

Universidade de Brasília

Instituto de Ciências Exatas
Departamento de Ciência da Computação

Framework TASI para Revisão da Literatura por meio do Serviço de Nuvem Function-as-a-Service

Otávio Souza de Oliveira

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

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

Brasília
2023

Universidade de Brasília

Instituto de Ciências Exatas
Departamento de Ciência da Computação

Framework TASI para Revisão da Literatura por meio do Serviço de Nuvem Function-as-a-Service

Otávio Souza de Oliveira

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

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

Prof.a Dr.a Edna Dias Canedo Prof. Dr. Daniel de Paula Porto
CIC/UnB CIC/UnB

do Bacharelado em Ciência da Computação

Brasília, 18 de dezembro de 2023

Dedicatória

Dedico este trabalho ao senhor Raimundo Nonato Gomes de Oliveira, meu pai, meu herói, a quem eu me espelho como homem. Dedico a ele por ser meu melhor amigo, pelos conselhos em momentos difíceis, pelos ensinamentos e por todo esforço feito para que minha trajetória fosse mais tranquila e sem os percalços que ele enfrentou em sua grande trajetória.

Agradecimentos

Em primeiro lugar, agradeço a Deus por me dar forças quando eu já não tinha mais e pensava em desistir de finalizar este trabalho. Agradeço aos meus pais, que me proporcionaram uma excelente educação e todas as condições necessárias para que eu chegasse até aqui. Agradeço à Universidade de Brasília (UnB) por me capacitar como profissional da área de TI. Agradeço à professora Dra. Aletéia Araújo pela paciência em me orientar, pelo apoio e pela grande ajuda que me deu para a conclusão deste trabalho. Agradeço ao amigo que fiz na Dataprev, Leonardo Rebouças de Carvalho, pela ajuda com este trabalho e por todos os "puxões de orelha" necessários para o meu aprendizado. Um agradecimento especial também aos produtores de café; vocês foram essenciais para a conclusão deste trabalho, nas várias madrugadas que foram necessárias para conseguir finalizá-lo.

Resumo

A computação em nuvem revoluciona a gestão de recursos, proporcionando modelos flexíveis e simplificados. Dentre os diversos serviços de nuvem disponíveis, este trabalho destaca o *Function as a Service* (FaaS) e propõe a modernização do *framework Trend Advanced Search Indicator* (TASI), adaptando-o para uma arquitetura baseada em FaaS. O objetivo deste trabalho é avaliar o desempenho do *framework* em diferentes cenários - cenários estes que são as novas funcionalidades implementadas utilizando o serviço de nuvem FaaS - em comparação com a antiga arquitetura monolítica que o *framework* TASI possuía. O trabalho evidencia a transição bem-sucedida do *framework* para uma abordagem FaaS, mostrando sua adaptação em situações de alta demanda.

Palavras-chave: Computação em Nuvem, FaaS, Function as a Service, desempenho, arquitetura, framework

Abstract

Cloud computing revolutionizes resource management, providing flexible and simplified models. Among the various cloud services available, this work highlights *Function as a Service* (FaaS) and proposes the modernization of *framework Trend Advanced Search Indicator* (TASI), adapting it to a FaaS-based architecture. The objective of this work is to evaluate the performance of the *framework* in different scenarios - scenarios that are the new functionalities implemented using the FaaS cloud service - in comparison with the old monolithic architecture that the *framework* TASI had. The work highlights the successful transition of *framework* to a FaaS approach, showing its adaptation in high demand situations.

Keywords: Cloud Computing, Function as a Service - FaaS, performance, architecture, framework

Sumário

1	Introdução	1
1.1	Objetivos	2
1.2	Organização deste Trabalho	2
2	Computação em Nuvem	3
2.1	Considerações iniciais	3
2.2	Características da Computação em Nuvem	4
2.3	Modelos de Nuvem	5
2.4	Modelos de Serviço	6
2.5	Principais Provedores de Nuvem Pública	8
2.5.1	Amazon Web Services	8
2.5.2	Google Cloud	9
2.5.3	Microsoft Azure	10
2.6	Precificação de Nuvens	12
3	Function as a Service (FaaS)	14
3.1	Considerações Iniciais	14
3.2	Características Principais do FaaS	15
3.3	FaaS nos Provedores de Nuvem Pública	16
3.3.1	AWS Lambda	16
3.3.2	Azure Microsoft Functions	19
3.3.3	Google Cloud Function	21
4	Framework TASI	24
4.1	Considerações Iniciais	24
4.2	A linguagem JavaScript	25
4.3	NodeJS	26
4.4	Funcionalidades do Framework TASI	27
4.5	Arquitetura do Framework TASI	28
4.6	Trabalhos Relacionados	30

5	Resultados Obtidos	32
5.1	Considerações Iniciais	32
5.2	Implantação de FaaS no <i>framework</i> TASI	32
5.2.1	Metodologia Utilizada	32
5.2.2	Função que retorna lista de artigos sem repetições	33
5.2.3	Função que Retorna Métricas de Artigos	39
5.3	Considerações Finais	45
6	Conclusão	46
	Referências	48

Lista de Figuras

2.1	Modelos de serviços.	6
2.2	Quadrante Mágico para infraestrutura em nuvem e serviços de plataforma.	8
2.3	Mapa da infraestrutura global da AWS. ¹	10
2.4	Mapa da infraestrutura global da GCP. ²	11
3.1	Execução do AWS Lambda [1].	17
3.2	Processamento de arquivos PDF usando o Azure Functions [1].	20
3.3	Fluxo do Cloud Function [1].	23
4.1	Página de busca <i>Framework</i> TASI	27
4.2	Página de categorias <i>Framework</i> TASI	27
4.3	Página de filtragem de palavras <i>Framework</i> TASI	27
4.4	Antiga arquitetura do <i>Framework</i> TASI	28
4.5	Nova arquitetura do <i>Framework</i> TASI.	29
5.1	Tempo médio de conexão com o servidor para 20 requisições HTTP/s.	35
5.2	Tempo de amostragem para 20 requisições HTTP/s.	35
5.3	Tempo de latência para 20 requisições HTTP/s.	36
5.4	Vazão para 20 requisições HTTP/s.	36
5.5	Tempo médio de conexão com o servidor para 200 requisições HTTP/s.	37
5.6	Tempo de amostragem para 200 requisições HTTP/s.	38
5.7	Tempo de amostragem para 200 requisições HTTP/s.	38
5.8	Vazão para 200 requisições HTTP/s.	39
5.9	Tempo médio de conexão com o servidor para 20 requisições HTTP/s.	41
5.10	Tempo de amostragem para 20 requisições HTTP/s.	42
5.11	Tempo de amostragem para 20 requisições HTTP/s.	42
5.12	Vazão para 20 requisições HTTP/s.	43
5.13	Tempo médio de conexão com o servidor para 200 requisições HTTP/s.	43
5.14	Tempo de amostragem para 200 requisições HTTP/s.	44
5.15	Tempo de amostragem para 200 requisições HTTP/s.	44

5.16 Vazão para 200 requisições HTTP/s.	45
---	----

Lista de Tabelas

3.1	Modelo de tarifação da AWS ³	18
3.2	Modelo de tarifação da AWS.	18
3.3	Modelo de tarifação da AWS.	19
3.4	Modelo de tarifação da Microsoft Azure. ⁴	21
3.5	Modelo de tarifação da Microsoft Azure.	21
3.6	Modelo de tarifação da Google Cloud. ⁵	22
3.7	Linguagens suportadas por cada plataforma de computação sem servidor (X: não suportada; ✓: suportada)	22

Lista de Abreviaturas e Siglas

API *Application Programming Interface.*

AWS *Amazon Elastic Compute Cloud.*

BPaaS *Business Process as a Service.*

CaaS *Communications as a Service.*

CSS *Cascading Style Sheets.*

DaaS *Desktop as a Service.*

FaaS *Function as a Service.*

GCP *Google Cloud Platform.*

HTML *Linguagem de Marcação de Hipertexto.*

HTTP *Hypertext Transfer Protocol.*

IaaS *Infrastructure as a Service.*

IBM *International Business Machines.*

MaaS *Monitoring as a Service.*

NaaS *Network as a Service.*

NIST *National Institute of Standards and Technology.*

NPM *Node Package Manager.*

PaaS *Platform as a Service.*

PDF *Portable Document Format.*

REST *Representational State Transfer.*

SaaS *Software as a Service.*

SLA *Service Level Agreement.*

TASI *Trend Advanced Search Indicator.*

TIC *Tecnologia da Informação e Comunicação.*

UnB *Universidade de Brasília.*

URL *Uniform Resource Locator.*

XaaS *Everything as a Service.*

Capítulo 1

Introdução

A computação em nuvem redefine a abordagem aos recursos computacionais, oferecendo diversos modelos de utilização abstraídos de complexidades, permitindo aos usuários escolher o nível de envolvimento desejado. Em contraste com o modelo tradicional, onde recursos físicos são manipulados diretamente, a virtualização surge como solução para melhor gerenciamento interno. O *National Institute of Standards and Technology* (NIST) estabeleceu características [2] essenciais para plataformas em nuvem, incluindo autoatendimento sob demanda, acesso amplo por rede, recursos virtualizados, elasticidade rápida e serviços monitorados. Esses elementos proporcionam flexibilidade, escalabilidade e transparência na utilização de recursos computacionais.

A evolução contínua da computação tem redefinido as abordagens no desenvolvimento de sistemas, impulsionando a necessidade de soluções mais ágeis e escaláveis. Este trabalho propõe uma modernização da implementação original do *framework Trend Advanced Search Indicator* (TASI), que é um *software* projetado para facilitar o levantamento do estado da arte em bases acadêmicas. Ao longo deste estudo, explorou-se a trajetória do *framework* desde sua concepção monolítica até sua transformação em uma arquitetura baseada em Funções como Serviço (FaaS), destacando a resposta a desafios significativos de escalabilidade.

A visão panorâmica abordada aqui reconhece a importância de adaptar aplicações às demandas contemporâneas, na qual a elasticidade e a eficiência na gestão de recursos se tornam imperativas. Nesse contexto, o modelo de serviço de nuvem computacional Função como Serviço (FaaS, do inglês *Function as a Service*) emerge como uma solução eficaz para superar as limitações das aplicações monolíticas, enfatizando sua capacidade de proporcionar escalabilidade dinâmica em resposta às flutuações na demanda [3].

Além disso, na implementação realizada serão exploradas as estratégias da linguagem de programação JavaScript e do *software* NodeJS, destacando suas vantagens, tais como popularidade, simplicidade e versatilidade. A execução assíncrona do NodeJS e a sua

adequação ao desenvolvimento do *backend* podem contribuir significativamente para a eficiência do *framework* TASI.

Este trabalho também examina os resultados obtidos ao avaliar o desempenho do *framework* em diferentes cenários, comparando funções locais e orientadas a FaaS. A transição arquitetônica revelou-se bem-sucedida, superando desafios e demonstrando a capacidade de adaptação do *framework* em situações de alta demanda.

Assim, ao longo deste estudo, será fornecida uma compreensão abrangente das escolhas de arquitetura e desempenho do *framework* TASI, destacando sua relevância em um contexto computacional em constante evolução.

1.1 Objetivos

Este trabalho tem como objetivo geral modernizar a implementação do *framework* TASI por meio do uso de *Function as a Service* (FaaS). Assim sendo, para que o objetivo geral seja cumprido, faz-se necessário atingir os seguintes objetivos específicos:

- Analisar as vantagens do uso do modelo de serviço *Function as a Service*;
- Avaliar as diferentes propostas do modelo FaaS a partir dos três principais provedores de nuvem pública;
- Definir uma estratégia para selecionar quais funções a partir do TASI são adequadas ao modelo de FaaS.

1.2 Organização deste Trabalho

Este trabalho está organizado, além deste capítulo introdutório, em mais 5 capítulos. O Capítulo 2 apresenta os conceitos referentes ao ambiente de Computação em Nuvem. Na sequência, o Capítulo 3 apresenta os conceitos do modelo de serviço *Function as a Service*. O Capítulo 4 apresenta o *framework* TASI, sua arquitetura e suas características. Em seguida, o Capítulo 5 traz os resultados que foram obtidos neste trabalho. Para finalizar, o Capítulo 6 traz uma conclusão final do trabalho realizado e indica alguns trabalhos futuros para a continuidade deste trabalho

Capítulo 2

Computação em Nuvem

Neste capítulo serão apresentados os modelos, as características, e os principais provedores. Para isso, a Seção 2.1 está reservada às considerações iniciais, na qual é apresentada uma introdução geral sobre computação em nuvem. Na sequência, a Seção 2.2 aborda as características principais da computação em nuvem, e a Seção 2.3 descreve tipos de nuvem. A Seção 2.4 trás os principais modelos de serviço entregues pela computação em nuvem, caracterizando e exemplificando cada um desses modelos. A Seção 2.5 mostra os principais provedores de nuvem pública presentes no mercado atualmente, e descreve quais são os serviços que cada um desses provedores provisionam. Para finalizar, a Seção 2.6 aborda como é a precificação adotada pelos provedores de nuvem.

2.1 Considerações iniciais

A ideia precursora do ambiente de nuvem computacional (do inglês, *cloud computing*) surgiu a partir de uma evolução do conceito de grids computacionais [4]. Um *grid* consiste em um sistema que coordena recursos de maneira descentralizada, utilizando protocolos e interfaces padronizadas para entregar poder computacional com uma qualidade não trivial. Enquanto os *grids* possuem foco em uma infraestrutura que entrega recursos de armazenamento e computação, as nuvens objetivam oferecer recursos e serviços de maneira mais abstrata, a qual omite detalhes de configuração e instalação do seu usuário. A ideia é ofertar recursos computacionais como serviços, o qual o usuário deve pagar apenas pelo consumo do que tiver usado.

Assim, sob a definição de Buyya et al. [5], as nuvens são caracterizadas pelo forte suporte a virtualização e o provisionamento de recursos sob demanda. Virtualização é a configuração de um hardware de maneira que seus recursos possam ser encapsulados em diversas máquinas virtuais (instâncias) isoladas. Dessa maneira é possível, por exemplo, decompor *mainframes* e *datacenters* em instâncias customizadas de pequeno e médio

porte. Buyya et al. [5] indicaram em 2009 que a computação se tornaria um serviço de utilidade diária como telefonia e energia elétrica, tendo a nuvem computacional um papel fundamental na popularização dos serviços computacionais. Atualmente, após 14 anos, isso já é uma realidade para diversos usuários que têm incorporado em sua vida diária diversas facilidades proporcionadas pela nuvem, tais como armazenamento de terabytes de arquivos pessoais e/ou criação de máquinas virtuais para executar aplicações da empresa.

Em 2011, o *National Institute of Standards and Technology* (NIST) [2] publicou uma recomendação que se tornou a principal definição da literatura, conceituando computação em nuvem como um modelo para permitir acesso de rede onipresente, conveniente e sob demanda a um conjunto compartilhado de recursos de computação configuráveis (por exemplo, redes, servidores, armazenamento, aplicativos e serviços) que podem ser rapidamente provisionados e liberados com esforço mínimo de gerenciamento ou interação do provedor de serviços.

