INSTITUTO SUPERIOR
DE TECNOLOGIAS
AVANÇADAS DE LISBOA

AVALIAÇÃO DE PLATAFORMAS *SERVERLESS* QUE IMPLEMENTAM *CONTAINERS-AS-A-SERVICE*

Isaac Newton Melo Machado

Dissertação para obtenção do Grau de Mestre em

INFORMÁTICA

Júri

Presidente: Professor Doutor Paulo André Reis Duarte Branco

Arguente: Professor Doutor Paulo Jorge Gonçalves Pinto

Orientadora: Professora Doutora Isabel Maria Surdinho Borges Alvarez

Janeiro, 2022

Agradecimentos

Em primeiro lugar, agradeço e dedico este trabalho à minha esposa, Ana Caroline Dolivo Costa, pois sem ela esta conquista não seria possível. Agradeço à minha mãe, Marilene Carvalho Melo, pelo suporte contínuo e educação que me foi dada. Por fim, agradeço aos meus colegas de curso, em especial Juliano Sathler, e à minha orientadora, Isabel Alvarez.

Resumo

Serverless Computing ou *serverless*, é um tipo de computação que está, atualmente, a ganhar impulso na indústria da computação em nuvem. Visto que promete poucos ou nenhuns custos de administração, uma elasticidade infinita e custos mínimos [3][4] é uma escolha recorrente, para o desenvolvimento de aplicações *cloud native* [6]. Apesar de inicialmente, popularizar-se com o modelo de implementação *Function-as-a-Service* (FaaS) [4], o seu conceito não se limita apenas às funções e também pode ser implementado como *Container-as-a-Service* (CaaS). No que lhe concerne, tais tecnologias que utilizam *containers* fornecem uma melhor capacidade de migração das aplicações tradicionais, maior autoridade e controlo sobre o ambiente de execução dos programas [5]. Este trabalho tem como objetivo, discutir a adoção de *serverless computing,* utilizando o modelo de implementação CaaS e elencar os principais serviços e *frameworks* utilizados na indústria, bem como avaliá-los de modo a obter as suas características. Para este fim, desenvolve-se uma avaliação qualitativa e quantitativa. Na avaliação qualitativa, são comparadas as plataformas Knative, Openwhisk, OpenFaaS, Azure Container Instances, Cloud Run, AWS Lambda e AWS Fargate. Na avaliação quantitativa, por um *benchmarking* foram avaliadas as plataformas em destaque, Knative, Cloud Run e AWS Fargate, através de sete testes, os quais analisam o comportamento das plataformas, medem o tempo de resposta e a percentagem de sucesso das requisições em relação ao aumento de tráfego. Após a execução da experiência, em que foi necessário provisionar a infraestrutura das plataformas, configurar os componentes responsáveis pelo *benchmarking*, executar requisições HTTP, medir os tempos de resposta e armazená-los para consulta e análise, apurou-se que a Cloud Run é a plataforma que apresenta melhor escalabilidade, em seguida a *Knative* e por último a AWS Fargate, e conclui-se que as plataformas *CaaS*, além de auxiliarem o desenvolvimento de aplicações escaláveis, apresentam ferramentas e soluções para os desafios do modelo de computação *serverless*.

Palavras-chave: *Cloud Computing, Serverless, Container-as-a-Service*, *Benchmark*, *Scalable Applications*

Abstract

Serverless Computing or serverless is a computing type that is currently gaining relevance in the cloud computing industry. Since it promises few or no administration costs, infinite elasticity, and minimal costs [3][4], it is a recurring choice for cloud-native application development [6]. Although initially popularized as Function-as-a-Service (FaaS) [4], this concept is not only related to functions but could also be implemented as Container-as-a-Service (CaaS). Such technologies that run containers present better migration capability from traditional applications and more authority and control over the program execution environment [5]. This work aims to discuss the adoption of serverless computing using the CaaS implementation model, list the main services and frameworks adopted in the industry and evaluate them to obtain their characteristics. For this purpose, two evaluations were developed: a qualitative evaluation and a quantitative experiment. The qualitative assessment compares the platforms Knative, Openwhisk, OpenFaaS, Azure Container Instances, Cloud Run, AWS Lambda, and AWS Fargate. In the quantitative evaluation, the highlighted platforms (Knative, Cloud Run, and AWS Fargate) were evaluated through seven benchmarking tests, which analyze the platforms' behavior, measured by the response time and percentage of success of the requests about the traffic increase. To run the experiment, it was necessary to provision the platforms' infrastructure, configure the components responsible for benchmarking, execute HTTP requests, measure response times, and store them for querying and analysis. After the experiment, the results indicate that Cloud Run is the platform that presents better scalability, in second place Knative and finally AWS Fargate, and it is concluded that CaaS platforms, in addition to helping the development of scalable applications, present tools, and solutions to handle the challenges of the serverless computing model.

Keywords: *Cloud Computing, Serverless, Container-as-a-Service*, *Benchmark*, *Scalable Applications*

Conteúdo

Acrónimos

ACI Azure Container Instances
API Application Programming Interface
AWS Amazon Web Services
BaaS Backend-as-a-Service
CaaS Container-as-a-Service
CD Continuous Deployment
CI Continuous Integration
CLI Command Line Interface
CMP Container Management Platform
CRDs Kubernetes Custom Resource Definitions
CRI Container Runtime Interface
DBaaS Database-as-a-Service
EBS Amazon Elastic Block Store
EC2 Amazon Elastic Compute Cloud
FaaS Function-as-a-Service
GCP Google Cloud Platform
GB Gigabyte
IaaS Infrastructure-as-a-Service
IaC Infrastructure as Code
IDE Ambiente de Desenvolvimento Integrado
IoT Internet of Things
I/O Entrada e Saída (*Input* e *Output*)
K8s Kubernetes
MB Megabyte
OS Sistema Operativo
PaaS Platform-as-a-Service
RPS Requisições por segundo
SaaS Software-as-a-Service
SDK Software Development Kit
SLA Acordos de Nível de Serviço
TI Tecnologia da Informação

UI Interface de Utilizador (*User Interface*)

USD United States Dollar

VM Máquina Virtual

VMs Máquinas Virtuais

VMM Monitor de Máquina Virtual

VU Utilizador virtual (*Virtual User*)

Lista de Figuras

Lista de Tabelas

Capítulo I — Introdução

Antes de lançarem um novo sistema, as empresas devem considerar quantos utilizadores podem aceder à aplicação e estimar a quantidade de computação, rede e armazenamento que esta consumirá, para garantir que o sistema esteja sempre disponível. Estimar recursos, na maioria das vezes, é uma tarefa difícil e imprevisível, pois a procura pode crescer mais do que a capacidade da organização em implementar serviços, para atender aos requisitos necessários.

Com o advento da computação em nuvem, responder ao aumento do fluxo de solicitações de uma aplicação tornou-se mais cómodo em comparação ao modelo de implementação tradicional; os desenvolvedores mudaram de máquinas físicas para máquinas virtuais e passaram a ter capacidade de dimensionar os recursos de um sistema de forma elástica, em resposta às mudanças na procura e conforme as políticas de escalonamento automático. Este novo modelo, conhecido como *Infrastructure-as-a-service*, permite uma redução significativa nos custos operacionais, porém ainda exige desenvolvimento e manutenção da equipa para gerir explicitamente as suas máquinas virtuais [11].

Apesar da capacidade de escalabilidade da computação em nuvem pública, algumas aplicações ainda têm problemas para atender à procura de picos anormais de solicitações. Por exemplo, é comum os sites de comércio eletrónico experimentarem baixas devido à quantidade de requisições na *Black Friday*, quando diversas lojas fazem grandes descontos, iniciando os períodos de compras de Natal. De acordo com o *Gremlin*, isso custa aos gigantes do retalho mais de 11 mil dólares em vendas por cada minuto de inatividade [12][13]. Tal problema pode estar ligado a diversos fatores, como sejam a presença de arquiteturas monolíticas [10], imprevisibilidade de desempenho e velocidade de escalonamento das máquinas [9].

Como solução a este desafio, a computação sem servidor, *serverless computing* ou simplesmente *serverless,* tem ganho popularidade e está a emergir como um novo paradigma para a implantação de aplicações em nuvem escaláveis [4], pois o fornecedor de infraestrutura assume as responsabilidades de receber solicitações de clientes e respondê-las, planear a capacidade, agendar tarefas e monitorizar operações; os desenvolvedores precisam preocupar-se apenas com a lógica de negócio das solicitações dos clientes [11].

Por sua vez, o termo *serverless* ficou conhecido em 2014 pela Amazon na sessão *re:Invent* e foi associado ao produto *AWS Lambda*; os outros fornecedores seguiram em 2016 com a introdução do *Google Cloud Functions*, *Microsoft Azure Functions* e *IBM OpenWhisk*, sendo por isso, comum associar *serverless* com o modelo de implementação *Function-as-a-Service* [4].

Entretanto, o conceito de *serverless* não se limita apenas às funções e têm surgido mais *frameworks* e produtos que utilizam *containers* e ainda são considerados como soluções *serverless*. Como grande indício desta adoção do modelo de implementação *Container-as-a-Service,* podemos citar o lançamento do suporte ao uso de *containers* no *AWS Lambda* no evento re:Invent 2020 em dezembro de 2020 [14].

Por fim, esta nova forma de implementação citada, os seus conceitos, os principais produtos e *frameworks* e também as suas vantagens e desvantagens são o assunto desta dissertação.

1.1 Objetivos

Esta dissertação tem como objetivo discutir sobre a adoção de *serverless computing* utilizando o modelo de implementação *Container-as-a-Service* e elencar os principais serviços e *frameworks* utilizados atualmente na indústria, bem como avaliá-los de modo a obter os seus benefícios e desvantagens.

1.2 Formulação do Problema

A dissertação pretende responder essencialmente às seguintes questões:

- O que são e como podem as *frameworks serverless* ajudar os engenheiros de informática a construírem aplicações escaláveis na nuvem?
- Quais as principais vantagens em utilizar *Container-as-a-Service* para implementar o modelo de computação *serverless?*
- No contexto das tecnologias que implementam o modelo *Container-as-a-Service*, quais são as principais plataformas disponíveis no mercado e as suas respetivas características? Entre estas, que plataformas têm maior potencial de escalabilidade?

1.3 Justificação

Como citado anteriormente, *serverless* facilita a criação de aplicações escaláveis na nuvem por abstrair a necessidade de gestão dos servidores e regras de escalonamento.

Em contrapartida, tal tecnologia também apresenta desvantagens, estando estas, como o próprio conceito, relacionadas com o uso das funções. Entre elas, podemos citar a falta de controlo do ambiente de execução, iniciações lentas (*cold start*), limites de tempo de execução e memória, demasiadas ligações às bases de dados e bloqueio do fornecedor, conhecido como *vendor lock-in* [1].

Na tentativa de solucionar alguns destes problemas, surgiram novas tecnologias *serverless* que utilizam *containers* e fornecem maior autoridade e opções de controlo sobre o ambiente de execução dos programas, que diminuem as dependências para com os provedores, permitindo assim uma maior portabilidade.

Estes novos *frameworks* buscam atingir um nicho que já está presente em 30% dos sistemas de informação e seguem uma tendência de mercado, onde o emprego de *containers* em ambientes de produção subiu 300% desde 2016 [6].

Para além da ampla utilização na indústria, podem-se citar também trabalhos académicos como *Serverless Containers — rising viable approach to Scientific Workflows* [15] e *Serverless computing for container-based architectures* [16] que propõem o uso de *containers* em arquiteturas *serverless* para o processamento de grandes cargas de trabalho de aplicações científicas e de visão computacional, respetivamente.

Em resumo, os supracitados fatos justificam e ressaltam a importância do desenvolvimento deste trabalho que tem o propósito de abordar o uso dos *containers* em ferramentas *serverless.*

1.4 Estrutura da Dissertação

Neste capítulo, aborda-se a apresentação temática, objetivos, formulação do problema e justificações do presente trabalho. E ainda a presente estrutura da dissertação.

No *Capítulo II*, encontra-se o referencial teórico que fomenta as ciências ou tecnologias abordadas durante a investigação.

O *Capítulo III* foca-se em apresentar a metodologia, a definição e o resultado de uma comparação qualitativa das plataformas propostas na metodologia. Também define uma experiência quantitativa, que será executada posteriormente.

O *Capítulo IV* exibe a implementação, execução e os resultados da experiência de *benchmarking* tencionada.

O *Capítulo V* apresenta conclusões sobre o trabalho e propostas de futuros desenvolvimentos possíveis para a presente dissertação.

Capítulo II — Referencial Teórico

2.1 Virtualização

Na computação, virtualização é o ato de criar uma versão virtual de algo, incluindo plataformas de *hardware*, dispositivos de armazenamento e recursos de rede de computadores. Isto permite às organizações fragmentarem um único computador ou servidor físico, em várias máquinas virtuais.

A possibilidade de executar várias instâncias virtualizadas num servidor, fornece benefícios como a escalabilidade rápida e a melhor utilização dos recursos. Além disso, pode-se citar como a principal vantagem da virtualização a redução de custos, pois apenas com um servidor físico é possível montar uma estrutura com vários servidores virtualizados, dispensando a compra de equipamentos desnecessários. Operacionalmente, pode afirmar-se que também existem várias vantagens, pois, a administração torna-se totalmente centralizada e o tempo total de manutenção de um equipamento virtualizado é menor, quando comparado com um equipamento comum [19].

Uma das principais questões a respeito da virtualização é sobre a maturidade da tecnologia, já que se acredita ser recente; contudo apesar de ter ganho popularidade nos últimos anos, a sua origem é da década de 1960 e o conceito foi desenvolvido pela IBM. O foco principal, consistia em executar várias máquinas virtuais num único *hardware*, reduzindo assim os custos com equipamentos, os quais eram muito altos na época [20].

Descrevem-se, a seguir, as duas tecnologias de virtualização mais populares. Estas são baseadas em hipervisor e em *container* (também chamada de virtualização a nível do sistema operativo).

2.1.1 Virtualização baseada em Hipervisor

Na última década, a virtualização baseada em hipervisor tem-se estabelecido como um método popular para implementação de máquinas virtuais. Uma máquina virtual, em inglês *virtual machine* (VM), pode ser definida como uma duplicata isolada e eficiente da máquina real e simula os recursos de *hardware* que dão suporte ao sistema operativo (OS).

O hipervisor é um *software* responsável por criar e executar máquinas virtuais (VMs). Também chamado de monitor da máquina virtual, em inglês *virtual machine monitor*

(VMM), ele isola o sistema operativo do hipervisor e os recursos das máquinas virtuais e permite a criação e a gestão dessas máquinas. O *hardware* físico ou o sistema operativo onde o hipervisor está instalado é denominado de *host*, enquanto as diversas máquinas virtuais que utilizam os seus recursos são denominadas de *guests*.

Para executar VMs, todos os hipervisores precisam de alguns componentes a nível do sistema operativo, como gestor de memória, programador de processos, *stack* de entrada e saída, *drivers* de dispositivo, gestor de segurança, um *stack* de rede, entre outros. Além disso, os recursos como CPU, dispositivos de rede, memória e armazenamento são geridos como um *pool* que pode ser realocado com facilidade entre os *guests* existentes ou para novas máquinas virtuais [23].

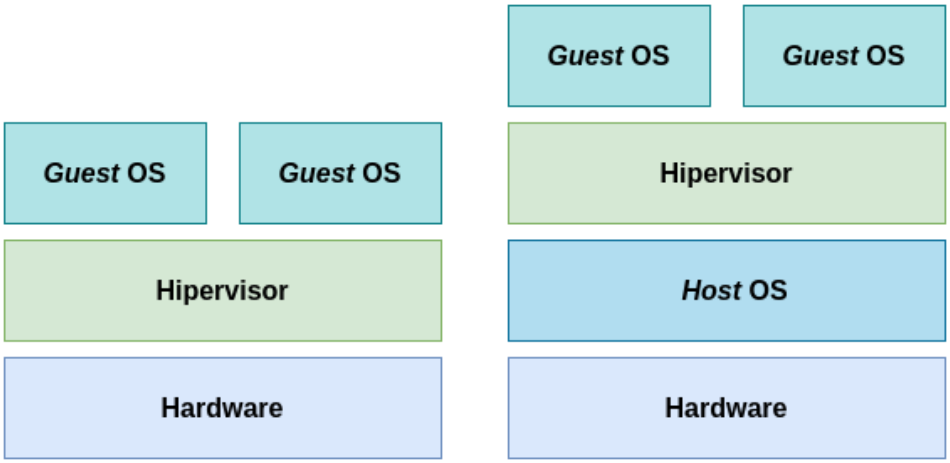
Ainda sobre hipervisores, estes podem ser classificados em dois tipos:

- Tipo I

Também são chamados de "hipervisor nativo" ou "*bare metal*", pois são executados diretamente na parte superior do *hardware* subjacente, conforme exemplificado na *Figura 2.1 a*. Neste caso, o VMM, além de gerir os *guests*, fornece *drivers* de dispositivo que o sistema convidado usa para aceder diretamente o *hardwar*e. São exemplos de hipervisores Tipo I: Microsoft Hyper-V, VMware ESX e Xen.

- Tipo II

Esses hipervisores, são executados como um aplicativo num sistema operativo normal. O sistema operativo *host* não tem qualquer conhecimento sobre o VMM Tipo II; trata-o como qualquer outro aplicativo, executando os pedidos de entrada e saída (E/S), conforme exemplificado na *Figura 2.2 b*. Por sua vez, as solicitações dos sistemas operativos *guest* são intercetadas pelo *host*, executadas pelos respetivos *drivers* e roteadas de volta ao solicitante [24]. São exemplos de hipervisores Tipo II: Oracle VirtualBox e VMWare Workstation.



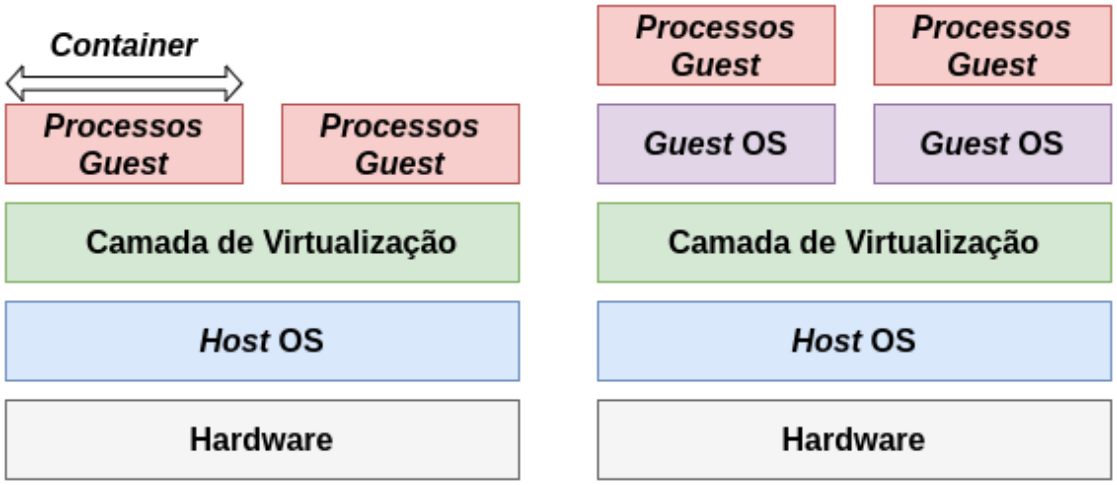
a) Tipo I b) Tipo II

Figura 2.1: Representação dos dois tipos de hipervisores. Fonte (Autor)

2.1.2 Virtualização baseada em Containers

Uma forma mais leve de virtualização, quando comparada aos hipervisores, é a virtualização baseada em *container* ou virtualização de nível de sistema operativo. Neste tipo de virtualização, os recursos de um sistema operativo são divididos ou compartilhados, criando ambientes chamados de *containers* em múltiplas instâncias de espaço de utilizador que estão isoladas e são executadas no topo do mesmo *kernel* do sistema operativo. Assim, os *containers* fornecem uma abstração do *kernel* do sistema operativo, permitindo que vários processos convidados sejam executados, num *container* isolado de outros *containers*.

Conforme exemplificado na *Figura 2.2,* cada *container* comporta-se como um sistema operativo independente e com o seu próprio conjunto de processos, sem a necessidade de uma camada intermediária.



a) Virtualização de *Containers* b) Virtualização de Hipervisores

Figura 2.2: Comparação entre Virtualização baseada em *Containers* e Virtualização baseada em Hipervisores. Fonte (Autor)

O isolamento de ambiente a nível de sistema operativo, é normalmente efetuado por *namespaces* do *kernel*, um recurso do *kernel* Linux, que permite que diferentes processos tenham uma visão diferente do sistema. Como os *containers* não devem ser capazes de interagir com elementos externos, muitos recursos globais são agrupados numa camada de *namespace,* que fornece a ilusão de que o *container* é seu próprio sistema. Por outro lado, a gestão de recursos em sistemas de virtualização baseados em *container* é normalmente feita por grupos de controle (*cgroup*), o que restringe o uso de recursos por grupos de processos. Por exemplo, usando *cgroups*, é possível limitar e priorizar o consumo de CPU, memória e uso de E/S para diferentes *containers*. Em certos casos, alguns sistemas utilizam as suas próprias implementações, para realizar a gestão de recursos devido à incompatibilidade com cgroups [25].

2.1.3 Comparação entre *Containers* e Máquinas Virtuais

Os *containers*, são por vezes vistos como substitutos dos hipervisores. Mas isso não é uma verdade absoluta, e não é sempre possível substituir máquinas virtuais por *containers*, já que os *containers* e a virtualização de máquinas atendem a necessidades diferentes. De forma geral, *containers* e VMs são similares. Ambos são ambientes de computação empacotados, que combinam vários componentes e os isolam do restante sistema. Mas também existem muitas diferenças importantes,

como a portabilidade e escalabilidade, fatos estes que influenciam, diretamente, a forma como estas tecnologias são utilizadas. A seguir, na *Tabela 2.1*, encontra-se um resumo das principais diferenças entre os dois tipos de virtualização.

Característica	**Containers**	**Máquinas Virtuais**
Guest OS	*C*ompartilham o mesmo sistema operativo e *kernel*. A imagem é carregada na memória física.	São executados em *hardware* virtual e o *kernel* é carregado na sua própria região de memória.
Comunicação	Mecanismos padrão de IPC como *signals*, *pipes*, *sockets*, etc.	Através de dispositivos Ethernet.
Segurança	O controlo de acesso obrigatório pode ser aproveitado.	Depende da implementação do Hypervisor.
Performance	Fornecem desempenho quase nativo em comparação com o *host* OS subjacente.	Sofrem pequena sobrecarga, pois as instruções da máquina são traduzidas do sistema operativo *guest* para o *host*.
Isolamento	Os subdiretórios podem ser montados de forma transparente e podem ser compartilhados.	Compartilhamento de bibliotecas, arquivos, etc. entre convidados e entre *hosts* convidados não é possível.

Tempo de Iniciação	Podem ser inicializados em alguns segundos.	Levam alguns minutos para inicializar.
Armazenamento	Os *containers* ocupam menos espaço de armazenamento, pois o sistema operativo base é compartilhado.	Ocupam mais armazenamento uma vez que todo o sistema operativo *kernel* e seus programas associados devem ser instalados e executados.

Tabela 2.1: Comparação entre *Containers* e máquinas virtuais [22].

2.2 Computação em Nuvem

A computação em nuvem está progressivamente a ser mais utilizada, e é apontada como a principal tecnologia dos próximos anos. A ascensão do serviço de nuvem possibilitou a pequenas empresas possuírem a mesma tecnologia e capacidade de crescimento de grandes empresas, e tem gerado mais valor no mercado.

Esse novo paradigma de computação em nuvem fornece um *pool* de recursos escaláveis, dinâmicos e de acordo com a procura, que tem como princípio oferecer computação, rede, armazenamento e serviços no modelo em que só paga pelos recursos que são consumidos e são obtidos de maneira *self-service*.

2.2.1 Características

Em resumo, o principal objetivo da computação em nuvem é fazer um melhor uso de recursos distribuídos, combinando-os para alcançar maior rendimento e ser capaz de resolver problemas de computação em grande escala. A computação em nuvem lida com virtualização, escalabilidade, interoperabilidade, qualidade de serviço e entrega modelos de nuvem, nomeadamente privados, públicos e híbridos [27].

Esse tipo de computação tem grandes vantagens em relação aos modelos *on-premise*, pois, além de fornecer recursos a um menor preço, vende soluções e compartilha as responsabilidades de gestão entre empresas. A seguir, um resumo das principais características do modelo [28]:

- Virtualização

Capacidade de virtualizar partes num conjunto único de recursos, é fornecida pela nuvem e é a base deste modelo computacional. Como explicado na *Seção 2.1,* a possibilidade de executar várias instâncias virtualizadas de forma centralizada oferece benefícios como a rápida escalabilidade, uma melhor utilização dos recursos e administração. Ambos os recursos de *software* (por exemplo, aplicativos) e *hardware* (armazenamento, rede, memória, processadores e mais) são cobertos por esta virtualização.

- Poder de processamento

Em ambientes baseados em nuvem o poder de processamento é centralizado para fornecer um eficiente acesso ao *pool* de recursos, com a menor taxa de energia e custo.

- Armazenamento

Um dos recursos computacionais básicos que é fornecido pelo provedor de serviços em nuvem é o armazenamento. Este armazenamento hospeda vários recursos, como aplicativos e dados.

- Conectividade

Os serviços baseados em nuvem apresentam dois tipos de conectividade: o primeiro está relacionado a cada camada de recursos numa nuvem, e o segundo é estabelecido entre utilizadores de um serviço em nuvem pela Internet.

- Compartilhamento

Apesar de compartilhar de uma infraestrutura de provedor de nuvem, nos ambientes de computação em nuvem não existe um compartilhamento real dos recursos, devido ao isolamento fornecido através da virtualização. Na verdade, os recursos são atribuídos separadamente ou existe um modelo de serviços para cada cliente.

- *Pool* de recursos

Estabelecer um *pool* de recursos é o principal objetivo de uma nuvem. Estes recursos podem ser armazenamento, infraestrutura, dados, aplicativo, plataforma ou serviços.

- Serviços sob demanda

De acordo com o modelo de computação em nuvem, os recursos de TI devem ser provisionados, reservados ou libertados pelos clientes conforme a sua própria indicação e necessidade.

- Elasticidade

A atribuição de recursos do *pool* para os clientes, tem de ser criada dinamicamente numa nuvem. Portanto, de acordo com a procura, os clientes podem remover ou adicionar uma maior capacidade de computação, para obter uma melhor utilização dos mesmos, nomeadamente, uma melhor capacidade de resposta e economia.

- Isolamento

Os ambientes baseados em nuvem, isolam um serviço dos demais provendo segurança e desempenho. Essa capacidade, deverá ser fornecida por meio do estabelecimento físico do provedor de infraestrutura e por definições de *software*. Garantindo desta forma a integridade, confidencialidade e disponibilidade dos recursos dos clientes.

- Distribuição

Ambientes de computação em nuvem, principalmente computação em nuvem pública, são globalmente distribuídos e possuem vários recursos ou consumidores em diferentes regiões. Assim, os sistemas baseados em nuvem têm de ser decompostos em componentes distintos, para fornecer a capacidade resposta almejada.

2.2.2 Modelos de implementação

Com tantas características positivas, quando comparadas com o modelo tradicional, é comum observar-se que as empresas precisam de adotar e implementar, soluções na nuvem para suportar os seus serviços. Por exemplo, se uma empresa nova no mercado tenciona desenvolver um aplicativo que tem um número de acessos muito instável e suscetível de grandes crescimentos, uma solução viável é a adoção de computação em nuvem. Pois a flexibilidade de aumentar ou diminuir o *pool* de recursos de TI para serviços dinâmicos e a melhor distribuição de custo CAPEX e OPEX para a indústria tornam-se vantagens competitivas [29].

Entretanto, para a implementação deste modelo de computação, tratando-se de uma solução de nuvem de uma organização, é necessário saber qual o modelo de implementação que melhor se adequa à realidade da empresa. Seguem-se os quatro principais modelos existentes para computação em nuvem:

- Privada

Uma nuvem privada, atende às necessidades de um único negócio ou empresa. É uma implementação de *datacenter* próprio e exclusivo de uma empresa e oferece todos os benefícios da nuvem pública, tais como a flexibilidade, escalabilidade,

aprovisionamento, automação e monitorização. Por exemplo, pode ser implementada internamente para atender diversas filiais.

- Pública

A nuvem pública, é definida como uma série de serviços de computação oferecidos por terceiros à Internet pública, os quais são disponibilizados a qualquer pessoa que queira utilizá-los ou comprá-los. Eles podem ser gratuitos ou vendidos sob pedido, permitindo que os clientes paguem apenas pelo seu consumo de ciclos de CPU, armazenamento ou largura de banda. Ao contrário das nuvens privadas, as nuvens públicas podem poupar as empresas dos enormes gastos de compra, gestão e manutenção e hardware local e infraestrutura de aplicação [29].

- Híbrida

Uma nuvem híbrida, é uma infraestrutura que inclui aplicações dos dois modelos anteriores. Apesar dos recursos estarem ligados neste modelo, estes permanecem em entidades únicas, o que permite que uma nuvem híbrida ofereça os benefícios de vários modelos de implantação de uma só vez. Por exemplo, uma nuvem híbrida pode oferecer a escalabilidade de CPUs da nuvem pública e o *compliance* de dados de uma nuvem privada. As dificuldades e responsabilidades, são partilhadas entre as equipas que implementaram a solução.

- Comunitária

Várias organizações constroem em conjunto e compartilham a mesma infraestrutura de nuvem, bem como políticas, requisitos, valores e preocupações. A comunidade da nuvem forma-se num grau de escalabilidade económica e equilíbrio democrático. A infraestrutura, pode ser hospedada por um fornecedor externo ou dentro de uma das organizações na comunidade.

2.2.3 Modelos de serviço

A nuvem, é um conceito muito amplo e abrange quase todos os tipos possíveis de serviços *online*, mas quando as empresas se referem à aquisição em nuvem, geralmente há três modelos principais de serviço em nuvem em consideração: Software *as a Service* (SaaS), *Platform as a Service* (PaaS) e *as Infrastructure as a Service* (IaaS). A principal diferença entre estes modelos é a divisão de responsabilidades de gestão entre o provedor e a organização. Como podemos observar na *Figura 2.3,* iniciando do modelo tradicional (*on-premise*), à esquerda, com total responsabilidade de gestão da organização (representado pela cor azul) e

com crescente responsabilidade compartilhada com o provedor (representado pela cor verde), para a direita.

