UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
Campus de São José do Rio Preto

Valter Canhizares Filho

# Desvendando As Nuvens: Uma Análise Comparativa De Desempenho De Aplicações *Serverless* E *Containers* Em Provedores De Computação Em Nuvem

São José do Rio Preto
2024

Valter Canhizares Filho

# Desvendando As Nuvens: Uma Análise Comparativa De Desempenho De Aplicações *Serverless* E *Containers* Em Provedores De Computação Em Nuvem

Dissertação apresentada como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação, junto ao Programa de Pós-Graduação em Ciência da Computação (PPGCC), do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista “Júlio de Mesquita Filho”, Campus de São José do Rio Preto.

Orientadora: Prof.ª. Dr.ª. Renata Spolon Lobato

São José do Rio Preto
2024

C222d

Canhizares Filho, Valter
Desvendando As Nuvens: Uma Análise Comparativa De Desempenho De Aplicações Serverless E Containers Em Provedores De Computação Em Nuvem / Valter Canhizares Filho. -- São José do Rio Preto, 2024
100 p.

Dissertação (mestrado) - Universidade Estadual Paulista (UNESP), Instituto de Biociências Letras e Ciências Exatas, São José do Rio Preto

1. Computação em Nuvem. 2. Computação sem Servidor. 3. Containers. I. Título.

**Sistema de geração automática de fichas catalográficas da Unesp. Biblioteca da Universidade Estadual Paulista (UNESP), Instituto de Biociências Letras e Ciências Exatas, São José do Rio Preto. Dados fornecidos pelo autor(a).**

**Essa ficha não pode ser modificada.**

Valter Canhizares Filho

# Desvendando As Nuvens: Uma Análise Comparativa De Desempenho De Aplicações *Serverless* E *Containers* Em Provedores De Computação Em Nuvem

Dissertação apresentada como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação, junto ao Programa de Pós-Graduação em Ciência da Computação (PPGCC), do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista “Júlio de Mesquita Filho”, Campus de São José do Rio Preto.

Orientadora: Prof.ª. Dr.ª. Renata Spolon Lobato

Banca Examinadora

Prof.ª. Dr.ª. Renata Spolon Lobato
UNESP – Câmpus de São José do Rio Preto
Orientadora

Prof. Dr. Rodrigo Capobianco Guido
UNESP – Câmpus de São José do Rio Preto

Prof. Dr. Márcio Andrey Teixeira
IFSP – Câmpus de Catanduva

São José do Rio Preto
28 de junho de 2024

# AGRADECIMENTOS

O Mestrado não é só uma realização acadêmica para mim, é também a realização de um sonho. Sempre tive o grande sonho de poder estudar em uma das mais renomadas instituições de ensino deste país, hoje posso dizer que subi um degrau da escada. Só quem vive essa experiência consegue dimensionar a dificuldade que é subir esse degrau, a dificuldade que é furar a bolha do conhecimento e continuar.

Em primeiro lugar, agradeço aos meus pais Valter e Rejane por me proporcionarem o privilégio da boa educação. O suor, o sacrifício e tudo que deixaram de fazer por vocês para fazer por mim, para que eu pudesse ter esse privilégio. Devo tudo isso a vocês.

A minha namorada, Maria Rita, que nunca mediu esforços para me ajudar e me motivar, me ouvir desabafar e não deixar que eu desanimasse. Uma das minhas maiores fontes de motivação para que o nosso futuro seja ainda mais brilhante.

À orientadora, Professora e Mentora, Renata, que foi fundamental para o bom caminho seguido por este trabalho. Pelas inúmeras reuniões e pela infinita paciência comigo. Tive um caminho conturbado durante esse processo e, mesmo assim não desistiu de mim. Obrigado pelos puxões de orelha e por toda motivação.

Aos meus grandes amigos de trabalho, Rafael Elói, Camila Brandão e Camila Okado, que acompanharam de perto toda essa jornada e me motivaram a completar meu objetivo. Estendo este agradecimento a todos os amigos que, direta ou indiretamente, me ajudaram a obter sucesso nesta etapa, que muitas vezes mostraram que a minha vitória também seria a deles.

A todo o corpo docente e demais funcionários da Universidade Estadual Paulista, pelo trabalho primoroso e cordial que todos prestam à sociedade.

Por último, mas não menos importante, uma pessoa que considero como uma segunda mãe, Teresa Cristina Castilho Gorayeb, que foi a primeira pessoa a me incentivar a começar a carreira de professor e de pesquisador.

# RESUMO

Nos últimos anos houve um aumento considerável no interesse comercial e acadêmico na computação em nuvem, alavancando também um novo paradigma da computação distribuída, a computação sem servidor. Essa abstração oferece a desenvolvedores uma maior agilidade para criação de uma aplicação, além de proporcionar redução de custos, escalabilidade e significante diminuição de tempo para um produto atingir seus clientes. O desenvolvedor de uma aplicação sem servidor cria e executa suas aplicações sem embargo de qualquer configuração e manutenção em servidores, máquinas virtuais e sistemas operacionais. A computação sem servidor oferece computação distribuída com custo granular inerente ao uso e oferece aplicações em diferentes escopos como desenvolvimento *web*, APIs, *chatbots*, processamento de dados, *Internet* das Coisas, redes de computadores, entre outros. Entretanto oferece desafios de desempenho, arranque a frio de funções, segurança, precificação e travamentos de provedor em nuvem. O desenvolvimento do trabalho ocorreu pela apresentação do estado da arte da computação sem servidor, comparação entre provedores de computação em nuvem que oferecem a computação sem servidor, e aplicação em caso prático em *backend* de uma aplicação *web*. Para a mesma aplicação, resultados indicam que o tempo de execução pode diferir de duas a quatro vezes dependendo do provedor. Em relação ao custo monetário das execuções, foi demonstrado uma diferença de até 25% entre arquiteturas *ARM* e x86, diferença de até 68% entre regiões diferentes e, mais de 48% entre os modelos de *Functions-As-a-Service* e *Containers-As-a-Service*.

**Palavras–chave:** Computação sem Servidor. Computação em Nuvem. Desafios sem Servidor. Benefícios sem Servidor.

# ABSTRACT

In recent years, there has been a considerable increase in commercial and academic interest in cloud computing, which has also boosted a new paradigm of distributed computing: the serverless computing. This abstraction provides developers with greater agility in creating applications, along with cost reduction, scalability, and significant time reduction for a product to reach its customers. A serverless application developer can create and run applications without any configuration and maintenance of servers, virtual machines, or operating systems. Serverless computing offers distributed computing with granular cost inherent to usage and supports applications in various scopes such as web development, APIs, chatbots, data processing, Internet of Things, computer networks, among others. However, it presents challenges in performance, cold start of functions, security, pricing, and cloud provider lock-in. The work developed involves presenting the state of the art in serverless computing, comparing cloud providers that offer serverless computing, and applying it in a practical case in the backend of a web application. For the same application, results indicate that the execution time can differ by two to four times depending on the provider. Regarding the monetary cost of executions, a difference of up to 25% was demonstrated between ARM and x86 architectures, a difference of up to 68% between different regions, and more than 48% between Functions-As-a-Service and Containers- As-a-Service.



**Keywords:** Serverless computing. Cloud computing. Serverless benefits. Serverless challenges.

# LISTA DE ILUSTRAÇÕES


Gráfico 1 – Gráfico do número de artigos obtidos baseado na palavra-chave "*serverless*". 16
Figura 1 – Modelos de Computação em Nuvem e alguns exemplos 22
Figura 2 – Modelo de processamento *FaaS* 26
Figura 3 – Arquitetura da computação sem servidor 27
Figura 4 – Exemplo de funcionamento do *AWS Lambda* em processamento de arquivos 28
Figura 5 – Exemplo de funcionamento do *AWS Lambda* em *IoT* 28
Figura 6 – Exemplo de funcionamento do *Google Functions* em IA 29
Figura 7 – Computação tradicional *versus* Computação em Nuvem *versus* Computação sem Servidor 31
Figura 8 – O *cold start* atrasa a execução do código 33
Figura 9 – Relação entre a taxa de transferência por segundo e o número de execuções concorrentes 38
Figura 10 - Sobrecarga para a simultaneidade da *CPU* 39
Figura 11 – Exemplo de computação em monólitos 48
Figura 12 – Exemplo de computação em Microsserviços 49
Figura 13 – Exemplo de computação *serverless* 50
Figura 14 – Desenvolvimento de Microsserviços e Funções 53
Figura 15 – Arquitetura de um serviço *web* usando *Node.JS* 55
Figura 16 - Definição do *handler* 59
Figura 17 – Encontrando o melhor *setup* para execução *serverless* 60
Figura 18 – *Report* do *CloudWatch* em função na região *us-east-1* 61
Figura 19 – *Report* do *CloudWatch* em função na região *sa-east-1* 62
Figura 20 – Diagrama da estrutura da aplicação na plataforma *AWS Lambda* 63
Figura 21 – Custo em dólares de 10 milhões de requisições *serverless* (*FaaS*) e *containers* (*CaaS*) no provedor *AWS Lambda* 64
Figura 22 - Período de reciclagem de instância e tempos de execução em segundos da plataforma *AWS Lambda* 66

Figura 23 – Arquitetura da aplicação em *Azure Functions* 71
Figura 24 – Precificação e comparação *CaaS* e *FaaS* no *Azure* 73
Figura 25 - Período de reciclagem de instância e tempos de execução em segundos da plataforma *Azure Functions* 75
Figura 26 - Exemplo da implantação da função na 2ª versão do *Google Functions* 76
Figura 27 - Arquitetura da aplicação na plataforma *Google Functions* 78
Figura 28 – Precificação e comparação *CaaS* e *FaaS* no *Google Functions* 79
Figura 29 – Período de reciclagem de instância e tempos de execução em segundos da plataforma *Google Functions* 80
Figura 30 – Custo de 10 milhões de execuções em três provedores de computação em nuvem 83
Figura 31 – Custo de provedores *FaaS* comparados por região 84
Figura 32 – Custo de provedores *CaaS* comparados por região 85
Figura 33 – Custo *FaaS* contra *CaaS* em três provedoras 86
Figura 34 – Relação entre custo $ na região *us-east-1* 86
Figura 35 – Relação entre custo $ na região *sa-east-1* 87
Figura 36 – Comparativo entre o tempo de execução da função *serverless* nos três provedores 88

# LISTA DE TABELAS

Tabela 1 – Número de artigos baseado na palavra “*serverless*” 16
Tabela 2 – Número de artigos baseados nas palavras-chaves “*cloud computing*” e “*Function-As-a-Service*” 17
Tabela 3 – Comparação entre *serverless* e computação tradicional 35
Tabela 4 – Lista dos três principais provedores de computação sem servidor – *FaaS* 36
Tabela 5 – Comparação entre custo das plataformas *FaaS* 37
Tabela 6 – Lista de plataformas *opensource* 40
Tabela 7 – Comparação entre plataformas *serverless* 43
Tabela 8 – Comparação entre desenvolvimento de *software on-premises*, em nuvem e *serverless* 44
Tabela 9 – Runtimes do *Node.JS* no *AWS Lambda* 58
Tabela 10 - Suporte a Sistema Operacional *Azure Functions* 68
Tabela 11 - Planos e a duração do tempo limite de funções 69
Tabela 12 – Comportamento do *cold start* em relação ao Plano 69
Tabela 13 – *Runtimes* do *Node.JS* no *Azure Functions* 70
Tabela 14 – Tabela de comparação de versões do *Google Functions* 76
Tabela 15 – *Runtimes* do *Node.JS* no *Google Functions* 77
Tabela 16 – Custo de 10 milhões de requisições para cada serviço, do mais barato para o mais caro 81

# LISTA DE ABREVIATURAS E SIGLAS

| | |
|---|---|
| ***AWS*** | Amazon Web Services |
| **IBM** | International Business Machines |
| **GCP** | Google Cloud Platform |
| **IEEE** | Institute of Electrical and Electronics Engineers |
| **CAPEX** | Capital Expenditure |
| **OPEX** | Operational Expenditure |
| **SaaS** | Software as a Service |
| **PaaS** | Platform as a Service |
| **IaaS** | Infrastructure as a Service |
| ***FaaS*** | Function as a Service |
| **BaaS** | Back-end as a Service |
| **REST** | Representational State Transfer |
| **API** | Application Programming Interfaces |
| **SDK** | Software Development Kit |
| **S3** | Amazon Simple Storage Service |
| **IoT** | Internet of Things |
| **FTTM** | Fast Time do Market |
| **QPS** | Queries Per Second |
| **CPU** | Central Processing Unit |
| **HTTP** | Hypertext Transfer Protocol |
| **IDE** | Integrated Development Environment |
| **CLI** | Command-Line Interface |
| **GUI** | Graphical User Interface |
| **MQTT** | Message Queuing Telemetry Transport |

| | |
|---|---|
| **RPC** | Remote Procedure Call |
| **REST** | Representational State Transfer |
| **SOAP** | Simple Object Access Protocol |
| **MB** | Megabyte |
| **RAM** | Random-Access Memory |
| **ID** | Identification |
| **HTML** | Hypertext Markup Language |
| **CSS** | Cascading Style Sheets |
| **JS** | JavaScript |
| **HTTP** | Hypertext Transfer Protocol |
| **HTTPS** | Hypertext Transfer Protocol Secure |
| **PNG** | Portable Network Graphic |
| **JPG** | Joint Photographic Group |
| **SDK** | Software Development Kit |
| **I/O** | Input/Output |
| **DEVOPS** | Development and Operations |
| **IAM** | Identity and Access Management |
| **ARM** | Advanced RISC Machine |
| **RISC** | Reduced Instruction Set Computer |
| **ASE** | App Service Environment |
| **SKU** | Stock Keeping Unit |
| **ACU** | Azure Compute Unit |
| **SSL** | Secure Sockets Layer |
| **vCPU** | Virtual Central Processing Unit |

## SUMÁRIO

**CAPÍTULO 1 - INTRODUÇÃO ..... 13**
1.1 Motivação E Objetivos ..... 14
1.2 Revisão Da Literatura ..... 15
1.3 Organização Do Texto ..... 17
**CAPÍTULO 2 – COMPUTAÇÃO EM NUVEM E *SERVERLESS* ..... 18**
2.1 Definições Da Computação Em Nuvem ..... 18
2.2 Modelos De Serviços De Computação Em Nuvem ..... 20
2.3 Computação Sem Servidor ..... 23
2.4 Exemplos Do Uso Da Computação Sem Servidor ..... 27
2.5 Vantagens E Desvantagens ..... 29
**CAPÍTULO 3 – PLATAFORMAS *SERVERLESS* ..... 36**
3.1 Plataformas Comerciais ..... 36
3.2 Plataformas *FaaS Opensource* ..... 39
3.3 Comparação Entre Plataformas Disponíveis e o Desenvolvimento Tradicional De *Software* ..... 42
3.4 Microsserviços E Suas Características ..... 46
3.5 Monólitos, Microsserviços, *Containers* E *Serverless* ..... 47
**CAPÍTULO 4 – DESENVOLVIMENTO ..... 54**
4.1 Arquitetura Da Aplicação *Web* ..... 54
4.2 Testes De Desempenho ..... 57
4.2.1 *AWS Lambda* ..... 57
4.2.2 *Azure Functions* ..... 67
4.2.3 *Google Cloud Functions* ..... 75
4.3 Resultados Dos Testes E Comparativos ..... 81
**CAPÍTULO 5 - CONCLUSÃO ..... 89**
**REFERÊNCIAS ..... 92**

# CAPÍTULO 1 - INTRODUÇÃO

Na última década os avanços da computação em nuvem são notáveis. Tais avanços podem ser notados pelo crescente investimento em soluções em nuvem por parte da indústria e estudos realizados pela academia. Segundo (UDUNWA, et al. 2019), o mercado da computação em nuvem chegará em 2025 a um volume de revenda de quase 700 bilhões de dólares, impulsionado por soluções relacionadas a infraestrutura em nuvem como serviço, inteligência artificial e computação sem servidor.

Aplicações em nuvem são oferecidas hoje por múltiplos provedores. São exemplos *Amazon* (*AWS*), *Microsoft* (*Azure*), *Google* (*GCP*), *IBM* (*IBM Cloud*) entre outros que distribuem seu poder computacional para consumidores dispostos a pagar pelo consumo.

(VARGHESE; BUYYA, 2018) destacam em seu trabalho que novas tendências e direções exponenciam a necessidade e popularidade da computação em nuvem. Direções como arquiteturas emergentes, tal qual computação definida por *software*, computação sem servidor serão tendências para as próximas gerações de computação em nuvem. Junto a isso, vale destacar tendências sobre infraestruturas em nuvens heterogêneas.

A computação em nuvem vem cada vez mais sendo adotada pelas empresas para otimizar seus processos, sejam eles disponíveis em infraestrutura como serviço, plataforma como serviço ou até mesmo *software* como serviço. Características como alta disponibilidade, elasticidade e medicação do consumo trazem os holofotes da computação atual para a computação em nuvem.

Os autores (RISTOV, et al. 2021); (BALLA, et al. 2020); (JINDAL, et al. 2021), PFANDZELTER; BERMBACH, 2020) exploram, com sucesso, o emprego da linguagem *Node.JS* na computação sem servidor, bem como o desempenho de aplicações na migração computação monolítica para a computação *serverless*,

Embasando ainda no contexto da pesquisa, (CASSEL et al, 2022) traz em sua taxonomia que uma das linguagens mais utilizadas no contexto *serverless* é *JavaScript* e *containers* como componente mais utilizado para o desenvolvimento. Além disso, a maioria das aplicações *serverless* trabalham na camada de aplicação, utilizando o protocolo *HTTP*. Todas essas três tecnologias estão sendo usadas no trabalho atual.

## 1.1 Motivação e Objetivos

Novos paradigmas da computação vêm emergindo com o passar dos anos e a computação em nuvem surgiu como um marco para a computação moderna. Novas propostas surgem com a finalidade de tornar mais dinâmico e ágil o grande fluxo de informações que trafegam diariamente entre usuários e servidores em rede.

A Internet amplamente distribuída necessita de práticas e modelos escalonáveis, confiáveis e altamente disponíveis, premissas que condizem com a exploração de soluções na nuvem. Diversos autores destacam esse crescimento e necessidade de processar tamanha conjuntura de dados e demandas de aplicações em nuvem crescem ao passar dos últimos anos. Cada vez mais as soluções convergem para a nuvem, e é necessário que progressos sejam feitos para atender a essa crescente demanda.

Como objetivo principal deste trabalho, uma aplicação *web* monolítica em *Node.JS* que atualmente está em um ambiente *on premises* é migrada para um ambiente em nuvem, mais especificamente em um ambiente *serverless* como *FaaS – Function-as-a-Service* e um ambiente de *container* como PaaS – *Platform-as-a-Service*.

Com as revisões bibliográficas foi possível implementar a aplicação *web* e medir seu desempenho e seu custo. O desempenho foi medido com testes de carga incremental e picos de carga por meio da aplicação *Apache JMeter* e o custo foi medido em três provedores de computação sem servidor: *Amazon*, *Google* e *Microsoft*. A mesma aplicação *serverless* foi implementada nos provedores e executadas milhões de vezes, custeadas por créditos cedidos pelas empresas para conta educacional do autor do trabalho.

Como resultado, é possível analisar se a computação sem servidor traz benefícios ao desempenho e custo da aplicação em comparação a mesma aplicação implementada em forma de *containers*, levando em conta os desafios encontrados pela computação sem servidor.

Ao final do trabalho, considerando os resultados obtidos, é elencado o melhor provedor de computação em nuvem para a execução da aplicação em *Node.JS*, com o menor tempo de execução possível e com o melhor preço possível, gerando assim uma melhor economia para uma aplicação que hoje é executada em um monólito.

## 1.2 Revisão da Literatura

Nesta subseção é explicitada a revisão sistemática da literatura atual sobre o tema, levantamento dos dados relacionados e problemáticas inerentes ao assunto principal do trabalho. A subseção mostra a metodologia utilizada, termos utilizados nas buscas e alguns resultados preliminares da bibliografia relacionada ao tema.

O primeiro passo foi realizar uma revisão sistemática da literatura encontrada nos principais veículos de artigos científicos, publicações de jornais e revistas da área da computação nos últimos anos.

No presente método de pesquisa da literatura é necessário reunir diversos estudos que testaram empiricamente hipóteses relacionadas a computação sem servidor. Para isso é necessária a revisão quantitativa das informações encontradas nos veículos de trabalhos relacionados, pesquisa denominada de meta-análise, que está focada em estimativas, relato de resultados quantitativos correlatos e exame dos relacionamentos entre eles (GALVÃO; RICARTE, 2020).

Segundo os mesmos autores, visando integrar a pesquisa qualitativa, a meta-síntese é necessária para integrar a pesquisa, com a finalidade de sintetizar os estudos qualitativos e localizar temas e conceitos para fornecer novas explicações sobre os paradigmas da computação em nuvem e sem servidor.

As bases de dados selecionadas foram revistas como *IEEE Xplore, ACM Digital Library, ScienceDirect*, entre outros.

As buscas nas revistas deram-se pelas palavras-chaves “*cloud computing*” (Computação em Nuvem), “*serverless”* (Sem Servidor) e “*Function-as-a-Service*” (Funções-como-serviço). Foram delimitados para a pesquisa trabalhos no período de seis anos, entre janeiro de 2019 até o mês de maio de 2024.

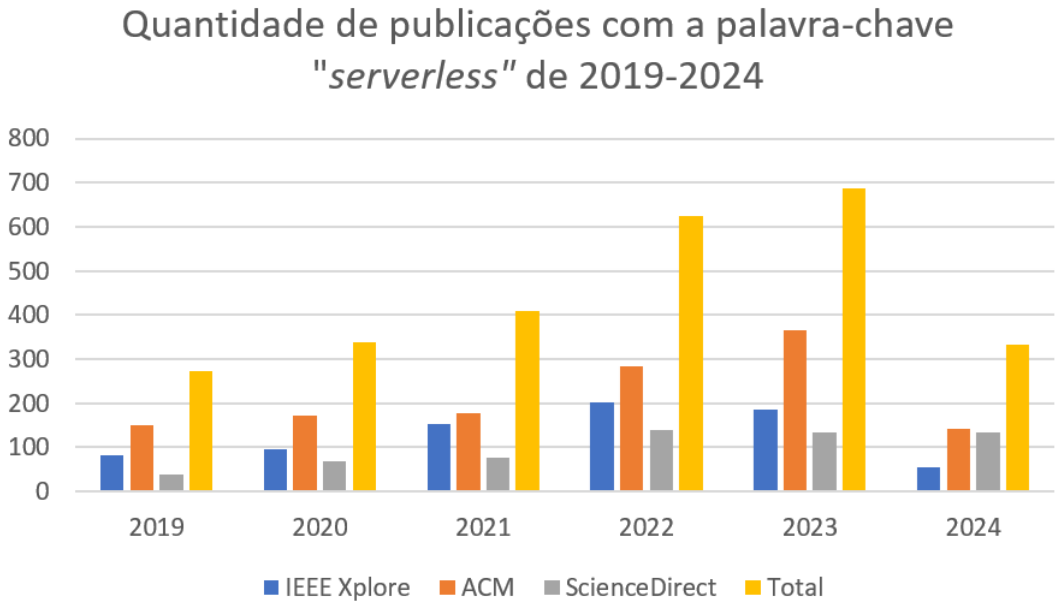
Nos últimos anos houve um aumento significativo de publicações acerca do tema, o que se faz entender que a academia está cada vez mais em busca do fomento a exploração do tema. Na Tabela 1 é representada a quantidade de publicações de trabalhos em revistas baseado na palavra-chave “*serverless*” no período de 2019 a 2024. Em seguida, o Gráfico 1 representa sua evolução.

Tabela 1 – Número de artigos baseado na palavra “*serverless*”

| | 2019 | 2020 | 2021 | 2022 | 2023 | 2024 |
|---|---|---|---|---|---|---|
| ***IEEE Xplore*** | 83 | 97 | 254 | 201 | 259 | 55 |
| ***ACM Digital Library*** | 151 | 172 | 178 | 283 | 416 | 143 |
| ***Science Direct*** | 39 | 69 | 78 | 140 | 147 | 135 |
| **Total** | 273 | 338 | 410 | 624 | 687 | 333 |

Fonte: Desenvolvido pelo autor (2024).

Gráfico 1 – Gráfico do número de artigos obtidos baseado na palavra-chave “*serverless*”




Fonte: Desenvolvido pelo autor (2024).

Após pesquisas nos meios citados, a leitura é realizada para seleção dos artigos mais relevantes e com maior pertinência relacionado ao trabalho. Com base nas evidências encontradas nas leituras é possível elevar pontos importantes e alicerçar novas pesquisas relacionadas ao tema.

As outras palavras-chaves “computação em nuvem” e “*Function-as-a-Service*” estão sumarizadas na Tabela 2, notavelmente endossa a ideia do avanço em pesquisas relacionadas ao tema do atual trabalho.

Tabela 2 – Número de artigos baseados nas palavras-chaves “*cloud computing*” e “*Function-As-a-Service*”

| | Cloud Computing | Function-As-a-Service | | | | |
|---|---|---|---|---|---|---|
| | (2018-2024) | 2020 | 2021 | 2022 | 2023 | 2024 |
| ***IEEE Xplore*** | 52.727 | 27 | 53 | 64 | 82 | 12 |
| ***ACM Digital Library*** | 142.984 | 35.216 | 36.972 | 36.523 | 39.423 | 15.313 |
| ***Science Direct*** | 140.684 | 7.680 | 8.242 | 9.534 | 9.633 | 6.163 |

Fonte: Desenvolvido pelo autor (2024).

## 1.3 Organização do texto

O texto encontra-se organizado da seguinte forma:

No **Capítulo 2** são apresentadas definições e modelos presentes na computação em nuvem, computação *serverless*, exemplo do uso e principais vantagens e desvantagens.

No **Capítulo 3** são apresentadas plataformas comerciais e *opensource* da computação sem servidor, comparação entre plataformas, microsserviços e suas características de interação com *serverless* e, como a evolução da computação monolítica culminou em tecnologias como *containers* e Funções como Serviço.

No **Capítulo 4** é apresentado o desenvolvimento do trabalho, funcionamento da arquitetura da aplicação *web* em *Node.JS*, testes de validação conduzidos em três provedores de computação *serverless* e seus resultados.

No **Capítulo 5** tem-se a conclusão e os trabalhos futuros.

# CAPÍTULO 2 – COMPUTAÇÃO EM NUVEM E *SERVERLESS*

A computação em nuvem é um mercado que está sendo explorado a anos e vem trazendo resultados otimistas para seus consumidores. É sabido que com o passar dos anos houve um aumento na adoção de computação em nuvem pelas empresas de todos os ramos (KHOUN, et al. 2020). O mercado da computação em nuvem inegavelmente cresceu nos últimos anos e tem tendência de crescimento para os próximos. Em 2021 movimentou quase 450 bilhões de dólares em todo mundo e é esperado que até 2026 o movimento seja de 950 milhões de dólares, crescimento composto anual de mais de 16%, segundo (MARKETSANDMARKETS, 2021).

Com a premissa de otimizar os recursos computacionais distribuídos, a computação em nuvem ganha destaque no contexto atual. Pequenas e médias empresas podem utilizar das mesmas tecnologias que grandes empresas usam, contando com a redução de custos e escalabilidade ao utilizar o poder de processamento de máquinas virtuais dos provedores de computação em nuvem (JONAS, et. al. 2019), tudo isso pagando conforme a utilização.

Em resumo, toda infraestrutura de computação em nuvem é um conjunto entre *hardware* e *software* que possibilitam as cinco principais características da computação em nuvem, segundo (MELL; GRANCE, 2011). Essa infraestrutura pode ser observada como o conjunto de um nível físico junto a uma camada de diferentes níveis de abstração. A parte física consiste no ecossistema de *hardware* que uma empresa provedora precisa ter para oferecer seus produtos aos seus clientes, como por exemplo processadores, servidores, memória, armazenamento em discos rígidos ou de estado sólido, componentes de redes como *switches*, roteadores e cabeamento. As camadas de abstração são os *softwares* desenvolvidos com base nos componentes físicos da infraestrutura.

## 2.1 Definições da Computação em Nuvem

Ao utilizar a computação em nuvem, o usuário utiliza recursos distribuídos oriundos de *datacenters* de provedores de computação em nuvem como *Microsoft* e *Amazon*. Empresas provedores oferecem serviço de rápido provisionamento, computação ubíqua e sob demanda com diferentes níveis de configuração do usuário deste modelo. Toda a parte de servidores físicos, redes, armazenamento e aplicações

podem, ou não, ser configuradas pelo usuário, trazendo assim níveis diferentes de abstração da computação em nuvem (MELL; GRANCE, 2011). Os mesmos autores elencam a composição do modelo de computação em nuvem em cinco principais características, três modelos de serviço e quatro modelos de implantação.

As cinco características principais da computação em nuvem são:

- **Serviço sob demanda**: O cliente do provedor de computação em nuvem pode, a qualquer momento, provisionar capacidade computacional como processamento, execução e armazenamento de qualquer dado, sem necessariamente ter interação humana no processo com seu provedor. Empresas provedores oferecem seus serviços de computação em nuvem por meio de interfaces gráficas práticas e usuais, o cliente cria uma conta e consegue utilizar os serviços de nuvem por meio de créditos em sua conta. A qualquer momento é possível aumentar, diminuir, replicar a demanda imposta ao provedor, custeado apenas o tempo que o usuário utilizar.

- **Acesso amplo à rede de computadores**: Os serviços de computação em nuvem são ofertados para qualquer tipo de cliente, de pequeno, médio e grande porte. As capacidades computacionais das provedores ficam à disposição do cliente por meio da *internet* e pode ser acessado por qualquer tipo de plataforma, heterogeneamente, como celulares, computadores portáteis ou estações de trabalho. Geralmente o acesso é feito via aplicativo ou navegador de *internet* que qualquer dispositivo tenha. Os serviços podem ser contratados e gerenciados de qualquer aparelho, em qualquer lugar, a qualquer hora.