2.2 Características da Computação em Nuvem

O paradigma de computação em nuvem disponibiliza recursos de armazenamento e de processamento sob demanda. Nesse caso, o cliente paga apenas pelo que utilizar, tornando-se desnecessária a aquisição de componentes de alto valor monetário e, em alguns casos, muito espaço físico necessário para prover esse ambiente de alto poder computacional. Assim sendo, NIST [2] destacou cinco características essenciais de um ambiente de nuvem, as quais são:

- **Autosserviço sob demanda:** o consumidor pode provisionar recursos de computação, como tempo de servidor e armazenamento em rede, por conta própria e sem necessitar de intervenção humana dos provedores de serviço;
- **Amplo acesso por rede:** os recursos computacionais estão disponíveis por meio da rede e são acessados através de mecanismos padronizados, os quais promovem o uso por dispositivos de diversas plataformas (como *smartphones*, *tablets*, *laptops* ou *desktops*);
- **Recursos virtualizados:** os recursos de computação de cada provedor são concebidos para servir vários clientes, cada um com diferentes recursos virtuais alocados dinamicamente. Armazenamento, processamento, memória, largura de banda de rede e máquinas virtuais são alguns exemplos de recursos disponibilizados dessa maneira;
- **Elasticidade rápida:** um importante conceito que surgiu no paradigma de nuvem, elasticidade diz respeito ao provisionamento ou liberação de recursos em tempo real,

de acordo com a necessidade do usuário. Ela pode ser implementada tanto pelo redimensionamento das máquinas virtuais alocadas ao usuário - chamada elasticidade vertical, quanto pela adição de novas máquinas virtuais ao serviço prestado - chamada elasticidade horizontal. Essa propriedade possibilita ao consumidor a aparência de que os recursos são ilimitados e podem ser provisionados a qualquer momento e em qualquer quantidade;

- **Serviço monitorado:** os sistemas na nuvem controlam o uso dos recursos por meio de medições em um nível de abstração apropriado para o tipo de serviço (como armazenamento, processamento, comunicação de rede e contas de usuário ativas). A utilização de recursos pode ser monitorada, controlada e informada, gerando transparência tanto para o fornecedor quanto para o consumidor do serviço utilizado.

2.3 Modelos de Nuvem

Os provedores de nuvem computacional podem ser classificados como público, privado, híbrido e comunitário. E cada uma dessas classificações possuem particularidades e características que as diferenciam. O NIST [2] define cada um desses modelos de implantação como:

- **Nuvem pública:** é a mais popular fornecida pelos provedores para utilização do público geral. No caso das nuvens públicas o software e o hardware são gerenciados pela empresa provedora, sendo que fica a cargo do usuário apenas a utilização e o pagamento pelo serviço demandado. A infraestrutura utilizada é compartilhada por diversos clientes;
- **Nuvem híbrida:** composta de duas ou mais nuvens com diferentes modelos de implementação, de maneira que tais nuvens se comuniquem através de protocolos e/ou portabilidade por software;
- **Nuvem comunitária:** provisionada para o uso de um grupo de pessoas/organizações com valores em comum. O seu controle e a operação podem ser feitos por organizações pertencentes à comunidade, agentes externos ou um misto entre eles;
- **Nuvem privada:** a infraestrutura em nuvem é fornecida para uso exclusivo por uma única organização composta por vários consumidores (por exemplo, unidades de negócios). Pode pertencer, ser gerenciada e operada pela organização, por terceiros ou por alguma combinação deles. Além disso, a nuvem privada pode existir dentro ou fora das instalações da instituição;

2.4 Modelos de Serviço

Existem três principais modelos de serviço que pertencem a computação em nuvem, os quais podem ser observados na Figura 2.1. Cada modelo de serviço possui diferentes níveis de responsabilidade quanto à gestão necessária para desenvolver determinada aplicação. De acordo com [2] esses modelos são definidos como:

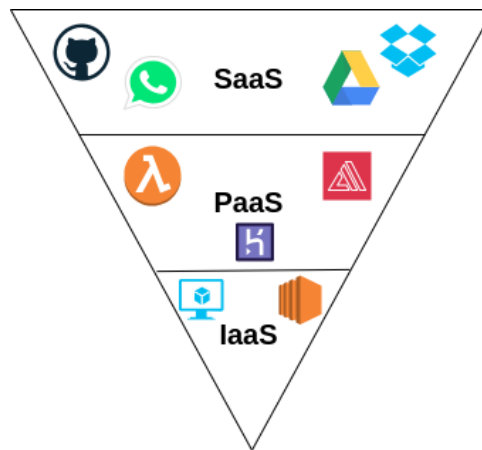


Figura 2.1: Modelos de serviços.

- ***Infrastructure as a Service (IaaS)***: um provedor de nuvem que fornece IaaS provisiona *hardware* virtual aos usuários a partir de *hardware* físico mantido pelo servidor (por exemplo, armazenamento, máquinas virtuais, infraestrutura de rede, etc.). Alguns exemplos de serviços do tipo IaaS oferecidos pelos maiores provedores de serviço de computação em nuvem são:
 - AWS - Amazon Elastic Compute Cloud (Amazon EC2);
 - GCP - Compute Engine;
 - Azure - Virtual Machines (VMs).
- ***Platform as a Service (PaaS)***: o cliente utiliza uma plataforma que pode ser utilizada por meio de bibliotecas, linguagens de programação, entre outros. O usuário é responsável pela gestão de aplicações executadas na máquina virtual. O PaaS possibilita uma camada de abstração na qual o desenvolvedor não necessita se preocupar com a infraestrutura. O serviço proporciona componentes de linguagem de programação prontos para uso, assim como bibliotecas e ambientes que facilitem o desenvolvimento de aplicações. Algumas soluções de Plataforma como Serviço são:
 - Google App Engine;
 - Oracle Apex;

- Kubernetes Engine (GKE);
 - Azure Cloud Services;
 - AWS Amplify.
- ***Software as a Service (SaaS)***: o usuário faz o uso direto das aplicações oferecidas pelo provedor, as quais são acessadas por meio de uma interface (*web browser*, aplicativos). O nível de controle do usuário está limitado a configurações específicas de usabilidade, sem acesso direto às máquinas virtuais utilizadas pelo serviço. Assim, no *Software* como um serviço a camada de abstração é ainda maior, pois o *software* é inteiramente disponibilizado como um serviço. Dessa forma, a segurança dos dados, a conectividade e os servidores necessários para o funcionamento do *software* são garantidos pela empresa provedora. Logo, é possível entender o SaaS como a disponibilização do serviço por meio de uma interface no formato web ou alguma outra interface de sistema, com foco no usuário final. Alguns exemplos de *Softwares* como Serviços são:
 - **Google Drive**: plataforma de armazenamento em nuvem;
 - **Netflix**: plataforma de *streaming* de vídeos;
 - **Clicksign**: plataforma de assinaturas digitais.

Contudo, com a criação de novos serviços possibilitados pela computação em nuvem, a definição de modelos de serviços criada pelo NIST [2] não tem conseguido suprir e enquadrar os novos serviços. Por isso, tem sido adotada a nomenclatura de tudo como um serviço (do inglês - *Everything as a Service* (XaaS)) [6].

A ideia da sigla "X" de XaaS possa ser substituído por "qualquer" serviço, dando lugar para qualquer outro serviço existente, tais como DataBse as a Service (DBaaS), Communications as a Service (CaaS) ou outros. Alguns exemplos de serviços que não se enquadram nas definições do NIST, sendo também motivadores da criação do "qualquer" coisa como serviço são:

- ***Desktop as a Service (DaaS)***: oferece desktops virtuais aos usuários finais;
- ***Network as a Service (NaaS)***: interliga diversos serviços baseados em nuvem;
- ***Monitoring as a Service (MaaS)***: sistema de monitoramento em tempo real com foco em garantir performance, segurança e disponibilidade;
- ***Business Process as a Service (BPaaS)***: integra serviços, processos, aplicações e infraestrutura para ampliação de negócios;

- **Communications as a Service (CaaS):** oferece serviços de comunicação, geralmente ligados à telefonia como a distribuição automática de chamadas (ACD);
- **Function as a Service (FaaS):** é um tipo de serviço orientado a eventos altamente elástico e escalável, o qual será descrito em detalhes no próximo capítulo.

2.5 Principais Provedores de Nuvem Pública

Com a crescente demanda de infraestrutura especializada e serviços de nuvem computacional, algumas empresas se tornaram referência no mercado. Gartner é uma empresa de consultoria que produz diversos relatórios e o Quadrante Mágico do Gartner [7] reúne os principais provedores de computação em nuvem, conforme apresentado na Figura 2.2. Nessa figura é possível perceber sete grandes empresas: Amazon Web Service, Azure Microsoft, Google Cloud Platform, Tencent Cloud, *International Business Machines* (IBM), Oracle e Alibaba Cloud. Dentre as sete empresas levantadas pelo Gartner, a *Amazon Elastic Compute Cloud* (AWS), a Azure e a *Google Cloud Platform* (GCP) lideram o mercado. Por isso, elas serão detalhadas nas próximas seções deste trabalho.



Figura 2.2: Quadrante Mágico para infraestrutura em nuvem e serviços de plataforma.

2.5.1 Amazon Web Services

A Amazon possui sua história iniciada em 1994 como uma empresa vendedora de livros, além de uma das maiores lojas online do mundo. Contudo, por volta do anos 2000 a Amazon identificou a crescente demanda de infraestrutura computacional, percebendo

uma oportunidade de ampliar sua carteira de serviços e atuar no mercado de computação em nuvem. Após seis anos de muita pesquisa e avanços tecnológicos, em 2006, surgiu a *Amazon Web Service* (AWS) [8].

Entre os setores que a AWS atua tem-se: publicidade e *marketing*, tecnologia e jogos, serviços financeiros e mídia, e entretenimento. A AWS atua com soluções em análises e *data lakes*, *machine learning*, armazenamento, computação sem servidor e outros. Atualmente, a Amazon Web Services disponibiliza mais de 200 produtos [9], entre esses:

- **Lambda:** permite a criação de funções sem servidor nas mais variadas linguagens de programação ??(por exemplo, Java, Node.js, Python etc);
- ***Application Programming Interface* (API) Gateway:** permite a criação e o gerenciamento de APIs do RESTful e APIs do WebSocket;
- **EC2:** permite ao usuário criar máquinas virtuais com opções diversas de armazenamento, processadores, redes e sistemas operacionais;
- **Route53:** esse serviço de sistemas de nomes de domínio é responsável por conectar requisições web às aplicações da *Internet*;
- **CloudWatch:** esse serviço monitora, coleta e visualizar *logs*, métricas e dados de eventos em tempo real, e com visualização em painéis;
- **AWS Amplify:** esse serviço permite ao usuário a criação e o desenvolvimento de aplicativos. É um plataforma que possibilita, além do desenvolvimento, a integração com sistemas implantados no github, gitlab e outros repositórios.

A Nuvem AWS abrange 102 zonas de disponibilidade em 32 regiões geográficas em todo o mundo. A empresa já divulgou planos para disponibilizar mais 15 zonas de disponibilidade e outras 5 regiões no Canadá, na Alemanha, na Malásia, na Nova Zelândia e na Tailândia [10]. A Figura 2.3 apresenta o atual mapa da infraestrutura da AWS.

2.5.2 Google Cloud

Em 1997 o domínio google.com foi registrado, projeto iniciado na Stanford University por Larry Page e Sergey Brin. Em 1998 a Google Inc. [11] foi fundada com o objetivo de evoluir como um software de motor de busca. Em 2008 surgiu a Google Cloud Platform(GCP), a solução de computação em nuvem que compete entre os três maiores provedores de serviços em nuvem. A GCP, disponibiliza mais de 100 produtos [12], entre eles:

¹Fonte: <https://aws.amazon.com/pt/about-aws/global-infrastructure/>



Figura 2.3: Mapa da infraestrutura global da AWS.¹

- **Cloud Function:** esse serviço permite o usuário criar funções sem servidor nas mais diversas linguagens de programação ??(por exemplo, Node.js, Java, Python, PHP, Go, etc);
- **Compute engine:** permite instanciar máquinas virtuais;
- **Voz:** software de Inteligência Artificial que permite converter texto em áudio e áudio em texto;
- **Google workspace:** fornece uma gama de APIs dos principais softwares da google cloud (por exemplo, google slide, google docs, google classroom etc).

A Google Cloud Platform, atualmente, conta com infraestrutura física em 39 regiões, 118 zonas, 187 localizações de borda de rede, e mais de 200 países e territórios [13]. A Figura 2.4 apresenta a infraestrutura do Google, e as suas atuais e futuras regiões.

2.5.3 Microsoft Azure

Bill Gates e Paul Allen iniciaram a Microsoft como é conhecida hoje em 1975. Os primeiros produtos desenvolvidos foram com foco em interpretadores de linguagens como BASIC e FORTRAN. Em 1980 eles voltaram a empresa para o ramo de desenvolvimento de sistemas operacionais. A Microsoft conta hoje com extenso e relevante catálogo de produtos, dentre eles, serviços de hardware como consoles do Xbox, *smartphones* como Microsoft Lumia, e *softwares* como o sistema operacional Windows, e o pacote Office(excel, powerpoint, word e outros). Além disso, há também o popular serviço de armazenamento em nuvem, chamado OneDrive.

²Fonte: <https://cloud.google.com/about/locations?hl=pt-br#regions>



Figura 2.4: Mapa da infraestrutura global da GCP.²

Em outubro de 2008, a Microsoft iniciou seus investimentos na área de computação em nuvem, fundando um dos três maiores provedores de serviços em nuvem, a Microsoft Azure. Atualmente, a Azure disponibiliza mais de 300 produtos [14], entre eles:

- **Azure function:** permite a criação de funções sem servidor nas mais variadas linguagens de programação (por exemplo, Python, Java, Node.js, .Net, etc);
- **Máquinas virtuais:** permite a criação de máquinas virtuais oferecendo opções de processadores, sistemas operacionais, armazenamento e redes;
- **Banco de dados:** oferece serviços de banco de dados relacionais (por exemplo MySQL, Postgres, MariaDB), não relacionais, distribuídos e sem servidor (por exemplo Azure Cosmos DB);
- **Serviços do kubernetes:** permite o gerenciamento de contêineres com o uso do kubernetes;
- **Endereços IP públicos:** permite a comunicação dos recursos do Azure com os recursos da *Internet*.

A rede global da Microsoft (WAN) é uma parte central do fornecimento de uma experiência em nuvem. Ela conecta os *datacenters* da Microsoft em mais de 60 regiões do Azure e grande malha do nós de borda. Eles são posicionados estrategicamente em todo o mundo, oferecendo assim a disponibilidade, a capacidade e a flexibilidade para atender a qualquer demanda [15].

2.6 Precificação de Nuvens

Segundo Al-Roomi et al. [16], o principal objetivo de um provedor de nuvem computacional típico é maximizar seus rendimentos, enquanto o principal objetivo de seus clientes é utilizar um serviço de melhor qualidade possível por um preço razoável. Diante desse conflito de interesses, é o processo de precificação que determina aquilo que um provedor receberá do usuário final em compensação por seus serviços prestados, caracterizando o custo final para um usuário. Weindhardt et al. [17] alegaram que o sucesso da nuvem pública no mercado somente poderia ser atingido por meio de técnicas de precificação adequadas por parte dos provedores. Este valor final pode ser estabelecido de algumas maneiras no mercado de serviços:

- **Fixa:** precificação realizada a partir de um valor pré-estabelecido em um catálogo de valores, no qual o pagamento deste valor garante o uso do recurso por um período ilimitado (*pay once*), ou por um intervalo de tempo (*pay-per-use*);
- **Dinâmica:** precificação cujo valor final é reajustado dinamicamente devido a característica do serviço, quantidade de volumes adquiridos ou preferências do cliente;
- **Dependendo do mercado:** precificação ajustada em tempo real devido a condições de mercado como leilões, nível de demanda do serviço e recursos disponíveis;

Custos iniciais, condições dos recursos oferecidos, custos de manutenção do provedor e personalização do serviço são alguns dos elementos relevantes para a determinação do preço final de serviços de nuvem. É importante ainda que fatores referentes a provedores concorrentes sejam ponderados, buscando oferecer melhores condições, custos ou outras vantagens comparativas para seus usuários.