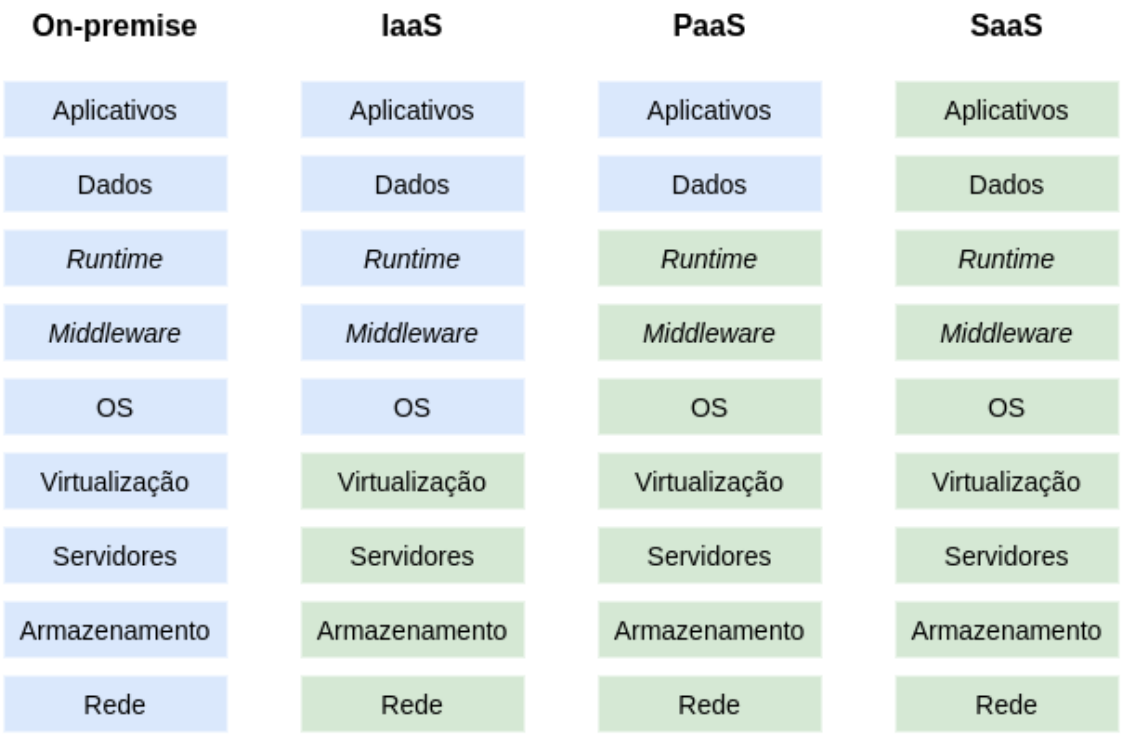


Figura 2.3: Principais modelos de implementação da computação em nuvem [49].

Segue a definição dos principais modelos de serviço:

- *Infrastructure-as-a-Service*

IaaS é o nível mais básico de soluções baseadas em nuvem. Neste modelo, a nuvem fornece aos utilizadores acesso a recursos de computação, como servidores, armazenamento e rede. Neste modelo, o cliente possui mais controlo e é responsável pela gestão dos recursos. Não é possível inferir que deter um maior controlo sobre os recursos das máquinas influencia o desempenho das operações realizadas pelas mesmas, no entanto, deter o controle de VMs, *containers* e orquestradores, é uma maneira de alcançar arquiteturas e configurações mais ajustadas ou voltadas para escalabilidade e alta disponibilidade.

- *Platform-as-a-Service*

Fornece um ambiente de hospedagem gerida, onde se podem implantar aplicações sem precisar gerir VMs ou recursos de rede. Este modelo permite que os desenvolvedores implementem aplicações e as hospedem na nuvem, fornecendo facilmente o balanceamento de carga, escalabilidade e alta disponibilidade, sem os custos de gestão e manutenção dos seus próprios ambientes.

- *Software-as-a-Service*

É um modelo de entrega de aplicações de *software* na Internet, no qual os provedores de nuvem hospedam e gerem as aplicações de software, sem a necessidade do utilizador as instalar localmente.

2.2.4 Modelos de serviço *serverless*

Para além dos mais tradicionais modelos de serviço, com o avançar dos anos podemos observar uma tendência, em que crescentemente, estas nomenclaturas têm sido usadas para denominar novos modelos ou até mesmo produtos, devido à grande popularidade no meio de negócio. Quanto aos novos modelos de computação em nuvem, podemos citar *Functions as a Service* (FaaS), *Containers as a Service* (CaaS) e Backend *as a Service* (BaaS).

Como indicado no *Capítulo I,* este trabalho objetiva abordar as tecnologias *serverless* que utilizam a implementação do modelo CaaS. Para isso, é necessário definir este conceito. Também citado no *Capítulo I,* o modelo de serviço de FaaS, foi um dos primeiros, desconsiderando os tradicionais (anteriormente abordados), que tiveram popularidade no mercado e também foi o primeiro a ser associado ao conceito de *serverless.* Por fim, BaaS é outro modelo que tem sido implementado pelas principais provedoras de computação em nuvem e também está associado ao conceito serverless. A seguir, uma definição mais detalhada acerca dos modelos supracitados:

- *Functions-as-a-Service*

No seu núcleo, o modelo FaaS é capaz de executar um código ou função de aplicação personalizado, para responder a uma solicitação de rede. Além disso, oferece suporte à multilocação de várias funções em execução simultaneamente, garantindo o seu isolamento de um para o outro e, finalmente, uma plataforma FaaS é escalonada conforme necessário, ou seja, deve ter a capacidade de processar muitas solicitações em paralelo [26]. Com o FaaS, todos os recursos são geridos automaticamente, pelo seu provedor de serviços em nuvem, permitindo que os desenvolvedores se preocupem, exclusivamente, em escrever funções individuais no código da aplicação.

- *Container-as-a-service*

CaaS é um modelo de serviço em nuvem que permite que desenvolvedores de *software* e departamentos de TI carreguem, organizem, executem, dimensionem e giram *containers* usando virtualização baseada em *container*, abordado na *Secção 2.1.2*. Este modelo é considerado um subconjunto da infraestrutura como serviço e é encontrado entre IaaS e PaaS. O mesmo possui em comum todas as características dos serviços em computação em nuvem implementados pelo modelo IaaS e em adição apresenta as mais valias da utilização de *containers* em comparação com VMs, encontrados na *Tabela 2.1*.

- *Backend-as-a-service*

BaaS, é um modelo de serviço em nuvem, no qual os desenvolvedores terceirizam requisitos de uma aplicação da web ou móvel, para poupar nos custos de desenvolvimento e manutenção. Os fornecedores de BaaS fornecem *software* pré-escrito para atividades comuns à maioria dos sistemas como autenticação de utilizador, gestão de banco de dados, atualização remota, notificações *push*, bem como armazenamento e hospedagem em nuvem, etc. Dentro da presente categoria, também é possível encontrar na literatura outros modelos de serviço populares como *Database as a Service* (DBaaS). Por sua vez, DBaaS é um serviço de banco de dados gerido, onde o provedor de nuvem é o responsável pela gestão, que vai desde atualizações periódicas a *backups*, para garantir que o sistema de banco de dados permaneça disponível, seguro e, em alguns casos, escalável [31]. Em resumo, DBaaS é considerado parte do conjunto de serviços oferecidos pelo modelo BaaS pois o objetivo é o mesmo, oferecer produtividade e menores custos de manutenção para as empresas.

2.2.5 Métodos de Dimensionamento

A escalabilidade, é definida como a capacidade de um recurso aumentar ou diminuir de acordo com a procura. Essa característica, é um dos recursos mais populares e benéficos da computação em nuvem, pois as empresas podem aumentar ou diminuir os seus recursos para atender às procuras baseadas em estação, projetos, ambiente, crescimento, entre outros.

Existem algumas maneiras principais de escalar os recursos na nuvem:

- *Scale-up* e *Scale-down*

Esta técnica de expansão é implementada com a adição ou remoção de recursos numa única máquina, tornando-a mais poderosa, equipando-a com um melhor processador, memória, rede ou armazenamento. Por exemplo, uma máquina virtual pode melhorar o processamento adicionando mais núcleos à máquina. Sendo esta uma maneira rápida de dar mais capacidade ao sistema, mas também é limitada e cara.

- *Scale-out* e *Scale-in*

Esta técnica permite adicionar ou reduzir a quantidade de máquinas ou nós de processamento do mesmo tipo num sistema, com base numa métrica escolhida. A tarefa é dividida por um mestre, geralmente um balanceador de carga, para os nós. Esta técnica tem um grande limite de processamento, mas dependendo dos módulos de hospedagem, o tempo para iniciar um nó pode ser um problema, por exemplo, como indicado na *Tabela 2.1*, o tempo de escalonamento de *containers* é mais rápido do que VMs.

- *Mixed scale*

Esta técnica permite ampliar ou reduzir os tipos de instâncias e ajustar a quantidade de instâncias ao mesmo tempo, por outras palavras, é a combinação das duas técnicas anteriores. O uso desta técnica em sistemas heterogéneos pode ser uma vantagem, para melhorar a qualidade e a eficiência dos serviços.

2.2.6 Dificuldades na Adoção

Apesar das grandes vantagens da *cloud computing*, nem todas as empresas têm migrado, facilmente, as suas aplicações para os provedores de nuvem. A seguir alguns exemplos de dificuldades que as empresas têm, em adotar esta relativamente nova tecnologia [27, 30]:

- Segurança

Organizações temem deixar os seus dados disponíveis de forma remota e na responsabilidade de terceiros. Além disso, a computação em nuvem trouxe desafios e técnicas de ataques que até então eram desconhecidas pela empresa adotante, dificultando o mapeamento e implementação de políticas de defesa.

- *Compliance*

Além da segurança, as empresas têm dificuldades com as regulamentações e localização dos dados. A regulamentação está relacionada com a região e o

segmento de negócio de cada empresa e os provedores devem estar em conformidade com os padrões exigidos.

- Modelo de custo

A mudança no modelo de custo, apesar de benéfica, é uma tarefa complexa de ser planeada e requer um esforço da organização para se adequar a novos processos deste modelo.

- Complexidade

É necessária uma mudança não só onde as aplicações vão ser executadas, mas também exige uma modificação de processos, refatoração aplicacional e planeamento de migrações.

- Competências

Requer o desenvolvimento de novas competências e profissionais qualificados para as funções, algo que é difícil no mercado, dada a pequena maturidade da tecnologia.

- Controlo

A infraestrutura, é desenhada e suportada exclusivamente pelo provedor de serviços, num modelo de compartilhamento de responsabilidades. A falta de acesso a toda a solução pode ser uma barreira para algumas empresas.

- *Service Level Agreement*

Normalmente, os contratos entre provedor e organizações possuem Acordos de Nível de Serviço (SLA). Estes acordos devem ser definidos de tal forma, que, tenham um nível adequado de granularidade, para que possam abranger a maioria das expectativas do consumidor e sejam relativamente simples de serem ponderados, verificados, avaliados e aplicados pelo mecanismo de atribuição de recursos na nuvem.

- Migração

Por fim, no caso de empresas que possuem serviços em execução que precisam de mais recursos, é necessária uma adequação ou migração dos serviços. Esta migração, muitas vezes, é complexa e possui riscos e custos que podem limitar a adoção do modelo em *cloud*. Adicionalmente, em casos complexos, a adoção do modelo de implementação híbrido, ver *Secção 2.2.1,* pode ser uma alternativa mais viável.

2.3 *Serverless*

A computação *serverless*, é um modelo de serviço de computação em *cloud*, altamente escalável, que permite a execução de aplicações, sem que a equipe de TI tenha de provisionar ou gerir os servidores e os recursos computacionais necessários. Os mesmos, apenas têm que criar e fazer a entrega do artefacto que contém a lógica das suas aplicações na plataforma, e o provedor de *serverless* encarrega-se de tudo o resto, garantindo que a plataforma é responsiva e consegue escalar de forma automática.

O termo "*serverless",* "sem servidor" em português, pode dar ideia de que não existem servidores, contudo esta noção está errada. Continua a existir uma infraestrutura e um ambiente de execução, que suporta o modelo que permite o armazenamento e a execução, só que esta infraestrutura é totalmente provisionada, gerida e mantida pelo provedor.

Adicionalmente, e dependendo da plataforma, caso o serviço não tenha nenhuma requisição numa janela de tempo, o sistema pode fazer o *scale-in,* ver Seção 2.2.4, para zero instâncias, desta forma não consumindo nenhum recurso de CPU e memória. Neste caso, o termo "*serverless"* possui mais sentido considerando o significado literal. Vale ressaltar que o termo se tornou o nome técnico na indústria de TI e não está necessariamente ligado a uma característica da tecnologia.

Serverless e *Functions as a Service* são frequentemente confundidos, FaaS é na verdade um subconjunto do modelo. *Serverless* está focado em qualquer categoria de serviço, seja computação, armazenamento, banco de dados, mensagens, *gateways* de API, etc. onde a configuração, gestão e faturação de servidores são invisíveis para o utilizador final [11]. FaaS, embora talvez seja a tecnologia mais central destas arquiteturas, é focado no paradigma de computação orientado a eventos em que o código da aplicação, ou *containers*, só é executado em resposta a eventos ou solicitações [32].

O presente trabalho, debruça-se sobretudo nas tecnologias *serverless* que oferecem recursos de computação (memória, CPU e ambiente de execução) e que implementam serviços por meio de *containers.* Apenas, como exemplo, há também *frameworks serverless* que oferecem os tipos de recursos: *networkin*g, armazenamento, autenticação, etc.

2.3.1 Arquitetura

Apesar da dificuldade em definir o termo e o modelo de computação *serverless* para serviços de computação, o seu requisito principal é objetivo: processamento de eventos. Tomando como exemplo o caso de um sistema que implementa o modelo de serviço de FaaS, conforme ilustrado na *Figura 2.3*. O serviço deve gerir um conjunto de funções definidas pelo utilizador, obter um evento enviado por HTTP ou recebido de uma fonte de evento, determinar para quais funções enviar o evento, encontrar ou criar uma instância para processamento, enviar o evento para a instância, esperar por uma resposta, recolher *logs* de execução, disponibilizar a resposta para o requerente e parar a instância quando não é mais necessária.

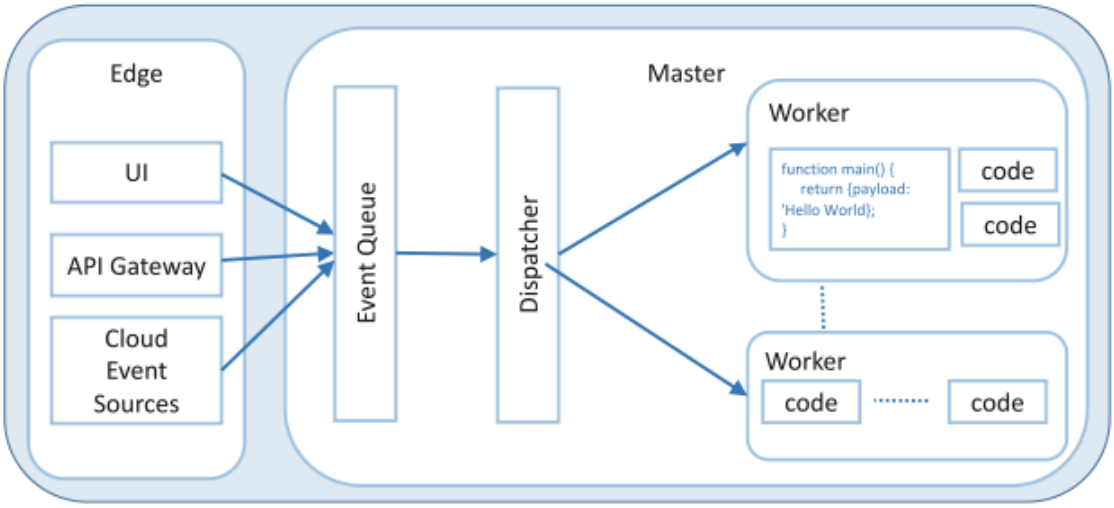


Figura 2.4: Arquitetura de plataforma FaaS de alto nível. Fonte (Autor)

2.3.2 Características

Além do objetivo principal, o processamento de eventos, e com o foco em adotar uma plataforma *serverless,* as equipas de TI deverão estar cientes das demais propriedades do modelo *serverless*. Desta forma, nesta subsecção são referidas as principais características desta tecnologia [4] em relação aos recursos de computação:

- Custo

Normalmente, o uso é medido e os utilizadores pagam apenas pelo tempo que os recursos foram usados, ou seja, quando as funções ou *containers* sem servidor estão em execução. Esta capacidade de escalar para zero instâncias é um dos principais diferenciais de uma plataforma sem servidor. Os recursos medidos, como memória

ou CPU, e o modelo de preços, como descontos fora do horário de pico, variam entre os fornecedores.

- Desempenho e limites

Há uma variedade de limites definidos nos requisitos de recursos de tempo de execução do código sem servidor, incluindo o número de solicitações simultâneas e os recursos máximos de memória e CPU disponíveis para uma chamada requisição. Alguns limites podem ser aumentados, quando as necessidades dos utilizadores aumentam, como o tempo de execução, enquanto outros são inerentes às plataformas, como o tamanho máximo da memória.

- Linguagens de programação

Os serviços sem servidor, oferecem suporte a uma ampla variedade de linguagens de programação, incluindo Javascript, Java, Python, Go, C # e Swift. A maioria das plataformas oferece suporte a várias linguagens de programação. Algumas das plataformas também suportam mecanismos de extensibilidade para código escrito em qualquer linguagem, desde que seja empacotado numa imagem Docker que ofereça suporte a uma API bem definida.

- Modelo de programação em partes

Atualmente, as plataformas *serverless*, normalmente, executam em várias pequenas funções ou pequenos serviços (microsserviços) toda a lógica de aplicação. Desta forma, os sistemas são altamente decompostos em unidades lógicas e de fácil escalonamento e fraca dependendência.

- Compossibilidade

As plataformas, geralmente, oferecem alguma maneira de invocar métodos simples, mas algumas plataformas fornecem mecanismos de nível mais alto, para compor esses métodos e podem facilitar a construção de aplicações mais complexas.

- Implementação abstraída

As plataformas esforçam-se para tornar a implementação o mais simples possível. Normalmente, os desenvolvedores só precisam fornecer um arquivo com o código-fonte da função. Além disso, há muitas opções onde o código pode ser empacotado como um arquivo com vários arquivos dentro ou como uma imagem Docker com código binário. Da mesma forma, há facilidades para criar versões ou agrupar APIs.

- Segurança e contabilidade

As plataformas sem servidor, são multi-locatárias e isolam o ambiente de execução dos e entre os utilizadores. Além disso, costumam ter definições de segurança rígidas, como autenticação e uso de HTTPS nativos.

- Monitorização e depuração

Cada plataforma, suporta a depuração básica usando instruções de impressão que são registadas nos logs de execução. Recursos adicionais, podem ser fornecidos para ajudar os desenvolvedores a rastrear erros e a compreender melhor as circunstâncias da execução da função.

2.3.3 Porquê, quando e como usar serverless?

Ainda que apresentados um exemplo e as características de arquitetura do modelo computacional em foco nas subseções anteriores, além de implementar arquiteturas orientadas a eventos, não são claros os motivos e casos de uso que podem ser resolvidos pela computação *serverless.* Por isso, a presente subsecção tem como objetivo elencar fatores que motivam a utilização, bem como casos de usos da tecnologia em questão.

Motivos

Na literatura, é comum perceber os principais benefícios da adoção de *serverless*, entre eles: menos tempo investido em operação e manutenção, desenvolvimento mais rápido devido ao uso de plataformas de BaaS, escalabilidade quase infinita e menores custos de infraestrutura. No entanto, nem todos esses benefícios são válidos em geral. Por exemplo, alguns trabalhos defendem que o modelo não é o estado da arte para menores custos, ou que não é claro que o desenvolvimento é mais fácil, por conta da pouca maturidade e da falta de ferramentas.

Desta forma, para entender os motivos dos profissionais optarem por adotar o *serverless*, foi investigado no artigo "*Serverless Applications: Why, When, and How?"* [33] um conjunto de aplicações e literaturas de diferentes fontes, para chegar a uma conclusão mais clara. Segundo a pesquisa, 47% das fontes optaram pela tecnologia, por conta da redução dos custos com infraestrutura, 34% escolhem esse tipo de plataforma em razão da capacidade nativa de escalonamento, também 34% alegaram escolher *serverless* para diminuir os custos de operação e por fim, em menores proporções, foram citadas as razões de melhor performance e rápido desenvolvimento, respetivamente 19% e 13%.

Os resultados anteriores, estão de acordo com outra pesquisa na comunidade em "*Serverless Community Survey 2020*" [51], com mais de 160 participantes, que relatam que os impactos positivos da escolha de uma arquitetura *serverless* são a adoção de uma arquitetura orientada a eventos (51%), menor custo dos recursos (44%), velocidade de desenvolvimento (36%), flexibilidade de escala (31%) e desempenho das aplicações (19%).

Casos de uso

Outro aspeto fundamental, para se perceber as mais valias em utilizar *serverless* são as suas implementações ou casos de usos. Como demonstrado anteriormente, o modelo tem o objetivo principal de responder a eventos, desta forma é amplamente utilizado em arquiteturas que seguem esse paradigma.

Além do processamento de eventos, por meio dos serviços de BaaS como provedores de autenticação, *API Gateways* e base de dados, é possível compor e gerir APIs HTTP, protocolo utilizado em grande parte dos servidores. Também, mais recentemente, estudos sugerem a utilização de *serverless* para a execução de *workloads* científicos, na indústria de IoT e para execução de *workflows* [34].

Por fim, é importante citar que ao implementar uma aplicação que utiliza *serverless*, os engenheiros enfrentam várias decisões, tais como selecionar o provedor em nuvem, a linguagem de programação, opções de *Backend-as-a-Service* e o nível de granularidade apropriado dos serviços que implementam as regras de negócio. Dentre essas, destaca-se a ampla utilização de serviços BaaS associados às soluções como *storage*, *database*, *messaging* e *workflows*; plataformas pertencentes a provedores de computação em nuvem pública e as linguagens de programação Python e JavaScript [33].

2.3.4 Desafios

Contudo, apesar das mais valias apresentadas na subsecção anterior, os sistemas *serverless*, como qualquer outro, apresentam as suas desvantagens ou dificuldades. Uma lista de algumas [4]:

- Desafios a nível do sistema
    - Custo — O custo é um desafio fundamental. Isso inclui minimizar o uso de recursos de uma função sem servidor, quando está em execução ou quando está ociosa. Outro aspeto é o modelo de precificação, incluindo

como o mesmo se compara a outras abordagens de computação em nuvem. Por exemplo, as funções *serverless*, são atualmente mais económicas para cálculos vinculados à CPU, enquanto as funções vinculadas a I/O podem ser mais baratas em VMs ou *containers* dedicados.

- Inicialização a frio (*cold start*) — Um dos principais diferenciais de *serverless* é a capacidade de escalar até zero ou não cobrar aos clientes o tempo ocioso. O dimensionamento para zero, entretanto, leva ao problema de inicialização a frio e ao pagamento da penalidade de deixar o código sem servidor pronto para execução. As técnicas para minimizar o problema de inicialização a frio e ainda escalar para zero são críticas.
- Limites de recursos — Os limites de recursos, são necessários para garantir que a plataforma possa lidar com picos de carga e gerir ataques. Os limites de recursos aplicáveis numa função *serverless* incluem memória, tempo de execução, largura de banda e uso de CPU. Além disso, existem limites de recursos agregados que podem ser aplicados numa série de funções ou em toda a plataforma.
- Segurança — O forte isolamento de funções é crítico, pois funções de muitos utilizadores são executadas numa plataforma compartilhada.
- Escalonamento — A plataforma deve garantir a escalabilidade e elasticidade das funções dos utilizadores. Isso inclui o aprovisionamento proativo de recursos em resposta à carga e em antecipação a uma carga futura. Este é um problema desafiador do *serverless* porque essas previsões e decisões de aprovisionamento devem ser feitas com pouco ou nenhum conhecimento no nível da aplicação. Por exemplo, o sistema pode usar comprimentos de fila de solicitação como uma indicação da carga, mas não conhece a natureza dessas solicitações.
- Nuvem híbrida — Conforme o servidor *serverless* em vindo a ganhar popularidade, pode haver mais de uma plataforma *serverless* e vários serviços *serverless* que precisam de trabalhar juntos. É improvável que uma plataforma disponha de todas as funcionalidades e funcione para todos os casos de uso.

    - Sistemas legados — Deve ser fácil aceder a sistemas mais antigos em nuvem e migrá-los a partir de código *serverless* executado em plataformas *serverless.*
    - *Vendor lock-in* — Dependendo da escolha de plataforma, se numa nuvem pública, é possível que o sistema fique limitado a continuar a consumir os produtos do mesmo provedor, por conta de limitações e altos custos de mudança, dado que os sistemas não são compatíveis com outras nuvens públicas.
- Desafios a nível de desenvolvimento e manutenção
    - Ferramentas — As ferramentas tradicionais que pressupõem o acesso aos servidores para monitorizar e depurar aplicações, não são aplicáveis em arquiteturas *serverless* e novas abordagens são necessárias.
    - Implantação — Os desenvolvedores devem ser capazes de usar abordagens declarativas, para controlar o que é implantado e ferramentas para apoiá-lo.
    - Monitoramento e depuração — Como os desenvolvedores não têm mais servidores que possam aceder, os serviços e ferramentas *serverless* precisam de se concentrar na produtividade do desenvolvedor. Como as funções *serverless* são executadas por períodos mais curtos de tempo, haverá muitas ordens de magnitude a mais delas em execução, dificultando a identificação de problemas. Quando as funções terminam, o único traço de sua execução é o que a infraestrutura de monitorização
    - da plataforma *serverless* registou.
    - IDEs — Recursos de desenvolvedor de nível superior, como funções de refatoração (por exemplo, funções de divisão e fusão) e reversão para uma versão mais antiga, etc. serão necessários e devem ser totalmente integrados com plataformas *serverless*.
    - Composição — Inclui a capacidade de chamar uma função a partir de outra, criando funções que chamam e coordenam várias outras funções e construções de nível superior, como execuções paralelas e gráficos. Serão necessárias ferramentas para facilitar a criação de composições e sua manutenção.

- Longa execução — Atualmente, as funções *serverless* são frequentemente limitadas em tempo de execução do *host*. Existem cenários que necessitam lógica de longa duração. Ferramentas e modelos de programação podem decompor tarefas de longa duração em unidades menores e fornecer o contexto necessário para rastreá-las, como uma unidade de trabalho de longa duração.
- *Stateful* — as aplicações geralmente exigem um estado e as plataformas *serverless*, intencionalmente, não possuem formas de manter o estado entre execuções, então é requerido que o estado seja gerido externamente por outra solução.
- Concorrência — A lógica de programação deve lidar com requisições concorrentes aos recursos.
- Semântica de recuperação — O processo de recuperação de falhas deve ser gerido pela lógica de aplicação.

2.3.5 Serverless e Containers

As plataformas *serverless*, aproveitam os containers para implementar o seu modelo de computação. Os mesmos são usados como ambiente de execução de rápido aprovisionamento, onde as funções ou artefactos são enviados e executados.

Em razão ao objetivo do trabalho, a fim de discutir sobre a adoção de *serverless computing* utilizando o modelo de implementação CaaS, a subsecção foca nas características, implementações, vantagens e desvantagens das plataformas que utilizam *containers*.

Características

Conforme demonstrado na Seção 2.1.3, os *containers* oferecem vantagens em relação às VMs, tais como a capacidade de isolamento, armazenamento, formas de comunicação e capacidade de escalonamento, etc. A utilização destes criou um caminho para o crescimento do padrão arquitetónico de microsserviços, essa arquitetura desacopla aplicações complexas em vários serviços pequenos, implantados de forma independente, que interagem por meio de interfaces REST. Logo, este padrão tornou-se um dos mais usados para a construção de sistemas escaláveis e surgiram diversas tecnologias que promovem a gestão do uso dos *containers*, *Container Management Platforms* (CMPs), como Kubernetes, Apache Mesos ou Docker Swarm [16].

Além disso, a capacidade de executar *containers* em escala, foi adotada por provedores de nuvem pública para criar computação *serverless* [4], seja para dar suporte aos ambientes de execução das funções, como no exemplo da Figura *2.4,* aos CMPs ou a plataformas de execução de *containers* de forma transparente dos orquestradores*.*

Em comparação com as plataformas que implementam FaaS, o modelo de programação, por meio de funções, que estes serviços impõem dificulta a adoção do serviço para a execução geral de aplicações [16]. No entanto, para as plataformas de *serverless* que implementam CaaS*,* a entrega do *software* é feita por meio das imagens de *containers*, facilitando assim a criação de diversas outras aplicações e melhora em aspetos que vão além da infraestrutura e simplificam também a forma como as aplicações são desenvolvidas, todas estas vantagens do empacotamento oferecido pelos *containers*.

Por fim, estas tecnologias *serverless* que utilizam *containers*, fornecem uma melhor capacidade de migração das aplicações tradicionais e criação de aplicações genéricas, mais autoridade e controlo sobre o ambiente de execução dos programas, tempo de execução ilimitado, melhor performance de hardware [5] e também promovem uma maior flexibilidade ao escolher uma plataforma, evitando o *lock-in* das soluções para com os provedores ou até mesmo CMPs.

Implementações

É possível dividir as plataformas *serverless* que implementam CaaS em dois grupos, *open source frameworks* e serviços de nuvem pública. A seguir, a definição e exemplos de ambos:

- *Open source frameworks*

São semelhantes aos CMPs ou são executadas no topo dos mesmos e são ferramentas de *software* livre que podem ser instaladas em qualquer máquina e usados para implementar esse modelo de computação. Muitas dessas *frameworks* são apresentadas como FaaS, e isso pode ser confuso. Na verdade, algumas fr*ameworks* utilizam *containers* como ambiente de execução de funções, mas o artefacto enviado para os mesmos é um arquivo, por exemplo Fission, e outros utilizam o próprio *container* como unidade de desenvolvimento, sendo estes considerados CaaS, por exemplo *Knative*. Embora o total controlo do ambiente seja

uma vantagem, estas f*rameworks* requerem alguma gestão e podem fugir do conceito de *serverless* por esse motivo.

- Serviços de nuvem pública

São os serviços *serverless* para execução de *container* desenvolvidos e disponíveis na nuvem pública. É Importante distinguir esse conjunto dos CMPs ou orquestradores geridos pelos *providers*. Em alguns casos, estes serviços podem até ser uma implementação dos próprios *frameworks open source* de forma gerida, como o *Google Cloud Run,* ou são totalmente transparentes ao utilizador, como o *AWS Lambda.* Nestes, a maior parte das responsabilidades e configurações são geridas pela nuvem, em contrapartida, geralmente apresentam algumas restrições.

2.4 Orquestração de *Containers*

A orquestração automatiza a implantação, a gestão, a escala e a rede dos *containers*, e é fundamental para as empresas que precisem de implantar e gerir centenas de milhares de *hosts* e *containers*, e é também base para vários sistemas e *frameworks*, como *serverless.* Os orquestradores de *containers* geralmente oferecem os seguintes recursos principais: controlo do limite de recursos, programação de tarefas, balanceamento de carga, verificação de integridade e, tolerância a falhas e escalonamento automático [41].

Além de oferecer suporte para *containers* em escala, uma das grandes vantagens dos orquestradores é a portabilidade, pois é possível implantar a mesma aplicação em ambientes, máquinas ou *clouds* diferentes sem grandes diferenças na infraestrutura ou no próprio ambiente de reprodução do programa.

Como outra vantagem, a utilização de *containers* e orquestradores propiciou o desenvolvimento das aplicações baseadas em microsserviços. Estes possibilitam a execução independente de várias partes de uma aplicação no mesmo hardware, com um controlo muito maior sobre os componentes individuais e ciclos de vida. Além de arquiteturas em microsserviços, os gerenciamentos do ciclo de vida dos *containers* desses sistemas estão fortemente relacionados com as práticas e processos de DevOps, como *Continuous Deployment (*CD*)* e monitoramento [42].

De forma geral, os orquestradores funcionam em *cluster*: por meio de uma ou mais VMs, que podem ou não escalar de acordo com a procura, formam o sistema de gestão e um *pool* de recursos de computação. As estruturas dos mecanismos de

orquestração, em alto nível, consistem em três camadas: gestão de recursos, programação de tarefas e gestão de serviços [42].

Algumas opções conhecidas são o Kubernetes, Docker Swarm e Apache Mesos; sendo as duas primeiras as principais ferramentas utilizadas para cumprir esta função, portanto, estas serão detalhadas posteriormente.

2.4.1 Kubernetes

O Kubernetes (K8s) é uma plataforma portátil, extensível e de código aberto para a gestão de cargas de trabalho e serviços em *containers,* que facilita a configuração declarativa e a automação da infraestrutura [45]. A plataforma foi desenvolvida pela Google em 2014, depois de mais de 15 anos de experiência executando cargas de trabalho de produção em escala com as melhores ideias e práticas da comunidade.