- **Aglomerado de recursos computacionais**: Os serviços oferecidos pelo provedor, em escala mundial, são distribuídos em diversos *datacenters* espalhados por várias regiões do mundo, com diferentes condições de precificação, latência, tipos específicos de serviços oferecidos. O usuário não enxerga onde essa capacidade computacional está distribuída. De forma invisível, ao provedor oferece recursos físicos e virtuais que podem ser escaláveis conforme a necessidade do cliente.

- **Rápida elasticidade**: Aplicações em nuvem conseguem rapidamente aumentar e diminuir de tamanho conforme a necessidade do cliente, muitas vezes de forma automática. Para o cliente, a capacidade computacional aparentemente é ilimitada e, pode controlar mínimos e máximos de elasticidade que deseja para sua aplicação. O monitoramento neste caso é fundamental, para ambos os lados.

- **Medição do serviço consumido**: Métricas são fundamentais para a computação em nuvem. Os sistemas em nuvem têm a capacidade de otimizar seus recursos computacionais e controlá-los metrificando sua capacidade. Serviços importantes como armazenamento de arquivos, processamento de dados, largura de banda são monitorados, controlados e reportados em tempo real, trazendo transparência entre o provedor da computação em nuvem e seu cliente que consegue tomar decisões baseadas no uso de seus recursos. Com tais definições, a computação em nuvem torna-se atrativa para seus consumidores como sendo uma alternativa a reduzir o investimento em bens de capital (chamado de *CAPEX – Capital Expenditure*), bem como despesas operacionais (*OPEX – Operational Expenditure*) de uma empresa. É possível que uma empresa exponha seu produto desenvolvido sem necessariamente ter uma infraestrutura de servidores, redes e pessoal para gerenciar a aplicação. A empresa neste caso simplesmente “aluga” a capacidade computacional de um provedor de computação em nuvem.

## 2.2 Modelos de Serviços de Computação em Nuvem

Diferentes modelos de computação em nuvem são oferecidos pelos provedores, níveis diferentes de abstração e controle do cliente sobre suas aplicações. É importante destacar que serviços diferentes estão inerentes a esses níveis de abstração e tais níveis podem estar disponíveis de diferentes formas, com diferentes métricas de precificação. (MELL; GRANCE, 2011) dividem a computação em nuvem em três modelos:

- **SaaS** – *Software as a Service* – *Software* como Serviço: Pode ser considerado o nível no qual o cliente tem o menor controle sobre determinado serviço oferecido em nuvem. Nessa camada, o usuário apenas usa aplicações oferecidas pelo provedor, como o *e-mail*, por exemplo. O nível de *software* como serviço é acessível

por meio de uma interface gráfica de navegador de *internet* ou programa. O usuário dessa aplicação não controla nem gerencia a infraestrutura do provedor do serviço como processamento, rede, sistema operacional, armazenamento, ou seja, não tem poder sobre o *hardware* da aplicação que utiliza.

No modelo de *Software* como Serviço, (CHATURVEDI; RASHID, 2019) elencam benefícios como rápida escalabilidade, acesso amplo de qualquer lugar da *internet*, despreocupação quanto à infraestrutura, níveis diferentes de serviços oferecidos no modelo e manutenção e suporte ágil.

- **PaaS** – *Platform as a Service* – Plataforma como Serviço: Nesse nível de abstração, o provedor oferece ao usuário o controle das linguagens de programação, bibliotecas, serviços e ferramentas suportadas na infraestrutura da nuvem. No caso o consumidor ainda não consegue controlar o sistema operacional, armazenamento e *hardware* em geral. É possível identificar os benefícios deste modelo como baixo custo, se comparado a ambientes *on premises*, desenvolvimento simplificado, ampla colaboração dos usuários envolvidos no modelo e, não necessidade de *upgrade*s no que tange a infraestrutura das empresas que se beneficiam de tal modelo (CHATURVEDI; RASHID, 2019).

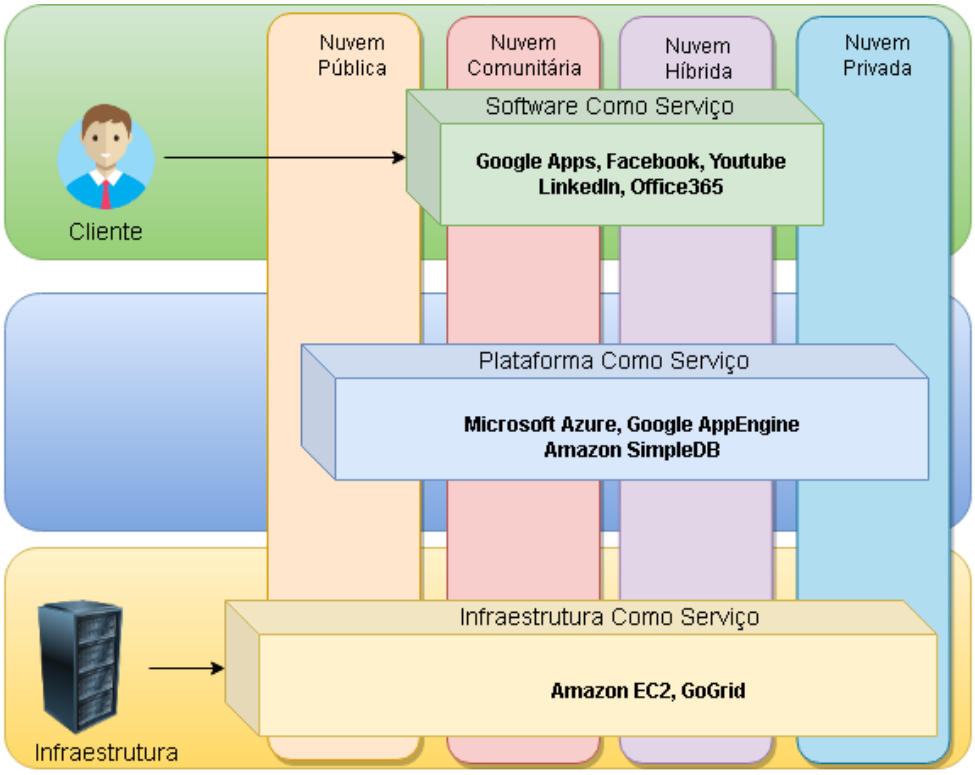
- **IaaS** – *Infraestructure as a Service* – Infraestrutura como Serviço: Pode ser considerado o nível onde o cliente tem maior acesso e controle sobre o serviço em nuvem contratado de um provedor. O cliente neste caso tem poder para provisionar e controlar o processamento, armazenamento e rede empregado em determinado serviço em nuvem. O usuário pode instalar o sistema operacional, fazer atualizações, se preocupar com a segurança de seu serviço.

Por fim, soluções que utilizam Infraestrutura como Serviço beneficiam-se de redução dos bens de capital, pagamento conforme o uso, acesso a uma gama de recursos computacionais de alto nível, ou *Entreprise* e possibilidade de elasticidade desses sistemas.

Os três modelos ainda sim são passíveis de vulnerabilidades de segurança, os autores (BASU, et al. 2018) apontam segurança, localidade, integridade, segregação, confidencialidade e acesso aos dados, segurança de redes, autenticação e autorização como sendo possíveis brechas em ambientes em nuvem.

Na Figura 1 é ilustrado o modo como esses modelos de serviço são distribuídos e alguns exemplos presentes em cada nível de abstração. É possível observar uma demonstração de quatro diferentes modelos de implantação, os quais compõem da mesma forma as três premissas de abstração da computação em nuvem (MELL; GRANCE, 2011). O modelo de implantação de nuvem privada tem como principal característica ser provisionado exclusivamente por uma única empresa composta por múltiplos clientes. Deve ser gerenciada pela própria organização, ou terceiros, ou ambos. Pode-se trabalhar nesse tipo de modelo de forma *on* ou *off premises*, isso é, o *hardware* e infraestrutura tem sua implantação fixada no local da empresa, o que requer melhor planejamento e consequentemente maiores custos, ou não.

Figura 1 – Modelos de Computação em Nuvem e alguns exemplos.




Fonte: Desenvolvido pelo autor (2022). Adaptado de (SURYATEJA, 2018; RASHID, CHATURVEDI, 2019; SUBASHINI, KAVITHA, 2010)

Outro modelo de implantação presente na Figura 1 é o modelo de nuvem comunitária, que tem como características ser provisionada exclusivamente para um grupo específico de consumidores que possuem preocupações em relação ao compartilhamento de recursos em nuvem, como por exemplo missões

governamentais. Pode ser gerenciada e operada por uma ou mais organizações, até mesmo terceiros. Também segue a possibilidade de ser *on* ou *off premises*.

Já o modelo de nuvem pública é provisionado para todos que desejarem assinar serviços de computação em nuvem. Pode-se utilizar esse tipo de modelo no meio acadêmico, governamental, no ramo de negócios e, diferente dos modelos anteriores, existe apenas *on premises* do provedor de computação em nuvem a qual o serviço é contratado, geralmente *datacenters* espalhados pelo globo.

A modalidade de nuvem híbrida, como o nome sugere, é a junção entre dois ou mais modelos de nuvem, citados na Figura 1. Há a possibilidade de heterogeneidade entre provedores de computação em nuvem, que possibilitam portabilidade de dados e aplicações.

## 2.3 Computação sem servidor

Aplicações convencionais em nuvem são ofertadas aos seus usuários por meio de máquinas virtuais hospedadas e virtualizadas dentro de *datacenters* dos provedores, distribuídas globalmente. Um consumidor de uma aplicação em nuvem neste contexto paga por toda a máquina virtual que está provisionada em sua aplicação. Métricas como custo, latência e elasticidade são levadas em consideração no desempenho de uma aplicação.

Diversas aplicações não utilizam a totalidade de recursos computacionais de uma máquina virtual para ser provisionada. Com isso, conceitos de otimização foram pensados nos últimos anos para aprimorar o uso da capacidade computacional em nuvem e oferecer para mais clientes uma solução menos custosa. Neste contexto, a abstração da computação sem servidor emergiu como uma alternativa viável, utilizando de um modelo de precificação diferente do convencional que cobra apenas pela aplicação em si que é executada, não toda a infraestrutura de máquina virtual que a aplicação usa.

A nomenclatura “sem servidor” não implica que a computação seja de fato sem um servidor. No contexto atual tem o significado que parte do servidor, parte de uma máquina virtual é alugada para execução de determinada aplicação, de uma parte de código, por um período menor do que a computação em nuvem convencional oferece (VARGHESE; BUYYA, 2018). Isso quer dizer que, na computação sem servidor, o

desenvolvedor tem o custo inerente à execução de determinado pedaço de código, não de uma máquina virtual inteira para a execução.

A abstração sem servidor da computação em nuvem permite que desenvolvedores executem suas aplicações com custo granular inerente ao uso, não havendo a necessidade de se configurar sistemas físicos e lógicos responsáveis pela execução do código (EYK, et al. 2018). Segundo (CASTRO, et al. 2019), tais aplicações abstraem uma complexidade na infraestrutura em nuvem, de forma a proporcionar ao desenvolvedor a liberdade para pensar apenas na execução do código, não no provisionamento de máquina virtual, configuração da mesma e outras preocupações.

A computação *serverless* teve seu primeiro modelo introduzido pela empresa *Amazon*, com sua plataforma *AWS Lambda* em 2014 e posteriormente, com avanços na indústria e da academia, começou a ser oferecida por outras empresas provedores de computação em nuvem como o *Google Serverless Computing* e *Azure Functions*. Com o intuito de agilizar o desenvolvimento de soluções, o provedor se encarrega de provisionar, gerenciar e escalar servidores, máquinas virtuais, *containers* para que desenvolvedores apenas carreguem o código de seus programas.

Por ser um paradigma emergente na computação em nuvem, a computação *serverless* é empregada em aplicações de aprendizado de máquina, aprendizagem profunda, computação numérica, processamento de vídeo, *Internet* das Coisas, análise de *big data* (WEN, et al. 2023), entre outros. Em 2025, (WEN, et al. 2023) indica que 50% das empresas usarão computação sem servidor, gerando um valor de mercado para tais aplicações de mais de 22 bilhões de dólares, segundo a (GARTNER, 2022).

As características únicas da computação sem servidor chamam atenção da comunidade de desenvolvedores de *software* para o desenvolvimento de aplicações *serverless* (*serverless-based applications*). Diversos trabalhos acadêmicos focam em desafios que envolvem a evolução da computação sem servidor (TAIBI, 2020), modelagem de aplicativos (EISMAN et al. 2021; EISMAN et al. 2020), modelagem de aplicações (YUSSUPOV, 2022), *framework* de programação para aplicações específicas (BERMBAC, et al. 2022; ZHANG, et al. 2021), desenvolvimento de plataformas de computação em nuvem heterogêneas (SAMPÉ, et al. 2020), aplicações *serverless stateful* (BARCELONA-PONS, et al. 2022), migração de aplicações de monólitos *on-premises* (RISTOV, et al. 2020), economia *serverless*

(ADZIC; CHATLEY, 2017), *datasets* (ESKANDANI; SALVANESCHI, 2021) e testes e *debugs* de aplicações (LENARDUZZI; PANICELLA, 2020).

Além disso, os desafios e avanços não estão presos à comunidade de desenvolvedores de *software*. Existem pesquisas que envolvem desenvolvimento de *software*, redes de computadores e segurança, pesquisas sobre gerenciamento de recursos (HOSEINYFARAHABADY, et al. 2017; DE PALMA, et al. 2020; ZHANG, et al. 2021), otimização de desempenho de arranque a frio (AKKUS, et al. 2018; FUERST; SHARMA, 2021; OAKES, et al. 2018), comunicação entre funções (JIA; WITCHEL, 2021; SHILLAKER; PIETZUCH, 2020), *frameworks* de programação generalizada (FOULADI, et al. 2019; JANGDA, et al. 2019), entre outros.

Alguns autores dividem a computação sem servidor em duas vertentes (NUPPONEN; TAIBI, 2020; STEINBACH et al. 2022; KOSCHEL et al. 2021; LI et al. 2022, STEINBACH; JINDAL; CHADHA, et al. 2022). São elas: *FaaS* e *BaaS*, Funções como Serviço e *Backend* como Serviço, respectivamente.

**Funções como Serviço** oferece um ambiente voltado ao desenvolvimento de *softwares* em nuvem com execução baseada em eventos. Tais sistemas baseados em nuvem dependem de uma combinação da lógica de programação por parte do cliente e chamadas de procedimentos advindas de serviços terceiros ao provedor da capacidade computacional em nuvem.

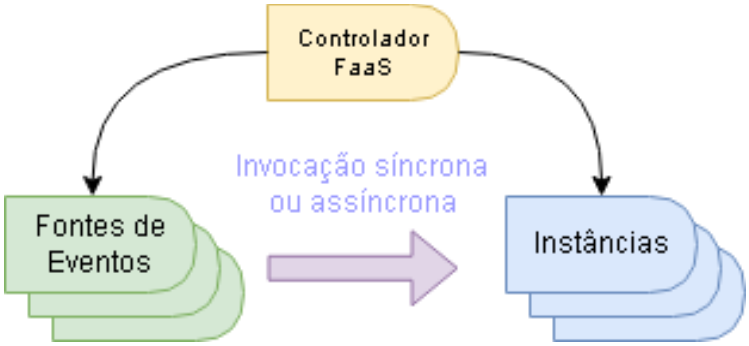
Desenvolvedores de aplicações *FaaS* programam seus códigos posteriormente executados em ambientes isolados na nuvem, cada porção do código representa uma parte de uma aplicação maior e, o tempo de execução é limitado pelo provedor do serviço, por exemplo 15 minutos para execuções na plataforma Lambda (*AWS*, 2022).

Funções ficam ociosas até que sejam acionadas por algum tipo de evento como requisição de usuário, clique do *mouse*, eventos gerados por sistemas terceiros e/ou entrada e saída de dados; essas funções não ficam ativas o tempo todo, pois implicam em maior custo. O provedor, neste caso, é a responsável por escalar a execução das funções quando elas são invocadas, baseado no número de execuções que ocorrem em resposta aos eventos que chegam.

As funções baseadas em eventos são executadas em *containers* isolados com tempo de vida útil de cada função podendo ser alguns milissegundos. *FaaS* naturalmente são isoladas, ou *stateless*, pois não há garantia em relação a integridade das informações que ali invocaram a função. O provedor do serviço limita o tempo de vida útil de uma função (PAAKKUNAINEN, 2019).

Um ambiente de *FaaS* é composto por um controlador, fonte de eventos e instancias de funções. Na Figura 2 ilustra-se a interação entre os três componentes do ambiente. O controlador gerencia o desenvolvimento das funções, além de gerenciar a escalabilidade e monitoramento das instâncias e controla as funções e fontes de eventos. Fontes de eventos acionam ou transmitem eventos para as instâncias e, as instâncias são os ambientes de execução para as funções.

Figura 2 – Modelo de processamento *FaaS*




Fonte: Desenvolvido pelo autor (2022). Adaptado de (PAAKKUNAINEN, 2019).

Tipicamente, o ciclo de vida de funções como serviço começa quando o código e as configurações são enviados pelo desenvolvedor para o provedor da computação em nuvem. Essas configurações contêm metadados que definem o comportamento e limitação da elasticidade da aplicação, definição das fontes de eventos e outras variáveis. Já na nuvem, o provedor recebe o código e configurações e converte em *containers* ou outro artefato que é implementado como uma instancia de funções. Após a criação do *container*, o controlador *FaaS* implementa e cria novas instâncias de funções baseadas em eventos. As funções ficam operantes até que a remoção seja solicitada pelo desenvolvedor da aplicação (CNCF, 2018).

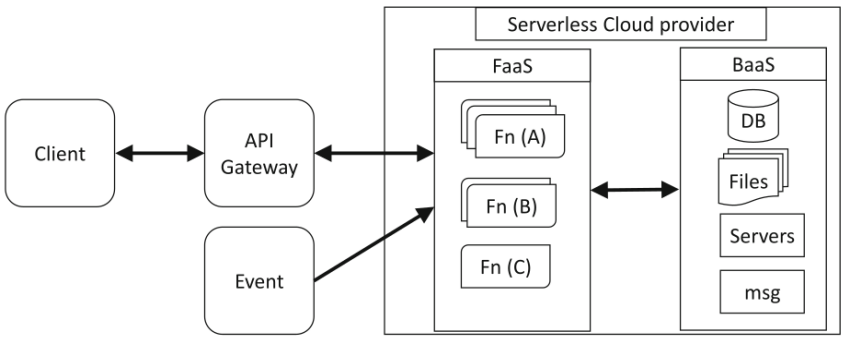
***Backend* como Serviço** permite comutar processos e componentes do lado do servidor e partir para a utilização de serviços de prateleira sob demanda. Esta modalidade de serviço possibilita aos desenvolvedores terceirizarem todos os aspectos por trás de qualquer aplicação, abstrair a escrita e manutenção de sistemas para o *frontend*, ou seja, diretamente ligado à experiência do usuário de uma aplicação. Exemplos disso são sistemas de autenticação de dois fatores, gerenciamento de banco de dados e armazenamento de arquivos em nuvem.

Segundo (AMORIM, 2017), serviços de *BaaS* caracterizam-se por pacotes hospedados na nuvem como *APIs* e *SDKs* para serem utilizados em demais aplicações. Utilizar um *backend* como serviço facilita tarefas de *login* e armazenamento de arquivos em diversos tipos de aplicações, todo o trabalho de autenticação, por exemplo, é terceirizado para o *BaaS*, reduzindo assim custos, tempo de configuração e manutenção, com possibilidade de elasticidade de aplicações que se beneficiam de tal serviço.

A computação sem servidor ainda não apresenta uma definição formal, pode tornar-se um desafio ao diferenciar os modelos de computação em nuvem, visto que *PaaS* e *SaaS* são modelos que também não possibilitam interação com a infraestrutura física dos servidores em nuvem. *PaaS*, por exemplo, abstrai os servidores do usuário, mas diferentemente do *PaaS*, o *FaaS* usa funções como unidade de implantação. Outrossim, os provedores da nuvem retem maior responsabilidade sobre a operação do *software* no modelo *FaaS*.

Outra constatação é que o modelo *SaaS* provém funções de *backend* e *frontend*, acessível para o usuário. *BaaS* oferece apenas as funcionalidades *backend*, que se faz necessário ainda uma aplicação para utilizar (EYK, et al. 2017). Caso a computação sem servidor fosse encaixada nos modelos de computação em nuvem encontrados na Figura 1, seria encaixada entre os modelos *PaaS* e *SaaS*. Na Figura 3 sintetiza-se a arquitetura da computação sem servidor e as categorias descritas.

Figura 3 – Arquitetura da computação sem servidor




Fonte: Survey on *serverless* computing. (HASSAN, et al. 2021).

## 2.4 Exemplos do uso da computação sem servidor

Na Figura 4 ilustra-se um exemplo do funcionamento de uma solução *serverless*, oferecida pela *AWS Lambda*. A imagem exemplifica o processamento de

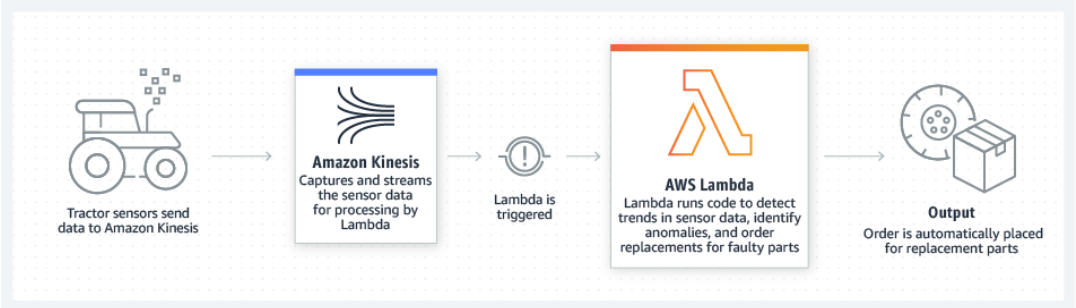
arquivos, uma das possibilidades do modelo sem servidor. Uma fotografia é tirada a partir de qualquer dispositivo, é carregada para o sistema de armazenamento em nuvem *Amazon S3*, este evento aciona o sistema *Lambda* que se encarrega de aplicar à imagem um algoritmo de redimensionamento de imagens e, finalmente, a fotografia é redimensionada para tamanhos otimizados para *web*, telefones celulares e *tablets*.

Figura 4 – Exemplo de funcionamento do *AWS Lambda* em processamento de arquivos

Fonte: Amazon Lambda. (*AWS*, 2022).

Outra aplicação do uso da computação sem servidor é relacionada ao IoT – *Internet of Things*. Na Figura 5 é possível observar o funcionamento da computação sem servidor em ambiente de *internet* das coisas. Sensores em um trator captam dados agindo como gatilho para plataforma *serverless*, que executa pedaços de código que detectam tendências, anomalias e acionam pedido de substituição de peças com defeitos. Por fim, como resultado da execução, automaticamente realiza pedido de novas peças para substituição.

Figura 5 – Exemplo de funcionamento do *AWS Lambda* em *IoT*



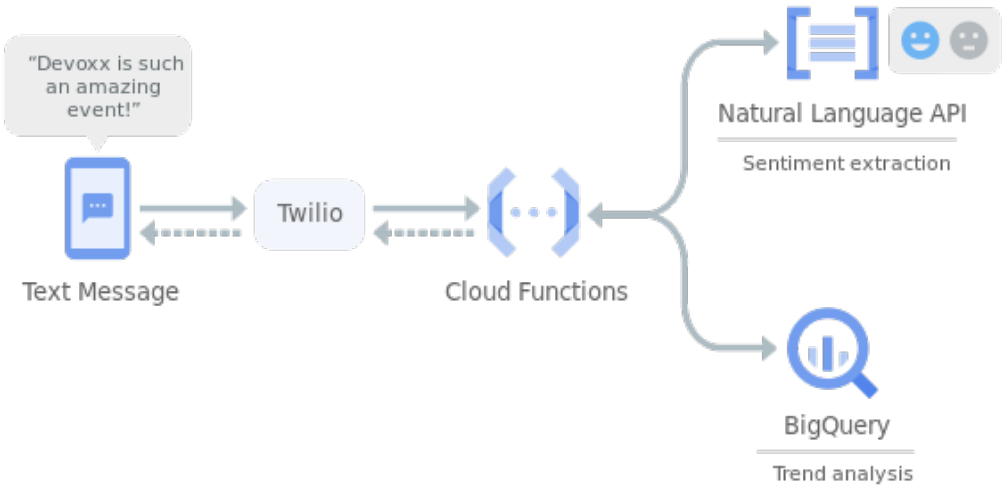

Fonte: Amazon Lambda. (*AWS*, 2022).

Com outras propostas de soluções sem servidor, o provedor *Google* oferece produtos como *Cloud Run*, possibilitando a criação de aplicativos containerizados em linguagens como *Go*, *Python*, *Java*, *Node.JS*, *.NET* em poucos segundos. *Cloud*

*Functions*, viabilizando execução de pequenas porções de código que respondem a eventos, ou gatilhos, correspondente a um produto de função como serviço (*FaaS*) de forma escalonável e que é medido por período de execução, tem como base o valor de 100 milissegundos para execução (GCP, 2022).

Relevante de se observar que todos os eventos e respostas são rapidamente integrados aos serviços já usados pelos provedores, como é o exemplo demonstrado na Figura 6, onde mostra-se o gatilho (mensagem de texto) acionando funções que analisam o conteúdo do texto e extração de sentimentos, por meio de inteligência artificial. Segundo o próprio provedor, é possível estender a usabilidade da computação sem servidor para criação de *sites* na *web*, integração de *APIs*, automação, serviços de *backend* e análise em tempo real de dados.

Figura 6 – Exemplo de funcionamento do *Google Functions* em IA




Fonte: Google Functions. (GCP, 2022).

## 2.5 Vantagens e Desvantagens

É possível encontrar vantagens e desvantagens ao utilizar a computação sem servidor e analisar como os aspectos positivos e negativos se comparam com os demais modelos de computação em nuvem. Como vantagens encontradas, tem-se o escalonamento ágil, sem despesa por ociosidade, sem custo adicional por escalabilidade, redução de custo operacional, *Fast Time to Market* e uso eficiente de recursos.

- **Escalonamento ágil**: Provedores de serviços em nuvem escalam seus serviços automaticamente e de forma transparente para o usuário (CNCF, 2018), uma aplicação ao utilizar a arquitetura sem servidor tem a característica de elasticidade, aumentar e diminuir de tamanho conforme a necessidade, seja pelo número de usuários utilizando determinado serviço, ou até mesmo pela quantidade de eventos ocorrendo no momento (IVANOV, 2019).

- **Sem despesas por ociosidade**: Ao utilizar os serviços de um provedor de computação em nuvem pode-se observar a capacidade de otimização de custos. No caso da computação sem servidor, quando determinada função ou aplicação está ociosa, o desenvolvedor não é cobrado por isso. Isso se dá pelo fato de as *FaaS* serem criadas sob demanda e, em combinação com o BaaS, os recursos são compartilhados e não alocados individualmente (CNCF, 2018). Neste cenário, não há custo dos *containers* ou máquinas virtuais quando o código não é executado, com o uso da computação sem servidor, um banco de dados não gera gastos ao seu desenvolvedor até que a primeira requisição ou conexão seja feita, o armazenamento não é custeado até que a primeira transferência ocorra (ADZIC, 2017).

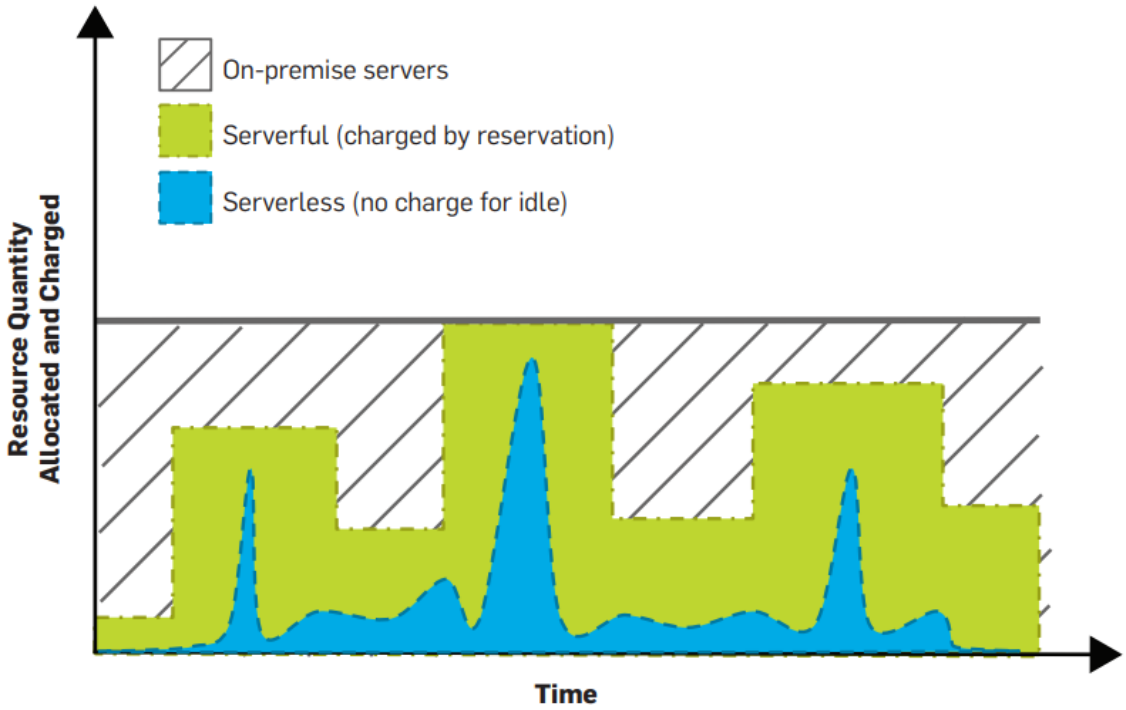
- **Sem custo adicional por escalabilidade**: O fato de não contabilizar a ociosidade das aplicações, como por exemplo custo, favorece também a elasticidade das aplicações. Existe o custo em relação ao tempo adicional da computação, mas não para aumentar ou diminuir determinada aplicação, isso implica em escalar do zero até milhares, milhões de requisições, o que acontece de forma transparente para o usuário, com o mínimo de custo e esforço, e máxima transparência (IVANOV, 2018). Serviços de computação sem servidor como *AWS Lambda* e *Google Functions* cobram do desenvolvedor a cada 100 ms de tempo de computação (ROBERTS, 2018).