Indo além de análise monetária, o *Service Level Agreement* (SLA) são valiosos artefatos para o exercício de comparação entre diferentes serviços e provedores. Os SLAs são termos de acordo que especificam os níveis de qualidade (QoS) oferecidos por um provedor, e as garantias mínimas entregues aos usuários de seu serviço, construindo assim um importante material de referência para eventuais auditorias sobre os serviços. Tipicamente, SLAs de nuvens especificam as métricas que deverão atingir durante a prestação do serviço contemplado. Alguns exemplos de garantias são tempo mínimo de disponibilidade (em porcentagem), tempo de resposta limite (em milissegundos) e recuperação a possíveis desastres [17].

Sob a perspectiva de usuários de Tecnologia da Informação e Comunicação (TIC), ter acesso aos recursos computacionais necessários para suas operações com custo otimizado se tornou uma prioridade que resulta em cada vez mais organizações aderindo ao mercado

de nuvens públicas. Além disso, o reconhecimento de demandas variadas dentro das organizações tem levado grande parte dessas organizações a diversificar também os serviços de nuvem contratados, com muitas delas utilizando nuvens de mais de um provedor [17].

Capítulo 3

Function as a Service (FaaS)

Neste capítulo será apresentado o conceito de Função como Serviço (FaaS, do inglês Function as a Service) ou computação *serverless* (sem servidor). A Seção 3.1 aborda um resumo geral explicando detalhadamente o que é FaaS. A Seção 3.2 trás as características principais do FaaS, suas vantagens e desvantagens. Para finalizar, a Seção 3.3 descreve como esse tipo de serviço é disponibilizado pelos principais provedores de nuvem pública.

3.1 Considerações Iniciais

O FaaS (Function as a Service) é a categoria de serviço de computação em nuvem orientada a eventos, ou seja, fornece uma interface simples para implementação e execução de funções [18]. Como citado, as funções são executadas por eventos (gatilhos), os *triggers*, que podem ser, por exemplo, métodos *Hypertext Transfer Protocol* (HTTP). A Função como um Serviço está ganhando espaço no mercado por se mostrar escalável, com alta disponibilidade e desempenho [19].

Outro ponto relevante que torna a solução atrativa é o custo sob demanda, pois a cobrança do serviço é feita com base no tempo de execução das funções implementadas. O serviço tira também qualquer responsabilidade que o cliente possa ter com infraestrutura, por isso o FaaS é conhecido também como um serviço *serverless* (do português, sem servidor) [18].

O FaaS é um serviço orientado a eventos, permitindo fácil integração com diferentes outros serviços, visto que possibilita ampla modularidade. Cada funcionalidade a ser implementada nesse modelo pode ser separada. Assim, essa função pode servir como gatilho para que outros serviços sejam executados. Da mesma forma, essa função pode ser chamada por outros serviços ou até mesmo outras funções. É importante perceber que, por ser um serviço sob demanda, quando os gatilhos não estão sendo acionados para que a função seja executada, o serviço fica "em *stand-by*", evitando custos desnecessários.

Os produtos de função como serviço possibilitam a implantação de aplicações escaláveis ao prover um serviço web que permite ao cliente rodar suas próprias aplicações, em forma de funções, sem se preocupar com a infraestrutura envolvida. Alguns exemplos de produtos oferecidos pelos maiores provedores de serviço de computação em nuvem são: AWS Lambda [20], Cloud Function [21] e Azure Functions [22], os quais serão descritos, em detalhes, nas próximas seções.

3.2 Características Principais do FaaS

A principal característica de FaaS é aumento da produtividade dos desenvolvedores, já que o serviço permite desenvolver, implantar e executar aplicativos (ou seus componentes) na nuvem, sem alocar e gerenciar servidores [23]. Além disso, os custos com esse serviço são baixos, existe uma diminuição dos riscos e ameaças diretas a infraestrutura, já que o usuário não precisa se preocupar em qual infraestrutura está implantada a aplicação. Isso tudo permite ao usuário um investimento maior do seu tempo inovando e desenvolvendo a aplicação. Assim sendo, as características principais dos serviços do tipo FaaS são [3]:

- **Programação de funções:** os programadores desenvolvem suas funções seguindo as regras fornecidas pelos fornecedores da plataforma no ambiente local. Depois disso, eles implantam as funções na plataforma usando a linha de comando na qual a plataforma primeiro salva essas funções em um banco de dados e envia os tempos de execução dessas funções para um repositório, depois retorna as URLs dessas funções para os programadores;
- **Serviço de função:** quando um usuário deseja invocar uma função, o usuário pode usar a *Uniform Resource Locator* (URL) fornecida pela plataforma ou um evento de acionamento configurado para invocar a função por meio de um *gateway* de API fornecido também pela plataforma. O balanceador de carga ou agendador da plataforma então busca a função em um banco de dados e prepara um ambiente de execução, chamado *sandbox*, obtendo o tempo de execução da função de um repositório remoto. Após isso, a plataforma inicializa o ambiente específico da aplicação e executa a função;
- **Sem host e elástico:** *Serverless* é sem *host* e elástico, o que implica que os usuários não precisam trabalhar sozinhos com os servidores. Como tal, a sobrecarga operacional poderia ser menor, pois os usuários não precisam atualizar os servidores e aplicar *patches* de segurança. No entanto, o retorno é um novo desafio, pois requer diferentes tipos de métricas (por exemplo, recursos, ocupação, tempo de execução)

a serem monitorados em uma aplicação. Dado o recurso sem *host*, os desenvolvedores não são necessários para gerir os recursos por si próprios, o que permite que o faturamento sem servidor conte apenas com os recursos sendo usados;

- ***Statelessness***: funções sem servidor em geral são executadas em contêineres efêmeros (*stateless*). Com isso, ser *stateless* implica que algumas técnicas que dependem da transferência de informações do estado não poderiam ser usadas com eficiência em aplicativos sem servidor, por exemplo, aprendizagem de máquina iterativo, algoritmos gráficos e grandes plataformas interativas e cálculo de dados. Este é um problema sério para alguns destes serviços mencionados;
- **Tempos de execução curtos**: especificamente, os tempos de execução de funções sem servidor podem durar de 100 ms a 15 min. Para esse recurso, o AWS Lambda limita particularmente as execuções de suas funções a 15 minutos e cobra dos usuários em uma unidade de 100 ms. Com isso, essa granularidade de faturamento ainda é grande para a maioria dos aplicativos, pois os tempos de execução da maioria das funções sem servidor não são tão longos. Isso demonstra o custo baixo que a computação sem servidor trás;
- **Isolamento de funções**: a maioria dos provedores públicos de serviços sem servidor atuais tais como, AWS, Google e Microsoft Azure, isolam e executam cada função em contêineres separados.

3.3 FaaS nos Provedores de Nuvem Pública

Os principais provedores de nuvem pública do mundo [24] Amazon Web Services, Microsoft Azure e Google Cloud oferecem o serviço de computação sem servidor ou FaaS. Todavia, embora cada um dos serviços de FaaS ofertado por essas plataformas tenham as características já mencionadas na Seção 3.2, existem diferenças em cada um desses serviços, (AWS, Google Cloud e Azure) que vão desde o desenvolvimento das funções até a implantação da função na plataforma, e a execução da função. E essas diferenças são fundamentais para que o usuário conheça, pois podem auxiliar na escolha da plataforma correta, para obter o desempenho ideal com base em suas cargas de trabalho.

3.3.1 AWS Lambda

A AWS Lambda foi lançada em 2014, sendo o primeiro serviço de computação sem servidor (*serveless*) disponível no mercado. O serviço disponibilizado amplo suporte para as linguagens de programação JavaScript, C#, Java, Go, Ruby, PHP e Python. Além

das linguagens com suporte nativo, pois a API Runtime da AWS permite a utilização de praticamente qualquer outra linguagem.

É possível perceber também na AWS Lambda uma forte possibilidade de integração com os demais serviços da AWS. Como por exemplo, ao implementar uma função, caso queira utilizar um *trigger* HTTP, deve-se manualmente fazer o vínculo com o serviço *API Gateway da Amazon*. Esse comportamento não é visto nos provedores concorrentes, uma vez que os *triggers* HTTP já vêm configurados. Outra forma de visualizar o forte incentivo na integração dos serviços é, por exemplo, no Amazon S3 [25] que é possível configurar que uma função seja acionada a cada vez que um novo arquivo for carregado no S3. Assim, é possível perceber que vários outros serviços possuem forte incentivo em suas conexões.

A Figura 3.1 mostra um exemplo de execução do FaaS da Amazon para processamento de imagens usando AWS Lambda. Nesse exemplo, quando o usuário carrega uma foto no serviço de armazenamento Amazon S3, um gatilho aciona a função Lambda que carrega a imagem e executa seu direcionamento.

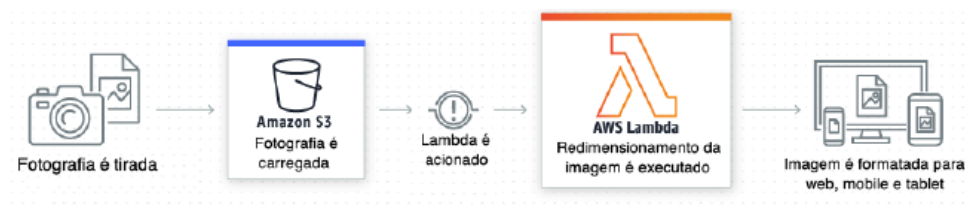


Figura 3.1: Execução do AWS Lambda [1].

O seu modelo de tarifação varia para cada região em que a função é implantada. Neste trabalho o modelo de tarifação usado foi a partir da Região da América do Sul (São Paulo), a qual pode ser vista na Tabela 3.1. Assim sendo, na Tabela 3.1 é possível notar as tarifações para arquiteturas do tipo x86 e ARM, sendo que os usuários são cobrados de acordo com as solicitações feitas, a função implantada, e a duração que a função é executada. Nas duas arquiteturas o valor 0,20 USD são cobrados por cada 1 milhão de solicitações feitas à função.

O custo da duração vai depender da quantidade de memória alocada. Para a função implantada é possível alocar qualquer quantidade de memória entre 128 MB e 10.240. A Tabela 3.2 detalha esses valores para a arquitetura x86, e a Tabela 3.3 detalha esses valores para a arquitetura ARM.

O AWS Lambda permite a implantação da função na própria plataforma, de forma didática e prática, fornecendo formulário com os campos de nome da função, linguagem, arquitetura(x86_64, arm64), políticas de permissões e configurações para a invocação da

¹Fonte: <https://aws.amazon.com/pt/lambda/pricing/>

Arquitetura	Duração	Solicitações
Preço do x86		
Primeiros 6 bilhões de GB-segundos/mês	0,0000166667 USD por cada GB-segundo	0,20 USD por 1 milhão de recompensas
Próximos 9 bilhões de GB-segundos/mês	0,000015 USD para cada GB-segundo	0,20 USD por 1 milhão de recompensas
Mais de 15 bilhões de GB-segundos/mês	0,0000133334 USD para cada GB-segundo	0,20 USD por 1 milhão de recompensas
Preço do Arm		
Primeiros 7,5 bilhões de GB-segundos/mês	0,0000133334 USD para cada GB-segundo	0,20 USD por 1 milhão de recompensas
Próximos 11,25 bilhões de GB-segundos/mês	0,0000120001 USD para cada GB-segundo	0,20 USD por 1 milhão de recompensas
Mais de 18,75 bilhões de GB-segundos/mês	0,0000106667 USD para cada GB-segundo	0,20 USD por 1 milhão de recompensas

Tabela 3.1: Modelo de tarifação da AWS¹.

Memória (MB)	Preço por 1 milissegundo
128	0,0000000021 USD
512	0,0000000083 USD
1.024	0,0000000167 USD
1536	0,0000000250 USD
2048	0,0000000333 USD
3072	0,0000000500 USD
4096	0,0000000667 USD
5120	0,0000000833 USD
6144	0,0000001000 USD
7168	0,0000001167 USD
8192	0,0000001333 USD
9216	0,0000001500 USD
10.240	0,0000001667 USD

Tabela 3.2: Modelo de tarifação da AWS.

função. Por exemplo, atribuir *endpoints* HTTP(S) à função a ser implantada. Outra forma de implantar uma função dentro da plataforma é utilizando o framework *serverless framework*. Nesse caso, é necessário acessar o *Identity and Access Management* (IAM) de usuário e atribuir algumas permissões para a implantação fora da plataforma da AWS

Memória (MB)	Preço por 1 milissegundo
128	0,0000000017 USD
512	0,0000000067 USD
1.024	0,0000000133 USD
1536	0,0000000200 USD
2048	0,0000000267 USD
3072	0,0000000400 USD
4096	0,0000000533 USD
5120	0,0000000667 USD
6144	0,0000000800 USD
7168	0,0000000933 USD
8192	0,0000001067 USD
9216	0,0000001200 USD
10.240	0,0000001333 USD

Tabela 3.3: Modelo de tarifação da AWS.

Lambda.

3.3.2 Azure Microsoft Functions

O Microsoft lançou o Azure Functions, serviço *serverless*, seguindo os passos da Amazon. Esse serviço foi lançado em março de 2016, sendo o segundo grande provedor a disponibilizar esse tipo de serviço. A Azure Functions possui forte integração com outros serviços da Microsoft, sendo um de seus principais diferenciais. As integrações vão desde linguagem de programação, possibilitando fácil importação de funções em linguagens da Microsoft como C# e F#.

Além disso, também é possível perceber uma grande facilidade em integrar com um dos principais serviços da Microsoft, o pacote *Office*. Essa integração possibilita atualizações automáticas nos documentos e planilhas, possibilitando ao usuário potencializar o uso dessas ferramentas. Além das linguagens próprias da Microsoft, o Azure Functions possui suporte para outras linguagens de programação como PowerShell, Python, Java nas versões 8 e 11, JavaScript(pacotes de Node) e TypeScript.

Esse serviço permite o carregamento de funções escritas em C#, F#, NodeJS, Java, Python ou PHP. O provedor possibilita diversidade de uso do seu serviço, como apresentado na Figura 3.2, que apresenta o processamento de arquivos *Portable Document Format* (*Portable Document Format* (PDF)). O modelo de tarifação do Azure Functions é baseado em dois tópicos:

- **Execuções:** as funções são cobradas com base no número total de execuções por mês. O primeiro milhão de execuções a cada mês é gratuito;
- **Consumo de recurso:** as funções são cobradas com base no consumo de recurso observado, medido em gigabytes por segundo.

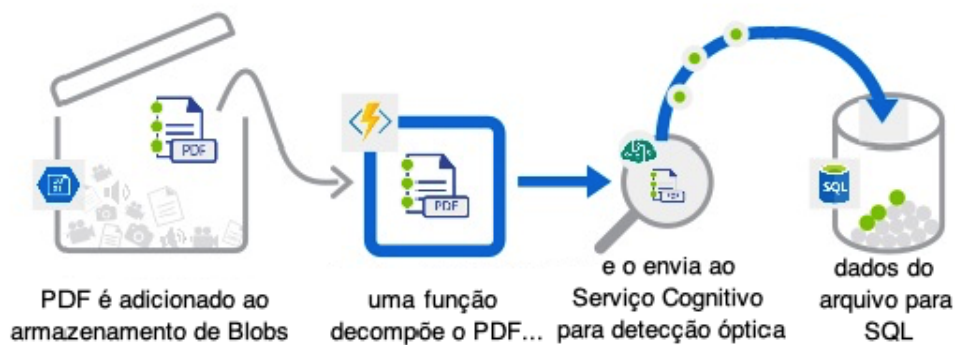


Figura 3.2: Processamento de arquivos PDF usando o Azure Functions [1].

Na Tabela 3.4 é o cálculo de cobrança de consumo de recursos e na Tabela 3.5 é o cálculo de cobrança de execuções para uma função com um consumo de memória igual a 512 MB sendo executada 3.000.000 vezes durante o mês com uma duração de execução de um segundo. O valor total para esse exemplo é de 18 USD. Neste exemplo os valores são referentes a implantação da função na região Sul do Brasil, região de implantação usada neste trabalho.