O Kubernetes (K8s) é um *cluster* e apresenta a arquitetura *master-slave,* para o qual um operador envia uma lista de aplicações a um nó *master* e, subsequentemente, a plataforma implanta-os em nós *slaves*. O nó *master* contém os componentes de controlo do *cluster* e pode ser replicado para garantir alta disponibilidade e tolerância a falhas [42].

De entre as capacidades que o sistema K8s possui, podemos destacar [45]:

- *Service Discovery* e balanceamento de carga — O K8s pode expor um *container* por meio do protocolo DNS ou um endereço IP. Se o tráfego para um *container* for de grande volume, o sistema balanceia a carga distribuindo o tráfego de rede para os nós; em adição, o número de nós pode ser aumentado ou diminuído de acordo com uma métrica pré-definida;
- Orquestração de armazenamento — Permite montar, automaticamente, um sistema de armazenamento, como armazenamentos locais, NFS e serviços de provedores de nuvem pública;
- *Rollouts* e *rollbacks* automatizados — Permitem definir ou mudar o estado dos *containers*: desta forma, o sistema opera para que o estado atual mude para o estado desejado, de forma controlada. Por exemplo, pode-se automatizar o K8s para criar novos *containers* para implantação, remover os *containers* existentes e atribuir todos os recursos necessários e pedidos para os novos;
- Gestão de recursos — É possível, definir no *cluster* os nós que podem ser usados para executar os *containers*, bem como a quantidade de CPU e memória que cada *container* precisa;

- Autocorreção — O sistema reinicia ou substitui os *containers* que falham e que não respondem à verificação de integridade definida pelo operador;
- Gestão de *secrets* e configurações —- O K8 permite armazenar e gerir informações confidenciais, como senhas, *tokens* OAuth e chaves SSH. É possível implantar e atualizar *secrets* e configurações de aplicação sem reconstruir as suas imagens.

Componentes

Como indicado anteriormente, o sistema do Kubernetes é composto por nós *masters* (*control plane*) e *slaves* (*workers*). Seguindo esta arquitetura, os seus componentes são divididos dado a função do nó, como na *Figura 2.;* o *control plane* possui as componentes:

- kube-apiserver — Componente que expõe a API do Kubernetes para ser invocada pelos próprios componentes, *containers*, clientes ou serviços externos, que objetivam alterar o estado do sistema;
- etcd — Banco de dados de armazenamento *key-value*, é usado para armazenar todas as informações de estado do *cluster*, por isso, tem propriedades de alta disponibilidade;
- kube-scheduler — Responsável por observar os *pods/ containers* recém-criados e atribuí-los a um *worker*: para tal, esse componente leva em consideração os requisitos de recursos individuais e coletivos, restrições de *hardware/ software*, especificações de afinidade, localização dos dados, interferência entre cargas de trabalho e prazos;
- kube-controller-manager — Responsável por executar os processos de controlo. Em resumo, este componente verifica periodicamente o estado atual do sistema e compara com o estado desejado; caso haja alguma alteração, executa as ações necessárias para atingir a desejada configuração;
- cloud-controller-manager — Este componente incorpora a lógica de controlo específica dos provedores de nuvem, as K8s, permitindo que o *cluster* faça requisições à API do provedor para gerir os serviços, que serão utilizados pelo mesmo, como discos, balanceadores de carga, credenciais, configurações DNS, etc.

Os *workers,* responsáveis por executar os *containers*, possuem os componentes:

- kubelet — Garante a execução dos *pods/ containers* em cada nó;

- kube-proxy — É um proxy de rede executado em cada nó do *cluster*, implementando parte do conceito de serviço Kubernetes;
- *Container Runtime — Software* responsável por executar os *containers:* alguns exemplos são o próprio Docker, Containerd, CRI-O e qualquer outra implementação que siga os padrões impostos pelo Kubernetes CRI (Container Runtime Interface).

Além dos componentes principais, existem componentes opcionais, ou *addons*, que são formados por k8s *resources* e podem estender as funcionalidades do *cluster*; alguns exemplos são:

- *Cluster* DNS — Componente que implementa um servidor DNS, com o objetivo de fornecer registos DNS para serviços do Kubernetes;
- *Dashboard — Interface* web de uso geral, que permite a gestão e resolução de problemas de aplicações em execução no *cluster*, bem como no próprio *cluster*;
- *Container Resource Monitoring* e *Cluster Level Logging* — Responsáveis por recolher, armazenar e gerir *time-series metrics* e logs, respetivamente, são o sistema e seus *containers.* Apesar de serem um componente opcional, estão presentes na maior parte dos sistemas e implementações do K8s, pela ampla adoção das práticas de DevOps.

Objetos

O K8s facilita a configuração declarativa e a automação de infraestrutura, dado que a sua configuração é definida por objetos, estes são armazenados no *etcd* e representam o estado do *cluster,* como as configurações dos *workers*, dos seus respetivos *containers* e a forma que eles respondem às requisições.

Os objetos, também podem representar uma intenção ou estado desejado, por exemplo, quando um objeto do tipo pod é criado através da *api kube-apiserver*: este componente cria um “valor de intenção” no etcd; em seguida o processo do *kube-controller-manager* verifica que há uma ação a ser tomada e, com ajuda *do kube-scheduler*, faz a implantação do pod num nó *worker.* Adicionalmente, caso o *worker* falhe, o *kube-controller-manager* perceberá e tratará de criar novamente o pod noutro nó disponível, ou seja, o *cluster* trabalha constantemente para manter o “estado desejado”.

A seguir, a definição de um objeto K8s em formato. yaml, formato padrão do sistema.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: Nginx-deployment
spec:
  selector:
    matchLabels:
      app: Nginx
  replicas: 2
  template:
    metadata:
      labels:
        app: Nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

O documento tem as seguintes partes principais:

- apiVersion — Versão da API K8s usada para criar o objeto.
- *kind* — O tipo de objeto, também pode ser referenciado como *"resource".*
- *Metadata* — Dados que ajudam a identificar o objeto, de maneira exclusiva e agrupada, como nome, UID e um *namespace.*
- spec — Estado desejado para o objeto.

Exemplificando, o aprovisionamento do objetivo pode ser feito através da K8s CLI, *kubectl,* com o comando*:*

```
kubectl apply -f https://k8s.io/examples/application/deployment.yaml --record
```

Como resultado da implantação deste objeto, o *cluster* cria um recurso do *kind igual a "Deployment"* e configura-o de acordo com as definições contidas na parte *spec,* que representa o estado desejado.

Posto a exemplificação de um aprovisionamento de infraestrutura no K8s, por fim, é necessário entender os diferentes tipos de objetos que podem ser usados para compor os sistemas. A seguir, a descrição dos principais:

- Pod — Representa um conjunto de *containers* em execução no seu *cluster*. Estão agrupados numa mesma unidade, e compartilham recursos de rede e de disco;
- *Deployment* e *ReplicaSet* — O *Deployment*, é um recurso que gere a carga de trabalho da aplicação no *cluster*, *o Deployment* contém um outro recurso aninhado que o mesmo controla, *ReplicaSet* (responsável por criar um conjunto de *Pods);*
- *StatefulSet* e *DaemonSet* — São semelhantes ao *Deployment;* têm o objetivo de executar cargas de trabalho. A diferença é que um executa Pods e os dispõe num volume permanente, *StatefulSet*, e o outro cria apenas um Pod por nó do cluster, *DaemonSet*;
- Cron e CronJob — Definem tarefas que são executadas até à conclusão e depois param. Os *Jobs* representam tarefas únicas, enquanto os *CronJobs* ocorrem de acordo com um período definido;
- Service — A ideia de um serviço é agrupar um conjunto de *endpoints* de vários Pods, por exemplo um *Deployment* ou selecionando *labels,* num único recurso de rede ou endpoint. Os serviços podem ser de quatro tipos:
- *ClusterIP* — Expõe um serviço que só pode ser acedido a partir do *cluster;*
- *NodePort* — Expõe um serviço por meio de uma porta estática no IP de cada nó;
- Balanceador de carga — Expõe o serviço por meio do balanceador de carga do provedor de nuvem;
- *ExternalName* — Mapeia um serviço para um campo endereço DNS definido;
- *Ingress* — Recurso que expõe rotas HTTP e HTTPS de fora do *cluster* para serviços dentro do *cluster*. O roteamento de tráfego, é controlado por regras definidas. Em resumo, é um componente com a função de um *proxy* reverso*,*

inclusive, pode ser implementado como uma extensão dos proxys já difundidos no mercado, como o Nginx;

- Objetos de armazenamento — O K8s fornece uma API para utilizadores e administradores, que abstrai detalhes de como o armazenamento é fornecido e como é consumido. Esta API fornece dois recursos principais:
- *PersistentVolume* — É uma peça de armazenamento no *cluster* que foi provisionada por um administrador ou provisionada dinamicamente, usando classes de armazenamento. Este objeto abstrai os detalhes da implementação do armazenamento, seja NFS, iSCSI ou um sistema de armazenamento específico do provedor de nuvem;
- *PersistentVolumeClaim* — Corresponde a uma solicitação de armazenamento por um operador. *PersistentVolumeClaims*, consomem recursos dos *PersistentVolume* e relaciona-os aos *Pods*;
- ConfigMap e Secrets — ConfigMaps, são objetos usados para armazenar dados não confidenciais, em pares chave-valor. Os *pods* podem consumi-los como variáveis de ambiente, argumentos de linha de comando ou como arquivos de configuração num volume. Os *secrets* são semelhantes e permitem armazenar e gerir informações confidenciais, como senhas, tokens OAuth e chaves SSH de forma mais segura.

É importante citar que, nesta explicação, foram selecionados apenas os principais tipos de recursos que são utilizados pelos desenvolvedores de aplicações, portanto, por não fazer parte do alcance do trabalho ou por ser além de uma introdução, foi omitida a etapa de configuração do *cluster K8s*. Em contrapartida, as demais funcionalidades ou ferramentas que fazem parte do K8s e são importantes para o objetivo em causa são abordadas na seção seguinte, *Extensões e Ferramentas*.

Extensões e Ferramentas

O Kubernetes é altamente configurável e extensível. Como resultado, raramente é necessário bifurcar ou enviar *patches* para o código do projeto Kubernetes [47]. Desta forma, uma vez que o código é aberto e a comunidade responsável incentiva a sua modificação e desenvolvimento, surgem inúmeras distribuições do projeto.

Além das distribuições, que se adaptam a diferentes nuvens, também existem extensões ou *plugins* a API do K8s que permitem a criação de novos tipos de recursos que são orquestrados com uma lógica customizada.

Por fim, além das distribuições e novos recursos que podem ser montados nos *clusters* K8s*,* também é importante citar as ferramentas que auxiliam e integram a configuração da sua infraestrutura, auxiliando os desenvolvedores a gerirem de forma mais rápida e otimizada o estado do sistema.

Exemplificando essas ferramentas, elas surgem para resolver específicas cargas de trabalho, como *Kubeflow* (voltado para execução de processos de *Machine Learning),* e KEDA (*container autoscaling* para filas e tópicos), ou para suporte às práticas de desenvolvimento como os *service-mesh*, que surgiram como resolução das necessidades das arquiteturas de microsserviços, e muitas outras que dão suporte aos processo de DevOps, como Tekton (ferramenta de CI/CD), Prometeus (ferramenta de captação de métricas) e Helm (ferramenta configuração e gestão de objetos K8s).

A seguir, a definição de algumas dessas supracitadas ferramentas, que são do âmbito da presente dissertação:

- *ServiceMesh* — De uma forma geral, estes tipos de componentes gerem o tráfego de rede entre os serviços ou microsserviços, tornando as comunicações entre os serviços na rede, seguras e confiáveis. Os mesmos permitem separar a lógica de negócios das aplicações e a capacidade de observação e regras de rede e segurança entre os mesmos;
- *Operator* — K8s *Operators* é um método de empacotar, implantar e gerir uma aplicação K8s. Um operador K8s é um controlador específico da aplicação que estende a funcionalidade da API Kubernetes para criar, configurar e gerir instâncias de aplicações complexas em nome de um utilizador Kubernetes. Em suma, a aplicação operadora é implantada no *cluster* e gere a instalação e manutenção de outra aplicação usando a API K8s;
- Helm — É um gestor de pacotes para Kubernetes que ajuda a instalar e gerir aplicações no *cluster*. A gestão é efetuada por artefactos denominados de "*Helm Charts"* que ajudam a definir os objetos, a instalar e atualizar aplicações complexas ou criar *templates* de aplicações personalizadas;

- *Logging* e monitorização — Os sistemas de *logging* e monitorização do K8s, seguindo a adoção das empresas pela cultura de DevOps, tornaram-se mais presentes nos *clusters* e têm-se tornado parte fundamental em várias das distribuições e aplicações. Estes sistemas de observabilidade são implantados de forma padrão por distribuições de provedores de nuvem pública, por meio de operadores ou como parte de outras aplicações e recursos, como no caso dos próprios serviços *meshes*;
- *Serverless frameworks* — Um dos objetos de estudo do presente trabalho, os *serverless frameworks* estendem a API do K8s e criam serviços e *containers* que podem "escalar para 0" e executar funções.

2.4.2 *Docker* e Docker Swarm

O *Docker*, é uma ferramenta *open source* desenvolvida para facilitar a criação, implementação e execução de aplicações a partir de *containers*. Os *containers* permitem aos desenvolvedores criar um ambiente e executar no mesmo uma aplicação com todos os requisitos necessários, tais como recursos, bibliotecas ou dependências da aplicação. É importante notar que, devido à popularidade da ferramenta, é comum referenciar *containers* como "*dockers*", entretanto o *Docker* é apenas uma das tecnologias que implementam esse tipo de virtualização.

O *Docker* usa uma arquitetura cliente-servidor, onde o cliente se comunica com o *Docker Daemon,* programa do *host* responsável pela construção, execução e distribuição de seus *containers* junto ao *kernel* do OS. O cliente e o *Docker Daemon* comunicam-se usando uma API REST, por soquetes UNIX ou uma interface de rede. Os outros componentes do sistema são o *registry,* servidor responsável por armazenar as imagens, e os *docker objects,* como o próprio nome sugere são artefactos ou configurações do sistema (redes, volumes, imagens e os em *containers* em si) [43].

Quanto à orquestração, o *Docker Swarm* ou simplesmente *Swarm* é uma plataforma open source criada em 2013 e é o mecanismo de *cluster* nativo do *Docker*, ou seja, é o sistema de orquestração de *containers Docker*.

O *Swarm* consiste em *nodes*, vários *hosts* ou VMs *Docker* que são executados no modo *Swarm* e agem como *managers* (para gerir o *cluster*) e *workers* (que executam *services*). Ao criar um *service* ou *task*, implantação de um ou mais *containers* com a mesma função, define-se um estado ideal (número de réplicas, recursos de rede,

armazenamento, CPU, etc). Dado um estado, o sistema verifica-o periodicamente e gere os seus componentes para manter o ponto desejado. Por exemplo, se um nó de trabalho ficar indisponível, o *Docker* agenda as tarefas desse nó noutros nós [44].

2.5 Arquiteturas de Software e DevOps

Outro fator importante para atingir o objetivo de construir aplicações escaláveis na nuvem, é implementar as arquiteturas a nível de componentes, pois é fundamental criar partes desacopladas de forma que essas possam escalar de forma horizontal sem problemas para o sistema.

Tradicionalmente, os sistemas são desenvolvidos em padrões monolíticos, ou seja, eles executam todas as suas funções num único servidor. Essa arquitetura dificulta o dimensionamento dos serviços, pois gera muito desperdício de recursos, uma vez que eles são genéricos.

Em contrapartida aos padrões monolíticos, existem padrões de arquitetura que usam a estratégia de dividir para conquistar, onde os sistemas dividem uma grande tarefa entre várias nós ou o sistema é dividido em várias partes, que crescem e diminuem de acordo com as necessidades de tornar o processamento mais eficiente.

Seguem-se alguns exemplos de arquiteturas que são adaptáveis a serviços em nuvem escaláveis [36]:

- *N-tier* — A arquitetura *N-Tier*, também é chamada de arquitetura de várias camadas, porque o software foi projetado para ter as funções de processamento, gestão de dados e apresentação, separados fisicamente e logicamente. Isso significa que essas funções diferentes são hospedadas em várias máquinas ou *clusters*, garantindo que os serviços sejam fornecidos sem que os recursos sejam compartilhados e entregues na capacidade máxima. Essa arquitetura é um ajuste natural para a migração de aplicações existentes que já usam uma arquitetura em camadas. Por esse motivo, a mesma é mais frequentemente vista em soluções de infraestrutura como serviço (IaaS) ou aplicações que usam uma combinação de IaaS e serviços geridos;
- Web-Queue-Worker — Nesta arquitetura, as aplicações possuem um *front-end web* que lida com solicitações HTTP e um *worker* no *back-end* que executa tarefas de uso intensivo da CPU. A comunicação entre o *front-end* e *o back-end* é feita por meio de uma fila de mensagens assíncronas. A arquitetura de

Web-Queue-Worker é adequada para domínios relativamente simples com algumas tarefas que exigem muitos recursos. O uso de serviços dissociados simplifica a implantação e as operações, mas com domínios complexos, pode ser difícil gerir dependências;

- *Microservices* — Para os domínios mais complexos, é recomendado usar a arquiteturas de microsserviços. Os microsserviços são uma técnica de desenvolvimento de software, uma variante do estilo estrutural da arquitetura orientada a serviços (SOA), que organiza uma aplicação como uma coleção de serviços vagamente acoplados. Numa arquitetura de microsserviços, os serviços são refinados e os protocolos são leves. Cada serviço pode ser construído por uma pequena equipe de desenvolvimento focada. Serviços individuais podem ser implantados sem muita coordenação entre equipes, o que incentiva atualizações frequentes. Esta subdivisão, facilita não apenas a manutenção do sistema, mas também a eficiência no escalonamento apenas dos serviços mais exigidos;
- *Event-driven architecture* — As arquiteturas orientadas a eventos usam um modelo de publicação-assinatura *(pub-sub)*, em que os produtores publicam eventos e os consumidores os assinam. Os produtores são independentes dos consumidores, e os consumidores são independentes um do outro. Uma arquitetura orientada a eventos, é geralmente usada para aplicações que ingerem e processam um grande volume de dados com latência muito baixa ou quando diferentes subsistemas devem executar diferentes tipos de processamento, nos mesmos dados do evento.

2.5.2 Arquiteturas e *serverless*

Conforme apresentado na subsecção anterior, uma arquitetura de aplicações de microsserviços divide o sistema em pequenos componentes responsáveis por partes do negócio. A decomposição das funcionalidades dos sistemas em microsserviços, permite uma melhor agilidade do desenvolvedor no suporte aos processos de desenvolvimento de *software* DevOps [39].

Além de se encaixar nas práticas de DevOps, esta arquitetura de microsserviços tem sido associada às plataformas *serverless,* pois, por questões de *performance* e facilidade, estas plataformas têm seu âmbito dividido (por exemplo, em várias funções). Por sua vez, a computação *serverless* e suas variantes, como FaaS e

CaaS, prometem aprimorar as arquiteturas de microsserviço existentes, executando-as como serviços geridos de forma mais centralizada [40].

De acordo com suas características, apresentadas na *Seção 2.3,* as plataformas de *serverless* suportam eventos naturalmente, desta forma são de grande valia para arquiteturas orientadas a eventos. Inclusive, esta arquitetura pode ser implementada por meio de microsserviços, pois as duas são compatíveis.

Fundamentalmente diferente da hospedagem de aplicativos IaaS ou PaaS, em *serverless*, as aplicações são decompostas e implantadas como módulos de código, sejam funções ou *containers*. Estes módulos são frequentemente utilizados para hospedar serviços *Web RESTful* e têm um tamanho de código pequeno, consequentemente podem ser chamados de microsserviços. [39]

2.5.3 DevOps

Amplamente utilizado pelas empresas para melhorar o processo de entrega de *software* [37], DevOps, além de uma forma de implantar o *software* desenvolvido, é um conjunto de práticas utilizadas para automatizar e integrar os processos entre o desenvolvimento de *software* e as equipas de infraestrutura, para que possam construir, testar e lançar o *software* de forma mais rápida e confiável.

Tradicionalmente, as equipas de *software* desenvolviam um conjunto de funcionalidades, ou até mesmo um sistema completo, e criavam artefactos que só eram entregues para as equipas de infraestrutura após a finalização do projeto. Esta metodologia, além de criar conflitos entre as equipas, era suscetível a erros, pois os ambientes de desenvolvimento e produção eram diferentes, sendo testadas as funcionalidades e a infraestrutura pelo cliente final apenas no momento da entrega.

O *Agile Software Development* pressupõe que a equipa pode lançar *software* com frequência em algum ambiente semelhante ao de produção [35]. Seguindo as mudanças e tendências da forma de desenvolver *software*, que substituem o tradicional modelo em cascata por metodologias iterativas, DevOps surge como uma evolução do movimento ágil aplicada aos processos de operações de infraestrutura e desenvolvimento com o objetivo de mais rapidez e melhor qualidade na implantação do software, agregando valor ao produto final.

Posto a relação entre *Agile* e DevOps, é importante salientar que ambas são metodologias culturais, ou seja, para a sua execução é preciso, além da implantação de processos e ferramentas, uma mudança na forma de avaliar e cooperar das

equipas de TI ou da filosofia das organizações. Em resumo, as equipas que produziam *software* em silos, passam a cooperar numa única unidade e têm uma melhor produtividade, pois, uma vez que as decisões e processos (como os de infraestrutura e qualidade) que eram executados fora de cada núcleo passam a ser executados internamente, as equipas têm maior capacidade de resolução e mais responsabilidades.

2.5.4 DevOps e *Serverless*

Serverless, tem como umas das características abstrair os ambientes de execução dos servidores para desenvolvedores, promovendo transparência de escala e manutenção. Estes modelos sugerem que não há necessidade de equipas de operações especializadas ou, pelo menos, que menos pessoas sejam necessárias para as operações [35].

Em contrapartida, apesar de diminuir a necessidade de gestão dos servidores, *serverless* não impede ou diminui a adoção de práticas DevOps, pois ainda é necessária uma implantação adequada do *softwar*e, ferramentas para registar ou monitorizar os ambientes *serverless* (funções ou *containers*) ou para testá-los antes que cheguem aos utilizadores em produção [38].

Também, semelhante ao que acontece nas plataformas de BaaS, ver *Seção 2.2.4,* surgem no mercado ferramentas de DevOps que dão suporte às plataformas de *serverless.* Estas ferramentas, por vezes, fazem parte da mesma solução, por exemplo ferramentas de monitorização, *tracing* e *deploy* de funções do AWS Lambda*,* ou , que por si só, são serviços que podem ser usados como ferramentas de DevOps e possuem as características dos *frameworks serverless*; em resumo, são ferramentas geridas e escaláveis sem necessidade da intervenção de uma equipe de operações e amigáveis para os desenvolvedores; como exemplos podemos citar AWS CloudWatch, Google Stackdriver e Azure Monitoring, todas ferramentas geridas de monitorização e *logging*.

Por fim, de um ponto de vista simplificado, práticas de DevOps e *serverless* são compatíveis e têm objetivos em comum como: a melhoria na qualidade dos serviços em escala e gestão dos serviços de forma automática ou gerida.

2.5.5 Ciclo de vida DevOps, Práticas e Ferramentas

Citado anteriormente, DevOps é uma cultura que preenche a lacuna entre as equipas de desenvolvimento e operação, que historicamente funcionavam em silos, e estende a metodologia *Agile*, promovendo entregas contínuas.

Devido a essa natureza contínua, os profissionais usam o laço infinito para mostrar como as fases do ciclo de vida do DevOps se relacionam. Embora pareça uma sequência de ciclos que se repetem, os ciclos são contínuos e sempre presentes. A ideia central é simbolizar a necessidade de colaboração constante e melhoria iterativa ao longo de todo o ciclo de vida [48].

Nomeadamente, o ciclo de vida do DevOps consiste em seis fases, representando os processos necessários para o desenvolvimento no lado esquerdo do laço e os processos necessários para as operações no lado direito do laço. Ao longo de cada fase, as equipas colaboram e comunicam-se para manter o alinhamento, a velocidade e a qualidade. O ciclo de vida do DevOps inclui fases para planear, *building*, integrar e implementar continuamente (CI/CD), monitorizar, operar e responder ao *feedback* contínuo.

Além do ciclo de vida DevOps, as organizações que implementam essas culturas partilham de práticas comuns a todo o ciclo. As práticas DevOps são processos, na maioria das vezes automatizados, que são utilizados para facilitar e melhorar as tarefas comuns em cada ciclo. A seguir, na *Tabela 2.5.1* estão apresentadas as principais práticas DevOps [50] e de forma agrupada por ciclo.

Ciclo	**Prática**	**Definição**
Planeamento	*Continuous planning*	Abordagem de planeamento, em que planos estáticos anuais ou semestrais são substituídos por um plano continuamente atualizado.
	Defining requirements	Levantamento incremental para definição de novos desenvolvimentos e de acordo com o *feedback* obtido.

	Prototyping application	Entrega de provas de conceitos para validar os requisitos.
	Designing architecture	Levantamento e definição de componentes, suas interligações e processos necessários.
	Integrated testing and deployment planning	Planeamento de entregas automatizadas e com qualidade, de acordo com a necessidade de mudanças do *software*.
Building	*Integrated configuration management*	Uso de práticas para versionamento e segurança de configurações e *secrets,* como o uso de KMS.
	Integrated change management	Fluxo bem definido e versionado de mudanças no *software*, uso de ferramentas git e repositórios integrados.
Integração	*Continuous integration*	Automatização do maior número possível de processos de qualidade. Podem ser automatizados testes unitários, de integração, de interface, verificação de qualidade e cobertura do código, segurança de dependências e de aplicação, etc.
	Automated testing	Automação de testes de *software* após modificação no código que são executados antes da implantação.
	Staging application	Criação de ambientes similares à produção para teste.

	Use of data to guide QA	Utilização de dados ou padrões de utilização de produção para alcançar maior parâmetro de qualidade.
	Continuous application performance	Automação de testes de carga e persistência para garantir que as aplicações respondem a uma grande quantidade de pedidos.
Deploy	*Continuous delivery*	Entrega regular de *software* ao ambiente final executada após uma modificação do código que resulta em um *pipeline* de ações.
	Continuous deployment	Similar a *Continuous delivery,* representa o estado final de maturidade das implantações, onde todas as modificações, de forma totalmente automatizada, são validadas e entregues ao ambiente final.
	Cooperative application configurations	Surgiu da necessidade de vários ambientes de execução. É a prática de gestão de configuração, *secrets* ou de dependências das aplicações em cada ambiente de forma facilitada.
Operação	*Process standardization*	Definição de processos e automatizações para aprovisionamento de infraestrutura.
	Infrastructure as Code	Infraestrutura como código (IaC) é a gestão da infraestrutura (redes, máquinas virtuais, balanceadores de carga, etc.) num modelo descritivo,

		usando o mesmo controlo de versão que a equipe de DevOps usa para o código-fonte.
	Modeling & Simulation	Prototipação e testes de implantação de infraestrutura. Prática que ganhou força com a adoção de IaC.
	Elasticity practice	Testes realizados para verificar se a mesma está a escalar os seus recursos de acordo com a procura. Também pode ser relacionado ao ciclo de *Continuous Integration*.
	Production support	Criação de rotinas e canais de comunicação, para resolução de problemas que venham a ocorrer na produção.
Monitorização	*Continuous monitoring*	Processo de observar e detetar problemas de conformidade, ou ameaças de segurança durante cada fase do *pipeline* de DevOps.
	Automated performance monitoring	Monitorização do uso de recursos da infraestrutura com alertas automáticos para configuradas ocorrências.
	Application monitoring and dashboards	Criação de *dashboards* para fácil leitura do estado da aplicação.
	Monitoring application and next development	Monitorização de mudanças ou de degradação de *performance* que possam acontecer no desenvolvimento ou após implantação de novas funcionalidades.

	Measure performance metrics	Avaliação das melhores métricas ou criação de novas de forma customizada para melhor perceber o estado do sistema.
Feedback	*The feedback loop between developers and operators*	Prática de aprendizagem, melhoramento ou unificação dos processos entre as equipas de desenvolvimento e infraestrutura.
	Automated feedback for performance	Avaliação completa da aplicação. Pode ser feito de forma direta por utilizadores ou indiretamente por meio de indicadores de performance. Os resultados dos inquéritos alimentam a próxima fase de planeamento.
	DevOps maturity evaluation model	Avaliação cíclica do processo de desenvolvimento e operação, a fim de automatizar ou melhorar os processos a cada período.

Tabela 2.5.1: Práticas comuns em organizações que utilizam DevOps.Fonte(Autor)

Por fim, para implementar os supracitados processos, muitas das equipas de TI utilizam inúmeras ferramentas ou tecnologias que dão suporte ou que transformam a forma que as aplicações são implementadas e verificadas. De forma superficial, a maioria das ferramentas tem o objetivo de gerir e automatizar todos os pormenores possíveis do desenvolvimento.

É importante notar que o uso de determinadas ferramentas apenas suporta a cultura DevOps, e não necessariamente que o uso das mesmas está relacionado a filosofia. As ferramentas DevOps podem ser divididas em seis categorias, relacionadas com os seus principais conceitos [36]: compartilhamento de conhecimento e comunicação, gestão de código, *build,* integração, implantação e monitorização. Logo, na *Tabela 2.5.2*, exemplos de cada uma dessas categorias.

Categoria	Exemplos	Conceitos
Compartilhamento de conhecimento e comunicação	- *Confluence* - Azure Wiki - GitLab Wiki - Trello - *Google Drive* - *Slack*	- Cultura de colaboração - Compartilhar de conhecimento - Quebra de silos
Gestão de código	- Git - SVN - Repositorios: *Gitla*b, *Github, Bitbucket*, etc	- Controlo de versão e qualidade de código - Cultura de colaboração - Compartilhar conhecimento - Quebra de silos
Build	- Maven - npm - JUnit - *Docker*	- Entrega de artefactos - Entrega contínua - Automação de testes - Gestão de configurações - Virtualização de ambientes
Integração	- Jenkins - Gitlab CI - Tekton - Travis - K6.io, Jmeter - Selenium - Sonarqube	- Integração contínua - Processo de implantação confiável - Testes de performance, segurança, unitários, de interface, etc. - Verificação de qualidade
Implantação	- K8s, Rancher - Chef, Puppet - Ansible - Terraform, Packer	- Processo de libertação frequente - Gestão de mudanças e implantações

	- *Cloud Formation* - Gitlab CD, Jenkins	- Gestão de configurações e *secrets* - Infraestrutura como código - Virtualização e *containers* - Integração com serviços em nuvem - Automação de configurações e ambientes
Monitorização	- Zabbix, Prometheus - ELK, FluentD - InfluxDb - Grafana - Graylog - StackDriver - Cloud Watch - Azure Monitoring	- Monitorização contínua - Verificação de desempenho, disponibilidade, escalabilidade - Resiliência, confiabilidade, automação - Métricas, alertas, experiências - Geto de logs - Verificação de segurança - Suporte e reações a incidentes

Tabela 2.5.2: Tabela de ferramentas DevOps.

Capítulo III – Materiais e Metodologia

Este capítulo, apresenta a metodologia que será utilizada para análise das tecnologias *serverless* baseadas em *containers* e tem o objetivo de elencar e avaliar as características, desvantagens e vantagens das existentes plataformas na indústria.

Para este fim, serão realizadas duas avaliações: uma qualitativa, focada nas características e capacidades, e outra quantitativa, que avaliará a eficiência de resposta e escalonamento dessas tecnologias por meio de um *benchmark*. Em adição, os resultados da avaliação qualitativa servem como ponto de partida da avaliação quantitativa, uma vez que por meio da mesma é possível focar nas tecnologias com maior valor agregado.

