

## RESUMO

Diante dos avanços alcançados na computação ao longo dos últimos anos, foram desenvolvidas tecnologias com foco no desenvolvimento de aplicações, e na gestão de recursos de *softwares*. Dentre estas inovações estão a *Cloud Computing*, que teve grandes evoluções com a criação do *Serverless* e do *Function-as-a-Services* (FaaS), paradigmas de computação que proporcionam a simplificação dos desenvolvimentos de *software* e da abstração de complexidades arquiteturais. Essas soluções oferecem performance, escalabilidade automática, gestão dos serviços e servidores, e a otimização de custos, favorecendo equipes de desenvolvimentos a trabalharem juntas e direcionarem o foco a novas melhorias e inovações. Diante desses benefícios, observa-se uma transição crescente de arquiteturas tradicionais para modelos baseados em *microservices* ou *serverless*, com objetivo de alcançarem redução de custos, resiliência, e o desenvolvimento de aplicações mais eficientes e ágeis. Essa transição também é impulsionada pela necessidade das organizações de responderem rapidamente às demandas do mercado, garantindo flexibilidade e adaptabilidade às mudanças. Além disso, a integração dessas tecnologias com outras ferramentas e serviços, como bancos de dados *NoSQL*, soluções de inteligência artificial e *machine learning*, amplificam ainda mais as possibilidades de inovação e criação de soluções robustas. No entanto, apesar dos benefícios evidentes, a adoção de paradigmas *serverless* e FaaS apresentam desafios, como a gestão adequada de recursos, segurança e monitoramento. Assim, este trabalho busca explorar profundamente esses temas, analisando casos de uso, vantagens, desafios e melhores práticas na implementação de arquiteturas *serverless* e FaaS no contexto atual da computação em nuvem. Neste estudo, é desenvolvido um *software* aplicando o modelo *serverless* por meio do FaaS, utilizando-se da plataforma em nuvem da *Amazon Web Service* (AWS). Foram utilizadas ferramentas como *API Gateway*, *Amazon Lambda Function* e *DynamoDB*, todos componentes *serverless* da AWS. Utilizou-se a Calculadora de Preços e o Gerenciamento de Custos da AWS para evidenciar os custos previstos e obtidos nesta implementação. A aplicação dessas tecnologias demonstrou eficácia ao otimizar o desenvolvimento de *softwares*, proporcionando resiliência, eficiência financeira e facilitando a manutenção e evolução do código com foco em inovação.

**Palavras chaves:** *Cloud computing*, Computação em nuvem, *microservices*, microsserviços, *serverless*, *Function-as-a-Services*, FaaS, desenvolvimento, *software*, AWS, *Lambda Function*, *NoSql*, *DynamoDB*

## ABSTRACT

Given the advancements in computing over recent years, technologies have emerged focusing on application development and software resource management. Among these innovations is Cloud Computing, which has significantly evolved with the introduction of Serverless and Function-as-a-Service (FaaS) models. These computing paradigms simplify software development and abstract architectural complexities. Such solutions offer performance, automatic scalability, service and server management, and cost optimization, enabling development teams to collaborate effectively and concentrate on innovation. Consequently, there's a growing shift from traditional architectures to microservices or serverless models, aiming for cost reduction, resilience, and the creation of more efficient and agile applications. This transition is also driven by organizations' need to swiftly respond to market demands, ensuring flexibility and adaptability. Additionally, integrating these technologies with tools like NoSQL databases, artificial intelligence, and machine learning further enhances innovation possibilities and robust solution development. However, despite the evident benefits, adopting serverless and FaaS paradigms poses challenges such as resource management, security, and monitoring. This research delves deep into these aspects, examining use cases, advantages, challenges, and best practices for implementing serverless and FaaS architectures in today's cloud computing landscape. In this study, a software is developed using the serverless model via FaaS on the Amazon Web Service (AWS) cloud platform. Tools such as API Gateway, Amazon Lambda Function, and DynamoDB—all serverless components of AWS—were employed. The AWS Pricing Calculator and Cost Management tools were utilized to highlight the expected and actual costs of this implementation. Implementing these technologies proved effective in optimizing software development, ensuring resilience, financial efficiency, and facilitating code maintenance and evolution with an emphasis on innovation.