O Azure Microsoft Functions permite a implantação das funções de duas maneiras, através da plataforma Visual Studio Code, nesse caso é necessário a extensão do Azure instalado na IDE. E a outra forma é implantando diretamente na própria plataforma. O Azure Microsoft Functions fornece algumas possibilidades como a escolha do sistema operacional (linux ou windows), e também a implantação da função por código ou imagem de contêiner. Em relação à execução, o Azure Microsoft Functions fornece um gatilho HTTPS para a invocação da função implantada e também permite a invocação na própria plataforma.

²Fonte: <https://azure.microsoft.com/pt-br/pricing/details/functions/>

Cálculo de Cobrança de Consumo de Recursos	
Consumo de recurso (segundos)	
Execuções	3 milhões de execuções
Duração da execução (segundos)	1 segundo
Total de consumo de recurso	3 milhões de segundos
Consumo de recurso (GB/s)	
Consumo de recurso convertido em GBs	512 MB / 1.024 MB
Tempo de execução (segundos)	3 milhões de segundos
Total de GB/s	1,5 milhões de GB/s
Consumo de recurso cobrável	
Consumo de recurso	1,5 milhões de GB/s
Concessão gratuita mensal	400.000 GB/s
Consumo cobrável total	1,1 milhões de GB/s
Custo mensal de consumo de recurso	
Consumo de recurso cobrável	1,1 milhões de GB/s
Preço do consumo de recurso	\$0,000016/GB/s
Custo total	\$17,60

Tabela 3.4: Modelo de tarifação da Microsoft Azure.²

Cálculo de cobrança de execuções	
Execuções cobráveis	
Total mensal de execuções	3 milhões de execuções
Execuções gratuitas mensais	1 milhão de execuções
Execuções cobráveis mensais	2 milhões de execuções
Custo de execuções mensal	
Execuções cobráveis mensais	2 milhões de execuções
Preço por milhão de execuções	\$0,20
Custo de execução mensal	\$0,40

Tabela 3.5: Modelo de tarifação da Microsoft Azure.

3.3.3 Google Cloud Function

O Google Cloud Function foi lançado em 2017. O modelo de tarifação é determinado de acordo com tempo de execução da função, quantas vezes ela é acionada e quantos recursos são necessários para o seu funcionamento completo. Um exemplo de tarifação na Tabela 3.6 de uma função HTTP, com 256 MB de memória e CPU de 400 MHz,

invocada 50 milhões de vezes por mês via HTTP e executada por 500 ms cada vez, que envia 5 KB de dados. Importante notar que a Google Cloud oferece um nível gratuito para seus usuários, são oferecidos 2 milhões de invocações, 400.000 GB/segundo ou 200 GHz/segundo de tempo de execução e 5GB de tráfego de saída da internet por mês.

Métrica	Valor bruto	Nível gratuito	Valor líquido	Preço unitário	Preço total
Invocações	10.000.000	2.000.000	8.000.000	US\$ 0,00000004	US\$ 3,20
GB-segundos	375.000	400.000	0	US\$ 0,00000025	US\$ 0,00
GHz-segundos	600.000	200.000	400.000	US\$ 0,0000100	US\$ 4,00
Rede	0	5	0	US\$ 0,12	US\$ 0,00
Total/mês: US\$ 7,20					

Tabela 3.6: Modelo de tarifação da Google Cloud.³

O Google Cloud Function permite a implantação da função diretamente na própria plataforma de forma simples e didática, fornecendo as opções das linguagens, região de implantação e opção de invocações autenticadas ou não autenticadas a função a ser implantada.

Assim sendo, a Google foi a última empresa, entre os grandes provedores, a disponibilizar um serviço *serverless*. O Cloud Functions possui suporte para as linguagens de programação Go, Java, .NET, C#, F#, NodeJS, Python e Visual Basic (conforme pode ser visto na Tabela 3.7). Assim como no Azure Functions, o Cloud Functions possui gatilhos (*triggers*) de forma nativa, facilitando a configuração e a utilização das funções. A Cloud Functions possui o diferencial da possibilidade de integração com o *Firebase*, que consiste em uma plataforma de criação de aplicativos *web* e *mobile*.

	Java				.NET					Python						PHP			Go	Ruby				PowerShell	Node.js					
Versão	21	17	11	8	3.1	4.8	6	7	8	3.7	3.8	3.9	3.10	3.11	3.12	7.4	8.1	8.2	1.x	2.6	2.7	3.0	3.2	7.2	20	18	16	14	12	
AWS	✓	✓	✓	✓	X	X	✓	X	X	✓	✓	✓	✓	✓	✓	X	X	X	X	✓	X	✓	X	✓	X	✓	✓	✓	✓	X
Google	✓	✓	✓	X	✓	X	✓	X	X	X	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	X	✓	✓	✓	✓	X
Azure (Windows)	X	✓	✓	✓	X	✓	✓	✓	✓	✓	X	X	X	X	X	X	X	X	X	X	X	X	X	✓	✓	✓	✓	✓	X	X
Azure (Linux)	X	✓	✓	✓	X	X	✓	✓	✓	✓	✓	✓	✓	✓	✓	X	X	X	X	X	X	X	X	✓	✓	✓	✓	✓	X	X

Tabela 3.7: Linguagens suportadas por cada plataforma de computação sem servidor (X: não suportada; ✓: suportada)

A Google disponibiliza dois tipos diferentes de *Cloud Functions*, que são as *HTTP Functions* e *Background Functions*. A diferença entre elas é justamente o *trigger* utilizado. As *Background Functions* são acionados por outros *cloud events*, ou métodos HTTPs. À Figura 3.3 mostra o funcionamento do serviço de FaaS do Google.

³Fonte: Capturado de <https://cloud.google.com/functions/pricing?hl=pt-br>



Figura 3.3: Fluxo do Cloud Function [1].

Assim, diante das vantagens e das limitações do uso do modelo de serviço FaaS apresentados neste capítulo, este trabalho propõe a implementação de algumas funções do *framework* TASI por meio do modelo de serviço sem servidor. Esta proposta será detalhada no Capítulo 4 deste trabalho de conclusão de curso

Capítulo 4

Framework TASI

Neste capítulo será apresentado o *framework* TASI, o qual facilita o levantamento do estado da arte em bases acadêmicas. Inicialmente, a Seção 4.1 apresenta uma visão geral sobre o *framework* TASI. Na sequência a Seção 4.2, aborda as características da linguagem de programação JavaScript, adotada na implementação do *framework*. Em seguida, a Seção 4.3 aborda as características da linguagem de programação NodeJS, adotada também no *framework*. Na sequência a Seção 4.4, mostra as funcionalidades do *framework* TASI. Em seguida, a Seção 4.5 descreve a arquitetura adotada na implementação do *framework*. Para finalizar, a Seção 4.6 apresenta os trabalhos relacionados ao *framework* TASI.

4.1 Considerações Iniciais

À medida que a computação evolui, com o surgimento de novas linguagens de programação e atualizações das já existentes, assim como o desenvolvimento e atualização dos serviços de nuvem, torna-se necessário ajustar as aplicações para atender às demandas atuais.

Nesse cenário, as aplicações monolíticas apresentam limitações que dificultam ganhos de escalabilidade. Para superar essa limitação, o uso de FaaS [26] torna-se essencial, pois proporciona elasticidade rápida e automática. Dessa forma, o provedor dimensiona a aplicação vertical ou horizontalmente, conforme a demanda exigida pela função, garantindo assim uma resposta eficiente.

O FaaS, como discutido no Capítulo 3, é um modelo de serviço que permite ao cliente carregar um trecho de código em uma linguagem de programação específica fornecida pelo provedor. O provedor, por sua vez, processa esse trecho de código a partir de uma chamada de serviço, geralmente realizada via API REST.

Outro aspecto relevante é a escolha da linguagem de programação no desenvolvimento de aplicações. Destaca-se o crescente uso e importância da linguagem de programação

JavaScript [8]. Essa linguagem é amplamente adotada pelos desenvolvedores na implementação de aplicações web [27]. É utilizada tanto para interatividade entre páginas da *Internet* quanto em *frameworks* que permitem o uso de JavaScript no lado do servidor, como o NodeJS [28].

Diante desse contexto, este trabalho propõe a modernização do *framework* TASI, disponível em <https://github.com/unb-faas/tasi>. Esse *framework* oferece aos usuários a capacidade de buscar documentos acadêmicos nas principais bases de dados online, incluindo ACM[29], arXiv[30], bioRxiv[31], IEEE[32], medRxiv[33], PubMed[34] e Scopus[35]. Além disso, o *framework* gera gráficos de diversas análises sobre os documentos obtidos, proporcionando uma experiência enriquecedora aos usuários do sistema.

4.2 A linguagem JavaScript

Em 1995 surgiu a linguagem JavaScript, criada por Brendan Eich a pedido da empresa Netscape, conhecida pelo seu navegador web Netscape Navigator. O objetivo na criação da linguagem era que ela fosse simples, fácil de usar e dinâmica [8].

O JavaScript é utilizado pelos desenvolvedores para criar dinâmicas ou ações entre as páginas Linguagem de Marcação de Hipertexto (HTML) de um sistema web. Essa linguagem permite a criação de menus, rodapés nas páginas com suas respectivas ações, efeitos dentro das páginas (por exemplo, *scroll*), efeitos em botões HTML para realizarem determinadas ações, interação, navegação, troca de informações entre as páginas, disparos de alertas, etc. Em resumo, o JavaScript permite a interação entre o usuário final e a aplicação web. O JavaScript é usado em complemento às linguagens HTML, *Cascading Style Sheets* (CSS), sendo primordial para que a estrutura de uma página (HTML) e a sua estilização (CSS) possam ter a dinâmica necessária para que haja um uso agradável do usuário final dentro do sistema web proposto.

Dessa forma, o JavaScript possui vantagens significativas para os desenvolvedores, incluindo [8]:

- **Popularidade:** de acordo com uma pesquisa realizada pela plataforma Stack Overflow em 2022 [27], o JavaScript é usado por 67.9% dos desenvolvedores profissionais, sendo a linguagem de programação mais utilizada no mundo consecutivamente nos períodos entre 2012 e 2022;
- **Simplicidade:** a sua estrutura é simples, tornando a linguagem fácil de aprender e implementar;
- **Velocidade:** os *scripts* são executados diretamente no navegador web, não sendo necessário se conectar a um servidor ou depender de um compilador;

- **Versatilidade:** o JavaScript é suportado nos principais navegadores presentes hoje em dia, tais como Firefox, Chrome, Edge e Safari.

4.3 NodeJS

Em 2009, Ryan Dahl propôs uma nova linguagem de programação, o NodeJS, devido ao grande sucesso da linguagem JavaScript [36]. O NodeJS é um software baseado no interpretador V8 do Google e se caracteriza por ser um ambiente que executa JavaScript fora de um navegador web [37].

O principal objetivo que o NodeJS se propõe a resolver é a questão da alta escalabilidade que as aplicações web demandam atualmente [36]. Com isso, o NodeJS permite a execução assíncrona de processos, e foi projetado para serviços de rede [37]. A vantagem da programação assíncrona e orientada a eventos é a utilização plena dos recursos do sistema. Portanto, esse *design* é muito adequado para implementar serviços de rede no backend de aplicações web [37].

Outra vantagem obtida com a utilização do NodeJS é a facilidade que os desenvolvedores têm ao usar essa linguagem. Isso ocorre porque o NodeJS leva para o processamento do lado do servidor a linguagem de programação mais consolidada no processamento do lado do cliente, o JavaScript [36], conforme dito na Seção 4.2 é a linguagem de programação mais usada no mundo [27]. Isso diminui a dificuldade no aprendizado de outra linguagem de programação. Além disso, ele conta com uma grande comunidade de suporte, que oferece uma grande quantidade de bibliotecas para reaproveitamento de código por meio do *Node Package Manager* (NPM)[38].

O NPM é o maior registro de software *Open Source* do mundo, permitindo que desenvolvedores do mundo inteiro compartilhem os pacotes que desenvolvem, facilitando a vida de muitas organizações que utilizam o NPM para gerenciar seus desenvolvimentos privados [38].

O NPM permite, por exemplo, a instalação do *framework express* [39], usado neste projeto, que atua como suporte para a construção de APIs *Representational State Transfer* (REST) usando NodeJS [36].

4.4 Funcionalidades do Framework TASI



Figura 4.1: Página de busca *Framework* TASI

A Figura 4.1 mostra a funcionalidade responsável por realizar buscas de documentos acadêmicos nas bases de dados listadas na Seção 4.1.

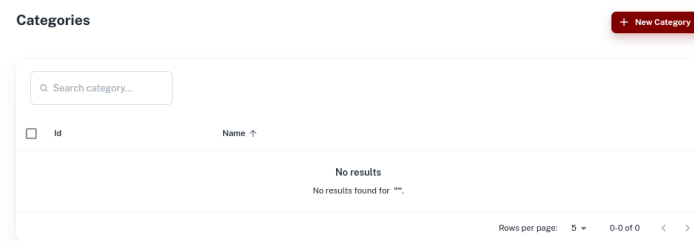


Figura 4.2: Página de categorias *Framework* TASI

A Figura 4.2 mostra a funcionalidade responsável por categorizar as buscas realizadas pelo usuário final.

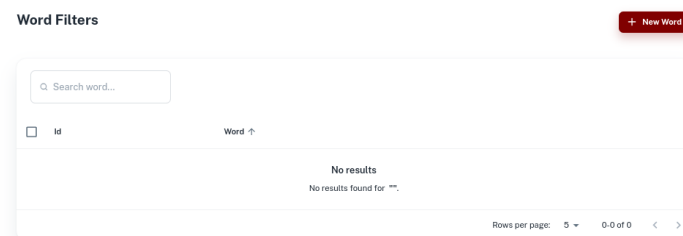


Figura 4.3: Página de filtragem de palavras *Framework* TASI

A Figura 4.3 representa a funcionalidade que permite ao usuário filtrar as palavras desejadas de uma busca realizada pelo *framework*.

4.5 Arquitetura do Framework TASI

Inicialmente, a arquitetura do *framework* TASI é composta por três módulos principais (*Frontend*, *Backend* e *Findpapers*), nos quais esses módulos se comunicam por meio de requisições HTTP, como apresentado na Figura 4.4. Além disso, na arquitetura do *framework* TASI, há a interface com os repositórios que mantêm diversos artigos científicos em todo o mundo.

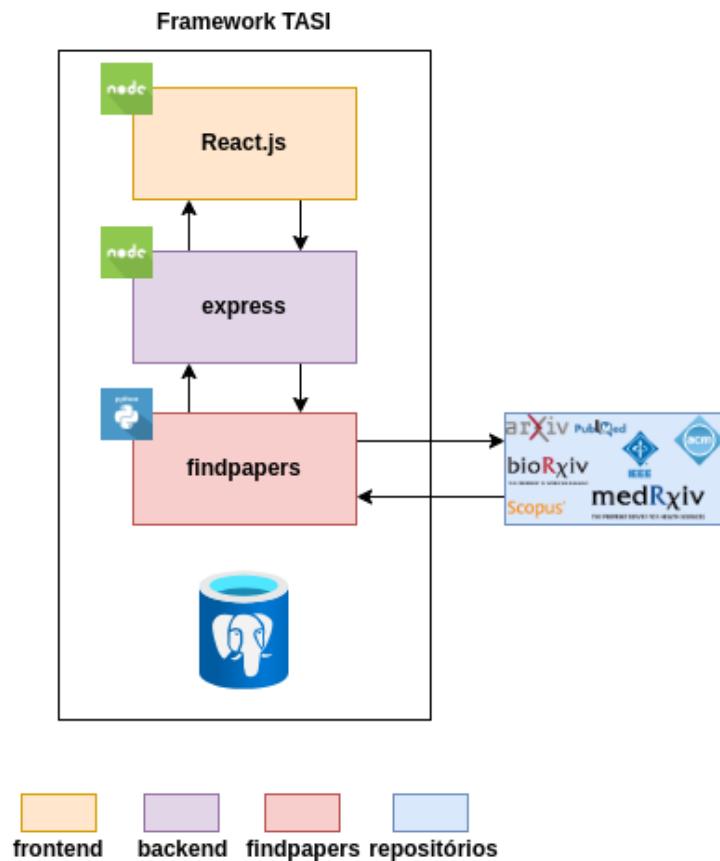


Figura 4.4: Antiga arquitetura do *Framework* TASI

No entanto, essa arquitetura monolítica enfrenta limitações de escalabilidade, conforme mencionado anteriormente na Seção 4.1. Para superar esse desafio, optou-se pela adoção de FaaS. Foram implementadas duas funções orientadas a FaaS, que serão detalhadamente descritas no Capítulo 5. Com essa abordagem, a aplicação pode usufruir da principal característica do FaaS, a elasticidade rápida. Isso permite que a aplicação escale de maneira horizontal e vertical de forma ágil e eficiente, proporcionando uma resposta rápida às mudanças na demanda.

