Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Framework Node2FaaS: Uma Abordagem Eficiente para Conversão Automática de Aplicações NodeJS para Function as a Service

Leonardo Rebouças de Carvalho

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Orientadora
Prof.a Dr.a Aletéia Patrícia Favacho de Araújo Von Paumgartten

Brasília
2020

Ficha Catalográfica de Teses e Dissertações

Está página existe apenas para indicar onde a ficha catalográfica gerada para dissertações de mestrado e teses de doutorado defendidas na UnB. A Biblioteca Central é responsável pela ficha, mais informações nos sítios:

http://www.bce.unb.br
http://www.bce.unb.br/elaboracao-de-fichas-catalograficas-de-teses-e-dissertacoes

Esta página não deve ser inclusa na versão final do texto.

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Framework Node2FaaS: Uma Abordagem Eficiente para Conversão Automática de Aplicações NodeJS para Function as a Service

Leonardo Rebouças de Carvalho

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Prof.a Dr.a Aletéia Patrícia Favacho de Araújo Von Paumgartten (Orientadora)
CIC/UnB

Prof.a Dr.a Alba Cristina M. A. de Melo
CIC/UnB

Prof. Dr. Josef Spillner
Zurich University of Applied Sciences/Suiça

Prof.a Dr.a Genaina Nunes Rodrigues
Coordenadora do Programa de Pós-graduação em Informática

Brasília, 05 de Novembro de 2020

Dedicatória

Dedico este trabalho a Susana Rebouças da Silva, minha mãe, cuja capacidade de se auto replicar para resolver vários problemas ao mesmo tempo e ainda assim parecer uma única pessoa sempre me faz lembrar como as nuvens computacionais funcionam. Tudo que me tornei para chegar até aqui foi fruto de estímulos fornecidos por ela desde a minha infância.

Agradecimentos

Primeiramente preciso agradecer aos meus pais que me cederam, cada um, metade do que eu sou, sem o sacrifício deles eu não existiria e tampouco este trabalho. Agradeço a Esteyse Glenaise Santana Carneiro pela parceria, paciência e companheirismo durante a execução deste trabalho, além das diversas revisões e contribuições que promoveu ao trabalho. Agradeço a Estela Barros Oliveira pela amizade, apoio e revisões realizadas durante a finalização deste trabalho. Agradeço aos meus colegas na Dataprev com os quais compartilhei alguns desafios enfrentados durante esta empreitada e que sempre me direcionaram a enxergar os problemas a partir de outros ângulos, enriquecendo as soluções, bem como a minha própria capacidade analítica. Agradeço ainda às empresas Red Bull e Coca Cola, cujos produtos me mantiveram acordado em diversas madrugadas que foram fundamentais para a evolução desta obra. Registro ainda um agradecimento especial às professoras Dra Priscila Barreto e a Dra Alba Melo por terem contribuído com este trabalho sendo membros da banca de qualificação e sugerido diversos aprimoramentos que enriqueceram este trabalho. Agradeço ao professor Dr. Paulo Faleiros que, juntamente com a Dra Alba Melo e a Dra Aletéia Araújo, integra o grupo de professores com os quais tive o privilégio de obter conhecimentos por meio de aulas no curso de Mestrado. Durante essas aulas pude, verdadeiramente, compreender o significado da ciência e sua importância. Por último, agradeço a minha orientadora, a professora Dra Aletéia Araújo, com quem é inevitável não desenvolver um vínculo de amizade a partir de uma certa convivência. Os direcionamentos dela foram fundamentais não apenas para o sucesso deste trabalho, mas também para o desencadeamento de uma verdadeira sede pelo conhecimento, sobretudo pelo fato de perceber o potencial transformador que o conhecimento pode imbuir quando bem aplicado.

Resumo

A computação em nuvem emergiu na área da ciência da computação com a proposta de significativa redução de custos e de tempo para operacionalização de infraestrutura. Dentre os diversos modelos de nuvem disponíveis, este trabalho destaca o *Function as a Service*, função como um serviço (FaaS), e propõe o *framework* Node2FaaS. O objetivo é promover a conversão automática e eficiente de aplicações escritas em NodeJS para trabalharem, de maneira transparente, com o modelo FaaS. Os experimentos mostraram que a adoção da abordagem proposta pelo Node2FaaS se converte em ganhos significativos, acima de 90%, no tempo de execução para aplicações com perfil de utilização intensiva de CPU, memória ou manipulação de disco.

Palavras-chave: Computação em Nuvem, FaaS, Function as a Service, NodeJS, Conversor Automático.

Abstract

Cloud computing emerged in the area of computer science with the proposal to significantly reduce costs and time for operationalization of infrastructure. Among the various cloud models available, this work highlights Function as a Service (FaaS) and proposes the Node2FaaS framework. The objective is to promote the automatic and efficient conversion of applications written in NodeJS to work, in a transparent way, with the FaaS model. The experiments showed that adoption of the approach proposed by Node2FaaS translates into significant gains of above 90% in the execution time for applications with a profile of intensive CPU usage, memory or disk manipulation.

Keywords: Cloud Computing, Function as a Service - FaaS, NodeJS, Automatic Conversor.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

AaaS *Analytics as a Service*, painéis analíticos como serviço.

AKS *Azure Kubernetes Service*, serviço de Kubernetes da Azure.

API *Application Programming Interface*, interface de programação de aplicações.

AWS *Amazon Web Services*.

BaaS *Backup as a Service*, cópia de segurança como serviço.

CaaS *Container as a Service*, contêiner como serviço.

CDN *Content Deliver Network*.

CIC Departamento de Ciência da Computação.

CLI *Command Line Interface*, interface por linha de comando.

CPU *Central Processing Unit*, unidade central de processamento.

CSAR *Cloud Service Archives*, arquivos de serviço em nuvem.

CSS *Cascading Style Sheets*.

DaaS *Data/Desktop/Database as a Service*, dados, área de trabalho ou banco de dados como serviço.

DevOps *Development + Operations*, união entre equipes de desenvolvimento e operação.

DNS *Domain Name Service*.

DRaaS *Disaster Recover as a Service*, recuperação de desastres como serviço.

DSL *Domain Specific Language*, linguagem de domínio específico.

EC2 *Elastic Compute Cloud*.

ETL *Extract, Transform and Load.*

FaaS *Function as a Service*, função como um serviço.

GCE *Google Compute Engine.*

GCP *Google Cloud Platform.*

HCL *HashiCorp Configuration Language).*

HOT *Heat Orchestration Template*, modelo de orquestração do Heat.

HPC *High-performance Computing*, computação de alto desempenho.

HTML *HyperText Markup Language.*

HTTP *Hypertext Transfer Protocol.*

I/O *Input/Output.*

IA *Implementation Artifact*, artefato de implementação.

IaaS *Infrastructure as a Service*, Infraestrutura como Serviço.

IaC *Infrastructure as Code*, infraestrutura como código.

IBM *International Business Machines.*

IEEE *Institute of Electrical and Electronic Engineers.*

IoT *Internet of Things*, Internet das Coisas.

IP *Internet Protocol*, protocolo da Internet.

JSON *JavaScript Object Notation.*

LaaS *Log as a Service*, registro ostensivo como serviço.

MaaS *Monitoring as a Service*, monitoramento como serviço.

MLaaS *Machine Learning as a Service*, aprendizado de máquina como serviço.

NASA *National Aeronautics and Space Administration.*

NFV *Network Functions Virtualization*, virtualização das funções da rede.

NIST *National Institute of Standards and Technology*, Instituto Nacional de Padrões e Tecnologia dos Estados Unidos.

noSQL *Not Only Structured Query Language*, bancos de dados não relacionais.

NPM *Node Package Manager*.

PaaS *Platform as a Service*, Plataforma como Serviço.

PDF *Portable Document Format*.

PHP *Hypertext Preprocessor*, pré-processador de hipertexto.

RDS *Relational Database Service*, serviço de banco de dados relacional.

REST *Representational State Transfer*, transferência de estado representacional.

RPC *Remote Procedure Call)*,chamada de procedimento remoto.

S3 *Simple Storage Service*, serviço simples de armazenamento.

SaaS *Software as a Service*, Software como Serviço.

SDKs *Software Development Kits*, coleção de softwares de desenvolvimento.

SDN *Software-Defined Networking*, rede definida por softwares.

SLA *Service Level Agreement*, acordo de nível de serviço.

SQL *Structured Query Language*, linguagem de consulta estruturada.

SSH *Secure Socket Shell*.

SSL *Secure Sockets Layer*, camada de interface segura.

STaaS *Storage as a Service*, armazenamento como serviço.

TACC *Texas Advanced Computing Center*.

TI Tecnologia da Informação.

TOSCA *Topology and Orchestration Specification for Cloud Applications*, especificação de topologia e orquestração para aplicativos em nuvem.

UnB Universidade de Brasília.

URI *Uniform Resource Identifier*, identificador uniforme de recurso.

VoIP *Voice over Internet Protocol*, voz sobre IP.

VPC *Virtual Private Cloud*, nuvem privada virtual.

WSGI *Web Server Gateway Interface*.

XML *Extensible Markup Language*, linguagem de marcação extensível.

YAML *Yaml Ain't Markup Language* (acrônimo auto recursivo).

Capítulo 1

Introdução

Com a proposta de significativa redução de custos e de tempo para disponibilização de infraestrutura, a computação em nuvem ganhou importante espaço na sociedade atual. Antes dessa proposta se tornar realidade, era essencial um considerável investimento na implantação de centros de processamento de dados (*datacenters*), para sustentar o processamento e o armazenamento, necessários para continuidade de projetos que demandavam um grande número de recursos computacionais [30].

Atualmente, o conceito de computação em nuvem é uma realidade bem estabelecida. Diversos provedores oferecem uma gama de serviços por meio da Internet, dispensando grandes investimentos em infraestrutura. Assim, o usuário de um ambiente de nuvem é capaz de ter, em instantes, acesso a um volume de recursos computacionais que há pouco tempo estava restrito às grandes organizações [2].

O importante diferencial econômico da computação em nuvem reside no fato do pagamento ocorrer sob demanda, ou seja, o cliente paga apenas pelo efetivo uso dos recursos, sem a necessidade de manter o investimento alocado durante longos períodos, como ocorria no passado com os investimentos em *datacenters*. Agora, caso uma instituição deseje realizar uma breve pesquisa que necessite de *clusters* com centenas de máquinas, basta contratar um serviço de nuvem pelo tempo necessário para o processamento, e pagar apenas uma fração do que seria o investimento total em infraestrutura.

Além do benefício econômico, esse modelo permite respostas rápidas a mudanças bruscas de comportamento de uma aplicação. Em épocas específicas do ano, como Dia das Mães e Natal, é comum que os *sites* de grandes lojas varejistas experimentem sobrecarga no acesso, causando lentidão e falhas. No entanto, responder rapidamente a esses problemas pode ser a diferença entre efetivar uma venda ou perder um cliente para um *site* concorrente. Assim, a computação em nuvem auxilia na prevenção e na resolução desse tipo de problema por meio da elasticidade, que é a principal característica do ambiente, proporcionando crescimento e diminuição da capacidade computacional da infraestrutura

de acordo com algum parâmetro pré-estabelecido, como utilização de CPU, por exemplo.

Nesse sentido, a computação em nuvem oferece diversas formas de prestação de serviços. Por meio dela, o cliente pode ter acesso a um ambiente virtualizado, cujos componentes de infraestrutura já estejam configurados ou até mesmo toda uma plataforma para desenvolvimento de soluções [2]. Além disso, é possível ter acesso a um software pronto para uso, sem qualquer interação com os detalhes da estrutura interna ou instalação adicional [31].

Essa nova forma de interagir com tecnologia computacional tem afetado tanto os usuários finais, quanto os próprios desenvolvedores de soluções, pois em plataformas diferentes de nuvem, eventualmente, o custo e o tempo necessários para a instalação e a configuração de ambientes de desenvolvimento podem atrasar, ou até mesmo inviabilizar, alguns projetos. Nesse contexto, a computação em nuvem tem contribuído positivamente diante dessa problemática. Assim, cada vez mais organizações têm optado por modelos de computação em nuvem, seja para os processos de negócio, ou para o desenvolvimento de soluções tecnológicas.

Diante disso, tem ganhado espaço um modelo de serviço que encapsula tanto a infraestrutura quanto a própria plataforma de desenvolvimento. Esse modelo entrega ao desenvolvedor apenas uma interface para inclusão de códigos fonte, escritos nas linguagens de programação mais usuais, que serão processados quando forem acionados. Esse modelo de serviço tem sido denominado de *Function as a Service*, função como um serviço (FaaS) [23], mas também pode ser referenciado como *serverless* modelo "sem servidor" [32], uma vez que o desenvolvedor não necessita se preocupar com nenhum aspecto de configuração da plataforma, tampouco da infraestrutura a ser usada.

Apesar dos benefícios que o modelo de FaaS oferece, é preciso ajustar os processos de desenvolvimento para se adequarem a esse novo paradigma. Além disso, alguns tipos de problemas computacionais possuem soluções nas quais a adoção de FaaS pode acarretar prejuízos ao tempo de execução, uma vez que isso adiciona camadas à solução original. Ademais, cada provedor pode ter uma forma particular de oferecer interação com o seus serviços, e o desenvolvedor precisará estar familiarizado para fazer uso desse modelo.

O NodeJS [33] é um ambiente de processamento web do lado do servidor que compartilha a mesma sintaxe da amplamente difundida linguagem de processamento, do lado do cliente das aplicações na Internet, o JavaScript [34]. Considerando a ampla adoção que o NodeJS vem experimentando no cenário tecnológico atual, este trabalho envidou esforços na construção de uma ferramenta que ampliasse ainda mais o rol de possibilidades do NodeJS, facilitando a promoção de software escrito nessa linguagem para FaaS.

Diante do exposto, este trabalho propõe um *framework* que torna o consumo de FaaS menos complexo, o chamado Node2FaaS. O Node2FaaS converte automaticamente apli-

cações escritas para NodeJS para trabalharem com o arquétipo de FaaS, em múltiplos provedores, de forma simples e transparente. Além disso, o *framework* executa uma avaliação do algoritmo das funções, com o propósito de definir, de forma eficiente, se a estrutura interna é adequada para a abordagem de *Function as a Service* e, então, decidir entre o processamento local ou remoto via FaaS.

Para isso, é necessário fornecer apenas as credenciais de um provedor e um código fonte em uma *Command Line Interface*, interface por linha de comando (CLI), e o *framework* será capaz de gerar, automaticamente, uma aplicação baseada na original, na qual o conteúdo das funções definidas aponte para o serviço de FaaS do provedor, caso a função seja considerada aderente ao modelo de FaaS.

Os resultados obtidos foram favoráveis e serão descritos na Seção 1.2, e de maneira detalhada no Capítulo 6.

1.1 Objetivos

O objetivo principal deste trabalho é construir um *framework* que permita, ao desenvolvedor de NodeJS, a conversão automática de aplicações, previamente escritas, para trabalharem com FaaS de forma transparente.

O código fonte definido em cada função deve ser publicado automaticamente no serviço de FaaS, e a respectiva *Uniform Resource Identifier*, identificador uniforme de recurso (URI) retornada pelo provedor deve ser incluída dentro da definição da função original e, então, compor a definição da respectiva função na aplicação convertida. Assim sendo, para cumprir o objetivo geral este trabalho possui os seguinte objetivos específicos:

- Desenvolver um processo prévio de análise de funções, a fim de definir sua aderência ao modelo de FaaS;
- Projetar no *framework* a possibilidade de migrar de um provedor para outro de forma simples e prática;
- Avaliar os orquestradores de nuvem disponíveis atualmente com o objetivo de definir a adoção de um deles como a interface entre o *framework* e os provedores de FaaS.

1.2 Resultados

O *framework* Node2FaaS, desenvolvido neste trabalho, está disponível no GitHub em https://github.com/node2faas e também foi publicado no gerenciador de pacotes NPM em https://www.npmjs.com/package/node2faas podendo ser instalado por meio do comando **npm install node2faas**.

Após revisão dos principais orquestradores de nuvem disponíveis atualmente, bem como um experimento de testes, ficou definido o Terraform [20] como o integrador do Node2FaaS com os provedores de nuvem. Os trabalhos de análise e experimentação dos orquestradores resultaram no artigo *Performance Comparison of Terraform and Cloudify as Multicloud Orchestrators*, aceito para publicação no evento *20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing - CCGRID 2020* [35], em Melbourne, na Austrália, Qualis evento A1.

A fim de avaliar o comportamento do *framework* Node2FaaS, bem como a abordagem adotada por ele, foram realizados dois experimentos práticos. O primeiro, sem a utilização do orquestrador de nuvem, foi realizado apenas sobre um serviço de FaaS e será melhor detalhado na Seção 6.2. Esse experimento mostrou que a adoção da abordagem proposta pelo *framework* Node2FaaS resultou em uma redução de 170% no tempo de execução para aplicações com uso intensivo de leitura e escrita em disco. Além disso, os testes também apresentaram benefícios para aplicações com uso intensivo de memória, porém somente após um certo limiar de concorrência. Já para aplicações com uso intensivo de CPU ou para operações simples, a utilização da abordagem não se mostrou vantajosa. Os resultados desse experimento preliminar, bem como a definição do Node2FaaS foram publicados no evento *9th International Conference on Cloud Computing and Services Science - CLOSER 2019*, realizado na ilha de Creta, na Grécia entre os dias 2 e 4 de maio de 2019, com o título *Framework Node2FaaS: Automatic NodeJS Application Converter for Function as a Service* [36], Qualis evento A2.

Além disso, um segundo experimento foi realizado. Desta vez foram utilizados serviços de FaaS em três provedores diferentes orquestrados pelo Terraform integrado ao *framework* Node2FaaS. Esse experimento será melhor detalhado na Seção 6.3. Os resultados mostraram que houve ganhos no tempo de execução com a utilização do *framework* para aplicações com uso intensivo de CPU, memória e disco em todos os provedores, exceto em um deles, cujo serviço de FaaS não suportou a concorrência dos processos de leitura e escrita de arquivos. Os ganhos no tempo de execução chegaram a 99% em um dos provedores para aplicações com uso intensivo de memória e I/O. Os trabalhos desse experimento resultaram no artigo *Remote Procedure Call Approach using the Node2FaaS Framework with Terraform for Function as a Service* que foi aceito para publicação no *10th International Conference on Cloud Computing and Services Science - CLOSER 2020* com período de realização de 7 a 9 de maio de 2020, Qualis evento A2, excepcionalmente por meio de *streaming online* devido à pandemia da COVID-19.

Em ambos os experimentos, o *framework* Node2FaaS mostrou-se eficaz na tarefa de converter aplicações previamente escritas em NodeJS para operarem internamente utilizando chamadas remotas aos serviços de FaaS, mantendo a mesma assinatura das funções,

bem como o comportamento esperado.

Outro resultado obtido por este trabalho foi um convite para realização de uma visita técnica junto ao *Service Prototyping Lab* (SPLAB) da *Zurich University of Applied Sciences*, na Suiça. Todavia, embora o convite feito pelo Dr. Josef Spillner, que gerencia o laboratório naquela Universidade, tenha sido aceito e a visita fora agendada para ocorrer em Junho de 2020, a mesma precisou ser cancelada devido à pandemia da COVID-19. Apesar disso, os estudos previstos para ocorrer durante a referida visita técnica foram realizados por meio de videoconferência. Esses estudos culminaram na elaboração de um Portal para fomento do processo de FaaSificação, que é definido pelo Dr. Spillner como sendo "o processo de conversão de uma estrutura de código em um formato que é executável no Function as a Service" [37]. A nova comunidade se concentra na plataforma disponível em http://www.faasification.com, cujo desenvolvimento se deu de forma colaborativa entre o SPLAB e a UnB, representada pelo autor desta Dissertação de Mestrado e sua orientadora.

1.3 Descrição deste Documento

Além deste capítulo introdutório, este documento possui outros seis capítulos. No Capítulo 2 são abordados os principais conceitos da computação em nuvem, além de uma visão geral dos mais importantes provedores públicos da atualidade. No Capítulo 3, são apresentados o conceito de *sky computing* e uma descrição dos principais orquestradores comerciais de nuvens, disponíveis atualmente. O Capítulo 4 conceitua o modelo de FaaS, assim como descreve os serviços desse modelo oferecidos por diversos provedores. Além disso, o Capítulo 4 confronta o modelo de desenvolvimento de software monolítico com a arquitetura de microsserviços, e ainda, apresenta uma área que tem se beneficiado significativamente com o modelo de FaaS, a Internet das Coisas. No Capítulo 5 é apresentada a proposta deste trabalho, ou seja, o *framework* Node2FaaS. Nesse capítulo, além da estrutura e do funcionamento do *framework*, é apresentado o processo que levou à definição do Terraform como o orquestrador multicloud integrado ao *framework*. No Capítulo 6 são apresentados os resultados de um experimento preliminar que fora realizado sem a utilização do Terraform e os resultados de um segundo experimento, já com o Terraform. Por último, o Capítulo 7 traz as considerações finais e os trabalhos futuros.

Capítulo 2

Computação em Nuvem

Neste capítulo serão apresentados os principais conceitos de computação em nuvem. O capítulo está dividido em seis seções, sendo a primeira Seção (2.1) reservada para as considerações iniciais, enquanto a última Seção (2.6) é reservada para as considerações finais. Na Seção 2.2 são descritas as características essenciais de nuvem. As Seções 2.3 e 2.4 apresentam as definições para os modelos de implantação e entrega de serviços, respectivamente. A Seção 2.5 descreve em detalhes os três principais provedores públicos de nuvem encontrados atualmente, além de uma visão geral de outros provedores que concorrem nesse disputado mercado.

2.1 Considerações Iniciais

A computação em nuvem trouxe uma forma diferenciada de tratar os recursos computacionais. Orientadas a serviços, as nuvens entregam diversos modelos de utilização de capacidade computacional, abstraindo as complexidades de acordo com a necessidade do cliente. Assim, o usuário decide qual nível de envolvimento deseja ter com o recurso computacional que estiver utilizando, seja uma máquina virtual, com acesso privilegiado às configurações, ou um serviço de correio eletrônico, cuja complexidade operacional é completamente encapsulada, oferecendo ao usuário somente a interface final do serviço.

No modelo tradicional de computação, recursos tais como processadores, memórias, discos e conexão com a rede são manipulados fisicamente. Os processos de manutenção e escala nesse tipo de abordagem, além de caros, são trabalhosos e lentos. Conforme a computação evoluiu, e uma quantidade considerável de poder computacional passou a estar disponível em apenas uma máquina, surgiu a necessidade de aprimorar o gerenciamento interno desses recursos. Nesse cenário, a virtualização de recursos foi a solução encontrada. Essa abordagem sedimentou a base da computação em nuvem, cujas principais características serão apresentadas em detalhes nas próximas seções.

2.2 Características Essenciais de Nuvem

Em 2011, o *National Institute of Standards and Technology*, Instituto Nacional de Padrões e Tecnologia dos Estados Unidos (NIST) [31] elencou algumas características fundamentais para uma plataforma computacional ser classificada como nuvem:

- **Auto serviço sob demanda:** o cliente é capaz de obter recursos computacionais à medida que necessitar, sem qualquer interação humana;
- **Acesso via Internet:** recursos são consumidos a partir de qualquer dispositivo conectado à rede, sejam telefones celulares, *tablets*, *laptops* ou computadores pessoais;
- **Agrupamento de recursos:** os recursos de computação do provedor são agrupados para atender a vários consumidores usando um modelo *multi-tenant*, com diferentes recursos físicos e virtuais, atribuídos e reatribuídos dinamicamente de acordo com a demanda do cliente. Há uma sensação de independência de localização em que o cliente geralmente não tem controle ou conhecimento sobre a localização exata dos recursos fornecidos, mas pode especificar a localização em um nível mais alto de abstração (por exemplo, país, estado ou *datacenter*);
- **Elasticidade rápida:** os recursos podem ser alocados e liberados elasticamente, em alguns casos automaticamente, para escalar rapidamente, de maneira horizontal ou vertical, de acordo com a demanda. Para o consumidor, os recursos disponíveis geralmente parecem ilimitados e podem ser adquiridos em qualquer quantidade, e a qualquer momento; e
- **Serviço monitorado:** os sistemas em nuvem controlam e otimizam automaticamente o uso de recursos, aproveitando um recurso de medição em algum nível de abstração apropriado ao tipo de serviço. O uso de recursos pode ser monitorado, controlado e relatado, fornecendo transparência tanto para o provedor quanto para o cliente do serviço utilizado.

A Figura 2.1 mostra o comportamento de um ambiente perante a demanda ao longo do tempo. É possível observar que a abordagem de escalabilidade tradicional, em especial a horizontal, representa um alto custo de implementação e, ainda assim, eventualmente, pode não atender satisfatoriamente a demanda ocasionando perda de clientes, conforme destacado na área azulada da imagem. Por outro lado, a curva da elasticidade acompanha a curva da demanda tanto nos picos como nos vales, demonstrando a eficiência econômica e funcional da abordagem. Esse comportamento é um dos principais diferenciais estratégicos que tem contribuído para o sucesso das nuvens [1].

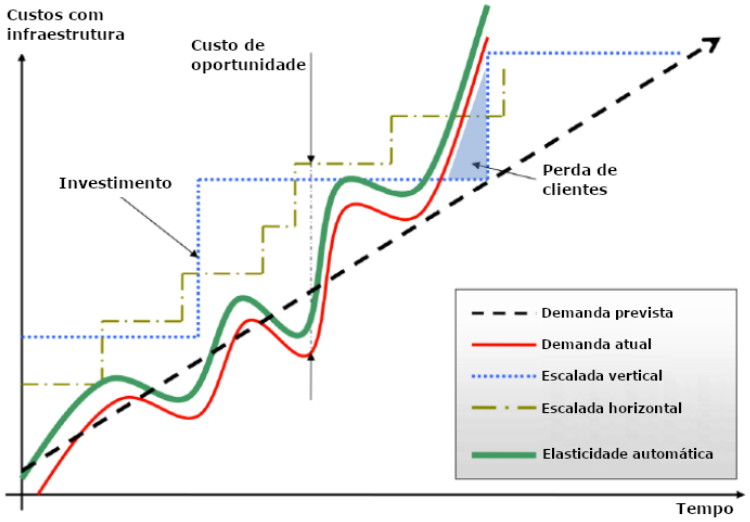

Figura 2.1: Escalabilidade *versus* elasticidade automática, adaptado de [1].

2.3 Modelos de Implantação

Além de definir as principais características de nuvem, o NIST [31] também definiu as formas de implantação do conceito, conforme a seguir:

- **Nuvem pública:** seu uso é aberto para o público em geral. Esse modelo pode ser de propriedade, operada e gerenciada por uma organização comercial, acadêmica ou governamental ou por alguma combinação deles;
- **Nuvem privada:** infraestrutura de uso exclusivo de uma única organização e seus vários consumidores, inclusive unidades de negócio. Ela pode ser implantada dentro ou fora das instalações da instituição, bem como operada e gerida por ela e/ou por terceiros;
- **Nuvem comunitária:** usada por uma comunidade de consumidores, cujas organizações compartilham requisitos como, por exemplo: missão, aspectos de segurança, política e conformidade. A instalação, a operação e a gestão podem ficar a cargo de uma ou mais organizações da comunidade, terceiros ou uma combinação deles. Uma estrutura comunitária pode usar tecnologias como *vCloud Director* [38] ou *OpenStack* [39] para prover um serviço de nuvem para as organizações envolvidas [1];
- **Nuvem híbrida:** composição de duas ou mais infraestruturas de nuvem distintas (privadas, comunitárias ou públicas) que permanecem como entidades exclusivas, mas unidas por tecnologia padronizada ou proprietária que permite a portabilidade de dados e de aplicativos.

A Figura 2.2 apresenta os três principais modelos de implantação. É possível observar que a utilização de nuvens públicas é aderente à projetos efêmeros, com curta duração, enquanto a utilização de nuvens privadas está associada a projetos de longa duração. Entre os dois diferentes paradigmas encontra-se as nuvens híbridas, mesclando ambas as abordagens.

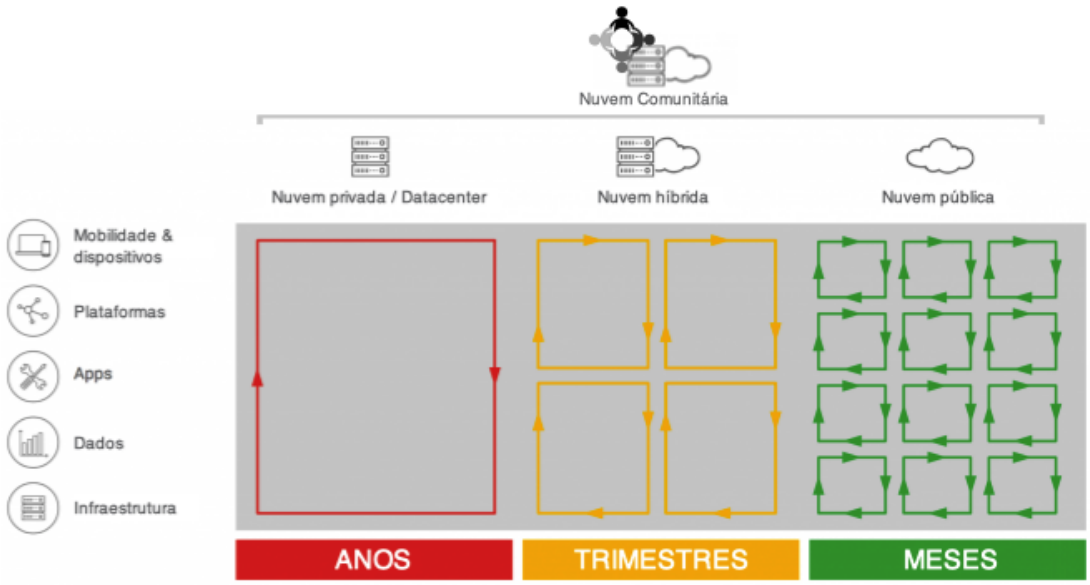

Figura 2.2: Principais modelos de implantação de nuvem, adaptado de [2].

2.4 Modelos de Serviço

Um modelo de serviço define a forma como os recursos computacionais são entregues ao cliente [1]. A definição de NIST [31] estabelece três modelos de serviço: *Infrastructure as a Service*, Infraestrutura como Serviço (IaaS); *Platform as a Service*, Plataforma como Serviço (PaaS); e *Software as a Service*, Software como Serviço (SaaS). Esses modelos serão descritos nas próximas seções.

2.4.1 Modelo IaaS

O modelo IaaS fornece componentes de infraestrutura. Esses componentes podem incluir máquinas virtuais, dispositivos de armazenamento, redes, *firewalls*, balanceadores de carga entre outros recursos tradicionalmente associados a recursos físicos. Com o IaaS, os clientes têm acesso direto ao software de nível mais baixo da pilha, ou seja, ao sistema operacional em máquinas virtuais ou ao painel de gerenciamento de um *firewall* ou

balanceador de carga [1]. É possível verificar na Figura 2.3 exemplos de serviços desse modelo tais como *Elastic Compute Cloud* (EC2) [24] e o *Google Compute Engine* (GCE) [40]. Ambos os serviço entregam máquinas virtuais com diversas possibilidades de configuração para memória, CPU, armazenamento, entre outros recursos de infraestrutura.

Em um serviço de IaaS o cliente pode rapidamente aumentar e diminuir os recursos da infraestrutura de acordo com a demanda, permitindo que ele pague apenas pelo que usar. Isso ajuda a evitar as despesas e a complexidade de comprar e gerenciar servidores físicos próprios, além de outros componentes da infraestrutura de um *datacenter*. Em geral, cada recurso é oferecido como um componente de serviço separado e o cliente só precisa alugar um em particular pelo tempo que precisar.

2.4.2 Modelo PaaS

Serviços de PaaS são um conjunto de serviços de nuvem que fornecem um ambiente para desenvolvimento, gerenciamento, implementação e integração de aplicativos [31]. Esse tipo de serviço é voltado para desenvolvedores de software e permite que novas soluções sejam desenvolvidas ou estendidas, sem que o desenvolvedor tenha a preocupação de configurar todos os *Software Development Kits*, coleção de softwares de desenvolvimento (SDKs), tampouco a infraestrutura [41].

Diversos provedores oferecem PaaS, usando, em geral, ferramentas baseadas na web para diminuir o tempo de desenvolvimento e reduzir custos para os desenvolvedores, tais como: controle de versão, planejamento ágil e de ciclo de vida, entre outros [41]. O *CloudFoundry* [42] e o *AppEngine* [9], também presentes na pirâmide da Figura 2.3, entregam plataformas via web para desenvolvimento e implantação de aplicativos, seguindo o modelo de PaaS.

2.4.3 Modelo SaaS

O modelo de software como serviço oferece aos usuários uma maneira fácil de acessar muitos de seus aplicativos e serviços comerciais, tais como pacotes de correio eletrônico, processamento de texto, planilhas eletrônicas, entre outros [1]. Esse modelo permite que os usuários acessem esses programas pela Internet, e suprime a necessidade de instalar e executar um software específico no computador. Ao invés de comprar o software a um preço relativamente alto, basta seguir o padrão "pague pelo que usar" e, com isso, reduzir o custo total. Esse modelo possibilita que empresas economizem custos, pois não há taxas de licenciamento e o pagamento é apenas pelo que usa e quando usa [1].

Esse modelo também elimina a necessidade de atualizar os pacotes de software, já que o provedor se encarrega de atualizá-los para que o usuário final esteja sempre com o

melhor serviço. Um dos maiores benefícios do SaaS é a possibilidade do usuário acessar seu conteúdo de interesse a partir de qualquer lugar conectado à Internet [1]. Os serviços *Office* 365 [8] e *GSuite* [43] são exemplos de software disponibilizado como serviço em nuvem. Por meio dessas aplicações é possível editar textos, planilhas e apresentações, além de manusear outras ferramentas de produtividade. Tudo isso apenas utilizando um navegador web, sem qualquer instalação adicional.

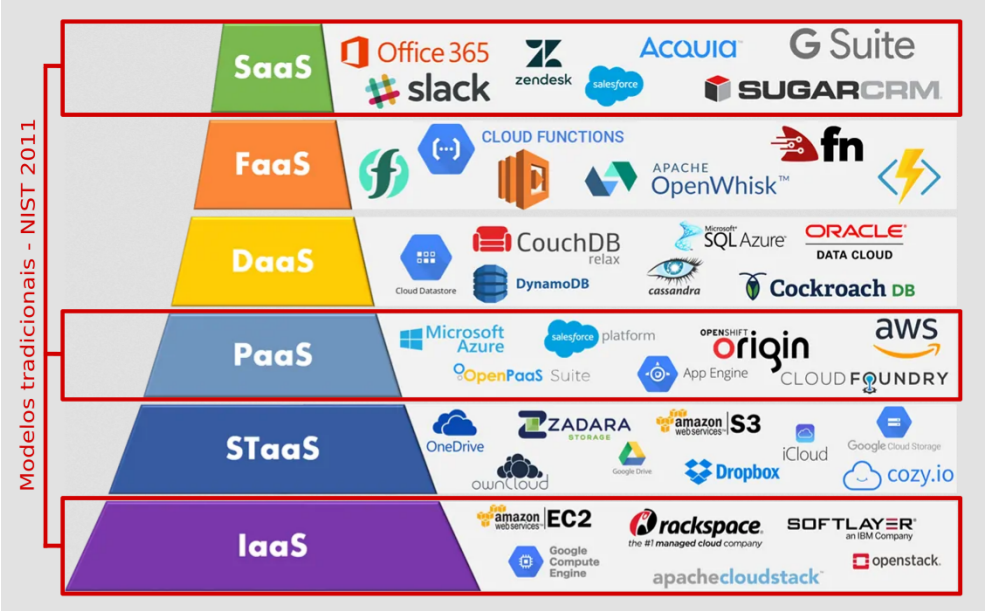

Figura 2.3: Modelos de serviços de nuvem [3].

A Figura 2.3 apresenta uma pirâmide com uma pilha de modelos de serviços de computação em nuvem. Na sua base encontra-se o modelo de IaaS, seu baixo nível representa sua maior proximidade com as máquinas físicas. No topo da pirâmide, o modelo de SaaS figura como o mais alto nível de abstração da pirâmide. Nesse modelo o usuário desconhece qualquer informação relacionada à estrutura interna do serviço. Entre o IaaS e o SaaS é possível encontrar diversos outros modelos, dentre eles o PaaS, cuja definição ocorreu ainda em 2011 [31], juntamente com a definição de IaaS e SaaS.

Devido ao aumento da quantidade de provedores de nuvem, assim como uma profusão de novos tipos de serviço, o arquétipo de nuvem vem recebendo diversos acréscimos de modelos. Seguindo o molde dos modelos de entrega de serviço definidas pelo NIST [31], tem surgido uma infinidade de novas siglas, cujo sufixo contém "aaS" (algo como serviço) [44]. Na Figura 2.3 é possível verificar: *Storage as a Service*, armazenamento como serviço (STaaS) [45], que representa modelos de serviço de armazenamento de arquivos; *Data/-Desktop/Database as a Service*, dados, área de trabalho ou banco de dados como serviço (DaaS) [46]; e *Function as a Service*, função como um serviço (FaaS) [23], apresentando

os serviços de entrega de processamento de funções como serviço, que será melhor descrito no Capítulo 4.

2.4.4 Modelos XaaS

Os provedores de nuvem têm amadurecido constantemente seus serviços, inclusive promovendo integração entre eles. À medida que a oferta de serviços cresce, as fronteiras previstas pelo NIST [31], em 2011, vão sendo redefinidas. Muitos provedores têm renomeado seus serviços, em especial os SaaS, visto que o termo "software" é algo muito amplo. Assim, o termo "XaaS" (*Everthing as a Service*) tem sido usado para definir essa classe de modelos de serviço em nuvem [44].