**Keywords:** Cloud computing, microservices, serverless, Function-as-a-Service, FaaS, software development, AWS, Lambda Function, NoSQL, DynamoDB.

## LISTA DE ILUSTRAÇÕES

<b>Figura 1</b> – Mark I	16
<b>Figura 2</b> - Matéria sobre o Harvard Mark I	17
<b>Figura 3</b> - Harvard Mark 1	18
<b>Figura 4</b> – Matéria The New York Times aclamando o ENIAC	19
<b>Figura 5</b> - ENIAC em operação	20
<b>Figura 6</b> - Representação visual da difusão da ARPANET em 1974	21
<b>Figura 7</b> - Mapas de cabos submarinos da conexão global de Internet	23
<b>Figura 8</b> - Arquitetura de referência da cloud computing NIST	25
<b>Figura 9</b> - Exemplos de serviços disponíveis para um consumidor de nuvem	27
<b>Figura 10</b> - Monólitos e Microserviços	29
<b>Figura 11</b> - Descentralização de Dados	30
<b>Figura 12</b> - Pesquisa IBM Market Development & Insights sobre os desafios e benefícios da adoção de microserviços	31
<b>Figura 13</b> - Abordagens de cloud computing comparadas a viagens do aeroporto: Serverful como aluguel de carro e Serverless como pegar um táxi	32
<b>Figura 14</b> - Serverless vs. Serverful: o Serverless oferece uma abstração entre as aplicações e os servidores subjacentes	34
<b>Figura 15</b> - Serverless vs. Serverful: na computação Serverless, os usuários pagam apenas pelos recursos consumidos, não pela capacidade ociosa reservada	35
<b>Figura 16</b> - Acionamento de Lambda Function por API Gateway	37
<b>Figura 17</b> - A evolução das arquiteturas de software	38
<b>Figura 18</b> - O crescimento do ecossistema de software sem servidor de 2014 a 2020	39
<b>Figura 19</b> - API Gateway – Endpoints e Rotas da API RESTful	41
<b>Figura 20</b> - Configurações de solicitação de método GET	42
<b>Figura 21</b> - Arquitetura da Aplicação	43
<b>Figura 22</b> - Lambda Function ProjectTccRead (GET) e Gatilho via API Gateway	47
<b>Figura 23</b> - Configurações da tabela no DynamoDB	48
<b>Figura 24</b> - Métricas CloudWatch – API Gateway	49
<b>Figura 25</b> - Efeitos Cold Start no primeiro acionamento	50
<b>Figura 26</b> - Gerenciamento de Custos AWS desta pesquisa	51
<b>Figura 27</b> - Implementação da Lambda Function Read (GET)	52
<b>Figura 28</b> - Implementação dos métodos findUser e findAllUser	53
<b>Figura 29</b> - Implementação do método findUserByLicensePlate	54
<b>Figura 30</b> - Lista de Lambdas Function por operações do CRUD	56
<b>Figura 31</b> - Cálculo de Custos do Projeto na Calculadora de preços da AWS	57

**LISTA DE TABELAS****Tabela 1** - Tabela de descrição dos endpoints

45

## LISTA DE SIGLAS

ENIAC	<i>Electronic Numerical Integrator and Computer</i>
TCP/IP	<i>Transmission-Control Protocol / Internet Protocol</i>
IBM	<i>International Business Machines Corporation</i>
ASCC	<i>Automatic Sequence Controlled Calculator</i>
EUA	Estados Unidos da América
ENIAC	<i>Electronic Numerical Integrator and Computer</i>
ARPA	<i>Advanced Research Projects Agency</i>
ARPANET	<i>Advanced Research Projects Agency Network</i>
FTP	<i>File Transfer Protocol</i>
NCP	<i>Network Control Protocol</i>
NSF	<i>National Science Foundation</i>
NASA	<i>National Aeronautics and Space Administration</i>
NSFNET	<i>National Science Foundation Network</i>
CIX	<i>Commercial Internet Exchange</i>
WWW	<i>World Wide Web</i>
CSP	<i>Cloud Services Provider</i>
GCP	<i>Google Cloud Platform</i>
NIST	<i>National Institute of Standards and Technology</i>
IaaS	<i>Infrastructure-as-a-Service</i>
PaaS	<i>Platform-as-a-Service</i>
SaaS	<i>Software-as -a-Service</i>
FaaS	<i>Function-as-a-Service</i>
BaaS	<i>Backend-as-a-Service</i>
API	<i>Application Programming Interface</i>
HTTP	<i>Hypertext Transfer Protocol</i>
TI	Tecnologia da Informação
AWS	<i>Amazon Web Services</i>
IA	Inteligência Artificial
REST	<i>Representational State Transfer</i>
RESTful	<i>Representational State Transfer Full</i>