Nesse contexto a Figura 4.5 representa a nova arquitetura do *framework*, com a adição de FaaS.

Com essa nova arquitetura, o módulo *Backend* passou por algumas alterações. Além de continuar a comunicação via requisições HTTP com os módulos *Frontend* e *Findpapers*, o *Backend* agora também contém os gatilhos HTTP responsáveis por fazer requisições nos provedores de nuvem AWS, GCP e Microsoft Azure. É nesses provedores que estão implementadas as funções utilizadas pelo *framework*. Assim sendo, os três principais módulos do *framework* TASI, conforme apresentado na Figura 4.5, são:

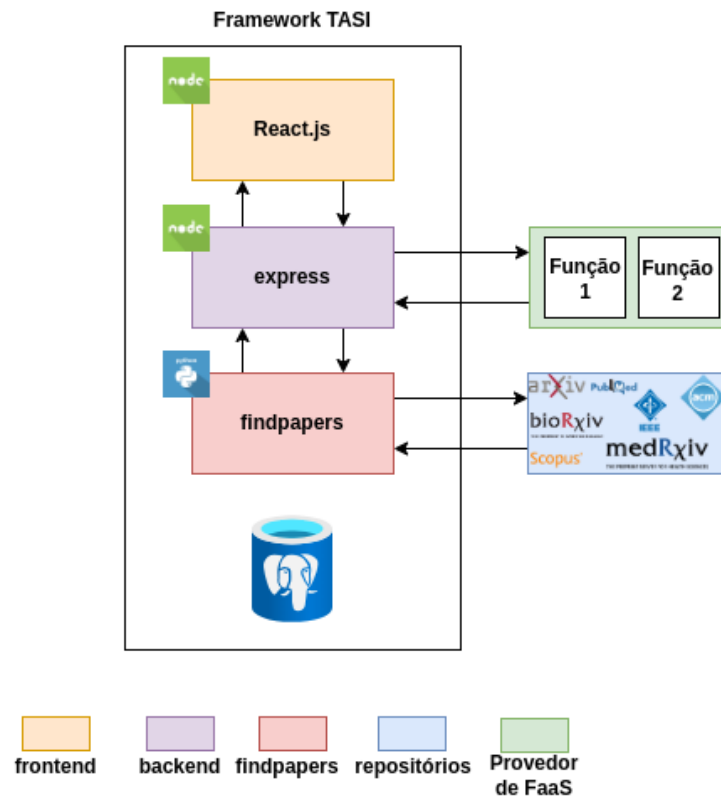


Figura 4.5: Nova arquitetura do *Framework* TASI.

- **Frontend:** este módulo é responsável por exibir ao usuário as páginas referentes ao *framework* (por exemplo, formulário de busca de artigos, amostragem de palavras-chave da busca realizada, tela de boas-vindas ao *framework*, página de gerência de buscas realizadas, etc.). Ele é responsável por abstrair os dados obtidos a partir do módulo *Backend* e exibi-los de forma sucinta ao usuário. Este módulo foi desenvolvido em um ambiente NodeJs 4.3 na versão 16.10.0, utilizando a biblioteca ReactJS [40], pertencente ao JavaScript 4.2, para criar interfaces de usuário em páginas web;
- **Backend:** Este módulo é responsável por conter as APIs REST que alimentam a base de dados PostgreSQL [41] e fornecem os dados necessários ao *Frontend* da aplicação. Utilizando o serviço de nuvem FaaS, o módulo inclui gatilhos HTTP que realizam requisições nos serviços AWS Lambda, Google Cloud Functions e Azure

Functions, pertencentes aos provedores de nuvem AWS, GCP e Microsoft Azure, conforme mencionado no Capítulo 3. Este módulo foi implementado em um ambiente Node.js na versão 14, utilizando o *framework* Express [28], que é especializado na criação de APIs REST.

- **Findpapers:** este módulo, desenvolvido em Python na versão 3.7 e obtido do projeto <https://github.com/jonatasgrosmann/findpapers>, visa fazer requisições nas bases de dados ACM[29], arXiv[30], bioRxiv[31], IEEE[32], medRxiv[33], PubMed[34] e Scopus[35] para obter os documentos científicos requisitados pelo usuário.

4.6 Trabalhos Relacionados

O levantamento bibliográfico realizado com o intuito de buscar outros sistemas que realizam tarefas semelhantes ao *framework* TASI foi bem-sucedido. No entanto, do ponto de vista de sistemas que utilizam FaaS como um componente importante, a pesquisa realizada não foi tão conclusiva, indicando que essa é uma área pouco explorada nesse tipo de sistema. Assim alguns exemplos de trabalhos relacionados a propsta do *framework* TASI são descritas a seguir.

No trabalho [42] o objetivo foi projetar uma ferramenta de resumo da web. O ponto principal no desenvolvimento desta ferramenta é apoiar cientistas acadêmicos e estudantes quando pesquisam referências científicas sobre tópicos de pesquisa.

As vantagens do trabalho [42] são:

- O protótipo funciona bem para sites de texto;
- A avaliação da relevância de um documento é mais fácil do que a avaliação dos resultados recuperados pelo Google.

Contudo, o sistema [42] possui desvantagens a serem consideradas:

- Alguns conteúdos dos resumos não refletem os conteúdos originais;
- Alguns dos resumos obtidos são difíceis de entender;
- No caso de algumas consultas, o sistema fornece mensagens de erro.

Em outro trabalho [43], foi desenvolvida uma plataforma chamada Rexplore, cujo objetivo é integrar análises estatísticas, tecnologias semânticas e análises visuais para fornecer suporte eficaz na exploração e na interpretação dos dados acadêmicos buscados pela plataforma.

O Rexplore [43] apresenta diversas vantagens:

- Visualização de tendências acadêmicas de forma extremamente detalhada, possibilitando uma análise aprofundada de padrões específicos em um contexto acadêmico;
- Métodos para identificar relações 'semânticas' entre autores;
- Capacidade de realizar buscas específicas e precisas para identificar especialistas acadêmicos em diversas dimensões ou áreas de conhecimento.

Essas características fazem do Rexplore [43] uma ferramenta valiosa para compreensão e exploração eficiente do cenário acadêmico. Quanto a possibilidades futuras de melhorias, o artigo [43] menciona a perspectiva de aprimorar as funcionalidades por meio da integração com outras fontes de dados relevantes para atividades acadêmicas.

O trabalho [44] propõe uma técnica de busca capaz de pesquisar o extenso número de publicações acadêmicas utilizando tecnologia de computação em grade, a fim de melhorar o desempenho da pesquisa. A técnica de pesquisa é implementada como serviços de rede interconectados, oferecendo um mecanismo para acessar diferentes locais de dados.

O GAPS [44], como é chamado, apresenta vantagens, como um tempo de resposta menor em comparação a outros sistemas de busca de documentos acadêmicos. Adicionalmente, seu tempo de execução é superior aos sistemas com o mesmo propósito, destacando-se pela eficiência nesse contexto.

O projeto RelPath [45] propõe um sistema para auxiliar na tarefa de encontrar artigos e autores relevantes para um determinado documento científico. O RelPath inclui o artigo submetido em uma rede de citações e estabelece a relevância das arestas da rede, por meio das quais é possível construir ramos de estudos e estabelecer um *ranking* de autores.

Uma desvantagem do RelPath [45] está na listagem de autores. Em algumas situações, o mesmo autor com nome composto é listado inúmeras vezes, em alguns casos somente com o primeiro nome e, em outros, com o primeiro e segundo nome.

O projeto [46] propõe um sistema auto-supervisionado de ponta a ponta, chamado *SciConceptMiner*, para a captura automática de conceitos científicos emergentes de fontes de conhecimento independentes (dados semiestruturados) e publicações acadêmicas (documentos não estruturados).

O *SciConceptMiner* [46] descobriu mais de 740 mil conceitos científicos em todas as áreas de pesquisa, provenientes de fontes como Wikipedia, UMLS e artigos acadêmicos, com alta precisão (94,7%). Esses conceitos estão integrados para construir o *Microsoft Academic Graph* [47], que publica uma das maiores taxonomias científicas de domínio. Isso permite a fácil exploração do conhecimento científico, bem como facilita diversas aplicações *downstream*, como recuperação, resposta a perguntas e recomendações.

Capítulo 5

Resultados Obtidos

Neste capítulo serão apresentados os resultados obtidos a partir da versão do *framework* TASI implementada neste trabalho por meio de FaaS, e uma análise comparativa com a versão monolítica do TASI. A Seção 5.1 inicia este capítulo dando uma visão geral dos testes realizados neste trabalho. A Seção 5.2 aborda a implantação de FaaS no *framework*, destacando a metodologia utilizada nos testes, os provedores de nuvem envolvidos, as tecnologias envolvidas nos testes e as métricas usadas neste trabalho para comparar FaaS com funções locais dentro do *framework*. Por fim, a seção 5.3 trás uma conclusão final dos resultados obtidos.

5.1 Considerações Iniciais

Com a implantação do serviço de nuvem Function-as-a-Service no *framework* TASI, foi necessário realizar uma análise comparativa das funções pertencentes aos serviços AWS Lambda 3.3.1, Azure Function 3.3.2, Google Function 3.3.3 e localmente no próprio *framework*. O objetivo é analisar os ganhos e perdas associados ao uso de FaaS no *framework* em comparação com as mesmas funções implementadas localmente no *framework* e com isso avaliar os cenários onde cada uma dessas abordagens se encaixam melhor.

5.2 Implantação de FaaS no *framework* TASI

5.2.1 Metodologia Utilizada

Neste trabalho, foram implementadas duas funções com a utilização do serviço de nuvem Function-as-a-Service (FaaS). As funções foram implementadas nas nuvens Microsoft Azure 2.5.3, Amazon Web Services 2.5.1 e Google Cloud 2.5.2, utilizando os serviços de

computação sem servidor disponibilizados por cada uma das nuvens, conforme descrito na Seção 3.3, e detalhados nas Seções 5.2.2 e 5.2.3.

As métricas foram traçadas utilizando o software Apache JMeter [48] na versão 5.6.2. O Apache JMeter é uma ferramenta de teste de desempenho que se concentra principalmente em medir o desempenho de aplicações web por meio de simulações de carga. Ele pode ser usado para simular uma carga pesada em um servidor, grupo de servidores e rede, testando assim sua resistência ou para analisar o desempenho geral do sistema sob diferentes tipos de carga. Dito isso, foram realizados testes de carga e testes de estresse na aplicação, com os testes focados nas funções implementadas no *framework*, tanto nas que utilizam a abordagem FaaS quanto nas locais. As métricas utilizadas neste trabalho foram:

- **Tempo de Conexão com o Servidor:** refere-se ao tempo que o cliente (Apache JMeter) leva para estabelecer conexão com o servidor;
- **Latência:** refere-se ao atraso entre o envio de uma solicitação e a recepção da resposta [49];
- **Tempo de Amostragem:** define o tempo necessário para o cliente coletar as amostras de dados;
- **Vazão:** representa a taxa de transferência de dados dentro de uma unidade de tempo [50].

5.2.2 Função que retorna lista de artigos sem repetições

No código 5.1, é apresentado o código JavaScript utilizado para implementar a função responsável por receber uma listagem de artigos científicos e retornar uma versão da lista sem repetições. Essa função desempenha um papel crucial no contexto do *framework*, uma vez que, durante a busca de documentos acadêmicos em diversas bases de dados promovida pelo *framework* TASI, é essencial verificar a presença de documentos duplicados. Tal verificação se torna imperativa para assegurar que outras funcionalidades do sistema não sejam impactadas negativamente. Essa função foi escolhida para ser implementada via FaaS devido à grande carga de dados que ela pode receber, aproveitando assim a principal característica proporcionada pelo FaaS: a elasticidade [3].

Listing 5.1: Função que retorna lista de artigos sem repetições

```
const localCheckPapers = (req, res) => {  
  const {data} = req.body  
  const map = new Map()  
  data.forEach(item => {  
    map.set(item.doi, item)  
  })  
  const uniqueList = Array.from(map.values())  
  return res.status(200).json(uniqueList)  
};
```

Esse trecho de código 5.1 em específico representa a função local implementada dentro do *framework* TASI. Porém, esse mesmo código foi usado para implantar as funções no AWS Lambda 3.3.1, Azure Function 3.3.2 e Google Function 3.3.3.

Para avaliar o desempenho e as limitações das funções implementadas foram realizados dois cenários de testes, os quais serão descritos em detalhes.

Cenário 1 - Testes de estresse para 20 requisições HTTP/s

Foram feitas 20 requisições HTTP/s simultâneas no servidor contendo o *framework* TASI, e suas respectivas funções. Com o objetivo de estressar o servidor para verificar as métricas listadas para a função local, e para avaliar as métricas nas funções que utilizam abordagem orientada a FaaS. As cargas de dados enviados para cada função foram de 20, 200 e 2000.

Assim, observa-se na Figura 5.1 que o tempo de conexão do cliente (Apache Jmeter) com a função local foi menor do que nas funções na nuvem. Porém para uma carga de 2000 a função local obteve o pior tempo de conexão com o servidor, e a função implantada no AWS Lambda apresentou o menor tempo para essa carga de dados (2000), com um nível de estresse ainda baixo no servidor.

Na Figura 5.2 nota-se que para as cargas de 20, 200 e 2000 dados enviados, a função local possui o menor tempo de amostragem, ou seja, o tempo em que os dados foram enviados do cliente (Apache Jmeter) ao servidor, até o tempo em que o cliente obteve a resposta e conseguiu processar todos os dados, a função local obteve o menor tempo.

Na Figura 5.3, a função local também obteve um tempo menor de latência em comparação com as outras funções. Isso se deve ao fato de que a maioria das LANs (*Local Area Network*) tem pouca ou nenhuma latência, enquanto o usuário da WAN (*Wide Area Network*) frequentemente lida com condições de alta latência que podem afetar significativamente o desempenho do aplicativo [49].

