na escrita de uma função que gerasse números primos. É possível notar que em ambas figuras foram definidas Teste 1, Teste 2 e Teste 3, também com 30 e 50 invocações.

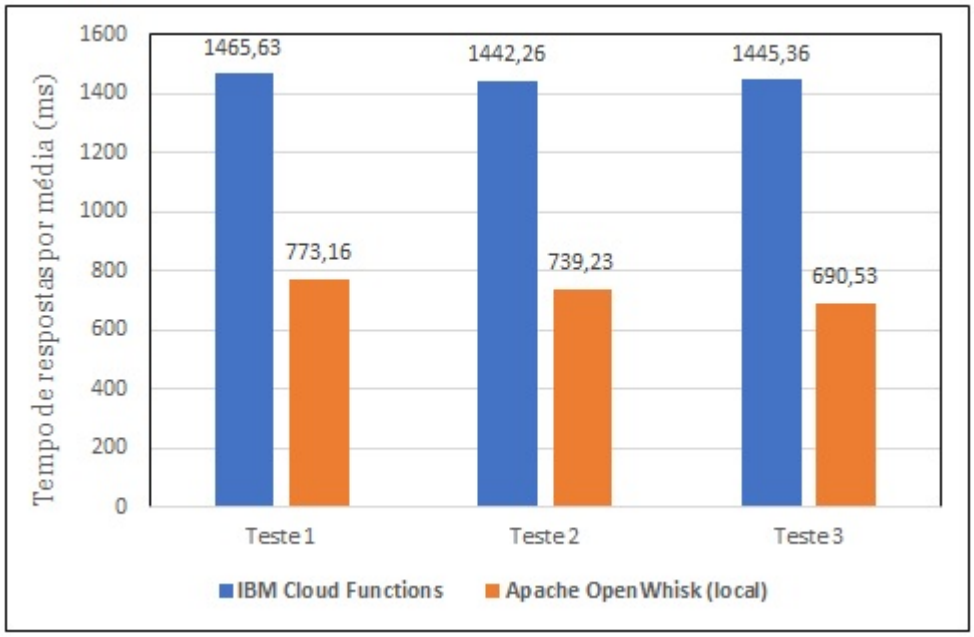

Figura 3.18: Latência por tempo de respostas (30 invocações).

Os resultados apresentados na figura 3.18, com 30 invocações, mostram que a implementação local com a plataforma Apache OpenWhisk apresentou a latência de tempo de respostas inferior à Plataforma IBM Cloud Functions, sendo que o Teste nomeado como 1 apresenta uma diferença de 30,93%, Teste 2 de 32,22% e o Teste 3 de 35,34%.