Na primeira secção do capítulo encontram-se os critérios de escolha e quais tecnologias serão avaliadas. Na *Secção 3.2,* contém a descrição das plataformas escolhidas. A *Secção 3.3* exibe a avaliação qualitativa, na qual é exposta uma comparação com algumas conclusões sobre as plataformas elencadas. Por fim, a *Secção 3.4* aborda os *benchmarks* de outros trabalhos da literatura relacionados, e também define a experiência prática a ser aplicada*,* a qual é executada no capítulo seguinte.

3.1 Plataformas *serverless* que implementam CaaS

O objetivo desta secção é elencar para posterior análise os principais *frameworks* e serviços *serverless* que dão suporte à execução de *containers*. O conjunto de plataformas *serverless* que podem ser avaliadas é dividido em dois grupos: *frameworks open source* e serviços de provedores públicos.

Do conjunto de *frameworks open source*, foram escolhidos os principais *frameworks* da comunidade de acordo com a quantidade de contribuidores e *stars* no site Github. Esses mesmos métodos e tipos de resultados foram utilizados anteriormente e expostos em "*An Evaluation of Open Source Serverless Computing Frameworks Support at the Edge"* [7]. O resultado desta pesquisa encontra-se na *Tabela 3.1.* De acordo com o critério anteriormente mencionado, foram escolhidos para futura análise os *frameworks open source*: Knative, Apache Openwhisk e OpenFaaS. Os demais, que não têm suporte a execução e *containers,* dão suporte apenas para

criação de *custom runtimes* (Kubeless e Fission), não sendo ativamente mantidos ou relevantes o suficiente segundo a métrica adotada.

Nome	Contribuidores	*Stars*
Knative [56]	218	3700
Apache Openwhisk [57]	191	5200
OpenFaaS [58]	156	19600
Kubeless* [59]	102	6400
Fission* [60]	105	6000
Fn Project [61]	86	4900
Nuclio [62]	61	3800
Iron Functions [63]	33	2900

Tabela 3.1: *Open source serverless frameworks,* contribuidores e *stars* em 3 de abril de 2021. Fonte (Autor)

Do segundo grupo, serão avaliados os produtos dos principais provedores de nuvem pública: Microsoft Azure (Azure), Google Cloud Platform (GCP) e Amazon Web Services (AWS) [8]. Os serviços escolhidos, foram retirados dos respetivos portais de cada provedor de nuvem pública e tem como principal critério de escolha a utilização de *containers* como um artefacto de execução e alto nível de abstração da gestão da infraestrutura. Como exemplos desses critérios pode-se elencar, respetivamente, a exclusão dos serviços *Azure Functions,* onde os *containers* funcionam apenas como um ambiente de execução, e dos serviços de orquestração Azure Kubernets Service, Google Kubernets Engine e Amazon Kubernets Service, pois uma vez que não escalam para zero e permitem acesso e configuração da infraestrutura associada, se aproximam mais ao modelo de serviço de IaaS. Serviços como o Google App Engine Flexible, Azure App Service e AWS Elastic Beanstalk também foram excluídos pelo mesmo motivo.

Provedor	Produto *Serverless*
Azure	Azure Container Instances [64]
GCP	Cloud Run [65]
AWS	ECS e Fargate [66]
	Lambda [67]

Tabela 3.2: Plataformas *serverless* das principais nuvens públicas.

3.2 Características das plataformas *serverless*

De acordo com o conjunto de plataformas relevantes que foram indicadas anteriormente, esta secção contém uma descrição detalhada dos dois conjuntos.

3.2.1 Knative

Knative, é uma *framework* baseada em Kubernetes para gestão de cargas de trabalho *serverless* e orientado a eventos. A exemplo do que foi abordado anteriormente na *Secção 2.4*, as APIs Knative baseiam-se e estendem as APIs Kubernetes existentes por meio de *Kubernetes Custom Resource Definitions* (CRDs); desta forma os seus recursos são compatíveis com outros recursos nativos do Kubernetes e são geridos por administradores de *cluster* usando as ferramentas Kubernetes existentes [knative]. Dentre as principais funcionalidades que caracterizam esta *framework* como *serverless* e para além da capacidade de fazer o escalonamento dos *containers* para zero, é possível citar o foco na fácil experiência do desenvolvedor e arquiteturas orientadas a eventos, escalonamento automático e uso de *Service Mesh* para controle de rotas e segurança [68].

Existem dois componentes principais do Knative que podem ser instalados e usados juntos, ou independentemente, para fornecer funções diferentes:

- Knative Serving

Gere os serviços do Kubernetes sem a necessidade de manter um estado ou *container* em execução, reduzindo o esforço necessário para escalonamento

automático e implementações. O diagrama da *Figura 3.2.1* ilustra a relação entre os objetos de serviço, roteamento, configuração e revisão.

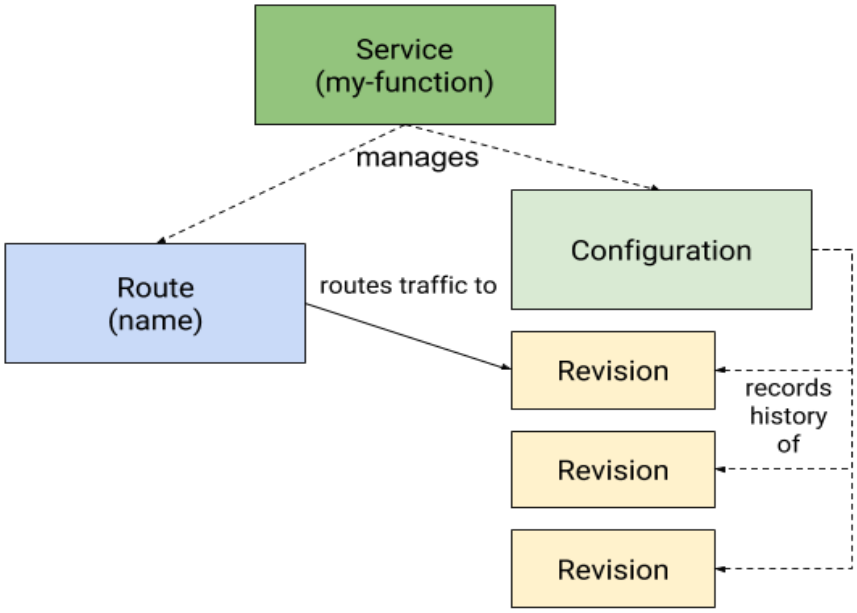


Figura 3.2.1: Diagrama da relação entre objetos de serviço, roteamento, configuração e revisão da plataforma Knative [56].

O recurso serviço (*service.serving.native.dev*) gere os *containers* da aplicação em todo o seu ciclo de vida, garantindo que esta tenha uma rota, configuração e revisão sempre que o serviço for atualizado. O objeto de rotas (*route.serving.native.dev*) mapeia cada endereço de rede para uma ou mais revisões e é útil para configurações mais avançadas de tráfego, como implementação de *blue- green deployments.* O recurso de configuração (*configuration.serving.native.dev*) mantém a implantação no seu estado desejado e ao mesmo tempo cria diferentes versões do mesmo serviço cada vez que há uma modificação da configuração, permitindo que diferentes versões sejam executadas simultaneamente. Por fim, o recurso de revisão (*revision.serving.native.dev*) fornece *snapshots* de cada modificação nas configurações da implantação. Adicionalmente, as revisões também podem ser escalonadas automaticamente com base no tráfego de entrada.

- *Knative Eventing*

Administra eventos entre componentes dentro e fora do *cluster*, expondo o roteamento de eventos como configuração e permitindo que eventos acionem funções e serviços baseados em *containers.* O *Knative Eventing* automaticamente coloca eventos numa fila onde são geridos pelo recurso *Knative Broker*, que

administra e encaminha esses eventos para os consumidores de acordo com as regras definidas pelo recurso *Knative Trigger*. Desta forma o sistema elimina a necessidade de os desenvolvedores escreverem *scripts* ou usarem *middlewares* para executar esta tarefa e a substituem por configurações.

3.2.2 Apache Openwhisk

O Apache *OpenWhisk* é uma plataforma *serverless* de código aberto que executa funções em resposta a eventos. O *OpenWhisk* gere a infraestrutura, os servidores e o dimensionamento da carga usando *containers* Docker, diminuindo desta forma a necessidade de gestão de infraestrutura para desenvolvedores.

O *framework* suporta o modelo de programação no qual os utilizadores escrevem lógica funcional chamada (Actions*).* Estas funções podem ser agendadas dinamicamente ou executadas em resposta a eventos associados (*Triggers*) de fontes externas (*Feeds*) ou de solicitações HTTP [57].

Apesar de, por defeito, apresentar suporte apenas às funções de um conjunto limitado de linguagens (*Runtimes*), é possível criar e executar qualquer ambiente de aplicação por meio de *containers docker* customizados; estes precisam apenas seguir os contratos especificados. O diagrama abaixo descreve a arquitetura *OpenWhisk*.

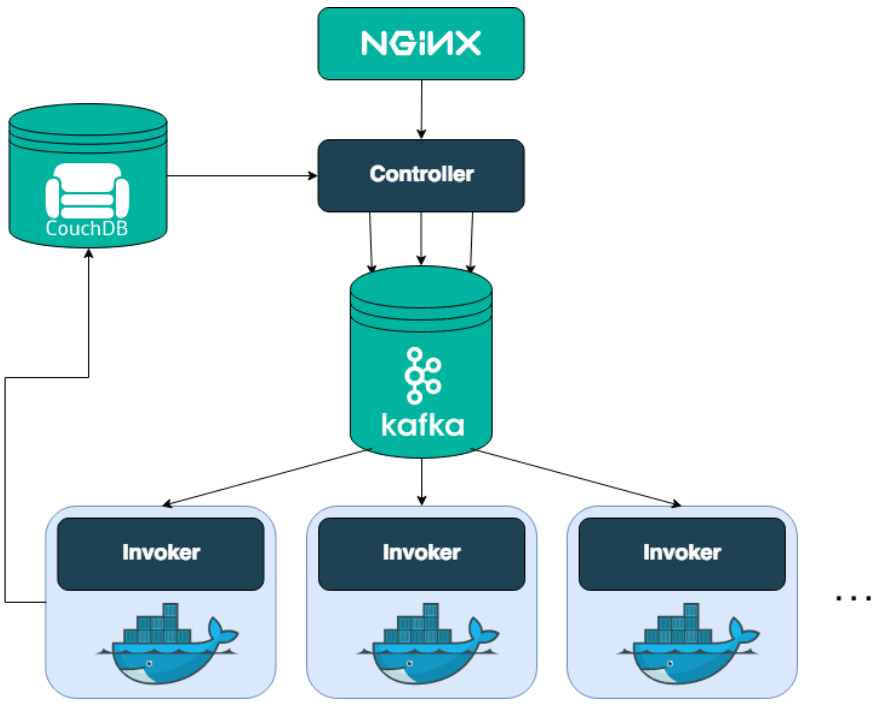


Figura 3.2.2: Diagrama de arquitetura da plataforma *OpenWhisk*. Fonte (Autor)

Ilustrando por meio de um caso de uso, as requisições chegam ao sistema por meio do componente de proxy (Nginx) e, depois de passarem pelo proxy reverso, atingem o *Controller* que atua como o porteiro do sistema. Uma vez que as mensagens são recebidas pelo *Controller*, este realiza a autenticação e autorização de cada requisição antes de passar o processamento para o próximo componente. Assim que a solicitação do utilizador for verificada e de acordo com o tipo de solicitação, o *Controller* invoca uma *Action*. Por fim, as requisições são executadas pela lógica implementada na correspondente *Action.*

Apesar de não mencionados anteriormente, os papéis executados pelo CouchDB e Kafka são fundamentais para o funcionamento dos componentes, pois servem para manter os estados dos sistemas e as aplicações escaláveis, respetivamente. Em mais detalhes, o Kafka armazena em *buffer* as mensagens enviadas pelo Controller antes de as entregar ao Invoker, provendo a ingestão de dados confiáveis e de alta velocidade.

Por fim, como vantagens competitivas do Apache OpenWhisk podem-se elencar a capacidade de aprovisionamento para diferentes orquestradores (K8s, Mesos e Swarm) e, por usar um dos brokers com maior utilização do mercado (Kafka) a quantidade de *plugins* e possibilidade de integração com uma grande variedade de ferramentas.

3.2.3 OpenFaaS

A OpenFaaS, é uma plataforma *serverless* de código aberto e, como muitas outras da categoria, tem a sua base no orquestrador de *containers* K8s. Esta *framework* implanta funções orientadas a eventos e microsserviços sem codificação repetitiva e padronizada. O código de aplicação é empacotado num binário ou numa imagem Docker e, após a implantação, é gerido pelo sistema, que exerce a responsabilidade sobre as métricas e escalonamento automático das funções.

O fluxo conceitual da *framework* OpenFaaS é iniciado pelo ponto de comunicação externa, OpenFaaS Gateway, que pode ser acedido por meio de sua API REST, de uma CLI ou por meio de uma interface *web*. Além de comandos de controlo dos sistemas, o Gateway também desempenha a função de *proxy* reverso e expõe os

serviços ou funções por meio de uma rota ou domínios personalizados. Adicionalemente, as requisições são processadas de forma assíncrona e por meio de filas. Essa funcionalidade, é suportada pelo componente de *broker,* o NATS Streaming. Após a gestão adequada de cada solicitação, as funções são direcionadas para execução no *faas-netes.* Durante todo o fluxo, o Prometheus coleta métricas relativas às requisições e funcionamento dos componentes do sistema, cujas métricas são usadas para escalonamento automático do mesmo [58].

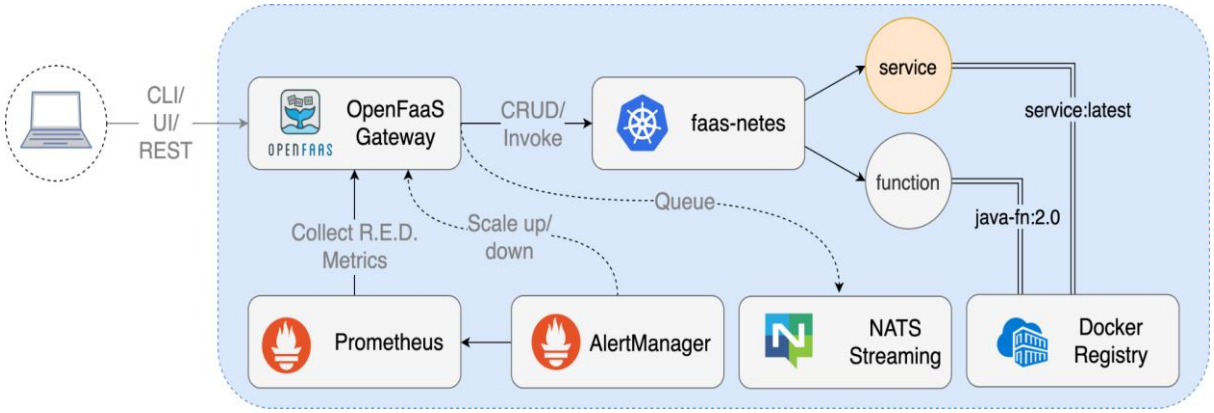


Figura 3.2.3: Diagrama de arquitetura da plataforma OpenFaaS [58].

Ainda sobre os componentes, o faas-netes é o provedor de orquestração mais popular do *framework* e é baseado em K8s, mas ainda existem outros que são baseados em *Docker Swarm*, *Hashicorp Nomad*, AWS Fargate/ ECS e AWS Lambda e que também foram desenvolvidos pela comunidade por meio do SDK do provedor OpenFaaS [58].

3.2.3 Azure Container Instances

O serviço *Azure Container Instances* (ACI) disponibiliza uma interface simples e rápida de executar *containers* na nuvem Azure sem a necessidade de gestão da infraestrutura; são uma ótima solução para qualquer cenário que possa operar em serviços isolados, incluindo aplicações simples e automação de tarefas [64].

O serviço pode ser utilizado de uma maneira simples por meio das interfaces Azure, como a consola e CLI, e também tem integração com as ferramentas *Docker* e *Compose* [69]. Uma vez que os *containers são* iniciados, a plataforma expõe os seus grupos diretamente para a Internet com um endereço IP e um nome de domínio

(FQDN). O acesso ocorre por HTTPS, usando TLS para proteger as conexões do cliente.

Além disso, a plataforma da Azure também oferece uma rápida inicialização dos *containers,* suporte a imagens com o sistema operativo Windows, montagem de volumes persistentes de diferentes origens, possibilidade do uso de GPU e integração com ferramentas de monitorização. Sobre os custos, o serviço segue um modelo de cobrança por segundo. O preço exato, varia de acordo com os recursos de CPU e memória usados por cada *container* ou grupo de *containers*.

Apesar de todas essas funcionalidades, a Microsoft recomenda o serviço ACI para aplicações mais básicas. Para cenários em que é preciso orquestração de *container* completa, incluindo descoberta de serviços em vários *containers*, dimensionamento automático e atualizações de aplicações coordenados, a provedora recomenda o Azure Kubernetes Service.

3.2.4 *Cloud Run*

O *Cloud Run*, é um serviço do Google Cloud Platform para executar *containers* de forma autónoma e abstraída de aprovisionamento de máquinas, *clusters* ou escalonamento automático. O *Cloud Run* implementa a maioria das partes da API Knative Serving. No entanto, a implementação subjacente da funcionalidade pode ser diferente da implementação de código aberto do Knative [65].

Na plataforma, após a entrega de uma imagem, o *container em* execução recebe um domínio com certificado TLS e suporta a execução de servidores HTTP. Então, o sistema faz o escalonamento automático com base em solicitações recebidas, inclusive pode escalonar para zero, e, relativo ao modelo de custos, cobra apenas durante o tempo de execução de cada solicitação que foi recebida.

Em adição de servidores HTTP, o *Cloud Run* integra-se ao serviço Pub/Sub estendendo a funcionalidade do serviço a arquitetura orientada a eventos. Cada evento lançado é entregue ao *container* via HTTP para o *endpoint* do servidor instalado no *container*. Dessa forma, é possível enviar eventos de maneira segura e privada para os *containers*, sem expô-los publicamente à Internet. Também é possível integrar o *Cloud Run* com o *Google Cloud Scheduler*, que de maneira semelhante, por meio de requisições HTTP seguras, implementa a capacidade de o serviço ser utilizado para a execução de tarefas periódicas.

Além das funcionalidades básicas mencionadas anteriormente, podemos elencar um conjunto de configurações que tornam o produto ainda mais útil, como o suporte WebSocket, o uso de revisões (favorece a implementação de rotinas de CD*),* configurações de concorrência e *warming* dos *containers,* opções de autenticação e ainda a opção de instanciar o serviço em um *cluster* Kubernetes da provedora.

3.2.5 ECS e *Fargate*

O Amazon *Elastic Container Service* (Amazon ECS) é um serviço de gestão de *containers*, semelhante ao K8s, mas com total propriedade da Amazon, altamente escalonável e rápido que facilita a execução, a interrupção e a gestão de *containers* num cluster. Os *containers* são definidos por uma *task* que pode executar tarefas individuais ou tarefas dentro de um *service.* Essas tarefas podem ser curtas, programadas e de longa duração*.* Um *service* é uma configuração que permite executar e manter um conjunto de *tasks* simultaneamente num *cluster*. Um *cluster* ECS pode ser implementado por dois tipos de serviços: EC2 e Fargate [66].

No modelo EC2, os *containers* são implantados em instâncias EC2 (VMs) criadas para o *cluster*, onde o ECS os gere junto com as *tasks* e *services*. No segundo modelo, o *cluster* funciona através de *containers serverless* e as tarefas de gestão são responsabilidade do serviço Fargate.

Com o Fargate, não é mais necessário provisionar, configurar ou dimensionar *clusters* de máquinas virtuais para executar *containers*, eliminando a necessidade de escolher os tipos de máquinas, decidir quando dimensionar ou otimizar o *cluster*. Além disso, o AWS *Fargate* apresenta recursos nativos de segurança e monitorização e no seu modelo de custos só se paga pelos recursos de vCPU e de memória utilizados [66].

3.2.6 Lambda

Já abordado durante o presente trabalho, a plataforma Lambda foi um dos primeiros serviços associados ao termo *serverless* [4] e, além da execução de funções, suporta a execução de *containers.* Essa funcionalidade foi lançada no evento re:Invent 2020 [14] em dezembro de 2020 e indica claramente o desenvolvimento da área abordada no presente trabalho, *serverless containers.*

A AWS fornece um conjunto de imagens código aberto para a criação de imagens de *container*. Estas imagens base, incluem uma interface de execução para gerir a interação entre o Lambda e o código. Uma vez que a interface de execução é *open source*, também é possível desenvolver novas imagens ou obtê-las de terceiros.

Uma vez que a imagem é construída e entregue ao serviço Lambda, o sistema implementa o mesmo comportamento das demais funções para os *containers,* como o mesmo ciclo de vida, monitorização, *triggers* e modelo de custo.

3.3 Síntese e comparação das plataformas *serverless*

Com o objetivo de comparar as plataformas *serverless* descritas anteriormente, e ter uma visão geral das mesmas em termos de funcionalidades e características, esta secção apresenta um conjunto de parâmetros e uma síntese de cada um dos atributos respetivos para cada plataforma.

Os parâmetros escolhidos permitem resumir as principais características e funcionalidades das plataformas de computação *serverless*. Além disso, os parâmetros também podem ser divididos em 6 conjuntos: *design*, *development*, testes e *deployment* (CI/CD), execução, monitorização e segurança, que são baseados no ciclo de desenvolvimento de *software* e de DevOps [2]. A seguir, na *Tabela 3.3*, são expostos os parâmetros escolhidos para a avaliação qualitativa dos *frameworks serverless* juntamente com uma breve explicação e referências de trabalhos similares que utilizam as mesmas métricas.

Grupo	**Parâmetro**	**Descrição**
Design	Licença [1] [7]	Licença de utilização e comercialização: *open source* ou proprietário.
	Pricing [52]	Modelo no qual é quantificado a valor do serviço.
	Workflows [2] [52]	Suporte para composição de ações que formam um *pipeline* de operações sequenciais por meio de operadores.

	Orquestrador [1] [7]	Orquestrador de *container*, no qual o sistema pode ser hospedado.
Development	*Trigger* [1] [2] [7] [15] [52]	Forma de execução, utilização ou protocolo de comunicação usado pelo serviço de *container.*
	Eventing [1] [2] [7]	Suporte a integração junto a plataformas de mensagens ou filas para implementação do paradigma orientado a eventos.
	Interface de uso [1] [2] [7]	Forma de uso que o desenvolvedor utiliza para configuração, desenvolvimento, configuração e monitorização.
	Registry [52]	Conjunto de sistema de armazenamento de imagens do qual a plataforma tem suporte.
	Requisitos da imagem [15]	Requisitos ou *templates* que as imagens dos *containers* devem seguir para serem executados.
CI/CD	CI/ CD [2]	Principais ferramentas para integrar práticas de *continuous integration e continuous delivery* ou que foram mencionadas na documentação.
	Container SO [15]	Sistemas operacionais suportados nos *containers.*
	Traffic splitting [15]	Configuração e divisão de tráfego de rede entre *containers.*

	Revisions [2]	Versionamento dos *deployments* executados onde as versões podem ser utilizadas em técnicas de *continuous delivery.*
Execução	Memória [15] [52]	Quantidade máxima de memória suportada para cada *container.*
	CPU [15] [52]	Quantidade máxima de vCPUs suportada para cada *container.*
	Storage [15] [52]	Suporte a *mounting* de volumes nos *containers.*
	GPU	Suporte ao uso de GPUs dentro dos *containers.*
	Tempo limite de execução [15] [52]	Tempo máximo de execução de uma *trigger request* feita ao *container.*
	Execuções concorrentes [15] [52]	Máximo de requisições que podem ser executadas ao menor tempo pelo sistema.
	Autoscaling [1] [7] [52]	Capacidade do sistema de responder a mudanças na quantidade de requisições na plataforma, alterando o número de *containers* sem a ação de terceiros.
	Autoscaling metric [1]	Métricas ou *triggers* utilizados como parâmetros de configuração da elasticidade do sistema.
	Warming	Configuração ou funcionalidade do sistema que permite ter uma quantidade mínima de

		containers esperando por requisições e diminuir o tempo de resposta para requisições espaçadas no tempo.
	Scale to zero	Capacidade do sistema de ter nenhum *container* em execução caso o número de requisições seja nulo por algum tempo.
Monitorizaçã o	*Logging* [1] [2] [7]	Ferramentas integradas que são utilizadas para capturar, guardar e consultar os logs dos *containers* e do sistema.
	Tracing [1] [2] [7]	Ferramentas integradas que são utilizadas para capturar a latência das requisições feitas aos *containers.*
	Metrics [1] [2] [7]	Ferramentas integradas que são utilizadas para capturar, guardar, consultar e alarmar as métricas dos *containers* e do sistema.
Segurança	Autenticação e Autorização [52]	Protocolos de autenticação e autorização ou *authentication providers* integrados à plataforma.
	Práticas de Segurança [2]	Conjunto de ferramentas e práticas que podem e devem ser utilizadas para garantir a segurança e integridade do sistema.

Tabela 3.3.1: Parâmetros escolhidos para a avaliação qualitativa dos *frameworks serverless*. Fonte (Autor)

A seguir, apresenta-se na *Tabela 3.3.2* a síntese das características, parâmetros anteriormente citados, das plataformas *serverless* pré-selecionadas. Todas as informações não referenciadas foram obtidas das documentações oficiais de cada plataforma durante o mês de agosto de 2021.

Parâmetros	**Knative**	**Openwhisk**	**OpenFaaS**	**ACI**	**Cloud Run**	**Lambda**	**Fargate**
Licença	Apache 2.0	Apache 2.0	MIT	Provider	Provider	Provider	Provider
Pricing	Por cluster – calculado por VMs	Por cluster – calculado por VMs	Por cluster – calculado por VMs	Recursos utilizados por segundo	Recursos usados por tempo de requisição	Recursos usados por tempo de requisição	Recursos utilizados por segundo
Workflows	Kogito [71] (Plugin)	Composer [70] (Plugin)		Azure Logic Apps	Google Workflows	AWS Steps	AWS Steps
Orquestrador	K8s	K8s, Docker, Mesos, Ansible e Vagrant	K8s, Docker e Mesos	Apenas na nuvem pública	Apenas na nuvem pública ou híbrida [75]	Apenas na nuvem pública	Apenas na nuvem pública
Trigger	HTTP, Event, CronJob e WebSocket	HTTP, Event e CronJob	HTTP, Events, CronJob e WebSocket	HTTP e WebSocket	HTTP, Events, CronJob e WebSocket	HTTP, Events, CronJob e WebSocket	HTTP, Events, CronJob e WebSocket
Eventing	Kafka, RabbitMQ, AWS SQS e WebSocket	Kafka	Kafka, Redis, RabbitMQ, NATS e AWS SQS		GCP Pub/Sub	AWS SQS e AWS SNS	AWS SQS e AWS SNS

Interface de uso	CLI, K8s API	CLI, API, SDK	UI, CLI, API	Docker CLI, Azure CLI, SDK, API, Cloud Console e IaC	GCP CLI, Cloud Console, SDK, API e IaC	AWS CLI, Cloud Console, SDK, API e IaC	AWS CLI, Cloud Console, SDK, API e IaC
Registry	Registries Públicos	Docker Hub	Apenas imagens locais	Registries Públicos	GCP Container e Registries públicos	AWS Elastic Container Registry	AWS Elastic Container Registry
Requisitos da imagem		Action Interface [72]	Watchdog [73]			AWS Lambda runtime API [74]	
CI/CD	Tekton	Ferramentas suportadas pelos orquestradores	Gitlab, Jenkins, Circle CI e Travis	Azure DevOps	Google Cloud Build	AWS Developer Tools	AWS Developer Tools
Container SO	Linux	Linux	Linux	Linux, Windows	*Linux*	Amazon Linux	*Linux*
Traffic splitting	Sim	Não	Sim	Não	Sim	Sim	Não
Revisions	Sim	Não	Não	Não	Sim	Sim	Sim

Memória	Alocação padrão do K8s, limites são impostos pelo *cluster*	512 MB	Alocação e limites a depender do orquestrador utilizado	16 GB	8 GB	10 GB	30 GB
CPU	Alocação padrão do K8s, limites são impostos pelo *cluster*	CPU compartilhada do *cluster*	Alocação e limites a depender do orquestrador utilizado	4 vCPUs	4 vCPUs	6 vCPU	4 vCPUs
Storage	Não	Não	Não	Sim	Sim	Sim	Sim
GPU	Sim	Sim	Sim	Sim	Sim [75]	Não	Não
Tempo limite de execução	Ilimitado	5 minutos	Configurável em minutos	Ilimitado	15 minutos	15 minutos	Ilimitado
Execuções concorrentes	Sujeito ao limite do *cluster*	100 requisições concorrentes e no máximo 120 por minuto [76]	Sujeito ao limite do *cluster*	60 requisições concorrentes por grupo de *containers*	1000 requisições concorrentes	1000 requisições concorrentes	1000 *containers* por região [77]

Autoscaling	Sim	Sim, limitado	Sim	Não	Sim	Sim	Sim
Autoscaling metric	Concorrência, RPS e CPU	RPS	CPU e memória		60% CPU e configuração de concorrência	Baseado em concorrência de funções	RPS, CPU, memória e métricas customizadas
Swarming	Sim	Não	Sim		Sim	Sim	
Scale to zero	Sim	Sim	Sim	Não	Sim	Sim	Não
Logging	Fluentbit	Logback	Grafana, Loki, K8s API e System API	Azure Monitor Logs	Google Cloud Logging	Cloud Watch	Cloud Watch
Tracing	Zipkin e Jaeger		System API	Azure Monitor Application Insights	Google Cloud Trace	AWS X-Ray	AWS X-Ray
Metrics	OpenTelemetry e Prometheus	Kamon ou Kafka	Prometeus e *System* API	Azure Monitor	Google Cloud Monitoring	Cloud Watch	Cloud Watch

Autenticação e Autorização		*Basic Auth*	Basic Auth, SSO, Bearer e WebHooks	Azure AD Integration	GCP Authentication		
Práticas de Segurança	TLS e cert-manager		TLS e cert-manager	*Azure security baseline* [78]	TLS/HTTPS, Redirect e Google Autentication	*Security in AWS Lambda* [79]	*Best Practices – Security* [80]

Tabela 3.3.2: Síntese das características das plataformas *serverless.*

De acordo com as tabelas comparativas anteriores, é possível retirar as seguintes conclusões acerca das plataformas:

- Levando em consideração a facilidade de desenvolvimento, um dos motivos que levam a adoção de *serverless* (ver *Seção* 2.3.3.1), as plataformas Apache OpenWhisk, OpenFaaS e AWS Lambda exigem que a imagem Docker implemente alguns padrões ou *templates*. Isto, além de exigir um esforço de implementação, pode dificultar o desenvolvimento de algumas soluções e a migração dos servidores para *containers*.
- Ainda com base nos motivos de adoção de plataformas *serverless*, a Azure *Container Instances* é a única plataforma que não possui a característica de *autoscaling,* segunda razão mais citada na *Secção 2.3.3.1*.
- Levando em consideração um dos objetivos do trabalho, a construção de aplicações escaláveis e reforçando a capacidade de *autoscaling*, destacam-se as plataformas Knative e Fargate pelas variedades de métricas que podem ser usadas para configuração de escalabilidade.
- No geral entre todas as plataformas, Apache OpenWhisk é a que tem menos funcionalidades. Possui uma escalabilidade limitada a 120 requisições por minuto, permite apenas 512MB de memória para os *containers*, tem o menor número de integrações em relação a sistemas de mensagens (utiliza-se apenas do Apache Kafka) e oferece pouco suporte a ferramentas de monitoramento. É importante também, acrescentar que a plataforma possui a documentação mais incompleta entre as avaliadas.
- Quanto aos custos, é notório que as plataformas *open source* apresentam maiores gastos, pois utilizam *clusters* de máquinas virtuais (considerando o custo de gestão e das máquinas virtuais). Ainda sobre o modelo de faturação, as plataformas Google Cloud Run e AWS Lambda, cobram por requisição, o que torna esses sistemas elásticos também em relação às despesas, ou seja, o custo é proporcional à carga de trabalho aplicada no sistema.
- As plataformas *open source* não suportam *mounting* de volumes nos *containers*.
- Apache *OpenWhisk* e *Azure Containers Instance* não apresentam suporte a versionamento de *deployments* e *traffic splitting* entre os *containers*.