## SUMÁRIO

<b>1</b>	<b>INTRODUÇÃO .....</b>	<b>11</b>
1.1	OBJETIVO GERAL .....	14
1.2	OBJETIVOS ESPECÍFICOS .....	14
1.3	JUSTIFICATIVA .....	15
1.4	METODOLOGIA .....	15
<b>2</b>	<b>FUNDAMENTAÇÃO TEÓRICA .....</b>	<b>16</b>
2.1	UM BREVE HISTÓRICO DOS COMPUTADORES .....	16
2.2	AS REDES DE COMPUTADORES .....	20
2.3	CLOUD COMPUTING .....	23
2.4	MICROSERVICES .....	27
2.5	SERVERLESS .....	32
2.6	FUNCTION AS A SERVICES .....	36
<b>3</b>	<b>DESENVOLVIMENTO .....</b>	<b>41</b>
3.1	SÍNTESE DA PROPOSTA .....	41
3.2	ARQUITETURA DO SISTEMA .....	42
3.3	API GATEWAY .....	43
3.4	LAMBDA FUNCTION .....	46
3.5	BANCO DE DADOS NOSQL .....	47
3.6	MONITORAMENTO E LOGGING .....	48
3.7	CUSTOS AWS .....	50
3.8	IMPLEMENTAÇÃO DE CÓDIGO .....	51
<b>4</b>	<b>RESULTADOS .....</b>	<b>55</b>
4.1	RESULTADOS DO DESENVOLVIMENTO .....	55
<b>5</b>	<b>CONCLUSÃO .....</b>	<b>59</b>
5.1	TRABALHOS FUTUROS .....	59
<b>6</b>	<b>REFERÊNCIAS BIBLIOGRÁFICAS .....</b>	<b>60</b>

## 1. INTRODUÇÃO

Com o advento da tecnologia nos últimos anos, tivemos grandes evoluções no que diz respeito a desenvolvimento e hospedagem de sistemas, com o surgimento do *Cloud Computing*, *Microservices* e *Serverless*.

Os avanços se dão desde o início dos computadores na década de 30, quando iniciaram o desenvolvimento do *Mark I*, seguindo para o *Electronic Numerical Integrator and Computer* (ENIAC), supercomputadores que revolucionaram a tecnologia da época, com a capacidade de solucionar cálculos e projetos com uma velocidade nunca vista para aquele período, ajudando principalmente forças militares no avanço de tecnologias e desenvolvimento de armamentos, trajetória de projéteis balísticos, radares, energia atômica, e indo além ajudando no desenvolvimento e pesquisa acadêmica (JOHN A. PAULSON SCHOOL OF ENGINEERING & APPLIED SCIENCE, 2023; PENN ENGINEERING, 2017; RICHEY, 1997).

Após estes grandes computadores, surgiu o desejo pela conexão entre vários computadores, iniciando-se a primeira rede de computadores, que foi inicialmente desenvolvida como uma rede local, interligando computadores acadêmicos e computadores militares utilizando-se de linhas telefônicas (FEATHERLY, 2023).

A *Advanced Research Projects Agency Network* (ARPANET), foi o nome dado a esse projeto com objetivo de criar a conexão entre os computadores das instituições governamentais e acadêmicas. Após esse projeto, foram iniciados vários outros projetos, e deram início às conexões entre países, originando a primeira rede de computadores (FEATHERLY, 2023; KAHN; DENNIS, 2023).