Dessa maneira, é possível encontrar diversos modelos de serviço, do tipo XaaS, sendo oferecidos pelos provedores. Como não existe um padrão estabelecido para a nomenclatura, alguns provedores oferecem serviços com entregas diferentes que compartilham a mesma sigla, como DaaS, que pode representar tanto um serviço de entrega de dados, como um serviço de banco de dados, e até mesmo, de desktop. A seguir são elencados alguns exemplos de serviços oferecidos atualmente, e que usam a mesma sigla para serviços diferentes e/ou siglas diferentes para os mesmos serviços:

- **DaaS:** *Database as a service*, banco de dados como serviço [46];
- **DaaS:** *Desktop as a service*, área de trabalho como serviço [47];
- **DaaS:** *Data as a service*, dados (informações) como serviço [48];
- **BaaS:** *Backup as a Service*, cópia de segurança como serviço [49];
- **DRaaS:** *Disaster Recover as a Service*, recuperação de desastres como serviço [50];
- **STaaS:** *Storage as a Service*, armazenamento como serviço [45];
- **LaaS:** *Log as a Service*, registro ostensivo como serviço [51];
- **MaaS:** *Monitoring as a Service*, monitoramento como serviço [52];
- **AaaS:** *Analytics as a Service*, painéis analíticos como serviço [52];
- **MLaaS:** *Machine Learning as a Service*, aprendizado de máquina como serviço [53];
- **CaaS:** *Container as a Service*, contêiner como serviço [54];
- **FaaS:** *Function as a Service*, função como um serviço [23].

Toda essa gama de serviços disponíveis exige que o usuário conheça detalhadamente o funcionamento daquilo que está sendo contratado. Eventualmente, o usuário pode

ser surpreendido por comportamentos inesperados, tais como perda de dados devido a desmobilizações realizadas pelo provedor de forma automática e até mesmo cobranças não previstas pelo cliente, como no caso de serviços que cobram pelo tráfego de rede gerado por conta da utilização de um determinado serviço.

2.5 Provedores Públicos

Os provedores públicos de serviços de nuvem são fornecedores que alugam seus recursos e serviços de computação em nuvem. Esses ativos são utilizados dinamicamente com base na demanda do cliente de acordo com um determinado modelo de negócios [40]. Essa relação, geralmente, é regida por um *Service Level Agreement*, acordo de nível de serviço (SLA) [2], no qual o provedor se compromete a entregar determinados requisitos como, por exemplo, um percentual de garantia de disponibilidade. O não cumprimento desse acordo pode ensejar desde bônus ao cliente, até multas ao provedor, a depender dos termos do contrato.

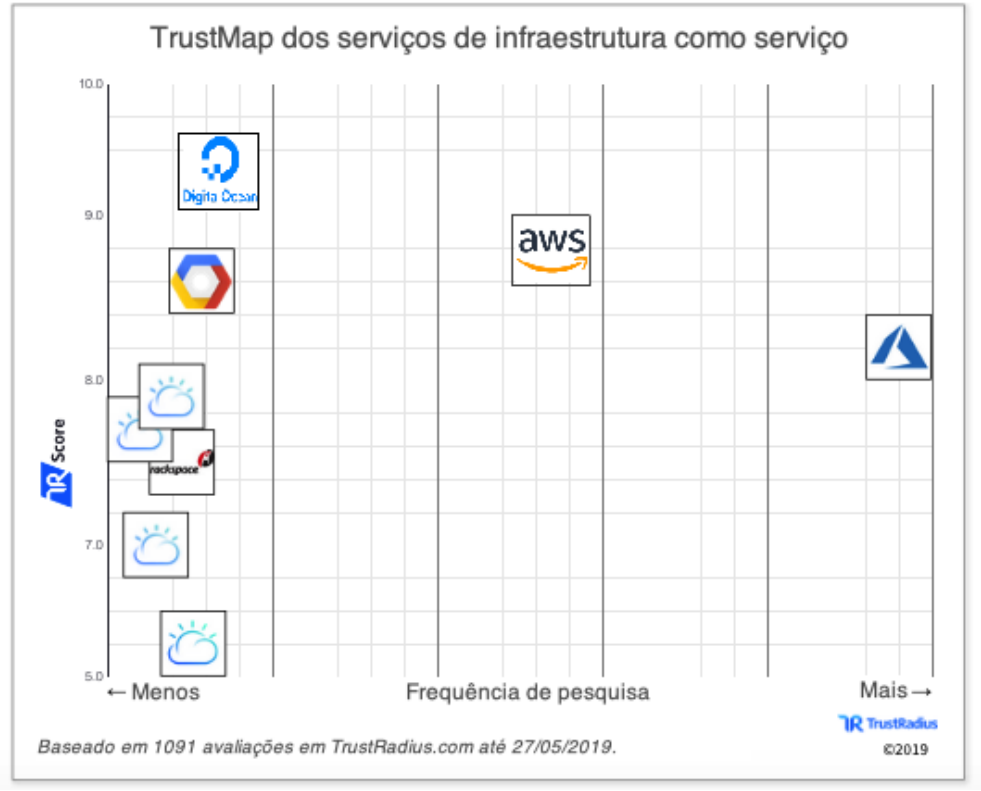

Figura 2.4: Mapa de soluções de IaaS da Trust Radius, adaptado de [4].

Quanto mais a computação em nuvem se expande, cresce junto a quantidade de empresas fornecendo esse tipo de serviço. Para lidar com as opções desse mercado, diversas empresas de consultoria publicam relatórios comparativos dos provedores.

A Trust Radius [55] recebe avaliações de serviços de diversos segmentos. A partir dos dados das avaliações é possível obter um mapa de acordo com a categoria do serviço. A Figura 2.4 mostra o mapa de soluções de IaaS no momento do acesso ao portal da consultoria. Nessa Figura 2.4 é possível perceber que o Microsoft Azure [8] e o *Amazon Web Services* (AWS) [56] lideram o segmento de IaaS, seguidos por Digital Ocean [57] e pelo *Google Cloud Platform* (GCP) [9].

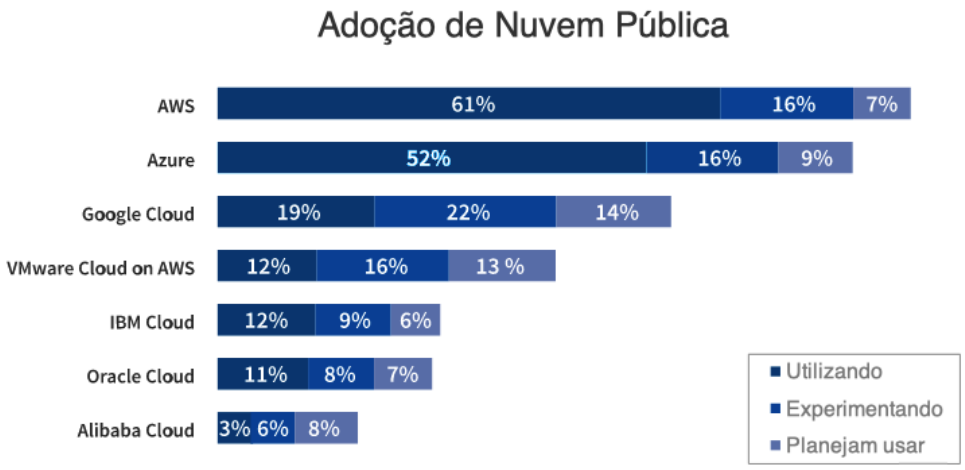

Figura 2.5: Adoção de nuvem pública em 2019 a partir do Right Scale, adaptado de [5].

Além disso, a consultoria Right Scale [5] elabora periodicamente relatórios estatísticos demonstrando o comportamento do mercado de nuvem. A Figura 2.5 mostra a adoção de provedores de nuvem em 2019. Esse estudo aponta que os três principais provedores de nuvem pública em 2019 eram: AWS, Azure e GCP.

Outra importante empresa de consultoria é o Gartner [58], que é uma empresa que presta assistência a negócios de vários segmentos e, periodicamente, publica dados posicionando os serviço de nuvem em quatro áreas: líderes, desafiadores, visionários e de nicho. Conforme pode ser visto na publicação de julho de 2019, mostrado na Figura 2.6, AWS, GCP e Azure fazem parte do quadrante de líderes em serviços de entrega de infraestrutura.

Considerando a posição de importância exercida por AWS, Azure e GCP no cenário da computação em nuvem, esses provedores serão melhor explorados nas seções a seguir deste trabalho.

2.5.1 Amazon Web Services - AWS

Em 2006, o *Amazon Web Services* (AWS), serviço de nuvem pública da Amazon, começou a oferecer infraestrutura de Tecnologia da Informação (TI) como serviços por meio da Internet, agora conhecido como computação em nuvem. Hoje, o AWS fornece uma

Figura 2.6: Quadrante mágico do Gartner para IaaS em 2019, adaptado de [6].

plataforma de infraestrutura altamente confiável, escalável e de baixo custo na nuvem que alimenta centenas de milhares de empresas em 190 países em todo o Mundo [56].

A infraestrutura de nuvem do AWS é baseada em regiões e zonas de disponibilidade. Uma região é um local físico no Mundo onde existem várias zonas de disponibilidade [7]. As zonas de disponibilidade consistem em um ou mais *datacenters* discretos, cada um com energia, rede e conectividade redundantes, instalados em locais separados.

As zonas de disponibilidade oferecem a capacidade de operar aplicativos de produção e bancos de dados em alta disponibilidade, tolerantes a falhas e melhor dimensionáveis do que seria possível em um único *datacenter*. O provedor AWS opera 64 zonas de disponibilidade em 21 regiões geográficas ao redor do Mundo, conforme apresentado na Figura 2.7.

O AWS é composto por muitos serviços de nuvem que podem ser usados em combinações personalizadas para as necessidades corporativas ou organizacionais. Para acessar esses serviços é possível usar o AWS *Management Console*, a *Command Line Interface*, interface por linha de comando (CLI) ou os *Software Development Kits*, coleção de softwares de desenvolvimento (SDKs) [56].

Assim, uma gama considerável de serviços é oferecida pelo portal do provedor AWS. Os principais são [56]:

Figura 2.7: Infraestrutura global do AWS [7].

- **Amazon EC2:** entrega máquinas virtuais e constitui seu serviço de IaaS mais importante, servindo como base para muitos outros serviços;
- **Amazon *Simple Storage Service*, serviço simples de armazenamento (S3):** serviço de armazenamento escalável;
- **Amazon Aurora:** banco de dados relacional de alta performance;
- **Amazon DynamoDB:** banco de dados não relacional (noSQL);
- **Amazon *Relational Database Service*, serviço de banco de dados relacional (RDS):** MySQL, PostgreSQL, Oracle, SQL Server e MariaDB;
- **AWS Lambda:** serviço para processamento de funções diretamente na nuvem, sem preocupação com servidores (FaaS);
- **Amazon *Virtual Private Cloud*, nuvem privada virtual (VPC):** permite provisionar uma seção da nuvem AWS isolada logicamente, na qual é possível executar recursos do AWS em uma rede virtual que o próprio usuário define;
- **Amazon Lightsail:** serviço de PaaS no qual são entregues servidores virtuais, armazenamento, bancos de dados e redes para habilitar o usuário ao desenvolvimento de aplicações; e
- **Amazon SageMaker:** fornece uma interface para criar, treinar e implantar rapidamente modelos de aprendizado de máquina. Ele permite rotular e preparar os dados, escolher um algoritmo, treinar, ajustar e otimizar o modelo para implantação, fazer previsões e tomar ação.

O provedor oferece ainda gratuidade para um certo nível de uso de alguns de seus serviços. Isso é relevante porque permite ao usuário fazer experimentos a fim de validar se aquele serviço atende à sua necessidade ou não.

2.5.2 Microsoft Azure

Em outubro de 2008 a Microsoft anunciou o lançamento da sua plataforma de oferta de serviços em nuvem, o Azure [59]. Porém, somente em fevereiro de 2010 o provedor efetivamente iniciou sua operação [60]. O Azure entrega serviços de nuvem dos três modelos tradicionais: SaaS, PaaS e IaaS. Ele é, atualmente, o principal rival do AWS em vários segmentos. Essa rivalidade é bastante evidente no próprio portal do Azure, no qual é possível encontrar diversos comparativos com ferramentas da concorrente, obviamente evidenciando a supremacia do provedor da Microsoft.

Figura 2.8: Infraestrutura global do Azure, adaptado de [8].

A Figura 2.8 mostra a distribuição dos *datacenters* do Azure. Atualmente, estão disponíveis 44 regiões, com previsão de implantação de 10 novas unidades. O portfólio de produtos e serviços do Azure é bastante diversificado, destacam-se [8]:

- **Máquinas virtuais:** provisionamento de máquinas virtuais Windows e Linux em segundos;
- **Contas de armazenamento:** armazenamento em nuvem durável, de alta disponibilidade e altamente escalável;
- **Banco de dados *Structured Query Language*, linguagem de consulta estruturada (SQL) do Azure:** banco de dados relacional como serviço;

- **Serviço de Aplicativo:** desenvolvimento de aplicativos de nuvem poderosos para a web, e para dispositivos móveis;
- **Azure Cosmos:** multi modelo de banco de dados, distribuído globalmente para qualquer escala;
- **PlayFab:** plataforma completa de *backend* do *LiveOps* para criar e operar jogos em tempo real;
- ***Azure Kubernetes Service*, serviço de Kubernetes da Azure (AKS):** gerenciamento das operações do *Kubernetes*[1];
- **Funções do Azure:** processamento de eventos com o código sem servidor; e
- **Serviços Cognitivos:** recursos de *Application Programming Interface*, interface de programação de aplicações (API) inteligente para habilitar interações contextuais.

O provedor oferece cerca de 25 serviços gratuitamente, além disso, para novos usuários há um bônus de R$ 750,00 para explorar diversos serviços durante 30 dias e, ainda, 12 meses de gratuidade em outra gama de serviços.

2.5.3 Google Cloud Platform

Em abril de 2008, apenas dois anos após o lançamento do AWS, o Google anunciava sua entrada no mercado de nuvem com o *Google Cloud Platform* (GCP) [62]. O serviço de nuvem do Google que chamou mais atenção, após seu lançamento, foi o Google Docs.

O serviço Google Docs entrega um pacote de utilitários para elaboração de textos, planilhas, apresentação de slides e formulários. O serviço segue o conceito da web 2.0, ou seja, sem a necessidade de instalação de qualquer software adicional, tudo pelo navegador [43]. Inicialmente, o serviço oferecia apenas o básico para edição do conteúdo, mas em 10 anos de evolução muitas funcionalidade foram adicionadas.

A infraestrutura do GCP, atualmente, conta com 61 zonas de disponibilidade em 20 regiões, além de 134 locais de extremidade de rede, conforme apresentado na Figura 2.9. O provedor prevê sua expansão com implantação de regiões em Seul (Coreia do Sul), Salt Lake City (EUA) e Jacarta (Indonésia).

Atualmente, o GCP oferece mais de 100 produtos distribuídos nas categorias: computação, armazenamento, rede, *big data*, transferência de dados, inteligência artificial, identidade e segurança, operações, e ferramentas para desenvolvedores. Os serviços que mais se destacam são [9]:

[1]Kubernetes é uma plataforma de código aberto que automatiza as operações dos contêineres Linux. Essa plataforma elimina grande parte dos processos manuais necessários para implantar e escalar as aplicações em contêineres [61].

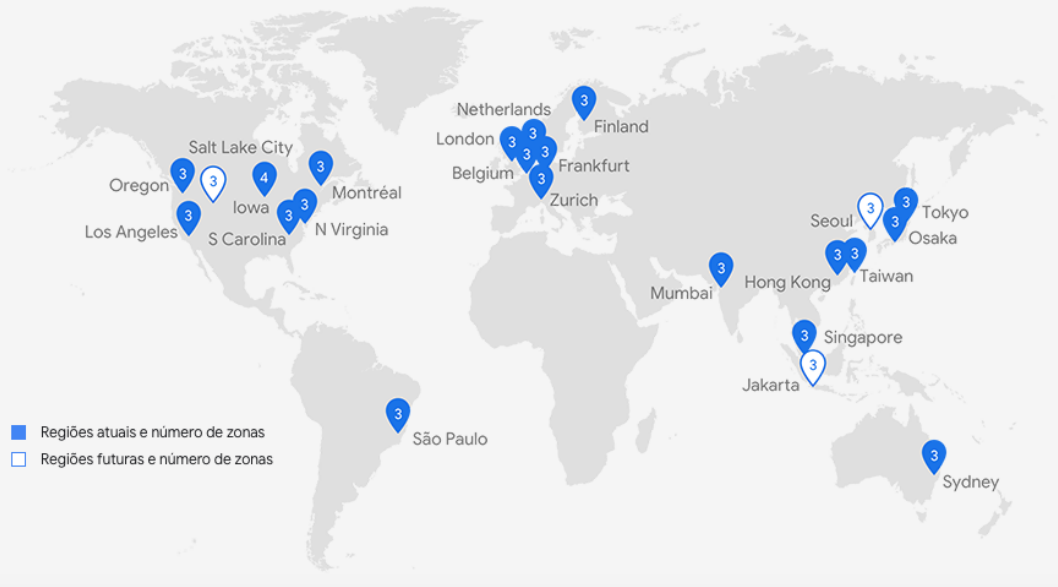

Figura 2.9: Infraestrutura global do GCP [9].

- **Compute Engine:** oferece máquinas virtuais que são executadas nas inovadoras centrais de dados do Google, e estão conectadas por meio de uma rede de fibra de nível mundial. As suas ferramentas e os seu fluxo de trabalho permitem escalar desde casos individuais até um ambiente de nuvem global com balanceamento de carga;
- **App Engine:** é uma plataforma para criar aplicativos web e *backends* móveis escaláveis. Esse serviço conta com APIs e serviços integrados que costumam ser utilizados na maioria dos aplicativos, como bases de dados noSQL, *Memcache* e uma API de autenticação de usuários;
- **Cloud Storage:** sistema de armazenamento de objetos unificado, para desenvolvedores e empresas, que abrange desde a provisão de dados ativos até a aprendizagem automática e a análise de dados, passando pelo arquivamento;
- **Cloud SQL:** facilita a configuração, a manutenção e a administração das bases de dados relacionais MySQL na nuvem;
- **Cloud *Domain Name Service* (DNS):** serviço escalável, confiável, administrado e certificado por DNS, com a mesma infraestrutura do Google. A sua baixa latência e a alta disponibilidade o tornam uma opção econômica na hora de oferecer aplicativos e serviços aos seus usuários. Com Cloud DNS é possível transformar solicitações de nomes de domínios como www.google.com em endereços *Internet Protocol*, protocolo da Internet (IP) como 74.125.29.101. Além disso, Cloud DNS é

programável, ou seja, pode publicar e administrar milhões de zonas e registros DNS por meio da sua interface de usuários simples, sua CLI ou sua API;

- **Cloud *Content Deliver Network* (CDN):** tira o máximo proveito dos pontos perimetrais que o Google distribui em todo o Mundo, a fim de acelerar a distribuição de conteúdo para os *sites* e os aplicativos fornecidos por meio de *Google Compute Engine* e *Google Cloud Storage*. O Cloud CDN reduz a latência da rede, diminui a carga das origens e reduz os custos de serviço;
- **Cloud Load Balancing:** distribui os recursos com balanceamento de carga em uma ou várias regiões, para que estejam mais perto dos usuários e cumpram com seus requisitos de alta disponibilidade. *Cloud Load Balancing* permite colocar recursos em um só IP *anycast* e desfrutar de uma elasticidade automática inteligente. Ele também dispõe de uma ampla gama de opções de personalização. Além disso, está integrado com *Google Cloud CDN* para otimizar a distribuição de conteúdo e de aplicativos;
- **Big Query:** um armazém de dados empresariais do Google a baixo custo, totalmente administrado e apto para analisar petabytes de dados. O *BigQuery* não precisa de servidor. Assim, não é necessário administrar nenhuma infraestrutura, nem é necessário um administrador de base de dados;
- **API Vision Cloud:** compreende o conteúdo de uma imagem por meio do encapsulamento de potentes modelos de aprendizagem automática em uma API *Representational State Transfer*, transferência de estado representacional (REST) fácil de usar. A API classifica imagens rapidamente em milhares de categorias (por exemplo "barco a vela", "leão" ou "torre Eiffel"), detecta objetos e rostos individuais dentro das imagens, além de buscar e ler palavras impressas nelas;
- **API Speech:** transforma áudio em texto aplicando potentes modelos de redes neurais em uma API fácil de usar. Essa API reconhece mais de 80 idiomas; e
- **API Translation:** oferece uma interface de programação simples para traduzir uma cadeia arbitrária em qualquer idioma admitido.

Assim como seus principais concorrentes, o GCP também oferece uma cota gratuita para experimentação de seus serviços de nuvem. Essa cota é adicionada em moeda local à conta do usuário, no ato do cadastramento, e tem validade de 12 meses.

2.5.4 Outros Provedores

Além dos principais provedores apresentados nas Seções 2.5.1, 2.5.2 e 2.5.3, existem diversas outras organizações que se lançaram nesse mercado. Empresas bem estabelecidas no cenário de tecnologia como a *International Business Machines* (IBM) [63] e o Oracle [64] vem tentando aproveitar o movimento encabeçado pelos gigantes citados. Assim, é importante comentar, ainda que brevemente, sobre alguns desses provedores de menor representatividade, mas que não devem ser considerados irrelevantes, cito:

- **IBM Cloud**

  O IBM Cloud [63] é uma plataforma de nuvem completa que abrange nuvens públicas, privadas e híbridas. Ele permite a construção de um conjunto robusto de ferramentas avançadas de dados e inteligência artificial [63]. Os seus serviços incluem soluções para provisionamento de máquinas virtuais (IaaS), integração e entrega contínua (PaaS), inteligência artificial com *chatbots*, usando o *Watson* (SaaS), e outros serviços de diferentes segmentos [63]. Atualmente, ele conta com 60 *datacenters* em 6 regiões, e 18 zonas de disponibilidade.

- **Oracle Cloud**

  O Oracle Cloud [64] fornece IaaS, PaaS, SaaS e DaaS (Dados como Serviço). Esses serviços são usados para criar, implantar, integrar e estender aplicativos na nuvem. Esse provedor suporta diversas ferramentas, tais como *Kubernetes*, *Hadoop*[2], *Kafka*[3], etc. Ele oferece ainda uma variedade de linguagens de programação, bancos de dados, ferramentas e *frameworks*, incluindo específicos da Oracle [64].

  O Oracle Cloud foi construído para cargas de trabalho de classe corporativa. A sua infraestrutura contém uma combinação de domínios de disponibilidade, *datacenters* autônomos independentes de falhas com energia e resfriamento independentes [64]. Ao todo são sete *datacenters* comerciais e dois governamentais.

- **Alibaba Cloud**

  O Alibaba Cloud [67] é um provedor chinês fundado em 2009 e conhecido também como *Aliyun* [68]. Ele oferece serviços de SaaS, PaaS e IaaS, e possui 56 zonas de disponibilidade (*datacenters*) instaladas em 19 regiões, tanto da Ásia quanto em alguns locais da Europa e até nos Estados Unidos. O Alibaba Cloud ostenta um

[2]Apache Hadoop é um *framework* que permite o processamento distribuído de grandes conjuntos de dados em *clusters* de computadores usando modelos de programação simples [65].

[3]Apache Kafka é uma plataforma de *streaming* que permite a publicação e inscrição em fluxos de registros, semelhantes a uma fila de mensagens ou sistema de mensagens corporativo. Trabalha de maneira durável e tolerante a falhas [66].

vasto rol de certificações em vários aspectos, como segurança, qualidade de operação, entre outros [67].

- **Digital Ocean**

  O Digital Ocean [57] é um provedor americano de IaaS e oferece servidores virtuais denominados *"droplets"* em 12 diferentes regiões de *datacenters*. Ele disponibiliza seis distribuições Linux e dezenas de aplicativos pré-configurados. O Digital Ocean também oferece serviços para contêineres usando *Kubernetes*.

- **Heroku**

  O Heroku [69] é um provedor de plataforma como serviço (PaaS) que suporta várias linguagens de programação, tais como Java, NodeJS, Scala, Clojure, Python, Ruby, *Hypertext Preprocessor*, pré-processador de hipertexto (PHP) e Go. O Heroku é uma das primeiras plataformas de nuvem, e está em desenvolvimento desde junho de 2007 [69]. Ele dispõe também de ferramentas para entrega contínua.

- **RackSpace**

  A Rackspace [70] é uma provedora de serviços de nuvem cuja empresa foi estabelecida em 1998 e oferece serviços de tecnologia desde então. Conta com serviços de IaaS e possui *datacenters* em 14 regiões, além de serviço de *colocation*[4] em outros 11 pontos. A Rackspace oferece ainda gerenciamento de nuvens públicas, privadas e hibridas, e utiliza a sua experiência de mercado para oferecer um serviço de suporte diferenciado, o *"Fanatical Support"* [70], que é um serviço voltado a resolução dos problemas que os clientes, eventualmente, enfrentem na operação dos seus serviços em nuvem.

- **Cloud Bees**

  A CloudBees [72] é uma fornecedora de serviços de entrega contínua de software a partir do Jenkins[5]. Ela disponibiliza serviços de PaaS para criar, executar e gerenciar aplicativos da web. O CloudBees suporta todo o ciclo de vida do aplicativo, desde o desenvolvimento até a implantação [72].

- **Cloud Foundry**

[4] *Colocation* é uma modalidade de serviço em que o cliente hospeda fisicamente o seu hardware no *datacenter*. Assim, o cliente obtém toda a infraestrutura com tecnologia de ponta necessária para manter suas aplicações *on-line* 24x7x365 com segurança. Conta com espaço em *racks*, fornecimento de energia elétrica, conexão dedicada com a Internet, segurança física, segurança lógica, *nobreaks*, monitoramento de acesso e ambiente climatizado [71]

[5] Jenkins é a solução mais amplamente adotada para entrega contínua, graças à sua extensibilidade e a uma comunidade ativa e vibrante. Com ele é possível construir *pipelines* para execução de tarefas automatizadas [72].

O Cloud Foundry [42] é um provedor de entrega contínua e suporta o ciclo de vida completo de desenvolvimento de aplicativos. A sua arquitetura é baseada em contêiner e executa aplicativos em qualquer linguagem de programação, em vários fornecedores de serviços em nuvem. Esse ambiente, com várias nuvens, permite que os desenvolvedores usem a plataforma de nuvem adequada às cargas de trabalho de aplicativos específicos e mova essas cargas, conforme necessário, em questão de minutos, sem alterações no aplicativo.

Os aplicativos implantados no Cloud Foundry acessam recursos externos por meio de uma API do *Open Service Broker*[6]. Na plataforma todas as dependências externas como bancos de dados, sistemas de mensagens, sistemas de arquivos e assim por diante, são consideradas serviços. O Cloud Foundry permite que os administradores criem seus próprios mercados de serviços, nos quais os usuários podem consumir sob demanda.

- **Engine Yard**

  O Engine Yard [74] é um provedor de PaaS focado em implantação e gerenciamento de aplicações escritas em Ruby on Rails. Ele também suporta PHP e NodeJS [74]. O Engine Yard possui uma estrutura dedicada no AWS para sustentar os seus serviços. Ele oferece ainda serviços de suporte, e segundo a plataforma, contratá-los é 90% mais barato do que montar uma equipe de desenvolvimeto e operação DevOps[7] própria [74].

- **Scaleway**

  Scaleway [75] é um provedor francês de serviços de IaaS focado na oferta de máquinas virtuais. Ele possui quatro *datacenters* instalados na França e um na Holanda. Além da criação de máquinas virtuais em seu portal, o Scaleway oferece serviços de *colocation* e *datacenter* como serviço, no qual o cliente reserva uma parte física do provedor, como um *rack* inteiro ou metade, por exemplo [75].

- **Kamatera**

  A Kamatera [76] é um provedor americano de IaaS que possui 20 anos de experiência em tecnologia e oferece serviços de baixo custo. Ele conta com 13 *datacenters* globais, sendo parte deles em Israel. A Kamatera está constantemente atualizando

[6] O projeto *Open Service Broker API* permite que fornecedores de software independentes, provedores e desenvolvedores de SaaS forneçam facilmente serviços de apoio para cargas de trabalho em execução em plataformas nativas da nuvem, como *Cloud Foundry* e *Kubernetes* [73].

[7] DevOps é uma cultura de gestão de equipes de tecnologia que visa arrefecer a animosidade que habitualmente ocorre entre times de desenvolvimento e operação, extraindo melhores resultados por meio da adoção de automação, mensuração e compartilhamento de informações [13].

sua infraestrutura de hardware e comunicações para manter seus produtos de computação em nuvem na vanguarda da tecnologia de servidores e, fornecer os tempos de processamento e de reação mais rápidos [76].

- **OVH**

  O OVH [77] é um provedor de serviços de IaaS, PaaS e SaaS francês que fornece servidores dedicados, hospedagem compartilhada e na nuvem, registro de domínio e serviços de telefonia *Voice over Internet Protocol*, voz sobre IP (VoIP). O OVH possui 28 *datacenters* em 19 países, e oferece serviços de nuvem para criação de instâncias virtuais, *Kubernetes*, armazenamento, incluindo *Database as a Service*, *big data* e *analytics*, entre outros [77].

- **Hetzner Cloud**

  O Hetzner Cloud [78] é um provedor alemão de IaaS de baixo custo que possui dois *datacenters* na Alemanha e um na Finlândia. Ele faz parte da empresa *Hetzner Online* que presta serviços de nuvem, de *colocation* e de *hosting* [78]. A sua plataforma de nuvem oferece interface para criação de instâncias virtuais e alocação de espaço para armazenamento.

2.6 Considerações Finais

Diante do exposto, é possível concluir que a computação em nuvem se estabeleceu com bases sólidas no contexto tecnológico global. A importância desse modelo transcende o âmbito dos negócios e pode ser considerado um ativo de impacto social. Assim sendo, a viabilização de projetos, que outrora seriam impossíveis de avançar, tem provocado mudanças significativas Mundo afora. Esse fato enfatiza a necessidade desse modelo seguir avançando e passar a entregar poder computacional de forma cada vez mais pervasiva. Quanto maior for o foco na solução de problemas, ao invés de detalhes tecnológicos, melhores serão os resultados auferidos por esse paradigma.

Diante da proliferação de provedores que é possível encontrar no cenário atual, observa-se que a clientela desse modelo de computação tem alterado seu comportamento. A mesma concorrência que derruba os custos em qualquer mercado, na computação em nuvem também propicia uma mudança no próprio modelo. A medida que o rol de opções de serviços se expande, eleva o interesse das organizações em utilizar, simultaneamente, serviços diferentes, inclusive de múltiplos provedores. Esse comportamento tem colaborado para a construção de um novo conceito, a *sky computing*, que será descrito em detalhes no próximo capítulo.

Capítulo 3

Sky Computing

Neste capítulo é apresentado o conceito que figura como uma tendência de abordagem para o gerenciamento de ambientes complexos orientados a nuvem, o *sky computing*. O capítulo está dividido em cinco seções, sendo a primeira Seção (3.1) reservada para as considerações iniciais, enquanto a última Seção (3.5) é reservada para as considerações finais. Na Seção 3.2 são apresentadas as características do conceito de *sky computing*, na Seção 3.3 são descritos os principais orquestradores comerciais disponíveis atualmente e na Seção 3.4 é apresentada uma avaliação dos orquestradores de nuvem e um experimento entre duas ferramentas que apresentaram aderência a abordagem proposta neste trabalho.

3.1 Considerações Iniciais

Conforme a adoção da computação em nuvem cresce, aumenta o interesse por soluções que possam aprimorar diversos aspectos dessa abordagem. Assim, a alta disponibilidade, a resiliência, a baixa latência e a eliminação do *lock-in vendor* [79] são aspectos importantes que vem sendo considerados por organizações que demandam recursos computacionais, visando garantir a continuidade dos seus negócios [10].

O mercado de provedores de nuvem tem se tornado mais e mais competitivo ao longo do tempo. Atualmente, há uma profusão de oferta de serviços em diferentes provedores. Muitos deles oferecem produtos semelhantes, mas por meio de abordagens diferentes. Nesse cenário, tem crescido dentro das organizações uma abordagem *multicloud*, na qual recursos oferecidos por diversos provedores compõem o arcabouço de sustentação tecnológica desses organismos. Essa abordagem vem sendo denominada *sky computing*, e também pode ser encontrada sobre as denominações de *multicloud* [80], *cross-cloud* [81], *federated clouds* [82], ou *inter-clouds* [83]. Em linhas gerais, essa diferença de nomenclatura está relacionada a forma como as arquiteturas lidam com o escalonador de recursos, uma vez

que esse componente pode ser intrínseco do provedor, ou um serviço externo provido por um *middleware*, por exemplo.

3.2 Características de Sky Computing

Sky computing pode ser definida como um nível acima da computação em nuvem, uma vez que seus recursos são dinamicamente provisionados em diversos provedores. Ele consiste em uma camada de gerenciamento de ambientes de nuvens, oferecendo capacidade variável de armazenamento com suporte dinâmico para demandas em tempo real [10]. A Figura 3.1 mostra um exemplo de arquitetura de *sky computing*.

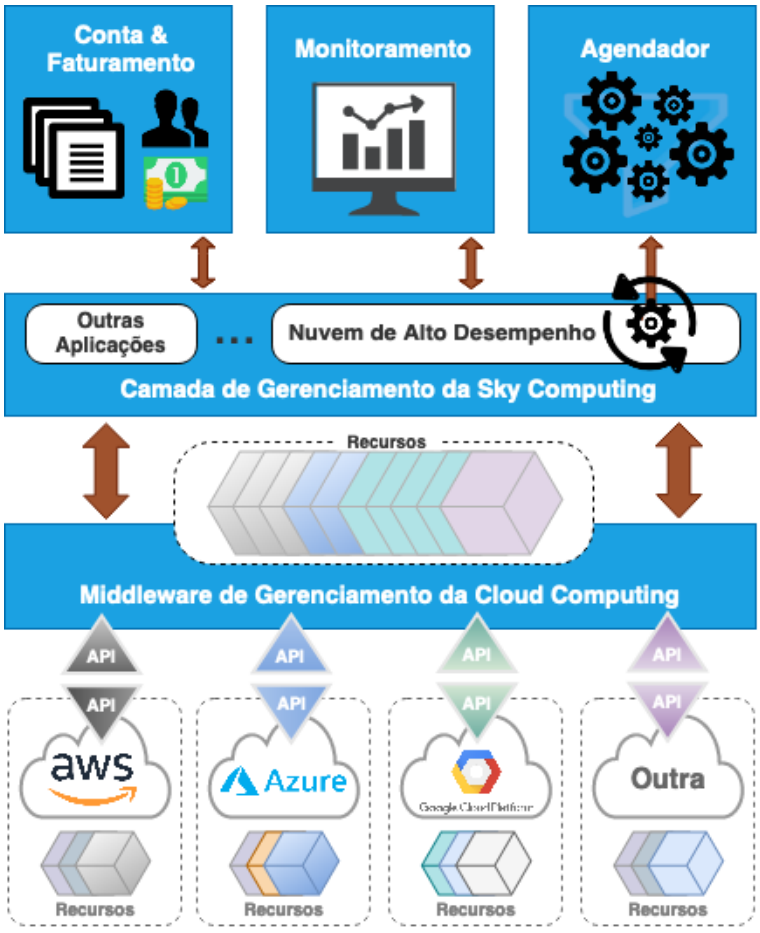

Figura 3.1: Exemplo de uma arquitetura de *sky computing*, adaptado de [10].

Nesse contexto, disponibilizar um serviço virtual sobre recursos distribuídos, combinando a capacidade de confiar em provedores remotos com um ambiente de rede de baixa confiabilidade [84], origina uma resposta altamente elástica a solicitações recebidas com um conjunto aparentemente infinito de recursos acessíveis.

Assim sendo, para gerenciar o enorme *pool* de recursos que esse conceito oferece, e considerando as diferentes formas de consumo, surgiu a necessidade de desenvolver um mecanismo capaz de gerenciar sistematicamente esses recursos computacionais, bem como de viabilizar a execução rápida e eficiente dos provisionamentos de ambientes em múltiplos serviços de diferentes provedores. Assim, em ambientes *multicloud* é preciso ter uma ferramenta que gerencie os papéis de cada tipo de recursos dentro do ambiente, bem como a origem (qual provedor) e suas respectivas integrações. A adoção dessas ferramentas visa abstrair a complexidade inerente à integração dos provedores de nuvem. Em geral, cada provedor oferece uma forma diferente de integração para consumo de seus recursos. Ainda que a abordagem predominante para interação com os provedores seja por meio de API REST, é comum a existência de tratamentos diferentes para conceitos semelhantes por parte dos provedores. Esse fenômeno acarreta dificuldades na migração de um provedor para outro, e até mesmo entre diferentes serviços do mesmo provedor.

As Figuras 3.2 e 3.3 mostram exemplos de requisições de máquinas virtuais nos provedores AWS e Azure. Assim, comparando as imagens é possível perceber significativas diferenças entre as formas de consumo de IaaS. Logo, ao propagar isso para o universo de recursos que um ambiente normalmente necessita, tais como armazenamento, balanceadores de carga, bancos de dados, entre outros, é possível notar o grau de complexidade que ambientes de *multicloud* podem atingir [10].