- Com exceção do *framework* Apache OpenWhisk e OpenFaaS que não suportam o protocolo *WebSockets,* todos os outros suportam os mesmos *triggers*: HTTP, *WebSockets*, *CronJobs* e eventos.
- Quanto à forma e tempo de execução, Azure *Containers Instancies* e AWS *Fargate* funcionam de forma tradicional (*containers* ativos continuamente), por isso não fazem *Scale to Zero* e não possuem limite de tempo de execução. Diferente destas anteriormente citadas, as demais plataformas iniciam o processo de atribuição de recursos para os *containers* apenas quando há requisições e possuem limite de tempo para cada requisição. Adicionalmente, as plataformas Knative, OpenFaaS e *Cloud Run* apresentam uma vantagem, pois podem ser configuradas com um número mínimo de *containers*, ou seja, funcionam de forma híbrida. Esta capacidade permite, por exemplo, que esses sistemas contornem o problema de *cold starts* e tenham mais performance.
- De uma forma geral, todos apresentam orientação a eventos, pois integram com sistemas de mensagens e possuem capacidade de composição (exceto a plataforma OpenFaaS).
- Os serviços de nuvem pública seguem todos a tendência de integrarem muitos dos seus produtos às suas respetivas plataformas *serverless*, como por exemplo, os sistemas de mensagem, monitorização e autorização.
- Quanto aos mecanismos de segurança das plataformas, os mais comuns são: autenticação e suporte a HTTP. Além disso, os provedores de computação em nuvem pública também oferecem guias de boas práticas como configurações de rede e escaneamento de vulnerabilidades nas imagens dos *containers.*

3.4 Benchmark das plataformas *serverless*

Dado que as plataformas abordadas podem apresentar diferentes desempenhos, além de uma avaliação qualitativa, é necessário estabelecer uma avaliação quantitativa com a finalidade de esclarecer ao leitor ou utilizador todas as características performáticas das plataformas, pois desta forma é possível concluir se estas estão aptas a corresponder aos requisitos e necessidades de um sistema escalável.

Posto isso, nesta secção, são apresentados alguns *benchmarks* existentes de trabalhos que comparam plataformas *serverless* e com base nestes é definida a

experiência prática de *benchmark* para o presente trabalho de acordo com os objetivos já apresentados.

3.4.1 *Benchmarking* existentes

Considerando o tempo ao qual já existem as plataformas *serverless* que implementam funções, a utilização de *containers* pode ser apontada como recente. Por este motivo, é comum encontrar inúmeros trabalhos com experiências de performance que testam plataformas de funções, mas poucos destes propõem o uso de *containers*. Porém, já que as plataformas *serverless* que utilizam *containers* e funções possuem as mesmas características, é possível aplicar os mesmos parâmetros e experiências para avaliar a performance destes frameworks [15] [53]. Desta forma, são relevantes para a presente revisão de literatura artigos que implementam FaaS e CaaS.

O artigo "*Benchmarking Serverless Computing Pl*atforms" [52] é um dos exemplos de trabalhos sobre *serverless* que focam plataformas de funções. Com o objetivo de avaliar diferentes *frameworks*, o trabalho baseia-se em diversas experiências relatados na literatura, para definir um conjunto coerente e unificado de testes. Dessa forma e através de um levantamento da literatura, foram definidos sete testes que avaliam escalabilidade, o efeito da memória atribuida, o desempenho para casos vinculados à CPU, o impacto do tamanho da carga útil, a influência da linguagem de programação, gestão de recursos (por exemplo, reutilização de *containers*) e a sobrecarga geral da plataforma. Além disso, foi desenvolvido um *software open source* para teste de forma automática das plataformas e exposto os resultados da comparação entre as plataformas AWS Lambda, Google Cloud Functions, Microsoft Azure Functions e IBM Cloud Functions.

Tratando-se de plataformas *serverless* que implementam *containers*, destaca-se o trabalho "*Serverless Containers -- rising viable approach to Scientific Workflows*" [15] que estende o trabalho anterior [53] (o qual analisa o uso de funções *serverless* para execução de *workflows* científicos) para os serviços de nuvem pública Google Cloud Run e Amazon Fargate, utilizando as mesmas experiências e os parâmetros, como citado anteriormente. Além de demonstrar a usabilidade destes *frameworks*, o trabalho cria *benchmarks* de custo e *performance*, os quais medem a latência, elasticidade e escalabilidade dos mesmos, usando quatro *workflows* de trabalho científicos que incluem tarefas intensivas de CPU e dados (Ellipsoids, Vina, KINC e

Soy-KB) e testando várias configurações de quantidade de memória atribuida e CPU para uma determinada experiência. Por fim, as experiências provaram que as plataformas de *containers serverless* podem ser aplicadas com sucesso em *workflows* científicos e deduziram uma série de características e limites das mesmas. Ainda sobre "*Serverless Containers -- rising viable approach to Scientific Workflows*" [15], os autores caracterizam duas formas de *design* de implementação dessas plataformas, quanto à forma de mapear a execução de tarefas nos *containers*: arquitetura de *queue-workers* e *one-to-one task to container mapping*. Sobre a arquitetura de *queue-workers*, embora possa ser implementada facilmente, não usufrui dos benefícios do modelo de computação *serverless,* pois as decisões de escalonamento automático precisam ser feitas ao lado do cliente para evitar o aprovisionamento insuficiente ou excessivo. Em contrapartida, o padrão de *one-to-one task to container mapping* cria para cada tarefa um *container* isolado o qual é finalizado após a conclusão da tarefa. Este mapeamento simplifica o *design* e aprimora totalmente os recursos do modelo *serverless*, em que a infraestrutura em nuvem é responsável pelo aprovisionamento de recursos, ou seja, a responsabilidade de escalonamento passa a ser do sistema *serverless*. Por fim, este mapeamento também simplifica o uso de recursos dentro de um *container* e produz tempos de execução mais estáveis, mas tem, porém, um custo adicional de inicialização do *container* para cada tarefa (*cold start*).

3.4.2 *Benchmark* proposto

Como nas experiências executadas nos trabalhos referenciados na subsecção anterior, o presente trabalho propõe um conjunto de sete testes que visam avaliar e comparar a *performance* das plataformas em diferentes aspetos.

As experiências propostas, têm como base as diversas referências da literatura listadas no trabalho "*Benchmarking Serverless Computing Platforms*" [52]*.* De uma forma geral, a maioria dos testes utilizam-se de estatísticas sobre a execução de requisições aos sistemas *serverless*, como tempo de execução e porcentagem de sucesso. Como já referenciado anteriormente, apesar da grande maioria da literatura dos *benchmarks* avaliar *serverless functions*, as experiências podem ser os mesmos para o modelo de containers, como sugerido em "Serverless Containers -- rising viable approach to Scientific Workflows" [15].

Ainda sobre os testes que serão executados, e de acordo com a problemática apresentada no capítulo de introdução, o trabalho tem como objetivo avaliar a capacidade das plataformas *serverless* responderem ao aumento do fluxo de solicitações nos sistemas informáticos. Os testes base foram modificados a fim de se introduzir uma terceira variável: utilizadores virtuais, em inglês *virtual users* (*vus*). Em resumo, as experiências avaliarão o comportamento das plataformas, medido pelo tempo de resposta em relação ao aumento de tráfego e, em alguns testes, uma terceira variável independente.

De acordo com o que foi anteriormente referido, seguidamente, descrevem-se os sete testes propostos no presente trabalho:

- **Teste de carga (T1)**: calcula a sobrecarga da plataforma *serverless* após uma sequência de chamadas para uma requisição, cujo esforço computacional é baixo. As chamadas serão executadas durante um determinado período e, exclusivamente neste teste, por um utilizador virtual. Para cada requisição, será recolhido o tempo de resposta da plataforma para posterior análise. Este teste, é uma experiência curta e tem como objetivo apenas perceber a latência ponta a ponta de cada uma das plataformas;
- **Teste de carga simultânea (T2)**: para avaliar a escalabilidade das plataformas, este teste mede a latência, a quantidade de requisições e a taxa de sucesso de vários utilizadores em paralelo. Este teste é semelhante ao primeiro, com a diferença que a quantidade de requisições e clientes cresce em paralelo ao decorrer da experiência. É a base para as experiências T4, T5, T6 e T7;
- **Teste de pico (T3)**: avalia a elasticidade das plataformas e a capacidade de responderem a piscos de carga. É semelhante a T2, porém a execução do teste é maior e atinge mais utilizadores em paralelo. O teste tem uma fase crescente e uma segunda fase decrescente de requisições para que o sistema possa tanto executar *scale-out* quanto *scale-in*;
- **Teste de tamanho de carga (T4)**: tem como base o teste T2, onde se avalia a escalabilidade da plataforma em relação ao aumento de utilizadores em paralelo, e uma terceira variável: tamanho do *payload* das requisições. Com isso, visa-se observar o impacto do tamanho (I/O) e da quantidade de requisições nas plataformas;

- **Teste de linguagem de programação (T5)**: da mesma forma que T4 tem o teste T2 como base, mas com a diferença que a variável independente é a linguagem de programação utilizada. Desta forma, será possível perceber como diferentes *containers* ou linguagens podem afetar o tempo de resposta e escalabilidade das aplicações;
- **Teste de tamanho de memória/CPU (T6)**: tem o objetivo de determinar a influência da memória e CPU atribuidos no desempenho das requisições. De forma resumida, executa o mesmo teste base (T2) para diferentes e proporcionais valores de memória e CPU;
- **Teste intensivo de computação (T7)**: mede o desempenho das plataformas ao executar requisições com intensivo uso de computação e memória usando uma função de Fibonacci recursiva. O teste tem como base o T2, e correlaciona diferentes números de Fibonacci, com o aumento de utilizadores virtuais.

3.4.3 Plataformas selecionadas para a experiência

Para obter conclusões de maior valor e diminuir o domínio da experiência, serão avaliadas apenas as plataformas com maior relevância de acordo com as características das plataformas serverless.

Entre as plataformas *open source* analisadas anteriormente, apenas a Knative é elegível para a experiência, pois, apesar de as demais possuírem suporte à execução de *container*, dependem de *custom images templates*, o que dificulta o desenvolvimento de aplicações. Tratando-se das plataformas de nuvem pública, AWS Lambda e Azure *Containers Instances* foram removidas por exigir *custom images templates* e não ter *autoscaling,* respetivamente.

Posto isso, serão avaliadas as plataformas: **Knative, Google Cloud Run e AWS Fargate** que foram selecionadas a partir das conclusões sobre a avaliação qualitativa da secção 3.3 e considerando, principalmente, os parâmetros de escalabilidade e fácil implementação de *containers*.

Capítulo IV – Execução da experiência de *benchmark*

O objetivo deste capítulo é descrever em detalhe a experiência prática que foi abordada no *Capítulo III*. Como definido, a experiência possui um conjunto de 7 testes que avaliam diferentes características das plataformas *serverless* selecionadas.
A *Secção 4.1* apresenta a forma, arquitetura, ferramentas e configurações utilizadas para execução. Em seguida, na Secção 4.2*,* apresentam-se os parâmetros utilizados e resultados obtidos para cada teste. Na *Secção 4.3* há uma estimativa de custo das plataformas e da experiência. Por fim, na *Secção 4.4* são apresentadas conclusões finais sobre o *benchmarking*.

4.1 Arquitetura da experiência

Para execução da experiência, é necessário provisionar a infraestrutura das plataformas, configurar os componentes responsáveis pelo *benchmarking*, executar requisições HTTP, medir os tempos de resposta e armazená-los para posterior consulta e análise. Com o objetivo de abordar todos estes processos, podemos dividir a experiência em três partes principais: máquina de *deploy*, plataformas em nuvem e máquinas de *benchmarking.*

4.1.1 Máquina de *deploy*

Apesar das mais-valias que os sistemas *serverless* apresentam quanto ao aprovisionamento e gestão de infraestrutura, a composição da presente experiência é uma tarefa extensa e suscetível a erros, pois a necessidade de ser implementada em múltiplas nuvens e com diversos parâmetros, que podem mudar de acordo com o teste em causa, torna a configuração complexa implicando consumo de tempo e dinheiro.
Tendo em conta estas dificuldades, a experiência foi preparada utilizando-se de práticas de DevOps; estas práticas foram implementadas para automatizar e integrar os processos entre o desenvolvimento do software dos testes e a configuração da infraestrutura das plataformas *serverless*, para que, com isso, se consiga implantar, lançar e executar os testes de forma mais rápida, confiável e, consequentemente, mais barata.
Dessa forma, criou-se um repositório de código que define toda a infraestrutura com a ajuda da ferramenta Terraform [54]. Além de definir a infraestrutura em nuvem, o

repositório também contém as definições de imagens Docker das aplicações, ferramentas e diversos *scripts* de automação de tarefas utilizados durante o teste.

Por fim, através da configuração do repositório numa máquina local, das suas dependências e credenciais de acesso a nuvens públicas, a experiência é preparada para execução. Segue a ordem de passos executados localmente para *deploy* da infraestrutura do teste:

1. Configuração do repositório, dependências e autenticação nas nuvens públicas;
2. Autenticação em repositórios de imagens de *containers, build* e *push* das imagens necessárias;
3. Execução da ferramenta Terraform para criação da infraestrutura das plataformas e infraestrutura responsável pelo *benchmarking*.
4. Execução dos testes;
5. Quando necessário, alteração e execução da ferramenta Terraform para variar parâmetros da infraestrutura requeridos nos testes (como memória, CPU e imagem de aplicação);
6. Por fim, após a execução de todos os testes e análises, executar o comando *destroy* do Terraform para remover a infraestrutura e evitar custos.

4.1.2 Plataformas em nuvem

Após a execução do comando Terraform, para o aprovisionamento da infraestrutura, serão criados os sistemas das três plataformas selecionadas e as máquinas de *benchmark* em duas nuvens públicas.

Sobre a plataforma *open source* Knative, a qual depende do orquestrador de *containers* K8s, foi decido que a mesma seria implantada no serviço Google Kubernets Engine (GKE). Esta decisão considera o fato que o produto *Google Cloud Run* se baseia em Knative e, uma vez que esta plataforma tem características, recursos de computação e rede similares, espera-se uma comparação mais justa de resultados, em que fatores como latência da rede e performance de *hardware* são menos relevantes.

Sobre a disposição global das plataformas, como é necessário um banco de dados comum para armazenar os dados obtidos das requisições, foram escolhidas duas regiões próximas entre si, para minimizar a latência e evitar qualquer problema durante a execução dos testes. Com esse intuito, foram escolhidas as regiões da

europa AWS Stockholm, Sweden (eu-north-1) e GCP *Hamina, Finland* (europe-north1). A Figura 4.2 Exibe a disposição geográfica das mesmas com uma distância de 526km. Além a disposição, para que seja desconsiderada a latência geográfica, as máquinas de *benchmarking* foram implantadas na mesma região e rede das plataformas.

Figura 4.1.1: Distância entre as regiões dos datacenters. Fonte (Autor)

Sobre as arquiteturas implementadas nas nuvens públicas, encontram-se na *Figuras 4.1.2* e *Figura 4.1.3* os principais componentes de cada plataforma a ser utilizada na experiência.

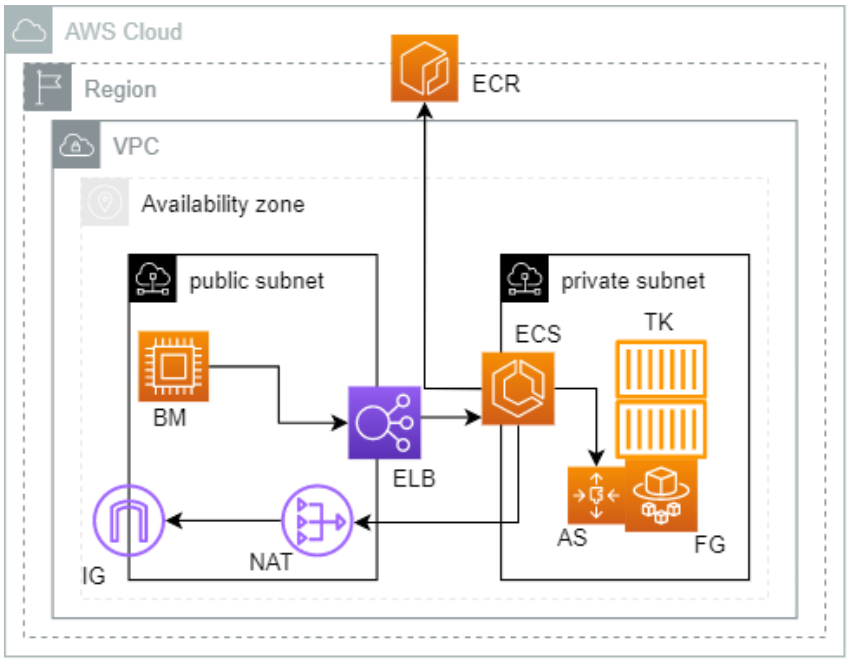


Figura 4.1.2: Arquitetura implantada na nuvem pública da AWS.
Fonte(Autor)

Apesar da nuvem da AWS ter apenas uma plataforma *serverless* a ser implementada, a sua configuração exige uma maior complexidade quanto à disposição de rede. Além de definir uma *Virtual Private Networking* (VPC), é necessário a criação de duas *subnets* em cada zona: uma pública e outra privada. Na *private subnet*, são definidos os componentes *serverless* responsáveis pela execução dos *containers* ou *tasks* (TK), sendo estes a criação de um *cluster* ECS com Fargate (FG), regras de *autoscaling* (AS) e de um *container registry* (ECR). Para a *public subnet*, são definidos os componentes de rede Internet *Gateway* (IG), NAT e um *load balancing* interno (ELB). Responsáveis por permitir conexões com a internet, criar uma ponte entre *subnets* e permitir rotear as chamadas HTTP da *subnet* pública para os serviços ECS situados na *subnet* privada, respetivamente. Por fim, como exibido na Figura 4.1.2, a máquina virtual de *benchmarking* (BM) encontra-se na *public subnet* e, consequentemente, possui acesso à Internet e ao *load balancing* que permite acesso aos *containers* do *cluster* ECS Fargate.

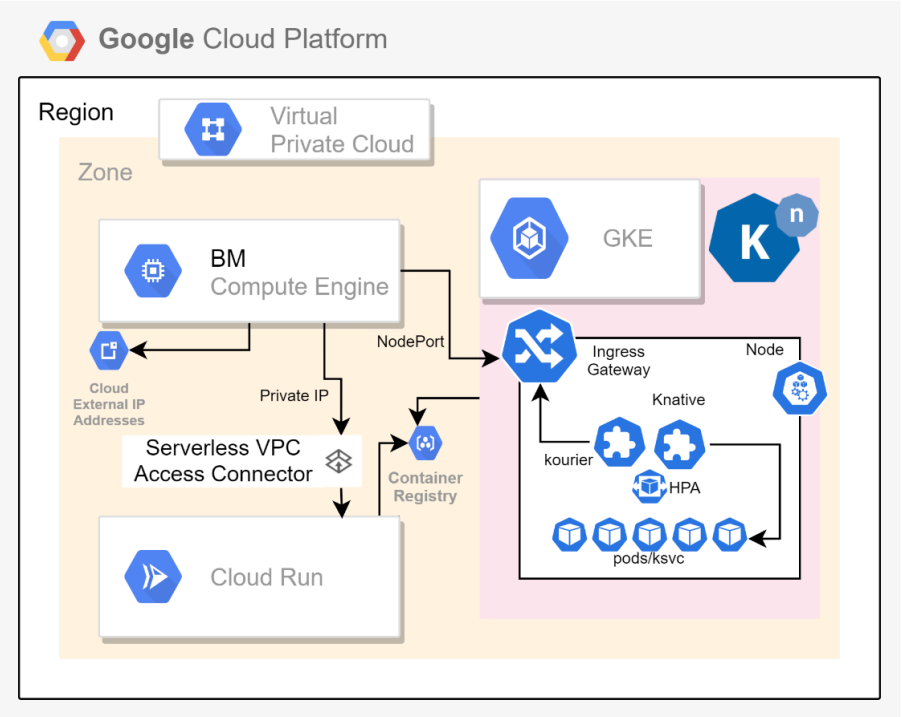


Figura 4.1.3: Arquitetura de nuvem pública implantada na GCP.

Fonte (Autor)

Tratando-se da nuvem na GCP, desenhada no *diagrama da Figura 4.1.3*, são configuradas duas plataformas, numa mesma região e rede: *Cloud Run* e Knative. O *Cloud Run*, além da configuração dos *containers* pelo próprio serviço, requer apenas um recurso chamado de "*Serverless VPC Access Connector*" para estar na mesma rede que os demais agentes. Logo que esse recurso esteja configurado, os *containers* são acedidos pela mesma VPC através de um DNS gerido pela nuvem. Quanto ao Knative, a plataforma é configurada sobre um *cluster* GKE de 3 VMs e2-standard-4 (16GiB de memória e 4vCPUs) e pode ser acedida através de um Ingress, que utiliza um serviço do tipo NodePort, ou seja, através de uma porta em qualquer nó do cluster K8s. Ainda sobre o Knative, internamente as requisições são roteadas pelo *service mesh* Kourier para os containers (*ksvc*), que por sua vez tem sua escalabilidade gerida pelo recurso de *Horizontal Pod Autoscaling* (HPA). Por fim, existem também os serviços de Container Registry e uma máquina virtual para benchmarking (BM) associada a um IP externo, endereço que é utilizado para receber e armazenar as métricas dos testes executados na outra máquina virtual de *benchmarking*, situada na AWS, no mesmo banco de dados.

Em adição à arquitetura e disposição dos componentes, é necessário definir as configurações padrões de cada plataforma, para que as mesmas possam ser capazes de responder a um teste de carga com a mínima configuração requerida, e ao mesmo tempo acentuem as suas similaridades para haver uma comparação justa. Deste modo, as plataformas foram configuradas com as mesmas métricas de escalonamento horizontal (60% de CPU), possuem *containers* com a mesma quantidade de recursos de memória (512MB) e valores proporcionais CPU, não utilizam *scale to zero*, a fim de evitar *cold starts* e utilizam apenas um *container* como configuração inicial de paralelismo. Também é importante mencionar que é mantido um longo tempo de execução entre os testes para que as plataformas possam voltar à configuração inicial de um *container*.

4.1.3 Máquinas de *Benchmarking*

Uma vez provisionadas as infraestruturas das plataformas, as máquinas de *benchmarking* são responsáveis por automatizar a execução dos testes. A *Figura 4.3* apresenta a descrição interna e o fluxo de execução dos testes. As máquinas foram configuradas para que no momento do arranque executem um comando *docker-compose*, para a criação de dois *containers*: o primeiro implementa um banco

de dados temporais (InfluxDb) [18], utilizado para armazenar as métricas, e o segundo utiliza uma imagem customizada Jupyter Notebook [55], um servidor usado para execução de *scripts* via *interface web*.

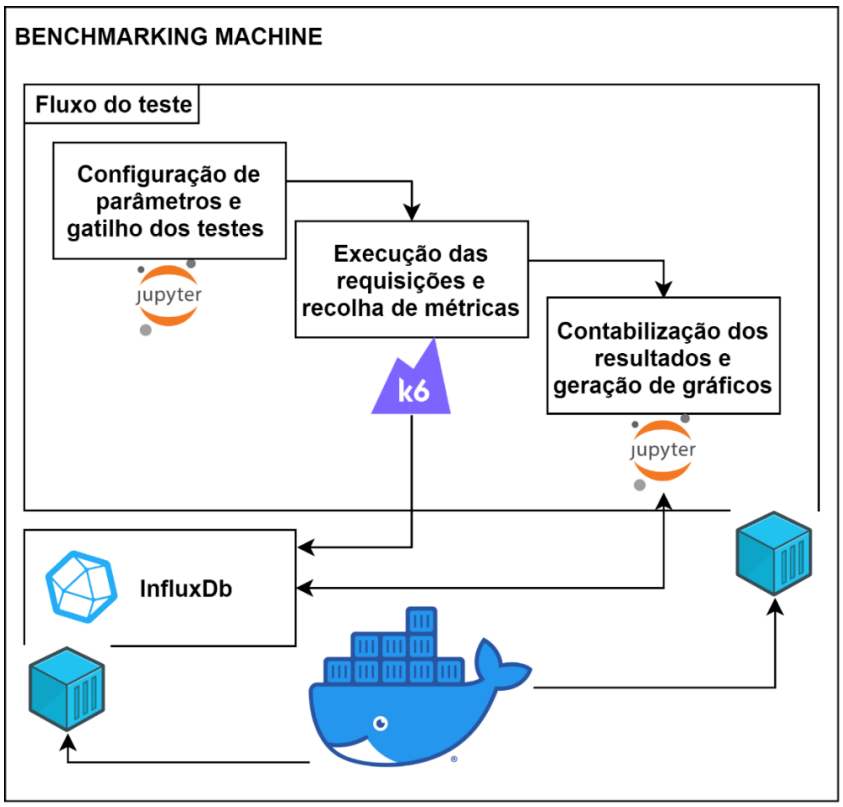


Figura 4.3: Arquitetura interna da BM e fluxo do teste. Fonte (Autor)

Conforme já foi mencionado, e a fim de centralizar os dados, o *container* InfluxDB é utilizado apenas na nuvem GCP e recebe métricas da BM situada na AWS; por este motivo, a BM da GCP tem um IP e uma porta aberta a conexões via internet. Além do endereço para comunicação de dados, ambas as BM possuem outra porta exposta e mapeada junto ao *container* do servidor de scripts Jupyter, que oferece uma *interface* web amigável para a execução dos testes.

Além de uma forma visual de executar os testes, a imagem do *container* Jupyter empacota todas as dependências do *benchmarking* como: scripts de automação, arquivos de configurações e, o principal agente dos testes, o aplicativo k6.io [16], que é uma ferramenta de *load testing* utilizada para simular o tráfego e obter métricas de sistemas. Logo que todos os componentes estejam prontos, é executado, pela *interface* do Jupyter Notebook, um *script bash* que ativa o aplicativo k6.io. O *script* tem como parâmetros variáveis que serão utilizadas na execução do teste e enviadas para o banco de dados para identificação das métricas, e que são: URL para

requisição, nome da plataforma, identificador do teste, identificador de execução e uma chave/valor a ser utilizada em casos de testes que utilizam uma terceira variável independente. Durante a execução do aplicativo k6.io, o mesmo é responsável por conectar-se e enviar as métricas de cada requisição para a base de dados InfluxDB. Por fim, após a execução do teste nas plataformas selecionadas, são executados na interface Jupyter *scripts* Python que recuperam as métricas do *benchmarking* (por meio de *queries* a base de dados) e contabilizam dados estatísticos e gráficos sobre as mesmas.

4.2 Configuração dos testes e resultados

A presente secção, tem como objetivo apresentar os resultados do *benchmarking* para cada um dos testes definidos na *Subsecção 3.4.2*, em acordo com a infraestrutura, configurações e fluxo definidos na secção anterior.

4.2.1 T1 - Teste de carga

Para calcular a sobrecarga das plataformas *serverless* após uma sequência de chamadas, são feitas requisições a um *endpoint* HTTP GET de baixo custo computacional que devolve "*Hello World*" e é implementado utilizando o *framework* JavaScript Node.js. As chamadas são executadas durante um período de um minuto por um utilizador virtual (vu). Os *containers* das plataformas Cloud Run e Knative foram configurados com 512MB/1CPU e da plataforma AWS Fargate com 512MB/0,25 CPU. Segue a *Tabela* 4.2.1 com o resultado das requisições feitas para as plataformas.

Plataforma	Cloud Run	Knative	AWS Fargate
Latência mínima (ms)	7,67	1,01	1,4
Latência máxima (ms)	156,26	59,61	20,38
Latência média (ms)	9,75	1,47	1,96
Latência 95º *percentil* (ms)	10,88	2,44	2,77
Latência 90º *percentil* (ms)	10,49	1,66	2,72
Nº de requisições	6096	39087	29443
% de sucesso	100	100	100
Requisições por segundo	101,59	651,43	490,69

Tabela 4.2.1: Resultados do teste de carga sobre as plataformas. Fonte (Autor)

E os respectivos gráficos para o teste em questão de cada plataforma. Nos gráficos, o eixo *y* corresponde à latência em milissegundos das requisições e o eixo *x* ao momento do teste em segundos que as chamadas foras executadas.

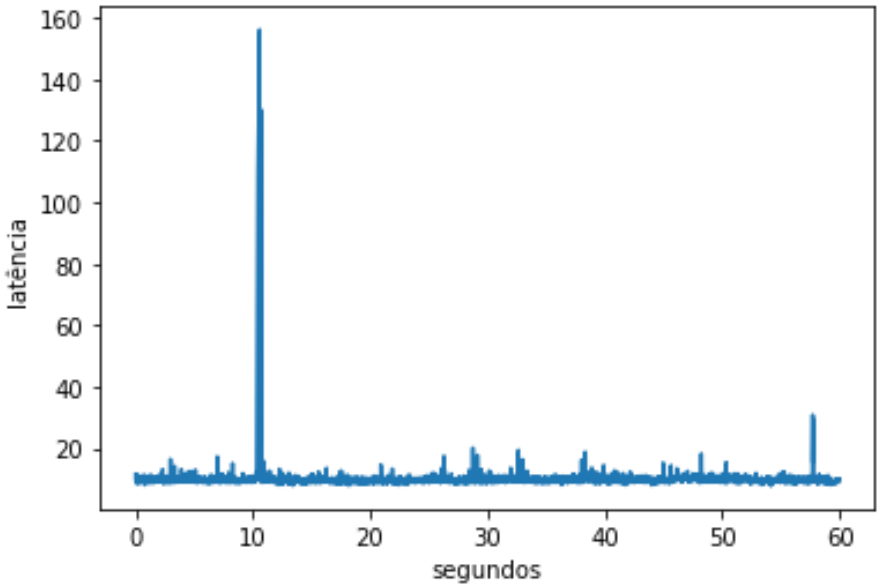


Figura 4.2.1.1: Gráfico de latência da plataforma Cloud Run durante o T1. Fonte (Autor)

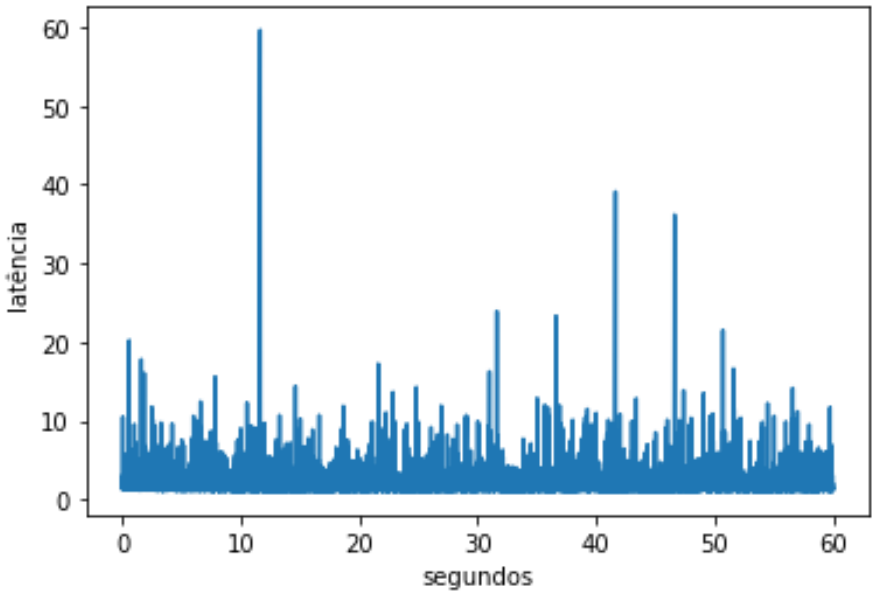


Figura 4.2.1.2: Gráfico de latência da plataforma Knative durante T1. Fonte (Autor)

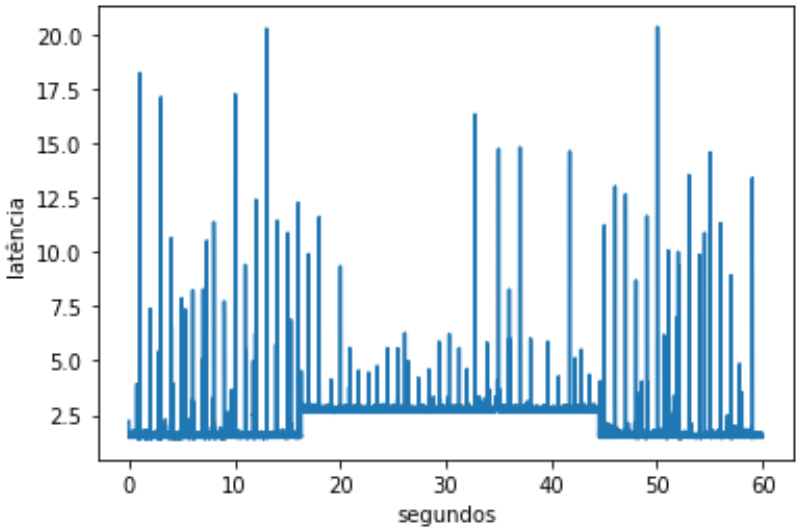


Figura 4.2.1.3: Gráfico de latência da plataforma AWS Fargate durante T1. Fonte (Autor)

De acordo com os resultados anteriores, é possível inferir as seguintes conclusões:

- Entre as plataformas, Cloud Run apresentou a pior *performance* entre as latências médias (9,75ms). Knative e ECS apresentaram *performances* bem parecidas (1,47ms e 1,96ms, respectivamente) como uma ligeira vantagem para a plataforma Knative;
- Ao analisar a *Figura 4.2.1.1*, que contém o gráfico referente à plataforma *Cloud Run*, é possível observar um pico na latência próximo ao momento dos 10 segundos. Esse *outlier* e outros picos menores, apesar de não causarem erros, diminuíram consideravelmente a capacidade de resposta da plataforma, que executou apenas 6096 requisições representando apenas 15% e 20% das outras plataformas;
- Na *Figura 4.2.1.3* por volta do 45º segundo, percebe-se uma melhoria nos tempos de resposta da plataforma AWS Fargate, provavelmente em virtude de um processo de *scale-out bem* executado;
- Considerando a amplitude entre os valores de máximo e mínimo e o desvio entre os 90º e 95º *percentis,* constata-se que a plataforma AWS Fargate apresenta uma maior estabilidade quanto as suas requisições. Knative, apesar de apresentar menor latência, é menos estável em comparação com AWS Fargate. Por fim, a plataforma *Cloud Run* demonstra ser instável, uma vez que apresenta uma grande amplitude e desvio.