Redução de custo operacional: O uso da computação sem servidor também implica no custo operacional de uma aplicação, por conta da não necessidade de provisionar, instalar, atualizar e gerenciar a infraestrutura de servidores responsáveis pela hospedagem das aplicações. Neste quesito, o modelo de plataforma como serviço assemelha-se a premissa de redução de custos operacionais (CNCF, 2018). A adoção de máquinas virtuais pode gerar uma economia se comparada a ambientes *on-premises* (servidores físicos), e a computação sem servidor pode gerar uma

economia se comparada as máquinas virtuais ofertadas pelos provedores dos serviços em nuvem.

Na Figura 7 é possível observar uma análise feita por (SCHLEIER-SMITH, et al. 2021) comparando a diferença de custo em ambientes em nuvem, servidores físicos e computação sem servidor, no que tange o custo envolvido nas diferentes modalidades. É de se notar a utilização da computação sem servidor pode gerar economia dependendo do contexto a ser utilizada, seguindo a modalidade de custo granular inerente a demanda.

Figura 7 – Computação tradicional *versus* Computação em Nuvem *versus* Computação sem Servidor




Fonte: SCHLEIER-SMITH, et. al (2021).

- **FTTM – *Fast Time to Market***: O tempo de uma aplicação para atingir o mercado pode ser reduzido consideravelmente quando há o uso da computação sem servidor. Premissa da necessidade de não se preocupar com a construção e provisionamento de uma infraestrutura adequada para aplicação, juntamente com o serviço compartilhado do BaaS reduzem o tempo de implementação de qualquer aplicação (ROBERTS, 2018).

- **Uso eficiente de recursos**: Por ter seus recursos compartilhados e ter um determinado tempo de vida útil, a computação sem servidor oferece uso dos recursos computacionais de feitos de forma mais eficiente. Autores (ROBERTS, 2018; KOOMEY; TAYLOR, 2017) mostram que os recursos computacionais provisionados na computação em nuvem não são totalmente utilizados. Com a computação sem servidor, provedores podem compartilhar recursos computacionais entre seus clientes de forma mais otimizada, sob demanda, sem a necessidade de reservar máquinas virtuais inteiras para apenas um cliente. Como exemplo deste ponto, para servir a cem requisições *HTTP* em cinco minutos, as instâncias seriam criadas sob demanda para servir as requisições, mas depois dos cinco minutos todas as instâncias seriam descartadas e a capacidade computacional poderia ser gerida para outra finalidade, para outro cliente.

Em comparação a computação tradicional, um servidor dedicado ficaria ligado para servir as mesmas requisições, mas ao término do tempo, o servidor dedicado ficaria ocioso, gastando recursos e capacidade computacional. Computação sem servidor neste sentido pode permear um caminho de computação verde, mais eficiente energeticamente.

As desvantagens são: desempenho, latência na inicialização, ou *cold start*, *lock-in* do provedor, segurança, precificação, sistemas legado e tempo de execução.

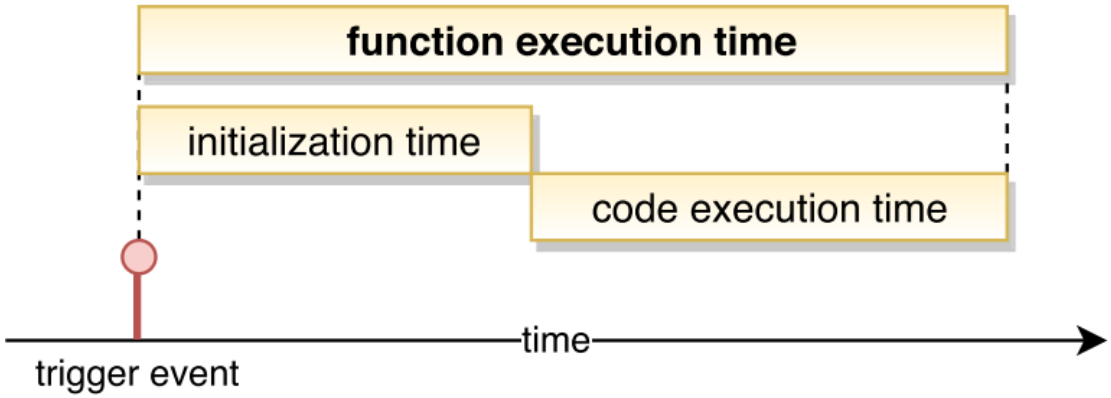
- **Desempenho**: A computação sem servidor possui desafios relacionados ao desempenho e orquestração dos *containers* usados na execução das funções, agendamento e sobrecarga de chamadas de serviço em uma determinada instância, ativação de funções, mapeamento de eventos, ou gatilhos que geram as chamadas das execuções. Ações como tempo de inicialização, tempo de carregamento de bibliotecas e inicializações específicas no código do usuário causam impacto do ponto de vista de funções como serviço.

- **Latência de inicialização – *cold start***: Ocorre quando uma função como serviço é invocada, com base em um evento ou gatilho, a mesma deve ser executada em um *container* já provisionado, ou um novo *container* precisa ser criado durante o processo. É dado o nome de *warm start*, início quente quando os recursos já estão executando e, *cold start*, início a frio, quando os recursos precisam ser alocados antes da execução. O início a frio leva mais tempo para inicializar por conta

do provisionamento, atrasando a execução de determinada função em alguns milissegundos, podendo chegar a alguns segundos (PAAKKUNAINEN, 2019).

Dependendo da aplicação isso pode ser um problema crítico, como aplicações *web* por exemplo. A repetição de inícios a frio depende do tráfego da aplicação. Aplicações com um tráfego consistente ou dez requisições por segundo se deparam com inícios a frio raramente. Já tráfegos periódicos e que oscilam demasiadamente, inícios a frio são mais frequentes (ROBERTS, 2018). O processo de como ocorre o atraso de inicialização de uma função como serviço é observado na Figura 8.

Figura 8 – O *cold start* atrasa a execução do código




Fonte: STEINBACH, et. al (2021).

- ***Lock-in* do provedor**: É notável a quantidade de provedores que oferecem seus serviços de nuvem para seus clientes, especificamente a computação sem servidor. É possível notar que os serviços de computação sem servidor são amplamente integrados a outros serviços do próprio provedor, como por exemplo serviços de armazenamento e inteligência artificial. Isso gera um desafio quanto um aprisionamento a um único provedor, trazendo dificuldades na hora de migrar de um provedor para outros.

Problemas dessa espécie podem ser mitigados posteriormente com padronizações e ferramentas avançadas relacionadas a computação sem servidor. Algumas ferramentas em desenvolvimento ajudam a desenvolver e implementar códigos independendo de provedores (ROBERTS, 2018), junto a isso, a *Cloud Native Computing Foundation* tem trabalhado na padronização da computação sem servidor, com a descrição, publicação e consumo de eventos (CNCF, 2018).

- **Segurança**: Atualmente preocupação para todo tipo de computação, seja sem servidor, na nuvem ou computação convencional. O desafio em questão é o isolamento das funções que são executadas. Por conta do compartilhamento de recursos com outros usuários e outras funções, processar e executar funções que atuam sob dados sensíveis torna-se uma tarefa preocupante no que diz respeito a segurança dessas funções.

- **Precificação**: O foco de provedores da computação sem servidor e reduzir ao máximo os recursos computacionais empregados na execução de tarefas e funções. O modelo de precificação é desafiador, um exemplo disso é o uso de *CPU* que em alguns provedores é menos custoso, em contrapartida entradas e saídas podem se tornar mais caras do que máquinas virtuais dedicadas. Desta forma existe uma necessidade de atualizar métricas para precificar este tipo de serviço em nuvem.

- **Sistemas Legado**: São sistemas mais antigos que usam técnicas e muitas vezes pré-requisitos mais obsoletos que atualmente estão fortemente presentes por muitas empresas. Em algum momento faz-se necessário levar esses sistemas legado para a nuvem e isso pode ser um desafio maior para a computação sem servidor. A migração deve acontecer em algum momento e deve chegar aos sistemas sem servidores.

- **Tempo de execução**: O modelo de funções como serviço introduz um novo paradigma para codificação e execução de aplicações na nuvem. Por acionar *containers* responsáveis pela execução das funções que facilmente podem ser descartados, todas as funções não devem salvar estado ou informação durante suas invocações. Os provedores de computação sem servidor restringem o tempo de execução das funções, se determinada função não é acionada e executada durante um tempo, tal função é descartada. Como exemplo, *AWS* restringe o uso de funções na *Lambda* em 15 minutos (*AWS*, 2022), já o Google, com seu serviço Google Functions, resistente a execução das funções em 9 minutos (GCP, 2022).

Por fim, uma comparação pode ser observada na Tabela 3, que leva em consideração aspectos como facilidade de desenvolvimento de aplicações, escalabilidade, ciclo de vida de uma função, segurança, resolução de problemas,

tolerância a falhas, custo de carga variável, carga estática e usuário para qual é voltado.

Tabela 3 – Comparação entre *serverless* e computação tradicional

| **Característica** | **Tradicional** | **Sem Servidor** |
|---|---|---|
| Desenvolvimento de aplicações | Difícil | Fácil |
| Escalabilidade | Difícil | Fácil |
| Ciclo de vida da função | Longa | Rápida |
| Segurança | Complexa | Complexa |
| Resolução de Problemas | Fácil | Difícil |
| Tolerância a falhas | Menos confiável | Mais confiável |
| Custo (Carga variável) | Caro | Barato |
| Custo (Carga estática) | Barato | Caro |
| Usuário | Administrador e Desenvolvedor | Desenvolvedor |

Fonte: Desenvolvido pelo autor (2022). Adaptado de (HASSAN, et al. 2021).

# CAPÍTULO 3 – PLATAFORMAS *SERVERLESS*

Neste presente capítulo diferentes formas de implementação da computação sem servidor são expostas, tanto do ponto de vista comercial quanto do ponto de vista *opensource*. Encontram-se também no capítulo alguns comparativos entre plataformas de computação sem servidor, e diferentes aplicabilidades.

## 3.1 Plataformas comerciais

Existem vários provedores de computação em nuvem que oferecem a computação sem servidor, diversas plataformas oferecem funcionalidades e linguagens de programação distintas. Provedores comerciais distribuem soluções prioritárias e vinculadas a outros serviços do próprio provedor, portanto existe uma crescente tendência a plataformas *serverless* de sistema aberto (MANOR; TEICH, 2019). Na Tabela 4 são apresentados os três principais provedores de computação sem servidor, suas respectivas linguagens de programação suportadas, preço por um milhão de requisições e preço por um milhão de segundos, ambos os preços em dólares.

Tabela 4 – Lista dos três principais provedores de computação sem servidor - *FaaS*

| ***Provedor*** | **Linguagens suportadas** | **Preço por 1M de requisições** | **Preço por 1M de segundos** |
|---|---|---|---|
| *AWS Lambda* | Java, Go, PowerShell, *Node.JS*, C#, Python, Ruby, ambientes de execução personalizados | $0.2 e $3.7 (HTTP) | $0,208 |
| *Azure Functions* | C#, JavaScript, F#, Java, Python, TypeScript, PHP, Batch, Bash, PowerShell | $0.2 | $0.2 |
| *Google Cloud Functions* | *Node.JS*, Python, Go | $0.4 | $0,213 |

Fonte: Adaptado de PAAKKUNAINEN (2019).

Dois dos três maiores provedores de computação sem servidor suportam pelo menos cinco linguagens de programação diferentes, *AWS Lambda* oferece ainda um

ambiente de execução personalizado, estendendo o suporte a outras linguagens de programação (*AWS*, 2022).

O mesmo autor elucida também a ideia de diferentes cargas de trabalho para os provedores de computação sem servidor e o custo inerente às cargas. É possível observar na Tabela 5 cargas de trabalho de 10 bilhões de requisições *HTTP* em 200 milissegundos e 10 milhões de requisições assíncronas em 60 segundos de cada uma dos provedores, os valores estão quantificados em dólares.

Tabela 5 – Comparação entre custo das plataformas *FaaS*

| Carga de Trabalho | *AWS* Lambda | Azure Functions | Google Cloud Functions |
|---|---|---|---|
| **10 bilhões de requisições 200ms** | $4116.75 | $600.00 | $862.50 |
| **10 milhões de requisições assíncronas** | $1252.25 | $1202.00 | $1391,50 |

Fonte: Adaptado de PAAKKUNAINEN (2019).

Provedores de computação sem servidor também possuem diferenças relacionadas a quão rápida uma função é executada, o que é importante pois afeta diretamente o custo de execução das funções como serviço (RANDALL, 2018).

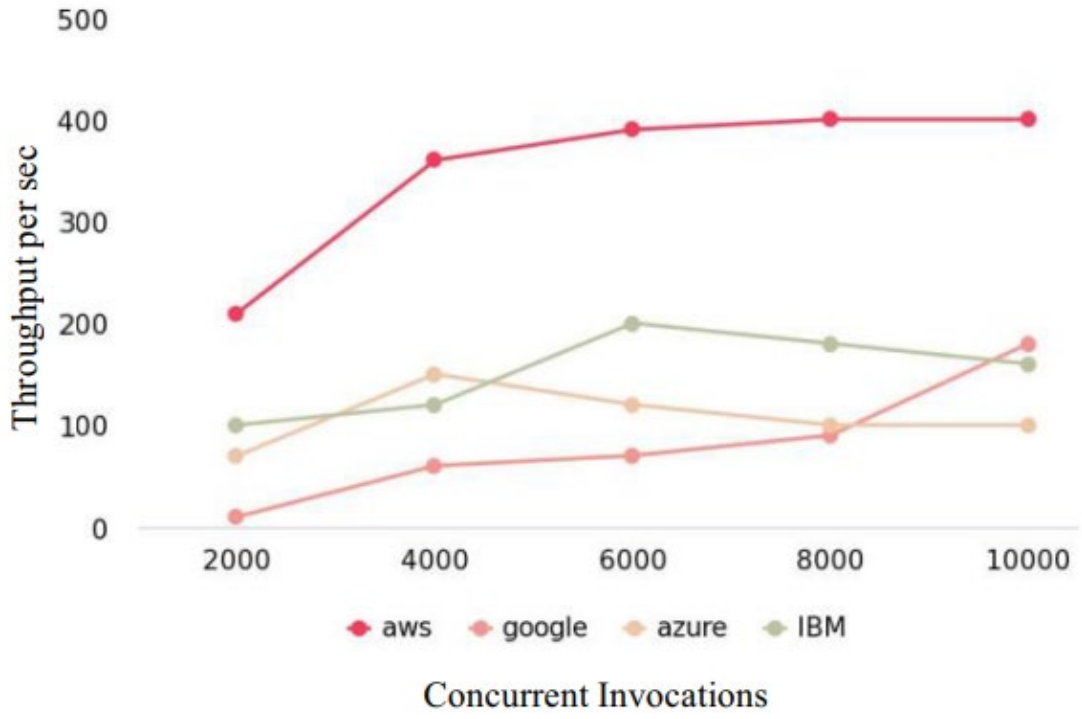
Tempos de execução, os inícios a frio e início quente (*cold* e *warm start*) diferem também dependendo da linguagem de programação empregada (ALLIAUME; LE ROUX, 2018).

Diversos comparativos entre as plataformas comerciais são feitos para diferentes funcionalidades que entregam aos consumidores, (KUMAR; SELVAKUMARA, 2022) realizaram testes e análises de desempenho e escalabilidade das plataformas da *Google*, *Microsoft*, *Amazon* e *IBM*. Na Figura 9 mostra-se o desempenho das quatro provedores de computação sem servidor em relação à execução 500 a 10000 funções concorrentes.

Com a análise realizada na Figura 9, é possível interpretar que ao início da execução, o serviço *AWS* atinge sua máxima capacidade de requisições concorrentes. *Azure Functions* e *IBM* têm um comportamento similar, atingindo um limite de 200 invocações e declinando conforme o número de execuções vai aumentando. *Google Functions* por sua vez, exibe um moderado e consistente

aumento na largura de banda, sendo mais rápido, ao final do experimento, que *IBM* e *Azure*.

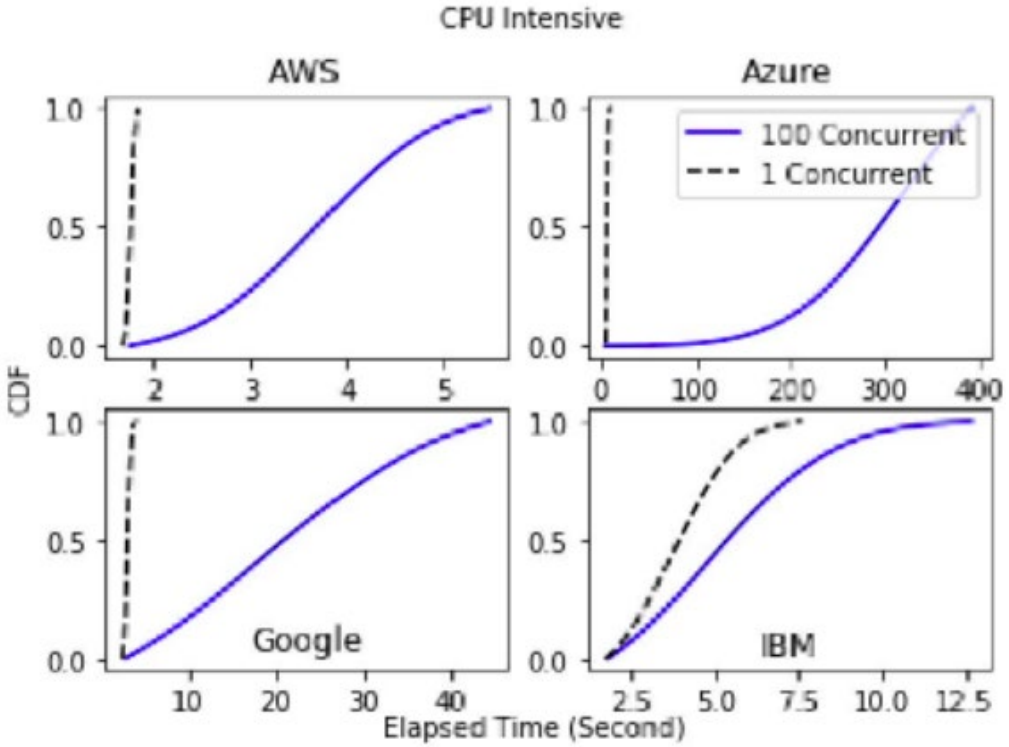
Figura 9 – Relação entre a taxa de transferência por segundo e o número de execuções concorrentes




Fonte: KUMAR; SELVAKUMARA (2022).

Outra análise feita pelo estudo compara a carga de trabalho do processamento da *CPU* (KUMAR; SELVAKUMARA, 2022). Os pesquisadores desenvolveram uma função de multiplicação de matrizes para estressar o poder de *CPU* executando uma função com invocações simultâneas na multiplicação de uma matriz bidimensional. Na Figura 10 ilustra-se a execução entre um e cem invocações simultâneas ao longo da duração de execução da função.

Figura 10 – Sobrecarga para a simultaneidade da *CPU*




Fonte: KUMAR; SELVAKUMARA (2022).

Tais razões elevam a importância de se escolher o melhor provedor de computação sem servidor, é necessário testar cuidadosamente a carga de trabalho e linguagem de programação para que o custo seja otimizado (BALDINI, et al. 2017).

## 3.2 Plataformas *FaaS opensource*

Plataformas de código aberto são uma opção viável para evitar aprisionamentos do provedor – *lock-in* discutidos na seção 2.5 – e não depender de apenas um provedor, possibilita assim uma construção heterogênea de soluções sem servidor (MOHANTY, 2018). Plataformas de código aberto permitem execução de cargas de trabalho em nuvens privadas. O ponto negativo desta prática da nuvem privada implica diretamente na escalabilidade da solução, que é limitada se comparada a uma nuvem pública, isso é um contraponto à computação sem servidor, que tem como premissas a rápida elasticidade de aplicações e redução de gastos.

É possível encontrar múltiplas plataformas de *FaaS* na revisão da literatura, é importante considerar alguns aspectos na hora de escolher a plataforma ideal para dada necessidade. Na Tabela 6 são listadas as plataformas de código aberto mais

populares e suas características de linguagens suportadas, métricas de auto escalonamento, orquestrador de *containers* e acionador das funções.

A computação sem servidor apresenta um novo conceito e abstração da computação em nuvem e tais plataformas, sejam comerciais ou de código aberto, ainda estão evoluindo. É possível visualizar a evolução das plataformas e melhorias futuras que estão sendo desenvolvidas continuamente pelas equipes. De todas as plataformas não há claramente uma vencedora ou uma como modelo a ser seguido, uma boa abordagem para os próximos passos seria a criação de padronizações e melhores práticas para se desenvolver e usar tais plataformas *FaaS*.

Tabela 6 – Lista de plataformas *opensource*

| Plataforma | Linguagens Suportadas | Métrica de auto escalonamento | Orquestrador de *container* | Acionador de função |
|---|---|---|---|---|
| **Fission** | C#, Go, *Node.JS*, Java, Pearl, Python, PHP, Ruby, ambientes de execução customizados | CPU | Kubernetes | HTTP, evento, agendamento |
| **Kubeless** | C#, Go, *Node.JS*, Java, Python, PHP, Ruby, Ballerina, linguagens customizadas | CPU, QPS, métricas customizadas | Kubernetes | HTTP, evento, agendamento |
| **Open*FaaS*** | C#, Go, *Node.JS*, Java, Python, PHP, ambientes de execução customizados | CPU, memória, QPS | Kubernetes, Docker Swarm, orquestradores customizados | HTTP, evento |
| **OpenWhisk** | Ballerina, C#, Go, Java, JavaScript, Python, Ruby, PHP, Swift, ambientes de execução customizados | QPS | Não necessita de orquestrador | HTTP, evento, agendamento |

Fonte: Adaptado de PAAKKUNAINEN (2019).

Em plataformas de funções sob demanda, alguns fatores importantes para a escolha da plataforma, são eles:

- **Desempenho**: Comparar o desempenho entre plataformas *FaaS* de código aberto mostra-se relevante por cada plataforma pode performar diferentemente dependendo dos eventos ou da linguagem de programação usada. (MOHANTY, 2018; BALDINI, 2017) observou um tempo de resposta três vezes maior entre a plataforma *FaaS* mais lenta e a mais rápida em certas situações, o mesmo estudo mediu que, enquanto pequena, existem diferenças de desempenho entre diferentes tipos de gatilhos que desencadeiam eventos.

- **Comunidade de desenvolvedores**: Projetos *opensource* podem ter o seu desenvolvimento lento, é necessário que o projeto tenha larga escala e abrangência para que se tenha uma comunidade ativa, ou até mesmo empresas que auxiliam projetos a avançar. Plataformas como *Fission, Kubeless, OpenFaaS* e *OpenWhisk* possuem uma comunidade relativamente ativa.

- **Linguagem de programação**: As plataformas listadas na Tabela 6 interagem com várias linguagens de programação, que abrangem vários tipos de casos de uso. Quando não há suporte para uma linguagem em particular é possível utilizar ambientes de execução customizado, mas pode se tornar uma tarefa difícil e custosa fazer a manutenção desses ambientes. Com isso, as linguagens mais populares podem performar de formas diferentes, por exemplo *Python* que pode se mostrar mais rápida que outras linguagens de programação. O desempenho deve ser testado em diferentes linguagens de programação, a partir de diferentes plataformas de computação sem servidor (BALDINI, 2017).

- **Métricas de auto escalonamento**: Auto escalonamento está inerente a capacidade das plataformas de aumentar ou diminuir de tamanho, boas métricas de auto escalonamento permitem melhor escalonamento granular. Para o uso intenso de *CPU* é mais usual utilizar métricas de auto escalonamento baseados em uso de *CPU*, métricas de *QPS* (requisições por segundo) são mais úteis em ambientes de execução com entrada e saída (I/O) mais intensas. *Kubeless* e *OpenFaaS* possuem maior repertório de métricas de escalonamento.

- **Orquestrador de *container***: Os orquestradores gerenciam a infraestrutura de *containers* da aplicação. É imprescindível que a plataforma *FaaS* tenha suporte aos orquestradores já existentes para não adicionar maior complexidade a aplicação, *OpenWhisk*, por exemplo, não precisa de um orquestrador separado e *Fission*, *Kubeless* e *OpenFaaS* oferecem suporte ao *Kubernetes*.

- **Acionadores de funções**: Plataformas *FaaS* suportam três diferentes tipos de gatilhos de funções: *HTTP*, evento e agendamento. *HTTP* são requisições *HTTP* que no início de uma execução de função retornam uma reposta ao cliente *HTTP*. Eventos envolvem transmissão de filas de mensagens externas, esses eventos podem ser originados de qualquer aplicação que tenha a capacidade de adicionar uma nova mensagem à fila. Os eventos acionam as funções de forma assíncrona (vide Figura 2). Agendamento dispara um gatilho para acionar o evento em um momento pré-definido pelo desenvolvedor da aplicação, por exemplo uma vez ao dia ou uma vez na semana.

## 3.3 Comparação entre plataformas disponíveis e o desenvolvimento tradicional de software

As plataformas de computação sem servidor chamam a atenção de grandes provedores de serviços em nuvem e dos desenvolvedores de plataformas *serverless opensource*, com o principal motivo de fugir dos aprisionamentos dos provedores. Na Tabela 7 é mostrada uma comparação das plataformas *serverless* que mais chamam atenção em trabalhos acadêmicos reunidos no presente estudo. A comparação aborda características como por exemplo interação da plataforma com o desenvolvedor, se há *plugin* que integra a alguma plataforma de desenvolvimento (*IDE*), estilo da cobrança feita pelo provedor, limitação de tempo e tamanho de pacote, qual tamanho da disponibilidade de alocação de memória, como é possível monitorar, ou observar, as execuções e se há uma prateleira, ou *marketplace*, de funções pré-existentes já executadas anteriormente e compartilhadas na plataforma.

Tabela 7 – Comparação entre plataformas *serverless*

| **Caracte-rísticas** | ***AWS* Lambda** | **Azure Functions** | **Google Functions** | **OpenWhisk** | **Open*FaaS*** |
|---|---|---|---|---|---|
| **Interface do usuário** | CLI, API e GUI | CLI, API e GUI | CLI, API e GUI | CLI e API | CLI, API e GUI |
| **Desenvol-vimento em IDE** | VS e VSCode | VS e VSCode | Não disponível | X-Code e VSCode | Não disponível |
| **Cobrança** | Tempo de execução e Alocação de memória | Tempo de execução e memória consumida | Tempo de execução e memória alocada | Tempo de execução e memória alocada | Desconheci-do |
| **Limitação de tempo** | 900 segundos | 600 segundos | 540 segundos | 600 segundos | 30 segundos |
| **Limitação de tamanho de pacote** | 250 MB descomprimido ou 50 MB comprimido | Sem limite | 500 MB descomprimido ou 100 MB comprimido | 48 MB descomprimido | Desconheci-do |
| **Alocação de memória** | 128 MB a 10.240 MB | 1536 MB | 128 MB a 4.096 MB | 128 MB a 2048 MB | Desconhecido |
| **Observa-bilidade** | *AWS* Cloud Watch, *AWS* CloudTrail | Azure Application Insights | Google Cloud Operations | Ferramentas terceiras | Ferramentas terceiras |
| **Prateleira *serverless*** | *AWS Serverless* Application Repository | Azure Marketplace | Não disponível | Não disponível | Open*FaaS* Function Store |

Fonte: Adaptado de WEN et. al (2023).

O desenvolvimento tradicional de *software* diferencia-se em relação ao desenvolvimento baseado na tecnologia *serverless* em alguns aspectos importantes. O desenvolvimento tradicional pode se dividir em uma tangente *on-premises* e outra usando a computação em nuvem. Na Tabela 8 são mostradas características importantes no desenvolvimento de *software* tradicional e o desenvolvimento baseado em *serverless*. Na tabela, características como desempenho, implementação de funcionalidades, padrão de invocação, limitação de execuções, localidade da execução e maturidade são comparadas.

Tabela 8 – Comparação entre desenvolvimento de *software on-premises*, em nuvem e *serverless*

| **Características** | **Desenvolvimento *on-premises*** | **Desenvolvimento em *cloud*** | **Desenvolvimento *serverless*** |
|---|---|---|---|
| **Desempenho** | Sempre ativo, sem *cold start*, mas sem flexibilidade | Ativo sob demanda, sem cold start, com flexibilidade | Ativo sob gatilhos, com *cold start*, com flexibilidade |
| **Implementação de funcionalidades** | Implementação desde o momento zero, baixa eficiência | Implementação parcial desde o momento zero, eficiência média | Código baseado em eventos, utilização do BaaS simplificada, altamente eficiente |
| **Padrão de Invocação** | Chamada pelo lado do cliente | Chamada pelo lado do cliente | Evento baseado em gatilho |
| **Limitação de execuções** | Depende da capacidade do servidor | Depende da capacidade alugada | Limitações inerentes fixas |
| **Localidade da execução** | Localmente | Nuvem | Nuvem |
| **Maturidade** | Alta | Média | Baixa |

Fonte: Adaptado pelo autor de WEN et al. 2023.