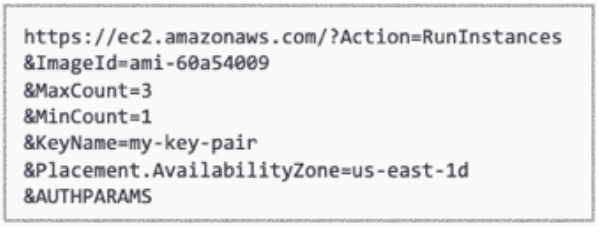

Figura 3.2: Exemplo de requisição de máquina virtual no AWS [11].

Não existe uma padronização acerca das formas de oferta de serviços em nuvem. Assim sendo, os provedores ficam livres para definir a forma de fazê-lo. Nesse contexto, considerando a necessidade de gerenciamento de ambientes compostos por recursos espalhados em diversos provedores, e que cada provedor possui uma forma particular de oferecer seus serviços, tem crescido um segmento específico de ferramentas de *Infrastructure as Code*, infraestrutura como código (IaC) [85], os orquestradores de nuvem. Normalmente as ferramentas de IaC são utilizadas em tarefas de provisionamento e configuração de infraestrutura. Por outro lado, atuando como orquestradores de nuvem, essas ferramentas podem gerenciar os diversos recursos de infraestrutura que uma aplicação necessite e que estejam presentes em um ou mais provedores, inclusive definindo aquele provedor

que melhor atende aos requisitos da aplicação naquele momento, em tempo real, podendo inclusive mover blocos da infraestrutura de um provedor para outro automaticamente.

Ferramentas de IaC tratam as definições de infraestrutura como software e, em geral, atuam como ferramentas de automação. Chef [86], Puppet [87] e Ansible [88] são exemplo de ferramentas de IaC para automação. Já os orquestradores de nuvem são responsáveis pelo aprovisionamento de recursos, escalas vertical e horizontal, integração com outras ferramentas, entre outras tarefas transversais aos diversos provedores acoplados o orquestrador.

```
{
  "location": "eastus",
  "name": "{vmName}",
  "properties": {
    "hardwareProfile": {
      "vmSize": "Standard_DS1_v2"
    },
    "storageProfile": {
      "imageReference": {
        "sku": "18.04-LTS",
        "publisher": "Canonical",
        "version": "latest",
        "offer": "UbuntuServer"
      },
      "osDisk": {
        "caching": "ReadWrite",
        "managedDisk": {
          "storageAccountType": "Premium_LRS"
        },
        "name": "myVMosdisk",
        "createOption": "FromImage"
      }
    },
    "osProfile": {
      "adminUsername": "{your-username}",
      "computerName": "{vmName}",
      "linuxConfiguration": {
        "ssh": {
          "publicKeys": [
            {
              "path": "/home/{your-username}/.ssh/authorized_keys",
              "keyData": "ssh-rsa AAAAB3NzaC1{snip}mf69/J1"
            }
          ]
        },
        "disablePasswordAuthentication": true
      }
    },
    "networkProfile": {
      "networkInterfaces": [
        {
          "id": "/subscriptions/{subscription-id}/resourceGroups/{resourceGrou
          "properties": {
            "primary": true
          }
        }
      ]
    }
  }
}
```

Figura 3.3: Exemplo de requisição de máquina virtual na Azure [12].

3.3 Orquestradores de Nuvens

Orquestradores de nuvens podem ser definidos como plataformas nas quais o desenvolvedor cria um descritor de infraestrutura que o orquestrador segue para implementar de forma autônoma, sem qualquer interação com o usuário [89]. Esse conceito garante que um descritor de infraestrutura, atualizado e testado, esteja disponível para os desenvolvedores iniciarem o processo de implantação de infraestrutura, fornecendo ao orquestrador simplesmente o código descritivo da infraestrutura [89].

Existem várias implementações acadêmicas para o conceito de "orquestração multi-cloud", tais como: Roboconf [90], Trans-Cloud [91], Live Cloud [92], SALSA [93], IM do GRyCAP [94], BioNimbuZ [95], The Celar Project[1] [96], Dicer[1] [97], OpenTOSCA[1] [98]; e muitos outros produtos com vocação comercial, tais como: Cloudify[1] [15], Heat[1] [17], Apache ARIA[1] [99], CloudFormation [18], Terraform[20], xOpera Orchestrator[1] [100], Alien4Cloud[1] [101] e Cloud Assembly [19].

Neste trabalho, apenas os orquestradores mais referenciados foram explorados devido a expectativa de que seu grau de maturidade permitisse a execução de testes simulando ambientes reais com múltiplos fornecedores. Assim, as próximas seções trarão uma visão geral do Cloudify, Heat, CloudFormation, Terraform e Cloud Assembly, a partir da definição do TOSCA, que é a base do Cloudify e do Heat.

3.3.1 TOSCA

A TOSCA [13] é um padrão emergente. O seu principal objetivo é aprimorar a portabilidade e o gerenciamento de aplicativos em nuvem. Tecnicamente, TOSCA é especificado usando uma definição de esquema *Extensible Markup Language*, linguagem de marcação extensível (XML). Os modelos de topologia são definidos como gráficos que consistem em nodos, e relacionamentos para especificar a estrutura topológica de uma aplicação [13].

Em TOSCA os modelos, os tipos de nodos e os tipos de relacionamento são definidos como mostrado na Figura 3.4. Eles são usados para criar modelos de nodos e modelos de relacionamentos correspondentes, baseados no modelo de topologia. Um sistema de tipos abrangentes pode ser introduzido porque os tipos podem ser derivados de outros tipos existentes no sentido de herança, como é usado, por exemplo, na programação orientada a objetos. Como exemplo, pode ser definido um tipo de nodo abstrato *"Java Servlet Container"*, que possui um nodo filho do tipo *"Apache Tomcat"*. Além desses, pode haver outros tipos de nodos derivados, como o *"Apache Tomcat 6.0"* e o *"Apache Tomcat 7.0"*.

Os tipos consistem em subelementos adicionais, assim as operações são anexadas a nodos e relacionamentos, por exemplo, para cobrir seu ciclo de vida (instalação, início,

[1]Projeto baseado na definição do padrão TOSCA [13].

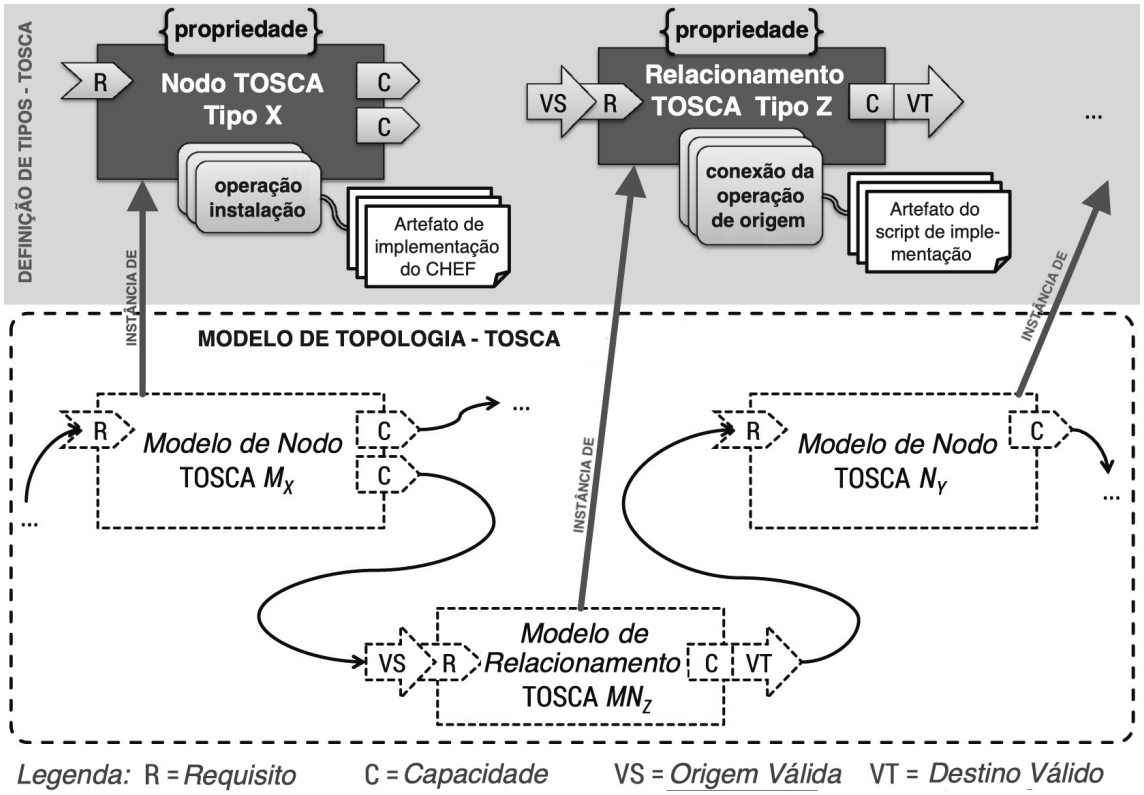

Figura 3.4: Modelos de definição de tipos em TOSCA, adaptado de [13].

parada, etc). Essas operações são implementadas por um *Implementation Artifact*, artefato de implementação (IA), que pode ser, por exemplo, uma "receita" do Chef [86] ou um *script* de *shell*. Um IA é executado quando a operação correspondente é invocada, por exemplo, pelo ambiente de tempo de execução do TOSCA [13].

As operações pertencentes a relacionamentos são diferentes das operações de origem e das operações de destino, pois um relacionamento vincula um nodo de origem a um nodo de destino. As operações de origem são executadas no nodo de origem, e as operações de destino no nodo de destino. Para permitir a vinculação de nodos e relacionamentos, eles expõem os requisitos e os recursos usados para fins de correspondência [13].

As propriedades podem ser definidas como estruturas de dados arbitrárias no esquema XML para tornar os nodos e os relacionamentos configuráveis. Todas as propriedades são expostas às operações e seus IAs, para que possam ser consideradas durante a execução [13].

Por fim, TOSCA especifica a estrutura do *Cloud Service Archives*, arquivos de serviço em nuvem (CSAR) como um formato de pacote autocontido e portátil para aplicativos em nuvem. Assim, apenas as definições XML de tipos e modelos fazem parte de um CSAR. Ele também contém todos os *scripts* e os arquivos que são referenciados, por exemplo, como IAs. Consequentemente, um CSAR é autocontido, permitindo que um ambiente

de tempo de execução TOSCA o processe, percorrendo o modelo de topologia para criar instâncias do aplicativo [13]. A Figura 3.5 apresenta um exemplo de arquivo de definição de arquitetura simples usando o TOSCA.

```
tosca_definitions_version: tosca_simple_yaml_1_0_0
description: Template for deploying a single server with predefined properties.
topology_template:
  node_templates:
    my_server:
      type: tosca.nodes.Compute
      capabilities:
        # Host container properties
        host:
         properties:
           num_cpus: 1
           disk_size: 10 GB
           mem_size: 4 MB
        # Guest Operating System properties
        os:
          properties:
            # host Operating System image properties
            architecture: x86_64
            type: linux
            distribution: rhel
            version: 6.5
```

Figura 3.5: Definição de arquitetura em TOSCA [14].

3.3.2 Cloudify

Cloudify [15] é uma estrutura de orquestração de nuvem de software livre que permite ao usuário modelar aplicativos e serviços, e automatizar todo o seu ciclo de vida. Ele inclui a implantação em qualquer ambiente de nuvem ou *datacenter*, monitorando todos os aspectos do aplicativo implantado, detectando problemas e falhas, e remediando-as manualmente ou automaticamente, além de tratar de tarefas de manutenção. No entanto, alguns recursos avançados, incluindo a interface do usuário da web, só estão disponíveis na edição *premium* [89]. As principais características dessa plataforma são:

- **Modelagem de aplicações:** a modelagem de aplicações permite descrever um aplicativo, com todos os seus recursos (infraestrutura, *middleware*, código de aplicativo, *scripts*, configuração de ferramentas, métricas e *logs*), de maneira genérica e descritiva [15]. A linguagem Cloudify *Domain Specific Language*, linguagem de domínio específico (DSL) é baseada no TOSCA;
- **Orquestração:** a orquestração permite manter e executar seu aplicativo. Além da instanciação, é possível executar operações contínuas, como dimensionamento, recuperação e manutenção [15];
- **Capacidade de extensão:** fornece abstração de componentes reutilizáveis para o sistema. É possível modelar qualquer coisa em uma linguagem descritiva, por

exemplo, IaaS, nuvens, ferramentas de gerenciamento de configuração, componentes *Software-Defined Networking*, rede definida por softwares (SDN), componentes *Network Functions Virtualization*, virtualização das funções da rede (NFV) e, assim por diante [15]. A Figura 3.6 mostra o console de criação da topologia de um ambiente;

- **Segurança:** significa proteger a comunicação com o *Cloudify Manager* e controlar quem tem permissão para usá-lo, e executar várias operações [15]. A comunicação segura é obtida usando o *Secure Sockets Layer*, camada de interface segura (SSL), que permite aos clientes validar a autenticidade do Cloudify Manager, e garantir que os dados enviados a partir e para ele sejam criptografados [15].

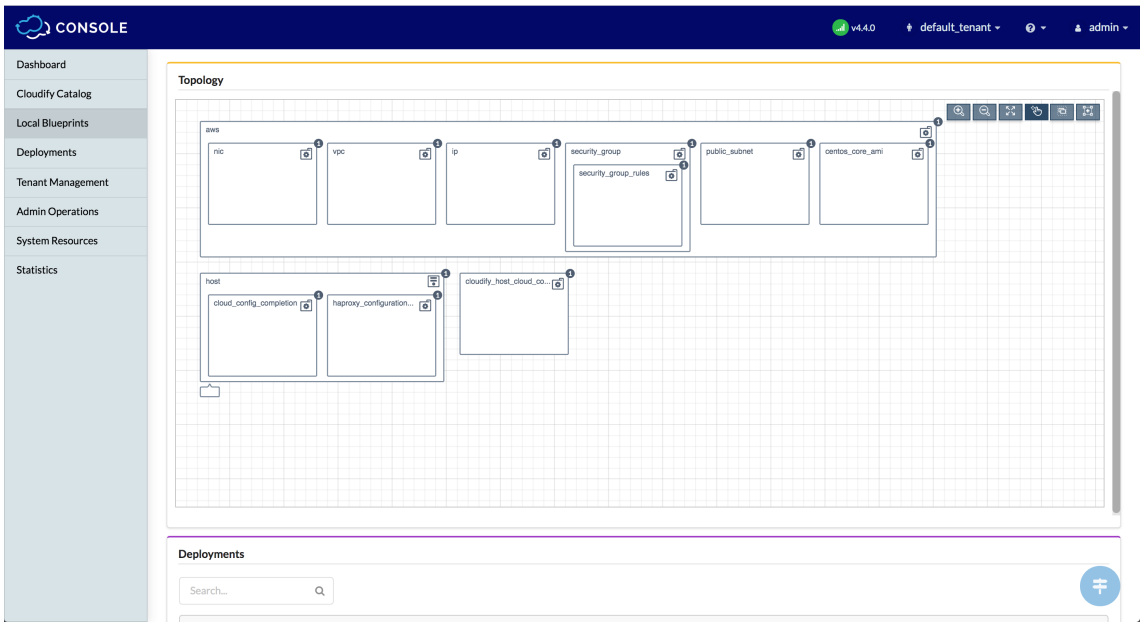

Figura 3.6: Console de criação de ambientes do CloudiFy [15].

3.3.3 Heat

O *Heat* [17] é um projeto incubado do *OpenStack*[2] que fornece orquestração. Ele permite descrever a infraestrutura de nuvem ou o aplicativo composto, chamado *stack* (pilha), em um arquivo de texto, escrito em uma linguagem própria chamada *Heat Orchestration Template*, modelo de orquestração do Heat (HOT) [16]. A Figura 3.7 ilustra o funcionamento de um processo de *deploy* por meio do *Heat*. É possível verificar que o *Heat*

[2]O *OpenStack* [39] é um sistema operacional em nuvem que controla grandes *pools* de recursos de computação, armazenamento e rede em todo o *datacenter*, todos gerenciados e provisionados por meio de APIs com mecanismos comuns de autenticação [39].

processa um arquivo de modelo escrito em HOT fornecido pelo administrador e, a fim de efetivar a alocação dos recursos, interage com os demais componentes do *OpenStack*, tais como Neutron [16] para recursos de redes; Nova para criação de máquinas virtuais, e Cinder para armazenamento em blocos [16].

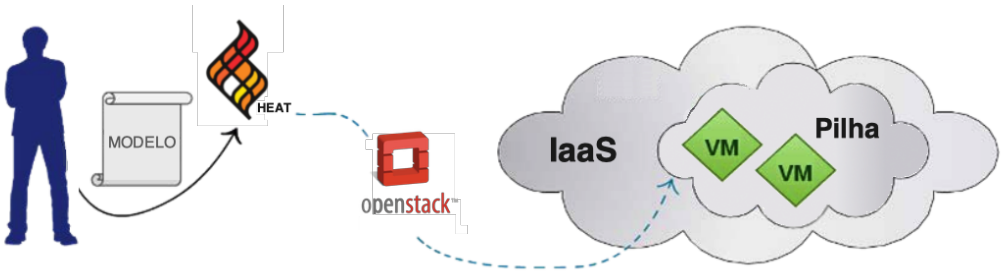

Figura 3.7: Fluxo de provisionamento com *Heat*, adaptado de [16].

O *OpenStack* [39] é um projeto para fornecer um modelo de serviço de computação em nuvem IaaS. O projeto é liberado sob os termos da licença Apache e é gerenciado pela *OpenStack Foundation*. Desde o seu lançamento em julho de 2010, pelo esforço da *National Aeronautics and Space Administration* (NASA) e da Rackspace, mais de 200 empresas aderiram ao projeto [16].

O *OpenStack* consiste em uma série de projetos inter-relacionados, escritos em Python que oferecem todos os principais serviços que um software IaaS deve oferecer. Os principais projetos pertencentes ao *OpenStack* são [16]: serviço de computação (Nova), serviço de armazenamento de objetos (Swift), serviço de armazenamento em bloco (Cinder), serviço de rede (Neutron), *dashboard* (Horizon), serviço de identidade (Keystone), serviço de imagens (Glance), e serviço de métricas (Ceilometer).

Infraestrutura Interna do *Heat*

O *Heat* é composto por três serviços principais na sua infraestrutura: *HeatEngine*, *Heat* API e *CloudWatch*. O *HeatEngine* é o serviço principal, que analisa modelos e fornece recursos, que são os elementos básicos de uma pilha. Depois de analisar um modelo, ele implantará e configurará os recursos na nuvem. O *Heat* expõe uma API REST nativa e uma API compatível com a Amazon AWS Query por meio do serviço *Heat* API. A Figura 3.8 apresenta a arquitetura e as integrações do serviços internos do *Heat* [16].

O *Heat* utiliza uma linguagem denominada acrshortHOT, e também oferece suporte ao formato compatível com o CloudFormation [17]. Os parâmetros de entrada definidos na seção de parâmetros de um modelo HOT permitem que os usuários personalizem um modelo durante a implementação. Por exemplo, isso permite fornecer nomes de pares de chaves personalizados ou identificadores de imagens para serem usados em uma implanta-

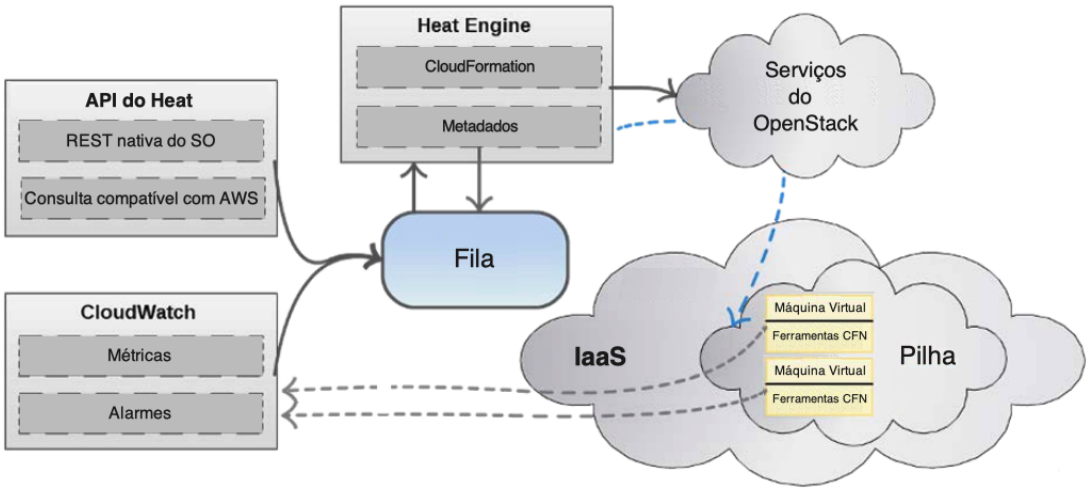

Figura 3.8: Arquitetura interna do *Heat*, adaptado de [16].

ção. Da perspectiva de um autor de modelo, a parametrização ajuda a tornar um modelo mais facilmente reutilizável, evitando suposições codificadas [17]. A Figura 3.9 apresenta um exemplo de infraestrutura definida em HOT para o *Heat* usando essa abordagem.

3.3.4 AWS CloudFormation

O AWS CloudFormation [18] oferece uma linguagem comum para descrever e provisionar todos os recursos de infraestrutura em um ambiente de nuvem. O CloudFormation permite usar um arquivo de texto simples para modelar e provisionar, de forma automática e segura, todos os recursos necessários para os aplicativos em todas as regiões e contas. Esse arquivo atua como única fonte confiável para o ambiente de nuvem [18].

O AWS CloudFormation oferece aos desenvolvedores e administradores de sistemas uma maneira fácil de criar e gerenciar uma coleção de recursos relacionados do AWS, provisionando e atualizando esses recursos de forma organizada e previsível [18] por meio de arquivos *JavaScript Object Notation* (JSON) ou *Yaml Ain't Markup Language* (acrônimo auto recursivo) (YAML). É possível usar os modelos de exemplo do AWS CloudFormation ou criar modelos próprios para descrever os recursos do AWS, e quaisquer dependências ou parâmetros de tempo de execução associados, necessários para executar um aplicativo [18]. Não é preciso conhecer a ordem de provisionamento dos serviços do AWS ou os detalhes para as dependências funcionarem. Assim, a plataforma do CloudFormation possui os seguintes recursos [18]:

- **Autoria com JSON/YAML:** permite modelar toda a infraestrutura em um arquivo de texto escrito em JSON ou YAML. Para projetar visualmente, o AWS

```
heat_template_version: 2015-04-30

description: Simple template to deploy a single compute instance

parameters:
  key_name:
    type: string
    label: Key Name
    description: Name of key-pair to be used for compute instance
  image_id:
    type: string
    label: Image ID
    description: Image to be used for compute instance
  instance_type:
    type: string
    label: Instance Type
    description: Type of instance (flavor) to be used

resources:
  my_instance:
    type: OS::Nova::Server
    properties:
      key_name: { get_param: key_name }
      image: { get_param: image_id }
      flavor: { get_param: instance_type }
```

Figura 3.9: Exemplo de arquivo HOT para o *Heat* [17].

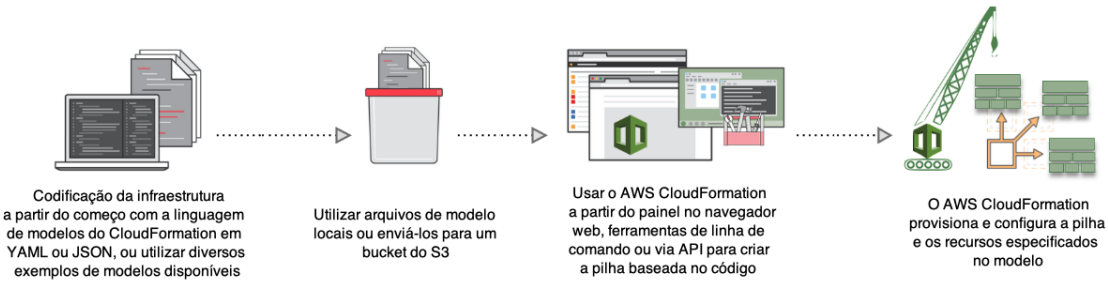

Figura 3.10: Como funciona o CloudFormation, adaptado de [18].

CloudFormation Designer permite começar a criar modelos por meio de uma interface web;

- **Controles de segurança:** a plataforma automatiza o provisionamento e a atualização da infraestrutura de forma segura e controlada. Não há etapas ou controles manuais, propensos a erros. É possível usar gatilhos de reversão para especificar o alarme do CloudWatch a ser monitorado pelo CloudFormation durante o processo de criação e atualização da pilha, assim, se qualquer um dos alarmes for acionado, o CloudFormation reverterá toda a operação da pilha para um estado previamente implantado;

- **Visualização das mudanças no ambiente:** é possível visualizar como as mudanças propostas em uma pilha podem afetar os recursos em execução. Assim, por exemplo, é possível verificar se as alterações excluirão ou substituirão algum recurso crítico. A efetivação somente ocorre após um processo de validação;
- **Gerenciamento de dependências:** gerencia automaticamente as dependências entre os recursos durante as ações de gerenciamento de pilha, suprimindo a necessidade de prévio estabelecimento de ordenação entre os processos;
- **Gerenciamento de várias contas e regiões:** permite provisionar um conjunto comum de recursos do AWS em várias contas e regiões com um único modelo do CloudFormation. As atividades de provisionamento, de atualização ou de exclusão de pilhas em várias contas e várias regiões são executadas de forma automática e segura. Esse recurso oferece o mesmo nível de automação, repetibilidade e confiabilidade a operações de gerenciamento de pilhas em várias regiões e contas; e
- **Extensibilidade:** é possível usar recursos personalizados, que são um mecanismo de extensão para criar provisionamento personalizado em uma função do AWS Lambda [56] e acionar esse provisionamento durante uma pilha do CloudFormation.

Todavia, embora o CloudFormation possa ser classificado como um orquestrador de nuvem, o fato de trabalhar apenas com um único provedor (o AWS) limita o potencial da sua utilização, uma vez que a impossibilidade de distribuir os recursos em múltiplos provedores fortalece o *lock-in vendor* e não permite que o utilizador desfrute de todos os benefícios do modelo de *sky computing*.

3.3.5 Cloud Assembly

Cloud Assembly [19] é um serviço de provisionamento de várias nuvens sustentado pela VMWare, que oferece a capacidade de criar uma nuvem privada. O usuário pode criar provisionamentos de provedor, como zonas de nuvem que fornecem serviços de computação, armazenamento, rede, balanceamento de carga e segurança para um conjunto específico de propósitos [19]. Essas zonas podem cobrir diferentes necessidades de conformidade e segurança, fornecer localização ou segregação de carga de trabalho, e serem usadas para *multi-tenancy*. Essas construções também mapeiam para construções equivalentes em nuvens públicas [19]. A Figura 3.11 apresenta o diagrama de provisionamento automático, no qual o Cloud Assembly se insere em diversas camadas transversais nas nuvens públicas, privadas e híbridas.

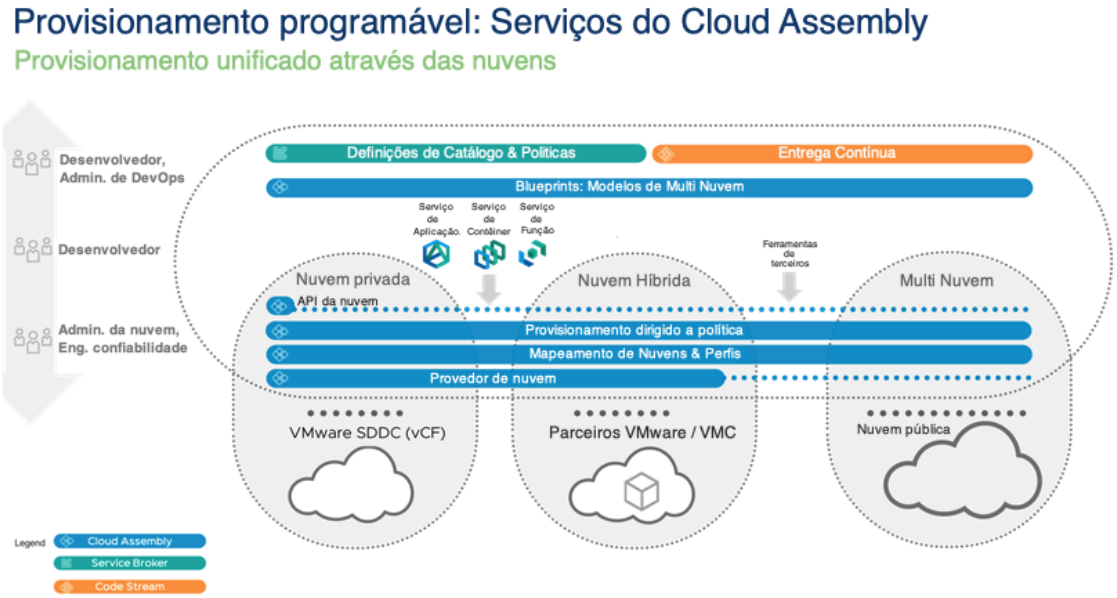

Figura 3.11: Serviços do Cloud Automation, adaptado de [19].

O Cloud Assembly também fornece uma camada de abstração em várias nuvens. Essa abstração normaliza as construções de infraestrutura, sem sacrificar a riqueza de cada uma das nuvens. As políticas de negócio usam essa camada de abstração para intermediar implementações de carga de trabalho entre várias nuvens [19]. Por exemplo, se as políticas ditarem locais diferentes para cargas de trabalho de desenvolvimento e teste, o Cloud Assembly as implantará nos locais corretos, assim se as políticas ditarem a camada de ouro para cargas de trabalho com uso intensivo de dados, o Cloud Assembly usará a camada de ouro para essas cargas de trabalho. Para ajudar na solução de problemas, o Cloud Assembly mapeia e mostra como avaliou as políticas e chegou a suas decisões [19].

3.3.6 Terraform

Terraform [20] é uma ferramenta de código aberto criada pela HashiCorp [20] que permite a definição de infraestrutura como código, usando uma linguagem de programação declarativa simples [20]. Ela possibilita a implantação e o gerenciamento dessa infraestrutura em vários provedores de nuvem pública (por exemplo AWS, Azure, GCP e Digital Ocean), e também em plataformas privadas de virtualização (por exemplo *OpenStack* e *VMWare*) usando alguns comandos [20]. Assim, por exemplo, na Figura 3.12 é possível ver todo o código necessário para configurar um servidor no AWS. Note que por meio dessa ferramenta não é necessário clicar manualmente em uma página web ou executar dezenas de comandos.

```
provider "aws" {
  region = "us-east-1"
}

resource "aws_instance" "example" {
  ami           = "ami-40d28157"
  instance_type = "t2.micro"
}
```

Figura 3.12: Configurando um servidor no AWS por meio do Terraform [20].

Atualmente, o Terraform suporta mais de 30 provedores diferentes. Além disso, ele implementa uma lógica gráfica complexa, que permite resolver dependências, inteligibilidade e confiabilidade. Quando se trata de servidores, o Terraform tem várias maneiras de configurá-los e conectá-los às ferramentas de gerenciamento de configuração existentes [102]. Não é independente de plataforma, mas permite o uso de vários provedores em um único modelo, e há maneiras de torná-lo uma plataforma um tanto quanto agnóstica [102]. A Figura 3.13 mostra o funcionamento do Terraform como orquestrador de nuvem, lendo um arquivo de definição e executando o provisionamento dos diversos recursos no provedor.

Dessa forma, o Terraform acompanha o estado atual da infraestrutura que cria e aplica alterações delta quando algo precisa ser atualizado, adicionado ou excluído [102]. Ele também fornece uma maneira de importar recursos existentes e segmentar apenas recursos específicos. É facilmente extensível com *plugins*, que devem ser escritos na linguagem de programação Go [102].

Figura 3.13: Fluxo de execução do Terraform, adaptado de [20].

O Terraform possui algumas características importantes, tais como:

- **Orquestração**: enquanto outras ferramentas de gestão de configuração se concentram nas tarefas de instalação e configuração dos itens de um ambiente, o Terraform vai além e também suporta a definição do provisionamento dos recursos, tais como

máquinas virtuais, componentes de rede, firewall, entre outros tipos de recursos disponibilizados pelos provedores. Isso viabiliza que a definição da infraestrutura parta de um ponto anterior, suprimindo a necessidade de um ambiente previamente provisionado para iniciar o uso efetivo da ferramenta;

- **Multicoud**: o Terraform oferece suporte a orquestração de várias nuvens e esse é o recurso mais versátil da ferramenta. Isso é especialmente útil quando são utilizados recursos em diferentes provedores de nuvem ao mesmo tempo;
- **Portabilidade**: é possível criar fluxos que transfiram ambientes de uma nuvem para outra utilizando o Terraform;
- **Infraestrutura imutável**: em geral, ferramentas de gestão de infraestrutura baseadas no modelos de IaC executam atualizações de software em servidores e isso pode levar a uma mudança na configuração da infraestrutura. Essas diferenças na configuração podem levar a erros que podem se tornar violações de segurança. O Terraform soluciona esse problema utilizando uma abordagem de infraestrutura imutável, na qual cada alteração na configuração leva a um instanciamento de configuração em separado, o que significa a implantação de um novo recurso no provedor e a exclusão do antigo. Dessa forma, atualizar um ambiente tende a ser mais tranquilo e livre de falhas, e até retornar à configuração antiga é tão fácil quanto reverter para uma versão específica;
- **Sintaxe simplificada**: o Terraform usa uma linguagem própria denominada HCL. Essa linguagem foi projetada para manter os artefatos legíveis por humanos e mais amigáveis;
- **Simulação da execução**: o Terraform suporta algo chamado "*Plan*". Essa forma de execução examina os artefatos do Terraform e determina as alterações que precisarão ser aplicadas na infraestrutura antes de efetivamente aplicá-las;
- **Arquitetura somente cliente**: o Terraform aproveita a API do provedor de nuvem para provisionar a infraestrutura, o que elimina a necessidade de verificações adicionais e também executa o sistema de gerenciamento de configuração em um servidor separado. Algumas ferramentas optam por fazer isso conectando-se através do *Secure Socket Shell* (SSH), mas com limitações. O Terraform trabalha com APIs e abre inúmeras opções, o que o torna mais seguro, confiável e fácil de usar.

3.4 Avaliação de Orquestradores Multicloud

Considerando as opções de orquestradores multicloud apresentadas, fez-se necessário avaliar a aplicação efetiva desses orquestradores em ambientes reais. Para isso, foi definida uma arquitetura para uma solução muito presente no cenário atual, o Wordpress[3] [103]. Essa arquitetura foi utilizada nos testes de desempenho dos orquestradores.

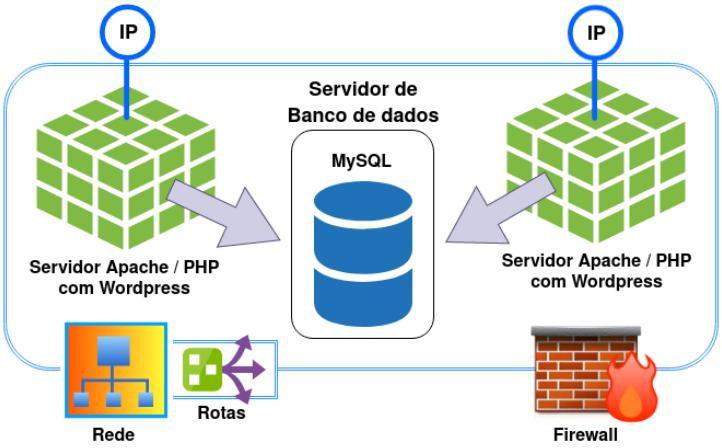

Figura 3.14: Arquitetura de referência do Wordpress para o experimento.

A Figura 3.14 mostra uma arquitetura para executar o Wordpress usando três máquinas virtuais, uma como servidor de banco de dados MySQL e as outras duas como servidores de aplicação Apache/PHP e Wordpress instalados. Para o funcionamento adequado desse ambiente foi necessário configurar os recursos de interconexão virtual (rede, sub-rede, IPs, interfaces e rotas) e segurança (regras de firewall).

Tabela 3.1: Quadro comparativo dos orquestradores de nuvem.

Orquestrador	Integração	Versão OpenSource	Artefato	CLI	API	Assistente gráfico
Cloud Formation	AWS	Não	YAML JSON	Não	Sim	Sim
Cloud Assembly	AWS, GCP, Azure e vSphere	Não	YAML	Sim	Sim	Sim
Heat	AWS e Openstack	Sim	HOT YAML	Sim	Sim	Não
Terraform	AWS, GCP, Azure, OpenStack, Oracle, vSphere e mais de 30	Sim	HCL JSON	Sim	Funções online	Não
Cloudify	AWS, GCP, Azure, OpenStack, vCloud, vSphere	Sim	TOSCA YAML	Sim	Sim	Sim

[3]O WordPress é baseado em PHP e MySQL e licenciado sob a GPLv2. É também a plataforma de escolha para mais de 34% de todos os *sites* da Internet [103].

Com base nos requisitos de arquitetura definidos na Figura 3.14 e nas características de *Sky Computing*, os orquestradores revisados foram analisados para definir aqueles que poderiam ser usados para provisionar ambientes automaticamente, conforme a arquitetura da Figura 3.14, nos três principais provedores de nuvem avaliados pelo Gartner [104], cito Amazon AWS, Google Cloud Platform e Microsoft Azure.