A ARPANET foi um hub de desenvolvimento para as inovações no decorrer dos seus 10 anos iniciais, criando novos aplicativos e protocolos que eram constantemente preparados e implantados na rede. Partindo do desejo pela conexão as outras redes, foi desenvolvido um novo protocolo de comunicação de computador, que ficou conhecido como *Transmission-Control Protocol / Internet Protocol* (TCP/IP), sendo testado na ARPANET em 1977, possibilitando uma rede transferir pacotes de dados para outra rede externa (FEATHERLY, 2023).

Com o ascendente avanço dos serviços e aplicações para internet, a sua comercialização foi acelerada. Junto da expansão da internet teve um grande crescimento no

desenvolvimento de novos aplicativos como um novo tipo de programa para computador. Então surgiram os navegadores, com interface de “apontar-e-clicar”, facilitando o uso por parte dos clientes comerciais. Seguindo os avanços, os navegadores foram melhorados para uma nova versão da internet, acelerando o crescimento da internet.

Com o crescimento acelerado da Internet e das empresas do meio, foi-se tendo uma grande adesão, gerando um grande volume de usuários, junto a isso os portais de internet conduziram as taxas de publicidades devido ao elevado número de acessos. Paralelo a isto, o mercado tentava acompanhar estes avanços com base em grandes especulações de mercado, gerados pelas publicidades nos sites, disponibilizando serviços gratuitos ou de baixo custo de vários tipos que foram visualmente aumentados com anúncios, causando no período o conhecido *dot-com-bubble* (KAHN; DENNIS, 2023; DUIGNAN, 2023).

Após esses avanços, surgiu o que se chama de “*Web 2.0*”, uma internet voltada para as redes sociais e a computação em nuvem. Esta modernização possibilitou novos modelos de negócios para o uso da web semântica. Um modelo de negócios criado nesta nova fase foi os alugueis de plataformas de computação de *hardware* e *software* pela internet que foi denominado de *Cloud Computing*. Surgindo a partir da convicção que a *Cloud Computing* fosse uma tendência futura prevista da computação, prometendo muitos benefícios, como nenhuma despesa de capital e baixo custo operacional, facilidade na manutenção e com rapidez na implantação reduzindo o tempo de lançamento no mercado (KAHN; DENNIS, 2023; GUHA; AL-DABASS, 2010).

*Cloud computing* são recursos de computação como serviços, que são disponibilizados via internet e sob demanda. São aplicativos, servidores (físicos ou virtuais), armazenamento de dados, ferramentas de desenvolvimento, recursos de rede, dentre outros que são hospedados em um data center remoto gerenciado por um *Cloud Services Provider* (CSP), que é um provedor de serviços em nuvem. Suprimindo tais necessidades, as empresas não precisam adquirir, configurar ou gerenciar a infraestrutura. Desta forma, o provedor disponibiliza esses recursos por uma assinatura mensal ou por um valor cobrado conforme o uso (IBM, 2023, GOOGLE CLOUD, 2023).

Na última década, observou-se a evolução dos paradigmas da computação. O *Cloud Computing* foi o mais popular dentre os demais, sendo um paradigma desenvolvido para utilizar a “computação como utilidade”, permitindo o desenvolvimento de novos produtos e serviços de Internet (DONNO et al., 2019; ARMBRUST et al., 2010).



A demanda por processamento de dados tem-se expandido para diversas áreas, nos últimos anos, e a *cloud computing* tem se provado como uma solução adequada para o processamento de dados, pois é um paradigma de computação de alto desempenho que disponibiliza seus serviços através da internet, executando grandes aplicações comerciais e científicas (DUBEY et al., 2019; OWENS, 2010; NASR et al., 2018).

A arquitetura de microsserviços é composta por pequenos serviços segregados que se comunicam, geralmente por *Application Programming Interface* (API) *Hypertext Transfer Protocol* (HTTP). Cada serviço é desenvolvido e implantado de maneira autônoma, permitindo flexibilidade em linguagens e tecnologias. O gerenciamento é descentralizado, e os serviços de armazenamento podendo ser adaptados individualmente para cada aplicação. (FOWLER; LEWIS, 2014).

Quando adotado uma arquitetura de *microservices*, as equipes de desenvolvimento podem atuar juntas, sem interferências no trabalho da outra, apresentando um ganho de produtividade, pois as equipes alcançaram bons resultados trabalhando simultaneamente (REDHAT, 2023).