- **Desempenho:** O ambiente de execução está sempre ativo em ambientes de desenvolvimento *on-premises*, o que faz com que as requisições da aplicação sejam respondidas imediatamente, em contrapartida, os recursos continuam sendo utilizados mesmo quando não há nenhuma requisição. Junto a isso existe o problema da flexibilidade dessas aplicações, a depender a quantidade de recursos computacionais que estão à disposição da aplicação no momento de muitas requisições, como é abordado em (INTERSOG, 2022) e (NETWORKINREVIER, 2022). Essa situação melhora ao migrar para o desenvolvimento em nuvem, que pode ter a flexibilidade sob demanda da aplicação, a elasticidade garante que independentemente do número de requisições, haverá recurso computacional suficiente para a resposta, mesmo que esse recurso sempre esteja ativo, aguardando

determinada requisição acontecer, o que se pode interpretar como um arranque a frio inexistente. O desenvolvimento *serverless* sofre com o arranque a frio, no qual a resposta a requisições depende de um evento que engatilha a execução do código, e esse tempo de preparação pode ser longo em comparação aos outros ambientes de desenvolvimento, existem alguns estudos tentam mitigar esse problema, como é visto em (AKKUS, 2018; CADDEN, et al. 2020; DAW, et al. 2020; OAKES, 2018; WANG, et al. 2019; ZUK; RZADCA, 2020).

- **Implementação de funcionalidades:** A implementação de funcionalidades no desenvolvimento de *software* em ambientes de servidores físicos precisa contar com um processo complexo de escolha da pilha de tecnologias a serem utilizadas, além de *frameworks* que precisam estar atualizados. Localmente é preciso preparar todo o ambiente para que determinada aplicação seja desenvolvida com a compatibilidade, versionamento e atualizações corretas. Em nuvem, o processo torna-se um pouco menos complexo, mas ainda sim necessitando da configuração do ambiente que vai receber a aplicação (NOVACONTEXT, 2022). A implementação em ambiente sem servidor permite aplicações desenhadas sob medidas para serem acionados baseado em um evento, funções que tem uma curta duração de execução, dessa forma o *backend* e o armazenamento, por exemplo, custam um menor tempo para o desenvolvedor.

- **Padrão de invocação:** Por ter um desempenho baseada em eventos pré-configurados, as aplicações *serverless* são invocadas automaticamente (JONAS, 2019; EYK, 2018). No desenvolvimento convencional, as invocações dependem das chamadas pelo lado do cliente, o que pode trazer maior complexidade para processos do servidor (ADZIC, 2017).

- **Limitação de Execução:** A definição da limitação de execuções em ambientes *on-premises* é muito incerta, pois depende muito da capacidade física de cada servidor. Em um ambiente de nuvem, o desenvolvedor tem flexibilidade de alocação de recursos para sua aplicação, ao passo que a limitação de execução acompanha a alocação de recursos (BONER, 2016). No desenvolvimento *serverless* existem uma série de limitações quanto ao *timeout* de execução de funções. Confinamento do tamanho da memória e armazenamento (EYK, et al. 2018).

- **Localidade da Execução:** A localidade da execução em aplicações *on-premises* é a mesma que a do desenvolvedor, já a localidade na nuvem e, consequentemente, *serverless* é executada de forma invisível ao desenvolvedor e depende da localidade na nuvem onde existem recursos suficientes para a execução, sob responsabilidade da disponibilidade do provedor do serviço em nuvem, pode não ter necessariamente a melhor latência (NEWMAN, 2021) e (TAIBI, 2020).

- **Maturidade:** A maturidade em ambientes *on-premises* é considerada alta pois funcionalidades de teste e *debugging* de aplicações podem ser desenvolvidas e avaliadas dentro do próprio ambiente local de desenvolvimento. Hoje em dia existe uma variedade de ferramentas que desempenham essas tarefas e estão amplamente distribuídas na comunidade de engenharia de *software*. Por ser uma tecnologia já estudada a muitos anos, ambientes de desenvolvimento em nuvem já possuem suas próprias ferramentas que ajudam a automatizar o processo de teste de uma aplicação, geralmente os próprios provedores de serviços em nuvem já trazem uma suíte de ferramentas que ajudam o desenvolvedor (A3LOGICS, 2022) e (COTRONEO, 2015). Já a emergente tecnologia *serverless* ainda carece de ferramentas de teste e *debug* (LENARDUZI; PANICHELLA, 2020), uma das razões é pelo funcionamento baseado em eventos, largamente distribuído e execução mascarada, ou invisível, para o desenvolvedor, o que torna o acompanhamento do fluxo de execução mais difícil de acompanhar e testar.

## 3.4 Microsserviços e suas características

Os microsserviços oferecem o desenvolvimento de aplicações que se baseia em subdividir uma aplicação em pequenos serviços independentes entre si, que trabalham em conjunto como um único sistema. Microsserviços se apresentam também como um paradigma eficaz para aplicações distribuídas e elásticas, sendo amplamente empregada na indústria e academia.

O desenvolvimento de microsserviços baseia-se na decomposição de uma aplicação em um conjunto de serviços, e cada um dos serviços é responsável por uma funcionalidade específica (BRINZA, 2022). Permite dessa forma o teste, desenvolvimento e implantação de forma independente, facilitando gerenciamento de determinada solução.

Um exemplo disso é um site de vendas, que pode ser modularizado em vários serviços, um sendo responsável pelo *login* do usuário, outro pelo cadastro, outro pelo pagamento de produtos, outros pelo cadastro de produtos. O exemplo permite que parte da aplicação continue funcionando mesmo quando as outras partes, ou módulos, deixem de funcionar. Esses módulos podem ser escritos em diferentes linguagens de programação e conversam entre si por meio de *API*. Módulos podem ser atualizados, substituídos, modificados sem refletir na aplicação como um todo.

Segundo o artigo de (BUSHONG, et al. 2021), os Microsserviços contam com vantagens como escalabilidade, flexibilidade, agilidade no desenvolvimento, pontos que se também se evidenciam na tecnologia *serverless*. Outros trabalhos, como de (TAIBI, et al. 2022) e (BARESI; GARRIGA, 2019) apontam também como uma boa alternativa para aplicações complexas que necessitam de escalabilidade para lidar com grande quantidade de tráfego de dados e substituição de aplicações hospedadas em monólitos. Da mesma forma que a computação sem servidor, o uso de microsserviços em nuvem permite o desenvolvimento escalável e flexível, com recursos computacionais alocados dinamicamente de forma invisível para o desenvolvedor e para o usuário, permitindo assim a elasticidade necessária para aplicações em nuvem heterógenas.

Por se tratar de serviços independentes interconectados, há a tolerância a falhas e resiliência de serviços nos Microsserviços (BUSHONG, 2021). A arquitetura de Microsserviços permite o isolamento de falas quando ocorrem, evita-se que uma falha afete outras partes da aplicação ou até mesmo a aplicação como um todo, a modularidade oferecida pelos Microsserviços reforça a premissa de implementação de recuperação e redundância, fortalecendo a resiliência da aplicação.

As aplicações que usam os microsserviços também apresentam desafios parecidos com o paradigma *serverless*, a segurança de serviços isolados e orquestração de tais serviços devem ser mitigados ao utilizar-se da tecnologia.

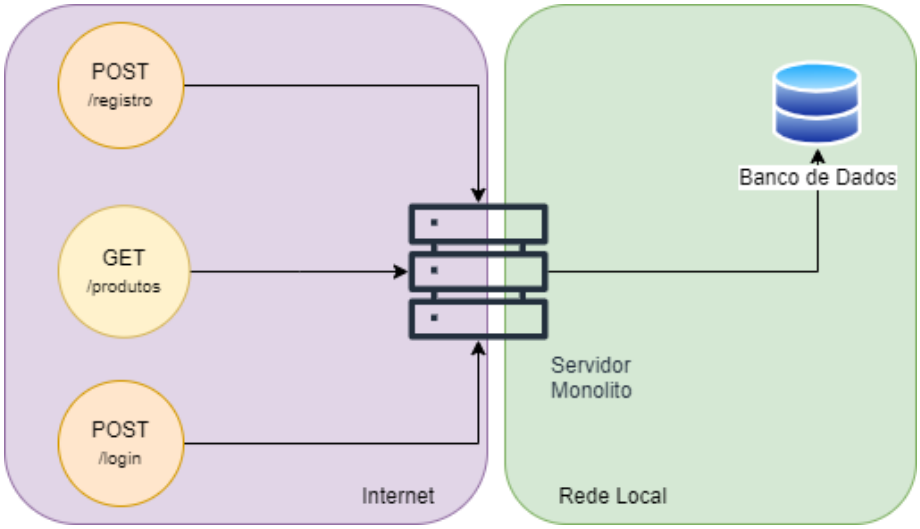
### 3.5 Monólitos, Microsserviços, *Containers* e *Serverless*

Várias aplicações podem ser desenvolvidas com diferentes objetivos e diferentes funcionalidades. Historicamente, as aplicações foram desenvolvidas e hospedadas em servidores chamados de monólitos, computadores com grande

capacidade de *hardware* hospedados localmente em empresas. Os avanços tecnológicos possibilitaram a migração para a computação altamente distribuída.

O modelo de computação de monólito, ou monolítica, é adequada para aplicações de pequeno porte. No entanto, algumas aplicações começam pequenas e vão ganhando implementações com o tempo e consequentemente aumentando de tamanho, o que se torna uma desvantagem para este modelo, que tem como premissa ter a execução, armazenamento e toda codificação em um único computador (KAZANAVICIUS; MAZEIKA, 2019). Na Figura 11 é ilustrado um exemplo de entrada e saída de dados em uma aplicação monolítica. É possível observar que o mesmo servidor interage com requisições *HTTP*, armazena, executa, manipula dados sozinho, de forma não distribuída.

Figura 11 – Exemplo de computação em monólitos




Fonte: Desenvolvido pelo autor (2023)

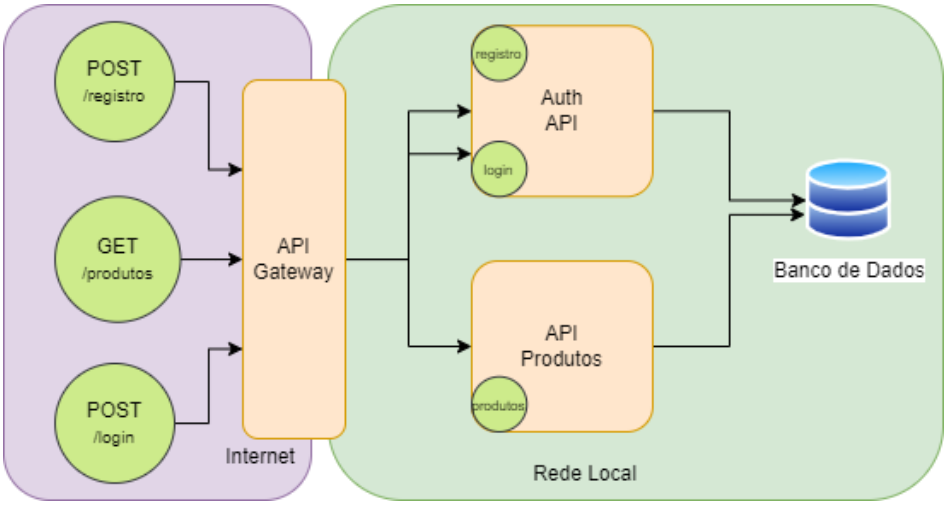
Quando a complexidade da aplicação e número de acessos simultâneos aumenta, bem como a necessidade de manutenção do código, torna-se difícil implementar funcionalidades sem comprometer a disponibilidade. Além disso, segundo (NEWMAN, 2021), a escalabilidade é comprometida quando há sobrecarga na aplicação, o monólito é deficiente no que tange a elasticidade vertical ou horizontal de uma determinada aplicação.

Como uma forma de mitigar problemas advindos dos monólitos, o conceito de microsserviços foi criado com o objetivo de adicionar maior granularidade à computação moderna. Os microsserviços passaram a atender funcionalidades de um

sistema, não o sistema como um todo, como é ilustrado na Figura 12. Os microsserviços podem ser comunicar através de protocolos *RPC, REST, SOAP* e podem utilizar de *API Gateways* para manter uma harmonia entre a execução deles, de forma orquestrada, sem perder a independência de cada serviço.

Os serviços podem ser desenvolvidos e escalar individualmente, dependendo do nível de carga de cada um. Seguindo o exemplo da Figura 12, se o serviço de *login* de um sistema receber múltiplas requisições por segundo, apenas o módulo de *login* pode aumentar de tamanho, isso serve para qualquer outro microsserviço.

Figura 12 – Exemplo de computação em Microsserviços



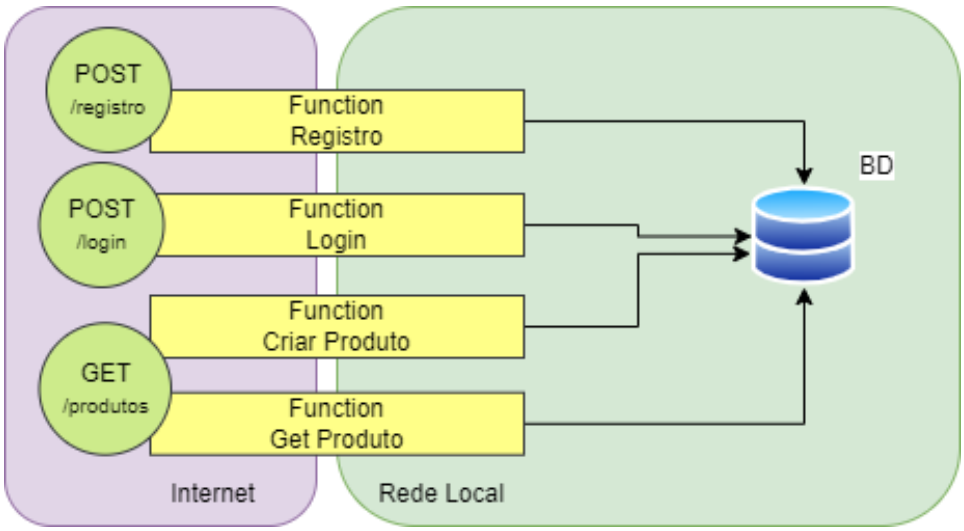

Fonte: Desenvolvido pelo autor (2023)

Microsserviços apresentam além da escalabilidade, facilidade na flexibilidade de cada função, à luz das necessidades do negócio, resiliência a falhas por conta da independência dos microsserviços e facilidade de desenvolvimento, pois são pequenos pedaços de código, semelhante às funções *serverless*. Entretanto, os microsserviços apresentam desafios como alta complexidade de gerenciamento, devida a alta distribuição de serviços.

Na Figura 13 ilustra-se a ideia central e principal diferença entre a computação sem servidor e os demais paradigmas da computação em nuvem e de alto desempenho. De forma a simplificar ainda mais os processos das aplicações, é possível com este modelo criar funções que executam partes específicas de código para determinada funcionalidade do sistema, execução baseada em gatilhos acionados pelo usuário ou pelo programador. No exemplo mostrado na Figura 13,

funções pré-estabelecidas são executadas quando um novo *login* ou cadastro é realizado, um novo produto é criado ou consultado.

Figura 13 – Exemplo de computação *serverless*




Fonte: Desenvolvido pelo autor (2023)

A ideia de dividir uma aplicação, ou parte dela, em funções, promove a economia de recursos computacionais e de custos dentro da computação em nuvem onde determinada aplicação pode ser hospedada.

Outro conceito que se refere o isolamento de processos, portabilidade e minimização da complexidade dos ambientes é o de *containers*, amplamente usado na indústria há dez anos, segundo (RANDAL, 2020). O conceito de dividir em *containers* é a possibilidade de isolar ou encapsular *kernel*, processos e recursos computacionais, onde cada *container* tem um número identificador único que, apesar de isolado, pode trabalhar em conjunto com outros *containers*, de forma orquestrada.

O *container* agrega a aplicação e suas dependências facilitando a implementação em qualquer ambiente, trazendo uma abstração a mais no contexto da nuvem e máquinas virtuais. Segundo (JAMBUNATHAN; YOGANATHAN, 2018), é interessante destacar possibilidades como a solução de problemas de conflitos de dependências: se módulos de aplicações rodam em diferentes versões de uma linguagem de programação, basta separar em *containers*; instalar uma nova aplicação em um novo ambiente pode ser simples se utilizado *containers*.

Em relação a orquestração de diversos *containers*, é importante automatizar o esforço operacional para executar aplicações conteinerizadas, implementando um

ciclo de vida de três etapas de um *container*, discutido por (KHAN, 2017): Criação ou instanciação com base em configurações pré-estabelecidas, escalonamento de forma horizontal com base em métricas pré-definidas e rede, que se baseia na adição automática em uma rede virtual previamente configurada.

Diversos estudos comparam as possibilidades das abstrações, a evolução é constante e os cenários possíveis são ilimitados, podemos colocar em perspectiva:

- ***Containers* e *Serverless***: Pode-se entender que a computação sem servidor é uma evolução dos *containers*, por conta da perspectiva de melhor eficiência de funções *serverless* em relação a implementação de *containers*, impactando no tempo e custo de uma aplicação. *Containers* reduzem a necessidade de gerenciamento de máquinas virtuais e possibilitam a escalabilidade, *serverless* dá um passo além e elimina a necessidade de gerenciar ainda mais a infraestrutura e *runtimes*, por exemplo, bastando apenas entregar o código a ser executado à plataforma *serverless*.

*Serverless* não exclui *containers* no desenvolvimento de aplicações, é possível até encapsular *containers* dentro de funções e conteinerizar funções. *Serverless* é uma técnica para desenvolver aplicações distribuídas e *container* é uma boa plataforma para construir essas funções. É possível visualizar que muitos benefícios podem ser extraídos entre a combinação dos dois.

- ***Containers* e Microsserviços**: *Containers* são leves e portáteis, podem ser implementados e provisionados com facilidade em qualquer ambiente, seja *on premises* ou na nuvem, é consideravelmente escalável e possibilita diversas linguagens diferentes em seu desenvolvimento. Da mesma forma que uma máquina virtual, possui seu ciclo de vida e é suportado por todas os provedores de computação em nuvem.

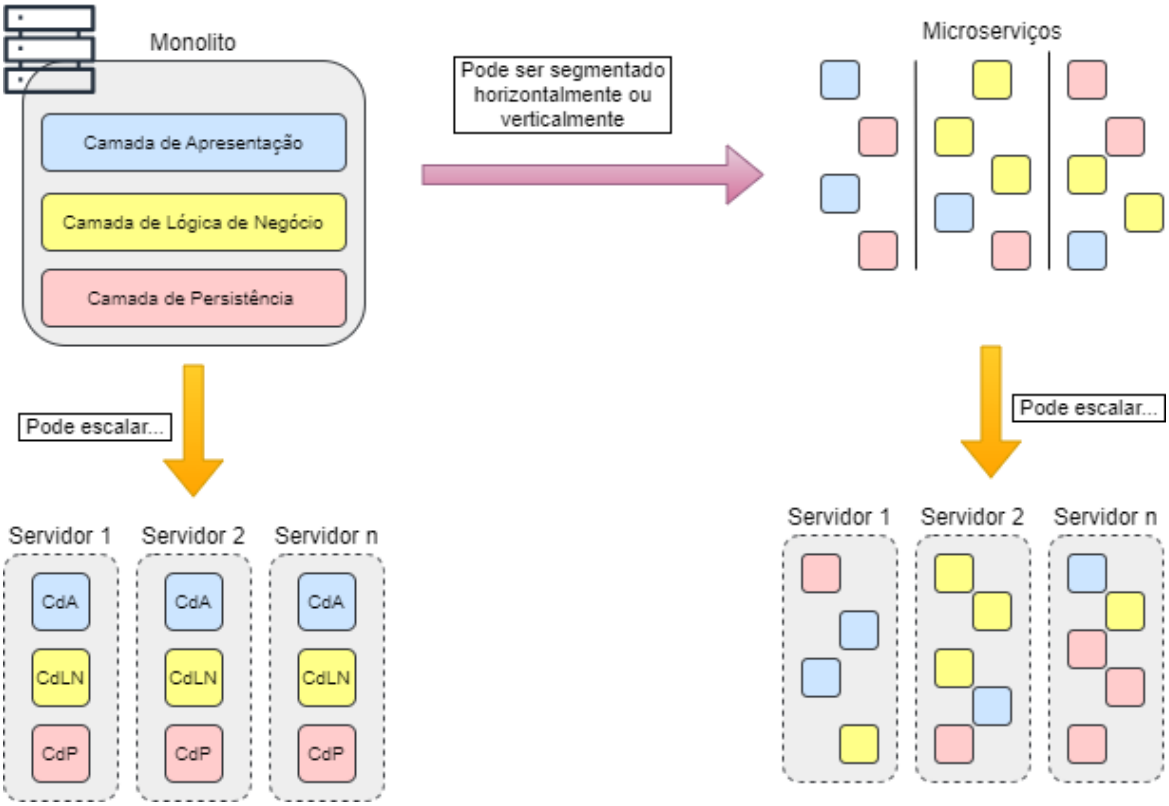
Os microsserviços são promovidos pela possibilidade e desafio de converter aplicações monolíticas, tem a premissa de decompor uma aplicação em diversos pequenos serviços integrados, microsserviços estes que se comunicam de forma síncrona ou assíncrona. Esses microsserviços são isolados e fáceis de escalar, além de tolerarem falhas, focado em aplicações distribuídas.

- **Microsserviços e *Serverless***: Um microsserviço é um pedaço de uma grande aplicação que foi “quebrada” em pequenos e modulares serviços. Sobre o teste, construção e implementação, traz benefícios para o desenvolvimento de aplicações de forma mais ágil apesar de ser complexo e desafiador quanto a migração de monólitos. *Serverless* é uma camada maior de abstração já constatada em microsserviços, onde o foco é o desenvolvimento de aplicações sem a preocupação nas camadas de segurança, escalabilidade e disponibilidade dos servidores. Promove dessa forma a melhor economia em relação ao *hardware* que suporta a aplicação, melhor gerenciamento dos recursos computacionais, sejam eles sendo acionados de forma elástica.

Computação sem servidor também é um facilitador de aplicações em microsserviços, permite o uso de uma infraestrutura e recursos computacionais baseados em eventos, controlado com base na necessidade da aplicação. Microsserviços e computação *serverless* podem complementar um ao outro, *serverless* permite que microsserviços sejam aplicados e executados com o mínimo de esforço possível por parte do desenvolvedor.

Para sintetizar as possibilidades, na Figura 14 demonstra-se como paradigmas da computação foram aperfeiçoando-se e evoluindo para possibilidades que são possíveis hoje. Todos os modelos apresentados na imagem são utilizados em larga escala e atendem a determinadas e especificas demandas, dependendo da aplicação. Na Figura 14 tem-se um exemplo da computação monolítica que pode ser escalada vertical ou horizontalmente, camadas de persistência, lógica de negócio e apresentação podem ser segmentadas, distribuídas, se transformar em *containers*, microsserviços ou até mesmo funções.

Figura 14 – Desenvolvimento de Microsserviços e Funções




Fonte: Desenvolvido pelo autor (2023).

# CAPÍTULO 4 – DESENVOLVIMENTO

Neste capítulo elucida-se a aplicação prática em um projeto de serviço *web*, quais tecnologias são usadas e o funcionamento do projeto.

É possível encontrar sucesso no trabalho de autores que migraram serviços monolíticos para Funções Como Serviço, constatações interessantes em relação ao desempenho das aplicações *web*, facilidade na implementação de código, custo reduzido e mitigação do *cold start* são dados a serem observados no presente trabalho.

## 4.1 Arquitetura da aplicação *web*

A aplicação *web* utilizada no desenvolvimento é baseada na linguagem de programação *JavaScript*, com um ambiente de execução e biblioteca, baseado em eventos assíncronos nomeado *Node.JS*. Essa tecnologia é amplamente adotada no desenvolvimento de aplicações *web* "*server-side*", ou seja, uma programação ao lado do servidor que o ele entende (*NODE.JS*, 2023).

Além disso, o *Node.JS* é baseado em eventos, é escalável, por conta da possibilidade de processamento simultâneo de várias requisições e, por conta disso está em conformidade com as premissas já vistas na computação sem servidor.

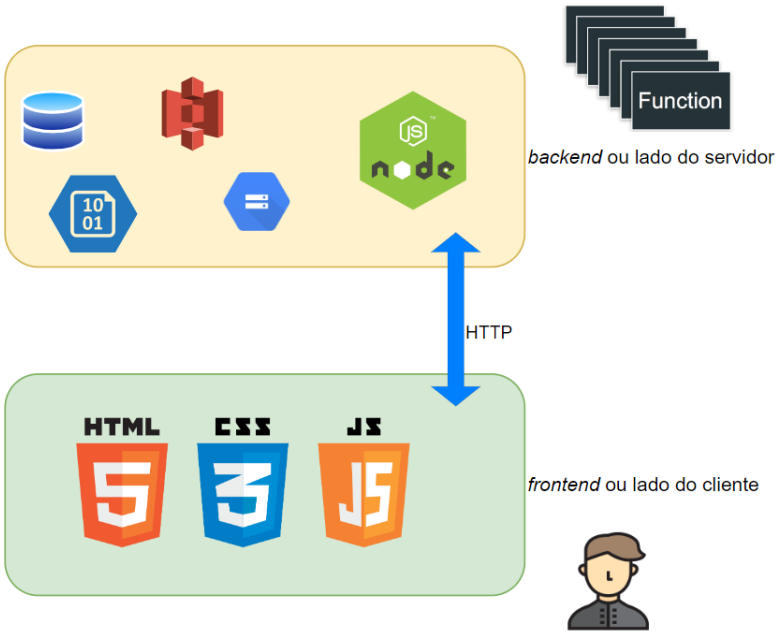
O foco do *Node.JS* no contexto da aplicação é agir como um *backend* do sistema, servir como um interceptor de requisições HTTP e HTTPS e respondê-las. Com isso é possível criar REST APIs, microsserviços, *containers* e funções sob demanda.

Há a possibilidade, com o *Node.JS*, integrar outros serviços de forma *stateful*, quando há a necessidade de uma instância conversar com um banco de dados, ou criar uma relação com um *socket* carregando um "estado". Outra forma é manter uma comunicação *stateless*, ou a independência de manter um estado, quando não há uma garantia de relação e integridade com outros objetos, segundo (PAAKKUNAINEN, 2019), o que mais pactua com o princípio de funções como serviço – *FaaS*.

Na Figura 15 é ilustrada a arquitetura de um serviço *web* usando *Node.JS*. É possível verificar algumas possibilidades quanto ao seu uso *stateless* ou *stateful*, a imagem representa a possibilidade de interação entre o ambiente *Node.JS* e um banco de dados (*stateful*) ou até mesmo a interação dele com serviços de

armazenamento em nuvem como o *Amazon S3*, *Azure Blob Storage* e *Google Cloud Storage* (*stateless*). Todo o pacote *backend* pode ser alocado em *containers*, ou acionar uma função como serviço.

Figura 15 – Arquitetura de um serviço *web* usando *Node.JS*




Fonte: Desenvolvido pelo autor (2023).

A aplicação *web* necessita ser facilmente elástica para aguentar diferentes cargas de estresse e fácil de ser manuseada, para que o time de programadores que fazem a manutenção do código possa entender e implementar código. Por essa razão, é interessante pensar em implementar a computação sem servidor para parte da aplicação, focada na arquitetura do *backend*.

A aplicação consiste em um *frontend* escrito pela trindade *HTML5*, linguagem de marcação, base para construção de páginas *web* interpretadas pelos navegadores de *Internet* que possibilita a criação e publicação de conteúdo como textos, imagens, formulários, entre outros elementos; *CSS3*, linguagem de estilos, que define formatos estéticos e visuais dentro de uma página *web*, possibilita efeitos visuais para textos e imagens e padrões de *layout* dessas páginas e *JavaScript*, linguagem de comportamento dos conteúdos dinâmicos, controles de mídia e interatividade da página *web* com o usuário.

A linguagem *JavaScript* tradicionalmente é utilizada no *frontend*, linguagem de *script web* do lado do cliente. O *Node.JS* estende essa capacidade do *JavaScript* para

além do *frontend*, possibilita ampliar a arquitetura *web* para o *backend*, do lado do servidor, assim como o paradigma *serverless* da computação em nuvem.

O serviço a ser testado consiste em uma única página *frontend* para um *backend*, que retorna um *REST API* para o *frontend*. Todos os dados inseridos no *frontend* são trazidos ao *backend* por meio de uma *API*. *Backend* e *frontend* são independentes entre si. Seu funcionamento ocorre da seguinte forma:

a) O usuário do sistema insere informações de cadastro e faz *upload* de uma imagem, que posteriormente será utilizada para autenticação desse usuário;
    i. A imagem salva pode estar em diferentes tamanhos e até mesmo diferentes formatos, por conta disso, o sistema aceita apenas tamanho máximo de 6 *MB* e formatos .*PNG* e .*JPG*.
b) A imagem capturada é salva em um diretório com a nomenclatura única do usuário;
    i. Um módulo do *backend Node.JS* chamado "*Sharp*" processa a imagem e a redimensiona para que pese menos no armazenamento do servidor.
c) Informações do usuário são enviadas para um banco de dados;
    i. Essa etapa do processo não foi abordada pelos testes e implementações do presente trabalho de dissertação.
d) O usuário utiliza o sistema com as suas credenciais, suas informações salvas e foto de perfil são retornadas para o uso do sistema.