3.4.1 Experimento para Avaliação dos Orquestradores

Em uma análise preliminar dos orquestradores, conforme mostrado na Tabela 3.1, percebe-se que apenas Cloudify e Terraform apresentam as possibilidades de uso efetivo nos moldes definidos, uma vez que somente eles possuem integração com AWS, GCP e Azure, e atuam com uma versão *Open Source*. Portanto, o experimento de avaliação foi realizado usando apenas essas duas ferramentas. A Figura 3.15 ilustra a arquitetura do experimento e, no cenário apresentado, o *blueprint* definido para o Wordpress é provisionado em cada um dos provedores usando seus recursos de rede, máquinas virtuais, regras de segurança e roteamento. Contudo, é importante comentar que a abstração das diferenças entre recursos equivalentes de um provedor para outro é um recurso esperado dos orquestradores.

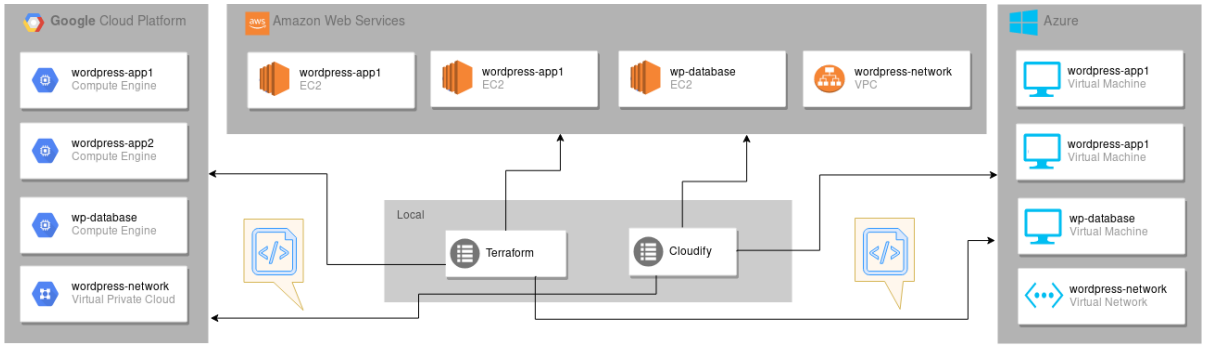

Figura 3.15: Arquitetura do experimento para avaliação dos orquestradores.

Uma vez definidos os orquestradores que atenderam aos requisitos, foi realizado um experimento de avaliação. Após elaborar os *blueprints* equivalentes para cada orquestrador, vários ciclos de provisionamento e desprovisionamento foram executados em diversas regiões geográficas dos diferentes fornecedores utilizando os *scripts* mostrados nas Listagens 3.1 e 3.2. O *script* apresentado na Listagem 3.1 coordena a execução dos procedimentos de provisionamento e desprovisionamento em ambos orquestrador, além de criar os arquivos de resultado que serão preenchidos a cada rodada dos testes. O *script* da Listagem 3.1 aciona o script apresentado na Listagem 3.2 para que este então execute o caso de teste em si, ou seja, comande o provisionamento e em seguida o desprovisionamento do ambiente, e posteriormente alimente os valores obtidos como resultado.

Listagem 3.1: *Shell script* para execução do experimento.

```
source common/.set_credentials.sh
DIR=$(pwd)
list_orchestrators='cloudify terraform'
list_providers='azure gcp aws'
list_aws_region='us-east-1 eu-west-2 sa-east-1'
list_gcp_region='us-east4 europe-west2 southamerica-east1'
list_azure_region='EastUS UKSouth BrazilSouth'
test_executions=3
echo "Starting tests..."
EXEC_DATE=$(date +%Y-%m-%d-%H-%M-%S)
mkdir -p ${DIR}/executions/${EXEC_DATE}
RESULTS=${DIR}/executions/${EXEC_DATE}/results
echo "date,process,orchestrator,provider\
,region,execution,timestamp,cpu,mem,io,net,duration" > ${RESULTS}
execution=1
while [ $execution -le $test_executions ]; do
  for orchestrator in $list_orchestrators; do
    source $orchestrator/test/cleanup.sh
    for provider in $list_providers; do
      source common/providers_regions.sh
      for region in $list_region; do
        source common/providers_zones.sh
        source ${orchestrator}/test/commands.sh
        source common/test_case.sh
      done
    done
  done
  execution=$((execution + 1))
done
echo "Cleaning up..."
for orchestrator in $list_orchestrators; do
  source ${DIR}/${orchestrator}/test/cleanup.sh
done
echo "Tests finished"
```

Listagem 3.2: *Shell script* para execução de um caso de teste.

```
LOCAL=$DIR/executions/${EXEC_DATE}/${orchestrator}/${provider}/${region}/${execution}
mkdir -p $LOCAL
PROVISION_RUN=true
CONT=0
RETRY=2
cd ${DIR}/${orchestrator}/${provider}
while [ $PROVISION_RUN ] && [ $CONT -le $RETRY ]; do
  echo "Starting provisioning process" > $LOCAL/provision.log
  test_monitor.sh ${RESULTS} ${EXEC_DATE} provision \
  ${orchestrator} ${provider} ${region} ${execution} &
  monitor_pid=$!
  $cmd_provision  2>&1 >> $LOCAL/provision.log
  kill -9 $monitor_pid &> /dev/null
  if [ "${orchestrator}" == "cloudify" ]; then
    cfy deployments outputs cloudify-${BPID}-wp-${provider} >> $LOCAL/provision.log
  fi
  echo "validating provisioning process..."
  if [ "$(cat $LOCAL/provision.log | grep apply-finished-wp)" == "" ]; then
    echo "Provision failed"
    if [ $CONT -eq $RETRY ]; then
      echo "No more retrys... skipping"
      break
    else
      echo "Retrying..."
    fi
    CONT=$((CONT + 1))
    echo "failed" >> $RESULTS
    source test/cleanup.sh
  else
    echo "Provision successfully"
    echo "success" >> $RESULTS
    PROVISION_RUN=false
    break
  fi
done
echo "waiting for unprovision..."
```

```
sleep 10
touch $LOCAL/unprovision.log
test_monitor.sh $RESULTS ${EXEC_DATE} unprovision \
${orchestrator} ${provider} ${region} ${execution} &
monitor_pid=$!
$cmd_unprovision  2>&1 > $LOCAL/unprovision.log
kill -9 $monitor_pid &> /dev/null
cd -
```

A Tabela 3.2 mostra os parâmetros usados no experimento. A equalização desses parâmetros entre os provedores visa mitigar a interferência das diferenças na prestação de serviços. Portanto, a configuração das máquinas virtuais usadas é equivalente entre os provedores. Além disso, as regiões onde os ambientes foram provisionados estão fisicamente próximas, apesar de explorar as diferenças no posicionamento global. As regiões foram definidas em Londres, Europa, no estado americano da Virgínia e em São Paulo, Brasil. Os outros parâmetros de configuração foram definidos de forma idêntica nos três provedores.

Tabela 3.2: Parâmetros do experimento para avaliação dos orquestradores.

	AWS	GCP	Azure
Máquinas virtuais	t2.micro	f1-micro	Standard_B1ls
Sistema Operacional	Ubuntu 14.04	Ubuntu 14.04	Ubuntu 14.04
MySQL	5.5	5.5	5.5
Apache	2.4.7	2.4.7	2.4.7
PHP	5.5.9	5.5.9	5.5.9
Wordpress	3.8.5	3.8.5	3.8.5
Regiões	- us-east-1 - eu-west-2 - sa-east-1	- us-east4 - europe-west2 - southamerica-east1	- EastUS - UKSouth - BrazilSouth
Execuções	3 em cada região 9 por orquestrador	3 em cada região 9 por orquestrador	3 em cada região 9 por orquestrador
Portas do Firewall	80, 3306 e 22	80, 3306 e 22	80, 3306 e 22

Durante as execuções de provisionamento, as informações do estado da máquina foram coletadas executando os orquestradores. Para esse experimento, foi usada uma máquina física com processador Intel Core i5, 8 GB de RAM executando o sistema operacional Linux CentOS 7.6. Para coletar o uso da CPU, memória, disco e tráfego de rede, foram usadas as ferramentas: mpstat, free, iotop e iftop, respectivamente. Esses aplicativos são controlados por *scripts* em *shell* específicos para o experimento. Os *scripts* e os *blueprints* usados nesta experiência estão disponíveis no GitHub[4].

As rodadas de processos de provisionamento e desprovisionamento realizadas durante o fluxo do experimento foram monitoradas por coletores de métricas instalados na mesma máquina em que os orquestradores estavam sendo comandados. Esses monitores registravam os valores calculados para a porcentagem de uso da CPU, memória, leitura e gravação de disco, e atividade de rede. Os dois primeiros (CPU e memória) foram calculados pela diferença média entre o término e os tempos iniciais registrados. As atividades de disco

[4]https://github.com/leonardoreboucas/cloud-orchestrators-exam

Tabela 3.3: Avaliação do experimento entre o Cloudify e o Terraform.

Orquestrador	Versão	Iniciação	Tipo de Iniciação	Tamanho da iniciação	Guia de referência	Execuções da arquitetura de referência para Wordpress											
						AWS				GCP				Azure			
						KB	RG	TMP	TMD	KB	RG	TMP	TMD	KB	RG	TMP	TMD
Terraform	0.12.6	Download	- Binário	~15mb	Fácil	12,0	17,0	3,3	1,5	12,0	10,0	4,6	1,9	16,0	19,0	6,2	5,1
Cloudify	Comunity 19.07.18	Imagem	- Docker - CLI	~1,73gb ~50mb	Difícil	12,0	20,0	4,3	3,5	8,0	7,0	4,3	3,5	16,0	19,0	15,3	9,6
		Instalação de ambiente	- Instação local - CLI - Depêndencias	~380mb ~50mb ~1,2gb													
		Imagem	- Máquina virtual - CLI	~3,3gb ~50mb													

KB - Tamanho do arquivo do blueprint - **RG** - Recursos gerenciados no blueprint - **TMP** - Tempo médio de provisionamento (min.) - **TMD** - Tempo médio de desprovisionamento

e de rede foram cumulativas. Os registros desses monitores também foram usados para calcular o tempo médio de execução.

A Tabela 3.3 mostra um quadro avaliativo geral dos resultados do experimento. Além das colunas para identificar os orquestradores e suas características que impactaram o experimento, os valores das médias globais dos tempos de execução do processo de provisionamento (colunas TMP) e desprovisionamento (colunas TMD) gravados por cada orquestrador em todos os provedores. Também estão presentes na Tabela 3.3 os valores de tamanho, em kilobytes (colunas KB), de cada *blueprint*, bem como a quantidade de recurso gerenciada em cada arquivo (colunas RG). Além disso, a coluna "Tamanho da inicialização" mostra a quantidade de dados que precisam ser baixados no ambiente local para usar os orquestradores. A coluna "Guia de referência" mostra uma avaliação subjetiva da documentação encontrada para cada ferramenta. Essa avaliação foi baseada na experiência da criação de projetos e da execução dos testes. Além disso, as células cuja coloração é esverdeada contém um resultado, que foi obtido pelo respectivo orquestrador, classificado como melhor do que o resultado obtido pelo seu oponente, enquanto as células avermelhadas possuem valores que indicam inferioridade do orquestrador naquele quesito.

A análise desses resultados permite inferir que, em geral, o Terraform obteve melhores resultados que o Cloudify, uma vez que a incidência pontos positivos (apresentados com fundo esverdeado) na linha do Terraform da na Tabela 3.3 é claramente maior do que a observada na linha Cloudify. Esse resultado está alinhado com a avaliação apresentada na coluna "Guias de referência", porque durante a construção de projetos para os três fornecedores, explorar a documentação fornecida pelos fabricantes seria uma maneira natural. No entanto, a experiência com a documentação oficial disponível para Cloudify no *site* do fabricante não foi suficiente para construir efetivamente os artefatos de definição de infraestrutura. A descrição detalhada dos atributos de cada recurso não é facilmente encontrada na documentação, requer que o desenvolvedor consulte exemplos fora da documentação oficial. Por outro lado, para a construção dos projetos da Terraform, basta

Tabela 3.4: Resultados consolidados do experimento.

Processo	Orquestrador	Provedor	Média de Execuções				
			Duração (minutos)	CPU (%)	RAM (Gb)	Disco (Mb)	Rede (Mb)
Provisionamento	Cloudify	GCP	5,9	24,43	0,95	1,84	2,27
Desprovisionamento	Cloudify	GCP	2,89	24,89	0,89	0,50	0,11
Provisionamento	Cloudify	AWS	4,34	10,64	1,00	2,29	0,97
Desprovisionamento	Cloudify	AWS	3,52	10,69	1,02	2,34	0,62
Provisionamento	Cloudify	Azure	15,33	20,95	0,96	2,06	3,49
Desprovisionamento	Cloudify	Azure	9,57	23,76	0,86	1,95	0,64
Provisionamento	Terraform	GCP	4,64	15,82	0,98	0,15	1,43
Desprovisionamento	Terraform	GCP	1,85	6,57	0,90	0,03	0,21
Provisionamento	Terraform	AWS	3,29	17,61	0,92	0,04	1,34
Desprovisionamento	Terraform	AWS	1,52	17,6	0,94	0,21	0,73
Provisionamento	Terraform	Azure	6,22	17,53	0,89	0,16	1,27
Desprovisionamento	Terraform	Azure	5,12	17,5	0,93	0,30	0,41

consultar o *site* da Hashicorp para obter exemplos que poderiam ser aprimorados até que os requisitos fossem atendidos completamente, sem nenhuma interação com orientações externas.

Além disso, a Tabela 3.4 mostra o resultado consolidado das diversas execuções de provisionamento e desprovisionamento. As linhas levemente esverdeadas contém informações de processos de provisionamento, enquanto as linhas levemente avermelhadas mostram dados relacionados os processos de desprovisionamento. Nessa tabela é possível avaliar a duração média de cada processo em cada provedor auferida pelos orquestradores. Da mesma forma é possível analisar os dados referentes a CPU, memória RAM, atividade de leitura e escrita em disco e tráfego de rede.

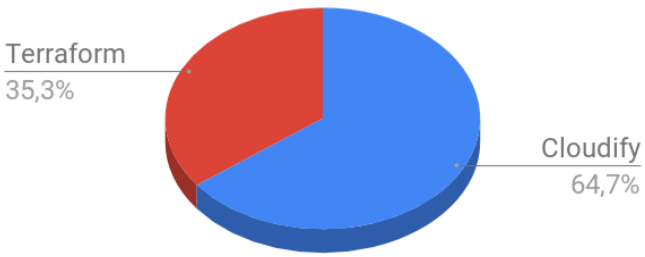

Figura 3.16: Alocação global de recursos.

A Figura 3.16 mostra a distribuição de todos os recursos computacionais envolvidos neste experimento entre os orquestradores. Nessa figura é possível notar que, durante a execução do experimento, o Cloudify manteve cerca de ⅔ dos recursos envolvidos no processo alocado, enquanto o Terraform exigiu apenas ⅓ desses recursos para realizar a mesma tarefa. Portanto, o Cloudify foi menos eficiente do que o Terraform no cenário do experimento, especialmente, em termos de uso da CPU, como mostra a Figura 3.17.

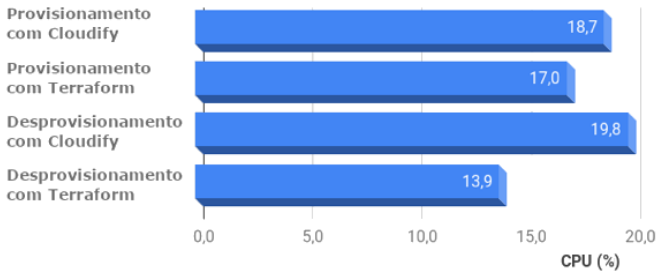

Figura 3.17: Alocação média de CPU.

Para a tarefa de provisionamento, o Terraform foi, em média 44% mais rápido do que o Cloudify. Além disso, como mostra a Figura 3.18, o Terraform levou uma média de 4,7 minutos para provisionar o ambiente de teste, enquanto o Cloudify levou uma média de 8,5 minutos. Da mesma forma, no processo de desprovisionamento, o Terraform executou a tarefa, em média, 47% mais rápido que o Cloudify.

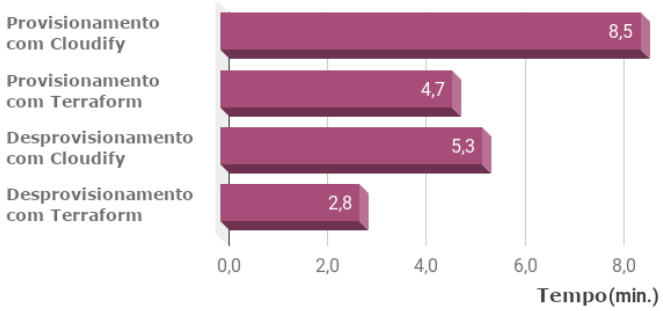

Figura 3.18: Tempo médio de execução.

O Cloudify provou ser muito mais intensivo em rede durante os testes do que o Terraform. A diferença foi ainda mais acentuada nos processos de provisionamento, pois enquanto o tráfego do Terraform atingiu, em média, 1,3 megabytes durante o processo, o Cloudify precisou de quase o dobro do tráfego para realizar a mesma operação, consumindo, em média, 2,3 megabytes. Essa disparidade pode ser vista claramente na Figura 3.19.

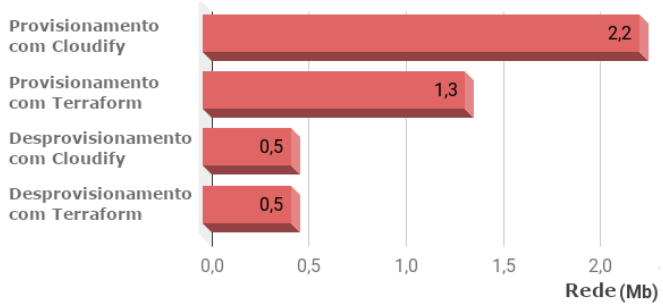

Figura 3.19: Tráfego médio de rede.

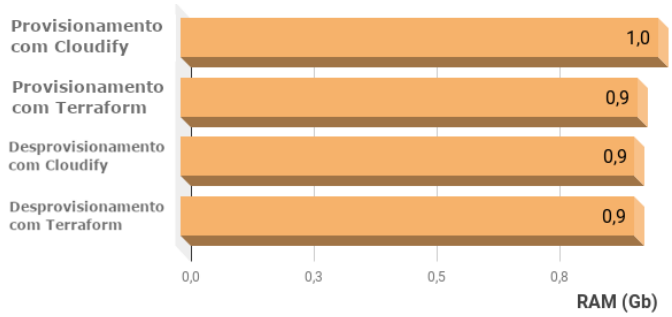

Figura 3.20: Alocação média de memória.

Na Figura 3.20 verifica-se que não houve diferenças significativas no consumo de RAM entre os orquestradores.

Todavia, embora houvesse diferenças de desempenho em tempo de execução, uso da CPU e tráfego de rede, nenhuma era tão significativa quanto a atividade do disco mostrada na Figura 3.21. O Terraform apresentou baixa atividade média do disco, registrando entre 0,1 e 0,2 megabytes acumulados durante os processos. Enquanto o Cloudify variou de 1,6 a 2,1 megabytes acumulados. Isso mostra que o Cloudify faz mais uso das operações do disco, enquanto o Terraform concentra seu processamento na memória. Esse comportamento certamente contribuiu para o bom resultado obtido pelo Terraform em relação ao tempo de execução.

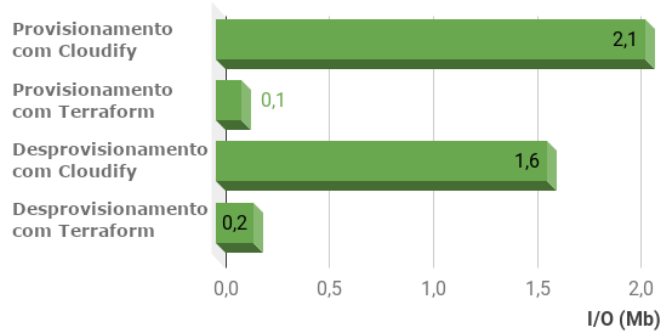

Figura 3.21: Atividade média de disco.

Diante do resultado do experimento, que mostrou que o Terraform apresentou um melhor desempenho geral na orquestração de um *blueprint* de uma aplicação de utilização real, optou-se por adotar esse orquestrador como a interface entre o *Framework* Node2FaaS (a ser apresentado no Capítulo 5) e os provedores de nuvem. Além disso, os resultados do referido experimento foram aceitos no *20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing - CCGRID 2020* com data de realização adiada para maio de 2021 devido a pandemia, em Melbourne, na Austrália.

3.5 Considerações Finais

O conceito de *sky computing*, apresentado neste capítulo, enfatiza o interesse existente no aprimoramento dos modelos de consumo de serviços em nuvem. O esforço contínuo para alinhar as necessidades dos negócios às abordagens computacionais existentes, corrobora o paradigma de computação em nuvem, no qual a abstração dos detalhes estruturais mantém o usuário concentrado no cerne da computação em si, ou seja, responder aos desafios enfrentados pelas organizações.

Assim, no sentido de tornar a computação cada vez mais focada na solução de problemas, tem crescido o modelo de entrega de serviços em nuvem que suprime os detalhes de processamento de um trecho específico de código. Esse modelo executa e retorna o resultado ao solicitante, sem a necessidade de definições de infraestrutura ou da plataforma, apenas a partir da inclusão do algoritmo. Nesse contexto, o modelo de serviço denominado *Function as a Service*, que foi brevemente comentado na Seção 2.4.4, será apresentado em detalhes no próximo capítulo.

Capítulo 4

Função como um Serviço

Neste capítulo serão apresentados os principais conceitos do modelo de entrega de serviços em nuvem denominado *Function as a Service*. O capítulo está dividido em seis seções, sendo a primeira Seção (4.1) reservada para as considerações iniciais, enquanto a última Seção (4.6) é reservada para as considerações finais. Na Seção 4.2 é apresentado o paradigma arquitetural de microsserviços. A Seção 4.3 traz as principais características que definem serviços FaaS. A Seção 4.4 descreve, em detalhes, os principais serviços de FaaS oferecidos atualmente pelos provedores. Na Seção 4.5 é apresentada a relação simbiótica que existe entre serviços FaaS e a Internet das Coisas.

4.1 Considerações Iniciais

Os modelos de programação têm evoluído ao longo do tempo, visando a otimização do software, bem como a sua capacidade de manutenção e de evolução, além do alinhamento ao negócio [105]. Inicialmente, o software era desenvolvido de forma monolítica, ou seja, toda a aplicação estava contida em um único bloco de software. Esse modelo ocasionava um alto acoplamento entre os componentes da aplicação, gerando degradação da manutenibilidade. Atualmente, é comum que os arquitetos de sistemas projetem suas aplicações de forma modular, separando os blocos de complexidade diferentes e agrupando conjuntos de funcionalidades relacionadas [106].

Para atender a demanda por plataformas tecnológicas que sustentem os paradigmas de desenvolvimento de software, seja monolítico ou segmentado, o modelo tradicional de administração de infraestrutura precisa investir maciçamente em tarefas de instalação e configuração, assim sendo, habilitar um ambiente para receber códigos fonte e executá-los adequadamente, em geral, consome muitas horas de trabalho. Para racionalizar esse esforço, a computação em nuvem definiu o modelo de serviço PaaS [31], cuja entrega consiste em uma plataforma completamente configurada e pronta para receber o código

fonte. Todavia, esse tipo de serviço ainda expõe alguns detalhes da infraestrutura para o desenvolvedor que, eventualmente, ainda precisa garantir, entre outros aspectos, a elasticidade do ambiente. Além disso, é preciso manter o ambiente provisionado durante o período de desenvolvimento, e isso aumenta o custo final do projeto.

Nesse contexto, os provedores passaram a oferecer um modelo de serviço que entrega uma plataforma completamente encapsulada para desenvolvimento de software e cobra apenas pelo processamento efetivamente executado pela plataforma. Esse tipo de serviço, conhecido como *Function as a Service*, função como um serviço (FaaS) [23], tem ganhado bastante evidência recentemente, uma vez que a sua abordagem vai ao encontro do modelo arquitetural que tem sido bastante difundido na atualidade, que é o paradigma de microsserviços, o qual será melhor abordado na próxima seção.

4.2 Microsserviços

O termo microsserviços tem sido utilizado desde 2014 em comunidades de desenvolvimento ágil [107]. Os padrões e os princípios que permeiam o conceito de microsserviços incluem [107]:

- Desenvolvimento orientado ao negócio e nativamente baseado em nuvem;
- Aplicação de múltiplos paradigmas de desenvolvimento, como funcional e imperativo;
- Aplicações executando em serviços leves de contêiner, como Docker [108];
- Entrega contínua descentralizada;
- Utilização de cultura DevOps (desenvolvimento integrado à operação);
- Utilização de API REST.

A decomposição de aplicações monolíticas em subconjuntos de módulos de funcionalidades específicas, geralmente demanda a aplicação de mecanismos de integração que promovam uma visão atômica da aplicação. Considerando o contexto de microsserviços, no qual esses módulos são executados em processos diferentes, até mesmo em máquinas separadas, uma alternativa para intercâmbio de informações é a adoção do "protocolo" REST [109].

O REST é um estilo arquitetural proposto por Roy Thomas Fielding [109] e definido como um conjunto coordenado das seguintes restrições arquiteturais: cliente-servidor, sem estado, uso de *cache*, interface uniforme, sistema em camadas e código opcional sob demanda. As três primeiras restrições também são predominantes na web desde sua

arquitetura inicial, enquanto as três seguintes foram definidas e aplicadas à medida que a arquitetura da web evoluiu [110].

Nos serviços web REST, o *Hypertext Transfer Protocol* (HTTP) é normalmente utilizado como o protocolo de comunicação, junto com o padrão URI que serve como um mecanismo universal para expressar identificadores de recursos. Mais especificamente, os métodos do HTTP (por exemplo, POST, GET, PUT, DELETE) são utilizados para manipular recursos (cito criar, recuperar, atualizar, excluir, etc), e os URIs HTTP são usados para identificar e localizar recursos informativos [110].

Para a construção de software, por meio dos princípios de microsserviços, é necessário segmentar o desenvolvimento das funcionalidades da aplicação [21]. A Figura 4.1 mostra a comparação da abordagem de microsserviços com a abordagem monolítica. Como pode ser visto, na abordagem monolítica toda a aplicação é colocada dentro de um único processo, enquanto que na abordagem de microsserviços, cada funcionalidade é colocada em um serviço diferente. Com isso, na abordagem monolítica toda a aplicação precisa ser escalada, no caso de microsserviços, apenas aqueles serviços com maior demanda serão escalados (proporcionando maior racionalidade no consumo de recursos) [21].

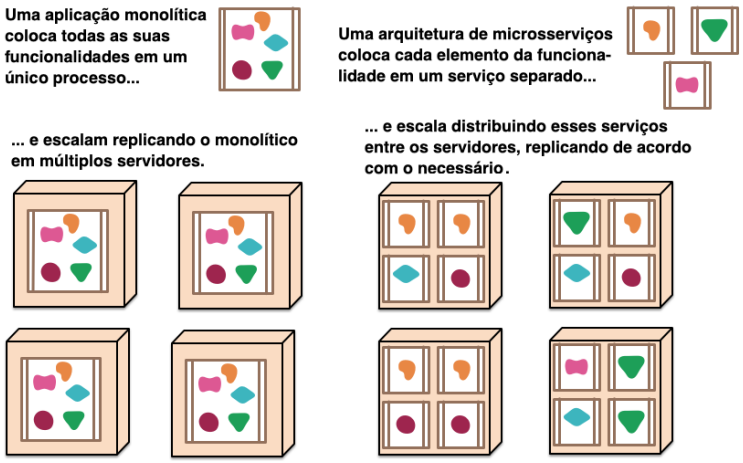

Figura 4.1: Organização de microsserviços, adaptado de [21].

A Figura 4.2 mostra a forma de execução de módulos em ambientes monolíticos e orientados a microsserviços. Assim, é possível observar na Figura 4.2 a segmentação das funcionalidades que ocorre na abordagem de microsserviços, na qual cada módulo da aplicação é executado em um processo diferente, enquanto no monolítico há um compartilhamento do processo entre todos os módulos da aplicação.

Assim, considerando que normalmente a utilização dos módulos não é uniforme, ou seja, cada módulo tem uma carga de trabalho diferente, é possível que no modelo monolítico ocorra um desperdício de recursos, embora alguns módulos não sejam completamente utilizados, eles precisam ser instanciados para permitir a escala dos módulos mais sobre-

carregados. Por outro lado, efeito diferente ocorre no modelo orientado à microsserviços, no qual somente os módulos mais utilizados serão efetivamente escalados [21].

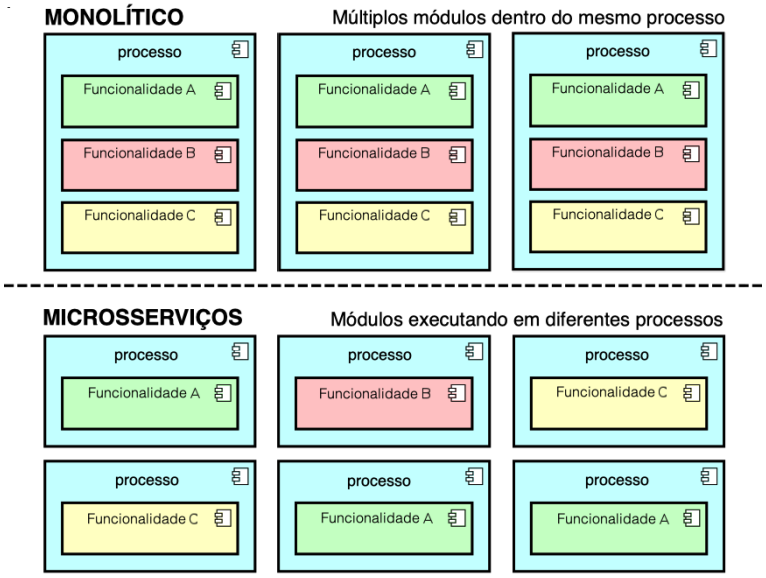

Figura 4.2: Segmentação de funcionalidades com microsserviços e monolítica, adaptado de [22].

Assim sendo, a arquitetura de microsserviços tem ganhado evidência por proporcionar boa capacidade de escala, bem como por melhorar o processo de manutenção de software [107]. Nesse contexto, os provedores de nuvem elaboraram um modelo de serviço aderente e adequado ao paradigma de microsserviços, o FaaS, cujas características fundamentais serão apresentadas na próxima seção.

4.3 Características Principais de FaaS

A fim de atender a demanda por infraestrutura para aplicações baseadas em microsserviços, entre outras aplicações, alguns provedores de nuvem passaram a oferecer o modelo de funções como serviço. Nele o cliente contrata a execução de uma função predefinida, carrega o código que deseja executar, e recebe um endereço de acesso ao serviço. As aplicações que utilizam esse tipo de serviço de nuvem têm sido denominadas aplicações *"serverless"*, uma vez que a aplicação não possui um servidor específico e seu funcionamento é baseado em requisições feitas à API do provedor [22].

A Figura 4.3 mostra o *workflow* de *Functions as a Service*. Esse *workflow* mostra como ocorre a interação entre o desenvolvedor e o serviço de FaaS, seja efetivamente carregando código, consultando as APIs dos SDKs do provedor ou configurando gatilhos

para invocação do serviço. Além disso, a Figura 4.3 mostra a elasticidade do serviço e seu modelo de pagamento sob demanda (*pay as you go*).

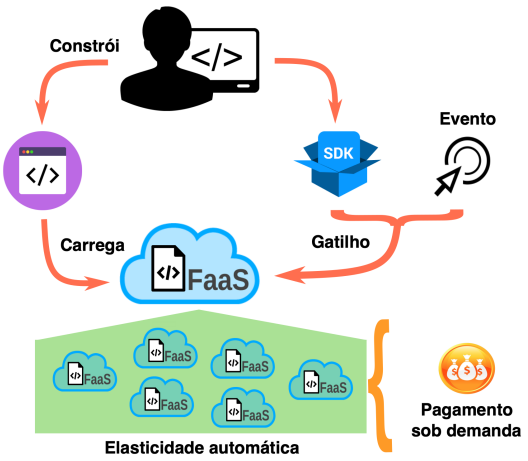

Figura 4.3: Fluxo de trabalho de FaaS, adaptado de [23].

Assim sendo, os serviços de FaaS oferecem como principais benefícios da sua adoção [24] [111]:

- Supressão da necessidade de gerenciamento de servidores;
- Elasticidade contínua automática;
- Pagamento sob demanda;
- Aumento da flexibilidade do projeto;
- Redução dos riscos.

Dentre esses benefícios, destaca-se a elasticidade, uma vez que em aplicações críticas a alta disponibilidade é uma característica fundamental. Além disso, utilizando FaaS o gerenciamento fica focado na aplicação e não na infraestrutura [8], liberando recurso para aprimorar a lógica de negócio.

Em serviços de PaaS há a necessidade de manter o ambiente em execução e gerando custos, a fim de garantir o atendimento às requisições de processamento. Já em serviços de FaaS, o provedor fica responsável por manter o serviço acessível sem interrupção, embora a cobrança seja feita somente quando o serviço é efetivamente acionado. Isso pode representar uma economia de custos considerável, sobretudo se o serviço em questão atravessar longos períodos de ociosidade. Nesses casos, o ambiente fora de uso não geraria qualquer custo ao proprietário, ao passo que se estivesse sustentado em um serviço de PaaS tradicional haveria um custo fixo inevitável.

Por outro lado, as desvantagens da adoção de modelos baseado em FaaS incluem a eventual necessidade de adaptações na aplicação em situações nas quais uma solução exija configurações personalizadas para determinados recursos do sistema operacional e que a utilização da abordagem de FaaS possa incorrer em um dificultador para a aplicação dessas configurações [112]. Além disso, pode ocorrer um atraso na resposta da primeira execução da função após algum tempo sem uso. Isso acontece devido a estratégia do provedor de deixar a instância que atende a função ligada apenas alguns minutos depois da última chamada [112]. Outro potencial problema são os limites estabelecidos pelo provedores para quantidade de CPU, memória, tempo de execução e limiares de elasticidade [23]. A latência ocasionada pela necessidade de atravessar a rede para executar o processamento é outra desvantagem desse modelo [111].

Além disso, uma dificuldade enfrentada pelos desenvolvedores para adoção de FaaS é a implementação. Conhecer adequadamente o funcionamento do serviço pode significar a diferença entre obter sucesso na adoção ou abandonar a abordagem após algum pequeno insucesso. Entretanto para atingir esse patamar, pode ser necessária uma quantidade de investimento em tempo de aprendizado tão significativa que desmotive a adoção do mesmo.

Contudo, o crescimento dos serviços de FaaS tem acompanhado o próprio crescimento da computação em nuvem. Atualmente, os principais provedores desse mercado oferecem opções para processamento sem servidor. Na próxima seção serão descritos os principais serviços desse modelo.

4.4 Serviços de FaaS

Atualmente, os principais provedores de nuvem pública possuem serviços de FaaS em seus catálogos. A pioneira Amazon, oferece o serviço AWS Lambda [24]. Esse serviço permite processamento de NodeJS, Python, Java, Go, Ruby e C# através do .NET Core e disponibiliza um pacote mensal de um milhão de requisições gratuitas, antes de iniciar a cobrança pelo serviço. O provedor possui diversas parcerias para a implantação, o monitoramento, o gerenciamento de código e a segurança. A Figura 4.4 mostra um exemplo de execução do FaaS da Amazon para processamento de imagens usando AWS Lambda. Nesse exemplo quando uma foto é carregada no serviço de armazenamento Amazon S3, então um gatilho aciona a função Lambda que carrega a imagem e executa seu redimensionamento.

O Google oferece o serviço Cloud Functions [25]. Esse serviço permite processamento de funções escritas em Python, NodeJS e Go, e disponibiliza uma cota inicial de dois

Figura 4.4: Execução do AWS Lambda, adaptado de [24].

milhões de requisições por mês e inicia a cobrança apenas quando essa cota é ultrapassada. A Figura 4.5 mostra o funcionamento do serviço de FaaS do Google.

Figura 4.5: Fluxo do Cloud Function, adaptado de [25].

A Microsoft, por meio do seu serviço de nuvem Azure, disponibiliza o serviço Azure Functions [113]. Esse serviço permite o carregamento de funções escritas em C#, F#, NodeJS, Java, Python ou PHP [113]. O provedor informa em seu portal que as funções são disponibilizadas em ambiente Windows, embora isso fique transparente para o usuário. Apesar disso, há uma previsão de disponibilização de ambientes usando Linux. O provedor possibilita diversidade de uso do seu serviço, como mostrado na Figura 4.6, que demonstra o processamento de arquivos *Portable Document Format* (PDF). Ele oferece gratuidade mensal para o primeiro milhão de chamadas ao serviço.

Figura 4.6: Processamento de arquivos PDF usando o Azure Functions, adaptado de [8].

O líder em nuvem do mercado asiático, Alibaba Cloud, oferece o serviço Function Compute [114], que permite o processamento de códigos escritos em NodeJS, Java, Python, Go, PHP e C#. Esse serviço também oferece uma cota inicial gratuita mensal de um milhão de requisições, e pratica tarifas diferentes em cada região onde opera.

O serviço IBM Cloud Functions [115], oferecido pelo IBM Cloud, processa funções escritas em NodeJS, Python, Go, Java, PHP, Ruby, Ballerina e Swift [116] através do Apache OpenWhisk[1]. Com o OpenWhisk é possível processar funções sobre contêineres Docker. Assim sendo, ele é capaz de estender as possibilidade de linguagens de processamento a qualquer uma que possa ser instalada em um contêiner Docker. O provedor oferece processamento gratuito para cinco milhões de requisições por mês.