O *serverless* é um modelo de *cloud computing* que permite equipes de desenvolvimento focarem exclusivamente na lógica de negócios, deixando gestão de recursos na responsabilidade dos provedores de nuvem. O modelo se utiliza de funções como serviço para executar processos de negócios, integrando-se a serviços de *backend* como bancos de dados (STEINBACH et al., 2022; TAIBI et al., 2021).

As arquiteturas *serverless* têm revolucionado o cenário de soluções em nuvem, proporcionando simplificação nos *pipelines* de desenvolvimento e abstração de complexidades arquitetônicas. Essa abordagem oferece escalabilidade automática conforme a demanda, promovendo redução de custos e aliviando sobrecargas para desenvolvedores. Diante desses benefícios, observa-se uma transição crescente de arquiteturas tradicionais para modelos baseados em "*microservices*" ou *serverless*, conforme destacado por (CHRISTIDIS et al., 2020; FOX et al., 2017).

Desde a introdução do AWS Lambda ao mercado, a computação *serverless*, baseada no modelo *Function-as-a-Services* (FaaS), essa tecnologia tem tido grande relevância em diversas áreas, como na acadêmica e corporativa. Grandes provedores de nuvem também incorporaram essa tecnologia em seus serviços, apresentando soluções como *Google Cloud Functions* e *Microsoft Azure Functions*. O modelo FaaS possibilita a decomposição de

aplicações convencionais em pequenas funções sem estado, executadas em contêineres com tempo de execução específico em plataformas *serverless* (STEINBACH et al., 2022).

No contexto da arquitetura de *microservices*, o modelo FaaS tem ganhado destaque significativo desde sua concepção, pois este modelo, ao permitir a execução de tarefas segmentadas, destaca-se especialmente na otimização do desempenho das aplicações. A principal motivação para a adoção da tecnologia *serverless* reside na redução de custos, diante da busca pela redução dos altos investimentos em infraestrutura, como a subutilização de recursos em máquinas virtuais. Além disso, estratégias de balanceamento de carga e otimização de custos surgem como componentes cruciais, evidenciando a eficácia da arquitetura *serverless* para economizar em diversas etapas, desde design até operações de leitura/escrita (CARVALHO; ARAÚJO, 2023; ZIMMERMANN, 2016; BARRAK et al., 2022; WANG et al., 2019; JARACHANTHAN et al., 2021; CHAHAL et al., 2020).

Tendo em vista estes grandes avanços da computação, propõe-se a seguinte questão de pesquisa: Como desenvolver uma aplicação no contexto de *cloud* fazendo uso da tecnologia *serverless* baseada no modelo FaaS diante dos desafios existentes na adoção?

Assim, esse estudo se constitui no desenvolvimento de um sistema no cenário de um estacionamento, com a aplicação direcionada a segregar as operações que existiriam em uma aplicação monolítica para que possa ser avaliado o contexto do *software* e construindo uma nova arquitetura pensada no contexto de funções como serviço, fazendo a adoção do *serverless*, fornecendo os benefícios embargados nessa tecnologia, favorecendo que empresas possa considerar a possibilidade de migrar seus sistemas para este modelo, alcançando alta disponibilidade, resiliência e principalmente a redução com os custos dos produtos.

### 1.1. Objetivo Geral

Desenvolver uma aplicação de um sistema de gerenciamento para estacionamentos utilizando tecnologia *serverless* com *lambda function*.

### 1.2. Objetivos Específicos

- Realizar o levantamento bibliográfico
- Desenvolver a arquitetura do sistema e construir a estrutura de Banco de Dados
- Desenvolver e Implantar as *Lambdas Functions* e a *API Gateway*

### 1.3. Justificativa

Com o avanço obtido nos últimos anos sobre as aplicações em nuvem, foi desenvolvido o conceito de *serverless*, abrangendo uma gama de serviços que não há necessidade de serem mantidos por um desenvolvedor, ficando na responsabilidade do próprio servidor escalar e manter uma aplicação em alta disponibilidade, reduzindo os custos com a operação.