Os passos previstos na execução do *backend* da aplicação, mais especificamente o processo de redimensionamento de imagens, dão luz a possibilidade de usar funções da computação sem servidor (*FaaS*) para executar o código *Node.JS* baseado em evento, o *upload* de uma imagem do usuário para um diretório qualquer.

Para alinhar os testes e obter resultados comparativos de desempenho da aplicação, o processo *b-i* da subseção 4.1 funcionará como um evento que aciona o gatilho de uma função executada em um provedor de computação sem servidor na nuvem. A investigação contempla a execução da parte de redimensionamento de imagem usando a biblioteca *Sharp* do *Node.JS* da aplicação *web* nos parâmetros de

computação em nuvem: Implementação da biblioteca e execução do código em três provedores de computação *serverless* em nuvem *Amazon AWS*, *Google Cloud Platform* e *Microsoft Azure*. Primeiro teste e comparativo engloba a execução e medição em um *container* na nuvem, atuando como um *PaaS* e comparar ao funcionamento de uma função *serverless FaaS*, em sequência, as funções dos três provedores serão comparadas.

As métricas que serão usadas no trabalho são:

i. Comparar desempenho de *containers* e *serverless*, nos provedores de computação em nuvem, acerca do tempo de execução das funções sem servidor. "*Quanto o tempo de execução vai impactar a aplicação? Será mais interessante implementar containers ou funções?"*

*ii.* Comparar provedores *serverless* em relação ao custo monetário da execução da aplicação. "*Qual provedor terá melhor custo para essa aplicação específica?"*

- Em ambos os casos, dois testes serão conduzidos: um teste de carga incremental e outro teste de pico de carga, de tráfego não convencional sobre as aplicações usando a ferramenta *Apache JMeter*.

## 4.2 Testes de desempenho

A primeira plataforma *serverless* a receber as requisições é *AWS Lambda*, seguida da *Azure Functions* e por fim *Google Functions*. Todos os testes foram executados em duas regiões diferentes, uma com melhor latência em relação ao computador que conduz os testes e outra região com melhor custo-benefício financeiro.

### 4.2.1 *AWS Lambda*

Em relação a plataforma *AWS*, a região que apresenta a melhor latência é São Paulo (*sa-east-1*), com aproximadamente 60ms de tempo de resposta, em contrapartida não é a região com o melhor custo financeiro. No que tange a região

com melhor custo-benefício para os testes, Virginia (*us-east-1*), no Leste dos Estados Unidos apresenta latência de 190ms.

Como linguagem de programação utilizada, a plataforma Lambda oferece *runtime* para *Node.JS* com diferentes *SDK* para *JavaScript*. Na Tabela 9 mostram-se os *runtimes* suportados.

Tabela 9 - *runtimes* do *Node.JS* no *AWS Lambda*

| ***Node.JS*** | | | | |
|---|---|---|---|---|
| **Nome** | **SDK** | **S. O** | **Arquitetura** | **Ciclo de Vida** |
| *Node.JS* 20 | 3.362.0 | Amazon Linux 2023 | X86_x64 e ARM64 | Não anunciado |
| *Node.JS* 18 | 3.362.0 | Amazon Linux 2 | X86_x64 e ARM64 | Não anunciado |
| *Node.JS* 16 | 2.1374.0 | Amazon Linux 2 | X86_x64 e ARM64 | 12 junho de 2024 |
| *Node.JS* 14 | 2.1374.0 | Amazon Linux 2 | X86_x64 e ARM64 | 27 novembro de 2023 |

Fonte: Desenvolvido pelo autor (2023)

No momento da criação da função é possível selecionar o *runtime*. Neste caso, a versão 18 é usada. O *console* da plataforma cria uma função de arquivo único nomeado "*index.mjs*", que possibilita edição e adição de código.

O arquivo "*index.mjs*" exporta a função chamada *handler* que recebe um objetivo de evento e um objeto de contexto. Essa é a função que o *Lambda* chamada quando a função é invocada. O tempo de execução da função obtém eventos de invocação e passa para o *handler*. Na configuração da função, o valor do *handler* é "*index.handler*". Quando a função é salva, o *Lambda* cria um pacote de implementação .*zip*. Na Figura 16 ilustra-se um exemplo da definição do *handler*.

Figura 16 - Definição do *handler*

```
exports.handlerName = function(event, context, callback) {
    ...
    // callback parameter is optional
}
```

Fonte: Desenvolvido pelo autor (2023).

Toda função conta com um objeto de contexto, que permite a interação entre a função e a plataforma de desenvolvimento da *AWS*, trazendo assim informações pertinentes à execução da função. O objeto de contexto conta com:

- ***AWS RequestID***, usado para rastrear invocações de funções *Lambda*, é um ID único para relatório de erros e contato com suporte da *AWS*;
- ***Duration***, contém o tempo de duração de execução da função, do momento em que foi acionada por meio de gatilho, até sua finalização;
- ***Billed Duration***, representa a duração que será cobrada do usuário ao final do mês;
- ***Memory Size***, indica o tamanho de memória alocada para a execução da função;
- ***Max Memory Used***, demonstra o máximo de memória usada. Há a particularidade de que, invocações subsequentes tendem a ir aumentando o uso máximo de memória, para obter execuções mais rápidas;
- ***Init Duration***, indica o tempo de inicialização da função, também conhecido como o *cold start*.

O *Node.JS* possui um modelo de *loop* de eventos único que faz seu comportamento de inicialização seja diferente de outros linguagens. Utiliza-se de um modelo de entrada e saída não bloqueante (*non-blocking I/O*), ou seja, suporta operações assíncronas.

Este modelo permite ao *Node.JS* desempenhar de forma eficiente para a maioria das cargas de trabalho, por exemplo: Se uma função faz uma chamada de rede, essa solicitação poderá ser designada como uma operação assíncrona e colocada em uma fila de retorno de chamada, ou *callback*. A função pode continuar o processamento de outras operações da pilha de chamadas principais sem ser

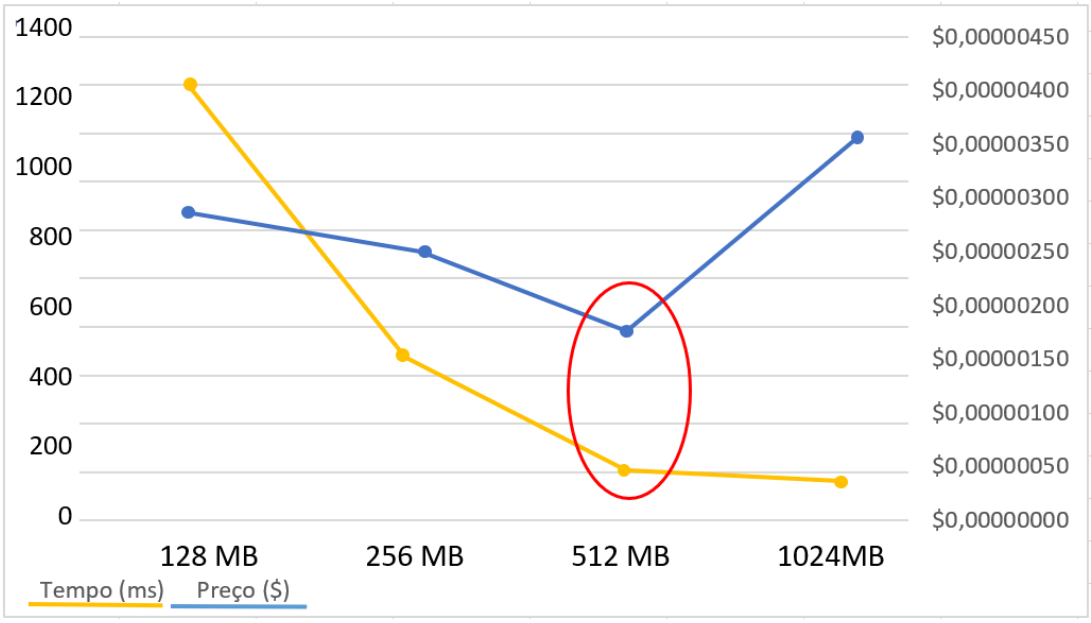
bloqueada enquanto aguarda o retorno da chamada de rede. Assim que a chamada de rede for concluída, seu retorno de chamada será executado e removido da fila de *callback*.

A arquitetura provisionada pode variar em relação à quantidade de memória alocada para a execução da função, de 128 MB à 1024 MB, quantidade de armazenamento temporário alocado, de 128 MB à 1024 MB, duração de execução em cada solicitação, em milissegundos e, por fim, a arquitetura do computador que executará a função, arquitetura ARM ou x86.

Para encontrar o melhor *setup* para execução, que consiste em encontrar o melhor *hardware* pelo melhor preço e o melhor tempo de execução para as funções, testes usando a função *Node.JS* foram aplicados na *AWS Lambda* e o preço a se pagar, em dólares (USD) foi coletado.

Na Figura 17 ilustra-se que o melhor balanço entre o tempo de resposta-custo-memória foi 512 *megabytes* alocados para cada execução, além de um único processador virtual. Paralelamente a isso, alocação em *container* manteve-se com as mesmas configurações. Destaca-se a importância de se manter as menores latências e custos possíveis, com a maior memória possível.

Figura 17 – Encontrando o melhor *setup* para execução *serverless*




Fonte: Desenvolvido pelo autor (2023).

Para melhor execução possível, foi necessário reproduzir testes aleatórios em configurações diferentes de execução da plataforma. Foi constatado um melhor custo-benefício, no que tange o uso de memória, com o emprego de 512 MB de RAM para execução. Em relação ao tempo, em milissegundos, de 200 ms. Este balanço proporciona um melhor dimensionamento entre custo monetário e tempo de CPU alocada para execução.

A plataforma *Lambda* permite alocar diferentes quantidades de memória RAM e tempo de execução, em milissegundos, de CPU para a função. A quantidade de RAM alocada impacta a quantidade de tempo de CPU e largura de banda que a função recebe. Simplesmente selecionando a menor quantidade de recursos computacionais pode não ser a melhor opção, pois pouca quantidade de memória adiciona latência à aplicação. É necessário encontrar uma melhor estratégia para cada função.

A precificação da plataforma é contabilizada em incrementos de 100 milissegundos de execução, o que se faz necessário economizar na alocação de recurso computacional.

Com isso, também é observado que o desempenho da função melhora logaritmicamente ao passo que alocação de recursos aumenta, com o teste de execução em diferentes configurações é possível encontrar o melhor cenário. O processamento é contabilizado do momento que a função é acionada, até o momento em que finaliza sua execução. O custo financeiro atrelado a ela está ligado a quantidade de memória alocada, juntamente com alocação de poder de CPU proporcionalmente provisionada para execução, aumento de memória consequentemente leva ao aumento de CPU, o que faz o custo aumentar também.

Todas as funções executadas podem ser analisadas posteriormente por meio da funcionalidade *CloudWatch Logs*, onde toda invocação gera um *report*, como ilustra-se na Figura 18.

Figura 18 - *Report* do *CloudWatch* em função na região *us-east-1*

```
REPORT RequestId: dedb4678-f424-4826-a929-0a49ee68309a
Duration: 10.22 ms Billed Duration: 11 ms Memory Size:
512 MB Max Memory Used: 82 MB Init Duration: 257.61 ms
```

Fonte: Desenvolvido pelo autor (2023).

No exemplo encontrado na Figura 18, a duração da execução da função foi de 10.22 milissegundos, a duração a ser cobrada é de 11 milissegundos, alocação de memória de 82 MB, tempo de inicialização da função de 257.61 milissegundos. É notável a diferença entre o *Init Duration* (acionamento do gatilho) e a *Duration* (execução da função). A diferença de mais de 240 milissegundos se dá pela região da execução da função (teste feito em *us-east-1*) e do *cold start*.

Na Figura 19, ilustra-se a mesma função, com a diferença de ser executada na região de São Paulo (*sa-east-1*):

Figura 19 - *Report* do *CloudWatch* em função na região *sa-east-1*

```
REPORT RequestId: c84d248d-b518-4719-ba2e-6f93bf9ece26
Duration: 3.79 ms Billed Duration: 4 ms Memory Size:
512 MB Max Memory Used: 80 MB XRAY TraceId: 1-
657a0602-1b8ed40d44fcbd8f2a707387 SegmentId:
46007ebf696af1bc Sampled: true
```

Fonte: Desenvolvido pelo autor (2023)

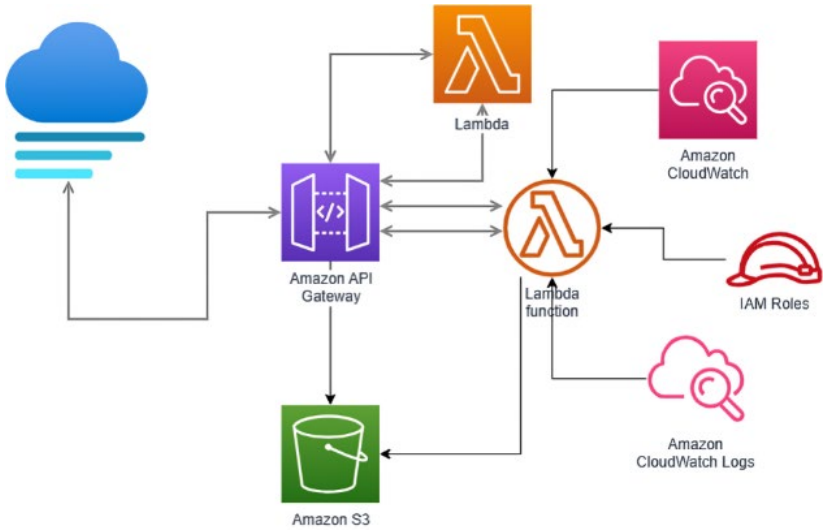
No *report* presente na Figura 19, a duração é de menos de 4 milissegundos e o tempo de inicialização não é exibido. Algumas informações se mostram diferentes dependendo da região onde são executadas as funções.

Uma funcionalidade presente na plataforma *Lambda*, e em outros produtos da *AWS*, é o *CloudWatch,* uma ferramenta de *DevOps*. A funcionalidade acrescenta gráficos, métricas e matemática as funções executadas e coletadas. Toda e qualquer função é monitorada por essa funcionalidade. Com ela é possível monitorar e gerar gráficos relacionados a quantidade de invocações da função, duração de execução, porcentagem de erros, execuções concorrentes, entre outros. Nos testes feitos neste trabalho, informações como quantidade de invocações, duração e gasto de recursos computacionais foram usados.

A arquitetura da aplicação no provedor *AWS* é apresentada na Figura 20, reúnem-se as diversas ferramentas que acompanham a execução da função *Lambda* e a ferramenta *CloudWatch*. *IAM Roles* permite delegação de acesso à usuários e outros serviços aos recursos executados, as credenciais podem ser temporárias ou permanentes. *Amazon API Gateway* torna-se fundamental em aplicações modernas

e permite que desenvolvedores criem, publiquem e protejam APIs, de forma a possibilitar o gerenciamento de múltiplas requisições de projetos, em qualquer escala. Por fim, *Amazon S3* permite armazenar desde cópias de segurança do projeto, até as imagens utilizadas nas aplicações. É importante elucidar que, ao criar uma função *serverless* no *Lambda*, automaticamente essas aplicações auxiliadoras são criadas, criando assim uma arquitetura pronta para ser escalável e altamente controlável.

Figura 20 - Diagrama da estrutura da aplicação na plataforma *AWS Lambda*



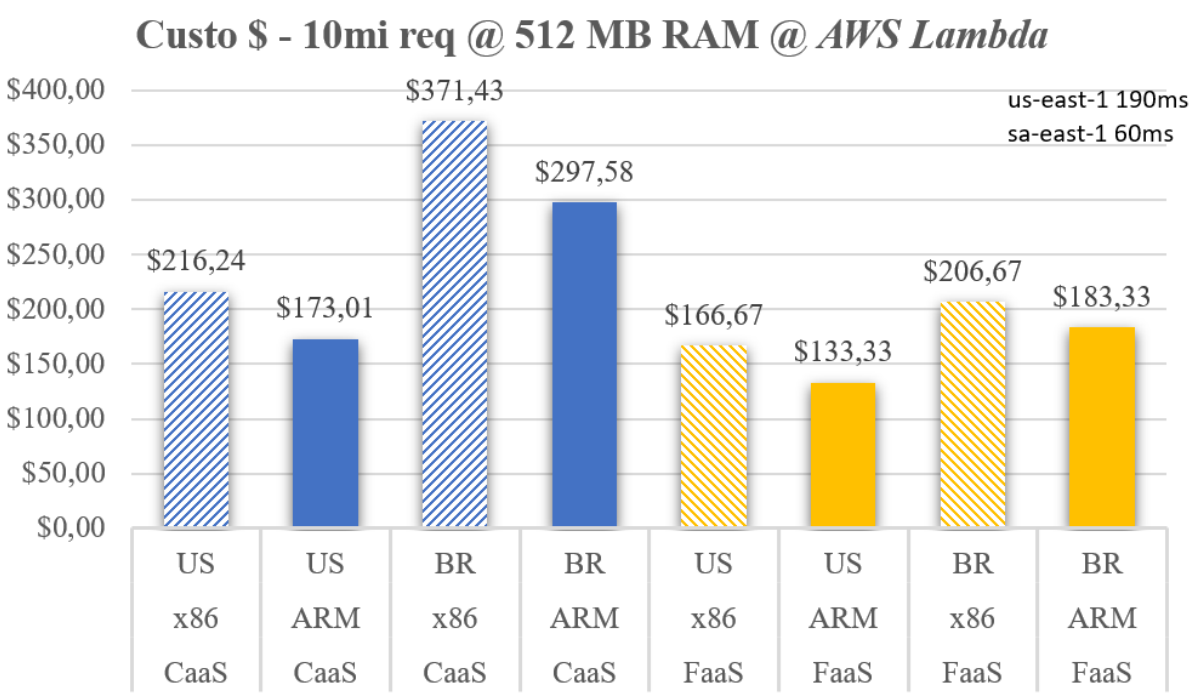

Fonte: Desenvolvido pelo autor (2024)

O diagrama representa a arquitetura da aplicação *serverless* utilizando o serviço do provedor *AWS*. O ícone da nuvem representa a interação com o *frontend* da aplicação, que aciona o gatilho da aplicação. *Amazon API Gateway* atua como ponto de entrada para a aplicação, gerenciando e roteando requisições do *frontend* para a função a ser executada, possibilita ocorrer paralelamente também a execução de múltiplas funções, o *gateway* se comunica com a função *Lambda* para processar as requisições. *AWS Lambda* são as funções *serverless* que executam o código em resposta a eventos, ou gatilhos. *Amazon S3* tem sua participação como serviço de armazenamento de objetos da *AWS*, é usado para armazenar arquivos de imagem, que a aplicação pode acessar e manipular, a sua interação direta com o *gateway* se dá pela rápida obtenção dos dados armazenados e envio para o *frontend*. *Amazon CloudWatch* é o serviço de monitoramento e gerenciamento de registros, coleta dados e armazena *logs* das funções executadas para análise e *debugging*, é possível criar alertas baseados em métricas de desempenho pré estabelecidas. *IAM Roles*, por fim,

gerencia as permissões e o acesso aos recursos da *AWS*, define quais ações as funções *Lambda* e o *gateway* podem realizar e em quais recursos da *AWS* podem operar.

Com a premissa de custo granular inerente ao uso, na Figura 21 demonstra-se como o custo monetário deu-se com base no comportamento de execução de aproximadamente 10 milhões de execuções, da forma de funções *serverless* e *containers*. É constatado que a execução na região us-east-1 é ligeiramente mais barata que a mesma execução na região sa-east-1, além disso é visível que arquiteturas ARM tendem a ser mais baratas que arquiteturas x86.

Figura 21 – Custo em dólares de 10 milhões de requisições *serverless* (*FaaS*) e *containers* (*CaaS*) no provedor *AWS Lambda*




Fonte: Desenvolvido pelo autor (2024).

Na Figura 21, representa-se uma estrutura em barras que compara custos associados a diferentes arquiteturas de computação *serverless* e *containers*, executando a mesma função *Node.JS*. O cenário avaliado envolve 10 milhões de requisições utilizando 512 *megabytes* de memória RAM. Os custos são comparados entre regiões diferentes, além de arquiteturas diferentes e modelos diferentes. Esse nível de customização só é apresentado no provedor da *Amazon*. O maior preço

apresentado nas execuções é na arquitetura x86, hospedada no Brasil no modelo *CaaS*, o menor preço apresentado é na arquitetura ARM, hospedada nos Estados Unidos, no modelo *FaaS*.

Com os testes que foram executados na plataforma *AWS Lambda*, constatou-se que:

No que se refere ao custo por região e arquitetura, os custos no Brasil (BR) são mais altos que na região dos Estados Unidos (US) tanto para arquitetura x86 quanto para ARM. Em ambas as regiões, ARM é mais barato que x86, refletindo a eficiência de custo da arquitetura ARM.

Considerando o custo por modelo de serviço, o modelo de função como serviço (*FaaS*) é geralmente mais barato que o modelo de *containers* como serviço (*CaaS*) em ambas arquiteturas e regiões.

A respeito do desempenho, a função *Lambda* da *AWS* é mais econômica quando se utiliza a arquitetura ARM e na região dos Estados Unidos (*us-east-1*). A região do Brasil (*sa-east-1*) apresenta custos mais elevados, o que é atribuído a diferentes fatores como infraestrutura e política de preços regionais do provedor *Amazon*, apesar disso, a região possui uma menor latência, fator que pode ser crucial dependendo do tipo de aplicação.

Por fim, optar pela arquitetura ARM pode resultar em economias significativas em custos operacionais, executar *workloads* na região dos Estados Unidos pode ser mais econômico do que no Brasil e, utilizar *FaaS* pode ser mais vantajoso do que *CaaS* em termos de custo, especialmente para *workloads* que se beneficiam de uma execução granular baseada em eventos.

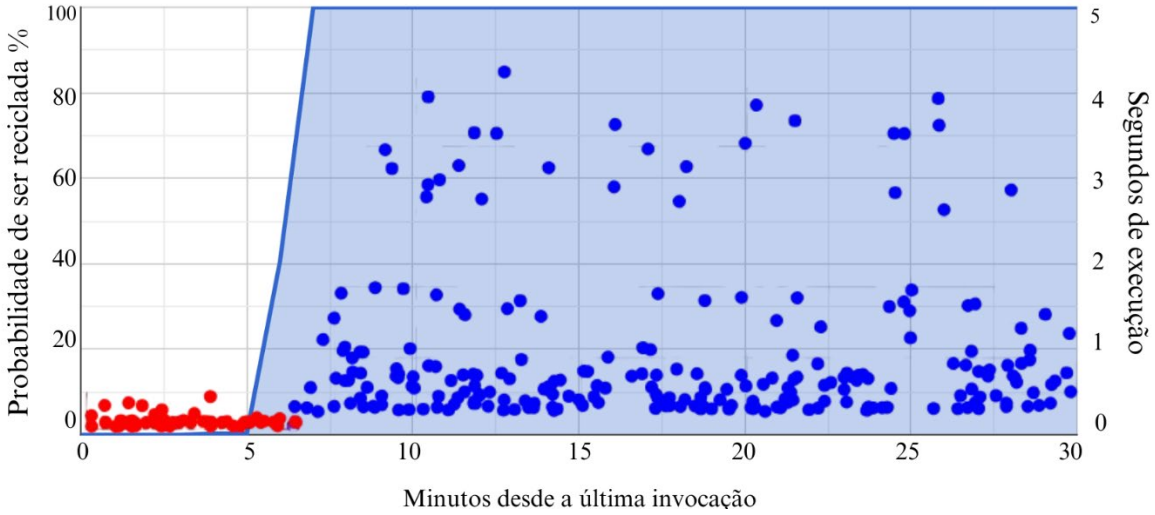
A partir de dados coletados acerca dos arranques a frio e arranques quentes, constata-se a maneira que as execuções comportam-se em relação ao tempo de execução. Os provedores, de um modo geral, oferecem, de forma dinâmica adição e remoção de instâncias que executam as funções. Quando uma nova instância trata sua primeira requisição, o tempo de resposta aumenta, o que é chamado de arranque a frio, ou *cold start*. A primeira inicialização a frio acontece quando a primeira requisição chega após o provisionamento da função.

Depois que a solicitação é processada, a instância permanece ativa para ser reutilizada em requisições subsequentes. A estratégia de reutilização difere entre um provedor e outra. A *AWS* possui políticas para reciclar uma instância ociosa após um período predefinido de cindo a sete minutos, em média. Com relação aos *containers*,

não há vida útil pré-definida pela plataforma, visto que é o desenvolvedor que a mantém.

Depois que a primeira requisição é processada, a instância continua ativa para poder ser reutilizada para requisições subsequentes. Na Figura 22 ilustram-se dados empíricos da probabilidade de uma instância ser reciclada após um período de inatividade no provedor *AWS*, representadas pelas linhas contínuas. Além disso, nas imagens é possível encontrar os segundos “quentes” de inicialização, em vermelho e, em azul, os arranques a frio, representadas pela distribuição de pontos.

Figura 22 - Período de reciclagem de instância e tempos de execução em segundos da plataforma *AWS Lambda.*




Fonte: Desenvolvido pelo autor (2024).

Na Figura 22 mostra-se a relação entre a probabilidade de uma função *serverless* ser reciclada e o tempo decorrido desde a última invocação, bem como a latência de execução dessa função. Ao observar as representações do tempo decorrido desde a última invocação (eixo X), representação da probabilidade de uma função ser reciclada (eixo Y esquerdo) e representação do tempo de execução da função em segundos (eixo Y direito) tem-se dados que mostram pontos de *warm starts* acontecem nos primeiros cinco minutos da invocação e por conta disso, zero por cento de chance de ser reciclada, demais pontos a partir do quinto minuto são mais propensos a serem reciclados. Após os cinco minutos, a probabilidade de reciclagem aumenta drasticamente, atingindo 100%. Isso sugere que, após esse período, as funções são recicladas com mais frequência, resultando em mais *cold starts*.

### 4.2.2 *Azure Functions*

Ao desenvolver um aplicativo de funções no *Azure*, é crucial selecionar uma opção de hospedagem que atenda adequadamente às necessidades do código. O *Azure* disponibiliza quatro principais opções de hospedagem: Plano de Consumo, Plano de Consumo Flexível, Plano Premium e Plano Dedicado. Essas opções são suportadas pela infraestrutura do *Azure*, operando em instâncias que podem ser tanto *Linux* quanto *Windows*. A escolha da opção de hospedagem influencia diretamente o dimensionamento do aplicativo de funções, os recursos disponíveis para cada instância e o suporte a funcionalidades avançadas, como a conectividade à Rede Virtual do *Azure*. Além disso, a opção selecionada afetará os custos de execução do código da função.

No Plano de Consumo, os recursos de computação são cobrados apenas quando as funções estão em execução, oferecendo escalonamento automático. As instâncias do host do Functions são dinamicamente adicionadas e removidas com base no número de eventos de entrada. Esse plano proporciona uma hospedagem verdadeiramente sem servidor, onde o pagamento é efetuado conforme o uso das funções e o escalonamento horizontal ocorre automaticamente, mesmo durante períodos de alta carga.

O Plano de Consumo Flexível, por sua vez, proporciona escalabilidade automática com diversas opções de computação e rede virtual, além de cobrança conforme o uso. As instâncias do host do Functions são ajustadas dinamicamente com base tanto no número de eventos de entrada quanto na simultaneidade por instância configurada. Este plano permite a redução das inicializações a frio, pois específica várias instâncias pré-provisionadas, e oferece suporte a rede virtual para maior segurança. Assim como no Plano de Consumo, o pagamento é feito por execução das funções, e o escalonamento horizontal é mantido mesmo em períodos de alta demanda.

O Plano Premium é projetado para escalar automaticamente de acordo com a demanda, utilizando trabalhos pré-ativados que permitem a execução de aplicativos sem atrasos após períodos ociosos. Este plano opera em instâncias com maior potência e permite conexão a redes virtuais. O Plano *Premium* é ideal para aplicativos de funções que operam de maneira contínua ou quase contínua, fornecendo maior controle sobre as instâncias e permitindo a implantação de vários aplicativos de

funções dentro do mesmo plano com escala orientada por eventos. É adequado para cenários com altas execuções e uso intensivo de *CPU* e memória, além de atender a requisitos específicos de conectividade de rede virtual ou a necessidade de uma imagem personalizada do *Linux* para a execução das funções.

Por fim, o Plano Dedicado permite que as funções sejam executadas em um plano do Serviço de Aplicativo com taxas regulares. Este plano é particularmente recomendado para cenários de execução prolongada, como o *Durable Functions*, que não é compatível com outros planos. O Plano Dedicado é ideal para o uso de máquinas virtuais subutilizadas, permitindo tanto o escalonamento manual quanto automático. Ele possibilita a execução de vários aplicativos *web* e funções dentro do mesmo plano, oferecendo acesso a *CPUs* de maior capacidade e suporte para isolamentos completos de computação. Além disso, proporciona alta memória e escalabilidade, especialmente em ambientes de Serviço de Aplicativo.

Nos testes conduzidos no presente trabalho, escolheu-se o Plano de Consumo Flexível, pois apresenta limite ilimitado de tempo limite de instâncias para funções, funcionalidade importante para comparação de resultados com outros provedores.

A plataforma *Azure Functions* permite escolher o sistema operacional *Windows* ou *Linux*, com algumas diferenças descritas na Tabela 10, as diferenças, de forma resumida, é o suporte ou não para *containers*. Até o momento, *Azure Functions* não oferece suporte a *containers* do *Windows*.