Em dezembro de 2018 o provedor Oracle Cloud anunciou que lançaria em 2019 o serviço Oracle Function, nos moldes de Function as a Service [117] e isso de fato ocorreu em agosto de 2019 [118]. Assim como o IBM Cloud Functions, o Oracle Functions é sustentado por uma plataforma de processamento de código livre, mas, diferentemente da IBM, o Oracle resolveu utilizar o *framework Fn*[2]. Como o *Fn* executa sobre Docker, ele potencialmente pode processar códigos em qualquer linguagem de programação.

Uma alternativa ao *Fn* e ao Apache OpenWhisk é o Kubeless [61]. O Kubeless é um *framework* de código aberto, nativamente sem servidor, que permite implementar pequenos trechos de código (funções) sem preocupação com a infraestrutura subjacente. Ele é projetado para ser implementado no topo de um *cluster Kubernetes* e aproveitar todas as primitivas desse escalonador de Docker [120]. Assim, semelhante ao Kubeless, o OpenFaaS [121] é um *framework* para criação de funções sem servidor com o Docker e o Kubernetes, que tem suporte de primeira classe para métricas. Qualquer processo pode ser empacotado como uma função, permitindo que você consuma uma variedade de eventos da web sem codificação repetitiva.

Além das abordagens adotadas pelo principais provedores de nuvem pública, tem surgido outras plataformas de processamento de FaaS com propostas específicas e estratégias diferenciadas. O Abaco [122], por exemplo, é um projeto do *Texas Advanced Computing Center* (TACC) [123] e suporta funções escritas em uma ampla variedade de linguagens de programação e conta com redimensionamento automático da infraestrutura (elasticidade). O Abaco implementa o modelo de ator no qual um ator é um tempo de execução do Abaco mapeado para uma imagem específica do Docker. Cada ator é executado em

[1]O Apache OpenWhisk é uma plataforma distribuída e de código aberto que executa funções em resposta a eventos em qualquer escala. O OpenWhisk gerencia a infraestrutura, os servidores e o dimensionamento usando contêineres Docker [116]

[2]O projeto *Fn* é uma plataforma sem servidor, de código aberto e nativo de contêiner, que pode ser executado tanto em nuvem, quanto localmente. É fácil de usar e suporta todas as linguagens de programação. Além disso, ele é extensível e de alto desempenho [119].

resposta às mensagens postadas em sua caixa de entrada. Além disso, o Abaco fornece monitoramento detalhado de eventos e estatísticas de contêiner, estado e execução.

Outra plataforma que apresenta uma estratégia diferenciada é o SAND [124]. Essa ferramenta consiste em uma plataforma FaaS leve e de baixa latência da Nokia Labs que fornece *sandbox* no nível do aplicativo e um barramento de mensagens hierárquico. Os autores afirmam que eles atingem uma aceleração de 43% e uma redução de latência de 22 vezes em relação ao Apache OpenWhisk em aplicativos de processamento de imagem comumente usados. Além disso, o SAND fornece suporte para encadeamento de funções por meio de fluxos de trabalho enviados pelo usuário. O SAND não suporta multilocação, apenas com isolamento no nível do aplicativo. Além disso, ele é de código fechado e, portanto, não pode ser baixado e instalado localmente.

Para processamento de cargas de alto desempenho, a solução funcX [29] fornece uma plataforma FaaS escalável e de baixa latência que pode ser aplicada aos recursos HPC existentes com o mínimo de esforço. Essa redução emprega contêineres no espaço do usuário para isolar e executar funções, evitando as preocupações de segurança que proíbem o uso de outras plataformas FaaS. Por fim, fornece uma interface intuitiva para executar cargas de trabalho científicas e inclui várias otimizações de desempenho para suportar casos de uso científicos amplos.

A oferta de serviços de FaaS pelos principais provedores, bem como o crescente surgimento de plataformas para sustentação desse tipo de serviço de nuvem, mostra o grande interesse que existe no modelo de FaaS. A Tabela 4.1 apresenta uma visão comparativa entre as mais comuns plataformas de sustentação dos serviços de FaaS citadas. É importante perceber a grande quantidade de linguagem suportada. Isso demonstra que a amplitude de cobertura desse serviço abrange aplicações desenvolvidas em uma variedade considerável de tecnologias. Logo, fica claro que o acionamento desse tipo de serviço por meio de requisições HTTP é um padrão entre todas as plataformas. Além disso, o Docker está se tornando a principal alternativa de virtualização entre as soluções.

Assim sendo, a importância do FaaS aumenta a medida em que mais aplicações são desenvolvidas utilizando essa abordagem. Esse fato implica a necessidade de obtenção de melhorias na forma de entrega e utilização desse modelo, seja por meio do aprimoramento das plataformas ou o desenvolvimento de soluções que promovam uma melhor estratégia para o embarque nessa nova abordagem, tais como a proposta deste trabalho que será melhor apresentada no Capítulo 5.

Contudo, embora os serviços de FaaS sejam apropriados para arquitetura orientadas a microsserviços, sua utilização não está restrita a esse paradigma. Diversos outros casos de uso podem se beneficiar das vantagens do processamento *serverless*, ou seja, sem servidor, tais como execução programada de tarefas, execuções de fluxos *Extract, Transform*

Tabela 4.1: Pesquisa taxonômica das mais comuns plataformas de FaaS, adaptado de [29].

Plataforma	Linguagens	Infraestrutura Pretendida	Virtualização	Gatilhos	Tempo Máximo de Execução (s)	Bilhetagem
Amazon Lambda	C#, Go, Java, Powershell, Ruby, Python e Node.js	Nuvem Pública e Borda	Firecracker (KVM)	HTTP e AWS services	900	Requisições, tempo de execução e memória
Google Functions	BASH, Go, Node.js e Python	Nuvem Pública	Indefinido	HTTP, Pub/Sub e storage	540	Requisições, tempo de execução e memória
Azure Functions	C#, F#, Java, Python e JavaScript	Nuvem Pública e Local	Imagens de S.O.	HTTP, APIM e serviços da Microsoft	600	Requisições, tempo de execução e ANS
OpenWhisk	Ballerina, Go, Java, Node.js e Python	Kubernetes, Nuvens Pública e Privada	Docker	HTTP, IBM Cloud e OW-CLI	300	IBM Cloud: Requisições e tempo de execução. Local: N/A
Kubeless	Node.js, Python .NET, Ruby Ballerina e PHP	Kubernetes	Docker	HTTP, Agendamentos e Pub/Sub	Indefinido	N/A
SAND	C, Go, Java, Node.js e Python	Nuvem Pública e Nuvem Privada	Docker	HTTP e eventos internos	Indefinido	Gatilhos
Fn	Go, Java, Ruby, Node.js e Python	Nuvem Pública e Kubernetes	Docker	HTTP e acionamento direto	300	N/A
Abaco	Container	Clusters TACC	Docker	HTTP	Indefinido	Indefinido
funcX	Python	Local, Nuvens, Clusters e Supercomputadores	Singularity, Shifter e Docker	HTTP e Globus Automate	Sem limite	HPC: Sus e créditos de nuvem. Local: N/A

and Load (ETL), entrega de conteúdo estático, entre outros. Além dos casos de uso já citados, existe uma área que tem se beneficiado sobremaneira do paradigma sem servidor, a Internet das Coisas, cuja descrição e maiores detalhes podem ser conferidos na próxima seção.

4.5 Internet das Coisas

O termo *Internet of Things*, Internet das Coisas (IoT) foi usado pela primeira vez em 1998 [125] e, nos últimos anos, o conceito associado a essa expressão tem despertado muito entusiasmo. Descritivamente, a IoT pode ser considerada uma série de tecnologias digitais disruptivas, influenciando o cotidiano de indivíduos e empresas [126]. Em consonância com esse fenômeno, as empresas estão se tornando mais inteligentes no desenvolvimento, adoção e adaptação de tecnologias disruptivas em seus processos de negócios, para aumentar sua eficiência e inovação através de fluxos de conhecimento e coleta de dados e informações [127].

A Figura 4.7 apresenta exemplos da lógica entre os dispositivos de IoT e seus serviços associados. Uma lixeira trabalhando como IoT, por exemplo, tem a sua capacidade de armazenamento relacionada ao serviço de recolhimento de lixo. Dessa forma, assim que sua capacidade atingir um determinado nível, os sensores detectam e acionam automaticamente o mecanismo de recolhimento do lixo, evitando um desagradável transbordamento de descartados.

COISA	+	É (Hardware / Software)	=	FUNÇÃO CONEXA (Física / Local)	+	SERVIÇO DA COISA (Digital / Global)
	Lâmpada	Pilha IoT	→	Iluminação		Segurança
	Lixeira	Pilha IoT	→	Capacidade de armazenamento		Recolhimento Automático
	Trator	Pilha IoT	→	Reboque de veículos		Manutenção preventiva, Otimização

Figura 4.7: Lógica de produto e serviço em IoT, adaptado de [26].

Diante da economia contemporânea orientada pelo conhecimento e pela tecnologia, e caracterizada por tendências como a globalização, a convergência tecnológica e industrial, as empresas de sucesso utilizam mecanismos específicos para gerenciar o conhecimento [127]. Tanto os políticos, quanto os profissionais, reconhecem cada vez mais a Internet das Coisas como uma oportunidade real de negócios, e as estimativas atualmente sugerem que a IoT poderá crescer a um mercado de 1 trilhão de dólares até 2020 [128].

Esquematicamente a IoT pode ser considerada uma rede de dispositivos físicos, objetos e sensores conectados para oferecer diversos serviços às pessoas [129]. Esses objetos são conectados por meio de blocos de informações e comunicação. Na prática, objetos do cotidiano passam a contar com acesso via Internet a outros dispositivos e podem colaborar entre si. Nesse contexto, exemplos de utilização de IoT incluem [27]: veículos autônomos, cidades inteligentes, detecção de desastres, monitoramento de atividades, entre outras, conforme pode ser observado na Figura 4.8, que mostra alguns exemplos de sistemas de IoT.

Dessa forma, espera-se que cerca de $^3/_4$ das pessoas vivam em cidades inteligentes e ambientes inteligentes até 2050 [129]. Assim, a IoT pode desempenhar um grande papel na implementação desse conceito. Para o bem-estar das pessoas, em termos econômicos, sociais e ambientais, a cidade inteligente pode fornecer uma boa plataforma. A rede de

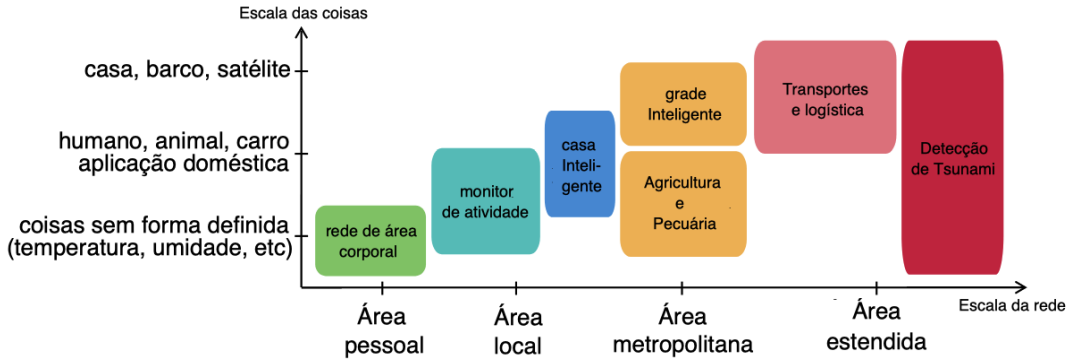

Figura 4.8: Classificação e exemplos de sistemas IoT, adaptado de [27].

sensores sem fio é a parte sensível da IoT que pode ser combinada com a infraestrutura urbana, formando uma cobertura espacial por meio de blocos de construção da IoT [129].

A IoT impulsionou o desenvolvimento de muitas aplicações. Com isso, o processamento de grandes quantidades de dados, gerados por dispositivos heterogêneos, tornou-se um desafio adicional [129]. Nesse cenário, a computação em nuvem figura como um importante aliado para o progresso do conceito de Internet das Coisas e, consequentemente, dos benefícios associados.

4.5.1 FaaS Aplicado a IoT

A área de IoT cresceu significativamente nos últimos anos e espera-se que atinja uma quantidade gigantesca de 50 bilhões de dispositivos até 2020 [130]. O surgimento de arquiteturas sem servidor levanta a questão da adequação de usá-las em ambientes de IoT. A combinação de IoT com *design* de arquitetura sem servidor pode ser eficaz ao tentar usar o poder de processamento existente em uma rede local de dispositivos de IoT, e criar uma camada de névoa, chamada de *Fog Commputing* [131], que aproveite os recursos de computação mais próximos do usuário final [130].

Trazer o conceito de FaaS para a IoT levanta vários problemas que desafiam os fundamentos da computação sem servidor (FaaS). Em primeiro lugar, os dispositivos de IoT, na maioria dos casos, não fazem parte da nuvem, mas estão na fronteira entre os domínios digital e o físico. Como tal, os dispositivos capturam, representam e manipulam o estado do Mundo real, o que torna a borda um ponto bastante relevante [28]. Outra questão é que, embora os cálculos possam ser efêmeros e de curta duração, sentir o Mundo físico é um processo contínuo, e as ações executadas podem ter consequências permanentes.

A Figura 4.9 mostra uma arquitetura de interação entre sensores IoT e uma nuvem através de serviços FaaS. Na arquitetura apresentada na Figura 4.9, utilizando API REST os sensores enviam informações aos serviços de FaaS, que por sua vez interagem com

outros componentes da nuvem como bancos de dados e serviços de armazenamento. Caso necessite encaminhar informações aos sensores, os serviços de FaaS também podem utilizar uma API REST.

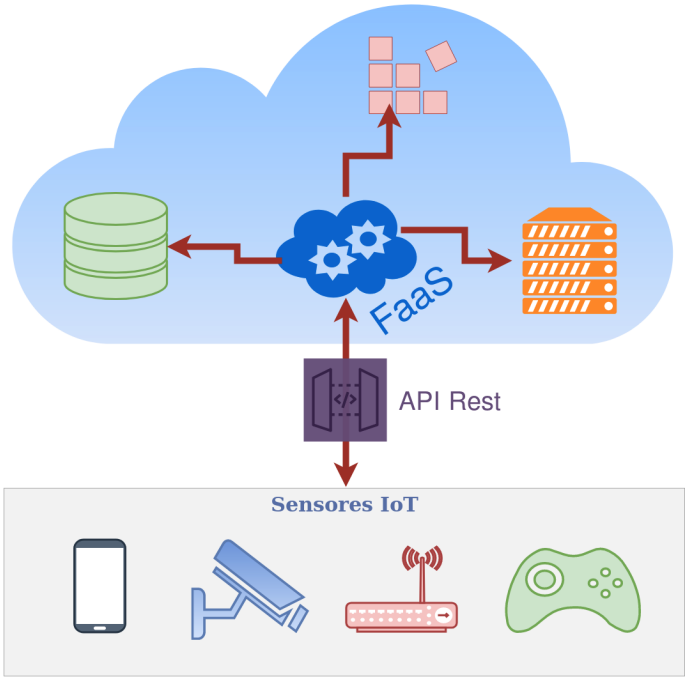

Figura 4.9: Integração entre sensores IoT e FaaS.

A abstração de hardware é a chave para um ecossistema próspero em torno de dispositivos IoT. Assim sendo, deve ser possível aos desenvolvedores escreverem e implantarem código, independentemente das especificidades dos dispositivos [28]. Da mesma forma, um dispositivo IoT deve ser capaz de consumir poder de processamento sem precisar conhecer as especificidades da linguagem de programação associada. Nesse contexto, a combinação entre IoT e FaaS, utilizando integração por meio de API REST, mostra-se como uma alternativa bastante eficaz.

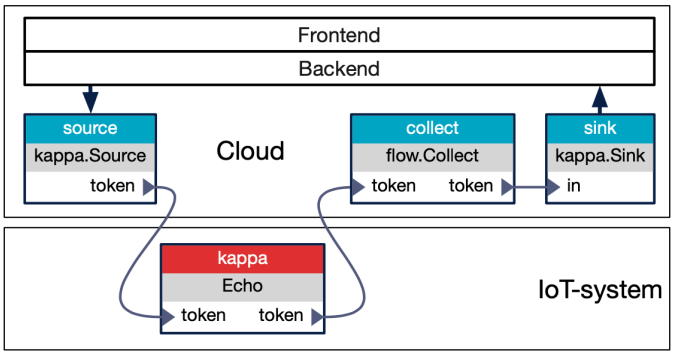

Figura 4.10: Arquitetura do *framework* Kappa [28].

Alguns trabalhos vem fazendo uso da composição de FaaS com IoT. O *framework* Kappa [28] é um exemplo dessa utilização. Essa solução foi construída para executar no topo da pilha sobre o *framework* IoT Calvin [132]. A sua arquitetura é mostrada na Figura 4.10, na qual é possível compreender que o Kappa atua como uma ponte entre a nuvem e o dispositivo IoT, utilizando uma abordagem de FaaS.

O artigo [131] apresenta a proposta de estender o conceito de FaaS para a computação em borda, na qual parte do processamento ocorre entre o cliente e o servidor, com o objetivo de melhorar o desempenho geral da aplicação e desonerar o servidor [133]. Utilizando um modelo baseado em dispositivos de IoT, essa abordagem foi denominada de *Fog Function* e é apropriada para cargas de trabalho que demandam uso intensivo de dados. Os autores elaboraram um mecanismo de orquestração orientado a contexto para acionar, configurar e otimizar dinâmica e automaticamente a implantação da Função *Fog* nos ambientes de nuvem e de borda.

4.6 Considerações Finais

Diante do exposto neste capítulo, nota-se claramente o potencial protagonismo que os serviços de FaaS podem exercer no cenário da computação em nuvem. Os benefícios da adoção desse modelo têm impulsionado o seu avanço, entretanto, ainda existem lacunas que dificultam a sua adesão como, por exemplo, a necessidade de conhecer as diversas API e SDKs dos provedores. Além disso, necessidades de ajustes em aplicações já desenvolvidas podem frear um processo de incorporação desse modelo, suprimindo suas vantagens.

Por essa razão, este trabalho propõe uma forma de mitigar essas dificuldades, reduzindo o espaço entre o desenvolvedor e os melhores resultados por meio do uso de serviços de FaaS de forma automatizada.

Assim sendo, acredita-se que o uso de um *framework* capaz de converter automaticamente aplicações monolíticas para trabalharem de maneira orientada a *Function as a Service*, de forma transparente e otimizada, é a alternativa mais apropriada para esse contexto. No próximo capítulo será detalhado o *framework* Node2FaaS, proposto n63,este trabalho, com o intuito de atender a essa demanda.

Capítulo 5

Framework Node2FaaS

Neste capítulo será apresentada a proposta deste trabalho, o *framework* Node2FaaS. O capítulo está dividido em dez seções, sendo a primeira seção (5.1) reservada para as considerações iniciais e a última seção (5.10) reservada para as considerações finais. Na Seção 5.2 é apresentada a linguagem de programação JavaScript, e na Seção 5.3 o NodeJS. As Seções 5.4 e 5.5 detalham as características da proposta deste trabalho tais como a arquitetura do Node2FaaS e a sua composição interna. Em seguida, na Seção 5.6, são apresentados os *blueprints* arquiteturais construídos para definir a integração entre o orquestrador utilizado como interface entre o Node2FaaS e os provedores de nuvem. A Seção 5.7 apresenta o fluxo de execução do *framework* Node2FaaS e em seguida a Seção 5.8 descreve o analisador de aderência ao paradigma do FaaS. Por último, na Seção 5.9, são analisados os trabalhos relacionados a esta proposta.

5.1 Considerações Iniciais

À medida em que a computação evolui, as arquiteturas de software precisam se ajustar às necessidades das aplicações em utilização no momento. Já foi o tempo em que máquinas gigantescas cuidavam de todo o processamento, e o software era executado majoritariamente nos grandes *mainframes*. Com a popularização dos computadores pessoais, o poder computacional disponível nessas máquinas passou a ser melhor utilizado e, então, o processamento do software foi gradativamente migrando para a execução no lado do cliente [134].

Nesse cenário, aplicações desenvolvidas usando a arquitetura monolítica possuem limitações arquiteturais que dificultam ganhos de escalabilidade. Uma forma de mitigar essa dificuldade é utilizar serviços de nuvem como FaaS [135] para processar segmentos do sistema. O FaaS, como apresentado no Capítulo 4 é um modelo de serviço no qual o cliente carrega um trecho de código, escrito em uma determinada linguagem, e o provedor

oferece o processamento desse código a partir de uma chamada do serviço, em geral via API REST [24]. Esse modelo encapsula toda a complexidade envolvida na infraestrutura.

Todavia, fazer uso do modelo de FaaS não é trivial, pois exige que o usuário conheça a forma de consumo que cada provedor oferece. Além disso, é necessário que o desenvolvedor construa as aplicações considerando o uso desse modelo, ou invista considerável tempo ajustando aplicações já desenvolvidas para trabalharem com FaaS.

Dessa forma, para utilizar os serviços de FaaS é necessário que o desenvolvedor conheça os detalhes das APIs de cada provedor, bem como saiba segmentar as funções das aplicações e convertê-las em chamadas adequadas à estrutura do serviço. Esse processo pode se tornar enfadonho, e desencorajar os desenvolvedores a adotarem uma abordagem baseada em FaaS. Com isso, muitos profissionais podem desistir dos benefícios que uma arquitetura baseada em nuvem pode oferecer, tais como: alta disponibilidade, resiliência, redução de custos, entre outros. Ademais, a linguagem de programação escolhida para consumir esse paradigma de computação pode ser determinante para o sucesso ou o fracasso dessa jornada.

Nesse contexto, é possível perceber que a linguagem JavaScript [136] tem ganhado muita importância para o sucesso de aplicações web. Atualmente, essa linguagem tem se tornado um padrão para inclusão de interatividade nas páginas da Internet, e diversos *frameworks* tem utilizado essa linguagem para incrementar o comportamento de aplicações no lado do cliente. Assim, o NodeJS [34] surgiu como uma alternativa para execução de códigos JavaScript no lado do servidor, e vem ganhando bastante espaço no mercado, estando presente, inclusive, em gigantes como PayPal, LinkedIn e NetFlix [34].

A larga adoção do JavaScript tem oferecido ao NodeJS um cenário propício para a sua adoção, já que a curva de aprendizado pode ser bastante atenuada pelo aproveitamento dos conhecimentos de JavaScript [34]. Ainda assim, a constante evolução dos modelos de desenvolvimento de software, bem como a adoção de novos paradigmas de computação, como a nuvem, exige que os processos de aprimoramentos das linguagens de programação mantenham um fluxo constante.

Diante do exposto, e considerando a necessidade de aprimoramentos no modelo de processamento de software em NodeJS, este trabalho propõe o *framework* Node2FaaS. Um *framework* para conversão automática e profícua de aplicações monolíticas, escritas em NodeJS, para aplicações orientadas a FaaS. Nas próximas seções serão abordados alguns aspectos que caracterizam a linguagem JavaScript e o NodeJS.

5.2 A Linguagem JavaScript

O JavaScript é a linguagem de programação da web [136]. A ampla maioria dos *sites* a utiliza e todos os navegadores modernos, computadores de mesa, *consoles* de jogos, *tablet* e *smartphones* incluem seus interpretadores, tornando-a a linguagem de programação mais onipresente da história [136]. A linguagem faz parte da tríade de tecnologias que todos os desenvolvedores web devem conhecer [136]: *HyperText Markup Language* (HTML), para especificar o conteúdo de páginas web; *Cascading Style Sheets* (CSS), para especificar a apresentação dessas páginas; e JavaScript, para definir o comportamento delas.

Ela é uma linguagem de alto nível, dinâmica, interpretada e não tipada, conveniente para estilos de programação orientados a objetos e funcionais [136]. A sua sintaxe é derivada da linguagem Java[1], das funções de primeira classe de Scheme[2], e da herança baseada em protótipos de Self[3]. O nome "JavaScript" é um pouco enganoso, pois, a não ser pela semelhança sintática superficial, JavaScript é completamente diferente de Java. Além disso, já deixou para trás suas raízes como linguagem de *script* há muito tempo, tornando-se uma linguagem de uso geral, robusta e eficiente.

Inicialmente, JavaScript era usada para pequenas validações e interações simples na web, hoje tornou-se um padrão e diversos *frameworks* passaram a fazer uso para incrementar a interatividade das aplicações. O *Institute of Electrical and Electronic Engineers* (IEEE) publicou um *ranking* [137] que lista as linguagens de programação mais utilizadas em 2019, e nessa lista o JavaScript ocupa a sexta posição. Um salto de duas posições em relação à avaliação anterior, na qual a linguagem figurava na oitava posição [138]. A proeminência dessa linguagem alçou a sua utilização, antes restrita ao lado do cliente, para o lado do servidor, culminando no surgimento do NodeJS, que será detalhado na próxima seção.

5.3 NodeJS

Devido ao sucesso alcançado pelo JavaScript, ainda em 2009, Rayan Dahl propôs o NodeJS [33]. Não se limitando a ser um *framework* JavaScript, e sem a pretensão de figurar como uma nova linguagem de programação, o NodeJS pode ser definido como um ambiente de execução de código JavaScript [33]. O principal desafio que o NodeJS se propõe a superar é a alta escalabilidade que as aplicações web demandam atualmente. Para isso, oferece suporte à execução assíncrona de processos, evitando que outros processos fiquem bloqueados aguardando a resposta de uma chamada [33].

[1] https://www.java.com
[2] https://www.scheme.com
[3] http://www.selflanguage.org

O NodeJS possui um modelo orientado a eventos que usa o acionamento de funções de retorno de chamada após a conclusão de uma tarefa ou geração de erro. Ele foi criado por trás da ideia de que outras linguagens dificultam a programação para execução simultânea [34]. O NodeJS foi construído desde o início com a finalidade de manuseio de *Input/Output* (I/O) de forma assíncrona, enquanto outras linguagens oferecem esse recurso fora dos seus núcleos de processamento, e assim acabam por executar de forma mais lenta que o NodeJS [34]. Alguns exemplos dessa abordagem são o *Event Machine* [139], para o Ruby; e o *Twisted* [140], para o Python.

O NodeJS leva para o processamento no lado do servidor, a linguagem de programação mais consolidada no processamento do lado do cliente, o JavaScript. Com isso, ele agrega ao seu conjunto de potenciais utilizadores uma gama de desenvolvedores web, sem a necessidade de superar uma nova curva de aprendizado, como ocorre no processo habitual de conhecimento de uma nova linguagem de programação. Além disso, ele conta com uma comunidade robusta de suporte, que oferece uma grande quantidade de bibliotecas para reaproveitamento de código por meio do *Node Package Manager* (NPM) [141].

O NPM é uma ferramenta de compartilhamento de bibliotecas que auxilia os desenvolvedores a buscarem por soluções para problemas já conhecidos [141]. A partir do NPM é possível facilmente baixar e instalar, no ambiente local, bibliotecas de software escritos em NodeJS por terceiros, e usar em uma aplicação [142]. Igualmente simples é o processo de publicar bibliotecas desenvolvidas localmente e disponibilizá-las para consumo público.

Com o simples comando *npm install express*, por exemplo, é instalada a biblioteca *express* [142], que atua como um *framework* muito popular por oferecer suporte para construção de APIs REST usando NodeJS. Porém, caso a necessidade seja por um *framework* para trabalhar tanto com APIs quanto com aplicações web, uma opção seria o Koa [143], também disponível por meio do NPM.

Atualmente, o NPM disponibiliza milhares de bibliotecas que permitem incrementar os trabalhos com NodeJS. Assim, a proposta deste trabalho inclui a publicação do *framework* Node2FaaS na plataforma NPM. As características desse *framework* serão detalhadas nas próximas seções.

5.4 Características do *Framework* Node2FaaS

O *framework* proposto neste trabalho, Node2FaaS, processa o código fonte original de uma aplicação NodeJS oferecida como insumo para sua execução, e a converte para uma aplicação cujas funções sejam executadas em um serviço de FaaS. O código interno das funções é convertido em *deploys* criados automaticamente no provedor. As chamadas à API do serviço de FaaS, que correspondem ao código original das funções, são colocadas

no lugar das definições originais das mesmas. O produto desse processamento é uma nova aplicação, baseada na original, cuja efetiva execução de suas funções não ocorre no ambiente no qual a aplicação estiver sendo executada, mas em um serviço externo de FaaS. Assim sendo, o *framework* conta com as seguintes características:

- **Facilidade de operação:** o processo de instalação do *framework* configura todo o aparato ferramental necessário para a sua utilização. O usuário deve configurar apenas as informações dos provedores e, ainda assim, o Node2FaaS também auxilia nessa tarefa, coletando as credenciais e criando os respectivos arquivos de configuração. Essa facilidade de adoção do *framework* visa encurtar a distância entre os usuários e os melhores resultados que o modelo proposto pode entregar;
- **Conversão automática:** o processo de conversão é completamente automático e não exige qualquer interação por parte do usuário. Uma vez apontada a aplicação alvo, o Node2FaaS simplesmente cumpre a sua missão, e no final entrega uma versão da aplicação original, cujas funções, caso sejam aderentes, são executadas em serviços de FaaS. Os detalhes do processo de conversão serão apresentados na Seção 5.7;
- **Análise das funções:** alguns tipos de algoritmo não logram vantagens sendo executados em serviços de FaaS. O Node2FaaS executa uma análise do código fonte interno de cada função, a fim de definir a melhor abordagem a ser adotada, seja transferindo a execução para um provedor de FaaS ou mantendo a execução local;
- **Execução otimizada:** a execução da aplicação convertida deve ser igual ou mais rápida do que a original. Para garantir isso, o *framework* mescla diferentes formas de tratamento das funções a fim de obter o melhor resultado em cada uma delas, otimizando a execução da aplicação de forma generalizada;
- **Flexibilidade:** eventualmente o desenvolvedor pode lidar com requisitos de negócio que exijam uma alteração no comportamento padrão do *framework*, tanto para evitar a publicação de alguma função em serviços de FaaS, quanto para garantir a sua publicação. Essa característica flexível do Node2FaaS agrega um conjunto maior de aplicações ao rol de candidatas a utilização do *framework*;
- **Múltiplos provedores:** considerando que o conceito de *sky computing* é uma abordagem em amplo crescimento, o Node2FaaS permite a publicação em diversos provedores através da utilização de um orquestrador *multicloud* integrado;
- **Aplicável em nível acadêmico e profissional:** o Node2FaaS consiste em uma solução tecnológica que pode ser efetivamente utilizada em processos de desenvolvi-

mento de aplicações NodeJS, servindo tanto para fins acadêmicos, quanto profissionais.

Durante a fase de inicialização do *framework* são verificados os pré-requisitos para o seu correto funcionamento, dentre eles a presença do Terraform no ambiente local. Na hipótese da máquina não possuir uma instalação do Terraform, o Node2FaaS fará a instalação da última versão disponível. Para viabilizar isso foi necessário desenvolver um pacote NPM denominado "terraform-latest". Esse pacote está disponível em https://www.npmjs.com/package/terraform-latest e pode ser instalado separadamente por meio do comando **npm install terraform-latest**.

Os detalhes da arquitetura do Node2FaaS serão descritos na próxima seção, e na Seção 5.7 será apresentado o fluxo de conversão de aplicações NodeJS para uma abordagem otimizada utilizando FaaS.

5.5 Arquitetura do *Framework* Node2FaaS

A arquitetura da solução proposta pelo Node2FaaS é inspirada no paradigma de Chamada de Procedimento Remoto (*Remote Process Call*, RPC), que é um paradigma útil para fornecer comunicação através de uma rede, entre programas escritos em linguagens de alto nível [144]. Apesar de seguir o modelo do RPC, para simplificar o tratamento o Node2FaaS adota uma abordagem de colocar tudo em uma mesma função, de modo a não considerar eventuais chamadas internas para outras funções usando o modelo de FaaS. A Figura 5.1 apresenta a solução proposta pelo *framework* na qual é possível observar que em cada módulo de uma aplicação podem existir funções. Dentro de cada função existe um código que executa alguma operação útil para o software. Uma vez submetido ao Node2FaaS, esse código será publicado em um provedor de nuvem e, em seu lugar, na aplicação convertida, aparecerá uma chamada URI apontando para a API REST fornecida pelo provedor como resultado da publicação. Em resumo, o código original das funções é transferido para o serviço de FaaS na nuvem e, então, consumido por meio de requisições utilizando o protocolo HTTP.

Para execução da sua missão, o Node2FaaS foi segmentado internamente em módulos que cumprem tarefas específicas e possuem integrações entre si. Assim, trabalhando de maneira coordenada, esses módulos recebem insumos e retornam resultados que viabilizam o processamento das aplicações, e geração de uma nova aplicação utilizando a abordagem proposta.

A Figura 5.2 apresenta a estrutura interna da versão 1.0 do *framework*, e nela é possível verificar a existência de um módulo principal, denominado *index*, cujo papel consiste em

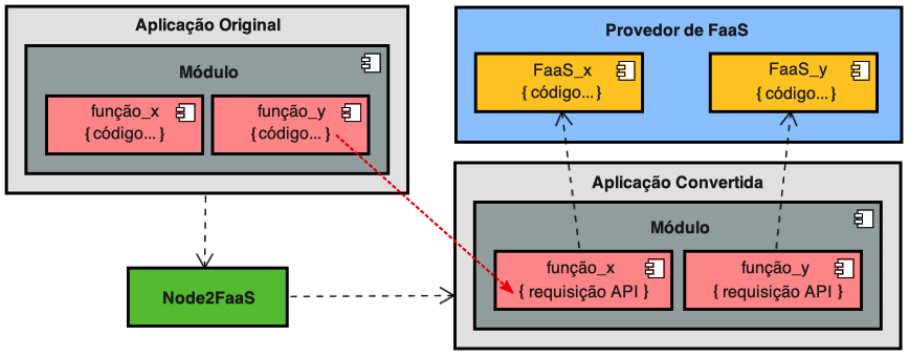

Figura 5.1: A arquitetura do *framework* Node2FaaS.

coordenar os demais módulos. Além do módulo principal, o Node2FaaS é composto pelos seguintes módulos:

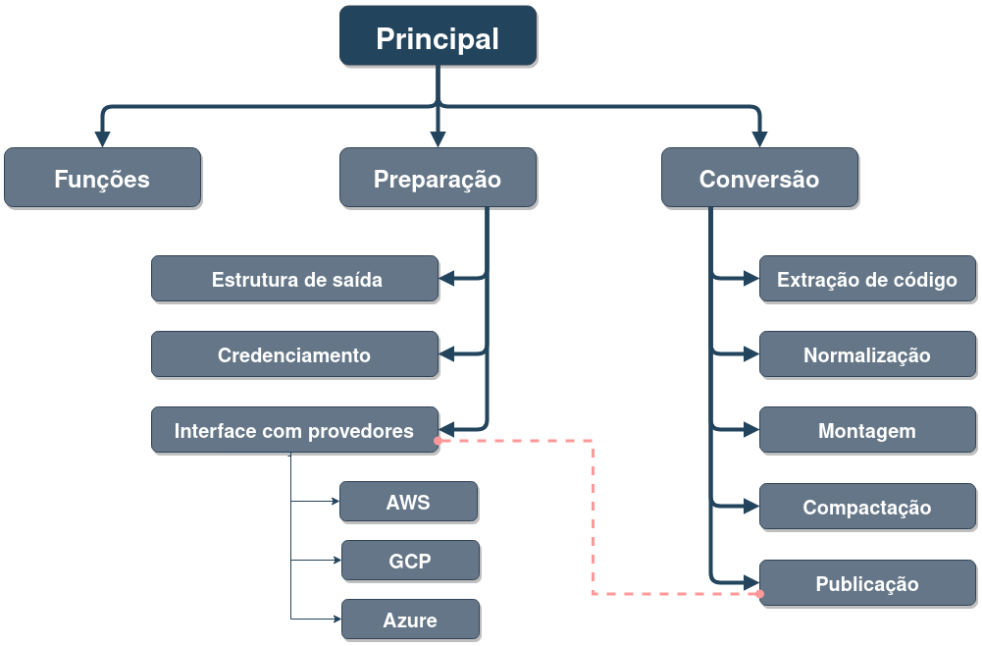

Figura 5.2: Composição do *framework* Node2FaaS.

- **Funções**: concentra um conjunto de utilitários que são de uso comum entre os demais módulos;
- **Preparação**: garante que os requisitos necessários para a execução adequada do *framework* estejam atendidos, e conta com o auxílio dos seguintes submódulos:
  - **Estrutura de saída**: responsável por criar o diretório da aplicação de destino;
  - **Credenciamento**: responsável por obter e armazenar as informações das credenciais nos provedores;

  - **Interface com provedor**: responsável por efetivar a comunicação com os serviços dos provedores e abstrair suas complexidades para os demais módulos. Ele possui uma segmentação interna para o tratamento das especificidades de cada provedor suportado.

- **Conversor**: coordena o processo de conversão e conta com o auxílio dos seguinte submódulos:

  - **Extração de código**: responsável por extrair o código fonte interno das funções;
  - **Normalização**: responsável por tornar o código fonte executável no serviço de FaaS;
  - **Montagem**: responsável pela montagem da nova definição das funções após a publicação no provedor;
  - **Compactação**: alguns serviços exigem que o código seja compactado antes da publicação, esse módulo é responsável por realizar essa tarefa;
  - **Publicação**: responsável por solicitar a publicação ao módulo de interface com o provedor e tratar o seu retorno.