Diante deste avanço, surgiu a *Lambda Function*, uma tecnologia desenvolvida sobre o conceito de *serverless*, sendo ela uma tecnologia que é acionada por eventos, trazendo consigo os benefícios do paradigma.

Com o pensamento pela busca da inovação e redução de custos, surgiu a proposta de estudo visando desenvolver uma aplicação para estacionamento, contendo as funcionalidades base do conceito de *Minimum Viable Product* (MVP), com a implementação das operações *Create, Read, Update e Delete* (CRUD).

### 1.4. Metodologia

Esse estudo será desenvolvido em 5 capítulos, assim distribuídos:

Capítulo 1 – Introdução, apresentando o tema de forma geral da pesquisa;

Capítulo 2 – Fundamentação Teórica, em que serão abordados a história e os conceitos básicos de cada ponto principal da aplicação, pensamentos e autores que embasaram o estudo através do levantamento bibliográfico;

Capítulo 3 – Desenvolvimento, no qual é dividido em 5 etapas: I) Síntese da proposta; II) Arquitetura do sistema; III) Descrição da *API Gateway*; IV) Descrição das *Lambdas Function*; V) Descrição do Banco de Dados *NoSQL*; VI) Descreve os monitoramentos e logs; VII) Descreve a implementação do código.

Capítulo 4 – Resultados, resultados da aplicação no conceito *serverless* com *lambda function* e uma visão geral dos procedimentos e testes adotados e suas respostas.

Capítulo 5 – Conclusão, é detalhado o trabalho no todo, com uma síntese dos resultados obtidos. Finalizando, apresentam-se as referências;

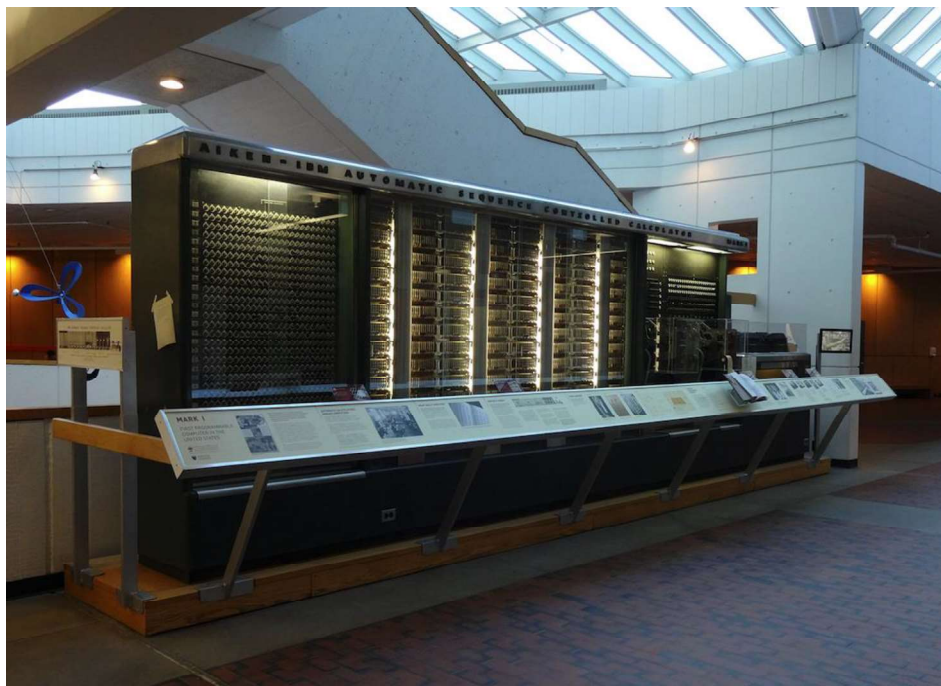
## 2. FUNDAMENTAÇÃO TEÓRICA

Neste capítulo serão apresentados marcos importantes para história da tecnologia da informação destacando uma trajetória desde o início dos computadores até a computação em nuvem e micros serviços com seus tipos e características, e a tecnologia sem servidor com suas aplicações e soluções.