Tabela 10 - Suporte a Sistema Operacional *Azure Functions*

| **Plano** | ***Linux*** | **Windows** |
|---|---|---|
| Plano de Consumo | Apenas código<br>Sem suporte a *container* | Apenas código |
| Plano de Consumo Flexível | Apenas código<br>Sem suporte a *container* | Apenas código |
| Plano *Premium* | Suporte a código e *container* | Apenas código |
| Plano Dedicado | Suporte a código e *container* | Apenas código |

Fonte: Desenvolvido pelo autor (2024).

Os diferentes planos de consumo refletem na duração do tempo limite para execuções das funções, esse tempo pode ser pré definido em uma propriedade do

código *“functionTimeout”*. Essa propriedade se aplica especificamente a execuções de função. Depois que o gatilho inicia a execução da função, a função precisa retornar/responder dentro da duração do tempo limite. As diferenças podem ser observadas na Tabela 11, os limites estão contabilizados em minutos. Apesar de constar máximo ilimitado em alguns planos, segundo a documentação da plataforma, garante apenas sessenta minutos de execução, tempos maiores não possuem garantia de plena execução.

Tabela 11 - Planos e a duração do tempo limite de funções

| **Plano** | **Padrão** | **Máximo** |
|---|---|---|
| Plano de Consumo | 5 | 10 |
| Plano de Consumo Flexível | 30 | Ilimitado |
| Plano *Premium* | 30 | Ilimitado |
| Plano Dedicado | 30 | Ilimitado |

Fonte: Desenvolvido pelo autor (2024).

Os planos diferem também no dimensionamento das funções e como as funções reagem a arranques a frio. É possível escalar de 20 até 200 instâncias por execução de função, dependendo do plano escolhido e, sobre o *cold start*, o comportamento de inicialização difere seguindo a descrição encontrada na Tabela 12.

Tabela 12 - Comportamento do *cold start* em relação ao Plano

| **Plano** | **Comportamento** |
|---|---|
| Plano de Consumo | Funções escaladas para zero quando ociosas, apresenta maior *cold start* |
| Plano de Consumo Flexível | Dá suporte a instâncias sempre prontas |
| Plano *Premium* | Dá suporte a instâncias sempre prontas ou instâncias permanentemente ativas |
| Plano Dedicado | Execução contínua em um número prescrito de instâncias |

Fonte: Desenvolvido pelo autor (2024).

Estas informações são cruciais para arquitetos de sistemas e desenvolvedores ao planejar e escalar suas aplicações *serverless*.

Como linguagem de programação utilizada, a plataforma *Azure Functions* oferece diferentes versões para *Node.JS*. Na Tabela 13 mostra-se os *runtimes* suportados, com seus respectivos ciclos de vida. É possível compreender que existe um menor controle em relação ao SDK, não visível na hora de configurar funções no *Azure*.

Tabela 13 - *Runtimes* do *Node.JS* no *Azure Functions*

| ***Node.JS*** | | |
|---|---|---|
| **Versão** | **S. O** | **Ciclo de Vida** |
| *Node.JS* 20 | Windows e Linux | Abril de 2026 |
| *Node.JS* 18 | Windows e Linux | Abril de 2025 |
| *Node.JS* 16 | Windows e Linux | Setembro de 2023 |
| *Node.JS* 14 | Windows e Linux | Abril de 2023 |

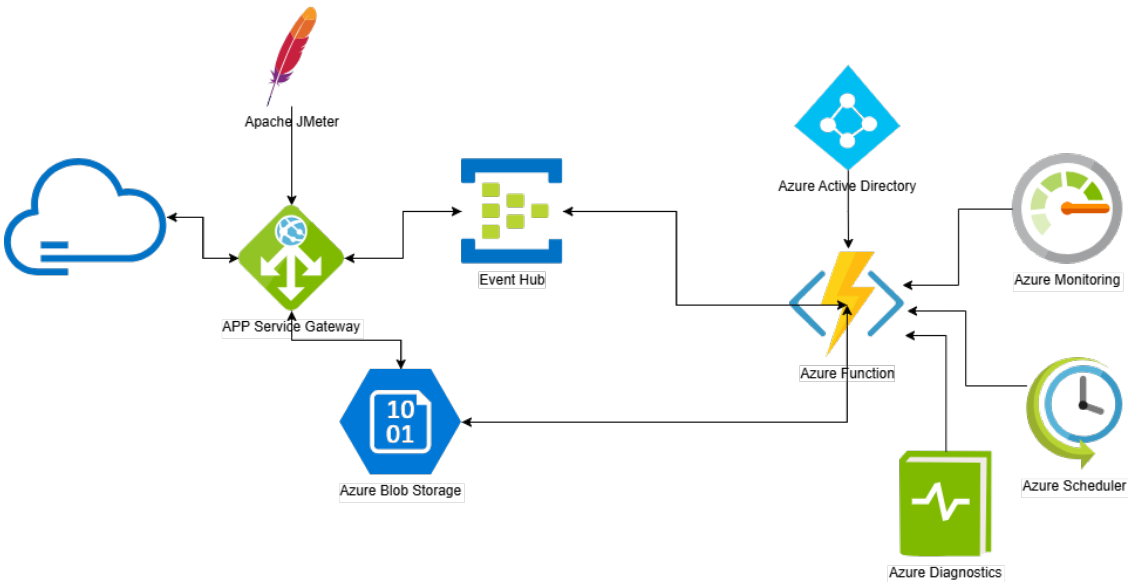
Fonte: Desenvolvido pelo autor (2024).

Em relação a arquitetura da aplicação na plataforma *Azure Functions*, na Figura 23 ilustra-se a organização dos processos envolvendo a execução na plataforma, que para testes, utiliza o ambiente *Linux*, no plano mais básico permitido pela plataforma. É importante destacar que a plataforma oferece menor nível de configurabilidade para as funções, ofertando a execução das funções como uma caixa-preta, um nível de configuração mais superficial se comparado com a plataforma *AWS Lambda*. Em contrapartida, o ambiente *Azure* oferece nativamente ferramentas de testes integradas ao ambiente de nuvem, com o *Apache JMeter*.

Na Figura 23 mostra-se uma arquitetura de aplicação *serverless* implementada no Microsoft Azure, integrando diversos serviços da plataforma para garantir uma operação escalável e eficiente. Os componentes da arquitetura, com foco em desenvolvedores que utilizam *Node.JS* e práticas *serverless*. Inicialmente, o *Azure Functions* é destacado como um serviço de computação *serverless* que permite a execução de código sob demanda sem a necessidade de provisionar explicitamente a infraestrutura. Com suporte para várias linguagens, incluindo *Node.JS, Azure Functions* executa funções específicas em resposta a eventos. Essas funções podem ser disparadas por outros serviços, como o *Event Hub*, ou por operações de

armazenamento, proporcionando uma abordagem flexível e eficiente para o processamento de eventos e dados.

Figura 23 – Arquitetura da aplicação em *Azure Functions*.




Fonte: Desenvolvido pelo autor (2024).

O *Azure Event Hub* é um serviço de ingestão de dados em tempo real, projetado para processar grandes volumes de eventos de forma eficiente. Este serviço serve como ponto de entrada para dados de eventos, que são processados por *Azure Functions*. Exemplos típicos de dados que podem ser enviados para o *Event Hub* incluem *logs* de aplicativos, dados de telemetria e outras formas de dados de eventos, permitindo um processamento contínuo e em tempo real.

Para gerenciar o tráfego de entrada, a arquitetura utiliza o *Azure Application Gateway*, um balanceador de carga da *web* que direciona o tráfego para os serviços apropriados dentro da infraestrutura *Azure*, incluindo *Azure Functions* e outros recursos *serverless*. Este componente garante que as solicitações dos usuários sejam distribuídas eficientemente, melhorando a disponibilidade e a escalabilidade da aplicação.

Além disso, a arquitetura incorpora o *Azure Blob Storage*, um serviço de armazenamento de objetos altamente escalável e durável para dados não estruturados. Este serviço é utilizado para armazenar arquivos de imagens e *logs*. *Azure Functions* podem ser configuradas para disparar execuções baseadas em

alterações ou uploads *no Blob Storage*, permitindo uma integração contínua entre armazenamento e processamento de dados.

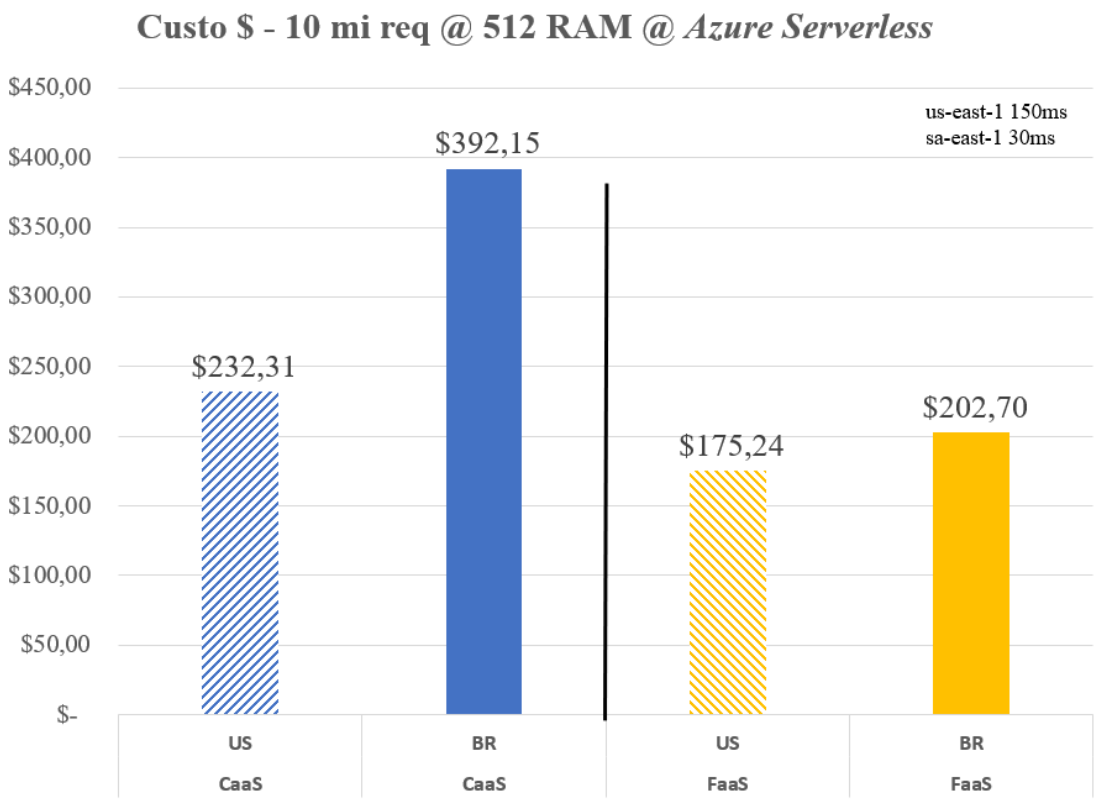
O monitoramento e diagnóstico são componentes críticos nesta arquitetura, sendo suportados por *Azure Monitoring* e *Azure Diagnostics. Azure Monitoring* oferece um conjunto de serviços para monitoramento e diagnóstico, incluindo métricas, *logs* e alertas. Isso garante que o desempenho e a integridade de todos os componentes da arquitetura sejam monitorados continuamente, permitindo a notificação de quaisquer problemas ou anomalias. *Azure Diagnostics* complementa esse serviço, coletando dados detalhados de diagnóstico, como logs e métricas de desempenho, auxiliando na solução de problemas e na otimização do desempenho da aplicação.

Para tarefas periódicas e agendadas, o *Azure Scheduler* é utilizado. Este serviço permite agendar trabalhos para serem executados em intervalos regulares, disparando *Azure Functions* ou outros processos baseados em um cronograma definido. Isso é particularmente útil para tarefas repetitivas, como limpeza de dados ou sincronizações periódicas, automatizando processos e garantindo a eficiência operacional.

*Azure Functions, Event Hub, Application Gateway, Blob Storage, Monitoring, Diagnostics* e *Scheduler* trabalham juntos para fornecer uma infraestrutura robusta e responsiva. Esta abordagem permite a construção de aplicações modernas que podem escalar de acordo com as necessidades, garantindo desempenho e disponibilidade superiores, permitindo aos desenvolvedores focarem na escrita de código e na implementação de funcionalidades sem preocupações com a administração da infraestrutura subjacente.

Em relação ao preço das execuções, na Figura 24 tem-se a precificação e comparação com o serviço de *container* oferecido pela *Azure*.

Figura 24 – Precificação e comparação *CaaS* e *FaaS* no *Azure*




Fonte: Desenvolvido pelo autor (2024)

Demonstra-se na Figura 24 os custos de processamento de 10 milhões de requisições utilizando 512 MB de *RAM* em um ambiente *Azure*, especificamente avaliando os serviços de *CaaS* e *FaaS* em duas regiões: Estados Unidos (US) e Brasil (BR). As latências das regiões são indicadas como *us-east-1* com 150 ms e *sa-east-1* com 30 ms, o que pode influenciar a escolha da região com base na proximidade geográfica e nos requisitos de latência das aplicações.

Na análise dos custos do serviço *CaaS*, observa-se que nos Estados Unidos o custo é de $232,31 para processar 10 milhões de requisições. Em contraste, no Brasil, o custo para o mesmo volume de requisições sobe para $392,15. Essa diferença significativa de custos pode ser atribuída a diversos fatores, incluindo diferenças na infraestrutura regional, custos operacionais e políticas de precificação específicas da *Azure* para cada região.

Para o serviço *FaaS*, os custos são consideravelmente menores em ambas as regiões quando comparados ao *CaaS*. Nos Estados Unidos, o custo para processar 10 milhões de requisições é de $175,24, enquanto no Brasil o custo é de $202,70. A menor disparidade de custos entre as duas regiões para o *FaaS*, comparada ao *CaaS*,

pode ser um indicativo de uma estrutura de preços mais homogênea para serviços *FaaS* em diferentes regiões, ou de uma maior eficiência operacional no serviço *FaaS* da *Azure*.

A escolha entre *CaaS* e *FaaS* deve considerar não apenas os custos diretos ilustrados no gráfico, mas também os requisitos específicos da aplicação, como a necessidade de gerenciamento de contêineres no caso do *CaaS* versus a simplicidade e escalabilidade do modelo *FaaS*. A menor latência na região *sa-east-1* (30 ms) em comparação com *us-east-1* (150 ms) também deve ser levada em conta, especialmente para aplicações sensíveis a latência.

O serviço *FaaS*, apresentando custos mais baixos em ambas as regiões, pode ser uma opção mais econômica para muitas aplicações, embora o serviço *CaaS* possa ser justificado por necessidades específicas de controle e configuração de contêineres. A latência mais baixa na região *sa-east-1* favorece ainda mais a escolha do Brasil para aplicações que exigem resposta rápida, apesar dos custos mais altos. Esta análise ressalta a necessidade de balancear custo e desempenho ao arquitetar soluções *serverless* na nuvem.

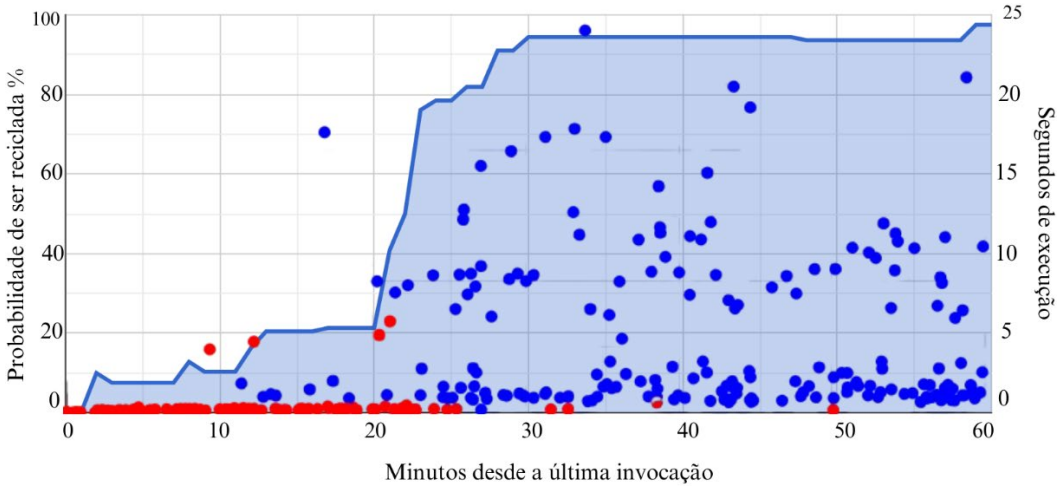
Na Figura 25 ilustra-se a relação entre o tempo decorrido desde a última invocação de uma função no *Azure Functions* e a probabilidade de uma nova instância ser reciclada, junto com o tempo de execução em segundos. Observa-se que, inicialmente, a probabilidade de reciclagem é baixa, com tempos de execução geralmente abaixo de 1 segundo. Após aproximadamente 10 minutos, a probabilidade começa a aumentar, atingindo cerca de 80% aos 20 minutos e estabilizando próximo a 100% após 25 minutos.

Essa tendência indica que instâncias de funções inativas por mais de 20 minutos são muito prováveis de serem recicladas. Os pontos azuis e vermelhos representam os tempos de execução das funções, onde os pontos vermelhos indicam inicializações quente e os pontos azuis, execuções a frio. É evidente que as inicializações a frio (em azul) ocorrem após períodos mais longos de inatividade. A maior concentração de pontos azuis após os 10 minutos sugere um aumento nos tempos de execução devido ao *overhead* das inicializações a frio.

No contexto de computação *serverless*, "*overhead*" refere-se ao tempo extra e aos recursos necessários para realizar operações adicionais além da execução direta da função. Especificamente, o *overhead* pode incluir o tempo necessário para inicializar uma nova instância da função (conhecido como "*cold start*"), carregar o

tempo de execução e o código, estabelecer conexões de rede e gerenciar recursos computacionais. Esse tempo extra pode causar latência adicional, impactando negativamente o desempenho da aplicação, especialmente se ocorrem muitas inicializações a frio devido a períodos de inatividade prolongados.

Figura 25 - Período de reciclagem de instância e tempos de execução em segundos da plataforma *Azure Functions*.




Fonte: Desenvolvido pelo autor (2024).

Sobre os segundos de execução, é observada a mesma tendência dos outros provedores de computação *serverless*, após o arranque a frio, as funções tendem a executar em tempos menores a um segundo. Um olhar mais detalhado no tempo de execução será ilustrado nas seção de comparação entre os três provedores.

### 4.2.3 *Google Cloud Functions*

O *Google Cloud Functions* oferece duas versões do produto de funções sob demanda, da mesma forma que a plataforma *Azure Functions*. O provedor recomenda usar a segunda versão, por oferecer recursos aprimorados, como demonstra-se na Tabela 14. A segunda versão, segundo o provedor, amplia possibilidades em relação ao tempo mais longo de processamento, executando maiores fluxos de código e cargas de trabalho por um tempo mais longo. Outra melhoria é o tratamento de instâncias de tamanho maior e alocações maiores em memória, simultaneidade na execução das funções com o objetivo de minimizar inicializações a frio e melhorar latência, suporte a tecnologias *CloudEvents* e *Eventarc*, integrações que possibilitam maior compatibilidade a eventos e gatilhos que acionam funções.

Tabela 14 - Tabela de comparação de versões do *Google Functions*

| **Recurso** | **1ª geração** | **2ª geração** |
|---|---|---|
| Tempo Limite da solicitação | Até 9 minutos | Até 60 minutos via HTTP e Até 9 minutos via eventos |
| Tamanho da instância | Até 8 GB RAM e 2 vCPUs | Até 16 GB RAM e 4 vCPU |
| Simultaneidade | 1 solicitação por instância | Até 1000 solicitações por instância |
| Divisão de tráfego | Sem suporte | Com suporte |
| Tipo de evento | 7 fontes de eventos | Mais de 90 fontes, suporte a *Eventarc* e *CloudEvents* |

Fonte: Desenvolvido pelo autor (2024).

O mesmo tipo de evento é utilizado para a condução dos testes na plataforma, como há a compatibilidade em estados de objetos armazenados em *cloud storage*, o estado de um novo objeto aciona o gatilho para execução da função "*google.cloud.storage.object.v1.finalized*", pode-se observar um exemplo de código na Figura 26. A ligação entre armazenamento, função e execução quando acionado pelo gatilho de novo objeto armazenado na nuvem.

Figura 26 - Exemplo da implantação da função na 2ª versão do *Google Functions*

```
gcloud functions deploy valterless \
--gen2 \
--trigger-event-filters="type=google.cloud.storage.object.v1.finalized" \
--trigger-event-filters="bucket=valterless_blob" \
...
```

Fonte: Desenvolvido pelo autor (2024).

A função, após implementada, pode ser escalonada e gerenciada automaticamente pelo próprio provedor, toda solicitação recebida atribui-se a instâncias de funções. Dependendo do volume de solicitações e número de instâncias, o provedor atribui a solicitação à instância existente ou recorre a criação de uma nova instância. Nos casos dos testes, onde o volume de solicitação aumentou repentinamente algumas vezes, houve a necessidade de configurar número máximo

de instâncias para responder aos eventos, contribuindo para a previsibilidade de custo monetário.

A segunda versão do *Cloud Functions* da *Google* é compatível com processamento de várias solicitações simultâneas em uma única instância de função, útil para minimizar *cold starts*, já que uma instância já aquecida processa várias solicitações simultaneamente.

O início de uma nova instância de função na plataforma também envolve o carregamento tanto do tempo de execução quanto do código da função. Esse processo, conhecido como inicialização a frio, pode resultar em latências maiores em comparação com solicitações direcionadas a instâncias já ativas. No entanto, quando a função é submetida a uma carga constante, o número de inicializações a frio tende a ser mínimo, exceto em casos de falhas recorrentes que requerem reinicializações frequentes do ambiente da função. Funcionalidade melhorada pela segunda versão do *Google Functions*.

Entre as linguagens de programação suportadas pelo provedor, o nível de configurabilidade é intermediário, ou seja, oferece um pouco mais de configurações possíveis que o provedor *Azure*, mas em contrapartida, ainda não oferece o nível de configurabilidade do provedor *Amazon*. Mesmo assim, na Tabela 15 descreve-se ambientes de execução possíveis na linguagem *Node.JS*, ambiente de sistema operacional e o tempo de vida do suporte a operações em diferentes versões de *runtimes*.

Tabela 15 - *Runtimes* do *Node.JS* no *Google Functions*

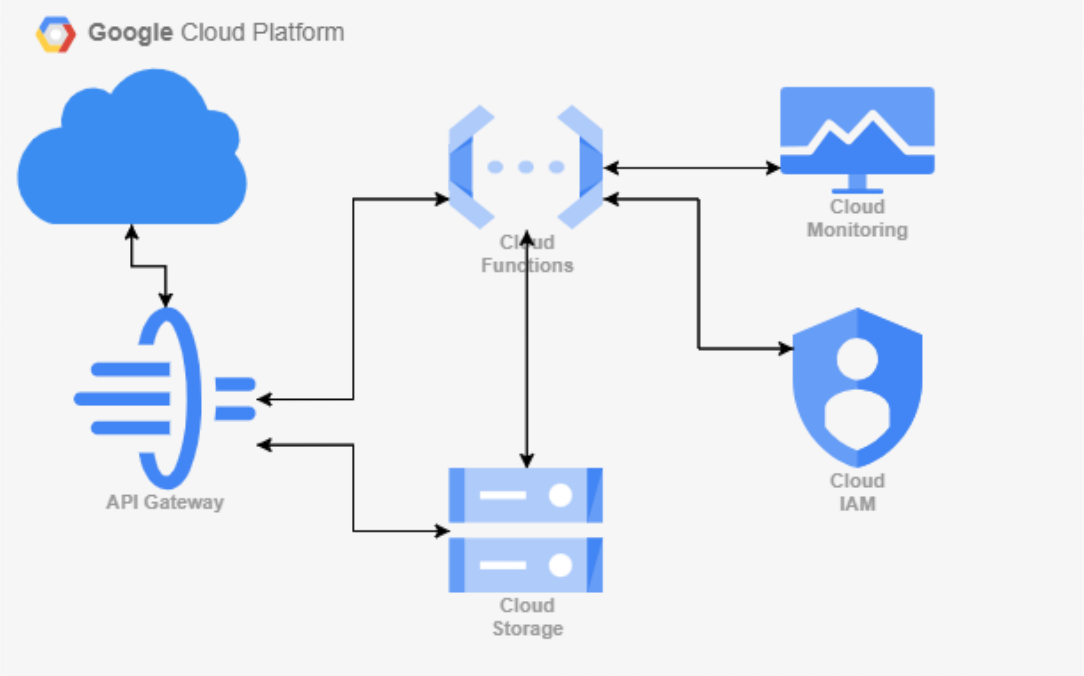
| ***Node.JS*** | | |
|---|---|---|
| **Versão** | **S. O** | **Ciclo de Vida** |
| *Node.JS* 22 | Ubuntu 22.04 | Outubro de 2027 |
| *Node.JS* 20 | Ubuntu 22.04 | Outubro de 2026 |
| *Node.JS* 18 | Ubuntu 22.04 | Outubro de 2025 |
| *Node.JS* 16 | Ubuntu 18.04 | Janeiro de 2025 |
| *Node.JS* 14 | Ubuntu 18.04 | Janeiro de 2025 |
| *Node.JS* 12 | Ubuntu 18.04 | Janeiro de 2025 |
| *Node.JS* 10 | Ubuntu 18.04 | Janeiro de 2025 |

Fonte: Desenvolvido pelo autor (2024).

Observa-se que o ambiente de desenvolvimento é *Linux Ubuntu* e o provedor ainda oferece suporte a versões mais antigas do *Node.JS*, encerrando o ciclo de vida da maioria deles em janeiro do ano de 2025. Esse suporte a versão mais antigas possibilita um tempo maior para atualização de funções em produção.

A arquitetura da aplicação na plataforma *Google Functions* está representada na Figura 27, com o plano mais básico de hospedagem, com mesmos parâmetros de configuração descritos na Seção 4.2.1, onde se localizou a configuração mais viável para execução no primeiro provedor e, consequentemente, os mesmos parâmetros estão empregados no provedor *Azure* e *Google*.

Figura 27 - Arquitetura da aplicação na plataforma *Google Functions*



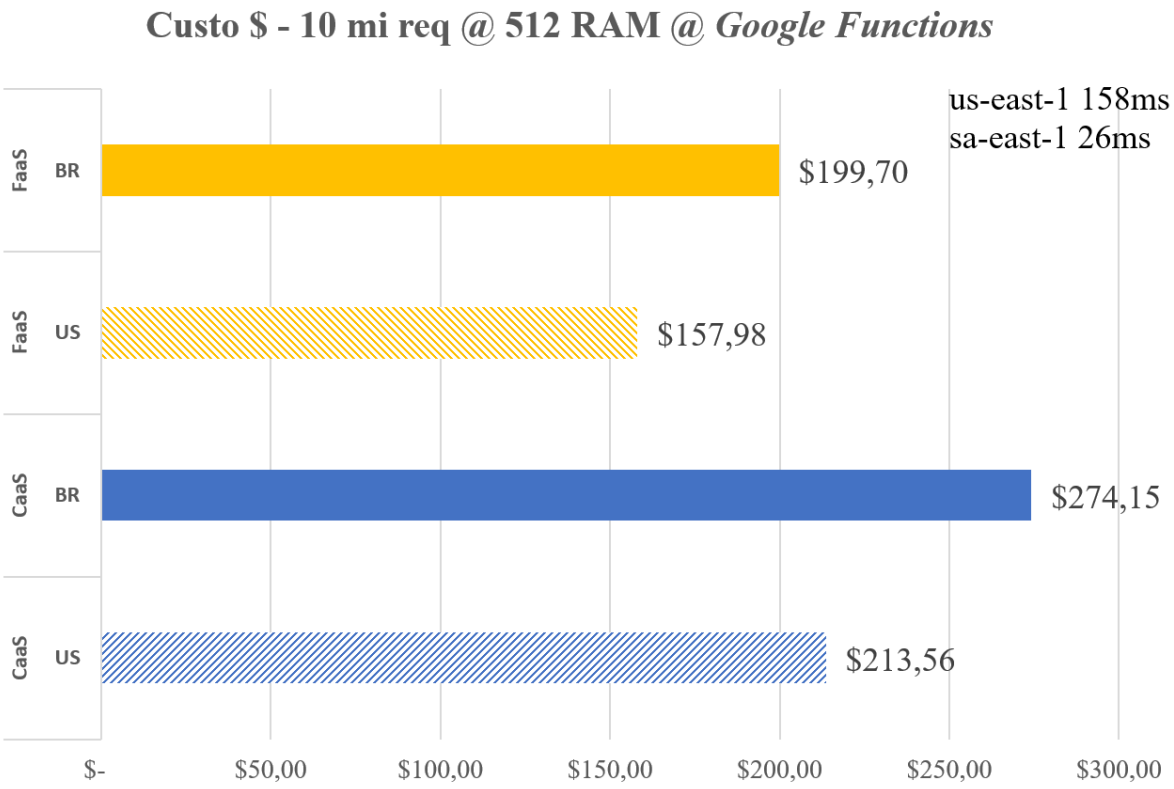

Fonte: Desenvolvido pelo autor (2024).

Na Figura 27, apresenta-se o fluxo de execução da aplicação *serverless*. O *API Gateway* atua como um ponto de entrada e balanceamento de carga para as solicitações do *backend*, gerenciando e roteando as solicitações. Seu papel é garantir a segurança, controle do tráfego e monitorização da aplicação. A *Cloud Function* é a função *serverless* que executa a biblioteca *sharp* na linguagem *Node.JS*, é acionada baseada no gatilho de mudança de estado dentro do *Cloud Storage*, serviço de armazenamento de objetos que recebe e armazena imagens, além de armazenar *logs* e *backups*. *Cloud IAM* gerencia identidades e define políticas de acesso aos módulos

da aplicação e recursos, tem o papel de garantir a autenticidade e integridade da aplicação. O módulo *Cloud Monitoring* oferece visibilidade sobre o desempenho, disponibilidade e integridade das aplicações no provedor, tem o papel de monitorar as funções, o *gateway* e demais componentes da aplicação, proporcionando alertas e *insights* para a detecção e resolução de problemas.