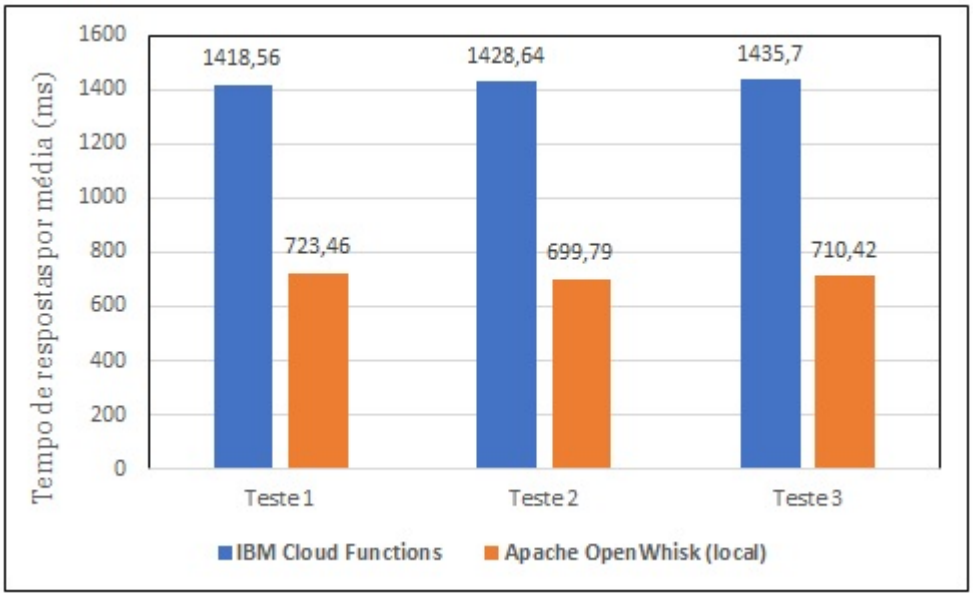

Figura 3.19: Latência por tempo de respostas (50 invocações).

Recorrendo aos resultados da figura 3.19, com 50 invocações, nota-se que a implementação local com a plataforma Apache OpenWhisk apresentou a latência de tempo de respostas inferior à Plataforma IBM Cloud Functions, sendo que no Teste nomeado como 1 apresenta uma diferença de 32,45%, Teste 2 de 34,24% e o Teste 3 de 33,79%.

Analisando o resultado das experiências realizadas, percebe-se que o desempenho da latência de rede influencia o desempenho referente ao tempo de respostas de invocações de funções. Os resultados apresentados nas figuras 3.16 e 3.17, em que foram influenciados pela latência de rede, a plataforma IBM Cloud Functions apresentou uma diferença ligeiramente inferior em relação à implementação local, envolvendo Apache OpenWhisk. As figuras 3.18 e 3.19 apresentam os resultados sem influência da latência de rede, a implementação local apresenta tempos de respostas muito inferiores comparativamente com a IBM Cloud Functions.

3.5 Conclusão

Neste capítulo foram apresentados os ambientes experimentais inicialmente propostos para a realização do trabalho, incluindo a sua caraterização, especificações de hardware e o software utilizado na sua implementação. O ambiente local que envolveu o Apache OpenWhisk e a ferramenta Kubernetes foi mais detalhado por requerer várias tecnologias e uma ligeira complexidade na sua implementação, foram também apresentadas as ferramentas de análise de de desempenho para invocações de funções, análise dos resultados experimentais e a comparação experimental do desempenho das duas plataformas em estudo.

Capítulo 4

Conclusão

4.1 Principais Conclusões

A Plataforma sem servidor é uma etapa importante na evolução da computação em nuvem. A plataforma visa abstrair a gestão de servidores e as decisões de infraestrutura de baixo nível dos programadores e um serviço real com pagamento de acordo com o uso, sem desperdício de recursos e reduzir a barreira para os programadores, confiando no provedor de nuvem para lidar com todas as suas complexidades operacionais. A plataforma é diferente das tecnologias de computação tradicionais, sendo que apenas é paga pelo tempo que o código atenda às requisições. Apache Openwhisk faz parte das plataformas open source e é usado como base para gestão de algumas plataformas sem servidor comercial, especificamente a IBM Cloud, através da sua plataforma sem servidor IBM Cloud Functions. Pode ser opção para clientes que não pretendam depender de fornecedores de nuvem pública, pode ser instalada localmente ou numa nuvem privada e para sua implementação é recomendada a ferramenta Kubernetes para ambiente de produção.

Implementar um ambiente sem servidor através da plataforma Apache OpenWhisk com Kubernetes e posteriormente comparar o desempenho com a plataforma sem servidor da IBM Cloud Functions do provedor de serviço de nuvem pública da IBM que usa Apache OpenWshisk como gestão de funções com serviços foi o objetivo principal deste estudo. Com base na experiência realizada entre duas plataformas, de acordo com as métricas recolhidas, permitiu-se perceber que o ambiente local apresentou o melhor desempenho em relação à IBM Cloud Functions na execução de função que gerasse números primos, numa diferença superior, na execução de função que pegasse e retornasse os valores de Rest API de Países, a IBM Cloud Functions apresentou um desempenho superior ligeiro em relação à implementação local.