5.6 *Blueprints* para o Orquestrador de Nuvem

Uma vez tendo sido escolhido o Terraform como orquestrador que faria a ponte entre o *framework* e os provedores (apresentado na Seção 3.4 do Capítulo 3), passou-se a construção dos artefatos de IaC que definem o modelo arquitetural (*blueprint*) das integrações do Terraform com cada provedor. Considerando o quadrante mágico do Gartner (Seção 2.5), que aponta os líderes de mercado, foram selecionados: AWS, GCP e Azure.

Os *blueprints* foram segmentados em três arquivos *.tf* da seguinte forma:

- **main.tf:** Contém a definição principal da arquitetura da ser provisionada no provedor, incluindo detalhes de rede, *firewall*, API *gateway* (quando aplicável) e o próprio serviço de *Function as a Service*;
- **output.tf:** Contém a definição das saídas que o *blueprint* deve gerar após a execução e que são utilizadas pelo *framework* no fluxo do seu processamento;
- **variables.tf:** Contém a definição das variáveis que cada *blueprint* recebe e utiliza na definição principal. Para isso, são utilizados diversos parâmetros na requisição que é montada pelo *framework* durante o seu processamento.

A Listagem 5.1 mostra a definição principal do *blueprint* do Terraform para o provedor AWS. Nessa definição é possível observar a presença do recurso que define o Lambda, a API Gateway e suas integrações, métodos, permissões e o *deployment*.

Listagem 5.1: *Blueprint* do Terraform para AWS.

```
provider "aws" {
  version = "~> 2.0"
  region = var.region
  access_key = var.access_key
  secret_key = var.secret_key
}

resource "aws_iam_role" "iam_for_lambda" {
  name = "iam_for_lambda-${var.region}-${var.name}"

  assume_role_policy = <<EOF
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": "sts:AssumeRole",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Effect": "Allow",
      "Sid": ""
    }
  ]
}
EOF
}

resource "aws_lambda_function" "function" {
  filename         = var.sourcecode_zip_path
  function_name    = "node2faas-${var.name}"
  description      = "Function ${var.name} automaticly created by node2faas"
  role             = "${aws_iam_role.iam_for_lambda.arn}"
  handler          = "${var.name}.${var.name}"
  source_code_hash = "${filebase64sha256("${var.sourcecode_zip_path}")}"
  runtime          = "nodejs10.x"
}

resource "aws_api_gateway_rest_api" "rest" {
  name        = "node2faas-rest-${var.name}"
  description = "REST API for function ${var.name} automaticly created by node2faas"
}

resource "aws_api_gateway_resource" "proxy" {
  rest_api_id = "${aws_api_gateway_rest_api.rest.id}"
  parent_id   = "${aws_api_gateway_rest_api.rest.root_resource_id}"
  path_part   = "{proxy+}"
}

resource "aws_api_gateway_method" "proxy" {
  rest_api_id   = "${aws_api_gateway_rest_api.rest.id}"
  resource_id   = "${aws_api_gateway_resource.proxy.id}"
  http_method   = "ANY"
  authorization = "NONE"
}

resource "aws_api_gateway_integration" "lambda" {
  rest_api_id             = "${aws_api_gateway_rest_api.rest.id}"
  resource_id             = "${aws_api_gateway_method.proxy.resource_id}"
  http_method             = "${aws_api_gateway_method.proxy.http_method}"
  integration_http_method = "POST"
  type                    = "AWS_PROXY"
  uri                     = "${aws_lambda_function.function.invoke_arn}"
}

resource "aws_api_gateway_method" "proxy_root" {
  rest_api_id   = "${aws_api_gateway_rest_api.rest.id}"
  resource_id   = "${aws_api_gateway_rest_api.rest.root_resource_id}"
  http_method   = "ANY"
```

```
  authorization = "NONE"
}

resource "aws_api_gateway_integration" "lambda_root" {
  rest_api_id             = "${aws_api_gateway_rest_api.rest.id}"
  resource_id             = "${aws_api_gateway_method.proxy_root.resource_id}"
  http_method             = "${aws_api_gateway_method.proxy_root.http_method}"
  integration_http_method = "POST"
  type                    = "AWS_PROXY"
  uri                     = "${aws_lambda_function.function.invoke_arn}"
}

resource "aws_api_gateway_deployment" "deploy" {
  depends_on = [
    "aws_api_gateway_integration.lambda",
    "aws_api_gateway_integration.lambda_root",
  ]

  rest_api_id = "${aws_api_gateway_rest_api.rest.id}"
  stage_name  = "${var.name}"
}

resource "aws_lambda_permission" "apigw" {
  statement_id  = "AllowAPIGatewayInvoke"
  action        = "lambda:InvokeFunction"
  function_name = "node2faas-${var.name}"
  principal     = "apigateway.amazonaws.com"
  source_arn    = "${aws_api_gateway_rest_api.rest.execution_arn}/*/*"
}
```

A Listagem 5.2 mostra a definição principal do *blueprint* do Terraform para o provedor GCP. É possível perceber que a definição do *blueprint* para o GCP é bastante diferente daquela feita para o provedor AWS. No GCP é necessário apenas definir um *bucket* para alocar o código fonte da função e os parâmetros básicos que identificam a função, além de uma permissão. Como pode ser notado, o *blueprint* do GCP é significativamente mais simples do que a definição feita para o provedor AWS, mostrando a simplicidade com que o provedor trata a interação com o seu serviço de FaaS.

Listagem 5.2: *Blueprint* do Terraform para GCP.

```
provider "google" {
  credentials = "${file("gcp.json")}"
  project     = var.project
  region      = var.region
}

resource "random_string" "bucket_name" {
 length = 3
 special = false
 upper = false
 lower = false
 number = true
}

resource "google_storage_bucket" "bucket" {
  name = "node2faas-${var.project}-bucket-${random_string.bucket_name.result}"
}

resource "google_storage_bucket_object" "archive" {
  name   = "index.zip-${var.project}-${var.name}"
  bucket = "${google_storage_bucket.bucket.name}"
  source = var.sourcecode_zip_path
}

resource "google_cloudfunctions_function" "function" {
  name                  = "node2faas-${var.project}-${var.name}"
  description           = "Automatic created by node2faas for process function -> ${var.name}"
  runtime               = "nodejs10"
```

```
  available_memory_mb   = var.memory
  source_archive_bucket = "${google_storage_bucket.bucket.name}"
  source_archive_object = "${google_storage_bucket_object.archive.name}"
  trigger_http          = true
  timeout               = 180
  entry_point           = var.name
  labels = {
    my-label = var.name
  }

  environment_variables = {
    CALL = var.name
  }
}

resource "google_cloudfunctions_function_iam_member" "invoker" {
  project = "${var.project}"
  region = "${var.region}"
  cloud_function = "${google_cloudfunctions_function.function.name}"

  role = "roles/cloudfunctions.invoker"
  member = "allUsers"
}
```

A Listagem 5.3 mostra a definição principal do *blueprint* do Terraform para o provedor Azure. Diferente do GCP que se mostrou mais simples do que o AWS, a definição para o provedor Azure é bem mais complexa do que aquela feita para o provedor AWS e, consequentemente, para o GCP. Neste *blueprint* são necessárias definições de vários outros recursos, em nível de detalhes mais alto, até que seja possível alcançar o mesmo objetivo. Isso mostra o grau elevado de complexidade na entrega do serviço oferecida pelo provedor Azure.

Listagem 5.3: *Blueprint* do Terraform para Azure.

```
provider "azurerm" {
  version = "=1.28.0"
  subscription_id = var.subscription_id
  client_id = var.client_id
  client_secret = var.client_secret
  tenant_id = var.tenant_id
}

resource "random_string" "rg_name" {
 length = 3
 special = false
 upper = false
 lower = false
 number = true
}

resource "azurerm_resource_group" "rg" {
 name = "node2faas${random_string.rg_name.result}"
 location = var.region
}

resource "azurerm_storage_account" "storage" {
 name = "node2faas${random_string.rg_name.result}"
 resource_group_name = "${azurerm_resource_group.rg.name}"
 location = "${azurerm_resource_group.rg.location}"
 account_tier = "Standard"
 account_replication_type = "LRS"
}

resource "azurerm_storage_container" "storage_container" {
 name = "node2faas${random_string.rg_name.result}"
 resource_group_name = "${azurerm_resource_group.rg.name}"
 storage_account_name = "${azurerm_storage_account.storage.name}"
 container_access_type = "blob"
}
```

```
resource "azurerm_storage_blob" "storage_blob" {
 name = "${var.name}.zip"
 resource_group_name = "${azurerm_resource_group.rg.name}"
 storage_account_name = "${azurerm_storage_account.storage.name}"
 storage_container_name = "${azurerm_storage_container.storage_container.name}"
 type = "block"
 source = var.sourcecode_zip_path
}

data "azurerm_storage_account_sas" "storage_sas" {
  connection_string = "${azurerm_storage_account.storage.primary_connection_string}"
  https_only        = true

  resource_types {
    service   = false
    container = false
    object    = true
  }

  services {
    blob  = true
    queue = false
    table = false
    file  = false
  }

  start  = "2019-09-01"
  expiry = "2029-09-01"

  permissions {
    read    = true
    write   = false
    delete  = false
    list    = false
    add     = false
    create  = false
    update  = false
    process = false
  }
}

resource "azurerm_app_service_plan" "plan" {
 name = "node2faas${random_string.rg_name.result}"
 location = "${azurerm_resource_group.rg.location}"
 resource_group_name = "${azurerm_resource_group.rg.name}"
 kind = "functionapp"
 sku {
 tier = "Dynamic"
 size = "Y1"
 }
 depends_on = [azurerm_resource_group.rg]
}

resource "azurerm_application_insights" "insights" {
  name                = "node2faas${random_string.rg_name.result}"
  location            = "${azurerm_resource_group.rg.location}"
  resource_group_name = "${azurerm_resource_group.rg.name}"
  application_type    = "Web"
  depends_on = [azurerm_resource_group.rg]
}

resource "azurerm_function_app" "function" {
  name = "node2faas${random_string.rg_name.result}"
  location = "${azurerm_resource_group.rg.location}"
  resource_group_name = "${azurerm_resource_group.rg.name}"
  app_service_plan_id = "${azurerm_app_service_plan.plan.id}"
  storage_connection_string = "${azurerm_storage_account.storage.primary_connection_string}"
  version = "~2"

  app_settings = {
    "APPINSIGHTS_INSTRUMENTATIONKEY" = "${azurerm_application_insights.insights.instrumentation_key}"
    "FUNCTION_APP_EDIT_MODE" = "readonly"
    "https_only" = true
    "functionTimeout" = "00:30:00"
    "HASH" = "${filebase64sha256("${var.sourcecode_zip_path}")}"
    "WEBSITE_NODE_DEFAULT_VERSION" = "10.14.1"
```

```
    "WEBSITE_USE_ZIP" = "https://${azurerm_storage_account.storage.name}.blob.core.windows.net/${azurerm_\
       storage_container.storage_container.name}/${azurerm_storage_blob.storage_blob.name}\
       ${data.azurerm_storage_account_sas.storage_sas.sas}"
    "WEBSITE_RUN_FROM_PACKAGE" = "https://${azurerm_storage_account.storage.name}.blob.core.windows.net\
       /${azurerm_storage_container.storage_container.name}/${azurerm_storage_blob.storage_blob.name}\
       ${data.azurerm_storage_account_sas.storage_sas.sas}"
  }
  depends_on = [azurerm_resource_group.rg]
}

resource "azurerm_template_deployment" "function_keys" {
  name = "node2faas${random_string.rg_name.result}"
  parameters = {
    "functionApp" = "${azurerm_function_app.function.name}"
  }
  resource_group_name     = "${azurerm_resource_group.rg.name}"
  deployment_mode = "Incremental"

  template_body = <<BODY
  {
      "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
      "contentVersion": "1.0.0.0",
      "parameters": {
          "functionApp": {"type": "string", "defaultValue": ""}
      },
      "variables": {
          "functionAppId": "[resourceId('Microsoft.Web/sites', parameters('functionApp'))]"
      },
      "resources": [
      ],
      "outputs": {
          "functionkey": {
              "type": "string",
              "value": "[listkeys(concat(variables('functionAppId'),
                       '/host/default'), '2018-11-01').functionKeys.default]"}
      }
  }
  BODY
}
```

Apesar da necessidade de construir *blueprints* para cada provedor e com isso adicionar uma nova camada à solução do *framework*, essa decisão se justifica pelo fato de que, caso fosse construída uma integração utilizando a API nativa de cada provedor, seria necessário desvendar as especificidades de cada uma delas. Além disso, este trabalho poderia ser altamente instável, uma vez que alterações de versão nas APIs poderiam destruir toda a integração. Todavia, embora seja fato que isso ainda possa ocorrer, a diferença é que agora conta-se com um orquestrador sustentado pela comunidade. Assim sendo, caso isso ocorra é esperado que seja lançada, com brevidade, uma nova versão do orquestrador corrigindo o problema.

5.7 Fluxo de Execução do *Framework*

Uma vez disponível no ambiente local (após instalação), o fluxo do Node2FaaS é iniciado a partir da execução da aplicação "node2faas". Caso o *framework* tenha sido instalado via NPM, essa aplicação estará registrada no *path* da máquina e poderá ser executada diretamente a partir do comando "*node2faas –target [caminho para a aplicação a ser convertida]*". Caso contrário, será necessário acessar o diretório onde o *framework* foi baixado, conceder permissão de execução, para somente então, executar o Node2FaaS.

Caso o usuário não conheça as opções da ferramenta poderá informar o parâmetro *–help* (ou *-h*) e visualizará a tela mostrada na Figura 5.3.

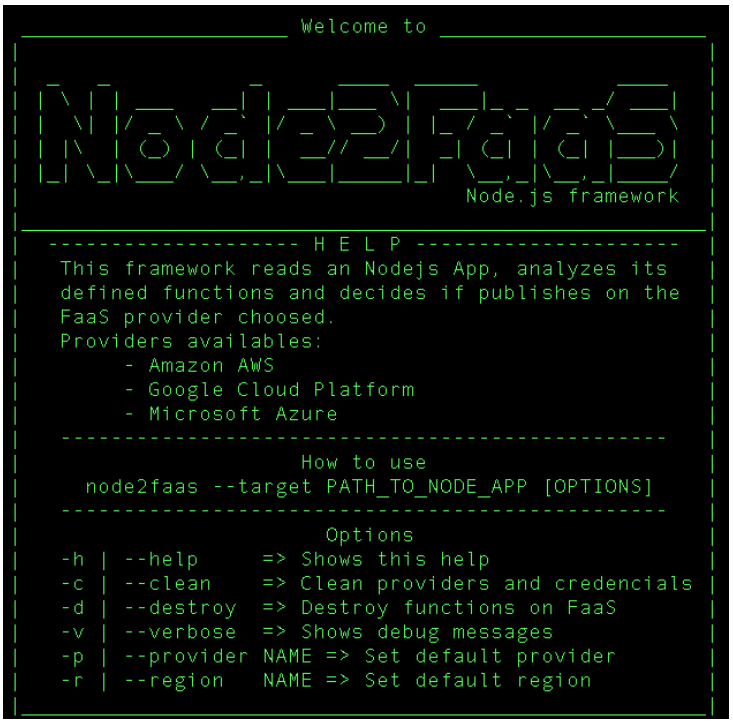

Figura 5.3: Tela de ajuda do *Framework* Node2FaaS.

Assim sendo, o *framework* Node2FaaS oferece as seguintes funcionalidades por meio da sua CLI:

- **help:** Exibição das opções da ferramenta;
- **clean:** Remove as informações locais relacionadas ao provedor, bem como as credenciais. Assim, será necessário informar novamente tais informações em uma nova execução do *framework*;
- **destroy:** Acessa o provedor e promove a destruição das funções que tenham sido criadas;
- **verbose:** Exibe detalhadamente o passo a passo do processamento do *framework*;
- **provider:** Permite alterar o provedor padrão atual;
- **region:** Permite alterar a região padrão do provedor atual.

Diante do exposto, possuir uma conta ativa em um provedor que se deseja executar a função é essencial, pois inicialmente o *framework* busca por credenciais de acesso à nuvem. Caso não encontre o arquivo de credenciais, a aplicação solicita que o usuário forneça as informações de credenciais para acesso aos serviços da nuvem. Após isso, o sistema criará

o arquivo de credenciais e não mais solicitará esse preenchimento em execuções futuras, a menos que o usuário forneça um parâmetro explicitando sua intenção.

Uma vez obtida a credencial e informado pelo usuário o caminho de uma aplicação para conversão pelo Node2FaaS, esta é submetida a um processo de conversão que analisará o código da aplicação buscando por definições de funções para realizar a conversão, conforme apresentado na Figura 5.4.

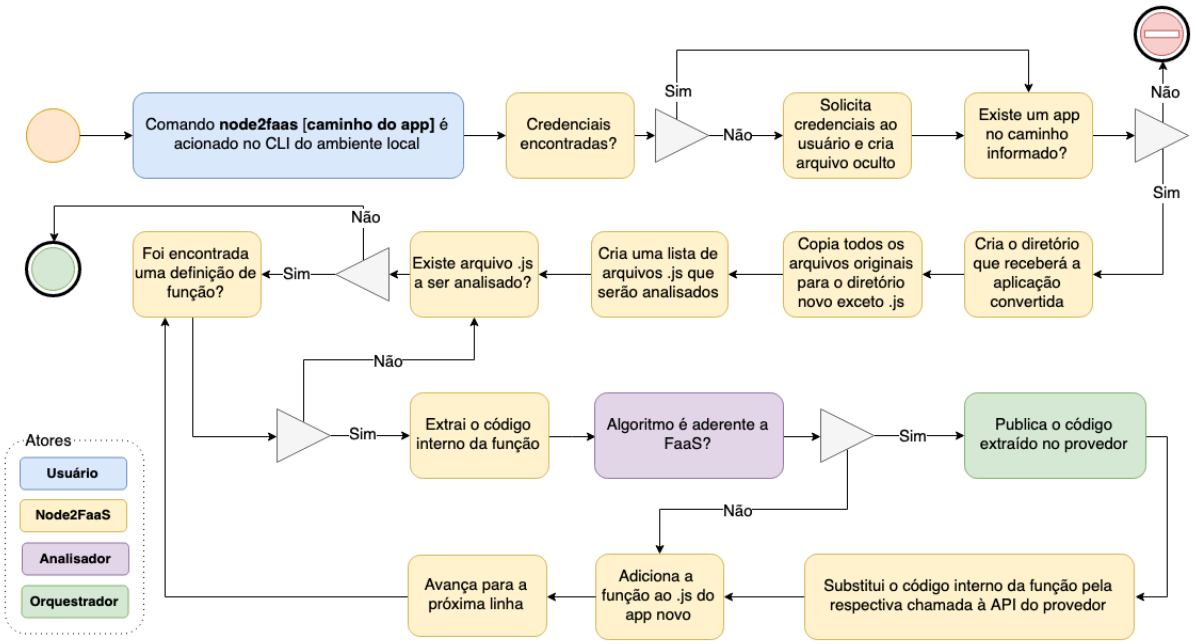

Figura 5.4: Processo de conversão de aplicações do Node2FaaS.

Durante o processo, caso seja encontrado um comando de inclusão de arquivo (*include*), então o arquivo alvo também é vasculhado em busca de funções candidatas à conversão, e esse processo se repete recursivamente, até que nenhum arquivo para inclusão seja encontrado.

Dessa maneira, quando a aplicação encontra uma função, ela verifica se a mesma está elegível para submissão à nuvem. Em caso positivo, é feita uma preparação no código da função de modo a normalizá-la ao funcionamento da respectiva nuvem e, em seguida, o código é entregue ao orquestrador. O orquestrador, por sua vez, efetiva o acesso à nuvem para criação de uma nova função FaaS. Depois de receber a confirmação da criação da função, a aplicação obtém a URI de acesso ao serviço, e cria a requisição dentro da definição da função original. Dessa forma, a chamada à função permanece inalterada e o seu funcionamento na plataforma de nuvem é feito de maneira totalmente transparente.

No final, o Node2FaaS terá gerado todos os arquivos que deveriam compor a aplicação inicial, porém, com o código original das funções substituído por chamadas HTTP ao serviço FaaS do provedor de nuvem. A aplicação convertida mantém a mesma assinatura

das funções originais, permitindo que o seu uso se mantenha inalterado para os processos requisitantes das funções. Isso elimina a necessidade de realizar ajustes na aplicação.

5.8 Analisador de Aderência ao FaaS

O processamento de funções em serviços de FaaS inevitavelmente penaliza o tempo de execução da aplicação, uma vez que existe a necessidade de atravessar a Internet para solicitar a execução e obter o resultado. Para que esse ônus seja compensado, a função a ser executada precisa ser suficientemente onerosa a ponto de ser mais vantajoso encaminhar a execução para o provedor e usufruir de um intenso paralelismo, do que manter a execução local e impactar o tempo de execução por conta do eventual enfileiramento que pode ocorrer em caso alto número de requisições concorrentes.

Assim sendo, neste trabalho foi incluída, no processo de conversão do *framework*, uma fase de verificação da aderência das funções ao paradigma de FaaS. Para isso, são realizadas duas verificações sobre o código. Primeiramente é calculada a quantidade de caracteres presentes no respectivo trecho de código da função, e caso essa quantidade ultrapasse 2220 caracteres a função será classificada como aderente ao FaaS. Esse número foi definido após a realização de diversos testes utilizando funções reais encontradas em projetos no GitHub. A segunda verificação busca pela utilização de parâmetros dentro de laços. Nesse caso, a função também será classificada como aderente, visto que a carga da função será afetada pelo valor informado em algum parâmetro e, potencialmente, seu tempo de execução será aumentado. Caso a função não se enquadre em nenhuma das verificações, então será mantida executando localmente.

Apesar de simples, essa fase de verificação promove uma filtragem importante no processo de conversão, evitando que funções muito simples, cujo tempo de execução é baixo, tenham sua execução impactada pela necessidade de atravessar a Internet para obter o seu resultado. Entretanto, essa fase pode ser aprimorada para verificar outros aspectos da função, tais como complexidade do algoritmo, comportamento do tempo de execução em simulações utilizando valores hipotéticos para os parâmetros, entre outros aspectos. Além disso, seria apropriado equalizar o processo de análise às limitações do provedor de destino para o tempo de execução, tamanho da requisição, quantidade de memória requerida, etc. Essas outras possibilidades serão incluídas como trabalhos futuros desta dissertação.

5.9 Trabalhos Relacionados

O levantamento bibliográfico realizado sobre a utilização de FaaS com o propósito de tornar o processamento de aplicações executáveis sobre esse paradigma mostrou que essa é uma área ainda pouco explorada. Existem alguns trabalhos que propõem uma abordagem semelhante à proposta deste trabalho, porém para outras linguagens de programação como Python e Java, tais como o Lambada [37] e o Podilizer [145]. Existem plataformas, tais como Serverless [32] e Vercel (anteriormente conhecida como Zeit) [146], que promovem o gerenciamento dos *deployments* de código em serviços de FaaS, neste caso atuando como uma camada de abstração acima dos provedores. Claudia.js [147], Zappa [148] e PyWren [149] integram uma classe de ferramentas que trata os *deployments* em serviços de FaaS sem o compromisso de transformar um código já escrito em algo executável sobre FaaS, mas apenas auxilia no desenvolvimento de aplicações cuja abordagem seja nativamente orientada a FaaS.

O trabalho [37] traz uma abordagem de conversão de aplicações escritas em Python para *deployments* Python no serviço AWS Lambda. A aplicação construída por Spillner, denominada Lambada, processa uma aplicação em Python e o converte para o código apropriado para ser instanciado na nuvem. Caso o usuário tenha o cliente da AWS instalado e devidamente configurado na máquina, o Lambada [37] executa o *deploy* automaticamente, porém todo o processo de configuração do cliente fica por conta do usuário. Isso limita o uso deste conversor a usuários que sejam capazes de configurar corretamente o cliente do provedor em seus ambientes locais.

Além disso, o referido artigo [37] limita-se a utilizar o Lambada para conversão de aplicações com função única. Em aplicações de ambiente produtivo é comum a composição de múltiplas funções para a execução de uma aplicação. O Node2FaaS, proposto neste trabalho, por outro lado, possibilita uma melhor utilização em aplicações reais, não ficando limitado a ambientes experimentais, como o Lambada.

Spillner e Dorodko [145] aplicam a mesma abordagem adotada no Lambada, porém para aplicações desenvolvidas em Java. Nesse trabalho os autores questionam a viabilidade econômica de executar uma aplicação Java inteiramente usando FaaS, e se há a possibilidade de automatizar o processo de conversão da aplicação. Eles implementaram uma ferramenta denominada Podilizer, e realizaram experimentos utilizando-a. Os resultados foram classificados como promissores, porém, apenas para fins acadêmicos, não apresentando capacidade de aplicação efetiva, uma vez que os autores encontraram dificuldades em utilizar as aplicações resultantes da conversão. Por outro lado, o Node2FaaS não se destina simplesmente a experimentação acadêmica, mas à efetiva utilização por desenvolvedores NodeJS.

O *framework* Serverless [32] oferece uma plataforma para execução de *deploys* de aplicações nos principais provedores de serviços de FaaS, tais como: AWS Lambda, Azure Functions, Google Cloud Functions, IBM OpenWhisk, além de *frameworks* como *Fn*. Ele abstrai as diferenças entre os serviços e permite a execução dos *deploys* nos serviços de FaaS através de uma CLI.

Escrito em NodeJS, o Serverless [32] precisa ser instalado diretamente no ambiente local e possui modelos para interação com cada provedor que precisam ser customizado de acordo com a necessidade do usuário. É uma ferramenta de código aberto, entretanto, dispõe de uma versão *enterprise* paga que oferece diversos recursos adicionais, tais como: *dashboard* via web, ferramentas para depuração de erros, injeção de boas práticas, simplicidade na segurança, entre outros.

Todavia, embora o Serverless [32] seja uma plataforma robusta para gerenciamento de *deploys* em FaaS, ele trabalha com as diversas linguagens de programação que cada provedor dispõe, e com isso não há um foco específico em NodeJS. Dessa forma, o *framework* não executa qualquer avaliação a respeito da natureza do código e sua aderência à abordagem de *Function as a Service*, deixando com o desenvolvedor a responsabilidade de decidir quais funcionalidade devem ser alçadas aos serviços de FaaS.

Vercel [146] é uma rede de implantação global construída sobre todos os provedores de nuvem existentes. Isso torna as equipes produtivas, removendo servidores e configurações, proporcionando uma experiência de desenvolvimento contínua para criar aplicativos da web escaláveis e modernos [146]. Na prática esse serviço atua como uma camada acima dos provedores de FaaS. Ela permite a criação de aplicações web através da sua plataforma web ou integração com ferramentas de versionamento de código como o GitHub[4].

A plataforma Vercel (anteriormente denominada Zeit) [146] possui um modelo de cobrança semelhante aos provedores que sobrepõe. Ela oferece uma cota diária para execuções e alocação de recursos que permite ao usuário experimentar o serviço gratuitamente. Além disso, ele dispõe de interface CLI e cliente Desktop para Windows e MacOS. Ela suporta uma gama variada de linguagens de programação e conta com diversos *plugins* que permitem integração com outras ferramentas de desenvolvimento.

Assim como o Serverless [32], a Vercel [146], se limita a receber o código e repassá-lo ao provedor, atuando como uma intermediária entre o desenvolvedor e o serviço de FaaS. Como ela trabalha com diversas linguagens, não há um foco específico em NodeJS, tampouco uma avaliação da aderência do algoritmo construído pelo desenvolvedor ao paradigma sem servidor implementado pelos serviços de FaaS.

Claudia.js [147] é um *framework* que facilita a implantação de projetos NodeJS no AWS Lambda. Ele automatiza todas as tarefas de implantação e configuração propensas

[4]https://github.com.

a erros. Esse *framework* atua como uma camada anterior ao AWS Lambda, cuja função se resume a simplificar a interação junto ao serviço de FaaS da Amazon. Por interagir apenas com um provedor, o *framework* Claudia.js limita as possibilidades de utilização do modelo de nuvem, fortalecendo o *vendor lock-in* e assim, não representando ganhos significativos para os seus utilizadores.

O Zappa [148] é uma ferramenta de linha de comando que converte aplicativos Flask[5] [150] e Django[6] [151] para execução em AWS Lambda. É um *framework* que empacota e implanta aplicativos Python compatíveis com *Web Server Gateway Interface* (WSGI) em uma função do AWS Lambda e no AWS API *gateway*. O Zappa é definido pelo autor como um "botão fácil" para a implantação *serverless* de Flask e Django e também permite criar aplicativos híbridos orientados a eventos que podem ser escalados para trilhões de eventos por ano sem nenhum esforço adicional. Apesar disso, o Zappa não conta com um processo de análise da aderência das aplicações ao paradigma de FaaS e com isso pode, eventualmente, degradar o desempenho de um sistema ao aplicar seu modelo. Ademais, de forma semelhante ao Claudia.js, o Zappa promove as aplicações apenas para um único provedor (o AWS), o que também reforça o *vendor lock-in*.

O PyWren [149] expõe um mapa primitivo contínuo do Python sobre o AWS Lambda. Embora o AWS Lambda tenha sido projetado para executar microsserviços controlados por eventos em escala, extraindo dinamicamente o código do S3 (serviço de armazenamento do AWS), ele faz com que cada chamada do AWS Lambda execute uma função diferente. O PyWren serializa uma função Python, capturando todas as informações relevantes, bem como a maioria dos módulos que não estão presentes no tempo de execução do servidor. Isso elimina a maioria dos esforços gerais do usuário sobre implantação, empacotamento e versionamento de código. Essa ferramenta envia a função serializada junto com cada dado serializado, colocando-os em chaves globalmente exclusivas no S3 e, em seguida, chama uma função Lambda comum. No lado do servidor, ele aplica cada função sobre os dados, ambos extraídos do S3. O resultado da chamada de função é serializado e colocado de volta no S3 em uma chave pré-especificada, e a conclusão do trabalho é sinalizada pela existência dessa chave. O PyWren, tal qual o Claudia.js e o Zappa, não oferece qualquer processo de análise (ainda que simples) sobre a aderência daquilo que está sendo encaminhado para processamento no provedor com o paradigma de FaaS e também restringe sua utilização à um único provedor (o AWS), o que reforça o *vendor*

[5]Flask é um *framework* de aplicativo da *web* leve. Ele foi projetado para facilitar e acelerar o início de projetos, com a capacidade de expansão para aplicativos mais complexos. Começou como um invólucro simples e se tornou um dos *framework* de aplicativos *web* Python mais populares [148].

[6]O Django é um *framework* Python de alto nível que incentiva o desenvolvimento rápido e o design limpo e pragmático. Construído por desenvolvedores experientes, ele cuida de grande parte do aborrecimento do desenvolvimento na *web*, para que o desenvolvedor possa se concentrar em escrever seu aplicativo sem precisar reinventar a roda. É de código aberto e gratuito [151].

Tabela 5.1: Quadro comparativo de soluções orientadas a FaaS.

Característica	Lambada [37]	Podilizer [145]	Serverless [32]	Vercel [146]	Claudia.js [147]	Zappa [148]	PyWren [149]	Node2FaaS
Linguagens suportadas	Python	Java	Múltiplas	Múltiplas	NodeJS	Python	Python	**NodeJS**
Aplicabilidade efetiva	Acadêmica	Acadêmica	Uso Geral	Uso Geral	Uso Geral	Uso Geral	Uso Geral	**Uso Geral**
Suporte multicloud	Sim	Não	Sim	Sim	Não	Não	Não	**Sim**
Orquestrador multicloud	Não	Não	ND	ND	Não	Não	Não	**Sim**
Análise de aderência a FaaS	Não	Não	Não	Não	Não	Não	Não	**Sim**

lock-in, como já foi mencionado.

A Tabela 5.1 apresenta um comparativo entre as soluções apresentadas. Nessa tabela é possível observar que duas dessas soluções não detém aplicabilidade abrangente, apenas acadêmica, e desse modo não oferecem, efetivamente, valor agregado por meio do modelo de FaaS. Entre as demais soluções, apenas o Node2FaaS oferece execução de avaliação para classificação da aderência de um código à abordagem de FaaS. A utilização de um orquestrador *multicloud* também é um diferencial do Node2FaaS. Assim, o *framework* acompanha a evolução do orquestrador e à medida que são incluídas novas integrações com outros serviços e/ou outros provedores, o Node2FaaS também é afetado por esses aprimoramentos.

Conforme já foi mencionado na Seção 6.1, essas ferramentas integram a plataforma www.faasification.com, que foi o resultado de um trabalho conjunto entre a SPLAB, da Suiça, e a UnB. Essa plataforma visa concentrar as iniciativas da comunidade relacionadas a FaaS, em especial aquelas cuja abordagem de assemelha à proposta deste trabalho.

5.10 Considerações Finais

Neste capítulo foi apresentado o *framework* Node2FaaS que converte aplicações escritas em NodeJS em aplicações que executem suas funções utilizando RPC em serviços de FaaS. Vale ressaltar que o resultado dos esforços envidados na construção do *framework* Node2FaaS estão disponíveis no GitHub no endereço https://github.com/node2faas.

Nesse contexto, o Node2FaaS se apresenta como uma opção mais vantajosa do que as soluções encontradas atualmente, tais como: Lambada [37], Podilizer [145], Serverless [32], Vercel [146], Claudia.js [147], Zappa [148] e PyWren [149]. Contudo, para comprovar os eventuais ganhos dessa abordagem foram realizados experimentos utilizando o *framework* Node2FaaS, cujos resultados são detalhados no próximo capítulo.

Capítulo 6

Resultados

Neste capítulo serão apresentados os resultados obtidos a partir deste trabalho, o *framework* Node2FaaS. O capítulo está dividido em quatro seções, sendo a primeira Seção (6.1) reservada para as considerações iniciais. Na Seção 6.2 são apresentados os resultados obtidos a partir de um experimento preliminar realizado sem a adoção de um orquestrador de nuvem. Em seguida, na Seção 6.3 são trazidos os resultados conclusivos, obtidos a partir de um experimento realizado após a integração do orquestrador Terraform. Por último, a Seção 6.4 finaliza com as considerações finais sobre os resultados.

6.1 Considerações Iniciais

A fim de explorar o potencial de uma aplicação NodeJS convertida para FaaS, usando o *framework* proposto neste trabalho, foram conduzidos alguns experimentos. O objetivo desses experimentos é avaliar o comportamento de uma aplicação cujo processamento esteja sendo encaminhado para um provedor de nuvem, conforme define a solução proposta pelo Node2FaaS. Para isso, foram monitorados os principais aspectos de um ambiente, cito: CPU, memória, tráfego de rede, leitura e escrita em disco, além do tempo médio de execução.

O primeiro experimento foi realizado sem a utilização de um orquestrador e somente sobre um provedor, enquanto o segundo, mais completo, contou com o orquestrador Terraform integrado ao Node2FaaS e, com isso, foi possível realizar os testes sobre três provedores de FaaS.

6.2 Experimento sem Orquestrador

Para o experimento preliminar foram desenvolvidas quatro funções como casos de teste que fazem parte de uma mesma aplicação. As funções estão descritas a seguir:

- **Simples:** Executa um operação simples de soma, conforme mostrado na Listagem 6.1;
- **Estresse sobre CPU:** Explora a capacidade de processamento do servidor, estressando recursos de CPU, conforme mostrado na Listagem 6.2;
- **Estresse sobre Memória:** Explora a capacidade de processamento do servidor, estressando recursos de memória, por meio de sucessivos laços com preenchimento de vetores, conforme mostrado na Listagem 6.3;
- **Estresse sobre Disco:** Explora a capacidade de processamento do servidor, estressando recursos de leitura e escrita em disco, por meio de sucessivos laços com criação e remoção de arquivos, conforme mostrado na Listagem 6.4.

A concepção do Node2FaaS não define qual provedor deve ser utilizado, mas para o experimento foi utilizado o serviço de FaaS da Amazon AWS Lambda. Afinal, a Amazon foi classificada em 2018 como líder em solução de nuvem IaaS no consagrado quadrante mágico do Gartner [152], como apresentado no Capítulo 2.

A aplicação contendo os casos de teste foi hospedada em uma instância *t2.micro* do serviço *Elastic Compute Cloud* (EC2), da AWS. Esse tipo de máquina conta com 1 gigabyte de memória, 1 CPU e utiliza o sistema operacional *Red Hat* 7.6.

A aplicação original foi submetida ao Node2FaaS para conversão e, então, a aplicação resultante (convertida) foi hospedada em uma segunda instância *t2.micro* da AWS. Dessa forma, ficaram duas aplicações em execução no provedor. A primeira realizando o processamento diretamente na instância EC2, enquanto a segunda utilizava o AWS Lambda para processamento, conforme a arquitetura mostrada na Figura 6.1.