O custo das execuções está disposto na Figura 28, apresenta-se uma análise comparativa dos custos de execução de 10 milhões de requisições utilizando *Google Functions* com 512 MB de *RAM* nas regiões *us-east-1* e *sa-east-1*, com latências de 158 ms e 26 ms, respectivamente, além de comparar o modelo *CaaS* e *FaaS*.

Figura 28 – Precificação e comparação *CaaS* e *FaaS* no *Google Functions*




Fonte: Desenvolvido pelo autor (2024).

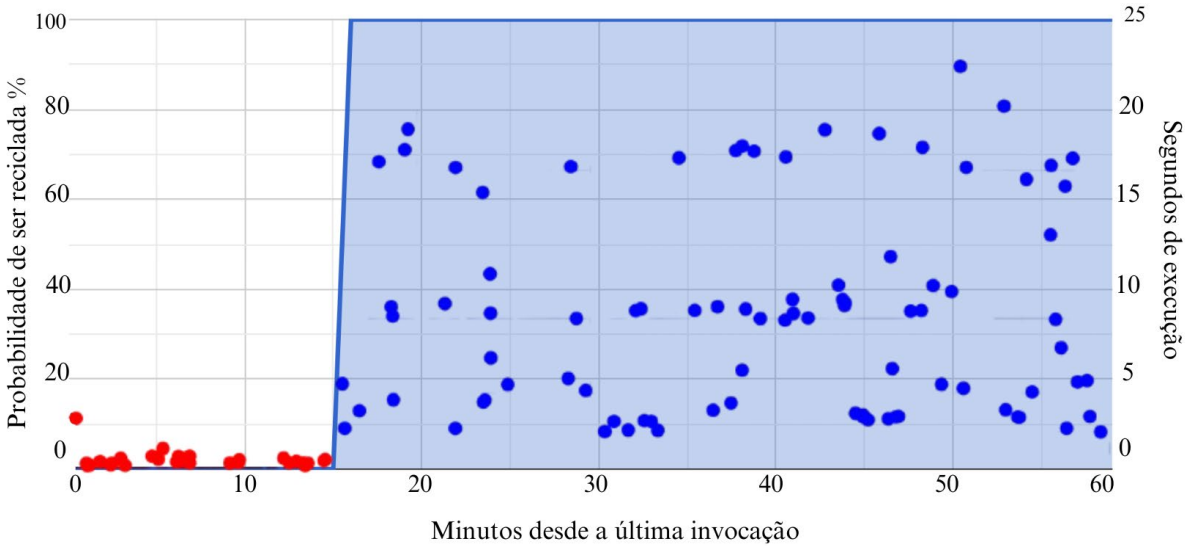
Ao analisar o custo do *CaaS*, observa-se que nos EUA o custo é de $213,56, enquanto no Brasil esse valor sobe para $274,15, indicando um aumento de aproximadamente 28%. No contexto de *FaaS*, o custo nos EUA é de $157,98, sendo mais econômico do que no Brasil, onde o custo é de $199,70, refletindo um aumento de cerca de 26%. Portanto, o *FaaS* apresenta-se como uma alternativa mais

econômica em ambas as regiões comparado ao *CaaS*, com uma diferença de custo de 26% nos EUA e 37% no Brasil.

Além dos custos, a latência também varia entre as regiões: a região do Brasil (*sa-east-1*) tem uma latência de 26ms, significativamente menor comparada aos 158ms nos EUA (*us-east-1*). Isso sugere que, apesar dos custos mais altos no Brasil, a latência mais baixa pode ser uma vantagem considerável dependendo das necessidades de desempenho da aplicação.

Como estratégia de reciclagem de instâncias e funções após tempo de inatividade e tempo de execução, tem-se na Figura 29 a distribuição entre pontos indicando as execuções de funções. Apresenta a probabilidade de uma função no *Google Functions* ser reciclada em relação ao tempo decorrido desde sua última invocação, além do tempo de execução (latência) das solicitações.

Figura 29 – Período de reciclagem de instância e tempos de execução em segundos da plataforma *Google Functions*




Fonte: Desenvolvido pelo autor (2024).

Nos primeiros 10 minutos, a probabilidade de reciclagem é baixa, como indicado pelos pontos vermelhos (de *warm start*) na parte inferior do gráfico. Isso implica que funções invocadas dentro deste intervalo tendem a ser atendidas por instâncias já em execução, resultando em tempos de execução mais baixos e consistentes.

Entretanto, após cerca de 10 minutos, a probabilidade de reciclagem aumenta drasticamente, atingindo 100% em aproximadamente 20 minutos. Este aumento é

representado pela área azul crescente no gráfico. Quando a função é reciclada, novas instâncias precisam ser iniciadas, o que é refletido nos pontos azuis correspondentes a latências mais altas e variáveis, chegando a até 25 segundos.

### 4.3 Resultados dos Testes e Comparativos

Ao analisar os resultados obtidos na Subseção 4.2, resultados de 10 milhões de requisições em cada um dos três provedores de computação *serverless* e *containers*, algumas relações podem ser observadas, destacando variação de preço (em dólares) entre as *Functions-as-a-Service* e *Containers-as-a-Service*, além das diferenças entre regiões dos Estados Unidos (*us-east-1*) e Brasil (*sa-east-1*). Na Tabela 16 encontram-se os dados filtrados do menor preço, para o maior.

Tabela 16 – Custo de 10 milhões de requisições para cada serviço, do mais barato para o mais caro

| Provedor | as-a-Service | Região | Custo |
|---|---|---|---|
| *AWS* | Function | US | $ 133,33 |
| Google | Function | US | $ 157,98 |
| *AWS* | Container | US | $ 173,01 |
| Azure | Function | US | $ 175,24 |
| *AWS* | Function | BR | $ 183,33 |
| Google | Function | BR | $ 199,70 |
| Azure | Function | BR | $ 202,70 |
| Google | Container | US | $ 213,56 |
| Azure | Container | US | $ 232,31 |
| Google | Container | BR | $ 274,15 |
| *AWS* | Container | BR | $ 297,58 |
| Azure | Container | BR | $ 392,15 |

Fonte: Desenvolvido pelo autor (2024).

Ao analisar dados contidos na Tabela 16, tem-se que:

- Na região dos Estados Unidos, o *AWS Lambda* é o mais econômico ($133,33), sendo 18,4% mais barato que o *Google Function* ($157,98) e 31,4% mais barato que o *Azure Function* ($175,24). Quando comparado ao custo de *containers*, o *AWS Lambda* é 23% mais barato que o *AWS Fargate* ($173,01), 37,5% mais barato que o *Google Containers* ($213,56) e 42,6% mais barato que o Azure Container ($232,31).
- Na região Brasil, há um aumento geral nos custos. O *AWS Lambda* custa $183,33, sendo 8,2% mais barato que o *Google Function* ($199,70) e 9,6% mais barato que o *Azure Function* ($202,70). Em relação aos containers, o *AWS Lambda* é 38,4% mais barato que o *AWS Fargate* ($297,58), 34,4% mais barato que o *Google Containers* ($274,15) e 53,3% mais barato que o *Azure Containers* ($392,15).
- Comparando entre regiões, o *AWS Lambda* na região *sa-east-1* é 37,5% mais caro que na região *us-east-1*, enquanto o *Google Function* é 26,4% mais caro e o *Azure Function* é 15,7% mais caro na região BR. Para containers, a diferença é ainda mais acentuada: o *AWS Fargate* na região do Brasil é 72% mais caro que na região US, o *Google Container* é 28,4% mais caro e o *Azure Container* é 68,8% mais caro na região BR.

Essas análises mostram que a escolha do provedor e da região de implantação pode ter um impacto significativo nos custos operacionais. O *Azure* é consistentemente o mais caro, especialmente para *containers* na região Brasil, enquanto o *AWS* oferece a opção mais econômica para funções e *containers* na maioria dos casos.

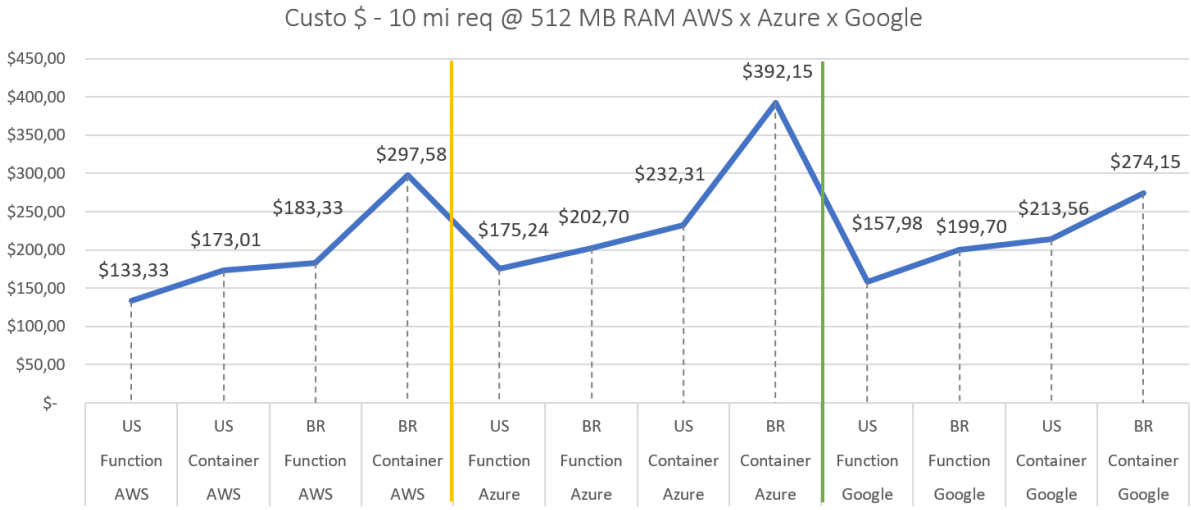
Para melhor visualização das relações de custo por provedor, é apresentado na Figura 30 o custo das execuções, separados pelos provedores.

A *AWS Lambda* na região US apresenta o menor custo, $133,33, destacando-se como a opção mais econômica. O custo do *AWS Fargate* na mesma região é 29,7% maior, atingindo $173,01. Na região BR, o *AWS Lambda* custa $183,33, um aumento de 37,5% em relação à mesma função na região US, enquanto o custo do *AWS Fargate* é significativamente maior, $297,58, representando um aumento de 72%.

O custo do *Google Functions* na região US é $157,98, 18,4% superior ao do *AWS Lambda*, enquanto na região do Brasil é $199,70, 26,4% mais alto que na região US. O *Google Containers* custa $213,56 na região US, sendo 35,2% mais caro que o

*Google Functions*. Na região do Brasil, o *Google Containers* custa $274,15, um aumento de 28,4% em comparação à região US.

Figura 30 – Custo de 10 milhões de execuções em três provedores de computação em nuvem




Fonte: Desenvolvido pelo autor (2024).

Para o *Azure*, o custo do *Azure Functions* na região US é $175,24, sendo 31,4% mais caro que o *AWS Functions* na mesma região. Na região BR, o *Azure Functions* custa $202,70, representando um aumento de 15,7%. O custo do *Azure Containers* na região US é $232,31, 32,6% mais alto que o *Azure Functions*, enquanto na região do Brasil o custo do *Azure Containers* é o mais elevado entre todos, $392,15, representando um aumento de 68,8% em relação à região US.

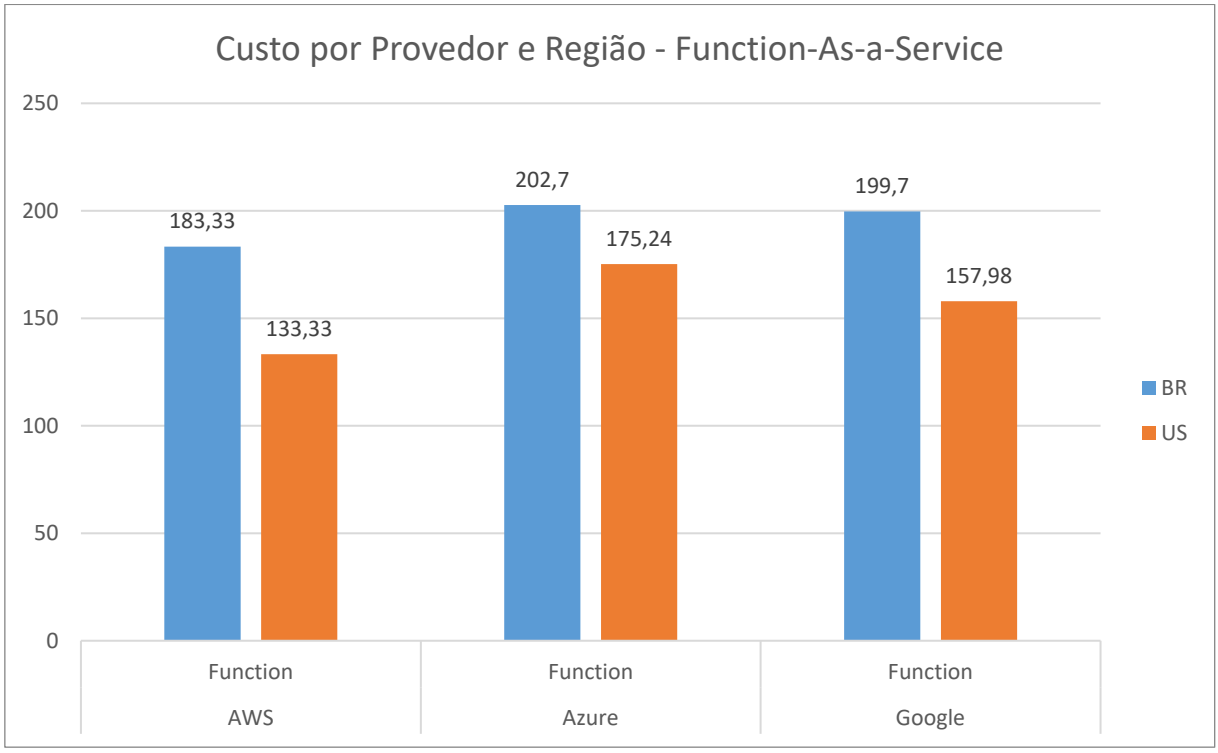
Essa análise revela variações significativas nos custos entre funções e containers, assim como entre diferentes regiões, sendo a *AWS* consistentemente mais econômica e o *Azure* apresentando os maiores custos, especialmente na região BR.

Para apresentar como uma visualização de resultados individuais dos modelos de computação, tem-se na Figura 31 o custo por provedor e região, nas execuções *Function-as-a-Service*.

Observa-se que a *AWS* apresenta os menores custos em ambas as regiões, com $133,33 nos EUA e $183,33 no Brasil. Comparativamente, o custo na região BR é 37,5% maior que na região US. Para a *Azure*, o custo nos EUA é $175,24, enquanto no Brasil é $202,70, um aumento de 15,7%. A *Google*, por sua vez, apresenta custos de $157,98 nos EUA e $199,70 no Brasil, resultando em um aumento de 26,4%.

Essas diferenças destacam como os custos variam significativamente não apenas entre os provedores, mas também entre as regiões. A *AWS* é consistentemente a opção mais econômica, especialmente na região US. A *Azure*, embora competitiva nos EUA, apresenta um aumento substancial de custo no Brasil, de 15,7%. A *Google*, por sua vez, mostra uma tendência similar à *Azure*, mas com um aumento percentual mais elevado de 26,4% entre as regiões. Essas variações reforçam a importância de considerar tanto o provedor quanto a localização geográfica ao planejar a implementação de soluções serverless.

Figura 31 – Custo de provedores *FaaS* comparados por região




Fonte: Desenvolvido pelo autor (2024).

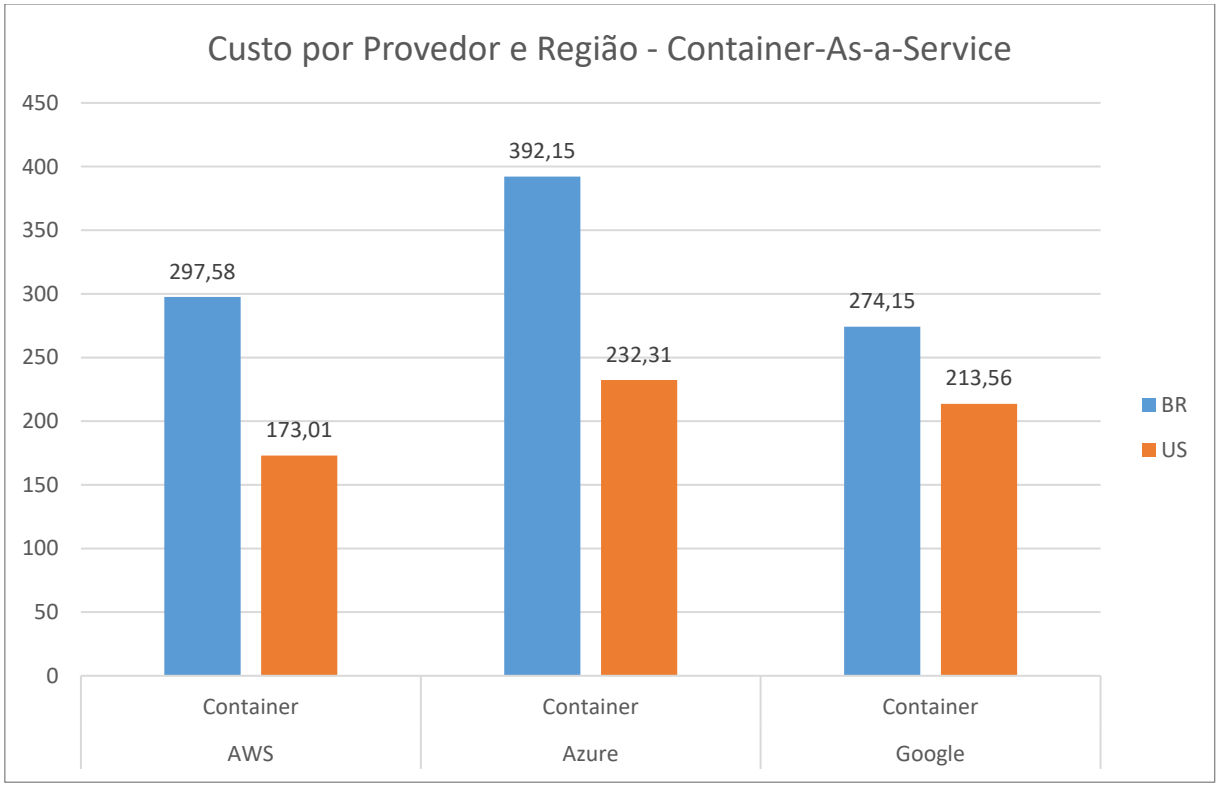
Observando os dados presentes na Figura 32, percebe-se que o *AWS* apresenta um custo de R$ 297,58 no Brasil e R$ 173,01 nos Estados Unidos. Essa diferença significativa representa um aumento de aproximadamente 72% nos custos quando os serviços são operados no Brasil. Essa disparidade pode ser atribuída a diversos fatores, como impostos, infraestrutura local, demanda e política de preços do *AWS* para a região brasileira.

A *Azure*, por outro lado, mostra o maior custo entre os provedores, com $ 392,15 no Brasil e $ 232,31 nos Estados Unidos. Semelhante à *AWS*, há uma

diferença marcante nos custos entre as duas regiões, com o Brasil apresentando um custo aproximadamente 69% maior do que nos Estados Unidos.

A *Google* exibe um comportamento um pouco diferente. Enquanto os custos no Brasil ($ 274,15) e nos Estados Unidos ($ 213,56) são menores em comparação com *Azure* e *AWS*, a diferença de custo entre as regiões é menos pronunciada, com um aumento de aproximadamente 28% no Brasil.

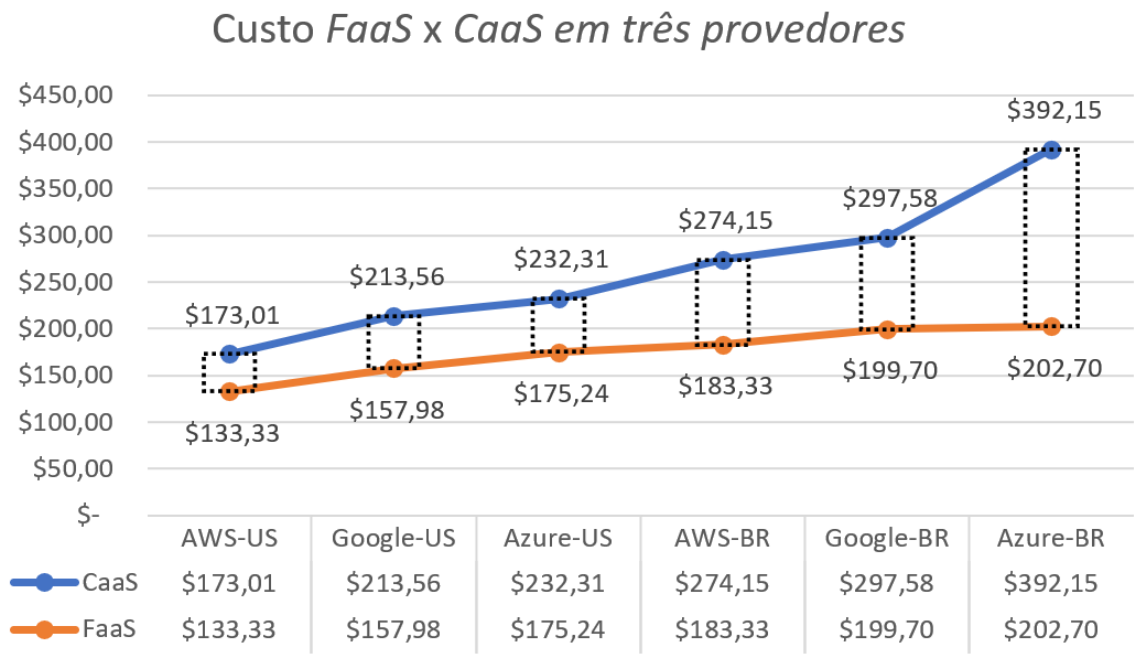
Figura 32 – Custo de provedores *CaaS* comparados por região




Fonte: Desenvolvido pelo autor (2024).

Na Figura 33 ilustra-se comparativamente duas linhas de custo de ambos os modelos, oferecendo uma visão melhor em relação às diferenças entre os modelos *FaaS* e *CaaS*. Entre as linhas, há a diferença entre custos entre o modelo de funções e o modelo de *containers*. A *AWS* apresenta menor diferença entre custos dos modelos, quando a execução é localizada nos Estados Unidos. A maior disparidade entre valores ocorre na região do Brasil, pelo provedor *Azure*.

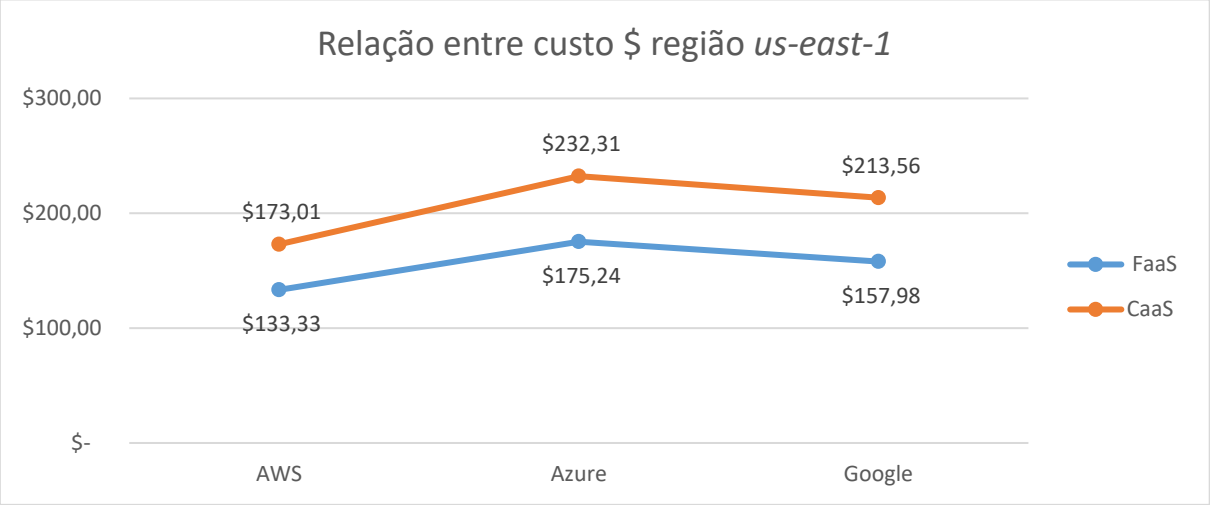
Figura 33 – Custo *FaaS* contra *CaaS* em três provedoras.




Fonte: Desenvolvido pelo autor (2024).

A relação mais detalhada entre o custo das regiões *us-east-1* e *sa-east-1* pode ser observada na Figura 34 e na Figura 35, respectivamente. A região dos Estados Unidos apresentou menor diferença entre os modelos, enquanto a região *sa-east-1* apresentou maior diferença entre modelos. Plataforma *Google Containers* apresentou melhor custo se comparado a *AWS Fargate*. Mesmo assim, a aplicação testada em *Node.JS* teve melhor custo no modelo de *Functions-as-a-Service* no provedor *AWS*.

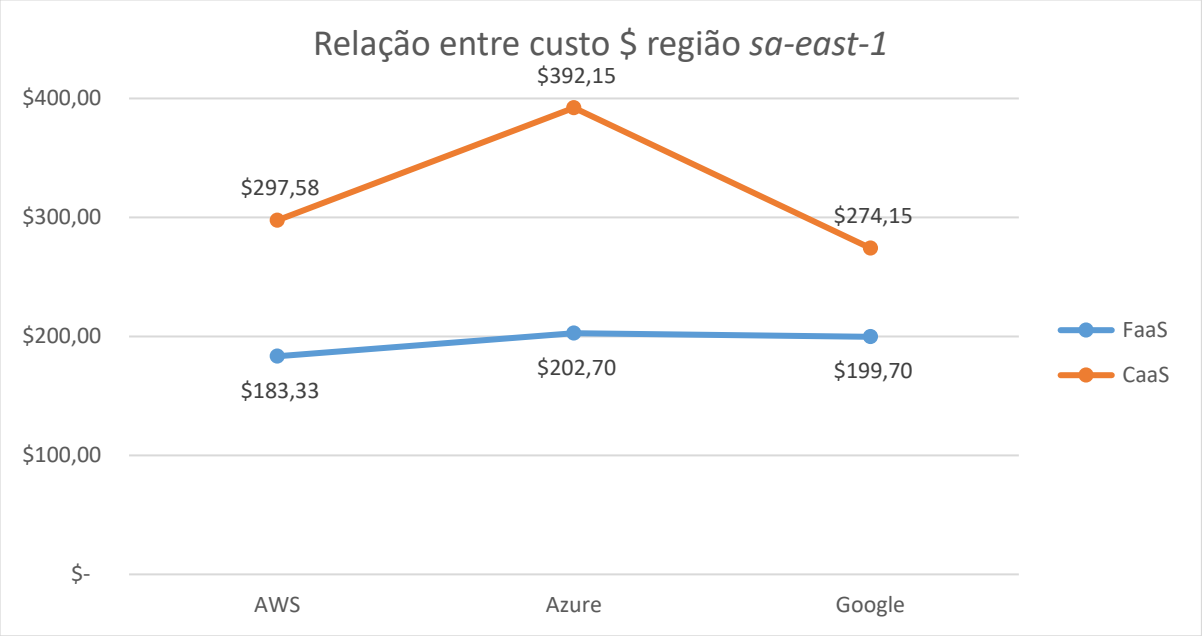
Figura 34 – Relação entre custo $ na região *us-east-1*




Fonte: Desenvolvido pelo autor (2024).

Na região *sa-east-1*, ainda sim o serviço *AWS Lambda* apresentou melhor custo. A maior disparidade de custo também ocorreu na região com o provedor *Azure*.

Figura 35 – Relação entre custo $ na região *sa-east-1*

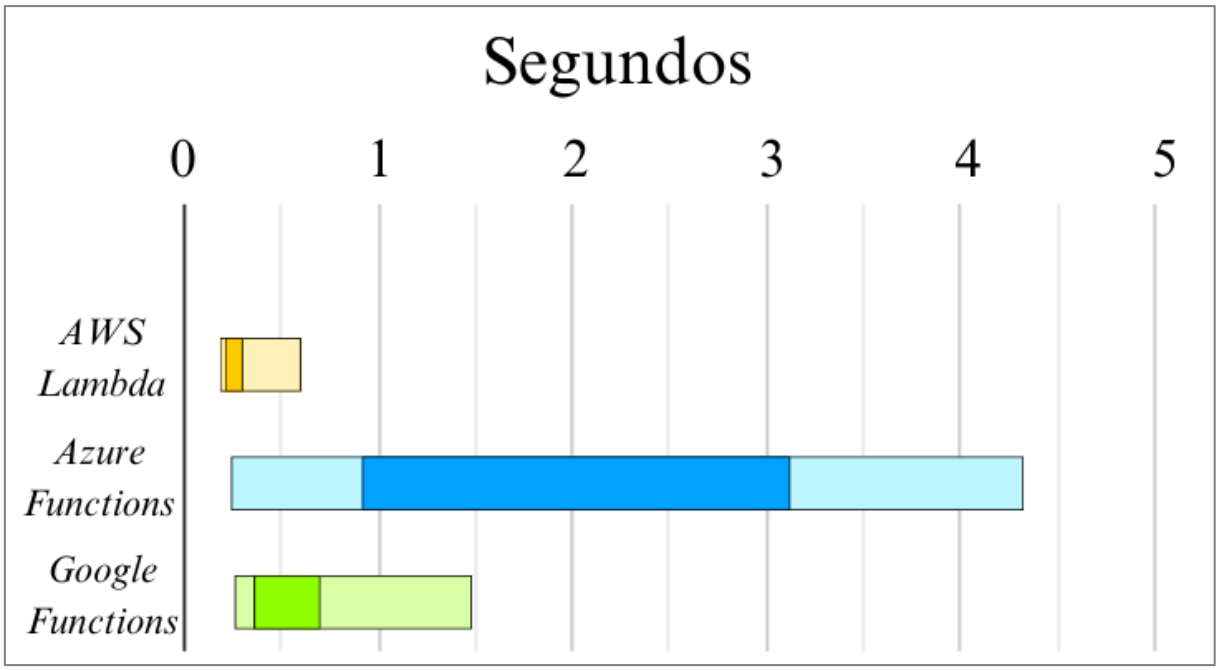



Fonte: Desenvolvido pelo autor (2024).