4.2 Sugestões para Trabalho Futuro

A abordagem apresentada neste trabalho pode ser expandida e mais pesquisas podem ser realizadas. Portanto, algumas direções interessantes para trabalho futuro incluem envolver a criação de ambiente sem servidor na plataforma Apache OpenWhisk no cluster Kubernetes numa nuvem privada usando OpenStack, a mesma implementação numa provedora de serviço pública envolvendo as duas e alargando os números de testes.

Bibliografia

[AKC19] Palade Andrei, Aqeel Kazmi, and Siobhan Clarke. An evaluation of open source serverless computing frameworks support at the Edge. *Proceedings - 2019 IEEE World Congress on Services, SERVICES 2019*, pages 206–211, 2019. 27, 28

[Ans16] Ravi Gupta Anshu Awasthi. Multiple hypervisor based Open Stack cloud and VM migration. *2016 IEEE 6th International Conference - Cloud System and Big Data Engineering (Confluence)*, 2016. 8

[Apa20] Apache OpenWhisk. Open Source Serverless Cloud Platform, 2020. Available from: `https://openwhisk.apache.org/` [cited 19-11-2020]. xv, 28, 29, 30, 32

[Apa21a] Apache OpenWhisk. Metrics and prometheus support, 2021. Available from: `https://github.com/apache/openwhisk-deploy-kube/blob/master/docs/configurationChoices.md` [cited 19-03-2021]. 47

[Apa21b] Apache OpenWhisk. OpenWhisk CLI, 2021. Available from: `https://github.com/apache/openwhisk/blob/master/docs/cli.md{#}openwhisk-cli` [cited 02-04-2021]. 44

[AWS] AWS. AWS Lambda Features. Available from: `https://aws.amazon.com/lambda/features/`. 22, 23

[AWS21] AWS Lambda. Lambda quotas. 2021. Available from: `https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html` [cited 15-03-2021]. 25

[Azu21a] Azure. Documentação das Funções do Azure, 2021. Available from: `https://docs.microsoft.com/pt-pt/azure/azure-functions/` [cited 15-12-2020]. 24

[Azu21b] Azure Functions. Opções de hospedagem de funções Azure Functions. 2021. Available from: `https://docs.microsoft.com/pt-pt/azure/azure-functions/functions-scale` [cited 15-03-2021]. 25

[BCC+17] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, and Philippe Suter. Serverless computing: Current trends and open problems. *Research Advances in Cloud Computing*, pages 1–20, 2017. xv, 20, 21, 22

[BDK+17] Nilton Bila, Paolo Dettori, Ali Kanso, Yuji Watanabe, and Alaa Youssef. Leveraging the Serverless Architecture for Securing Linux Containers.

Proceedings - IEEE 37th International Conference on Distributed Computing Systems Workshops, ICDCSW 2017, pages 401–404, 2017. 34

[BIID20] Robert Botez, Calin Marian Iurian, Iustin Alexandru Ivanciu, and Virgil Dobrota. Deploying a Dockerized Application with Kubernetes on Google Cloud Platform. *2020 13th International Conference on Communications, COMM 2020 - Proceedings*, pages 471–476, 2020. xv, 14, 15

[BMS20] David Balla, Markosz Maliosz, and Csaba Simon. Open Source FaaS Performance Aspects. *2020 43rd International Conference on Telecommunications and Signal Processing, TSP 2020*, pages 358–364, 2020. 26

[BPGLR+19] Daniel Barcelona-Pons, Pedro García-López, Álvaro Ruiz, Amanda Gómez-Gómez, Gerard París, and Marc Sánchez-Artigas. FaAS orchestration of parallel workloads. *WOSC 2019 - Proceedings of the 2019 5th International Workshop on Serverless Computing, Part of Middleware 2019*, pages 25–30, 2019. 33