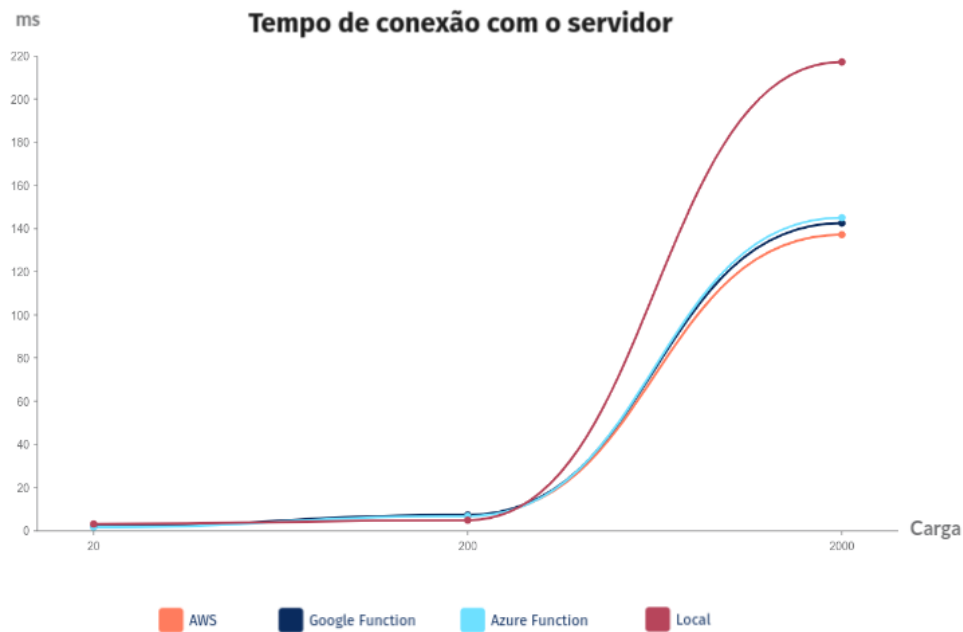


Figura 5.1: Tempo médio de conexão com o servidor para 20 requisições HTTP/s.

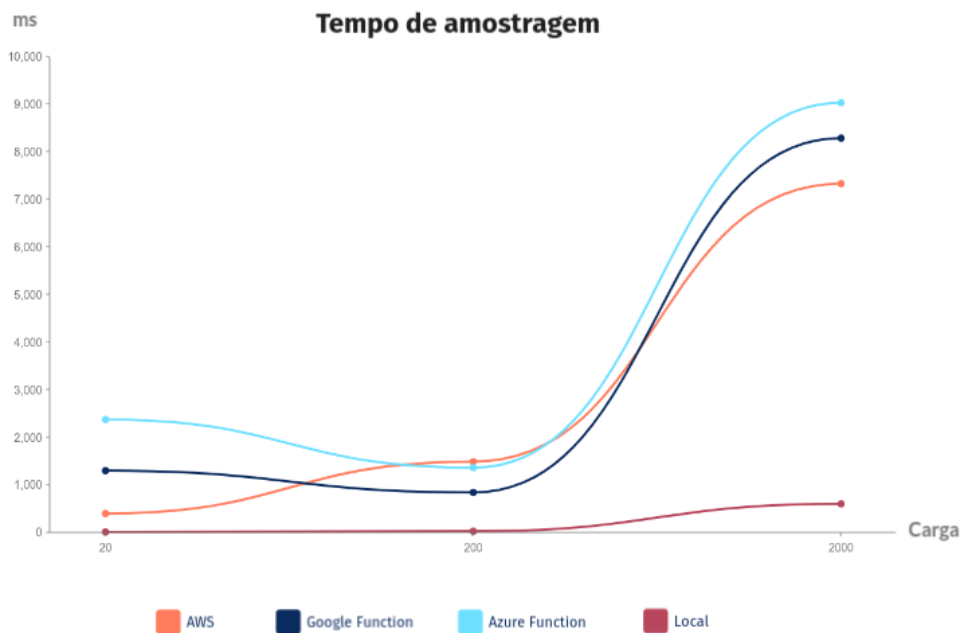


Figura 5.2: Tempo de amostragem para 20 requisições HTTP/s.

Na Figura 5.4 nota-se que a vazão da função local é maior comparando com as abordagens orientadas a FaaS, as requisições feitas na função local obtiveram em média uma taxa de transferência 12 bytes/segundos para uma carga de 2000 dados. Isso pode indicar um desempenho melhor em termos de capacidade de processamento de solicitações na função local.

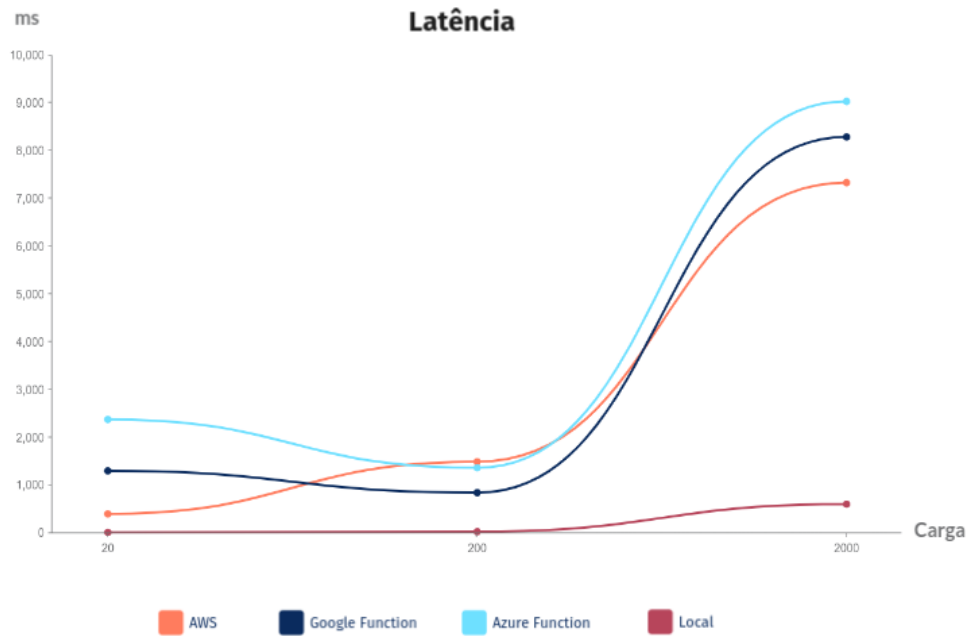


Figura 5.3: Tempo de latência para 20 requisições HTTP/s.

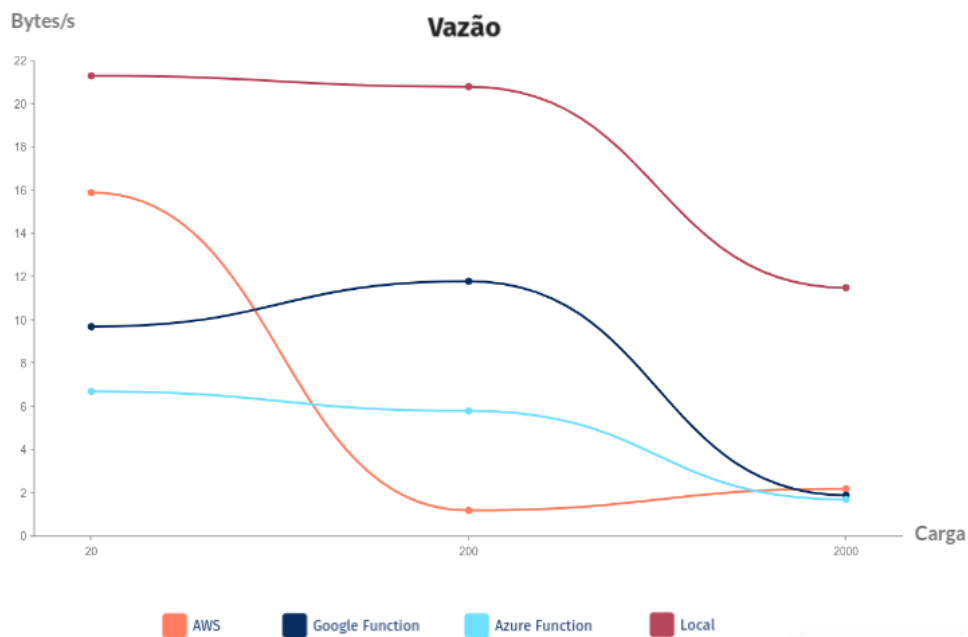


Figura 5.4: Vazão para 20 requisições HTTP/s.

Com isso é possível concluir que utilizar esses níveis de carga e estresse para uma abordagem orientada a FaaS não é o ideal, não representou nenhum ganho nas métricas mostradas, isso também se deve ao fato de que FaaS possui um *overhead* diretamente associado a sua invocação. Portanto, para essa abordagem FaaS não é uma boa opção.

Cenário 2 - Testes de estresse para 200 requisições HTTP/s

Neste segundo cenário foram feitas 200 requisições HTTP/s simultâneas no servidor contendo o *framework*. As cargas de dados usadas foram de 20, 200 e 2000 em cada uma das funções presentes no *framework*.

Assim sendo, observa-se na Figura 5.5 que para as cargas de 20 e 200 dados o tempo médio de conexão com o servidor é parecido para todas as funções. Porém para as cargas de 2000 e 3000 dados o AWS Lambda possui o menor tempo, seguido por Google Function, Azure Function, por sua vez, a função local possui o pior tempo de conexão.

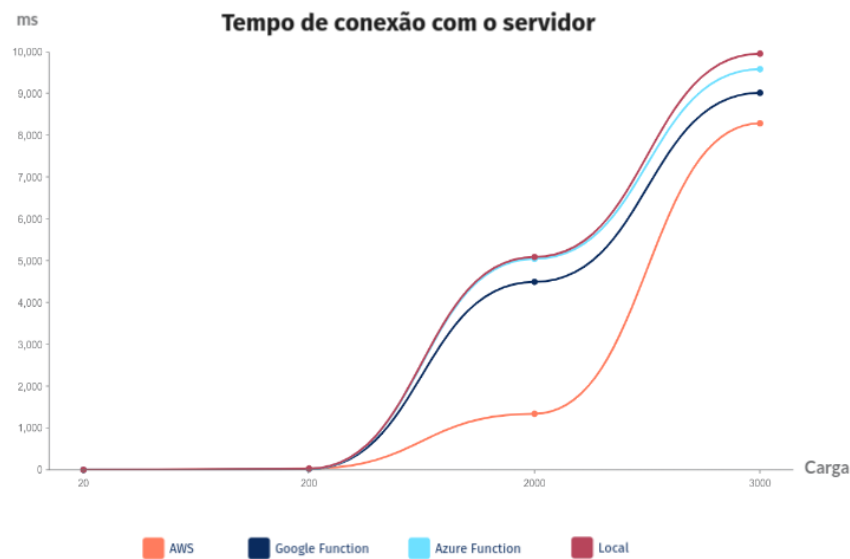


Figura 5.5: Tempo médio de conexão com o servidor para 200 requisições HTTP/s.

Nos tempos de amostragem Figura 5.6 e latência Figura 5.7, os tempos das funções são ligeiramente parecidos. É importante observar que a função local entre as cargas de 20 e 200 possui os melhores tempos de latência e amostragem. Porém isso se inverte completamente em uma carga de dados igual a 3000.

Na Figura 5.8 temos a vazão média por segundo para cada uma das funções. Inicialmente para uma carga pequena de 20 dados enviados a função local se sobressai bastante com relação às outras, isso representa uma maior taxa de transferência de dados por segundo. Porém, para cargas maiores de 2000 e 3000 dados esses valores se igualam para uma taxa de transferência de aproximadamente 0,2 bytes por segundo. Essa redução brutal na taxa de transferência de dados é o primeiro sintoma de problemas de capacidade, onde os servidores não conseguem acompanhar o número de requisições feitas, acompanhadas de um grande volume de dados [50].

É notado por sua vez que para grandes cargas de dados as funções orientadas a FaaS possuem um desempenho melhor que a função local. Isso se justifica pelo fato de FaaS

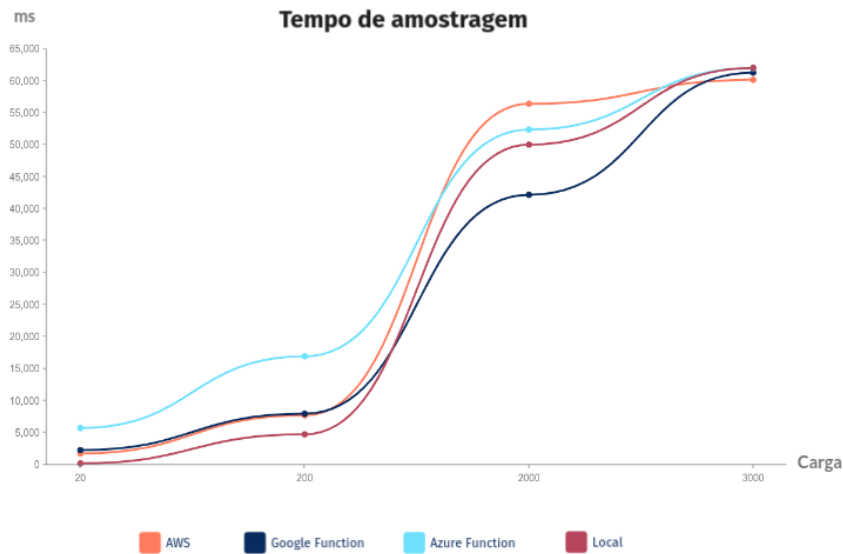


Figura 5.6: Tempo de amostragem para 200 requisições HTTP/s.

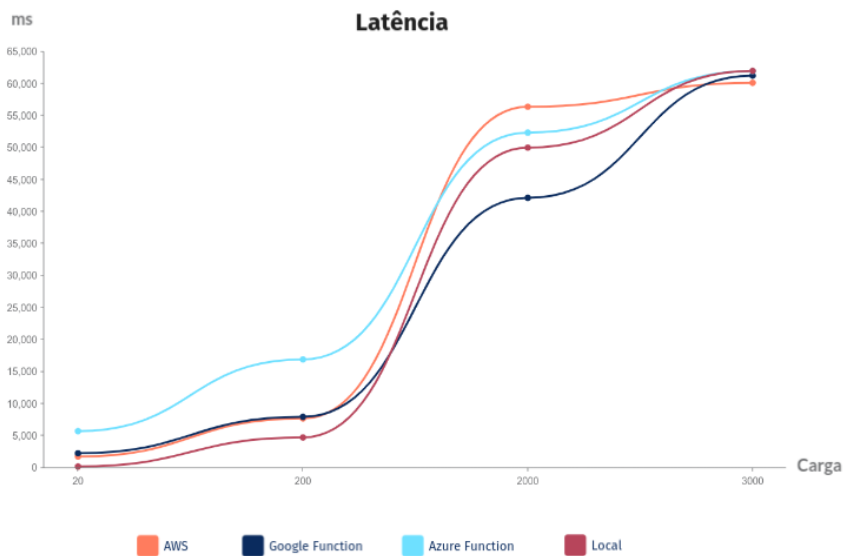


Figura 5.7: Tempo de amostragem para 200 requisições HTTP/s.

ter por característica elasticidade [3]. Portanto, essas funções se adaptam dinamicamente à demanda exigida, escalando recursos para cima ou para baixo. Logo, com o aumento das cargas de estresse e de dados enviados para níveis mais altos, o comportamento de FaaS é mais adequado para esse tipo de abordagem. Por outro lado, a função local passa a ter um desempenho pior, pois à medida que mais requisições chegam no servidor, maior será a quantidade de recursos necessários para atender essa demanda. Isso significa que em determinado momento esses recursos ficam perto de se esgotar, devido a grande quantidade de dados e ao enfileiramento das requisições no servidor.

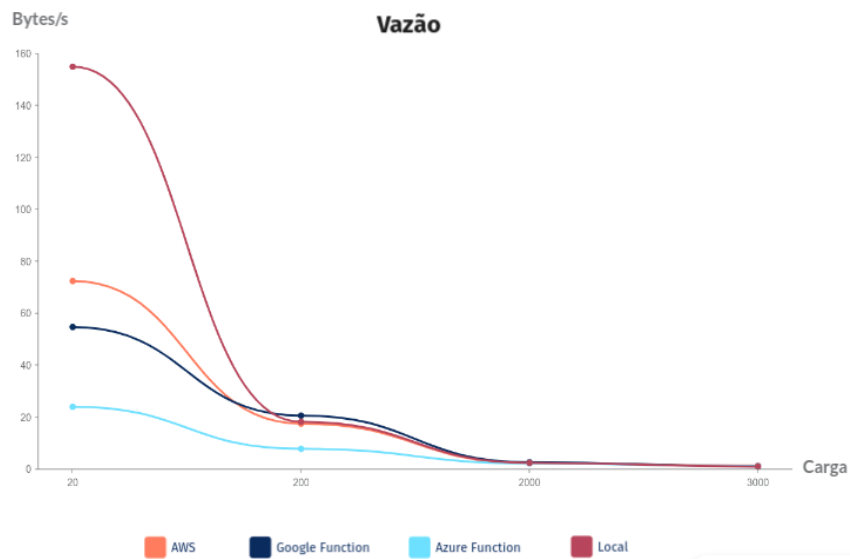


Figura 5.8: Vazão para 200 requisições HTTP/s.