Listagem 6.1: Função para caso de teste simples no 1º experimento.

```
exports.simple_load = function(req, res) {
  var a = 20;
  var b = 10;
  var c = a + b;
  res.json(c);
};
```

Listagem 6.2: Função para caso de teste estressando a CPU para o 1º experimento.

```
exports.cpu_load = function(req, res) {
  var c = 0;
  var arr = new Array();
  for(var i = 0; i < 99;i++){
    for(var z = 0; z < 99;z++){
        for(var y = 0; y < 99;y++){
          arr.push(i+z+y);
          c += 1;
      }
    }
  }
  res.json(c);
};
```

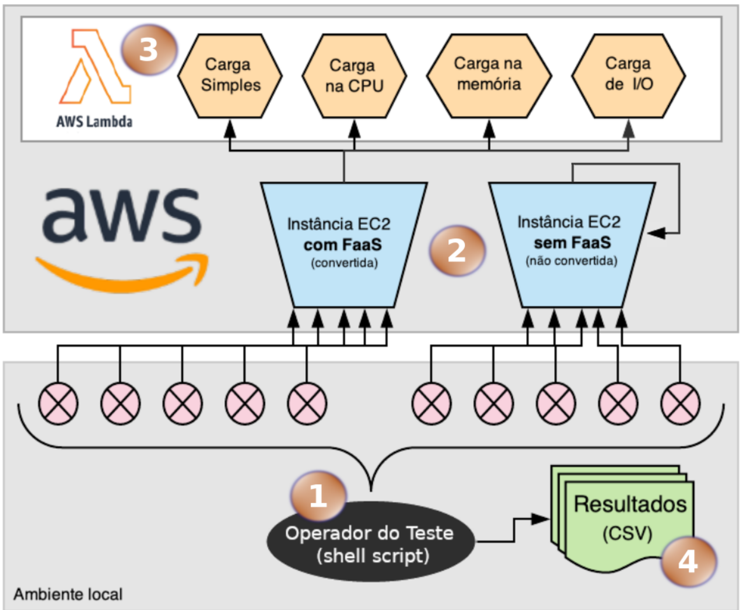

Figura 6.1: Arquitetura do experimento preliminar.

Listagem 6.3: Função para caso de teste estressando a memória para o 1º experimento.

```
exports.mem_load = function(req, res) {
  var c = 0;
  var arr = new Array();
  for(var i = 0; i < 99;i++){
    var string = 'ABCDABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZ\
    ABCDEFGHIJKLMNOPQRSTUVXZABCDEFGABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKL\
    MNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZABCDEFGABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZ\
    ABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZABCDEFGABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKL\
    MNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZABCDEFGABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZ\
    ABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZABCDEFGABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKL\
    MNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZABCDEFGABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZ\
    ABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZABCDEFGABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKL\
    MNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZABCDEFGABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZ\
    ABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZABCDEFGABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKL\
    MNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZABCDEFGABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZ\
    ABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZABCDEFGABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKL\
    MNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZABCDEFGABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZ\
    ABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZABCDEFGABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKL\
    MNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZABCDEFGABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZ\
    ABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZABCDEFGABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKL\
    MNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZABCDEFGABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZ\
    ABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZABCDEFGABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKL\
    MNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZABCDEFGABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZ\
    ABCDEFGHIJKMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZ';
    for(var z = 0; z < 50;z++){
        string += 'ABCDEFGHIJKLMNOPQRSTUVXABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZ\
        ABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZABCDEFGABCDEFGHIJKLMNOPQRSTUVXZABCD\
        EFGHIJKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZABCDEFGABCDEFGHI\
        JKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTU\
        VXZABCDEFGABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZAB\
        CDEFGHIJKLMNOPQRSTUVXZABCDEFGABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZABCDEFG\
        HIJKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZABCDEFGABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKL\
        MNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZABCDEFGABCDEFGHIJKLMNOPQ\
        RSTUVXZABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZABCDE\
        FGABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJ\
        KLMNOPQRSTUVXZABCDEFGABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKLMNO\
        PQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZABCDEFGABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRST\
        UVXZABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZABCDEFGZABCDEFGHIJKLMNOPQRSTUVXZ\
        ABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZ';
        for(var y = 0; y < 50;y++){
          string += 'ABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRABCDEFGHIJKLMNOPQRSTUVXZABCDEF\
          GHIJKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZABCDEFGABCDEFGHIJ\
          KLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTU\
```

```
            VXZABCDEFGABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZA\
            BCDEFGHIJKLMNOPQRSTUVXZABCDEFGABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZABCDE\
            FGHIJKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZABCDEFGABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHI\
            JKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZABCDEFGABCDEFGHIJKLM\
            NOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZ\
            ABCDEFGABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZABCD\
            EFGHIJKLMNOPQRSTUVXZABCDEFGSTUVXZABCDEFGHIJKLMNOPQRSTUVXZABCDEFGHIJKLMNOPQRSTUVXZAB\
            CDEFGHIJKLMNOPQRSTUVXZ';
            arr.push(string);
            c += 1;
        }
      }
    }
    res.json(c);
};
```

Listagem 6.4: Função para caso de teste estressando I/O para o 1º experimento.

```
exports.io_load = function(req, res) {
  var c = 0;
  var arr = new Array();
  for(var i = 0; i < 9;i++){
    for(var z = 0; z < 9;z++){
        for(var y = 0; y < 99;y++){
          var fs = require('fs');
          fs.writeFileSync('/tmp/fs.tmp'+i+z+y, i+z+y);
          c += 1;
      }
    }
  }
  res.json(c);
};
```

Para o experimento foi desenvolvido um *script* em *shell* a fim de executar requisições simultâneas para cada caso de teste, e coletar os resultados, inclusive, o tempo de execução. Os resultados do primeiro caso de teste, que executou uma simples função aritmética, são apresentados na Figura 6.2. Nessa figura é possível notar que para a maioria das requisições, a aplicação executando o processamento local obteve melhores resultados ao ser considerado o tempo de execução. Enquanto a média do tempo de execução das requisições para a aplicação sem FaaS foi de 0,46 segundos, a média da aplicação com FaaS foi de 1,45 segundos, conforme pode ser visto na Tabela 6.1. Isso representa uma diferença de 215%. Assim, fica claro que para algoritmos simples, a adoção de FaaS por meio da abordagem proposta, não representa ganho no tempo de execução, pois, sendo o processamento tão rápido, o *overhead* gerado pela passagem através da rede não é compensado pelo tempo de execução do FaaS.

O segundo caso de teste, para sobrecarga na CPU, tem seu resultado mostrado na Figura 6.3. Nela é possível observar que a aplicação convertida manteve estabilidade no tempo de execução, variando entre 1,12 segundos e 2,61 segundos, enquanto a outra aplicação apresentou continua degradação, iniciando em 0,30 segundos e finalizando a 2,87 segundos, conforme apresentado na Tabela 6.1. Isso demonstra que em relação ao consumo de CPU, a partir de um certo ponto, uma aplicação usando FaaS apresenta tempos de execução inferiores à mesma aplicação executando localmente.

A degradação de desempenho que se verifica na aplicação executando localmente se deve à concorrência, pois à medida que mais requisições simultâneas chegam para tra-

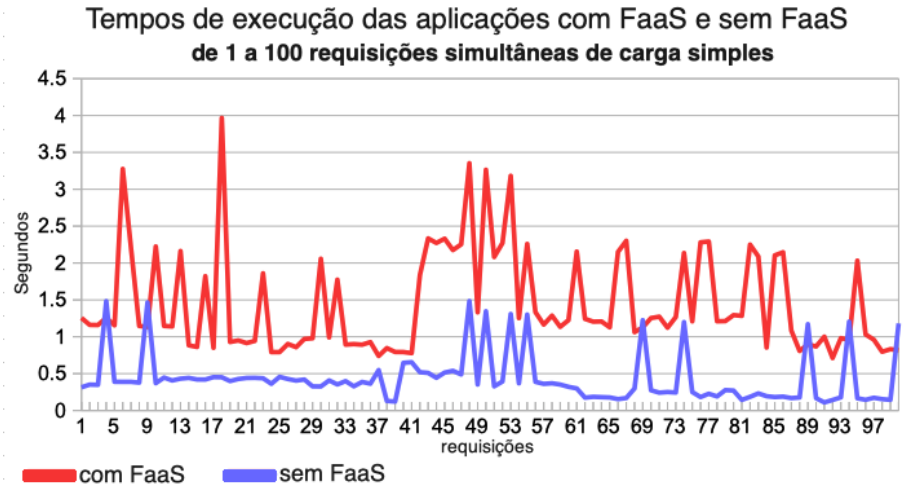

Figura 6.2: Resultados dos testes preliminares de carga simples.

tamento pelo servidor, maior é a quantidade de recursos que a máquina precisa alocar para esse tratamento. Esses recursos tendem a se esgotar, já que a instância local é uma máquina limitada, e com isso o servidor pode, inclusive, parar de responder, retirando a aplicação de operação. Esse comportamento pode ser verificado na Figura 6.3 por volta da quantidade 82 de requisições simultâneas, na qual ocorre um salto repentino no tempo de execução da aplicação sem *FaaS*. É provável que naquele ponto o servidor atingiu o limite da sua capacidade de processamento e passou a enfileirar as requisições, aumentando o tempo para tratamento dos processos. Esse aumento perdurou até por volta da requisição 93, quando o *overhead* do enfileiramento passou a interferir menos no tempo de execução. Um comportamento semelhante pode ser observado para aplicação com *FaaS* por volta de 57 requisições simultâneas, ou seja, bem antes do mesmo fato ocorrido na aplicação sem *FaaS*. Essa antecipação se explica pelo fato da instância disponibilizada pelo provedor possuir menor capacidade de processamento do que a instância destinada à execução da aplicação sem *FaaS* e, portanto, saturou mais cedo. Apesar disso, usando os serviços de *FaaS* na forma proposta, o provedor garante a elasticidade automaticamente, e o serviço potencialmente pode suportar uma quantidade muito superior de concorrência em comparação com a abordagem sem *FaaS*, pois novas instâncias poderão ser disponibilizadas automaticamente para atender esse aumento de processamento e garantir sua execução.

No caso de teste com sobrecarga da memória, mostrado na Figura 6.4, enquanto a variação do tempo de execução da aplicação usando FaaS se manteve estável, a curva da aplicação sem FaaS apontou no sentido ascendente. Entretanto, o cruzamento das curvas ocorreu mais rápido se comparado ao teste com CPU. Isso mostra que o alto consumo de memória degrada mais significativamente o tempo de execução do que o consumo de CPU, neste caso.

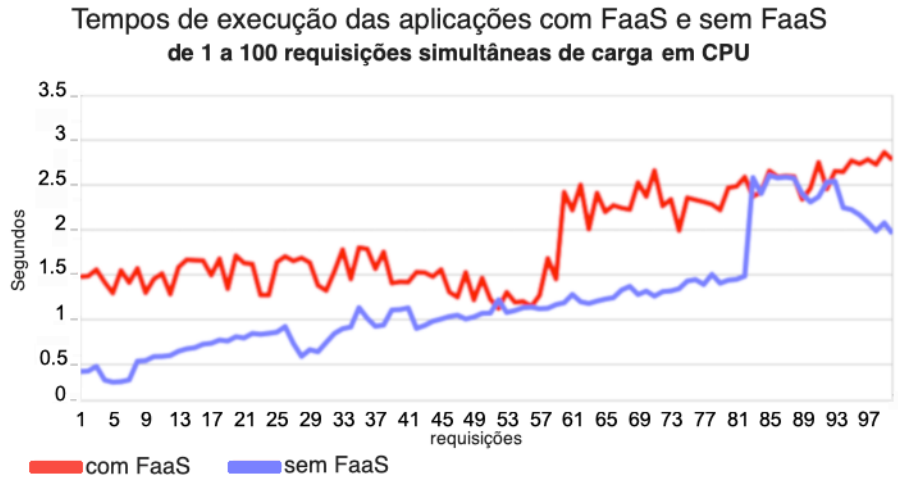

Figura 6.3: Resultados dos testes preliminares de carga sobre CPU.

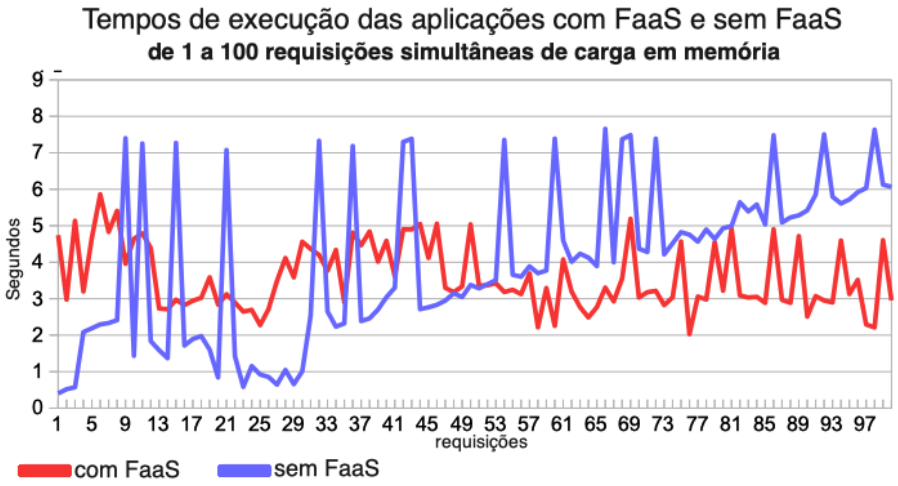

Figura 6.4: Resultados dos testes preliminares de carga sobre memória.

No último caso de teste, mostrado na Figura 6.5, observa-se que nas primeiras requisições a aplicação sem FaaS alterna entre tempos acima e baixos daqueles registrados pela aplicação com FaaS. Mas a partir da trigésima terceira requisição simultânea o tempo de execução da aplicação sem FaaS passa a registrar valores continuamente maiores do que os resultados da aplicação com FaaS, inclusive demonstrando uma tendência de alta gradativa. Na Tabela 6.1 é possível verificar que os piores resultados observados com FaaS e sem FaaS foram 20,43 segundos e 55,64 segundos, respectivamente. Isso mostra que, no teste de sobrecarga de I/O, houve uma diferença de 172% no pior caso.

A Tabela 6.1 apresenta o comparativo consolidado dos resultados obtidos em cada tipo de teste. Assim, é possível observar que os testes com carga simples e CPU obtiveram tempos médios de execução inferiores no servidor sem FaaS. Já a execução com sobrecarga de memória e I/O foram, em média, mais rápidas em servidores usando FaaS. Isso é

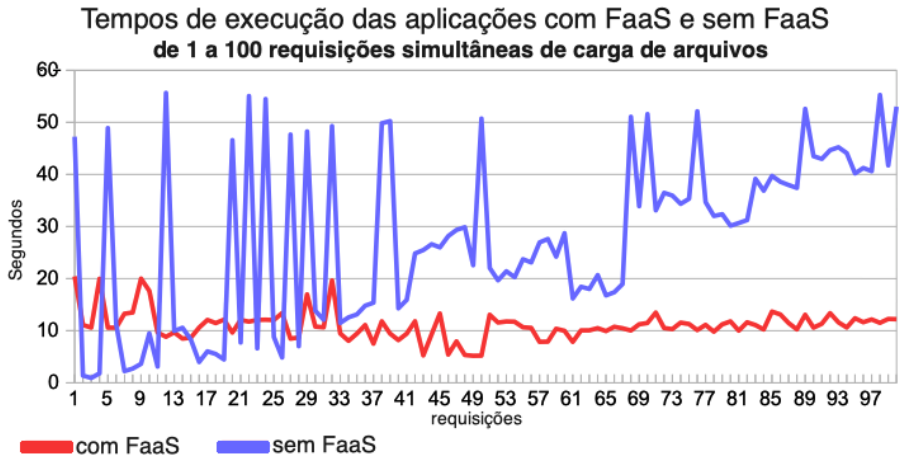

Figura 6.5: Resultados dos testes preliminares de carga sobre disco (I/O).

Tabela 6.1: Tempos de execução de aplicação sem FaaS e convertida a FaaS por meio do Node2FaaS.

Aplicação	Teste	Melhor	Pior	Média
Sem FaaS	Simples	0,71	3,96	0,43
Com FaaS	Simples	0,10	1,48	1,45
Sem FaaS	CPU	0,30	2,87	1,24
Com FaaS	CPU	1,12	2,61	1,72
Sem FaaS	Memória	0,40	7,65	3,99
Com FaaS	Memória	2,02	5,86	3,59
Sem FaaS	I/O	0,93	55,64	27,54
Com FaaS	I/O	5,19	20,43	11,04

explicado pelo fato do provedor de FaaS estar entregando elasticidade automática para o consumo de recursos, enquanto que no servidor sem FaaS a incumbência de gerenciar essa concorrência é do próprio servidor, e isso provoca enfileiramento, que bloqueia o processamento e atrasa a execução.

A análise dos resultados permite inferir que, de uma forma geral, para poucas requisições, o tempo de resposta da aplicação executando sem FaaS tende a ser melhor. Para aplicações que demandam muito recurso de CPU, caso ocorra pouca concorrência, aplicações sem FaaS apresentam melhores resultados, entretanto, a partir de um limiar, que nos experimentos ficou em torno de 80 requisições simultâneas, ambas abordagens apresentam desempenho semelhante.

Para aplicações com alto consumo de memória e I/O, os benefícios do uso de FaaS são evidentes, já a partir de baixos níveis de concorrência. Isso ocorre porque em aplicações monolíticas o consumo desse tipo de recurso provoca muitos bloqueios no processamento. Por outro lado, em aplicações orientada a FaaS, ocorre um alto paralelismo no consumo

desses recursos, reduzindo o efeito dos bloqueios. Assim, aplicações com esse tipo de características são fortes candidatas para adoção do modelo de FaaS, usando a abordagem proposta pelo Node2FaaS.

O código fonte deste trabalho, assim como aplicações de exemplo, podem ser encontrados no gerenciador de versionamento do Node2FaaS[1]. Além disso, o *framework* pode ser instalado no ambiente local utilizando o *Node Package Manager* (NPM), por meio da execução do comando *npm install node2faas*. Entretanto, ele está disponível apenas para sistemas operacionais MacOSX a partir da versão 10.14.

Os resultados desse experimento preliminar foram aceitos para publicação no *9th International Conference on Cloud Computing and Services Science - CLOSER 2019*, realizado entre 02 e 04 de maio de 2019, na ilha de Creta, na Grécia.

A próxima seção apresentará os resultados obtidos com o experimento conclusivo, já a partir da introdução do orquestrador Terraform.

6.3 Experimento com Orquestrador

Uma vez que o orquestrador Terraform foi incorporado ao *framework* Node2FaaS, fez-se necessário executar um experimento mais amplo a fim de avaliar o comportamento de uma aplicação NodeJS, agora convertida utilizado o Node2FaaS com o auxílio do Terraform e sobre múltiplos provedores. Para isso, foi desenvolvida uma aplicação NodeJS, cujas funções se propõem a aplicar uma carga de estresse controlável sobre os recursos computacionais do servidor, tais como CPU, memória e disco. Essa aplicação foi convertida para processar suas funções nos serviços de FaaS e aproveitar todos os benefícios associados ao paradigma da computação em nuvem, sobretudo, a elasticidade. Isso permite uma melhor distribuição da carga de trabalho, proporcionando que as aplicações mantenham um bom desempenho, mesmo em situações de estresse. As referidas funções são apresentadas nas Listagens 6.5, 6.6 e 6.7.

Listagem 6.5: Função de teste estressando a CPU para o 2º experimento.

```
exports.cpu = function(req, res) {
  var result = 0;
  for (var x in req.params){
    for (var i = 0;
         i < parseInt(req.params[x]);
         i++) {
      result = parseInt(result)*1
               +
               parseInt(req.params[x])*1;
    }
  }
  res.json(result);
} ;
```

[1]https://github.com/node2faas

Essas funções foram incluídas em uma aplicação NodeJS que foi convertida para a abordagem de RPC em *Function as a Service* utilizando o *framework* Node2FaaS para os três principais provedores de FaaS: AWS Lambda, Google Functions e Azure Functions. Tanto a aplicação original quanto as aplicações convertidas foram submetidas às baterias de testes que permitiram avaliar seus desempenhos em termos de tempo de execução.

Listagem 6.6: Função de teste estressando a memória para o 2º experimento.

```
exports.memory = function(req, res) {
  var result = new Array();
  for (var x = 0;
       x < parseInt(req.params["a"]);
       x++) {
    result[x] = new Array();
    for (var i = 0;
         i < parseInt(req.params["b"]);
         i++) {
      result[x][i] = x+i;
    }
  }
  result = eval(result.join("+"));
  res.json(result);
};
```

Listagem 6.7: Função de teste estressando I/O para o 2º experimento.

```
exports.io = function(req, res) {
  var rd = Math.random()*1000;
  var prefix = Math.floor(rd);
  var result = 0;
  const fs = require("fs");
  for (var x = 0;
       x < parseInt(req.params["a"]);
       x++) {
    for (var i = 0;
         i < parseInt(req.params["b"]);
         i++) {
      fs.writeFileSync("/tmp/"+
                       prefix+
                       x+i,
         "Node2FaaSTest",
         function(err) {
          if(err) {
            return console.log(err);
          }
            console.log("File saved!");
      });
      fs.unlinkSync("/tmp/"+
                    prefix+
                    x+i);
      result++;
    }
  }
  res.json(result);
};
```

A Tabela 6.2 mostra a composição dos parâmetros dos testes que foram realizados. Ao todo foram executados 1.080 casos de teste. Esse conjunto de testes foi utilizado para traçar um eixo médio para cada uma das 10 execuções de cada tipo (CPU, memória e I/O). O ambiente sobre o qual os teste locais foram executados contava com a seguinte configuração:

- **Máquina:** Macbook Pro 13"Mid 2012;
- **CPU:** Intel Core i5 2,5 GHz;

- **Memória:** 16GB 1600 Mhz DDR3;
- **Sistema Operacional:** MacOSX Mojave(X.14.6).

Tabela 6.2: Parâmetros dos testes para o experimento mais amplo.

Teste	Provedores	Cargas	Concorrência	Execuções
CPU	Local, GCP, AWS, Azure	1s, 5s, 10s	10, 50, 120	10
Memória	Local, GCP, AWS, Azure	1s, 5s, 10s	10, 50, 120	10
I/O	Local, GCP, AWS, Azure	1s, 5s, 10s	10, 50, 120	10

Neste caso, foram executadas duas rodadas de testes, em semanas diferentes. A primeira rodada ocorreu na semana de 08 a 14 de dezembro de 2019, enquanto a segunda rodada ocorreu na semana de 22 a 28 de dezembro de 2019. Para isso, foi desenvolvido um *"shell script"* para comandar os testes e armazenar os resultados. Ao final, os dados foram consolidados a partir do cálculo das médias das execuções de cada caso de teste e, então, foram gerados os gráficos com os resultados apurados para cada provedor e para o ambiente local em cada tipo de teste (CPU, memória e I/O).

Tabela 6.3: Resultados dos testes usando orquestrador.

Aplicação	Provedor	Esperado	Sucessos	Falhas	Taxa de confiabilidade	Média de tempo de execução	Menor tempo de execução	Maior tempo de execução	Amplitude do tempo de execução
CPU	Local	5400	5400	0	100,00%	280,100	1,128	1321,867	1320,739
CPU	AWS	5400	5324	76	98,59%	9,826	0,000[2]	22,657	22,657
CPU	GCP	5400	5291	81	98,50%	16,015	2,001	55,184	53,183
CPU	Azure	5400	2012	3388	37,26%	58,759	6,325	120,150	113,824
Memory	Local	5400	5400	0	100,00%	388,521	0,532	2386,063	2385,530
Memory	AWS	5400	3552	1848	65,78%	2,483	1,122	6,593	5,471
Memory	GCP	5400	3155	2245	58,43%	6,705	1,843	26,486	24,643
Memory	Azure	5400	1628	3772	30,15%	69,964	9,283	114,796	105,513
I/O	Local	5400	5400	0	100,00%	415,181	1,272	3801,251	3799,979
I/O	AWS	5400	5319	81	98,50%	2,019	0,450	4,846	4,395
I/O	GCP	5400	5323	77	98,57%	4,948	0,662	30,504	29,842
I/O	Azure	5400	0	5400	0,00%	N/A	N/A	N/A	N/A
Desvio padrão	-	-	**1889**	**1891**	**35,03%**	**163,5**	**2,890**	**1275,270**	**1275,905**

$$Média_{ponderada} = (TempoExecução_{média} * QtdeExecuçõesEsperadas/QtdeExecuçõesBemSucedidas) \quad (6.1)$$

[2]Valor original (sem arredondamento): 0,00000000982603751126972 segundos

As figuras 6.6, 6.7 e 6.8 apresentam os tempos de execução apurados no segundo experimento. Cada figura é composta por dois gráficos, sendo que os gráficos marcados com o número 1 foram gerados a partir do cálculo das médias considerando apenas as execuções bem sucedidas. Já os gráficos marcados com o número 2 foram gerados utilizando um cálculo ponderado conforme a Equação 6.1. Os resultados mostraram uma diferença acentuada no tempo de execução dos testes locais em relação aos tempos registrados pelas aplicações adotando a abordagem de RPC. Conforme pode ser verificado nos resultados apresentados na Tabela 6.3, e na Figura 6.6, que mostra o consolidado dos tempos médios para os casos de teste de CPU. É possível notar que enquanto na execução local da aplicação com estresse de CPU a média ficou em torno de 280 segundos, os serviços de FaaS se saíram muito melhores, chegando a atingir 9,8 segundos, no caso do AWS. Mesmo o provedor com pior resultado, Azure, ainda teve um resultado significativamente inferior àquele apurado pela execução local.

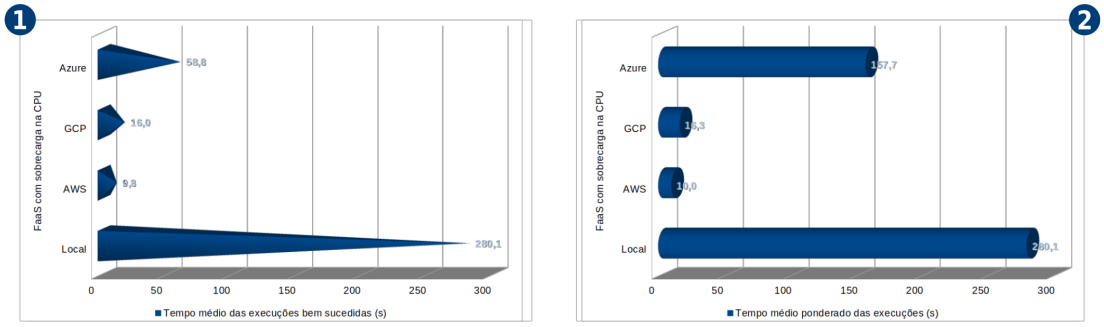

Figura 6.6: Resultados dos testes de carga sobre CPU.

A Figura 6.7 apresenta um resultado ainda mais discrepante entre os tempos de execução locais e nos serviços de FaaS, nesse caso, para os testes de alto consumo de memória. O AWS e o GCP registraram tempos médios de 2,4 e 6,7 segundos, respectivamente. O provedor Azure registrou um resultado mais elevado, com um tempo médio de 69,9 segundos, mas ainda assim muito inferior aos 388,5 segundos registrados como a média do processamento dos testes na máquina local.

O caso de testes que objetivava estressar a atividade de disco apresentou um resultado intrigante. Enquanto os provedores AWS e o GCP registraram médias de 2,0 e 4,9 segundos de tempo de execução, e os testes locais tenham resultado em uma média de 415,1 segundos, conforme pode ser verificado na Figura 6.8, o provedor Azure não conseguiu finalizar nenhum caso de teste. Considerando que os casos de teste variavam a concorrência entre 10, 50 e 120 requisições simultâneas, e que para uma requisição única o provedor Azure conseguia atender à requisição com sucesso, é possível inferir que o serviço que FaaS

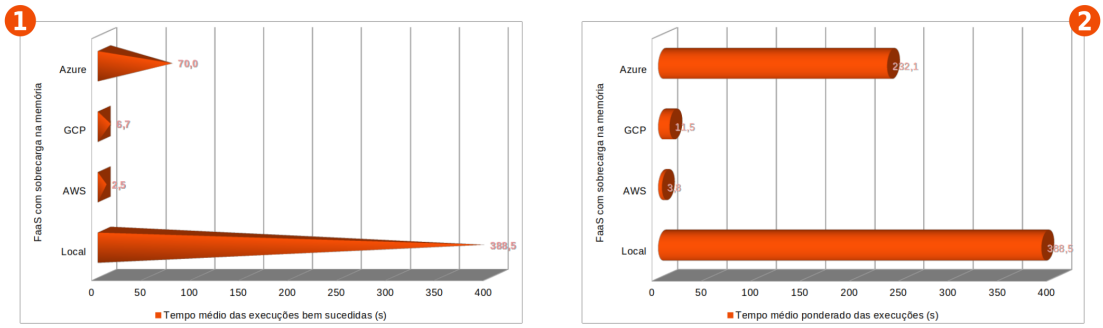

Figura 6.7: Resultados dos testes de carga sobre memória.

da Azure não consegue lidar satisfatoriamente com situações de concorrência envolvendo gravação e exclusão de arquivos.

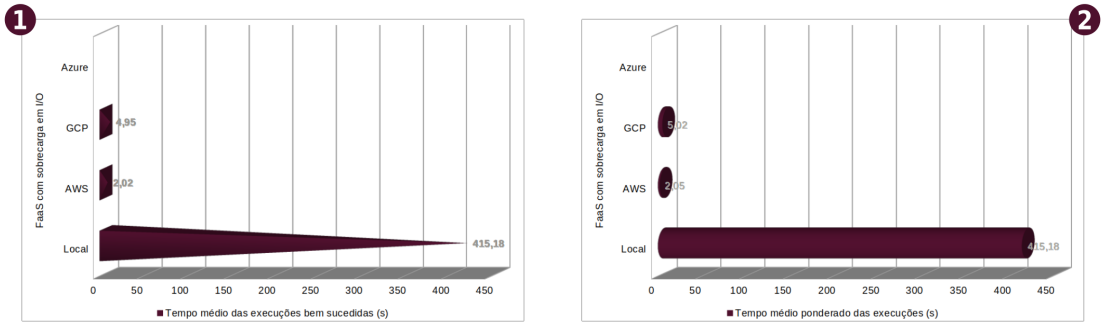

Figura 6.8: Resultados dos testes de carga sobre disco.

A Tabela 6.4 apresenta os percentuais de redução do tempo de execução entre a execução local e em cada provedor calculados conforme Equação 6.2. É possível observar que para o teste de I/O foi obtida uma redução de 99% no tempo de execução, utilizando o processamento do provedor da AWS.

$$Redução = 100 - (TempoExecuçãoLocal_{média}/TempoExecuçãoProvedor_{média} * 100) \quad (6.2)$$

Tabela 6.4: Percentuais de redução nos tempos de execução.

Teste	GCP	Azure	AWS
CPU	94,28%	79,02%	96,49%
Memória	98,27%	81,99%	99,36%
I/O	98,80%	–	**99,51%**

Apesar de promissores, os resultados apresentados até aqui permitem a analise de apenas uma característica desse contexto, o tempo de execução. Observando atentamente

Tabela 6.5: Mensagens de erro reportadas no 2º experimento.

Provedor	Mensagem de erro
AWS	{\"message\": \"Internal server error\"}
AWS	{"errno": "ENOTFOUND","code": "ENOTFOUND", "syscall": "getaddrinfo", "hostname": "5xyg1589i5.execute-api.us-east-1.amazonaws.com"}
AWS	{"errno": "ETIMEDOUT","code": "ETIMEDOUT","syscall": "connect","address": "99.84.27.18","port":443
GCP	\n<html><head>\n<meta http-equiv=\"content-type\"content=\"text/html;charset=utf-8\">\n<title>500 Server Error</title>\n</head>\n<body text=#000000 bgcolor=#ffffff>\n<h1>Error: Server Error</h1>\n<h2>The server encountered an error and could not complete your request.<p>Please try again in 30 seconds.</h2>\n<h2></h2>\n</body></html>\n
GCP	{"errno": "ENOTFOUND", "code": "ENOTFOUND","syscall":"getaddrinfo", "hostname": "us-central1-node2faas-248113.cloudfunctions.net"}

a Tabela 6.3 é possível observar as colunas “Esperado”, “Sucessos” e “Falhas”. Essas colunas registram a quantidade de casos de teste previstos, quantos destes obtiveram êxito e quantos falharam, respectivamente. As mensagens de erro mais recorrentes nos provedores AWS e GCP são mostradas na Tabela 6.5. O provedor Azure não consta na Tabela 6.5 devido ao fato das ocorrências de falha nos testes desse provedor implicarem somente interrupção precoce da conexão.

A partir dos dados das colunas “Esperado”, “Sucessos” e “Falhas” da Tabela 6.3 foi possível calcular uma taxa de confiabilidade para cada tipo de teste realizado, conforme mostrado na Figura 6.9. Nessa figura é possível perceber que a execução local manteve um nível elevado de confiabilidade em todos os seus testes, afinal, embora o tempo de execução tenha sido alto, nenhum teste deixou de ser executado com êxito. Para os testes relacionados a CPU e I/O os provedores GCP e AWS mantiveram níveis altos de confiabilidade, acima de 90%. Porém, é notável que aplicações que requerem grandes quantidade de memória podem sofrer com falhas em qualquer um dos provedores, pois o melhor deles, AWS, só conseguiu atingir 65% de confiabilidade nesse teste. O provedor Azure, de uma forma geral não se saiu bem no quesito confiabilidade em nenhum dos testes, sendo o seu melhor desempenho próximo de 37% no teste de CPU. No teste de I/O o provedor Azure fracassou totalmente registrando 0% de confiabilidade, devido a ocorrência de falhas em todos os testes.

Na Tabela 6.3 também é possível observar os registros dos tempos, mínimos e máximos, de execução para cada teste. Esses registros mostram os tempos de espera que as aplicações de teste tiveram que enfrentar durante as baterias de testes, e a partir dessas dados foi elaborada a Figura 6.10, na qual é possível perceber nitidamente a diferença do perfil de espera entre as execuções locais e as execuções nos provedores. Enquanto nos serviços de FaaS as curvas de mínimo e máximo andam muito próximas, as respectivas

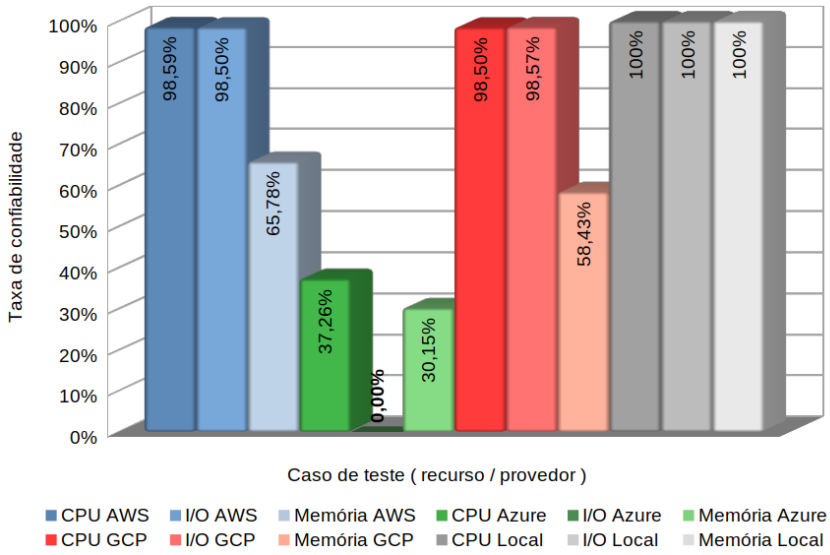

Figura 6.9: Taxa de confiabilidade apurada em cada teste.

curvas para os serviços locais possuem uma grande lacuna entre elas, demonstrando uma grande disparidade no desempenho esperado, já que a amplitude de variação é alta.

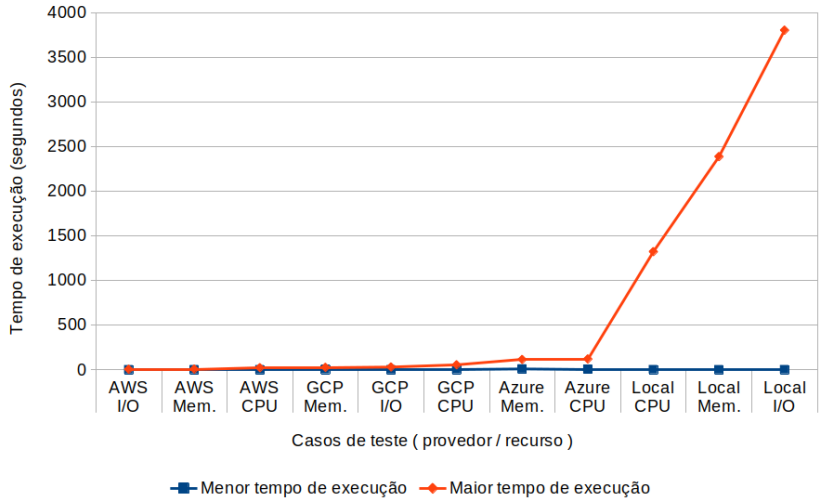

Figura 6.10: Maiores e menores tempos de execução em cada teste.

Os resultados dos testes apresentados (sem e com o orquestrador) demonstram, claramente, que a abordagem proposta pelo *framework* Node2faaS confere significativa redução no tempo de execução de aplicações com uso intensivo de recursos tais como CPU, memória e atividade em disco (I/O). As adequações necessárias para tornar as funções internas de uma aplicação NodeJS tradicional executáveis em serviços de FaaS pode ser um processo enfadonho. Além disso, esse processo manualmente tende a ser altamente propenso a erros quando realizado por um desenvolvedor. Por outro lado, a utilização do *framework* Node2faaS entrega a uniformidade e a consistência que uma aplicação necessita

para esse tipo de operação, e ainda proporciona uma grande agilidade para o desenvolvedor por conta das automações que o *framework* executa. Além disso, a utilização de um orquestrador sustentado pela comunidade confere a esse processo maior flexibilidade, pois havendo necessidade de transferir o processamento de um provedor para outro, basta acionar novamente o Node2FaaS apontando para o provedor e, então, esse movimento ficará por conta do *framework* e seu orquestrador acoplado. Tudo muito simples, rápido e seguro.