Por fim, na Figura 36 é possível observar o tempo de execução de todas as funções no modelo *Function-as-a-Service*, nos três provedores de computação *serverless*. As dez milhões de requisições de cada provedor estão sumarizadas e, o tom mais escuro da cor da barra representa uma concentração de 67% dos tempos de execução.

O tempo de execução do serviço *AWS Lambda* é o menor, indicando o menor tempo entre os três provedores analisados. O tempo de execução é menor que 0,5 segundo na maioria das vezes, o que demonstra a melhor eficiência dos três. A execução do *Azure Functions* é significativamente maior, com variação entre 0,3 e 4,5 segundos, com maior parte das execuções variando entre 1 e 3 segundos. A latência maior é um ponto crítico a considerar, tornando *Azure Functions* menos adequados para esse cenário. *Google Functions* apresenta tempo de execução intermediário, com variação entre 0,3 segundos até 1,5 segundos, com maior número de execuções na média do meio segundo.

Figura 36 – Comparativo entre o tempo de execução da função *serverless* nos três provedores




Fonte: Desenvolvido pelo autor (2024).

Ao analisar os tempos de execução médios das funções *serverless* fornecidos pelos três provedores observa-se que *AWS Lambda* tem a menor média de tempo de execução, de 0,5 segundos. Quando comparado a *Azure Functions*, que tem uma média de 2,45 segundos, *AWS Lambda* é significativamente mais rápido. A diferença percentual entre *AWS Lambda* e *Azure Functions* é de 390%, indicando que *Azure Functions* é quase quatro vezes mais lento que *AWS Lambda*.

*Google Functions*, com uma média de tempo de execução de 0,85 segundos, também é mais lento que *AWS Lambda*, mas a diferença é menor, com uma diferença percentual de 70%. Isso demonstra que *Google Functions* é 70% mais lento que *AWS Lambda*, mas ainda consideravelmente mais rápido que Azure Functions.

Comparando *Azure Functions* com *Google Functions*, a diferença percentual é de aproximadamente 188,24%. Isso indica que *Azure Functions* tem um tempo de execução significativamente maior, sendo quase duas vezes mais lento que *Google Functions*. Essas diferenças percentuais destacam a importância de considerar os tempos de execução ao escolher um provedor de serviços *serverless*.

Acerca dos *containers*, o tempo de execução deles são definidos pelo desenvolvedor, tornando-se disponíveis durante o tempo que estiver ligado, não há atraso no provisionamento de capacidade computacional para eles.

# CAPÍTULO 5 - CONCLUSÃO

Neste capítulo, são descritas as conclusões do estudo e trabalhos futuros, objetiva-se uma integração e um confronto dos resultados na parte prática, com fundamentação bibliográfica apresentada. As constatações encontradas foram uma construção de vários meses e estudos, que culminaram na publicação de trabalhos em eventos:

- *"Serverless Computing - O data center agora é o computador",* em 2022 nos Anais Da XIII Escola Regional De Alto Desempenho De São Paulo. (CANHIZARES FILHO; LOBATO, 2022).
- "*Desvendando as Nuvens: Uma análise comparativa de desempenho de aplicações serverless (FaaS) e containers (CaaS) em provedores cloud"*, em 2023 no XII Workshop do Programa de Pós-Graduação em Ciência da Computação da Unesp (WPPGCC 2023).
- *"From cold starts to pricing: A comparative analysis in executing Node.JS application on AWS Lambda and Azure Functions.,"* em 2024: CISTI'2024 – *19th Iberian Conference on Information Systems and Technologies.* (CANHIZARES FILHO, et al., 2024).

Esta dissertação objetivou realizar uma discussão acerca do estado da arte da computação sem servidor, enumerando e avaliando o serviço de três principais provedores da indústria da computação em nuvem, a fim extrair suas vantagens à benefício de uma aplicação *web* monolítica, além de esbarrar também em desvantagens competitivas no âmbito de desenvolver melhores aplicações em nuvem.

Os modelos *Functions-as-a-Service* e *Containers-as-a-Service* foram empregados a uma aplicação em *Node.JS* para se obter suas diferenças do ponto de vista de implementação, custo e execução, com a finalidade de realizar uma migração de monólitos e escolha de melhor provedor.

Ao comparar provedores, observa-se que *AWS Lambda* oferece maior configurabilidade em relação às instâncias a serem provisionadas, em comparação com os demais provedores. Isso pode ser atribuído, em parte, ao maior tempo de mercado do serviço.

De todos os desafios elencados na subseção 2.5, foi asseverado que o aprisionamento do provedor de computação em nuvem acontece, pois, a execução de funções e de *containers* na nuvem sempre é acompanhado de aplicações suporte proprietários do provedor. Aplicações que armazenam *logs*, armazenamento, controle de acesso e permissões de usuários para determinadas tarefas. Nos três provedores, o código precisou sofrer pequenos ajustes que não impactaram diretamente na sua execução, e sim na compatibilidade com os demais serviços oferecidos pelo próprio provedor, como por exemplo a *API Gateway* e armazenamento de cada um deles.

Em relação aos modelos *FaaS* e *CaaS*, os *containers* oferecem maior controle sobre a vida útil do ambiente, mas, por outro lado, as funções podem ser mais variáveis, o que torna o custo imprevisível em muitos casos.

Para se extrair do melhor das funções, é possível criar diferentes estratégias como *keep alive*, que consiste em implementar invocações periódicas para manter funções sempre aquecidas, refinamento nas configurações de funções para aliar tempo de execução e custo e saber que sempre existe o *trade-off*, manter funções mais rápidas faz-se necessário desprender de maior recurso financeiro, cada aplicação tem que se chegar ao melhor custo-benefício, e isso só se atinge fazendo testes nas aplicações.

Embora o *AWS Lambda* seja o mais maduro e popular dos três, o *Azure Functions* parece ter recursos muito semelhantes e, em alguns aspectos, mais opções para acomodar casos especiais. O *Google Cloud Functions* tem alguns recursos a menos, mas ainda é bastante comparável aos outros dois.

O detalhe está nos pormenores. Há outros fatores a serem considerados ao comparar esses três serviços *FaaS*. Mas, seja qual for o caso, a principal conclusão aqui é que o *FaaS* apresenta os melhores resultados, e é uma tecnologia verdadeiramente transformadora que só continuará a ganhar adoção.

É visível que ambos os modelos podem ser combinados com a finalidade de se tirar melhor proveito das características dos dois, a depender do tamanho da aplicação, a forma que pode ser fragmentada na nuvem e, a forma que se integram, uma boa possibilidade é utilizar-se dos dois modelos de forma integrada.

Dada a importância do tema, para trabalhos futuros há o que se percorrer no campo da investigação sobre *serverless computing*, sugerindo-se análise de desempenho e custo de mais linguagens de programação utilizadas no mercado, como *Python*, *Java*, *Go*, *Ruby*, entre outros.

Além disso, efetuar testes para outros provedores de computação *serverless* como *Alibaba Functions*, *IBM Bluemix OpenWhisk*, *CloudFlare Workers*, *Netlify Functions* e *Vercel Functions*, ajuda-se assim a amadurecer outras plataformas. Plataformas *opensource* também podem ser exploradas.

Por fim, outras linhas de pesquisa decorrente do paradigma *"As-A-Service"*, como *AIaaS - Artificial-Intelligence-as-a-Service, DaaS – Data-as-a-Service e DBaaS – Database-as-a-Service* podem ser beneficiadas pelos avanços no tema.

# REFERÊNCIAS

ADZIC G., CHATLEY R. *Serverless* Computing: Economic and Architectural Impact. **ESEC/FSE 2017 Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering** Pages 884-889, 2017.

ALLIAUME, E.; LE ROUX, B.; **Cold Start / Warm Start with *AWS* Lambda**. Disponível em https://blog.octo.com/en/cold-start-warm-start-with-*AWS*-lambda/ Acesso em 22 jun. 2022.

AMORIM, R. **O que é um BaaS – Backend as a Service?** Disponível em: http://rsamorim.azurewebsites.net/2017/12/05/o-que-e-um-baas-backend-as-a-service/. Acesso em 19 jun. 2022.

*AWS* L**. Amazon Web Services Lambda**. Disponível em *AWS*.amazon.com/lambda. Acesso em 14 jun. 2022.

*AWS*. **Amazon Web Services**. Disponível em: *AWS*.amazon.com. Acesso em 9 jun. 2022

*AWS*. ***AWS* Lambda Runtimes**. Disponível em https://docs.*AWS*.amazon.com/lambda/latest/dg/lambda-runtimes.html Acesso em 22 jun. 2022

AKKUS, I. E.; CHEN, R.; RIMAC, I.; STEIN, M.; SATZKE, K.; BECK, A. ADITYA, P.; HILT, V. 2018. SAND: Towards high-performance *serverless* computing. **In Proceedings of the 2018 USENIX Annual Technical Conference**. 923-935.

AZURE. **Microsoft Azure**. Disponível em: azure.microsoft.com. Acesso em 9 jun. 2022.

A3LOGICS. 2022. **Top 10 Tools for IaaS Cloud Computing**. Disponível em: https://a3logics.com/blog/tools-for-iaas-cloud-computing/. Acesso em 21 abr 2023.

BALDINI, I., et al. ***Serverless* Computing: Current Trends and Open Problems**. IBM Research, New York, USA, 2017.

BALLA, D.; MALIOSZ, M.; SIMON, C.; GEBERER, D. 2020. **Tuning Runtimes in Open Source *FaaS***. Internet of Vehicles. Technologies and Services Toward Smart Cities. Springer Link. 250-266

BARCELONA-PONS, D.; SUTRA, P.; SÁNCHEZ-ARTIGAS, M.; PARÍS, G.; GARCÍA-LÓPEZ P. 2022. Stateful *serverless* computing with crucial. **ACM Transactions on Software Engineering and Methodology** 31, 3 (2022), 1-38.

BARESI, L.; GARRIGA, M. **Microservices: The Evolution and Extinction of Web Services?** 2019. Springer Link. Microservices.3-28.

BASU, S.; BARDHAN, A.; GUPTA, K.; et al. Cloud computing security challenges & solutions-A survey. **2018 IEEE 8th Annual Computing and Communication Workshop and Conference (CCWC)**, 2018. doi10.1109/CCWC.2018.8301700.

BERMBACH, D.; BADER, J.; HASENBURG J.; PFANDZELTER T.; THAMSEN L. 2022. AuctionWhisk: Using an auction-inspired approach for function placement in *serverless* fog platforms. **Software: Practice and Experience** 52, 5 (2022), 1143-1169.

BONER, J. 2016. **Reactive microservices architecture**. Editora O'Reilly Media

BRINZA. A. 2022. **Metodologia para Conteinerização de micro-serviços**. Dissertação de Mestrado. Universidade de Évora.

BUSHONG, V.; ABDELFATTAH, A.S.; MARUF, A.A.; DAS, D.; LEHMAN, A.; JAROSZEWSKI, E.; COFFEY, M.; CERNY, T.; FRAJATAK, K.; TISNOVSKY, P.; et al. On Microservice Analysis and Architecture Evolution: A Systematic Mapping Study. Appl. Sci. 2021, 11, 7856. https://doi.org/10.3390/app11177856 CADDEN, J.; UNGER, J.; AWAD, Y.; DONG, H.; KRIEGER, O.; APPAVOO, J. 2020. SEUSS: skip redundant paths to make *serverless* fast**. In Proceedings of the 15th European Conference on Computer Systems**. 1-15.

CANHIZARES FILHO, Valter; LOBATO, Renata Spolon. Serverless Computing - O data center agora é o computador. In: ESCOLA REGIONAL DE ALTO DESEMPENHO DE SÃO PAULO (ERAD-SP), 13., 2022, Online. **Anais** [...]. Porto Alegre: Sociedade Brasileira de Computação, 2022. p. 25-28. DOI: https://doi.org/10.5753/eradsp.2022.222115.

CANHIZARES FILHO, Valter et al. From cold starts to pricing: A comparative analysis in executing Node.JS application on *AWS* Lambda and Azure Functions. In: 2024 19th Iberian Conference on Information Systems and Technologies (CISTI). IEEE, 2024. A ser publicado.

CASSEL, G. A. S.; RODRIGUEZ, V. F.; RIGHI, R. R.; BEZ, M. R.; NEPOMUCENO, A. C.; DA COSTA, C. A. 2022. *Serverless* computing for Internet of Things: A systematic literature review. **In Future Generation Computer Systems**. Volume 218, p 299-316. Doi 10.1016/j.future.2021.10.020.

CASTRO, P.; ISHAKIAN, V; MUTHUSAMY, V.; SLOMINSKI. The rise of *serverless* computing. **Communications of the ACM**, 62 (12), 44-54, doi 10.1145/3368454

CERNY, T.; ABDELFATTAH, A. S.; BUSHONG, V.; MARUF, A. A.; TAIBI. **Microvision: Static analysis-based approach to visualizing microservices in augmented reality**. 2022. Doi 10.48550/arXiv.2207.02974

CNCF. The Cloud Native Computing Fundation, 2018. **CNCF WG-*Serverless* Whitepaper**. Disponível em https://github.com/cncf/wg-*serverless*/raw/master/whitepapers/*serverless*-overview/cncf_*serverless*_whitepaper_v1.0.pdf. Acesso em 19 jun. 2022.

COTRONEO, D.; FRATTINI, F.; PIETRANTUONO, R.; RUSSO, S. 2015. State-based robustness testing of IaaS cloud platforms. **In Proceedings of the 5th International Workshop on Cloud Data and Platforms**. 1–6.

DAW, N.; BELLUR, U.; KULKARNI P. 2020. Xanadu: Mitigating cascading cold starts in *serverless* function chain deployments. **In Proceedings of the 21st International Middleware Conference**. 356-370

DE PALMA, G.; GIALLORENZO, S.; MAURO, J.; ZAVATTARO, G. 2020. Allocation priority policies for *serverless* function-execution scheduling optimisation**. In Proceedings of the International Conference on Service-Oriented Computing**. Springer, 416-430.

EISMANN, S.; SCHEUNER, J.; VAN EIK, E. V.; SCWNINGER, M.; GROHMANN, J.; HERBST, N.; ABAD, C.; IOSUP, A. 2020. *Serverless* applications: Why, when, and how? **IEEE Software** 38, 1 (2020), 32-39.

EISMANN, S.; SCHEUNER, J.; VAN EIK, E. V.; SCWNINGER, M.; GROHMANN, J.; HERBST, N.; ABAD, C.; IOSUP, A. 2021. The state of *serverless* applications: collection, characterization, and community consensus. **IEEE Transactions on Software Engineering** (2021).

ESKANDANI, N.; SALVANESCHI, G. 2021. The wonderless dataset for *serverless* computing. **In Proceedings of 2021 IEEE/ACM 18th International Conference on Mining Software Repositories**. IEEE, 565-569.

EYK, E.; TOADLER, L.; TALLURI, S.; VERSLUIS, L.; UTA, A.; IOSUP, A. *Serverless* is more: From PaaS to Present Cloud Computing. **IEEE Internet Computing**, 22(5), 8-17, doi doi:10.1109/mic.2018.053681358

EYK E. IOSUP A., SEIF S., THOMMES M. The SPEC cloud group's research vision on *FaaS* and *serverless* architectures. **WoSC '17 proceedings of the 2nd. International Workshop on *Serverless* Computing**, pages 1-4, 2017.

FOULADI, S.; ROMERO, F.; ITER, D.; LI, Q.; CHATTERJEE, S.; KOZYRAKIS, C.; ZAHARIA, M.; WINSTEIN, K. 2019. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In **Proceedings of the 2019 USENIX Annual Technical Conference**. 475-488.

FUERST, A.; SHARMA, P. 2021. *FaaS*Cache: keeping *serverless* computing alive with greedy-dual caching. **In Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems**. 386-400.

GARTNER. **The CIO's guide to *serverless* computing**. Disponível em: https://www.gartner.com/smarterwithgartner/the-cios-guide-to-*serverless*-computing. Acesso em 25 de abril de 2023

GCP. **Google Cloud Platform**. Disponível em: cloud.google.com. Acesso em 9 jun. 2022.

GCP. **Google Cloud Functions**. Disponível em: cloud.google.com/Functions. Acesso em 14 jun. 2022.

HASSAN B.; BARAKAT, SAMAN A.; SARHAN, QUSAY I. Survey on *serverless* computing. **Journal of Cloud Computing**, v. 10, n. 1, 2021.

HOSEINYFARAHABADY, M.; CHOON LEE, Y.; ZOMAYA, A.; TARI, T. 2017. A qos-aware resource allocation controller for function as a service (*FaaS*) platform. **In Proceedings of the International Conference on Service-Oriented Computing**. Springer, 241-255.

IBM. **IBM Cloud**. Disponível em: cloud.ibm.com. Acesso em 9 jun. 2022.

INTEL. **O que é o Intel NUC?** Disponível em https://www.intel.com.br/content/www/br/pt/products/details/nuc.html. Acesso em 10 jun. 2023.

INTERSOG. **Cloud development vs traditional software development**. Disponível em: https://intersog.com/blog/cloud-development-vs-traditional-development/. Acesso em 20 abr 2023.

IVANOV, V**. Implementation of DevOps pipeline for *Serverless* Applications**. Tese de Mestrado, Aalto University, School of Science, Espoo, Finlandia, 2018. Disponível em: https://aaltodoc.aalto.fi/bitstream/handle/123456789/32432/master_Ivanov_Vitalii_2018.pdf Acesso em 19 jun 2022.

JANGDA, A.; PINCKNEY, D.; BRUN, Y.; GUHA, A. 2019. Formal foundations of *serverless* computing. **Proceedings of the ACM on Programming Languages** 3, 149 (2019), 1-26

JAMBUNATHAN, B.; YOGANATHAN, K. 2018. Architecture Decision on using Microservices or *Serverless* Functions with Containers. **Proceeding of 2018 IEEE International Conference on Current Trends toward Converging Technologies**, Coimbatore, India.

JIA, Z.; WITCHEL, E. 2021. Nightcore: efficient and scalable *serverless* computing for latency-sensitive, interactive microservices. **In Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems**. 152-166.

JINDAL, A.; FIELINGHAUS, J.; CHADHA, M.; GERNDT, M. 2021. Courier: delivering *serverless* functions within heterogeneous *FaaS* deployments**. In UCC '21: Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing**. Doi 10.1145/3468737.3494097

JONAS, E., SCHLEIER-SMITH, J., SREEKANTI, V., TSAI, C.-C., KHANDELWAL, A., PU, Q., SHANKAR, V., MENEZES CARREIRA, J., KRAUTH, K., YADWADKAR, N., GONZALEZ, J., POPA, R. A., STOICA, I., and PATTERSON, D. A. (2019). **Cloud programming simplified: A berkeley view on *serverless* computing**. Technical Report UCB/EECS-2019-3, EECS Department, University of California, Berkeley.

KAZANAVICIUS, J.; MAZEIKA, D. 2019. Migrating legacy software to microservices architecture. **In 2019 Open Conference of Electrical, Electronic and Information Sciences** (eStream), pages 1–5. IEEE.

KHAN, A. 2017. Key characteristics of a container orchestration platform to enable a modern application. **IEEE cloud Computing**, 4(5):42–48

KUMAR, S. N.; SELVAKUMARA, S. S. *Serverless* Computing Platforms Performance and Scalability Implementation Analysis**. 2022 International Conference on Computer, Power and Communications (ICCPC)**, Chennai, India, 2022, pp. 598-602, doi: 10.1109/ICCPC55978.2022.10072137.

KOOMEY J., TAYLOR J., 2017. **Zombie And Comatose Servers Redux**. Disponível em https://blog.anthesisgroup.com/zombie-servers-redux. Acesso em 19 jun. 2022

KOSCHEL, A.; KLASSEN, S.; JDIYA, K.; et al. Cloud Computing: *Serverless*. **2021 12th International Conference on Information, Intelligence, Systems & Applications (IISA)**, 2021. Disponível em: <https://ieeexplore.ieee.org/document/9555534>. Acesso em: 16 jun. 2022.

KHOUNG, V.; HARTLEY, K.; KANKANHALLIi, A. Predictors of cloud computing adoption: A cross-country study. **Telematics and Informatics**, v. 52, p. 101426, 2020. 10.1016/j.tele.2020.101426

LENARDUZZI, V.; PANICHELLA, A. 2020. *Serverless* testing: Tool vendors' and experts' points of view. **IEEE Software** 38, 1 (2020), 54-60.

LI, Y.; LIN, Y.; WANG, Y.; et al. *Serverless* Computing: State-of-the-Art, Challenges and Opportunities. **IEEE Transactions on Services Computing**, p. 1–1, 2022. Disponível em: <https://ieeexplore.ieee.org/document/9756233 >. Acesso em: 16 jun. 2022.

MANOR, E.; TEICH, O.; **Announcin Cloud Run, the newest member of our *serverless* compute stack**. 2019. Disponível em https://cloud.google.com/blog/products/*serverless*/announcing-cloud-run-the-newest-member-of-our-*serverless*-compute-stack Acesso em 22 jun. 2022.

MARKETSANDMARKETS.**Cloud Computing Market Size, Share and Global Market Forecast to 2026**. Disponível em: https://www.marketsandmarkets.com/Market-Reports/cloud-computing-market-234.html. Acesso em: 05 jun. 2022.

MELL, P. M.; GRANCE, T. **The NIST Definition of Cloud Computing**. Recommendations of the National Institute of Standards and Technology, 2011. Disponível em: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf

MOHANTY, S. K. **Evaluation of *Serverless* Computing Frameworks Based on Kubernetes**. Tese de Mestrado, Aalto University, School of Science, Espoo, Finlandia, 2018. Disponível em https://aaltodoc.aalto.fi/bitstream/handle/123456789/33680/master_Mohanty_Sunil_2018.pdf

MOREIRA, R.D.; BARRETO, P.S. A Framework for Improving Cold Start Time in Function-as-a-Service (*FaaS*). **2019 XLV Latin American Computing Conference (CLEI)**. doi 10.1109/clei47609.2019.235112

NEWMAN, S. 2021. **Building microservices**. Editora O'Reilly Media.

NETWORKINREVIEW. 2022. ***Serverless* architecture vs traditional architecture**. Disponível em: https://networkinterview.com/*serverless*-architecture-vs-traditional-architecture/. Acesso em 20 abr 2023.

*NODE.JS*. **About | *Node.JS***. Disponível em: nodejs.org/en/about. Acesso em: 21 jul. 2023.

NOVACONEXT. 2022. **What you need to know before implementing Infrastructure as a Service (IaaS)**. Disponível em: http://novacontext.com/what-you-need-to-know-before-implementing-iaas/. Acesso em 21 abr 2023.

NUPPONEN, J.; TAIBI, D. *Serverless*: What it Is, What to Do and What Not to Do. **2020 IEEE International Conference on Software Architecture Companion (ICSA-C)**, 2020.

OAKES, E.; YANG, L.; ZHOU, D.; HOUCK, K.; HARTER, T.; ARPACI-DUSSEAU, A.; ARPACI-DUSSEAU, R. 2018. SOCK: Rapid task provisioning with *serverless*-optimized containers. **In Proceedings of the 2018 USENIX Annual Technical Conference**. USENIX Association, 57-70

PAAKKUNAINEN, O. ***Serverless* computing and *FaaS* platform as a web application backend**. Tese de Mestrado, Aalto University, School of Science, Espoo, Finlandia, 2019. Disponível em: urn.fi/URN: NBN:fi:aalto-201906264219.

PFANDZELTER, T.; BERMBACH, D. tiny*FaaS*: A Lightweight *FaaS* Platform for Edge Environments. 2020. **In 2020 IEEE International Conference on Fog Computing (ICFC)**. Doi 10.1109/ICFC49376.2020.00011

RASHID, A.; CHATURVEDI, A. Cloud Computing Characteristics and Services: A Brief Review. **International Journal of Computer Sciences and Engineering JCSE**. Vol 7 Issue 2, (2019). E-ISSN 2347-2693

RANDAL, A. 2020. The ideal versus the real: Revisiting the history of virtual machines and containers. **ACM Computing Surveys** (CSUR), 53(1):1–31.

RANDALL, J. **Azure Functions – Significant Improvements in HTTP Trigger Scalling**. Disponível em https://www.azurefromthetrenches.com/azure-functions-significant-improvements-in-http-trigger-scaling/ Acesso em 22 jun. 2022

RESEARCHANDMARKETS. **Cloud Computing Services, Platforms Infrastructure and Everything as a Service 2022 - 2027**. Researchandmarkets.com. Disponível em: https://www.researchandmarkets.com/reports/5118343/cloud-computing-services-platforms. Acesso em: 8 jun. 2022.

RISTOV, S.; PEDRATSCHER, S.; WALLNOEFER, J.; FAHRINGER, T. 2020. Daf: Dependency-aware *FaaS*iier for *Node.JS* monolithic applications. **IEEE Software** 38, 1 (2020), 4853

ROBERTS, M., 2018. ***Serverless* Architectures**. Disponível em: martinfowler.com/articles/*serverless*.html. Acesso em 19 jun 2022.

SCHLEIER-SMITH, J.; SREEKANTI, V.; KHANDELWAL, A.; CARREIRA, J.; YADWADKAR, N. J., POPA, R. A., … PATTERSON, D. A. (2021). What *serverless* computing is and should become. **Communications of the ACM**, 64(5), 76–84. doi:10.1145/3406011

SAMPÉ, J.; GARCIA-LOPEZ, P.; SANCHEZ-ARTIGAS, S.; VERNIK, G.; ROCA-LLABERIA, P.; ARJONA, A. 2020. Toward multicloud access transparency in *serverless* computing. **IEEE Software** 38, 1 (2020), 68-74.

SHILLAKER, S.; PIETZUCH, P. 2020. *FaaS*m: Lightweight isolation for eicient stateful *serverless* computing. **In Proceedings of the 2020 USENIX Annual Technical Conference**. 419-433.

STEINBACH, Markus; JINDAL, Anshul; CHADHA, Mohak; et al. Tpp*FaaS*: Modeling *Serverless* Functions Invocations via Temporal Point Processes**. IEEE Access**, v. 10, p. 9059–9084, 2022. Disponível em: <https://ieeexplore.ieee.org/document/9684420 >. Acesso em: 16 jun. 2022.

SURYATEJA, P. S. Threats and Vulnerabilities of Cloud Computing: A Review. **International Journal of Computer Sciences and Engineering JCSE**. Vol 6. Issue 3, (2018), E-ISSN:2347-2693.

SUBASHINI S.; KAVITHA V**.** A Survey on security issues in service delivery models of cloud computing. **Journal of Network Computer Applications** (2010), doi10.1016/j.jnca.2010.07.006

TAIBI, D.; SPILLNER, J.; WAWRUCH, K. 2020. *Serverless* computing-where are we now, and where are we heading? **IEEE software** 38, 1 (2020), 2531.

UDUNWA U.; NSOR-ANABIAH, S.; KADAMBI, G.; MALATHI, S. Review of Cloud Computing in Low-Income Countries (LICs): Prospects and Challenges. **International Journal of Applied Engineering Research** (2019), ISSN 0973-4562, col. 14 (16) p. 3531-3536.

VARGHESE, B.; BUYYA, R. (2018). Next generation cloud computing: new trends and research directions. **Future Generation Computer Systems**, 79, 849–861. doi: 10.1016/j.future.2017.09.020

WANG, K. T. A.; HO, R.; WU, P. 2019. Replayable execution optimized for page sharing for a managed runtime environment. **In Proceedings of the Fourteenth EuroSys Conference 2019**. 1–16

WEN, J.; CHEN, Z.; XIM, J.; LIU, X.; Rise of the Planet of *Serverless* Computing: A Systematic Review. **ACM Transactions on Software Engineering and Metodology**. Doi: 10.1145/3579643

YUSSUPOV, V.; SOLDANI, J.; BREITENBÜCHER U.; LEYMANN, F. 2022. Standards-based modeling and deployment of *serverless* function orchestrations using BPMN and TOSCA. **Software: Practice and Experience** (2022).

ZHANG, M.; KRINTZ, C.; WOLSKI R. 2021. Edge-adaptable *serverless* acceleration for machine learning Internet of Things applications. **Software: Practice and Experience** 51, 9 (2021), 1852-1867.

ZHANG, H.; TANG, Y.; KHANDELWAL, A.; CHEN, J.; STOICA, I. 2021. Caerus: NIMBLE task scheduling for *serverless* analytics. **In Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation**. 653-669.

ZUK, P.; RZADCA, K. 2020. Scheduling methods to reduce response latency of function as a service**. In Proceedings of the 2020 IEEE 32nd International Symposium on Computer Architecture and High-Performance Computing**. 132–140.