5.2.3 Função que Retorna Métricas de Artigos

No Código 5.2, é apresentada a implementação em JavaScript usada para desenvolver a função que retorna métricas dos artigos recebidos. Em resumo, a função recebe uma lista de artigos acadêmicos, agrupa-os pelo ano de publicação, calcula as palavras mais frequentes em cada um dos anos e, em seguida, retorna uma lista de anos com suas respectivas palavras. Essa é uma métrica importante a ser usada no *framework* TASI, para se ter uma visão geral de quais temas ou assuntos são os mais buscados no *framework*. Essa função foi escolhida para ser implementada via FaaS devido à grande carga de dados que ela pode processar, aproveitando assim a principal característica proporcionada pelo FaaS: a elasticidade [3].

Listing 5.2: Função que retorna métricas dos artigos (palavras mais frequentes/ano)

```
const postMetricsPapersLocal = async (req, res) => {
  const {data} = req.body
  const stopWords = ["the", "a", "an", /* ... lista de palavras
    omitidas ... */ "is "]
  const groupedByYear = {}
  const tokenizer = new natural.WordTokenizer()
  data.forEach(item => {
    const year = item.publication_date.split('-')[0];
    if (!groupedByYear[year])
      groupedByYear[year] = []
    const texts = item.abstract
    const tokens = tokenizer.tokenize(texts)
    tokens.forEach(token => {
      const word = token.toLowerCase()
      if (!stopWords.includes(word) && !groupedByYear[year].
        includes(word)) {
        groupedByYear[year].push(word)
      }
    })
  })
  let year
  const wordCountMap = {}
  const resultMetrics = []
  for (const yearKey of Object.keys(groupedByYear)) {
    year = yearKey
    groupedByYear[year].forEach((word) => {
      wordCountMap[word] = !wordCountMap[word] ? 1 :
        wordCountMap[word] + 1;
    })
    const sortedWords = Object.keys(wordCountMap).sort(
      (a, b) => wordCountMap[b] - wordCountMap[a]
    )
    const topWords = sortedWords.slice(0, 15)
    resultMetrics.push({ [year]: topWords })
  }
  return res.status(200).json(resultMetrics)
}
```

Esse trecho 5.2, em específico, representa a função local implementada no *framework*

TASI. Porém, esse mesmo código foi usado para implantar as funções no AWS Lambda 3.3.1, Azure Function 3.3.2 e Google Function 3.3.3.

Para avaliar o desempenho e as limitações desta função implementada no *framework* foram realizados dois cenários de testes, os quais serão descritos em detalhes.

Cenário 1 - Testes de estresse para 20 requisições HTTP/s

Neste cenário de testes foram feitas 20 requisições HTTP/s simultâneas em cada uma das funções presentes no *framework* TASI. Para isso, foram utilizadas em cada uma das requisições concorrentes cargas de dados de 20, 200 e 2000.

Na Figura 5.9 é possível observar que os tempos de conexão com os servidores são parecidos para cargas entre 20 e 200 dados. Porém, no pior caso em que 2000 dados foram enviados aos servidores, o Google Function, obteve o pior desempenho, atingindo aproximadamente 160 milissegundos.

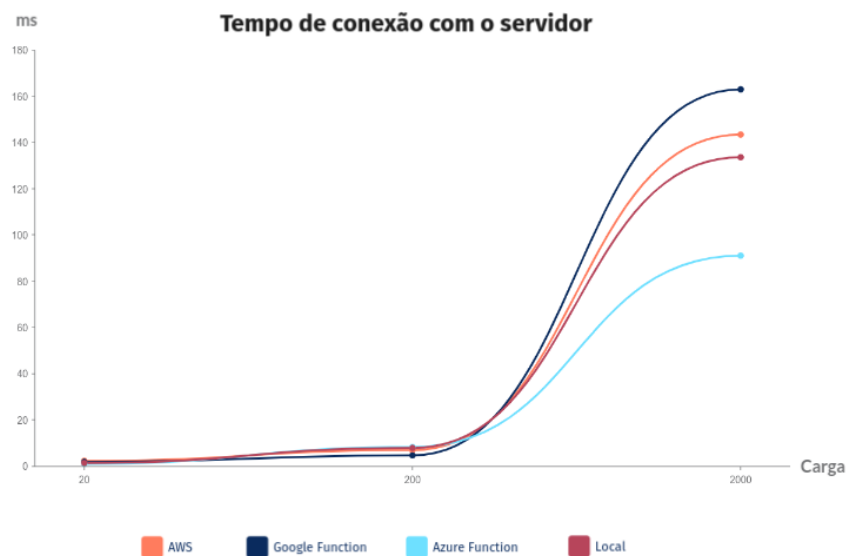


Figura 5.9: Tempo médio de conexão com o servidor para 20 requisições HTTP/s.

Nas Figuras 5.10, 5.11 a função local obteve um menor tempo para ambas as métricas em todas as cargas feitas neste teste. Com isso fica claro que para uma carga de dados baixa, e também um número baixo de requisições feitas, a adoção de FaaS não se faz necessário para casos desse tipo.

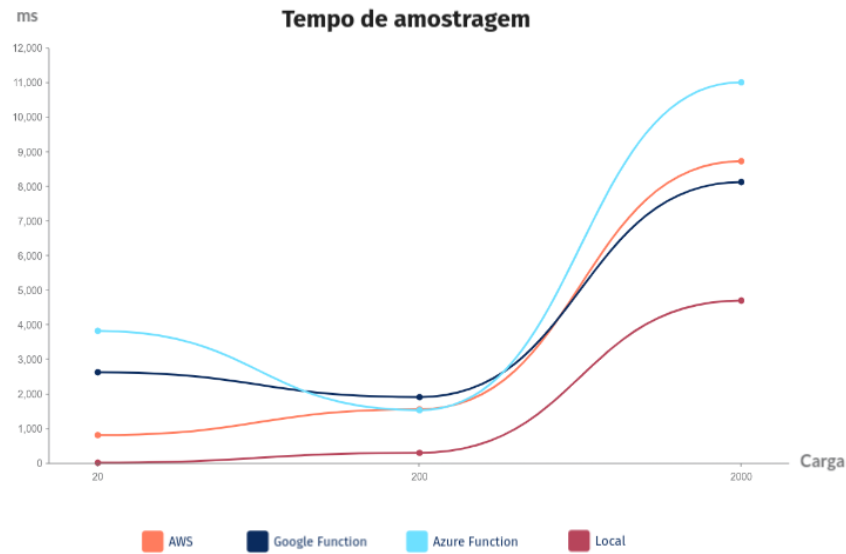


Figura 5.10: Tempo de amostragem para 20 requisições HTTP/s.

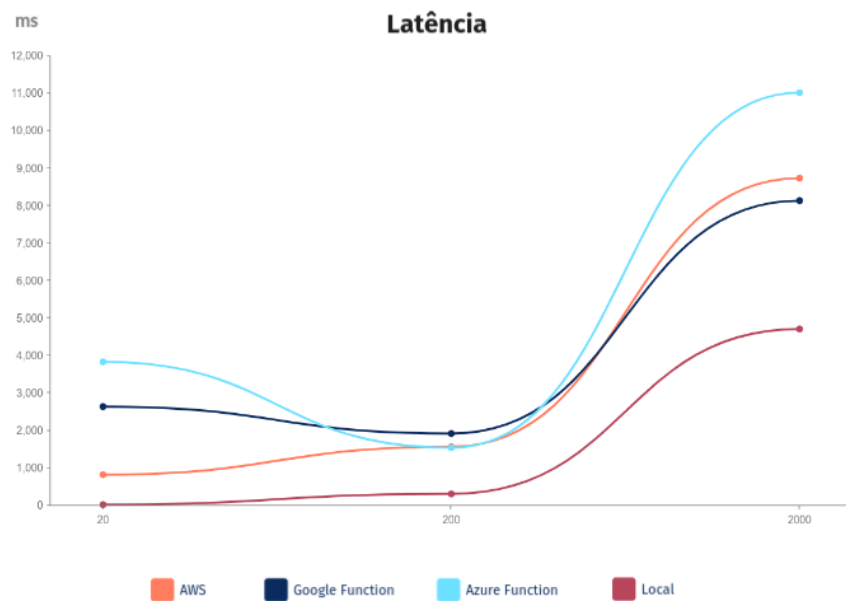


Figura 5.11: Tempo de amostragem para 20 requisições HTTP/s.

Na Figura 5.12 observa-se que a vazão em todas as cargas realizadas é maior na função local. Apenas em uma carga maior para 2000 dados que a abordagem FaaS consegue equiparar a função local.

Com esses dados é possível mais uma vez notar que para esses níveis de carga e estresse a abordagem orientada a FaaS possui o pior desempenho, como já dito, o *overhead* associado a sua invocação é perceptível para cargas baixas e poucas requisições feitas a uma função simples, sendo assim, para essa abordagem a melhor opção é a função local.

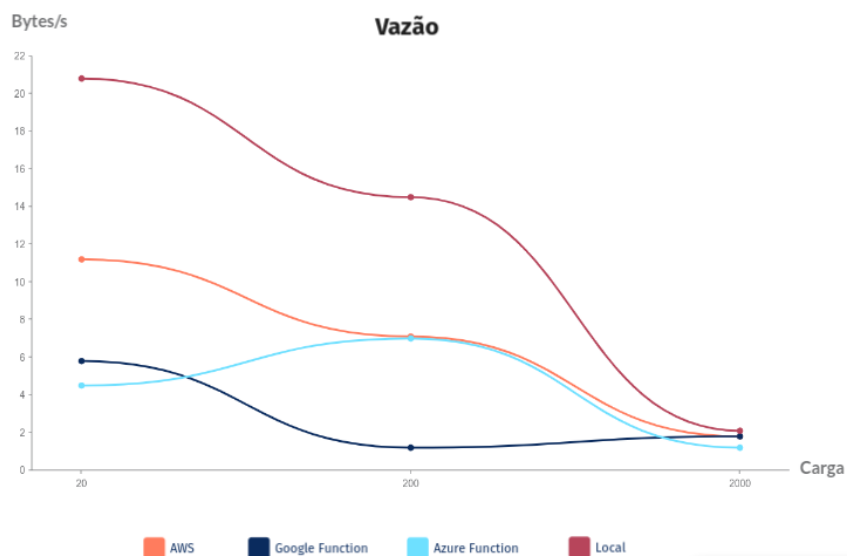


Figura 5.12: Vazão para 200 requisições HTTP/s.

Cenário 2 - Testes de estresse no servidor para 200 requisições HTTP/s

Neste cenário foram feitas 200 requisições HTTP/s simultâneas em cada uma das funções. E a carga de dados utilizada em cada uma das requisições foi de 20, 200, 2000 e 3000 dados enviados. Observa-se na Figura 5.13 o tempo de conexão médio com os respectivos servidores. Para cargas de dados acima de 3000 o função local obteve o pior tempo, enquanto as abordagens orientadas a FaaS tiveram tempos similares.

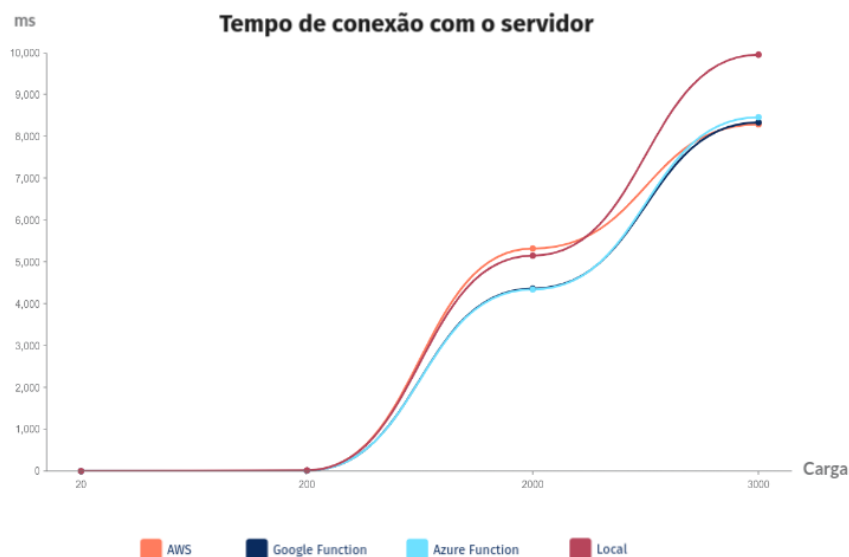


Figura 5.13: Tempo médio de conexão com o servidor para 200 requisições HTTP/s.

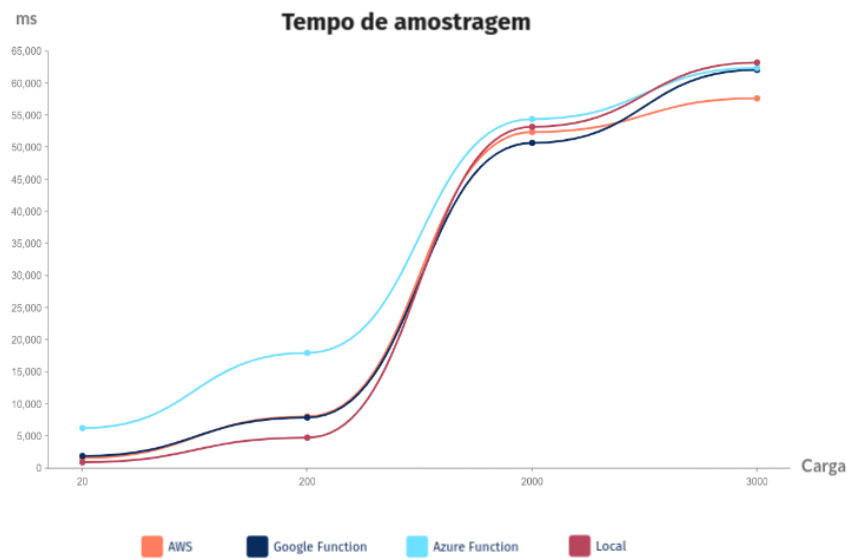


Figura 5.14: Tempo de amostragem para 200 requisições HTTP/s.

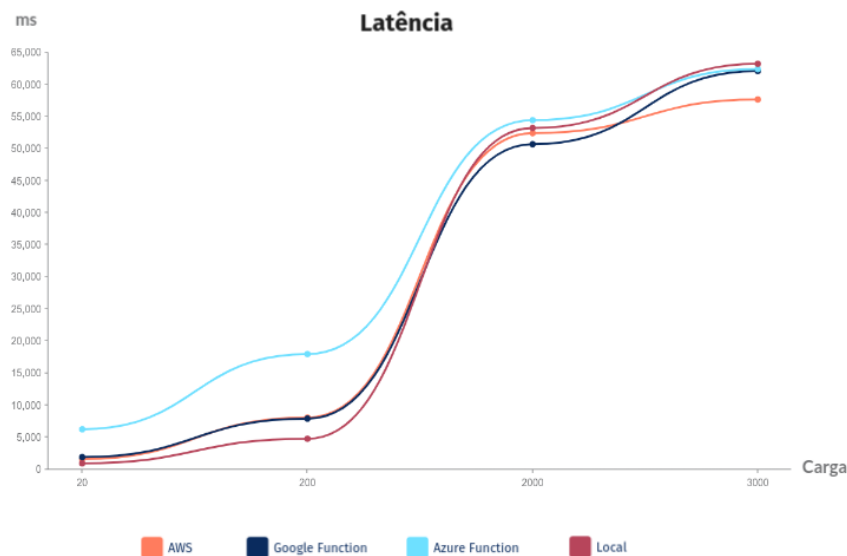


Figura 5.15: Tempo de amostragem para 200 requisições HTTP/s.