[Clo21] Cloud Functions. Quotas. 2021. Available from: `https://cloud.google.com/functions/quotas` [cited 15-03-2021]. 25

[CNC] CNCF Serverless Working Group. CNCF WG-Serverless Whitepaper v1.0. Available from: `https://github.com/cncf/wg-serverless/blob/master/whitepapers/serverless-overview/cncf_serverless_whitepaper_v1.0.pdf` [cited 19-11-2020]. 20, 21

[DKP20] Karim Djemame, Email Kdjemameleedsacuk, and Matthew Parker. Open-source Serverless Architectures : an Evaluation of Apache OpenWhisk. pages 329–335, 2020. 1, 35

[Doc20a] Docker. Docker overview, 2020. Available from: `https://docs.docker.com/get-started/overview/` [cited 23-12-2020]. xv, 10, 11, 12

[Doc20b] Docker. What is Kubernetes, 2020. Available from: `https://www.docker.com/products/kubernetes` [cited 15-12-2020]. 13

[EK16] Michael Eder and Holger Kinkelin. Hypervisor- vs. Container-based Virtualization. *Network Architectures and Services*, (July):1–7, 2016. Available from: `https://www.net.in.tum.de/fileadmin/TUM/NET/NET-2016-07-1/NET-2016-07-1_01.pdf`. 8, 9

[Fis20] Fission. Documentation. 2020. Available from: `https://docs.fission.io/docs/` [cited 15-12-2020]. 27

[FJFCSG+20] Rafael Fayos-Jordan, Santiago Felici-Castell, Jaume Segura-Garcia, Jesus Lopez-Ballester, and Maximo Cobos. Performance comparison of container orchestration platforms with low cost devices in the fog, assisting

Internet of Things applications. *Journal of Network and Computer Applications*, 169(July):102788, 2020. Available from: https://doi.org/10.1016/j.jnca.2020.102788. 14

[GADP14] Stefan Groesbrink, Luis Almeida, Mario De Sousa, and Stefan M. Petters. Towards certifiable adaptive reservations for hypervisor-based virtualization. *Real-Time Technology and Applications - Proceedings*, 2014-October(October):13–24, 2014. 8

[Goo] Google Cloud. Cloud Functions Execution Environment. Available from: https://cloud.google.com/functions/docs/concepts/exec [cited 15-12-2020]. 23

[IAR20] Md Shazibul Islam Shamim, Farzana Ahamed Bhuiyan, and Akond Rahman. XI Commandments of kubernetes security: A systematization of knowledge related to kubernetes security practices. *Proceedings - 2020 IEEE Secure Development, SecDev 2020*, pages 58–64, 2020. 13

[IBM19] IBM Cloud Education. Virtualization, 2019. Available from: https://www.ibm.com/cloud/learn/virtualization-a-complete-guide [cited 23-11-2020]. 7

[IBM20a] IBM. Serverless Computing. 2020. Available from: https://researcher.watson.ibm.com/researcher/view{_}group{_}subpage.php?id=9368 [cited 19-11-2020]. 29

[IBM20b] IBM Cloud. Apache OpenWhisk, 2020. Available from: https://developer.ibm.com/components/apache-openwhisk/. 29

[IBM20c] IBM Cloud. IBM Cloud Functions, 2020. Available from: https://cloud.ibm.com/functions/ [cited 15-12-2020]. 24, 25

[IBM21a] IBM Cloud. Account types, 2021. Available from: https://cloud.ibm.com/docs/account?topic=account-accounts [cited 27-04-2021]. 45

[IBM21b] IBM Cloud Functions. Detalhes e limites do sistema, 2021. Available from: https://cloud.ibm.com/docs/openwhisk?topic=openwhisk-limits [cited 15-03-2021]. 25