6.4 Considerações Finais

Diante do exposto neste capítulo, percebe-se que os resultados dos experimentos mostram a efetividade da atuação da elasticidade que a computação em nuvem oferece. A discrepância percebida entre a execução local e FaaS se deve ao enfileiramento do processamento que ocorre na execução local. Na execução em FaaS ocorre, de fato, um processamento paralelo, uma vez que o provedor se encarrega de entregar mais recursos computacionais, à medida que estes são percebidos como necessários. Contudo, entregar a responsabilidade de processar suas funções a um serviço de FaaS é assumir que falhas podem ocorrer e por conta disso é importante que as aplicações estejam preparadas para lidar com esse cenário, ou que eventuais prejuízos ocasionados por falhas não tenham repercussões catastróficas para a aplicação ou para o negócio. Nesse contexto, o Node2FaaS atua como um facilitador no processo de adoção da abordagem de processamento remoto escalável utilizando FaaS.

No próximo capítulo serão apresentadas as conclusões sobre este trabalho, assim como indicações de trabalhos futuros.

Capítulo 7

Considerações Finais e Trabalhos Futuros

A extensa utilização de serviços de nuvem é uma realidade atual. Assim sendo, maximizar a eficiência de utilização é uma necessidade constante nesse contexto. Logo, modelos de serviço que possibilitem melhorias aos clientes precisam ser aplicados para impactar positivamente os resultados.

Dessa forma, *Function as a Service* mostra ser um modelo de serviço de nuvem aderente às necessidades atuais. Entretanto, as dificuldades de estruturação e consumo desse modelo de serviço pode tornar o seu processo de adesão demasiadamente complexo, a ponto de desincentivar o uso. Além disso, os resultados dos experimentos realizados neste trabalho mostraram que esse modelo não confere resultados positivos para qualquer tipo de aplicação. Isso evidencia a necessidade de realização de um processo de verificação da aderência do algoritmo das funções ao paradigma de FaaS.

Diante disso, o *framework* proposto neste trabalho, Node2FaaS, mostrou-se eficiente na tarefa de converter aplicações NodeJS monolíticas para trabalharem com FaaS. Os experimentos mostraram que após a conversão feita pelo Node2FaaS houve ganhos significativos no tempo de execução de aplicações com uso intensivo de CPU, memória e manipulação de disco.

Os resultados preliminares deste trabalho foram aceitos para publicação no evento *9th International Conference on Cloud Computing and Services Science - CLOSER 2019* [36], e sua apresentação ocorreu no dia 03 de maio de 2019 na cidade de Heraklion, ilha de Creta, na Grécia. Na ocasião, o Node2FaaS recebeu diversos comentários e questionamentos que, além de enriquecer o trabalho, demonstram o interesse gerado pela abordagem proposta.

As conclusões preliminares a respeito dos resultados para as aplicações que consomem muita memória e/ou muita leitura e escrita de arquivos, mostraram que os ganhos obtidos após a conversão para FaaS chegaram a 170% em experimento preliminar. Além disso,

para obter o ganho com FaaS não foi necessário investimento em ajustes na aplicação, pois o Node2FaaS realizou todo o trabalho de conversão e de publicação de código no provedor de nuvem automaticamente.

Com o propósito de tornar o *framework* Node2FaaS aderente ao conceito de Sky Computing foi conduzida avaliação dos orquestrador de nuvem atualmente disponíveis a fim de escolher aquele que auxiliaria o *framework* na tarefa de publicação das funções em múltiplos provedores. Os resultados do referido estudo foram aceitos no *20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing - CCGRID 2020* com data de realização adiada para maio de 2021 devido a pandemia, em Melbourne, na Austrália. Assim, considerando os resultados desse estudo ficou definido que o Terraform integraria a pilha tecnológica do *framework* Node2FaaS, atuando como orquestrador *multicloud*.

A partir da adoção do Terraform foi possível executar um experimento mais amplo utilizando os três principais provedores do mercado atualmente: AWS, GCP e Azure. Os resultados apontaram ganhos de até 92% para aplicações com uso intensivo de CPU, 96% para aplicações com uso intensivo de leitura e escrita em disco, e de até 99% para aplicações com uso intensivo de memória.

Os resultados conclusivos deste trabalho foram aceitos para publicação na *10th International Conference on Cloud Computing and Services Science - CLOSER 2020* [153], e sua apresentação ocorreu no dia 07 de maio de 2020 em evento de *streaming online* em razão da pandemia da COVID-19.

Diante do exposto, nota-se que a abordagem proposta neste trabalho pode conferir a aplicações NodeJS significativos ganhos do tempo de processamento quando se comparados ao tempo de processamento local. A tríade Node2FaaS, Terraform e FaaS consiste em uma plataforma de amplificação do desempenho de aplicações que demandam utilização intensiva de recursos computacionais, tais como CPU, memória RAM e atividade de disco.

Todavia, mesmo o *framework* tendo uma camada para análise e decisão sobre o envio das funções para os serviços de FaaS, essa camada poderá ser aprimorada para fazer uma análise semântica mais profunda no código fonte de modo que a decisão seja tomada utilizando métricas relacionadas a aspectos do comportamento da aplicação.

Além disso, embora o *framework* esteja preparado para trabalhar com os principais provedores, seu orquestrador de nuvem, Terraform, oferece suporte a outros provedores. Portanto, em trabalhos futuros esse suporte pode ser facilmente ampliado, de modo a oferecer uma maior gama de opções de provedores aos utilizadores do *framework*, tais como: Alibaba Cloud Functions e IBM Cloud Functions, por exemplo.

O *framework* Node2FaaS oferece atualmente a execução das suas operações somente

por meio de uma interface de linha de comando (CLI) e isso limita a sua utilização aos desenvolvedores familiarizados com esse tipo de interface. Futuramente o *framework* pode ser estendido para receber uma interface *web*, na qual o usuário indique o endereço de um repositório (GitHub por exemplo), entre outros parâmetros, e comande a execução do Node2FaaS sobre o respectivo código.

Um comportamento dos provedores que pode penalizar a primeira requisição para um serviço, depois de um certo período de inatividade, é a desmobilização das instância que atende ao respectivo serviço. Isso pode acarretar uma variação no desempenho de uma aplicação orientada a FaaS. Atualmente, o Node2FaaS não possui nenhum mecanismo para evitar esse comportamento e é importante incluir uma estratégia que atenue esse efeito em trabalhos futuros.

Outro fato que chamou atenção durante os experimentos foi a ocorrência de falhas nas chamadas para os serviços de FaaS. É importante incluir futuramente um tratamento para esse fato, de modo a garantir que as tentativas de processamento sejam repetidas em caso de falha.

Referências

[1] JAMSA, D. K.: *Cloud computing: SaaS, PaaS, IaaS, virtualization, business models, mobile, security and more.* Jones and Bartlett Learning, Maio 2013. `Http://web.mit.edu/smadnick/www/wp/2013-01.pdf`, acesso em 24/06/2020. x, 7, 8, 9, 10, 11

[2] MALATHI, M.: *Cloud computing concepts.* Em *2011 3rd International Conference on Electronics Computer Technology*, volume 6, páginas 236–239, Abril 2011. x, 1, 2, 9, 13

[3] IMELGRAT.ME: *Cloud services delivery models.*, 2019. `https://imelgrat.me/cloud/cloud-services-models-help-business/`, acesso em 26/05/2019. x, 11

[4] TRUST RADIUS: *Infrastructure-as-a-service solutions trustmap*, 2019. `https://www.trustradius.com/infrastructure-as-a-service-iaas`, acesso em 27/05/2019. x, 13

[5] RIGHT SCALE: *State of the cloud report™*, 2019. `https://www.rightscale.com/lp/state-of-the-cloud`, acesso em 27/05/2019. x, 14

[6] GARTNER: *Magic quadrant for cloud infrastructure as a service, worldwide*, Julho 2019. `https://www.gartner.com/en/documents/3947472`, acesso em 04/03/2020. x, 15

[7] AWS: *AWS global infraestructure*, 2019. `https://aws.amazon.com/pt/about-aws/global-infrastructure/`, acesso em 28/05/2019. x, 14, 16

[8] MICROSOFT: *Azure*, 2019. `https://azure.microsoft.com`, acesso em 28/05/2019. x, xi, 11, 13, 17, 53, 55

[9] GOOGLE: *Google cloud platform*, 2019. `https://cloud.google.com/`, acesso em 28/05/2019. x, 10, 14, 18, 19

[10] MONTEIRO, A., C. Teixeira e J. S. Pinto: *Sky computing: exploring the aggregated cloud resources.* Cluster Comput, 20:621, 2017. x, 25, 26, 27

[11] AWS: *Api reference: Runinstances*, 2019. `https://docs.aws.amazon.com/AWSEC2/latest/APIReference/API_RunInstances.html`, acesso em 02/06/2019. x, 27

[12] MICROSOFT: *Create vm via restapi*, 2019. `https://docs.microsoft.com/en-us/azure/virtual-machines/linux/create-vm-rest-api`, acesso em 02/06/2019. x, 28

[13] WETTINGER, Johannes, Uwe Breitenbücher, Oliver Kopp e Frank Leymann: *Streamlining devops automation for cloud applications using tosca as standardized metamodel*. Future Generation Computer Systems, 56:317 – 332, 2016, ISSN 0167-739X. http://www.sciencedirect.com/science/article/pii/S0167739X15002496. x, 23, 29, 30, 31

[14] OASIS: *Topology and orchestration specification for cloud applications version 1.0*, 2019. http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.0/csd03/TOSCA-Simple-Profile-YAML-v1.0-csd03.html, acesso em 02/06/2019. x, 31

[15] CLOUDIFY: *Getting started*, 2019. https://cloudify.co/getting-started/, acesso em 02/06/2019. x, 29, 31, 32

[16] MICHELINO, D., J. C. Leon e L. F. Alvarez: *Implementation and testing of openstack heat*, setembro 2013. https://doi.org/10.5281/zenodo.7571. x, 32, 33, 34

[17] OPENSTACK: *Hot guide*, 2019. https://docs.openstack.org/heat/latest/template_guide/hot_guide.html, acesso em 02/06/2019. x, 29, 32, 33, 34, 35

[18] AWS: *AWS cloudformation*, 2019. https://aws.amazon.com/pt/cloudformation/, acesso em 02/06/2019. x, 29, 34, 35

[19] VMWARE: *Introducing vmware cloud assembly, vmware code stream and vmware service broker*, 2019. https://blogs.vmware.com/management/2018/08/introducing-cloud-automation.html, acesso em 02/06/2019. x, 29, 36, 37

[20] BRIKMAN, Y.: *Terraform: Up and Running: Writing Infrastructure as Code*. O'Reilly Media, 2017, ISBN 9781491977125. https://books.google.com.br/books?id=ZH1VDgAAQBAJ, acesso em 05/07/2020. x, 4, 29, 37, 38

[21] LEWIS J, Fowler M: *Microservices—a definition of this new architectural term*, 2018. http://martinfowler.com/articles/microservices.html, acesso em 25/11/2018. xi, 51, 52

[22] SAVAGE, Neil: *Going serverless*. Commun. ACM, 61(2):15–16, janeiro 2018, ISSN 0001-0782. http://doi-acm-org.ez54.periodicos.capes.gov.br/10.1145/3171583. xi, 52

[23] SPOIALA, Cristian: *Pros and cons of serverless computing.*, 2017. https://assist-software.net/blog/pros-and-cons-serverless-computing-faas-/comparison-aws-lambda-vs-azure-functions-vs-google, acesso em 13/10/2017. xi, 2, 11, 12, 50, 53, 54

[24] AMAZON: *AWS*, 2019. https://aws.amazon.com/, acesso em 14/06/2019. xi, 10, 53, 54, 55, 64

[25] GOOGLE: *Cloud functions*, 2019. https://cloud.google.com/functions/, acesso em 14/06/2019. xi, 54, 55

[26] WORTMANN, F. e K. Flüchter: *Internet of things: Technology and value added*. Bus Inf Syst Eng, 57:221, 2014. https://doi-org.ez54.periodicos.capes.gov.br/10.1007/s12599-015-0383-3. xi, 59

[27] KAWAMOTO, Yuichi, Hiroki Nishiyama, Nei Kato, Naoko Yoshimura e Shinichi Yamamoto: *Internet of things (iot): Present state and future prospects*. IEICE Transactions on Information and Systems, E97.D(10):2568–2575, 2014. xi, 59, 60

[28] PERSON, Per e Ola Angelsmark: *Kappa: Serverless iot deployment*. Em *Proceedings of the 2Nd International Workshop on Serverless Computing*, WoSC '17, páginas 16–21, New York, NY, USA, 2017. ACM, ISBN 978-1-4503-5434-9. http://doi.acm.org/10.1145/3154847.3154853. xi, 60, 61, 62

[29] CHARD, Ryan, Tyler J. Skluzacek, Zhuozhao Li, Yadu Babuji, Anna Woodard, Ben Blaiszik, Steven Tuecke, Ian Foster e Kyle Chard: *Serverless supercomputing: High performance function as a service for science*, 2019. xii, 57, 58

[30] YOO, C. S.: *Cloud computing: Architectural and policy implications*. Springer Science and Business Media, Maio 2011. 1

[31] MELL, Peter e Tim Grance: *The NIST definition of cloud computing*. National Institute of Standards and Tecnology, Setembro 2011. 2, 7, 8, 9, 10, 11, 12, 49

[32] SERVERLESS: *Serverless: Build apps with radically less overhead and cost*, 2019. https://serverless.com, acesso em 14/06/2019. 2, 79, 80, 82

[33] SATHEESH, Mithun, Bruno Joseph D'Mello e Jason Krol: *Web Development with MongoDB and NodeJS*, página 4. Packt Publishing, 2015, ISBN 9781788395083. 2, 65

[34] SHAH, Hezbullah e Tariq Soomro: *Node.js challenges in implementation*. Global Journal of Computer Science and Technology: Network, Web and Security, 17, Junho 2017. 2, 64, 66

[35] de Carvalho, L. R. e A. Patricia Favacho de Araujo: *Performance comparison of terraform and cloudify as multicloud orchestrators*. Em *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, páginas 380–389, May 2020. 4

[36] CARVALHO, L. e Aletéia P. F. Araújo: *Framework node2faas: Automatic nodejs application converter for function as a service*. Em *Proceedings of the 9th International Conference on Cloud Computing and Services Science - Volume 1: CLOSER,*, páginas 271–278. INSTICC, SciTePress, 2019, ISBN 978-989-758-365-0. 4, 98

[37] SPILLNER, Josef: *Transformation of python applications into function-as-a-service deployments*. CoRR, abs/1705.08169, 2017. http://arxiv.org/abs/1705.08169, acesso em 05/07/2020. 5, 79, 82

[38] VMWARE: *vcloud director*, 2019. https://www.vmware.com/br/products/vcloud-director.html, acesso em 20/06/2019. 8

[39] OPENSTACK: *What is openstack?*, 2019. https://www.openstack.org/software/, acesso em 08/06/2019. 8, 32, 33

[40] YOUSSEF, Ahmed E.: *Exploring cloud computing services and applications*. Journal of Emerging Trends in Computing and Information Sciences, 2012. 10, 13

[41] AHMED, Firas e Amer Al Nejam: *Cloud computing: Technical challenges and cloudsim functionalities*. International Journal of Science and Research, 2:2319–7064, fevereiro 2013. 10

[42] CLOUD FOUNDRY: *Open source cloud application platform*, 2019. https://www.cloudfoundry.org/, acesso em 29/05/2019. 10, 22

[43] MACHADO, A. C. T.: *A ferramenta google docs: Construção do conhecimento através da interação e colaboração*. Revista científica de educação à distância, Junho 2009. http://periodicos.unimesvirtual.com.br/index.php/paideia/article/download/73/51, acesso em 05/07/2020. 11, 18

[44] DUAN, Y., G. Fu, N. Zhou, X. Sun, N. C. Narendra e B. Hu: *Everything as a service (xaas) on the cloud: Origins, current and future trends*. Em *2015 IEEE 8th International Conference on Cloud Computing (CLOUD)*, volume 00, páginas 621–628, Junho 2015. doi.ieeecomputersociety.org/10.1109/CLOUD.2015.88, acesso em 05/07/2020. 11, 12

[45] DELL: *Storage as a service.*, 2019. https://www.dellemc.com/pt-br/glossary/storage-as-a-service.htm, acesso em 26/05/2019. 11, 12

[46] ZHENG, Xi: *Database as a service - current issues and its future*. CoRR, abs/1804.00465, 2018. http://arxiv.org/abs/1804.00465, acesso em 05/07/2020. 11, 12

[47] CITRIX: *Deliver virtual windows apps and desktops simply and securely*, 2019. https://www.citrix.com/products/citrix-managed-desktops.html, acesso em 26/05/2019. 12

[48] ABE, John Olorunfemi e Burak BERK: *A data as a service (daas) model for gpu-based data analytics*. Em *IEEE/IFIP NTMS Workshop on Big Data and Emerging Trends*. IEEE/IFIP, 2017. 12

[49] IBM: *IBM resiliency backup as a service*, 2019. https://www.ibm.com/br-pt/marketplace/managed-backup-services, acesso em 26/05/2019. 12

[50] VEEAM: *Garanta proteção de dados com a recuperação de desastres*, 2019. https://www.veeam.com/br/disaster-recovery-as-a-service-draas.html, acesso em 26/05/2019. 12

[51] ZAWOAD, Shams, Amit Kumar Dutta e Ragib Hasan: *Seclaas: Secure logging-as-a-service for cloud forensics*. CoRR, abs/1302.6267, 2013. http://arxiv.org/abs/1302.6267, acesso em 05/07/2020. 12

[52] PASSERINI, Katia, Ayman El Tarabishy e Karen Patten: *Information Technology for Small Business: Managing the Digital Enterprise*. Springer, 2012, ISBN 978-1-4614-3040-7. 12

[53] IBM: *Watson machine learning*, 2019. `https://www.ibm.com/cloud/machine-learning`, acesso em 26/05/2019. 12

[54] SUSE: *Plataforma suse caas*, 2019. `https://www.suse.com/pt-br/products/caas-platform/`, acesso em 26/05/2019. 12

[55] TRUST RADIUS: *Trust radius*, 2019. `https://www.trustradius.com/static/about-us`, acesso em 27/05/2019. 13

[56] AWS: *AWS overview*, 2018. `https://d1.awsstatic.com/whitepapers/aws-overview.pdf`, acesso em 28/05/2019. 14, 15, 36

[57] DIGITAL OCEAN: *Welcome to the developer cloud*, 2019. `https://www.digitalocean.com/`, acesso em 29/05/2019. 14, 21

[58] GARTNER: *Gartner equips executives across the enterprise to make the right decisions and stay ahead of change*, 2019. `https://www.gartner.com/en/about`, acesso em 27/05/2019. 14

[59] MICROSOFT AZURE: *Microsoft announces windows azure and azure services platform*, 2008. `https://azure.microsoft.com/pt-br/blog/microsoft-announces-windows-azure-and-azure-services-platform/`, acesso em 28/05/2019. 17

[60] HAUGER, D.: *Windows azure general availability*, 2010. `https://blogs.microsoft.com/blog/2010/02/01/windows-azure-general-availability/`, acesso em 28/05/2019. 17

[61] RED HAT: *What is kubernetes*, 2019. `https://www.redhat.com/pt-br/topics/containers/what-is-kubernetes`, acesso em 29/05/2019. 18, 56

[62] GOOGLE: *Introducing google app engine + our new blog*, april 2008. `http://googleappengine.blogspot.com/2008/04/introducing-google-app-engine-our-new.html`, acesso em 28/05/2019. 18

[63] IBM: *IBM cloud*, 2019. `https://www.ibm.com/cloud/`, acesso em 28/05/2019. 20, 21

[64] ORACLE: *Oracle cloud*, 2019. `https://cloud.oracle.com`, acesso em 29/05/2019. 20, 21

[65] APACHE: *Apache hadoop*, 2019. `https://hadoop.apache.org/`, acesso em 20/06/2019. 21

[66] APACHE: *Apache kafka*, 2019. `https://kafka.apache.org/`, acesso em 20/06/2019. 21

[67] ALIBABA: *Alibaba cloud*, 2019. https://www.alibabacloud.com, acesso em 28/05/2019. 21

[68] YAN, H., H. Wang, X. Li, Y. Wang, D. Li, Y. Zhang, Y. Xie, Z. Liu, W. Cao e F. Yu: *Cost-efficient consolidating service for Aliyun's cloud-scale computing*. IEEE Transactions on Services Computing, 12(1):117–130, Janeiro 2019, ISSN 1939-1374. 21

[69] HEROKU: *Developers, teams, and businesses of all sizes use heroku to deploy, manage, and scale apps.*, 2019. https://www.heroku.com/, acesso em 29/05/2019. 22

[70] RACKSPACE: *Especialistas apaixonados, dedicados ao seu sucesso.*, 2019. https://www.rackspace.com/pt-br, acesso em 29/05/2019. 22

[71] HUANG, Y., S. Shekhar e H. Xiong: *Discovering colocation patterns from spatial data sets: a general approach*. IEEE Transactions on Knowledge and Data Engineering, 16(12):1472–1485, 2004. 22

[72] CLOUDBEES: *Cloudbees acquires electric cloud*, 2019. https://www.cloudbees.com, acesso em 29/05/2019. 22

[73] CLOUDFOUNDRY FOUNDATION: *The open service broker api connects developers to a global ecosystem of services*, 2019. https://www.openservicebrokerapi.org/, acesso em 20/06/2019. 23

[74] ENGINE YARD: *The full-stack ruby on rails devops experts*, 2019. https://www.engineyard.com/, acesso em 29/05/2019. 23

[75] SCALEWAY: *Scaleway*, 2019. https://www.scaleway.com/en/, acesso em 29/05/2019. 23

[76] KAMATERA: *Kamatera*, 2019. https://www.kamatera.com, acesso em 29/05/2019. 23

[77] OVH: *Instâncias public cloud*, 2019. https://www.ovh.pt/public-cloud/instances/, acesso em 29/05/2019. 23, 24

[78] HETZNER: *Hetzner cloud*, 2019. https://www.hetzner.com/cloud, acesso em 29/05/2019. 24

[79] KOLB, S., J. Lenhard e G. Wirtz: *Application migration effort in the cloud - the case of cloud platforms*. Em *2015 IEEE 8th International Conference on Cloud Computing*, páginas 41–48, Junho 2015. 25

[80] KRITIKOS, K. e D. Plexousakis: *Multi-cloud application design through cloud service composition*. Em *2015 IEEE 8th International Conference on Cloud Computing*, páginas 686–693, Junho 2015. 25

[81] ELKHATIB, Y.: *Defining cross-cloud systems*. CoRR, abs/1602.02698, 2016. http://arxiv.org/abs/1602.02698, acesso em 05/07/2020. 25

[82] PARAISO, F., N. Haderer, P. Merle, R. Rouvoy e L. Seinturier: *A federated multi-cloud paas infrastructure*. Em *2012 IEEE Fifth International Conference on Cloud Computing*, páginas 392–399, Junho 2012. 25

[83] GROZEV, Nikolay e Rajkumar Buyya: *Inter-cloud architectures and application brokering: taxonomy and survey*. Software: Practice and Experience, 44(3):369–390, 2014. `https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2168`, acesso em 05/07/2020. 25

[84] KEAHEY, Katarzyna, Maurício Tsugawa, Andréa Matsunaga e José A. B. Fortes: *Sky computing*. Em *IEEE Internet Computing*, página 43–51. IEEE Computer Society, 2009. 26

[85] MORRIS, K.: *Infrastructure as Code: Managing Servers in the Cloud*. Safari Books Online. O'Reilly Media, 2016, ISBN 9781491924396. `https://books.google.com.br/books?id=BIhRDAAAQBAJ`, acesso em 05/07/2020. 27

[86] CHEF: *Devops dashboard for complete operational visibility into the coded enterprise*, 2019. `https://www.chef.io/products/automate/`, acesso em 20/06/2019. 28, 30

[87] SHAMBAUGH, R, A. Weiss e A Guha: *Rehearsal: A configuration verification tool for puppet*. SIGPLAN Not., 51(6):416–430, junho 2016, ISSN 0362-1340. `http://doi.acm.org/10.1145/2980983.2908083`, acesso em 05/07/2020. 28

[88] RED HAT: *Ansible: Automation for everyone*, 2019. `https://www.ansible.com/`, acesso em 20/06/2019. 28

[89] KOVÁCS, József e Péter Kacsuk: *Occopus: a multi-cloud orchestrator to deploy and manage complex scientific infrastructures*, march 2018. `https://doi-org.ez54/10.1007/s10723-017-9421-3`, acesso em 20/06/2019. 29, 31

[90] PHAM, L M, A Tchana, D Donsez, N De Palma e V Zurczak: *Roboconf: a hybrid cloud orchestrator todeploy complex applications*. Em *2015 IEEE 8th International Conference on Cloud Computing*, páginas 41–48, Junho 2015. 29

[91] CARRASCO J, Durán F. e Pimentel E.: *Trans-cloud: Camp/tosca-based bidimensional cross-cloud*. Computer Standards & Interfaces, 58:167–179, 2018, ISSN 0920-5489. 29

[92] WANG, X, Z. Liu, Y. Qi e J. Li: *Livecloud: A lucid orchestrator for cloud datacenters*. Em *4th IEEE International Conference on Cloud Computing Technology and Science Proceedings*, páginas 341–348, Dezembro 2012. 29

[93] LE, D., H. Truong, G. Copil, S. Nastic e S. Dustdar: *Salsa: A framework for dynamic configuration of cloud services*. Em *2014 IEEE 6th International Conference on Cloud Computing Technology and Science*, páginas 146–153, Dezembro 2014. 29

[94] CABALLER, Miguel, Damián Segrelles, Germán Moltó e Ignacio Blanquer: *A platform to deploy customized scientific virtual infrastructures on the cloud*. Concurrency and Computation: Practice and Experience, 27(16):4318–4329, 2015. https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.3518, acesso em 05/07/2020. 29

[95] WALTER, Maria, Maristela Holanda, Guilherme Vergara, Michel Rosa, Aleteia Araujo e Breno Moura: *Bionimbuz: a federated cloud platform for bioinformatics applications*. International Journal of Data Mining and Bioinformatics, 18:144, janeiro 2017. 29

[96] LOULLOUDES, Nicholas: *The celar project*, 2019. https://github.com/CELAR/c-Eclipse, acesso em 05/07/2020. 29

[97] GUERRIERO, Michele: *Dicer*, 2019. https://github.com/DICERs/DICER, acesso em 05/07/2020. 29

[98] OPEN TOSCA: *Open tosca*, 2019. http://www.opentosca.org/, acesso em 05/07/2020. 29

[99] APACHE: *Apache aria tosca orchestration engine*, 2019. https://ariatosca.incubator.apache.org/, acesso em 05/07/2020. 29

[100] BOROVŠAK, Tadej: *xopera orchestrator*, 2019. https://github.com/xlab-si/xopera-opera, acesso em 05/07/2020. 29

[101] ALIEN4CLOUD: *Alien4cloud*, 2019. http://alien4cloud.github.io, acesso em 05/07/2020. 29

[102] SHIRINKIN, Kirill: *Getting Started with Terraform*. Packt Publishing, 2017, ISBN 1786465108, 9781786465108. 38

[103] WORDPRESS.ORG: *Liberdade para publicar*, 2019. https://br.wordpress.org/about/, acesso em 05/07/2020. 40

[104] GARTNER: *Gartner forecasts worldwide public cloud revenue to grow 17.5 percent in 2019*, 2019. https://www.gartner.com/en/newsroom/press-releases/2019-04-02-gartner-forecasts-worldwide-public-cloud-revenue-to-g, acesso em 02/06/2019. 41

[105] BASILI, V. R., M. Lindvall, M. Regardie, C. Seaman, J. Heidrich, J. Münch, D. Rombach e A. Trendowicz: *Linking software development and business strategy through measurement*. Computer, 43(4):57–65, Abril 2010, ISSN 0018-9162. 49

[106] LARRUCEA, X., I. Santamaria, R. Colomo-palacios e C. Ebert: *Microservices*. IEEE Software, 35(3):96–100, Maio 2018, ISSN 0740-7459. 49

[107] ZIMMERMANN, O.: *Microservices tenets: Agile approach to service development and deployment*. Comput Sci Res Dev, 32:301, 2017. 50, 52

[108] ANDERSON, Charles: *Docker.* IEEE Software, 32(3):102 – c3, 2015, ISSN 07407459. http://search-ebscohost-com.ez54.periodicos.capes.gov.br/login.aspx?direct=true&db=iih&AN=102288020&lang=pt-br&site=ehost-live, acesso em 05/07/2020. 50

[109] FIELDING, R. T.: *Architectural styles and the design of network-based software architectures [ph. d. thesis]: Irvine, university of california.* Information and Computer Science Department, 2000. 50

[110] ATHANASOPOULOS, Michael e Kostas KONTOGIANNIS: *Extracting rest resource models from procedure-oriented service interfaces.* Journal of Systems and Software, 100:149 – 166, 2015, ISSN 0164-1212. http://www.sciencedirect.com/science/article/pii/S0164121214002362, acesso em 05/07/2020. 51

[111] PAULA, Gabriel Souza de: *Avaliação de serviços serverless: um experimento piloto*, 2018. http://repositorio.roca.utfpr.edu.br/jspui/handle/1/10033, acesso em 28/06/2018. 53, 54

[112] BILLOCK, Matt: *The pros and cons of AWS lambda*, 2017. https://dzone.com/articles/the-pros-and-cons-of-aws-lambda, acesso em 05/07/2020. 54

[113] MICROSOFT: *Azure functions*, 2019. https://azure.microsoft.com/pt-br/services/functions/, acesso em 14/06/2019. 55

[114] ALIBABA: *Alibaba functions*, 2019. https://www.alibabacloud.com/help/doc-detail/52895.htm?spm=a2c63.l28256.b99.1.34b73c94c0mQb4, acesso em 14/06/2019. 56

[115] IBM: *IBM cloud functions*, 2019. https://www.ibm.com/br-pt/cloud/functions, acesso em 14/06/2019. 56

[116] APACHE: *What is apache openwhisk?*, 2019. https://openwhisk.apache.org/, acesso em 14/06/2019. 56

[117] ORACLE: *Announcing oracle functions*, 2019. https://blogs.oracle.com/cloud-infrastructure/announcing-oracle-functions, acesso em 14/06/2019. 56

[118] Oracle: *O oracle functions agora está geralmente disponível*, 2020. https://www.oracle.com/br/cloud/cloud-native/functions/, acesso em 2020-06-07. 56

[119] FN: *Fn project: Open source. container-native. serverless platform.*, 2019. https://fnproject.io/, acesso em 14/06/2019. 56

[120] KUBELESS: *Kubeless - the kubernetes native serverless framework: Build advanced applications with faas on top of kubernetes*, 2019. https://kubeless.io/, acesso em 14/06/2019. 56

[121] OPENFAAS: *Openfaas - serverless functions made simple*, 2019. https://docs.openfaas.com, acesso em 14/06/2019. 56

[122] STUBBS, Joe, Rion Dooley e Matthew Vaughn: *Containers-as-a-service via the Actor Model*. Em *Gateways 2016 proceedings*, janeiro 2017. https://figshare.com/articles/Containers-as-a-service_via_the_Actor_Model/4490747, acesso em 05/07/2020. 56

[123] TEXAS ADVANCED COMPUTING CENTER: *What is TACC?*, 2020. https://www.tacc.utexas.edu, acesso em 07/06/2020. 56

[124] AKKUS, Istemi Ekin, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya e Volker Hilt: *SAND: Towards high-performance serverless computing*. Em *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, páginas 923–935, Boston, MA, julho 2018. USENIX Association, ISBN 978-1-939133-01-4. https://www.usenix.org/conference/atc18/presentation/akkus, acesso em 05/07/2020. 57

[125] PERERA, C., C. H. Liu e S. Jayawardena: *The emerging internet of things marketplace from an industrial perspective: A survey*. IEEE Transactions on Emerging Topics in Computing, 3(4):585–598, Dezembro 2015, ISSN 2168-6750. 58

[126] KIM, Suwon e Seongcheol Kim: *A multi-criteria approach toward discovering killer iot application in korea*. Technological Forecasting and Social Change, 102:143 – 155, 2016, ISSN 0040-1625. http://www.sciencedirect.com/science/article/pii/S0040162515001237, acesso em 05/07/2020. 58

[127] SANTORO, Gabriele, Demetris Vrontis, Alkis Thrassou e Luca Dezi: *The internet of things: Building a knowledge management system for open innovation and knowledge management capacity*. Technological Forecasting and Social Change, 136:347 – 354, 2018, ISSN 0040-1625. http://www.sciencedirect.com/science/article/pii/S0040162517302846, acesso em 05/07/2020. 59

[128] IDC: *Idc forecasts worldwide spending on the internet of things to reach $745 billion in 2019, led by the manufacturing, consumer, transportation and utilities sectors*, 2019. https://www.idc.com/getdoc.jsp?containerId=prUS44596319, acesso em 08/06/2019. 59

[129] AHMED, Syed Hassan e Shalli RANI: *A hybrid approach, smart street use case and future aspects for internet of things in smart cities*. Future Generation Computer Systems, 79:941 – 951, 2018, ISSN 0167-739X. http://www.sciencedirect.com/science/article/pii/S0167739X17309949, acesso em 05/07/2020. 59, 60

[130] PINTO, Duarte, João Pedro Dias e Hugo Sereno Ferreira: *Dynamic allocation of serverless functions in iot environments*. CoRR, abs/1807.03755, 2018. http://arxiv.org/abs/1807.03755, acesso em 05/07/2020. 60

[131] CHENG, Bin, Jonathan FÜRST, Gurkan SOLMAZ e Takuya SANADA: *Fog function: Serverless fog computing for data intensive iot services*, 2019. 60, 61

[132] PERSSON, P. e O. Angelsmark: *Calvin – merging cloud and iot*. Procedia Computer Science, 52, dezembro 2015. 61

[133] SHI, W., J. Cao, Q. Zhang, Y. Li e L. Xu: *Edge computing: Vision and challenges*. IEEE Internet of Things Journal, 3(5):637–646, 2016. 61

[134] HAYES, J.: *Thin clients' fat challenge [it desktop computing]*. Engineering Technology, 4(21):52–53, Dezembro 2009, ISSN 17509637. 63

[135] CHAPIN, John e Michael Roberts: *What is Serverless*. Oreilly, Junho 2017, ISBN 9781491984161. 63

[136] FLANAGAN, D.: *JavaScript: O Guia Definitivo*. Bookman Editora, 2011, ISBN 9788565837484. https://books.google.com.br/books?id=zWNyDgAAQBAJ, acesso em 05/07/2020. 64, 65

[137] IEEE: *Interactive: The top programming languages 2019*, 2019. https://spectrum.ieee.org/computing/software/the-top-programming-languages-2019, acesso em 15/04/2020. 65

[138] IEEE: *Interactive: The top programming languages 2018*, 2018. https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2018, acesso em 25/11/2018. 65

[139] BAYSHORE NETWORKS: *Ruby/eventmachine*, 2019. http://www.inbudapesthotels.com/rubyeventmachinecom/, acesso em 24/06/2019. 66

[140] TWISTED MATRIX LABS: *What is twisted?*, 2019. https://twistedmatrix.com/trac/, acesso em 24/06/2019. 66

[141] NPM: *About npm*, 2018. https://docs.npmjs.com/about-npm/, acesso em 25/11/2018. 66

[142] EXPRESS: *Express install*, 2018. http://expressjs.com/pt-br/starter/installing.html, acesso em 25/11/2018. 66

[143] KOA: *Koa: next generation web framework for node.js*, 2019. https://koajs.com/, acesso em 24/06/2019. 66

[144] BIRRELL, Andrew e Bruce Nelson: *Implementing remote procedure calls*. ACM Transactions on Computer Systems, 2:39–, fevereiro 1984. 68

[145] SPILLNER, Josef e Serhii Dorodko: *Java code analysis and transformation into AWS lambda functions*. CoRR, abs/1702.05510, 2017. http://arxiv.org/abs/1702.05510, acesso em 05/07/2020. 79, 82

[146] VERCEL: *Develop. preview. ship.*, 2020. https://vercel.com, acesso em 08/06/2020. 79, 80, 82

[147] CLAUDIA.JS: *Claudia.js: Serverless javascript, the easy way*, 2019. https://claudiajs.com/, acesso em 14/06/2019. 79, 80, 82

[148] JONES, Rich: *Zappa - serverless python*, 2020. https://github.com/Miserlou/Zappa, acesso em 08/06/2020. 79, 81, 82

[149] JONAS, Eric, Qifan PU, Shivaram VENKATARAMAN, Ion STOICA e Benjamin RECHT: *Occupy the Cloud: Distributed Computing for the 99%*. 2017. 79, 81, 82

[150] THE PALLETS PROJECTS: *Flask*, 2020. `https://palletsprojects.com/p/flask`, acesso em 08/06/2020. 81

[151] DJANGO SOFTWARE FOUNDATION: *Meet django*, 2020. `https://www.djangoproject.com`, acesso em 08/06/2020. 81

[152] GARTNER: *Magic quadrant for cloud infrastructure as a service, worldwide*, 2018. `https://www.gartner.com/en/documents/3875999`, acesso em 27/05/2019. 84

[153] CARVALHO, L. e Aleteia P. F. de Araújo: *Remote procedure call approach using the node2faas framework with terraform for function as a service*. Em *Proceedings of the 10th International Conference on Cloud Computing and Services Science - Volume 1: CLOSER,*, páginas 312–319. INSTICC, SciTePress, 2020, ISBN 978-989-758-424-4. 99