Nas Figuras 5.14, 5.15 a função local obteve os menores tempos até uma carga de 200 dados enviados, a partir de cargas de dados superiores, a função local obteve o pior tempo em comparação às abordagens orientadas a FaaS.

Na Figura 5.16 é possível notar uma tendência na qual, à medida que a carga de dados enviadas aos servidores aumenta, a função local obtém uma vazão muito baixa obtendo níveis piores que o das funções orientadas a FaaS.

Com isso é bastante claro que para grandes cargas de dados e estresse no servidor da aplicação, funções orientadas a FaaS possuem um melhor desempenho e são adequadas

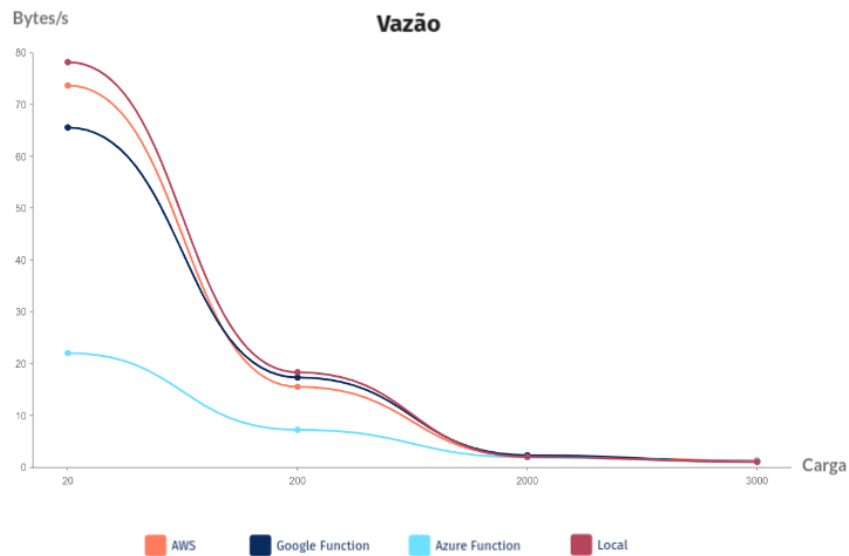


Figura 5.16: Vazão para 200 requisições HTTP/s.

a essas situações, devido a uma de suas principais características mencionadas anteriormente, a elasticidade.

5.3 Considerações Finais

Diante dos resultados apresentados nota-se quão importante é a elasticidade fornecida pela abordagem orientada a FaaS, e as consequências desta característica em um sistema que poderá ser submetido a níveis altos de requisições e estresse. Por outro lado, em cargas menores e com menos requisições feitas ao sistema, as funções locais apresentam um desempenho melhor de acordo com as métricas usadas neste trabalho.

Assim sendo, nota-se que o uso do modelo de serviço Function as a Service (FaaS) não é adequado para qualquer carga de trabalho, e nem para qualquer aplicação. É necessário que o usuário conheça as vantagens e as limitações deste modelo de serviço, assim como a aplicação a ser desenvolvida, pois só a partir deste conhecimento será possível tomar a melhor decisão diante da gama de tipos de serviços ofertada pelos provedores de nuvem.

Capítulo 6

Conclusão

Ao final deste trabalho conclui-se que o *framework* TASI, um *software* que se destaca pela facilitação do levantamento do estado da arte em bases acadêmicas, foi revisado a partir de uma visão abrangente, destacando a evolução da computação e a necessidade de ajustar aplicações para se alinharem às demandas contemporâneas. Surge, a partir disso, a importância das funções como serviço (FaaS), que se apresentam como uma solução eficaz para as limitações de escalabilidade associadas às aplicações monolíticas (*framework* TASI), enfatizando sua capacidade de proporcionar elasticidade rápida e automática em resposta às variações na demanda.

Pode-se concluir que a escolha estratégica da linguagem de programação JavaScript, em conjunto com o *software* NodeJS, foi acertada, observando as vantagens e características que ambos oferecem, como a popularidade, simplicidade, velocidade e versatilidade do JavaScript. O NodeJS oferece execução assíncrona e facilidade no desenvolvimento de serviços de rede, além de permitir o uso do JavaScript fora do navegador web, neste caso, possibilitando o uso no desenvolvimento do *backend* do sistema.

É importante salientar também que a alteração do *framework*, inicialmente na forma monolítica e, posteriormente, na versão aprimorada baseada em FaaS, foi bem-sucedida. Esse avanço arquitetônico superou os desafios de escalabilidade que o sistema possuía e adicionou ao sistema a elasticidade oferecida por FaaS, destacando-se em situações de alta demanda, como pode ser visto no Capítulo 5.

Para dar continuidade a este trabalho, seria de extrema importância a implementação de uma página *web* que apresentasse os resultados obtidos na função descrita na Seção 5.2.3. Isso facilitaria a visualização das métricas retornadas pela função para o usuário final.

Outra possibilidade que pode enriquecer significativamente este trabalho é a realização de uma comparação entre escalabilidade e desempenho. Explorar a relação entre a capacidade de escalabilidade do sistema e o impacto do seu desempenho pode fornecer

métricas valiosas sobre como a arquitetura e as implementações afetam o comportamento do sistema em diferentes condições de carga. Isso permitiria não apenas avaliar a capacidade do sistema de lidar com aumentos de demanda, mas também entender como essas variações afetam seu desempenho em termos de tempo de resposta, eficiência e utilização de recursos (CPU, memória). Dessa forma, será possível fornecer uma análise mais abrangente e informada sobre a eficácia do sistema em ambientes dinâmicos e sob diferentes níveis de carga.

Referências

- [1] Carvalho, Leonardo R. de e Aletéia P. F. Araújo: *Framework node2faas: Automatic nodejs application converter for function as a service*. CLOSER 2019: Proceedings of the 9th International Conference on Cloud Computing and Services Science, página 55, 2019. ix, 17, 20, 23
- [2] Mell, Peter e Timothy Grance: *The nist definition of cloud computing*. NIST, páginas 6–7, 2011. 1, 4, 5, 6, 7
- [3] Li, Yongkang; Lin, Yanying; Wang Yang; Ye Kejiang; e Chengzhong Xu: *Serverless computing: State-of-the-art, challenges and opportunities*. IEEE Transactions on Services Computing, páginas 3–5, 2023. 1, 15, 33, 38, 39
- [4] Giordanelli, Raffaele e Carlo Mastroianni: *The cloud computing paradigm: Characteristics, opportunities, and research issues*. Istituto de Calcolo e Reti ad Alte Prestazioni (ICAR), página 5, 2010. 3
- [5] Buyya, Rajkumar; Yeo, Chee Shin; Venugopal Srikumar; Broberg James; Brandic Ivona: *Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility*. Future Generation Computer Systems, páginas 519–616, 2009. 3, 4
- [6] Duan, Tucong; Fu, Guohua; Zhou Nianjun; Sun Xiaobing; Nanjangud C. Narendra; Hu Bo: *Everything as a service (xaas) on the cloud: Origins, current and future trends*. IEEE 8th International Conference on Cloud Computing, páginas 621–628, 2015. 7
- [7] Gartner: *Gartner*, n.d. <https://www.gartner.com.br/pt-br>, Acessado em: 05 de dezembro de 2023. 8
- [8] Wirfs-Brock, Allen e Brendan Eich: *Javascript: The first 20 years*. Proceedings of the ACM on Programming Languages, páginas 2–4, 2020. 9, 25
- [9] Amazon Web Services: *AWS Documentation*, 2023. https://docs.aws.amazon.com/?nc2=h_q1_doc_do, acessado em 01/12/2023. 9
- [10] Amazon Web Services: *AWS Global Infrastructure*, 2023. <https://aws.amazon.com/about-aws/global-infrastructure/>, acessado em 01/12/2023. 9
- [11] Levy, Steven: *In the Plex: How Google Thinks, Works, and Shapes Our Lives*. Simon & Schuster, 2011. ISBN: 9781416596585. 9

- [12] Google Cloud: *Google cloud platform - produtos em destaque*, 2023. <https://cloud.google.com/products?hl=pt-br#featured-products/>, Acessado em 01/12/2023. 9
- [13] Google Cloud: *Google cloud - infraestrutura global*, 2023. <https://cloud.google.com/about/locations>, Acessado em 01/12/2023. 10
- [14] Microsoft: *Microsoft azure - portal de serviços*, 2023. <https://portal.azure.com/#allservices/category/All>, Acessado em 01/12/2023. 11
- [15] Microsoft: *Microsoft azure - mapa de datacenters*, 2023. <https://www.microsoft.com/en-us/cloud-platform/global-datacenters>, Acessado em 01/12/2023. 11
- [16] Al-Roomi, May; Al-Ebrahim, Shaikha; Buqrais Sabika; e Ahmad; Imtiaz: *Cloud computing pricing models: a survey*. International Journal of Grid and Distributed Computing, páginas 93–106, 2013. 12
- [17] Weinhardt, Christof; Anandasivam, Arun; Blau Benjamin; Borissov Nikolay; Meinel Thomas; Michalk Wibke; e Jochen Stöber: *Cloud computing - a classification, business models and research directions*. Business Information Systems Engineering, páginas 391–399, 2009. 12, 13
- [18] Shahradd, Mohammad; Balkind, Jonanthan; Wentzla David: *Architectural implications of function-as-a-service computing*. Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, páginas 1063–1075, 2019. 14
- [19] Bandeira, Wallif; Rebouças, Carvalho; Araujo Aleiteia: *Análise do custo-benefício de funções como serviço e infraestrutura como serviço*. ANAIS DA IV ESCOLA REGIONAL DE ALTO DESEMPENHO DO CENTRO-OESTE, páginas 38–40, 2021. 14
- [20] Amazon Web Services: *Aws lambda - amazon web services (aws)*, 2023. <https://aws.amazon.com/lambda/>, Acessado em 01/12/2023. 15
- [21] Google Cloud Platform: *Google cloud functions - google cloud platform (gcp)*, 2023. <https://cloud.google.com/functions>, Acessado em 01/12/2023. 15
- [22] Microsoft Azure: *Azure functions - microsoft azure*, 2023. <https://azure.microsoft.com/services/functions/>, Acessado em 01/12/2023. 15
- [23] Kounev, Samuel; Herbst, Nikolas; Abad Cristina; Iosup Alexandru; Foster Ian; Shenoy Prashant; Rana Omer; e Andrey Chien: *Serverless computing: What it is, and what it is not?* página 7, 2023. 15
- [24] Forbes: *The cloud is still a multibillion-dollar opportunity. here's why*. Forbes, 2023. 16
- [25] Amazon Web Services: *Amazon s3 - amazon web services (aws)*, 2023. <https://aws.amazon.com/s3/>, Acessado em 01/12/2023. 17

- [26] Roberts, Mike; Chapin, John: *What is serverless?* O'Reilly, 2017. 24
- [27] Overflow, Stack: *Programming, scripting, and markup languages*, 2022. <https://survey.stackoverflow.co/2022#programming-scripting-and-markup-languages>, Fonte: Stack Overflow, acessado em 17 de novembro de 2023. 25, 26
- [28] Shah, Hezbullah; Tariq Soomro: *Node.js challenges in implementation*. Global Journal of Computer Science and Technology, 2017. 25, 30
- [29] Association for Computing Machinery: *Acm - association for computing machinery*, 2023. <https://www.acm.org/>, Acessado em 19 de novembro de 2023. 25, 30
- [30] arxiv, 2023. <https://info.arxiv.org/about/index.html>, Acessado em 19 de novembro de 2023. 25, 30
- [31] Cold Spring Harbor Laboratory: *About biorxiv*, 2023. <https://www.biorxiv.org/about-biorxiv>, Acessado em 19 de novembro de 2023. 25, 30
- [32] IEEE: *About ieee*, 2023. <https://www.ieee.org/about/>, Acessado em 19 de novembro de 2023. 25, 30
- [33] medRxiv: *About medrxiv*, 2023. <https://www.medrxiv.org/content/about-medrxiv>, Acessado em 19 de novembro de 2023. 25, 30
- [34] National Center for Biotechnology Information (NCBI): *Pubmed*, 2023. <https://pubmed.ncbi.nlm.nih.gov/>, Acessado em 19 de novembro de 2023. 25, 30
- [35] Elsevier: *O que é a visualização do scopus?*, 2023. https://service.elsevier.com/app/answers/detail/a_id/15534/supporthub/scopus/#tipsw, Acessado em 19 de novembro de 2023. 25, 30
- [36] Carvalho, Leonardo R. de e Aletéia P. F. Araújo: *Framework node2faas: Automatic nodejs application converter for function as a service*. CLOSER 2019: Proceedings of the 9th International Conference on Cloud Computing and Services Science, página 65, 2019. 26
- [37] Liang, Li; Zhu, Ligu; Shang Wenqian; Feng Dongyu; Xiao Zida: *Express supervision system based on nodejs and mongodb*. IEEE/ACIS 16th International Conference on Computer and Information Science (ICIS), página 2, 2017. 26
- [38] NPM: *About npm*, 2023. <https://docs.npmjs.com/about-npm>, Fonte: NPM, acessado em 17 de novembro de 2023. 26
- [39] Express.js: *Express framework web rápido, flexível e minimalista para node.js*, 2023. <https://expressjs.com/pt-br/>, Acessado em: 02 de dezembro de 2023. 26
- [40] Facebook e React community: *The library for web and native user interfaces*, 2023. <https://react.dev/>, Acessado em 19 de novembro de 2023. 29
- [41] Group, PostgreSQL Global Development: *What is postgresql?*, 2023. <https://www.postgresql.org/about/>, Acessado em 19 de novembro de 2023. 29

- [42] Vazhenin, Denis; Ishikawa, Satoru; Satake Satoru; Klyuev Vitaly: *A document retrieval framework for scientific publications*. IEEE Region 8 International Conference on Computational Technologies in Electrical and Electronics Engineering, 2008. 30
- [43] Osborne, Francesco; Motta, Enrico; Mulholland Paul: *Exploring scholarly data with rexplore*. The Semantic Web – ISWC 2013, 2013. 30, 31
- [44] Bakri Bashir, Mohammed; Shafie Abd Latiff, Muhammad; Muhammad Abdulhamid Shafi'i; Tek Loon Cheah: *Grid-based search technique for massive academic publications*. 2014 Third ICT International Student Project Conference (ICT-ISPC), 2014. 31
- [45] Fonseca Guilarte, Orlando; Diniz Junqueira Barbosa, Simone; Pesco Sinesio: *Rel-path: an interactive tool to visualize branches of studies and quantify the expertise of authors by citation paths*. Scientometrics (2021), 2021. 31
- [46] Shen, Zhihong; Wu, Chieh Han; Ma Li; Chen Chien Pang; Wang Kuansan: *Sciconceptminer: A system for large-scale scientific concept discovery*. Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing: System Demonstrations, 2021. 31
- [47] Microsoft: *Microsoft academic graph*, s.d. <https://www.microsoft.com/en-us/research/project/microsoft-academic-graph/>, Acessado em: 02 de dezembro de 2023. 31
- [48] JMeter, Apache: *Apache jmeter*. <https://jmeter.apache.org/>, Acessado em: 04 de dezembro de 2023. 33
- [49] Molyneaux, Ian: *The art of application performance testing*. O'Reilly, página 22, 2014. 33, 34
- [50] Molyneaux, Ian: *The art of application performance testing*. O'Reilly, página 29, 2014. 33, 37