[Kae18] Chanwit Kaewkasi. *Docker for Serverless Applications: Containerize and orchestrate functions using OpenFaas, OpenWhisk, and Fn*. Packt Publishing, 2018. 32, 33

[Kal21] Shubham Kale. Cloud Computing – Types of Cloud. 2021. Available from: https://www.esds.co.in/blog/cloud-computing-types-cloud/{#}sthash.QsyM2lKU.QYXMoHu2.dpbs [cited 18-02-2021]. 19, 20

[KGB21] Daniel Kelly, Frank G. Glavin, and Enda Barrett. Denial of wallet—Defining a looming threat to serverless computing. *Journal of Information Security and Applications*, 60(June):102843, 2021. Available from: https://doi.org/10.1016/j.jisa.2021.102843. 1

[KNJ18] Aleksandr Kuntsevich, Pezhman Nasirifard, and Hans Arno Jacobsen. Demo abstract: A distributed analysis and benchmarking framework for apache openwhisk serverless platform. *Middleware 2018 - Proceedings of the 2018 ACM/IFIP/USENIX Middleware Conference (Posters)*, pages 3–4, 2018. 33

[KPR20] Ali Akbar Khatami, Yudha Purwanto, and Muhammad Faris Ruriawan. High availability storage server with kubernetes. *2020 International Conference on Information Technology Systems and Innovation, ICITSI 2020 - Proceedings*, pages 74–78, 2020. 14

[Kub20a] Kubeless. Documentation, 2020. Available from: https://kubeless.io/docs/ [cited 15-12-2020]. 27, 28

[Kub20b] Kubernetes. Production-Grade Container Orchestration. 2020. Available from: https://kubernetes.io/ [cited 15-12-2020]. 12

[Kub20c] Kubernetes. What is Kubernetes, 2020. Available from: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/ [cited 15-12-2020]. 13

[Kub21] Kubernetes. Creating a cluster with kubeadm, 2021. Available from: https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm/ [cited 05-04-2021]. 40

[Lab21] Grafana Labs. What is Grafana. 2021. Available from: https://grafana.com/docs/grafana/latest/getting-started/ [cited 19-04-2021]. 47

[LBG18] Ashish Lingayat, Ranjana R. Badre, and Anil Kumar Gupta. Performance Evaluation for Deploying Docker Containers on Baremetal and Virtual Machine. *Proceedings of the 3rd International Conference on Communication and Electronics Systems, ICCES 2018*, (Icces):1019–1023, 2018. 10

[Li20] Yu Li. Towards fast prototyping of cloud-based environmental decision support systems for environmental scientists using R Shiny and Docker. *Environmental Modelling and Software*, 132, 2020. 10

[LKW+18] Wei Tsung Lin, Chandra Krintz, Rich Wolski, Michael Zhang, Xiaogang Cai, Tongjun Li, and Weijin Xu. Tracking causal order in AWS lambda applications. *Proceedings - 2018 IEEE International Conference on Cloud Engineering, IC2E 2018*, pages 50–60, 2018. Available from: ABC2018. 22

[Lou18] André Filipe Castro Louro. Comparação do Desempenho de Infraestruturas Virtualizadas de Elevada Disponibilidade Usando Hypervisors Nativos ou Containers. 2018. Available from: https://ubibliorum.ubi.pt/handle/10400.6/9837 [cited 23-11-2020]. 7

[LRLE17] Theo Lynn, Pierangelo Rosati, Arnaud Lejeune, and Vincent Emeakaroha. A Preliminary Review of Enterprise Serverless Cloud Computing (Function-as-a-Service) Platforms. *Proceedings of the International Conference on Cloud Computing Technology and Science, CloudCom*, 2017-Decem:162–169, 2017. 23

[LSF18] Hyungro Lee, Kumar Satyam, and Geoffrey Fox. Evaluation of Production Serverless Computing Environments. *IEEE International Conference on Cloud Computing, CLOUD*, 2018-July:442–450, 2018. 35