4.3.2 T2 - Teste de carga simultânea

Para avaliar a escalabilidade das plataformas, este teste mede o impacto na latência quando vários utilizadores virtuais (*vus*) em paralelo fazem requisições sobre as plataformas. É semelhante ao primeiro teste (T1), executa requisições por 60 segundos a um *endpoint* de baixo custo computacional (“Hello World” implementado pelo *framework* JavaScript Node.js), porém o número *vus* cresce ao decorrer do teste com uma frequência de um utilizador por segundo, indo de 1 *vu* no início do teste até 60 *vus* ao final da rotina. Os *containers* das plataformas mantêm a configuração de recursos padrão, 512MB/1CPU para *Cloud Run/Knative* e 512MB/0,25CPU na plataforma AWS Fargate.

Segue a *Tabela* 4.2.2 com as estatísticas sobre as requisições com crescentes *vus* executados sobre as plataformas.

Plataforma	Cloud Run	Knative	AWS Fargate
Latência mínima (ms)	5,41	0,68	1,37
Latência máxima (ms)	790,22	104,75	114,06
Latência média (ms)	11,56	10,13	12,47
Latência 95º *percentil* (ms)	26,26	23,13	60,49
Latência 90º *percentil* (ms)	15,65	18,49	52,66
Nº de requisições	151700	173437	140854
% de sucesso	100	100	100
Requisições por segundo	2528,21	2890,51	2347,34

Tabela 4.2.2: Resultados do teste de carga simultânea sobre as plataformas.

Fonte (Autor)

A *Figura 4.2.2* apresenta no mesmo gráfico os resultados dos testes nas três plataformas. O gráfico representa a latência em milissegundos das requisições (eixo *y*) em relação à quantidade de *vus* ativos no momento das respectivas requisições.

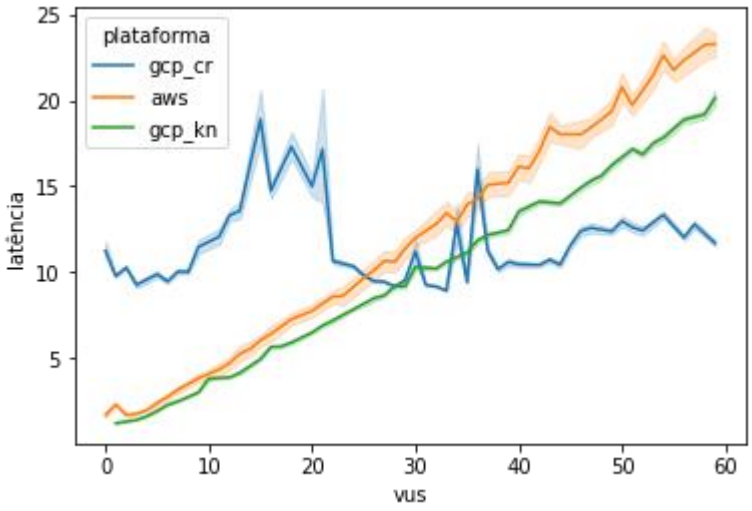


Figura 4.2.2: Gráfico de latências das plataformas em relação ao número de vus ativos no teste T2. As legendas *gcp_cr*, *aws* e *gcp_kn* representam as plataformas Cloud Run, AWS Fargate e Knative, respectivamente.
Fonte (Autor)

De acordo com os resultados anteriores, é possível inferir as seguintes conclusões:

- Considerando a latência média, a quantidade de requisições e o percentual de acerto, todas as plataformas têm resultados semelhantes. Destaca-se que a plataforma Knative continua a demonstrar a menor latência média (10,13ms) e que a plataforma *Cloud Run* tem sua eficiência melhorada, apresentando o segundo melhor resultado, 11,56ms contra 12,47ms da plataforma AWS Fargate;
- A plataforma *Cloud Run* continuou a apresentar instabilidade, fato que pode ser perceptível pela amplitude, pelo desvio dos *percentis* e gráfico sinuoso na *Figura 4.2.2*;
- Sobre a crescente quantidade de utilizadores virtuais, pode-se inferir que a plataforma *Cloud Run* é a que melhor respondeu ao aumento das requisições em paralelo, pois apresenta valores quase constantes durante o teste, caso se desconsidere os seus picos e vales que caracterizam a sua instabilidade. Essa resposta melhorada influenciou positivamente a latência média e quantidade de requisições em comparação ao teste T1;
- Ainda sobre a capacidade de responder ao acréscimo de utilizadores virtuais, as plataformas Knative e AWS Fargate demostram uma proporção entre

latência e *vus*. Sobre essas proporções ou funções lineares, a plataforma AWS Fargate possui um maior coeficiente angular se comparada à plataforma Knative, ou seja, a latência da plataforma AWS Fargate tem a tendência de crescer mais que a latência da plataforma Knative quando há o aumento das requisições em paralelo.

4.3.3 T3 - Teste de pico

O teste de pico ou, em inglês, *spike test* tem o objetivo de avaliar a elasticidade e a capacidade dos sistemas responderem a picos de carga. Este teste utiliza o mesmo *endpoint* e configurações de memória e CPU do teste T2, porém a execução do teste é mais longa e atinge uma maior quantidade de *vus*. O teste tem uma fase crescente em *vus* e uma segunda decrescente em *vus* e prevê que o sistema execute *scale-out* quanto *scale-in*. Especificamente, a execução do teste é dividida em 5 períodos de 1 minuto cada, cada período tem uma taxa de crescimento constante de *vus* com o objetivo de alcançar 60, 120, 240, 120 e 60 *vus* no final de cada período. A relação entre tempo de execução e quantidade de utilizadores virtuais pode ser visualizada através do gráfico na *Figura 4.2.3.1.*

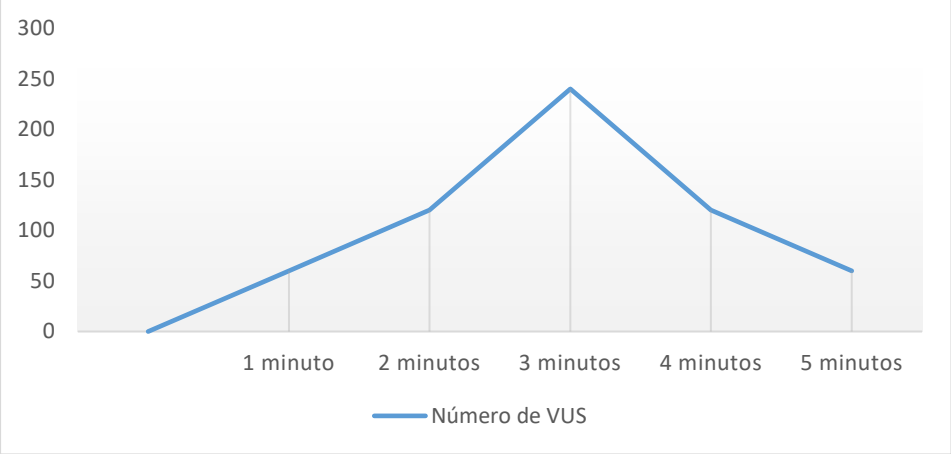


Figura 4.2.3.1: Gráfico da relação entre tempo de execução e a quantidade de *vus* no teste T3. Fonte (Autor)

A seguir, a *Tabela* 4.2.3 apresenta as estatísticas dos *spike tests* das plataformas em análise.

Plataforma	Cloud Run	Knative	AWS Fargate
Latência mínima (ms)	5,23	0,64	0,57
Latência máxima (ms)	1200	303,97	275,59
Latência média (ms)	24,24	14,03	40,63
Latência 95º *percentil* (ms)	56,94	37,29	96,9
Latência 90º *percentil* (ms)	45,45	28,65	88,59
Nº de requisições	1402617	2410359	838909
% de sucesso	99,99	99,99	100
Requisições por segundo	4675,34	8034,45	2796,31

Tabela 4.2.3: Resultados do teste de pico sobre as plataformas. Fonte (Autor)

As figuras a seguir apresentam os resultados do teste de pico (T3) para as plataformas. O gráfico representa a latência em milissegundos das requisições (eixo *y*) em relação ao momento em segundos do teste (eixo *x*).

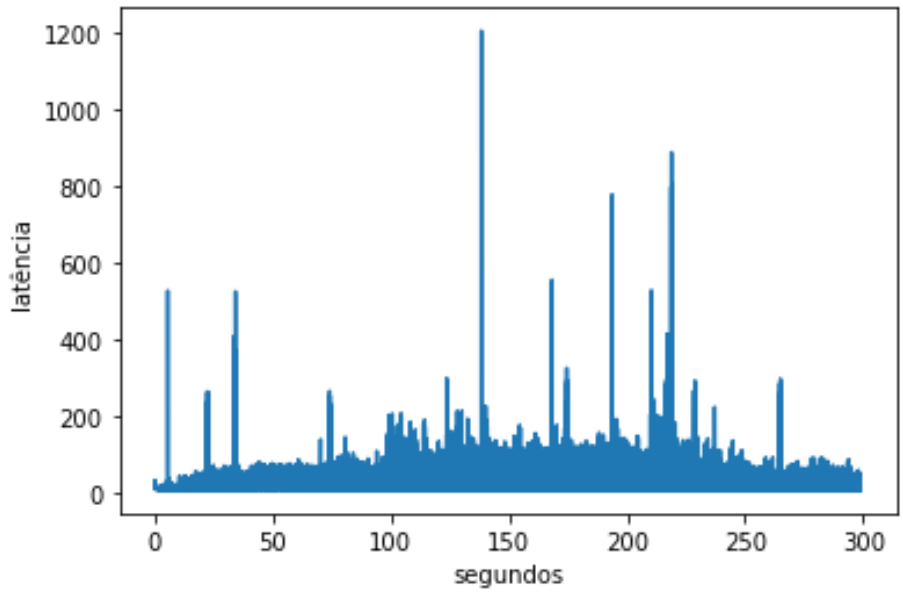


Figura 4.2.3.2: Gráfico de latência da plataforma Cloud Run durante o teste T3. Fonte (Autor)

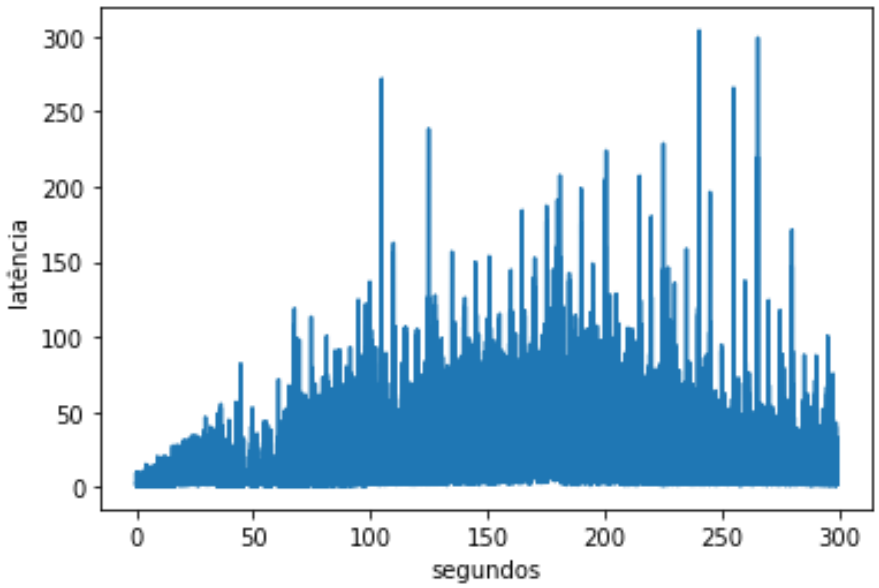


Figura 4.2.3.3: Gráfico de latência da plataforma Knative durante o teste T3. Fonte(Autor)

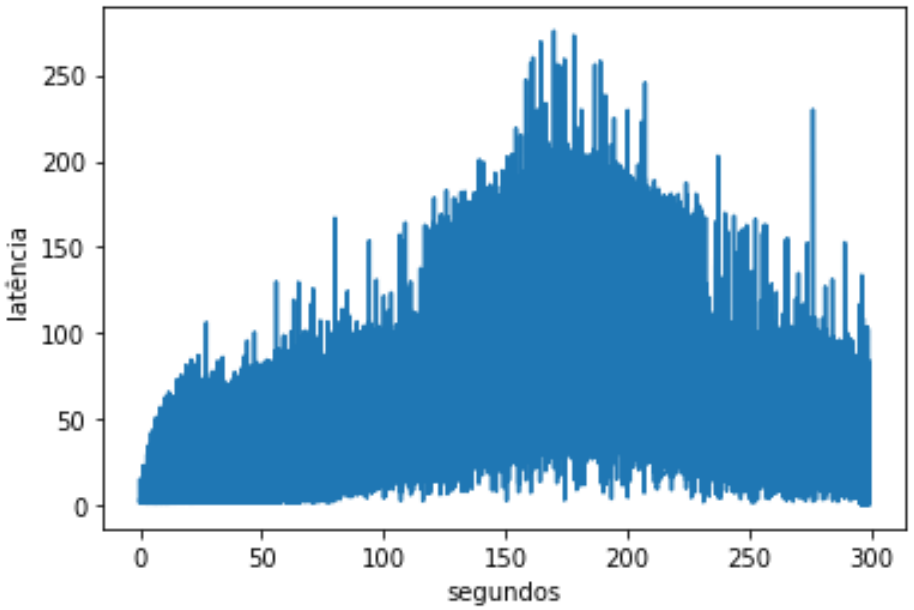


Figura 4.2.3.4: Gráfico de latência da plataforma AWS Fargate durante o teste T3. Fonte (Autor)

De forma composta, a *Figura 4.2.3.4,* condensa os três gráficos anteriores, de cada plataforma, em apenas uma imagem.

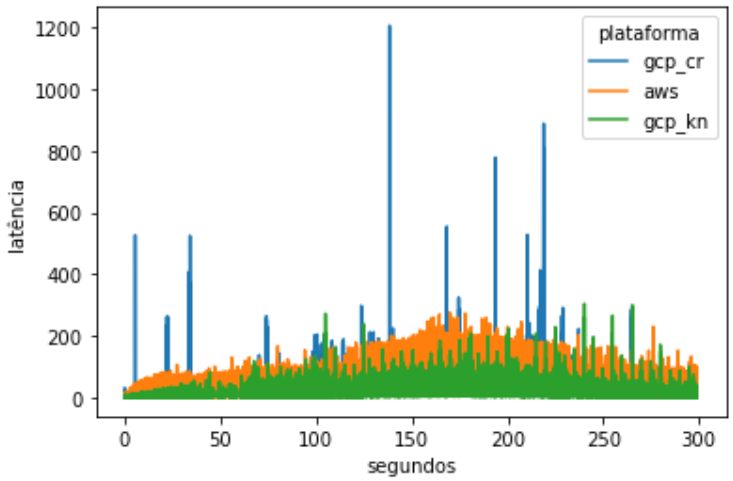


Figura 4.2.3.5: Gráfico de latência da plataforma AWS Fargate durante o teste T3. As legendas *gcp_cr*, *aws* e *gcp_kn* representam as plataformas Cloud Run, AWS Fargate e Knative, respectivamente. Fonte (Autor)

O gráfico da *Figura 4.2.3.6* exibe outra visualização do teste de pico, a qual relaciona a latência e a quantidade de utilizadores virtuais, onde o eixo *y representa* a latência das requisições e o eixo *x* a quantidade de *vus*.

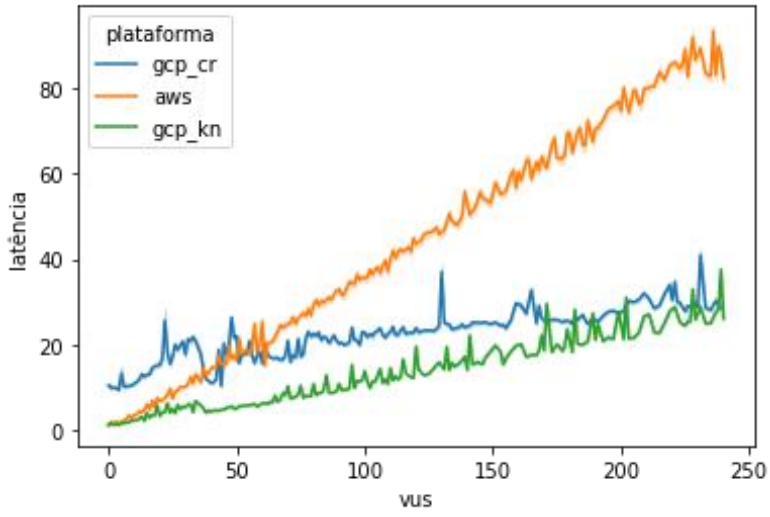


Figura 4.2.3.6: Gráfico de latência das requisições para as plataformas em relação a quantidade de utilizadores virtuais durante o teste T3. As legendas *gcp_cr*, *aws* e *gcp_kn* representam as plataformas Cloud Run, AWS Fargate e Knative, respectivamente. Fonte (Autor)

De acordo com os resultados anteriores é possível inferir as seguintes conclusões:

- Knative, apresentou uma latência média de 14,03ms, o que é apenas 1,90ms a mais do que em T2, indicando boa eficiência em escala. Enquanto Cloud Run e AWS Fargate apresentaram latências de 24,24ms e 40,63ms, com aumentos significativos de 12,68ms e 28,16ms, respectivamente;
- Cloud Run, continuou a apresentar grande instabilidade no tempo de resposta das suas requisições;
- Comparando o gráfico de *vus* em cada segundo da *Figura 4.2.3.1* e os resultados das plataformas, percebe-se uma grande semelhança entre a *Figura 4.2.3.1* e *Figura 4.2.3.4,* o que demonstra novamente uma proporção constante durante todo o teste entre número de *vus* o impacto na latência na plataforma AWS Fargate; tal resultado indica que a plataforma não respondeu bem ao teste de pico, apesar de ter baixo desvio padrão nas suas latências. Quanto às demais plataformas, ao observar os gráficos de latência, por momento, identifica-se uma acentuação das latências no momento do pico de paralelismo, o que demonstra uma melhor capacidade de elasticidade dessas plataformas, em especial da plataforma Cloud Run;
- O gráfico da *Figura 4.2.3.6*, assim como em T2, mostra que as plataformas apresentam uma função linear para a experiência de latência em relação aos utilizadores virtuais. AWS Fargate continua com o maior momento angular, seguido de Knative e *Cloud Run*. Por fim, a plataforma *Cloud Run* inicia com requisições mais longas, mas com o acréscimo de *vus,* iguala-se em performance a AWS Fargate em cerca de 40 *vus* e a Knative por volta dos 250 *vus.* Desta forma, apresenta o melhor tempo de resposta entre as plataformas em relação a quantidade de utilizadores em paralelo;
- Quanto à taxa de sucesso, as plataformas Cloud Run e Knative apresentaram alguns pouco erros e tiveram 99,99% de acerto. A plataforma AWS Fargate, apresentou-se estável e teve 100% de taxa de sucesso.

4.3.4 T4 - Teste de tamanho de carga

O teste de tamanho de carga avalia a escalabilidade das plataformas em relação ao aumento de utilizadores virtuais e o tamanho do *payload* das requisições, com o objetivo de observar o impacto das operações de I/O no desempenho das

plataformas. Este teste tem como base o teste T2 e apresenta as mesmas características (duração, configuração de memória/CPU e quantidade de *vus)* com exceção do *endpoint,* o qual é substituído por um novo *endpoint* HTTP POST*,* também codificado no *framework* JavaScript Node.js, que recebe um *body (payload)* e responde com o HTTP *status 200.*

A tabelas seguintes, apresentam os resultados estatísticos das requisições sobre as três plataformas em causa para diferentes tamanhos de *payload* em kilobytes: 64, 256, 512.

Cloud Run	64	256	512
Latência mínima (ms)	5,91	5,41	5,68
Latência máxima (ms)	784,67	790,45	613,14
Latência média (ms)	16,03	18,85	17,5
Latência 95º *percentil* (ms)	43,29	53,11	51,62
Latência 90º *percentil* (ms)	29,78	44,56	41,53
Nº de requisições	109649	93286	100559
% de sucesso	99,99	99,99	100
Requisições por segundo	1827,31	1554,71	1675,91

Tabela 4.2.4.1: Resultados do teste de tamanho de carga na plataforma Cloud Run. Fonte (Autor)

Knative	64	256	512
Latência mínima (ms)	0,63	0,62	0,63
Latência máxima (ms)	79,59	119,87	72,78
Latência média (ms)	5,37	5,68	5,26
Latência 95º *percentil* (ms)	13,15	13,63	13,23
Latência 90º *percentil* (ms)	10,02	10,58	10,18
Nº de requisições	323837	306719	330356
% de sucesso	100	100	100
Requisições por segundo	5397,06	5111,73	5505,66

Tabela 4.2.4.2: Resultados do teste de tamanho de carga na plataforma Knative. Fonte (Autor)

AWS Fargate	64	256	512
Latência mínima (ms)	1,31	1,3	1,32
Latência máxima (ms)	150,03	110,81	97,94
Latência média (ms)	9,57	9,51	9,64
Latência 95º *percentil* (ms)	54,6	54,72	54,98
Latência 90º *percentil* (ms)	20,44	19,02	19,06
Nº de requisições	183036	184217	181686
% de sucesso	100	100	100
Requisições por segundo	3050,32	3069,48	3027,05

Tabela 4.2.4.3: Resultados do teste de tamanho de carga na plataforma AWS Fargate. Fonte(Autor)

Apresentando o resultado das três plataformas em uma imagem, o gráfico da *Figura 4.2.4* relaciona latência à quantidade de utilizadores virtuais, levando em consideração o tamanho do *payload* em *kilobytes*, onde o eixo *y representa* a latência das requisições, o eixo *x* a quantidade de *vus* e o estilo da linha diferentes tamanhos de *payload*.

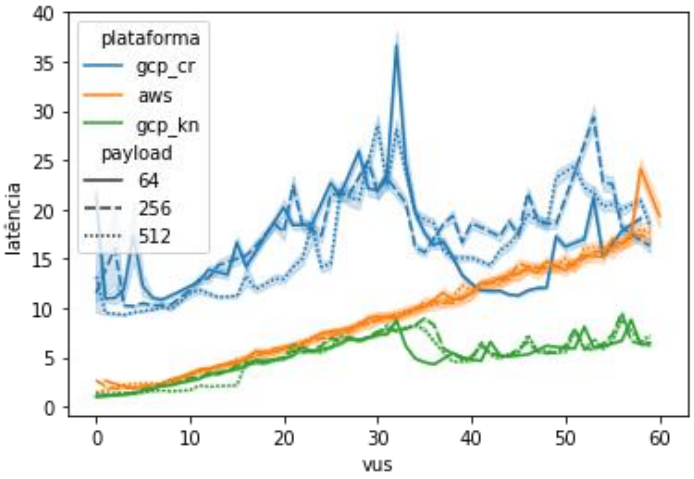


Figura 4.2.4: Gráfico de latência das requisições para as plataformas, em relação à quantidade de utilizadores virtuais e tamanho do *payload* das requisições durante o teste T4. As legendas *gcp_cr*, *aws* e *gcp_kn* representam as plataformas *Cloud Run*, AWS *Fargate* e *Knative*, respectivamente. Fonte (Autor)

De acordo com os resultados anteriores, é possível inferir as seguintes conclusões:

- Quanto aos valores médios em T4 em comparação aos mesmos valores em T2, pode-se observar um aumento na latência apenas para a plataforma Cloud Run e uma melhoria de latência para as demais plataformas; tal resultado é interessante, pois esperava-se um aumento das latências em razão da demora das transmissões de dados sobre o protocolo TCP;
- A plataforma Cloud Run apresentou elevada amplitude, entre as latências mínima e máxima, e falhas em algumas nas requisições (0,01% falhas com *payloads* de 64kB e 256kB). Em resumo, mostra-se instável como nos outros testes;
- Para as requisições com *payload* de 512kB, percebe-se uma melhoria nas respostas das plataformas, além de se observar algumas latências médias inferiores para valores menores de *payload*; as latências máximas em todas as plataformas diminuíram significativamente;
- Por fim, com exceção da plataforma AWS Fargate que não apresenta qualquer mudança em relação a diferentes tamanhos de *payload*, percebe-se através do gráfico da *Figura 4.2.4* uma melhoria nos tempos de resposta a partir de 30 *vus* nas plataformas *Knative* e *Cloud Run*. A provável causa da melhoria de *performance* é que as operações de I/O requerem maior uso de recursos computacionais e a utilização desses recursos de memória e CPU causam gatilhos de rotinas de *scale-out* nas plataformas *serverless*. Este comportamento também fomenta as conclusões anteriormente levantadas.

4.3.5 T5 - Teste de linguagem de programação

Da mesma forma que T4, o presente teste tem o T2 como base, mas com a diferença que a terceira variável independente é a linguagem de programação utilizada. Dessa forma, será possível perceber como diferentes *containers* ou *frameworks* podem afetar o tempo de resposta e escalabilidade das aplicações nas plataformas *serverless*. Posto isso e além do *endpoint* utilizado em outros testes, para execução deste teste foram implementados mais 5 *endpoints de* baixo custo computacional (HTTP GET que retorna “Hello World”) para as demais linguagens ou frameworks de programação. Os *frameworks* utilizados foram: Node.js (JavaScript), Flask (Python), Golang(Go), Spring (Java) e .Net Core (C#). No caso da linguagem PHP, foi utilizado servidor HTTP Apache para a implementação.

As tabelas seguintes, apresentam os resultados estatísticos das requisições junto das três plataformas em estudo em relação às diferentes linguagens de programação *web*.

Cloud Run	Node	Python	PHP	Go	Java	C#
Latência mínima (ms)	6,15	7,23	5,83	5,39	5,82	6,34
Latência máxima (ms)	1,02	1700	924,11	974,92	17000	1460
Latência média (ms)	16,61	61,36	21,21	17,66	34,7	19,91
Latência 95º *percentil* (ms)	49,84	156,79	60,65	39,08	76,41	56,48
Latência 90º *percentil* (ms)	36,13	125,31	52,66	32,53	69,37	41,36
Nº de requisições	105949	28744	83036	99679	50833	88347
% de sucesso	100	100	99,99	99,97	99,96	99,99
RPS	1765,75	479,05	1383,89	1661,26	847,18	1472,19

Tabela 4.2.5.1: Resultados do teste de linguagem de programação na plataforma *Cloud Run*. Fonte (Autor)

Knative	Node	Python	PHP	Go	Java	C#
Latência mínima (ms)	0,66	2,11	0,73	0,58	0,66	0,64
Latência máxima (ms)	77,42	49,65	94,78	53,75	87,81	99,07
Latência média (ms)	6,28	67,2	9,35	5,12	6,52	9,35
Latência 95º *percentil* (ms)	15,34	196,22	40,24	11,73	30,12	23,34
Latência 90º *percentil* (ms)	11,89	97,64	20,24	9,56	11,41	18,54
Nº de requisições	277938	26015	187559	340123	268082	187573
% de sucesso	100	100	100	100	100	100
RPS	4632,14	433,56	3125,46	5668,49	4467,85	3125,75

Tabela 4.2.5.2: Resultados do teste de linguagem de programação na plataforma Knative. Fonte (Autor)

AWS Fargate	Node	Python	PHP	Go	Java	C#
Latência mínima (ms)	1,37	1,93	1,97	1,06	1,19	1,23
Latência máxima (ms)	96,7	183,6	548,7	110, 95	414,68	137,71
Latência média (ms)	12,23	71,77	12,44	5,4	21,02	8,44
Latência 95º *percentil* (ms)	60,31	164,83	73,85	17,96	83,89	54,96
Latência 90º *percentil* (ms)	50,99	115,44	54,56	10,54	79,62	16,35
Nº de requisições	143654	24560	140828	322274	83688	207135
% de sucesso	100	100	100	100	100	100
RPS	2393,96	409,26	2346,79	5370	1394,46	3451,49

Tabela 4.2.5.3: Resultados do teste de linguagem de programação na plataforma AWS Fargate. Fonte (Autor)

De forma gráfica, as figuras que se seguem apresentam os resultados do teste de linguagem de programação (T5) para as três plataformas em análise. O gráfico representa a latência em milissegundos das requisições (eixo *y*) em relação a quantidade vus (eixo *x*). As diferentes linhas, diferenciadas por cores, representam as linguagens de programação.

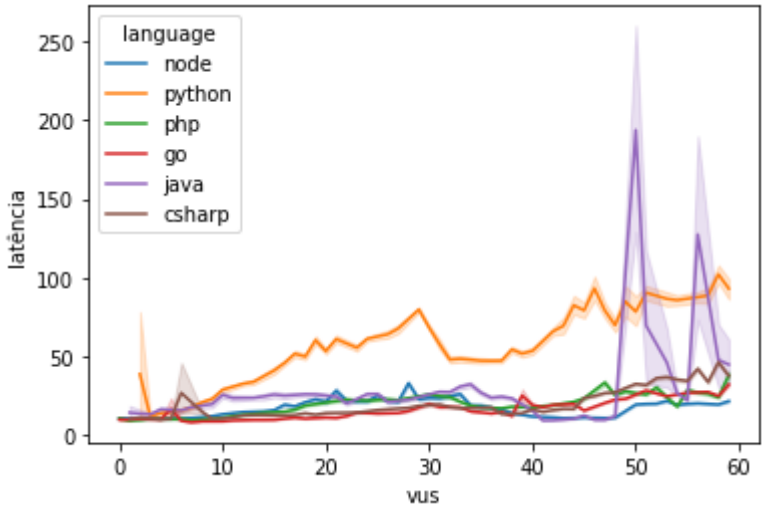


Figura 4.2.5.1: Gráfico de latência das linguagens de programação em relação à quantidade de vus na plataforma Cloud Run. Fonte (Autor)

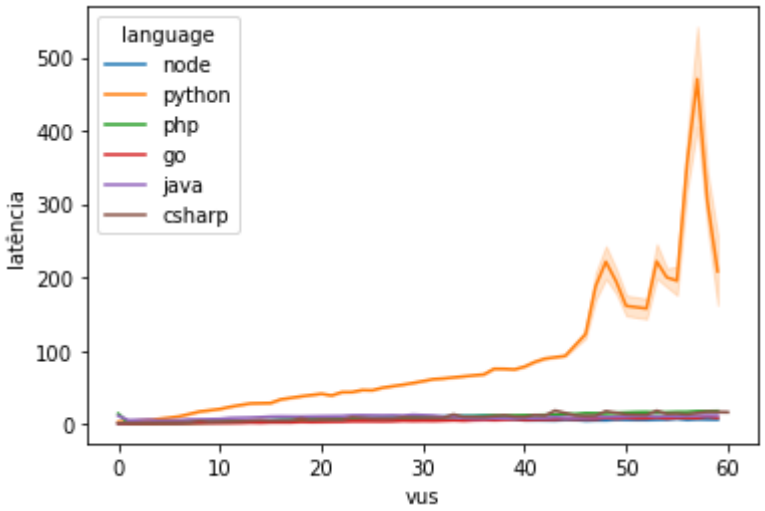


Figura 4.2.5.2: Gráfico de latência das linguagens de programação, em relação à quantidade de *vus* na plataforma Knative. Fonte(Autor)

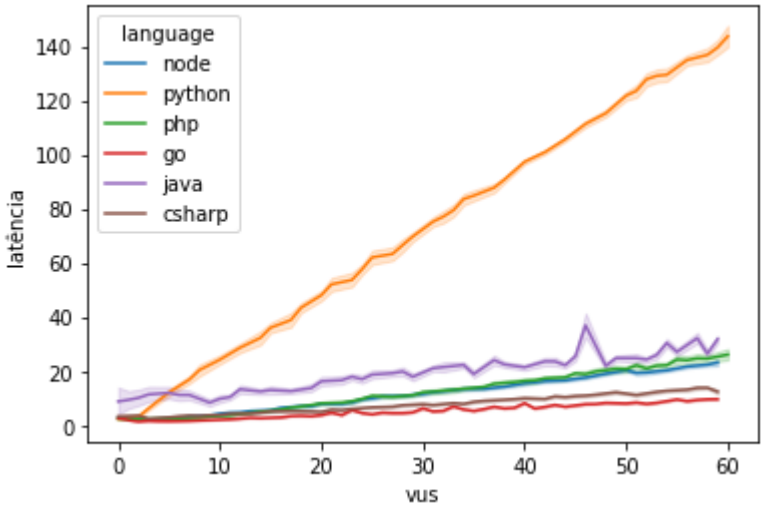


Figura 4.2.5.3: Gráfico de latência das linguagens de programação em relação à quantidade de *vus* na plataforma AWS Fargate. Fonte(Autor)

De forma global, o gráfico da *Figura 4.2.5.4* que se segue, exibe a relação da latência e a quantidade de utilizadores virtuais para as linguagens de programação, onde o eixo *y representa* a latência das requisições e o eixo *x* a quantidade de *vus*. As

linguagens de programação estão representadas por linhas de diferentes cores e de acordo com a legenda.

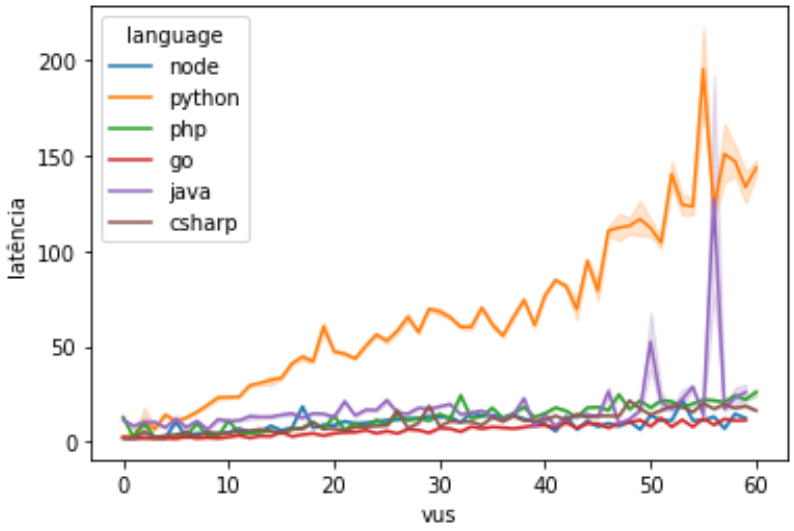


Figura 4.2.5.4: Gráfico de latência das linguagens de programação em relação a quantidade de *vus* nas plataformas *serverless*. Fonte (Autor)

De acordo com os resultados anteriores, é possível inferir as seguintes conclusões:

- Entre as três plataformas, a Knative destaca-se com latências médias das linhagens abaixo dos 10ms (exceto a linguagem Python, com 67,2ms de latência). As demais apresentam maiores valores e desvios de latência média para diferentes linguagens;
- A plataforma Cloud Run apresenta a pior *performance* de entre todas as linguagens. Ainda, apresentou requisições com erros para linhagens PHP, Go, Java e C#;
- Os *containers* Go nas plataformas Knative e AWS Fargate apresentam as melhores *performances* entre todo o teste.
- A linhagem Python tem grande dificuldade em lidar com requisições em paralelo em qualquer plataforma;
- De uma forma geral, os *containers* ou linhagem de programação afetam o tempo de resposta das plataformas, em relação à quantidade de utilizadores virtuais. As linguagens com melhores *performances* em escala são Go e Node. E as linguagens com piores rendimentos em escala são Java e Python.

4.3.6 T6 - Teste de tamanho de memória/ CPU

O teste de tamanho de memória e CPU tem o objetivo de determinar a influência de diferentes configurações de memória e CPU no desempenho das plataformas *serverless* que implementam CaaS. Idêntico ao teste T2, o teste executa requisições de baixo valor computacional durante 1 minuto e com uma quantidade crescente de utilizadores virtuais. A distinção deste teste em comparação ao T2 é a configuração de recursos de memória e CPU dos *containers.* Enquanto todos os testes anteriores possuem 512 MB de memória, o presente teste varia essa configuração para os valores de 1024 MB de memória e 2028 MB de memória.

Quanto à CPU, é desejado manter-se valores proporcionais para as plataformas. Uma vez que a plataforma *Cloud Run* apenas aceita valores inteiros, os respectivos valores configurados para as memórias de 512MB, 1024MB e 2048MB são 1 CPU, 2 CPUs e 3 CPUs. A plataforma Knative, por possuir flexibilidade de valores e parentesco com o serviço *Cloud Run*, apresentam os mesmos valores de memória e CPU. Por fim, a plataforma AWS Fargate possui pouca flexibilidade nos valores de memória e CPU, por esse motivo ao configurar os valores de memória para 512MB, 1024 MB e 2048, os valores proporcionais de CPU são 0,25 CPU, 0,5 CPU e 1 CPU, respetivamente.

As tabelas seguintes, apresentam os resultados estatísticos das requisições junto às três plataformas em estudo para diferentes conjuntos de configuração de memória e CPU.

Cloud Run	512	1024	2048
Latência mínima (ms)	5,61	6,21	6,13
Latência máxima (ms)	1200	865,33	1130
Latência média (ms)	15,11	23,03	17,64
Latência 95º *percentil* (ms)	45,62	46,77	35,32
Latência 90º *percentil* (ms)	27,14	42,64	27,46
Nº de requisições	116415	76469	99716
% de sucesso	100	99,96	99,98
Requisições por segundo	1940,14	1274,43	1661,67

Tabela 4.2.6.1: Resultados do teste de tamanho de memória e CPU na plataforma *Cloud Run.* Fonte (Autor)

Knative	512	1024	2048
Latência mínima (ms)	0,70	0,66	0,64
Latência máxima (ms)	82,67	103,5	77,05
Latência média (ms)	6,76	7,11	5,2
Latência 95º *percentil* (ms)	16,73	16,6	12,53
Latência 90º *percentil* (ms)	12,84	13,02	10,25
Nº de requisições	258633	246053	334655
% de sucesso	100	100	100
Requisições por segundo	4310,40	4100,73	5577,36

Tabela 4.2.6.2: Resultados do teste de tamanho de memória e CPU na plataforma Knative. Fonte (Autor)

AWS Fargate	512	1024	2048
Latência mínima (ms)	1,25	1,11	1,23
Latência máxima (ms)	105,95	73,67	61,49
Latência média (ms)	9,4	5,37	5,32
Latência 95º *percentil* (ms)	56,96	14,95	14,06
Latência 90º *percentil* (ms)	17,56	10,91	10,28
Nº de requisições	1186205	323995	327318
% de sucesso	100	100	100
Requisições por segundo	3102,45	5399,36	5454,87

Tabela 4.2.6.3: Resultados do teste de tamanho de memória e CPU na plataforma AWS Fargate. Fonte (Autor)

Nas figuras a seguir, apresentam-se os resultados do teste de tamanho de memória e CPU para as três plataformas em estudo. O gráfico representa a latência em milissegundos das requisições (eixo *y*) em relação a quantidade vus (eixo *x*). As diferentes linhas, diferenciadas por cores, são relativas aos diferentes conjuntos de configurações de memória e CPU.

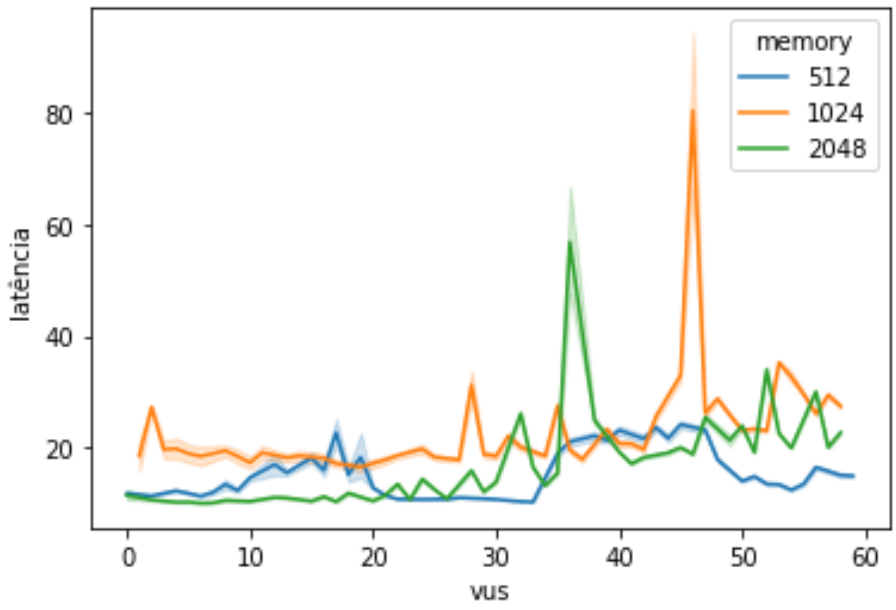


Figura 4.2.6.1: Gráfico de latência dos conjuntos de configurações de memória e CPU em relação a quantidade de *vus* na plataforma Cloud Run. Fonte (Autor)

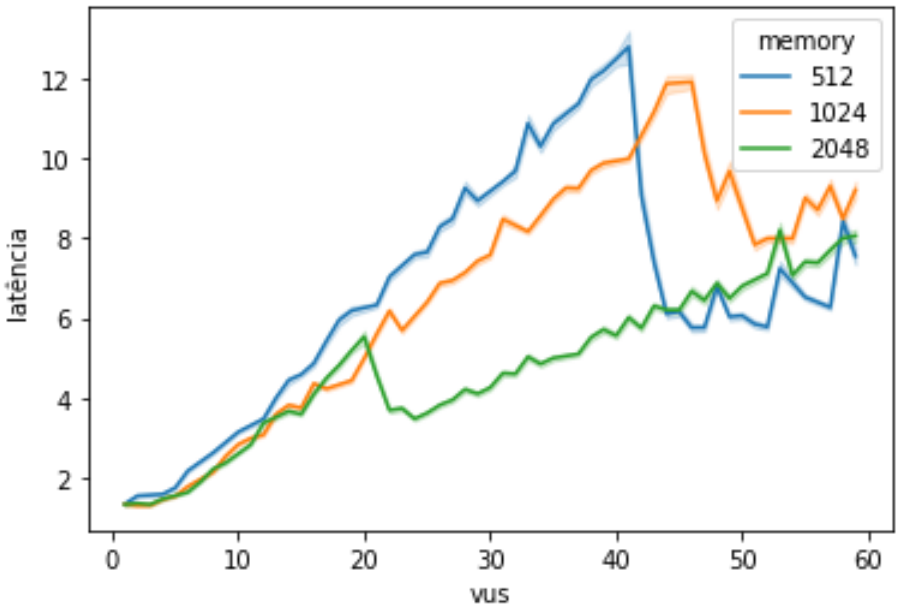


Figura 4.2.6.2: Gráfico de latência dos conjuntos de configurações de memória e CPU em relação à quantidade de *vus* na plataforma Knative. Fonte (Autor)

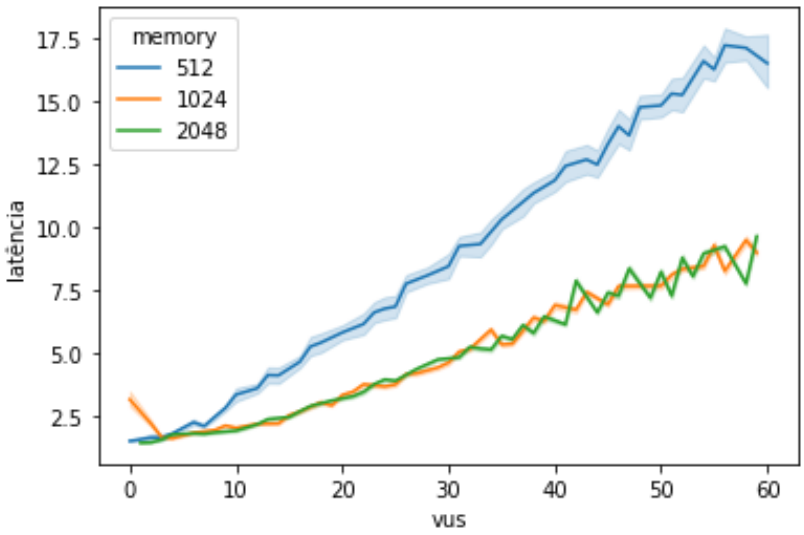


Figura 4.2.6.3: Gráfico de latência dos conjuntos de configurações de memória e CPU em relação à quantidade de *vus* na plataforma AWS Fargate. Fonte (Autor)

Por fim, a *Figura 4.2.6.4* exibe a relação entre diferentes configurações de memória e CPU e a latência das requisições quando a quantidade de utilizadores virtuais cresce.

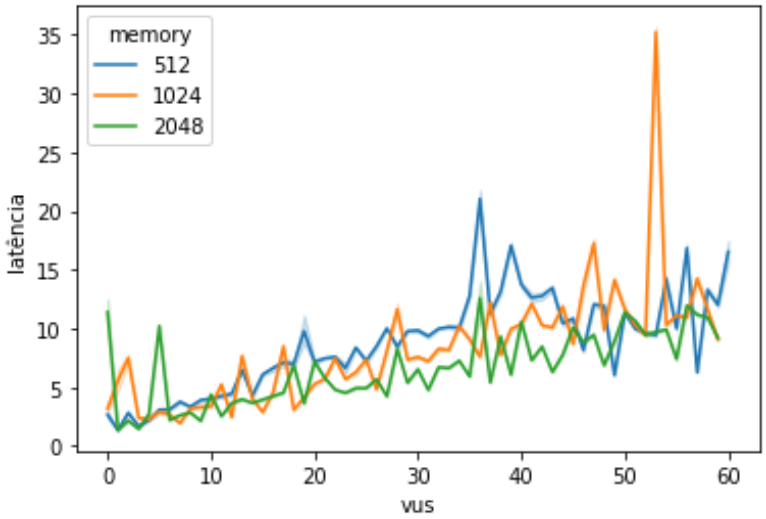


Figura 4.2.6.4: Gráfico de latência dos conjuntos de configurações de memória e CPU em relação a quantidade de *vus* nas plataformas. Fonte (Autor)

De acordo com os resultados anteriores, é possível inferir as seguintes conclusões:

- As plataformas Knative e AWS Fargate apresentaram melhorias significativas de *performance* quanto ao aumento dos recursos dos *containers.* Quanto mais memória e CPU atribuidos, melhor o tempo médio de resposta das requisições;
- Além da melhor *performance* das requisições, percebe-se na *Figura 4.2.6.2* que os recursos de memória e CPU afetam diretamente a forma que a plataforma Knative realiza o escalonamento dos *containers*;
- As *performances* para 1024 e 2048 megabytes na plataforma AWS Fargate mostraram-se semelhantes, indicando que há um ponto de saturação no aumento de recursos de forma vertical (*scale-up*). Fato esse que pode ser observado na *Figura 4.2.6.3*;
- Além de apresentar sua característica instabilidade, a plataforma Cloud Run obteve resultados contrários as demais plataformas e mesmo com mais recursos de memória e CPU apresentou maiores valores de latência média e não executou 100% das requisições com sucesso;
- De uma forma geral, a quantidade de recursos pode melhorar a *performance* das plataformas *serverless;* entretanto, como se percebe pela *Figura 4.2.6.4*, este ganho de *performance* pode não ser tão relevante, pois depende de outros fatores, como a forma ou eficiência que as aplicações utilizam os recursos dos *containers* ou os parâmetros que foram definidos para as regras de escalonamento.

4.3.7 T7 - Teste intensivo de computação

O presente teste mede o desempenho das plataformas, ao executar requisições com intensivo uso de computação e memória. Para isso, o teste utiliza-se de um novo *endpoint* HTTP GET, que implementa uma função que calcula um número de Fibonacci de forma recursiva. A exemplo dos demais, o teste tem como base T2, ou seja, segue todas as configurações padrões com a única diferença, o *endpoint* anteriormente citado.

Posto isso, no presente foram calculados os números de Fibonacci para 10, 20, 30 e 40. As tabelas a seguir apresentam os resultados estatísticos das requisições para as três plataformas.

Cloud Run	10	20	30	40
Latência mínima (ms)	6,21	6,5	37,43	1940
Latência máxima (ms)	932,69	579,74	1380	45030
Latência média (ms)	15,94	18,68	217,11	12210
Latência 95º *percentil* (ms)	50,32	55,55	797,13	35,53
Latência 90º *percentil* (ms)	33,59	47,24	705,34	25,88
Nº de requisições	110301	94158	8027	69
% de sucesso	100	100	100	100
Requisições por segundo	1838,27	1569,24	133,77	1,14

Tabela 4.2.7.1: Resultados do teste T7 na plataforma Cloud Run. Fonte (Autor)

Knative	10	20	30	40
Latência mínima (ms)	0,60	0,89	26,83	3820
Latência máxima (ms)	212,97	125,65	2380	24280
Latência média (ms)	6,18	6,18	135	9490
Latência 95º *percentil* (ms)	14,44	18,65	607,82	18370
Latência 90º *percentil* (ms)	11,45	14,18	340,43	16170
Nº de requisições	282407	282277	13009	126
% de sucesso	100	100	100	100
Requisições por segundo	4706,59	4704,14	216,80	2,09

Tabela 4.2.7.2: Resultados do teste T7 na plataforma Knative. Fonte (Autor)

AWS Fargate	10	20	30	40
Latência mínima (ms)	1,27	1,87	114,43	0,12
Latência máxima (ms)	106,2	231,91	15020	50850
Latência média (ms)	12,04	45,25	3470	177,61
Latência 95º *percentil* (ms)	59,44	93,79	7390	1,04
Latência 90º *percentil* (ms)	51,1	87,98	5210	1,01
Nº de requisições	145836	38924	465	1558
% de sucesso	100	100	96,77	0,64
Requisições por segundo	2430,41	648,48	7,74	25,96

Tabela 4.2.7.1: Resultados do teste T7 na plataforma AWS Fargate. Fonte (Autor)

De forma gráfica, as figuras que se seguem apresentam os resultados do teste intensivo de programação (T7) para as três plataformas em análise. O gráfico representa a latência em milissegundos das requisições (eixo *y*) em relação a quantidade vus (eixo *x*). As diferentes linhas, diferenciadas por cores, representam as latências para diferentes números de Fibonacci.

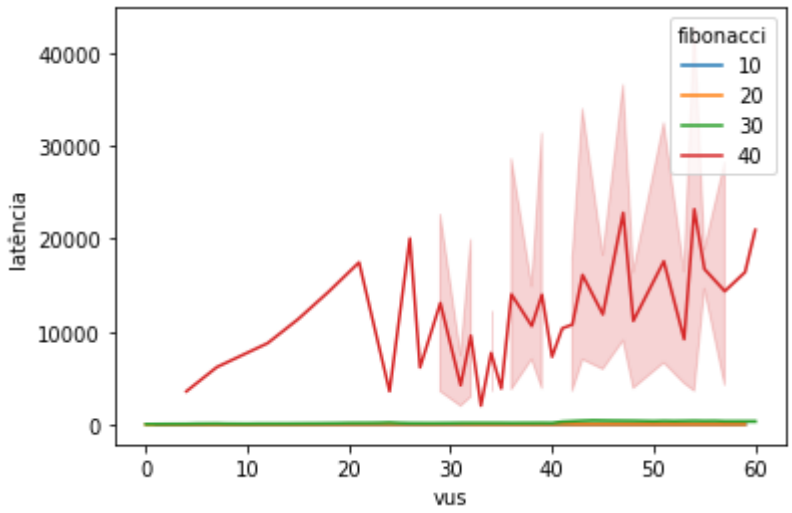


Figura 4.2.7.1: Gráfico da latência em relação à quantidade de *vus* na plataforma Cloud Run para diferentes números de Fibonacci. Fonte (Autor)

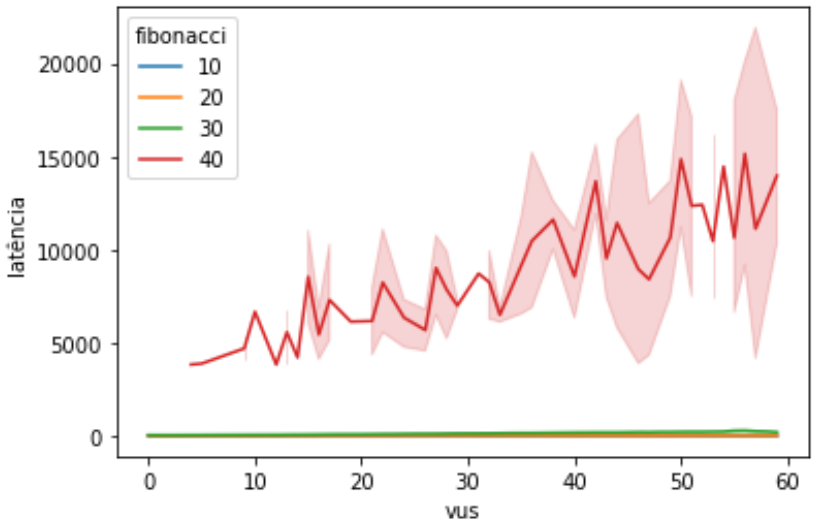


Figura 4.2.7.2: Gráfico da latência em relação à quantidade de *vus* na plataforma Knative para diferentes números de Fibonacci. Fonte (Autor)

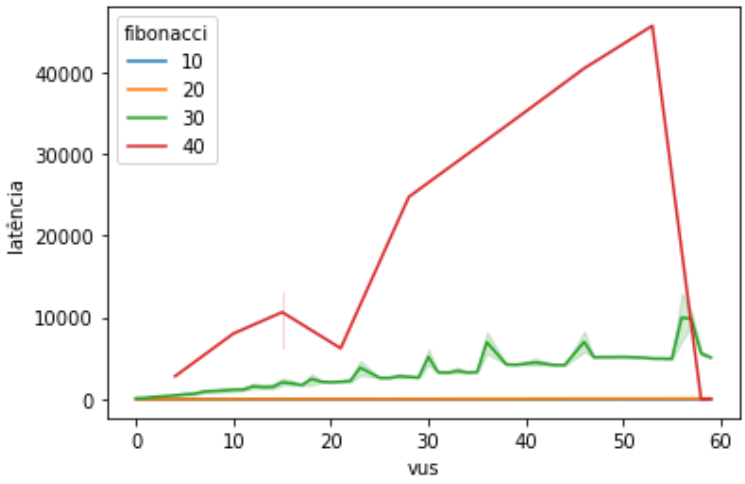


Figura 4.2.7.3: Gráfico da latência em relação à quantidade de *vus* na plataforma AWS Fargate para diferentes números de Fibonacci. Fonte (Autor)

Por fim, e para uma melhor visualização, estão condensados os resultados das três plataformas em um único gráfico. Segue na *Figura 4.2.7.4* a relação da latência, quantidade de utilizadores virtuais e o cálculo computacional intensivo da função de Fibonacci para as três plataformas, exibidas com estilos de linha diferentes. O eixo *y* representa a latência das requisições e o eixo *x* a quantidade de *vus*. As diferentes linhas, diferenciadas por cores, representam as latências para diferentes números de Fibonacci.

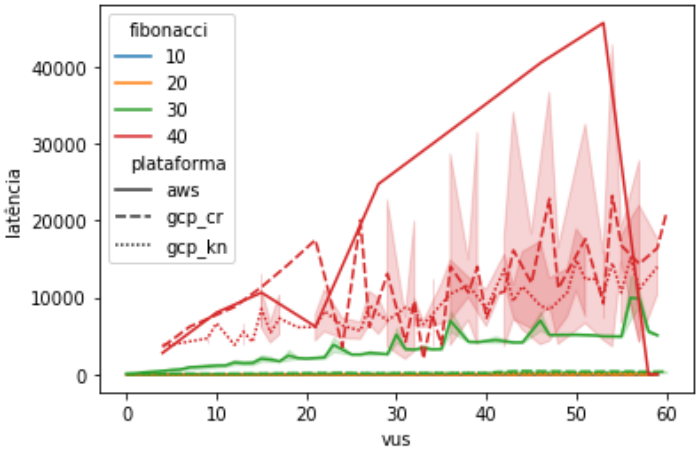


Figura 4.2.7.4: Gráfico da latência em relação a quantidade de *vus* para diferentes números de Fibonacci. Fonte (Autor)

De acordo com os resultados anteriores, é possível inferir as seguintes conclusões:

- A plataforma Knative apresentou a menor latência média e maior quantidade de execuções de cálculos de Fibonacci. Em segundo lugar em relação à *performance*, fica a plataforma *Cloud Run* e por último a plataforma AWS *Fargate*;
- A plataforma AWS *Fargate* não conseguiu calcular com sucesso os números de Fibonacci para valores de 40 e falhou em algumas requisições para valores de 30. Todas as outras plataformas tiveram 100% de sucesso nas chamadas;
- Através da comparação dos gráficos das figuras das plataformas *Cloud Run* e *Knative*, observa-se que, apesar do maior desvio apresentado, a plataforma *Cloud Run* apresenta uma melhor resposta para valores de *vus* mais altos, assim como sugere as conclusões do teste T3;
- Comparando os resultados das plataformas *Knative* e *Cloud Run* com testes anteriores, infere-se que os números de Fibonacci para 10 e 20 não interferem fortemente na *performance*. Quanto à plataforma AWS *Fargate*, apenas os cálculos do 10º número de Fibonacci não interferem na *performance*. Posto isto e outros fatores já apresentados, podemos constatar que a plataforma AWS *Fargate* é a menos eficiente para trabalhos intensivos de computação e isso deve-se ao fato da configuração de 512MB de memória corresponder a apenas 0,25CPU na plataforma.

4.3 Estimativa de custos

Assunto abordado na *Subsecção 2.3.3.1***,** um dos motivos para a adoção de tecnologias *serverless* é o custo associado. Sejam os custos de manutenção, diretamente ligados à mão de obra, ou os custos pagos aos provedores de nuvem pública pela infraestrutura. Dada a importância deste requisito, e em linha com o objetivo do presente trabalho, esta secção tem como objetivo estimar o custo das plataformas em análise na avaliação quantitativa proposta e também apresentar o valor gasto durante a experiência executada.

4.3.1 Estimativa de custos

Para as estimativas de custos, foram usados como parâmetros valores que remetem à experiência executada; desta forma é possível ter uma estimativa do preço valor gasto pelas plataformas durante a experiência com o objetivo de comparar o

resultado da estimativa com os valores reais gastos. Seguem as estimativas de custo das plataformas recolhidas das calculadoras de custo dos provedores de nuvem pública.

- *Cloud Run*

Diferente das demais, a plataforma *Cloud Run* cobra por recursos e tempo de execução. Então foram estimadas 13 589 037 requisições (soma de todas as requisições presentes nas tabelas de resultados anteriormente exibidos) em *containers* de 512MB de memória e um CPU. O tempo médio de execução foi retirado do teste base T2, 12ms, e o número de instâncias mínimas é um, pois foi desativado o *scale to zero* durante a experiência. Segue a estimativa de custo de acordo com a calculadora da GCP no dia 25 de setembro de 2021 [83].

Região: Finlândia Alocação de CPU: CPU é apenas alocada durante a requisição CPU: 1 Memória: 0.5 GiB Tempo de alocação de CPU: 156,000 vCPU-segundo Tempo de alocação de memória: 78,000 GiB-segundo Requisições: 13 598 037 Número mínimo de instâncias: 1 Preço do número mínimo de instâncias: 9.86 USD **Total: 14.49 USD por mês**

- Knative com GKE

Dado que a plataforma *open source* Knative depende de uma implementação em K8s, será estimado o custo do *cluster* situado na GCP configurado para a experiência. O *cluster* foi provisionado através do serviço GKE e possuía um *pool* de 3 máquinas e2-standard-4 (16GiB de memória e 4vCPUs). Segue a estimativa de custo de acordo com a calculadora da GCP no dia 25 de setembro de 2021 [83].

Região: Finlândia
2,190 total horas por mês
Tipo de máquina: REGULAR
Tipo de instância: e2-standard-4
Preço das instâncias: 323.15 USD
Preço do Sistema operativo: 0 USD
Custo de gestão do GKE *Cluste*r: 73.00 USD

Total: 396,15 USD por mês

- AWS Fargate

O serviço AWS Fargate, tem como modelo de *billing* a cobrança dos recursos de CPU e memória por minuto. Dado o uso em um *container* de 0.25 vCPU e 512 MB de memória, segue a estimativa de custo de acordo com a calculadora da AWS no dia 25 de setembro de 2021 [84].