[Mar19a] Francesco Marchioni. *Hands-On Cloud-Native Applications with Java and Quarkus*. Packt Publishing, 2019. 15

[Mar19b] Horácio José Morgado Martins. PLATAFORMAS DE COMPUTAÇÃO SERVERLESS : Estudo e Benchmark, 2019. Available from: http://hdl.handle.net/10316/86362 [cited 09-12-2020]. 34

[Mar20] Lennart Frantzell Marek Sadowski. *Serverless Swift: Apache OpenWhisk for iOS developers*. Apress, 1 edition, 2020. 30

[MB17] Garrett McGrath and Paul R. Brenner. Serverless Computing: Design, Implementation, and Performance. *Proceedings - IEEE 37th International Conference on Distributed Computing Systems Workshops, ICDCSW 2017*, pages 405–410, 2017. 34

[Mic19] Microsoft. Supported languages in Azure Functions, 2019. Available from: https://docs.microsoft.com/en-us/azure/azure-functions/supported-languages [cited 15-12-2020]. 24

[MPD18] Sunil Kumar Mohanty, Gopika Premsankar, and Mario Di Francesco. An evaluation of open source serverless computing frameworks. *Proceedings of the International Conference on Cloud Computing Technology and Science, CloudCom*, 2018-Decem:115–120, 2018. 26, 27, 28

[Nal14] Krishnan Nallaperumal. Engineering Research Methodology A Computer Science and Engineering and Information and Communication Technologies Perspective. (December 2013), 2014. 2

[NRdA+20] Diana M. Naranjo, Sebastián Risco, Carlos de Alfonso, Alfonso Pérez, Ignacio Blanquer, and Germán Moltó. Accelerated serverless computing based on GPU virtualization. *Journal of Parallel and Distributed Computing*, 139:32–42, 2020. Available from: https://doi.org/10.1016/j.jpdc.2020.01.004. 20

[NT20] Jussi Nupponen and Davide Taibi. Serverless: What it Is, What to Do and What Not to Do. *Proceedings - 2020 IEEE International Conference on Software Architecture Companion, ICSA-C 2020*, pages 49–50, 2020. 21

[Ope19] OpenWhisk. OpenWhisk system details, 2019. Available from: https://github.com/apache/openwhisk/blob/master/docs/reference.md{#}system-limits [cited 25-02-2021]. 31

[Ope20a] OpenFaaS. OpenFaaS - Serverless Functions Made Simple, 2020. Available from: https://docs.openfaas.com/ [cited 17-12-2020]. 26

[Ope20b] OpenWhisk. Documentation. 2020. Available from: https://openwhisk.apache.org/documentation.html [cited 19-11-2020]. 28, 30, 31

[Pet11] Peter Mell Timothy Grance. The NIST Definition of Cloud Computing. *Special Publication (NIST SP), National Institute of Standards and Technology, Gaithersburg, MD*, 2011. Available from: https://doi.org/10.6028/NIST.SP.800-145 [cited 18-02-2021]. 16, 18, 19, 20

[PKC19] Andrei Palade, Aqeel Kazmi, and Siobhan Clarke. An evaluation of open source serverless computing frameworks support at the Edge. *Proceedings - 2019 IEEE World Congress on Services, SERVICES 2019*, pages 206–211, 2019. 1

[Pro21] Prometheus. OVERVIEW/What is Prometheus, 2021. Available from: https://prometheus.io/docs/introduction/overview/ [cited 19-04-2021]. 46

[PTRC07] Ken Peffers, Tuure Tuunanen, Marcus A. Rothenberger, and Samir Chatterjee. A design science research methodology for information systems research. *Journal of Management Information Systems*, 24(3):45–77, 2007. xv, 2

[QMRD19] Sebastian Quevedo, Freddy Merchan, Rafael Rivadeneira, and Federico X. Dominguez. Evaluating Apache OpenWhisk - FaaS. *2019 IEEE 4th Ecuador Technical Chapters Meeting, ETCM 2019*, 2019. 34

[Red] Red Hat. Cloud-native applications. Available from: https://www.redhat.com/en/topics/cloud-native-apps/what-is-serverless [cited 20-12-2020]. 21, 22

[Red20] Red Hat. Containers vs VMs, 2020. Available from: https://www.redhat.com/en/topics/containers/containers-vs-vms [cited 23-11-2020]. 9

[Res19] ResellerClub. Type 1 and Type 2 Hypervisors: What Makes Them Different, 2019. Available from: https://medium.com/teamresellerclub/

type-1-and-type-2-hypervisors-what-makes-them-different-6a1755d6ae2c [cited 09-02-2021]. xv, 8

[Rib20] Ran Ribenzaft. Serverless Open-Source Frameworks, 2020. Available from: https://epsagon.com/tools/serverless-open-source-frameworks-openfaas-knative-more/ [cited 15-12-2020]. 26

[SB19] Nitin Sukhija and Elizabeth Bautista. Towards a framework for monitoring and analyzing high performance computing environments using kubernetes and prometheus. *Proceedings - 2019 IEEE SmartWorld, Ubiquitous Intelligence and Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Internet of People and Smart City Innovation, SmartWorld/UIC/ATC/SCALCOM/IOP/SCI 2019*, pages 257–262, 2019. 13

[SGKH20] Huaxin Shi, Xiaofeng Gu, Ping Kuang, and Hongyu Huang. An Improved Kubernetes Scheduling Algorithm for Deep Learning Platform. *2020 17th International Computer Conference on Wavelet Active Media Technology and Information Processing, ICCWAMTIP 2020*, pages 113–116, 2020. 13, 15

[SLC20] Sogand Shirinbab, Lars Lundberg, and Emiliano Casalicchio. Performance evaluation of containers and virtual machines when running Cassandra workload concurrently. *Concurrency Computation*, 32(17):1–14, 2020. 10

[SS18] Mohit Sewak and Sachchidan Singh. Winning in the Era of Serverless Computing and Function as a Service. *2018 3rd International Conference for Convergence in Technology, I2CT 2018*, pages 1–5, 2018. 1, 24

[The20] The Apache Software Foundation. System overview, 2020. Available from: https://github.com/apache/openwhisk/blob/master/docs/about.md{#}how-openWhisk-works [cited 19-11-2020]. xv, 31, 32, 33

[Tho18] Stefan Thorpe. Leveraging Serverless Architecture, 2018. Available from: https://dzone.com/articles/leveraging-serverless-architecture [cited 02-12-2020]. 21

[Vmw20] Vmware. Hypervisor. 2020. Available from: https://www.vmware.com/topics/glossary/content/hypervisor [cited 19-11-2020]. 8, 9

[Yfa20] Vissarion Yfantis. What is a hypervisor and its benefits, 2020. Available from: https://www.parallels.com/blogs/ras/hypervisor/. 8

[ZCB10] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing: State-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1(1):7–18, 2010. xv, 16, 17

[ZLL21] Song Zhang, Xiaochuan Luo, and Eugene Litvinov. Serverless computing for cloud-based power grid emergency generation dispatch. *International Journal of Electrical Power and Energy Systems*, 124(April 2020):106366, 2021. Available from: https://doi.org/10.1016/j.ijepes.2020.106366. 20

[ZLP16] Jie Zhang, Xiaoyi Lu, and Dhabaleswar K. Panda. Performance characterization of hypervisor-and container-based virtualization for HPC on SR-IOV enabled infiniband clusters. *Proceedings - 2016 IEEE 30th International Parallel and Distributed Processing Symposium, IPDPS 2016*, pages 1777–1784, 2016. 7, 10