Região: Estocolmo
1 *container* x 0.25 vCPU x 720h x 0.0445 USD = 8.01 USD for vCPU
1 *container* x 0.50 GB x 720h x 0.0049 USD = 1.76 USD for GB
8.01 USD for vCPU/h + 1.76 USD for GB/h

Total: 9.77 USD por mês

4.3.2 Custo da experiência

Para obter o custo real da experiência, obteve-se o relatório de cobrança das duas provedoras de nuvem utilizadas, GCP e AWS, durante os dias 31 de agosto e 1 setembro do ano de 2021, período em que foram executados os testes.

Os custos das plataformas *Cloud Run* e AWS *Fargate* foram 0,87 USD e 0,98 USD, respetivamente. Para a plataforma *open source Knative*, os custos podem ser divididos em dois serviços: 15,05 USD em máquinas virtuais (Google *compute Engine*) e 3,00 USD pelo custo de gestão do *cluster* K8s (Google *Kubernets Engine*). A *Figura 4.3.2* apresenta os valores de custo real e estimado para cada plataforma numa visualização em barras. O valor estimado para o gráfico foi calculado, levando

em consideração apenas dois dias do valor estimado mensal (trinta dias) obtido anteriormente.

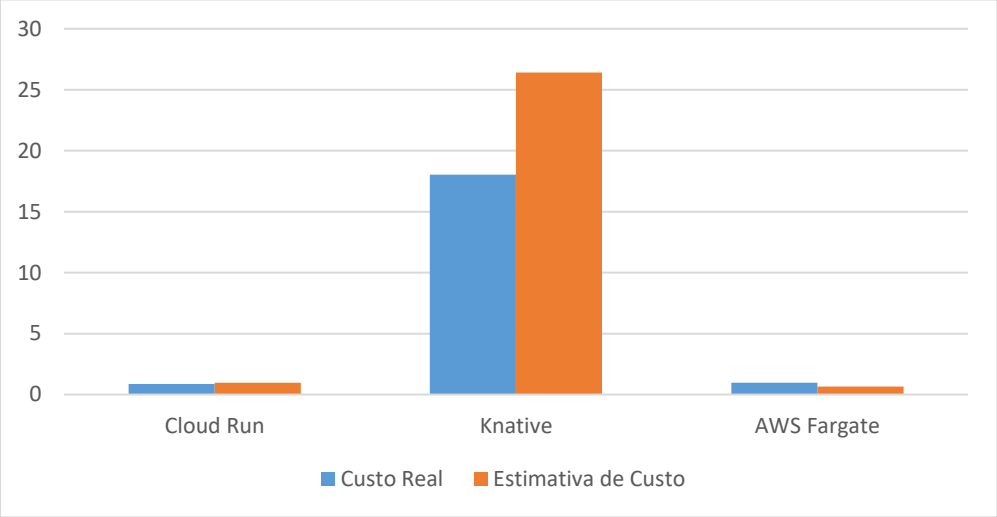


Figura 4.3.2: Comparação da estimativa e do custo real das experiências em cada plataforma. Fonte (Autor)

Apesar dos diversos fatores que foram deixados de lado na estimava, há uma proporção clara entre a estimativa e custo real, indicando assertividade nos parâmetros e cálculos.

4.4 Conclusões finais sobre a experiência de *benchmarking*

Acerca da realização dos testes e da estimativa de custos nas plataformas *serverless* Cloud Run, Knative e AWS Fargate obtêm-se as seguintes conclusões finais sobre o *benchmarking*:

- Em relação à latência das requisições sobre as plataformas, quando há pouca carga, Knative e AWS Fargate apresentam *performances* quase iguais e o serviço *Cloud Run* tem latências cinco vezes maiores. Entretanto, ao analisar-se as plataformas em esforço, a *Cloud Run* tende a ter uma melhor *performance* à medida que os utilizadores virtuais aumentam, ou seja, é a plataforma que apresenta melhor escalabilidade. Ainda sobre escalabilidade, a Knative ocupa o segundo lugar e por último a AWS Fargate;
- Apesar de apresentar a melhor resposta em escala, em todas as experiências a plataforma Cloud Run apresentou bastante instabilidade e diversos picos

onde a latência atingia altos valores, quando comparados com as outras plataformas;

- Essa instabilidade pode ser um motivo para a sua não utilização em sistemas críticos ou que requeiram tempos de resposta instáveis;
- Sobre a relação dos tamanhos de *payload* e o escalonamento das plataformas, com exceção da plataforma AWS Fargate, que não apresenta qualquer mudança em relação a diferentes tamanhos de *payload*, percebe-se uma melhora nos tempos de resposta nas plataformas Knative e *Cloud Run*. A provável causa da melhoria de *performance* é que as operações de I/O requerem maior uso de recursos computacionais e a utilização desses recursos geram gatilhos de *scale-out* nas plataformas;
- Quanto às linguagens de programação e o escalonamento das platafromas, diferentes *containers* podem afetar fortemente os tempos de resposta das plataformas. As linguagens Go e JavaScript mostraram ter as melhores *performances*, ou seja, pouco influenciam nas latências quando há aumento da carga em paralelo. As linguagens com piores rendimentos são Java e Python, as quais demostraram um grande aumento dos tempos de resposta, em especial Python;
- Quanto à quantidade de recursos (memória e CPU) e o escalonamento das plataformas, mais recursos pode melhorar a *performance* das plataformas *serverless;* entretanto esse ganho de *performance* pode não ser tão relevante pois depende de outros fatores, como a forma ou eficiência em que as aplicações utilizam os recursos dos *containers* ou os parâmetros que foram definidos para regras de escalonamento;
- Em relação a atividades de computação intensiva e escalonamento nas plataformas CaaS, a Knative apresentou a melhor eficiência, com maior número de cálculos e menor latência média. O destaque negativo foi a plataforma AWS Fargate, pois não conseguiu calcular grandes números de Fibonacci e apresentou alta latência;
- Quanto aos custos, a estimava e o custo real da experiência para as plataformas *Cloud Run* e AWS *Fargate* mostram-se similares; entretanto, se levarmos em consideração que a plataforma *Cloud Run* tem mais tempo de CPU alocado, possui *scale to zero* e cobra apenas pelo tempo que as

requisições são executadas, *Cloud Run* mostra-se a plataforma com o menor custo. Por fim, por possuir dependência de um *pool* de máquinas virtuais para compor um *cluster* K8s, a plataforma Knative é a mais cara das opções, sendo o seu uso aconselhado quando se precisa de algumas das mais valias das plataformas *open source,* como implementação em nuvem privada ou para evitar o *vendor lock-in*;

- Quanto à facilidade de implementação, a *Cloud Run* é a plataforma com menor curva de aprendizagem e pode ser ideal em projetos com poucos recursos, sendo possível criar serviços escaláveis, com apenas um comando de sua interface CLI. A plataforma Knative, caso seja desconsiderada a configuração do cluster K8s, exige apenas a instalação dos CRDs, uma vez que essa instalação não requer profundos conhecimentos de K8s podendo-se considerar esta plataforma em segundo lugar em facilidade de implementação. Ainda sobre Knative, é importante citar que é a plataforma que, teoricamente, vai exigir maior custo de manutenção. Por fim, AWS Fargate é a plataforma que possui o maior custo de implementação por exigir um número maior de componentes a serem configurados, bem como o conhecimento específico dos recursos de *networking* da AWS.

Capítulo V – Conclusão

Neste capítulo, são referidas as conclusões do estudo procurando uma integração e confrontação dos resultados obtidos na parte prática, com a fundamentação teórica apresentada. E é feita a apresentação das limitações sentidas ao longo do desenvolvimento da investigação e as suas aplicações em trabalhos futuros.

Esta dissertação de mestrado assumiu como objetivo, realizar uma discussão acerca da adoção de *serverless computing*, utilizando o modelo de implementação *Container-as-a-Service*, enumerando e avaliando os principais serviços e *frameworks* utilizados atualmente na indústria, a fim de obter as suas vantagens e desvantagens.

Para tal, esta pesquisa baseou-se na crescente procura sobre *serverless computing*, sendo este um novo paradigma para a implantação de aplicações em nuvem escaláveis. Esta popularidade deve-se ao fato de que o fornecedor de infraestrutura assume diversas responsabilidades, enquanto os desenvolvedores se preocupam apenas com a lógica de negócio das solicitações dos clientes.

O termo *serverless* ficou conhecido em 2014, e é geralmente associado com o modelo de implementação *Function-as-a-Service*. Porém, o conceito de *serverless* não se limita apenas às funções e têm surgido mais *frameworks* e produtos que utilizam *containers*, por exemplo, o modelo de implementação *Container-as-a-Service*, sendo este o destaque do presente trabalho.

Infere-se, que *frameworks serverless* contribuem para que engenheiros de informática construam aplicações escaláveis na nuvem, visto que menos tempo é investido em operação, possui elasticidade e gera menos custos de infraestrutura, apresentando assim, diversas vantagens.

Percebe-se ainda que as principais vantagens em utilizar *Container-as-a-Service* para implementar o modelo de computação *serverless* são: a melhor capacidade de migração, o tempo de execução ilimitado, com maior controlo sobre o ambiente e melhor *performance* de hardware; possui também uma alternativa ao problema de *vendor lock-in*, além de dispor de um maior número e ferramentas suporte.

No contexto das tecnologias que implementam o modelo *Container-as-a-Service*, as principais plataformas disponíveis no mercado e suas respetivas características, o presente trabalho desenvolveu uma avaliação qualitativa e quantitativa.

Na avaliação qualitativa, foram levantadas as principais plataformas da indústria *open source*, as quais foram escolhidas de acordo com a quantidade de contribuidores e *stars* no site Github, e os serviços das mais relevantes nuvens públicas: Microsoft Azure, Google Cloud Platform (GCP) e Amazon Web Services (AWS). Após o levantamento, foram avaliadas as platafromas Knative, *Openwhisk*, *OpenFaaS*, ACI, *Cloud Run*, Lambda e *Fargate*. Entre as plataformas analisadas anteriormente, Knative, Cloud Run e AWS Fargate destacam-se por possuírem *autoscaling* e não dependerem de *custom images templates* para implementação. Na avaliação quantitativa, a fim de obter conclusões de maior valor e diminuir o domínio da experiência, apenas as plataformas em destaque foram avaliadas através de uma experiência de *benchmarking*.

O *benchmarking* proposto, foi desenvolvido a partir de outros trabalhos semelhantes na literatura e possui um total de sete testes, os quais se utilizam de estatísticas sobre a execução de requisições aos sistemas *serverless* e avaliam o comportamento das plataformas, medido pelo tempo de resposta e porcentagem de sucesso em relação ao aumento de tráfego. Ainda, em alguns testes, foi adicionado uma terceira variável independente, como quantidade de recursos ou linguagem de programação, para análise de correlação.

Para a execução da experiência foi necessário provisionar a infraestrutura das plataformas, configurar os componentes responsáveis pelo *benchmarking*, executar requisições HTTP, medir os tempos de resposta e armazená-los para posterior consulta e análise.

Com isso, a partir da experiência apurou-se que, em relação à latência das requisições sobre as plataformas, quando há pouca carga, Knative e AWS Fargate apresentam performances quase iguais e o serviço *Cloud Run* tem latências cinco vezes maiores. Entretanto, ao analisar-se as plataformas em esforço, *Cloud Run* tende a ter uma melhor *performance* à medida que os utilizadores virtuais aumentam, ou seja, é a plataforma que apresenta melhor escalabilidade. Ainda sobre escalabilidade, Knative ocupa o segundo lugar e por último AWS *Fargate*. Apesar de apresentar a melhor resposta em escala, em todas as experiências a plataforma *Cloud Run* apresentou bastante instabilidade e diversos picos onde a latência atingia altos valores, comparados com as outras plataformas. Obtiveram-se ainda conclusões acerca de outros fatores, que podem afetar o desempenho destes

sistemas, como o tamanho da requisição, a linguagem de programação, os recursos alocados e as rotinas de computação intensiva.

Conclui-se que as plataformas *Container-as-a-Service*, além de auxiliarem o desenvolvimento de aplicações escaláveis, apresentam ferramentas e soluções para os desafios do modelo de computação *serverless*.

5.1 Limitações e Trabalhos futuros

Este estudo apresentou algumas limitações, nomeadamente ao nível da pesquisa de informações, visto que por ser um tema atual, ainda existem poucos trabalhos especificamente sobre *container-as-a-service*. Além disso, considerou-se complicado realizar a experiência, já que foi necessário considerar muitos pormenores de cada plataforma para configurar a infraestrutura; *Google Cloud* e Kubernetes foram mais facilmente executados, visto que o Autor já possuía conhecimentos prévios, o mesmo não se aplicando a AWS. Apesar das limitações identificadas, foi possível realizar a pesquisa de forma satisfatória.

Dada a importância do tema, considera-se que ainda há muito que percorrer no campo da investigação, sugerindo-se, portanto, para trabalhos futuros que as avaliações qualitativas e quantitativas possam ter um número maior de plataformas, por exemplo, novos produtos que surgiram recentemente, como GKE Autopilot [81], ou de outras provedoras de computação em nuvem, como Heroku Dynos [82]. Ainda sobre mudanças na metodologia, seria interessante que na avaliação quantitativa fossem exploradas as funcionalidades das plataformas orientadas a eventos, como *Knative Serving* e *Cloud Run Event Triggers*, além de testes que correlacionem diferentes parâmetros, como combinações de memória/ CPU e diferentes linguagens de programação ou números de Fibonacci.

Por fim, numa linha de pesquisa diferente podem ser investigados outros tipos de produtos *serverless* da indústria, como *Database-as-a-Service*.

Referências

[1] S. K. Mohanty, G. Premsankar and M. di Francesco, "An Evaluation of Open Source Serverless Computing Frameworks," 2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), Nicosia, 2018, pp. 115-120, doi: 10.1109/CloudCom2018.2018.00033.

[2] K. Kritikos and P. Skrzypek, "A Review of Serverless Frameworks" 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion), 2018, pp. 161-168, doi: 10.1109/UCC-Companion.2018.00051.

[3] Fox, Geoffrey & Isahagian, Vatche & Muthusamy, Vinod & Slominski, Aleksander. (2017). Status of Serverless Computing and Function-as-a-Service (FaaS) in Industry and Research. 10.13140/RG.2.2.15007.87206.

[4] Baldini, Ioana & Castro, Paul & Chang, Kerry & Cheng, Perry & Fink, Stephen & Isahagian, Vatche & Mitchell, Nick & Muthusamy, Vinod & Rabbah, Rodric & Slominski, Aleksander & Suter, Philippe. (2017). Serverless Computing: Current Trends and Open Problems. 10.1007/978-981-10-5026-8_1.

[5] Jain, Prerna & Munjal, Yogesh & Gera, Jatin & Gupta, Pooja. (2020). Performance Analysis of Various Server Hosting Techniques. Procedia Computer Science. 173. 70-77. 10.1016/j.procs.2020.06.010.

[6] "CNCF Annual Report 2020 | Cloud Native Computing Foundation", *Cloud Native Computing Foundation*, 2021. [Online]. Available: https://www.cncf.io/cncf-annual-report-2020/. [Acedido em: 27- Jan- 2021].

[7] A. Palade, A. Kazmi and S. Clarke, "An Evaluation of Open Source Serverless Computing Frameworks Support at the Edge" 2019 IEEE World Congress on Services (SERVICES), Milan, Italy, 2019, pp. 206-211, doi: 10.1109/SERVICES.2019.00057.

[8] Khot, Aditi. (2020). A Comparative Analysis of Public Cloud Platforms and Introduction of Multi-Cloud. International Journal of Innovative Science and Research Technology. 5. 448-454. 10.38124/IJISRT20SEP234.

[9] Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., & Zaharia, M. (2010). A view of cloud computing. Communications of the ACM, 53(4),50-58. doi:10.1145/1721654.1721672

[10] Dragoni N. et al. (2017) Microservices: Yesterday, Today, and Tomorrow. In: Mazzara M., Meyer B. (eds) Present and Ulterior Software Engineering. Springer, Cham. https://doi.org/10.1007/978-3-319-67425-4_12

[11] Adzic, Gojko & Chatley, Robert. (2017). Serverless computing: economic and architectural impact. 884-889. 10.1145/3106237.3117767.

[12] S. McElwee, S. McElwee, R. Bashara, E. Blitstein and R. Bashara, "Black Friday 2019 Website Performance Report by Uptime.com", *Uptime.com*, 2021. [Online]. Disponível em: https://uptime.com/blog/black-friday-2019. [Acedido em: 27- Jan- 2021].

[13] "Cost of Downtime for Top US eCommerce Sites", *Gremlin.com*, 2021. Disponível em: https://www.gremlin.com/ecommerce-cost-of-downtime/. [Acedido em: 27- Jan- 2021].

[14] D. Poccia, "New for AWS Lambda – Container Image Support | Amazon Web Services", *Amazon Web Services*, 2020. [Online]. Disponível em: https://aws.amazon.com/blogs/aws/new-for-aws-lambda-container-image-support/. [Acedido em: 27- Jan- 2021].

[15] Burkat, Krzysztof & Pawlik, Maciej & Balis, Bartosz & Malawski, Maciej & Vahi, Karan & Rynge, Mats & Ferreira da Silva, Rafael & Deelman, Ewa. (2020). Serverless Containers -- rising viable approach to Scientific Workflows.

[16] Perez, Alfonso & Moltó, Germán & Caballer, Miguel & Calatrava, Amanda. (2018). Serverless computing for container-based architectures. Future Generation Computer Systems. 83. 10.1016/j.future.2018.01.022.

[17] "What is Stress Testing? How to create a Stress Test in k6", K6.io, 2021. Disponível em: https://k6.io/docs/test-types/stress-testing. [Acedido em: 27- Jan- 2021].

[18] "InfluxDB Time Series Platform | InfluxData", InfluxData, 2021. Disponível em: https://www.influxdata.com/products/influxdb/. [Acedido em: 27- Jan- 2021].

[19] L. SIQUEIRA & J. BRENDEL, Linux Pocket Pro: Virtualização. São Paulo: Linux New Media, 2007, p. 4.

[20] Menezes, Diogo & Mattos, Ferrazani. (2008). Virtualização: VMWare e Xen.

[21] K. Hwang, X. Bai, Y. Shi, M. Li, W. Chen and Y. Wu. Cloud Performance Modeling with Benchmark Evaluation of Elastic Scaling Strategies, IEEE Transactions on Parallel and Distributed Systems, vol. 27, no. 1, pp. 130-143, 1 Jan. 2016.

[22] R. Dua, A. R. Raja, and D. Kakadia, "Virtualization vs Containerization to Support PaaS," 2014 IEEE International Conference on Cloud Engineering, Boston, MA, USA, 2014, pp. 610-614, doi: 10.1109/IC2E.2014.41.

[23] “What is a hypervisor?” *Red Hat*. [Online]. Disponível em: https://www.redhat.com/en/topics/virtualization/what-is-a-hypervisor. [Acedido em: 22- Feb- 2021].

[24] A. Desai, R. Oza, P. Sharma and B. Patel, "Hypervisor: A survey on concepts and taxonomy", Int. J. Innov. Technol. Explore. Eng., vol. 2, no. 3, pp. 2278-3075, Feb. 2013.

[25] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange and C. A. F. De Rose, "Performance Evaluation of Container-Based Virtualization for High-Performance Computing Environments," 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, Belfast, UK, 2013, pp. 233-240, doi: 10.1109/PDP.2013.41.

[26] T. Pfandzelter and D. Bermbach, "tinyFaaS: A Lightweight FaaS Platform for Edge Environments," 2020 IEEE International Conference on Fog Computing (ICFC), Sydney, NSW, Australia, 2020, pp. 17-24, doi: 10.1109/ICFC49376.2020.00011.

[27] Y. Jadeja and K. Modi, "Cloud computing - concepts, architecture and challenges," 2012 International Conference on Computing, Electronics and Electrical Technologies (ICCEET), Nagercoil, India, 2012, pp. 877-880, doi: 10.1109/ICCEET.2012.6203873.

[28] F. Fatemi Moghaddam, M. B. Rohani, M. Ahmadi, T. Khodadadi and K. Madadipouya, "Cloud computing: Vision, architecture and Characteristics," *2015 IEEE 6th Control and System Graduate Research Colloquium (ICSGRC)*, Shah Alam, Malaysia, 2015, pp. 1-6, doi: 10.1109/ICSGRC.2015.7412454.

[29] K. Rafique, A. W. Tareen, M. Saeed, J. Wu and S. S. Qureshi, "Cloud computing economics opportunities and challenges," *2011 4th IEEE International Conference on Broadband Network and Multimedia Technology*, Shenzhen, China, 2011, pp. 401-406, doi: 10.1109/ICBNMT.2011.6155965.

[30] T. Dillon, C. Wu and E. Chang, "Cloud Computing: Issues and Challenges," *2010 24th IEEE International Conference on Advanced Information Networking and Applications*, Perth, WA, Australia, 2010, pp. 27-33, doi: 10.1109/AINA.2010.187.

[31] "DBaaS (Database-as-a-Service)", Ibm.com, 2021. [Online]. Disponível em: https://www.ibm.com/cloud/learn/dbaas. [Acedido em: 28- Fev- 2021].

[32] "What is FaaS (Function-as-a-Service)?", Ibm.com, 2021. [Online]. Disponível em: https://www.ibm.com/cloud/learn/faas. [Acedido em: 20- Mar- 2021].

[33] S. Eismann *et al*., "Serverless Applications: Why, When, and How?" in *IEEE Software*, vol. 38, no. 1, pp. 32-39, Jan.-Feb. 2021, doi: 10.1109/MS.2020.3023302.

[34] P. Castro, V. Ishakian, V. Muthusamy and A. Slominski, "The rise of serverless computing", Communications of the ACM, vol. 62, no. 12, pp. 44-54, 2019, doi: 10.1145/3368454.

[35] "Princípios por trás do Manifesto Ágil", Agilemanifesto.org, 2021. [Online]. Disponível em: https://agilemanifesto.org/iso/ptbr/principles.html. [Acedido em: 20- Mar- 2021].

[36] "Architecture styles - Azure Application Architecture Guide", docs.microsoft.com, 2021. [Online]. Disponível em: https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/. [Acedido em: 20- Mar- 2021].

[37] L. Leite, C. Rocha, F. Kon, D. Milojicic and P. Meirelles, "A Survey of DevOps Concepts and Challenges", ACM Computing Surveys, vol. 52, no. 6, pp. 1-35, 2020. doi: 10.1145/3359981

[38] S. BANGERA, DevOps for Serverless Applications. PACKT Publishing Limited, 2018.

[39] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly and S. Pallickara, "Serverless Computing: An Investigation of Factors Influencing Microservice Performance," 2018 IEEE International Conference on Cloud Engineering (IC2E), Orlando, FL, USA, 2018, pp. 159-169, doi: 10.1109/IC2E.2018.00039.

[40] E. van Eyk et al., "The SPEC-RG Reference Architecture for FaaS: From Microservices and Containers to Serverless Platforms," in IEEE Internet Computing, vol. 23, no. 6, pp. 7-18, 1 Nov.-Dec. 2019, doi: 10.1109/MIC.2019.2952061.

[41] E. Casalicchio, "Container Orchestration: A Survey", Systems Modeling: Methodologies and Tools, pp. 221-235, 2018, doi: 10.1007/978-3-319-92378-9_14.

[42] I. M. A. Jawarneh et al., "Container Orchestration Engines: A Thorough Functional and Performance Comparison," ICC 2019 - 2019 IEEE International Conference on Communications (ICC), Shanghai, China, 2019, pp. 1-6, doi: 10.1109/ICC.2019.8762053.

[43] "Docker overview", Docker Documentation, 2021. [Online]. Disponível em: https://docs.docker.com/get-started/overview/. [Acedido em: 20- Mar- 2021].

[44] "Swarm mode key concepts", Docker Documentation, 2021. [Online]. Disponível em: https://docs.docker.com/engine/swarm/key-concepts/. [Acedido em: 20- Mar- 2021].

[45] "What is Kubernetes?", Kubernetes, 2021. [Online]. Disponível em: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/. [Acedido em: 20- Mar- 2021].

[46] "Kubernetes Components", Kubernetes, 2021. [Online]. Disponível em: https://kubernetes.io/docs/concepts/overview/components/. [Acedido em: 20- Mar- 2021].

[47] "Extending your Kubernetes Cluster", Kubernetes, 2021. [Online]. Disponível em: https://kubernetes.io/docs/concepts/extend-kubernetes. [Acedido em: 25- Mar- 2021].

[48] "What is DevOps? | Atlassian", Atlassian, 2021. [Online]. Disponível em: https://www.atlassian.com/devops. [Acedido em: 28- Mar- 2021].

[49] "What is DevOps? - Azure DevOps", docs.microsoft.com, 2021. [Online]. Disponível em: https://docs.microsoft.com/azure/devops/learn/what-is-devops. [Acedido em: 28- Mar- 2021].

[50] A. Mishra and Z. Otaiwi, "DevOps and software quality: A systematic mapping", Computer Science Review, vol. 38, p. 100308, 2020. doi: 10.1016/j.cosrev.2020.100308.

[51] E. Levinson. “Serverless Community Survey 2020”, https://bit.ly/SerComSurvey, Feb 2020

[52] Martins, H., Araujo, F. & da Cunha, P.R. Benchmarking Serverless Computing Platforms. J Grid Computing 18, 691–709 (2020). https://doi.org/10.1007/s10723-\020-09523-1

[53] M. Malawski, A. Gajek, A. Zima, B. Balis, and K. Figiela, "Serverless execution of scientific workflows: Experiments with HyperFlow, AWS Lambda and Google Cloud Functions", *Future Generation Computer Systems*, vol. 110, pp. 502-514, 2020. Available: 10.1016/j.future.2017.10.029 [Acedido em 18 de Julho de 2021].

[54] "Terraform by HashiCorp", Terraform by HashiCorp, 2021. [Online]. Disponível em: https://www.terraform.io/. [Acedido em: 13 de Setembro de 2021].

[55] "Project Jupyter", *Jupyter.org*, 2021. [Online]. Disponível em: https://jupyter.org/. [Acedido em: 13 de Setembro de 2021].

[56] "Home - Knative", *Knative.dev*, 2021. [Online]. Disponível em: https://knative.dev/docs/. [Acedido em 18 de Julho de 2021].

[57] "Apache OpenWhisk is a serverless, open source cloud platform", *Openwhisk.apache.org*, 2021. [Online]. Disponível em: https://openwhisk.apache.org/. [Acedido em 18 de Julho de 2021].

[58] O. Ltd, "Home", *OpenFaaS - Serverless Functions Made Simple*, 2021. [Online]. Available: https://www.openfaas.com/. [Acedido em 18 de Julho de 2021].

[59] "Kubeless", *Kubeless.io*, 2021. [Online]. Disponível em: https://kubeless.io/. [Acedido em 18 de Julho de 2021].

[60] "Fission", *Fission*, 2021. [Online]. Disponível em: https://fission.io/. [Acedido em 18 de Julho de 2021].

[61] "Fn Project - The Container Native Serverless Framework", *Fnproject.io*, 2021. [Online]. Disponível em: https://fnproject.io/. [Acedido em 18 de Julho de 2021].

[62] "Nuclio", *nuclio*, 2021. [Online]. Disponível em: https://nuclio.io/. [Acedido em 18 de Julho de 2021].

[63] "Iron.io Open Source - Functions", *Open.iron.io*, 2021. [Online]. Disponível em: https://open.iron.io/. [Acedido em 18 de Julho de 2021].

[64] "Serverless containers in Azure - Azure Container Instances", *Docs.microsoft.com*, 2021. [Online]. Disponível em: https://docs.microsoft.com/en-us/azure/container-instances/container-instances-overview. [Acedido em 18 de Julho de 2021].

[65] "Cloud Run Documentation | Google Cloud", *Google Cloud*, 2021. [Online]. Disponível em: https://cloud.google.com/run/docs. [Acedido em 18 de Julho de 2021].

[66]"What is AWS Fargate? - Amazon ECS", *Docs.aws.amazon.com*, 2021. [Online]. Disponível em: https://docs.aws.amazon.com/AmazonECS/latest/userguide/what-is-fargate.html. [Acedido em 30 de Setembro de 2021].

[67]"What is Lambda? - AWS Lambda", *Docs.aws.amazon.com*, 2021. [Online]. Disponível em: https://docs.aws.amazon.com/lambda/latest/dg/welcome.html. [Acedido em 30 de Setembro de 2021].

[68] I. Education, "knative", *Ibm.com*, 2021. [Online]. Disponível em: https://www.ibm.com/cloud/learn/knative#toc-benefits-1bZ7yTDN. [Acedido em 30 de Setembro de 2021].

[69] "Deploying Docker containers on Azure", *Docker Documentation*, 2021. [Online]. Disponível em: https://docs.docker.com/cloud/aci-integration/. [Acedido em 30 de Setembro de 2021].

[70] "GitHub - ibm-functions/composer: Composer is a new programming model for composing cloud functions built on Apache OpenWhisk.", *GitHub*, 2021. [Online]. Disponível em: https://github.com/ibm-functions/composer. [Acedido em 30 de Setembro de 2021].

[71] "Kogito", *Kogito.kie.org*, 2021. [Online]. Disponível em: https://kogito.kie.org/. [Acedido em 30 de Setembro de 2021].

[72] "openwhisk-runtime-docker · apache/openwhisk-runtime-docker", *GitHub*, 2021. [Online]. Disponível em: https://github.com/apache/openwhisk-runtime-docker/blob/master/README.md. [Acedido em 30 de Setembro de 2021].

[73] O. Authors, "Create functions - OpenFaaS", *Docs.openfaas.com*, 2021. [Online]. Disponível em: https://docs.openfaas.com/cli/templates/. [Acedido em 30 de Setembro de 2021].

[74] "Creating Lambda container images - AWS Lambda", *Docs.aws.amazon.com*, 2021. [Online]. Disponível em: https://docs.aws.amazon.com/lambda/latest/dg/images-create.html. [Acedido em 30 de Setembro de 2021].

[75] "Using NVIDIA GPUs | Cloud Run for Anthos | Google Cloud", *Google Cloud*, 2021. [Online]. Disponível em:

https://cloud.google.com/anthos/run/docs/configuring/compute-power-gpu. [Acedido em 30 de Setembro de 2021].

[76]"OpenWhisk system details", *GitHub*, 2021. [Online]. Disponível em: https://github.com/apache/openwhisk/blob/master/docs/reference.md. [Acedido em 30 de Setembro de 2021].

[77] "Amazon ECS service quotas - Amazon Elastic Container Service", *Docs.aws.amazon.com*, 2021. [Online]. Disponível em: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/service-quotas.html. [Acedido em 30 de Setembro de 2021].

[78] "Azure security baseline for Container Instances", *Docs.microsoft.com*, 2021. [Online]. Disponível em: https://docs.microsoft.com/en-us/security/benchmark/azure/baselines/container-instances-security-baseline. [Acedido em 30 de Setembro de 2021].

[79]"Security in AWS Lambda - AWS Lambda", *Docs.aws.amazon.com*, 2021. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/lambda-security.html. [Acedido em 30 de Setembro de 2021].

[80] "Best Practices - Security - Amazon Elastic Container Service", *Docs.aws.amazon.com*, 2021. [Online]. Disponível em: https://docs.aws.amazon.com/AmazonECS/latest/bestpracticesguide/security.html. [Acedido em 30 de Setembro de 2021].

[81] "Autopilot overview | Kubernetes Engine Documentation | Google Cloud", *Google Cloud*, 2021. [Online]. Disponível em: https://cloud.google.com/kubernetes-engine/docs/concepts/autopilot-overview. [Acedido em 30 de Setembro de 2021].

[82] "Heroku Dynos | Heroku", *Heroku.com*, 2021. [Online]. Disponível em: https://www.heroku.com/dynos. [Acedido em 30 de Setembro de 2021].

[83] "Google Cloud Platform Pricing Calculator", *Google Cloud*, 2021. [Online]. Disponível em: https://cloud.google.com/products/calculator. [Acedido em 30 de Setembro de 2021].

[84] "AWS Pricing Calculator", *Calculator.aws*, 2021. [Online]. Disponível em: https://calculator.aws/. [Acedido em 30 de Setembro de 2021].