### 2.1. Um breve histórico dos Computadores

Segundo a John A. Paulson School Of Engineering & Applied Science (2023), Aiken et al. (1947) e Ibm (2023), a história dos primeiros computadores iniciou-se em meados da década de 30, no ano de 1937, quando começou-se o desenvolvimento do *Harvard Mark I*, que foi originalmente chamado pela *International Business Machines* (IBM) de “*Automatic Sequence Controlled Calculator*” (ASCC), e muitas vezes referido como “*Calculadora Harvard*”, um projeto que foi liderado pelo estudante de graduação de *Harvard Howard H. Aiken*, com o propósito de ser o primeiro computador eletromecânico com a capacidade de solucionar problemas avançados de física matemática sem a interferência humana, sendo concluído no ano de 1944.

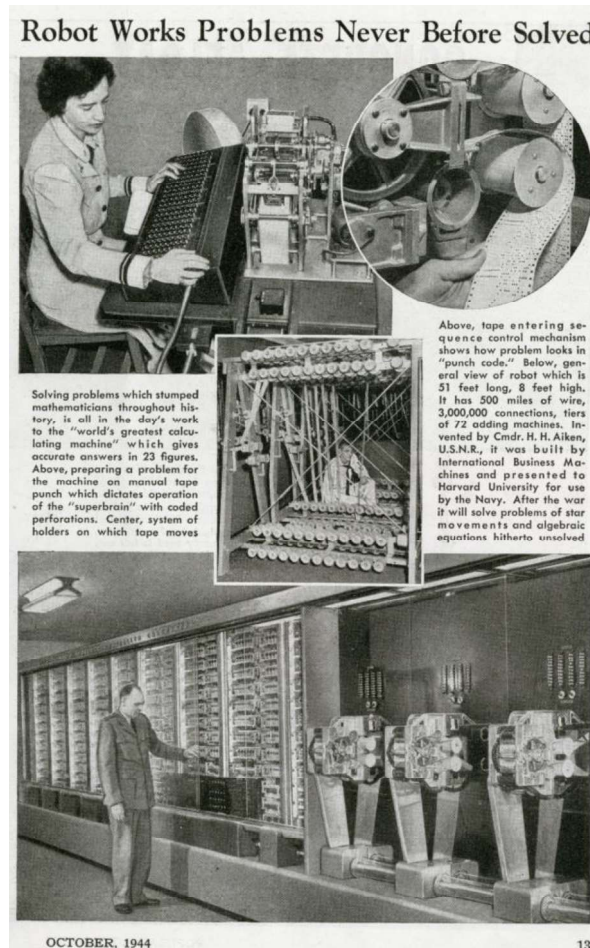
**Figura 1 – Mark I**



Fonte: JOHN A. PAULSON SCHOOL OF ENGINEERING & APPLIED SCIENCE, 2023

As operações do *Mark I* eram diárias, 24 horas por dia, e duraram de 1944 a 1959. O supercomputador da época teve como finalidade atividades voltadas para projetos militares, contribuindo junto a Marinha dos Estados Unidos da América (EUA) na realização do projeto de equipamentos como torpedos e sistemas de detecção subaquática, entre outros projetos em conjunto as Forças Armadas do EUA no desenvolvimento de lentes de câmeras de vigilância, radar e dispositivos de implosão para a bomba atômica no Projeto *Manhattan*, realizando cálculos de grandes tabelas matemáticas, como a resolução de um conjunto de equações diferenciais chamadas Funções de Bessel, sendo apelidado de “*Bessie*” (JOHN A. PAULSON SCHOOL OF ENGINEERING & APPLIED SCIENCE, 2023).

**Figura 2** - Matéria sobre o Harvard Mark I



Fonte: JOHN A. PAULSON SCHOOL OF ENGINEERING & APPLIED SCIENCE,  
2023

Os membros da tripulação no decorrer dos primeiros dois anos de operação aprenderam como usar e manter o Mark I com segmentos improvisados e aprendizado. Após a guerra, foi designado a Grace Hopper a responsabilidade de reunir todo o conhecimento relacionado à máquina que estava desorganizado e publicasse um manual de instruções, tarefa que ela se referiu ao projeto como uma “bíblia” do computador. Aplicou-se um tratamento minucioso aos componentes físicos, operação e manutenção do Mark I. O manual resultou no primeiro livro didático de programação de computadores e foi descrito cuidadosamente sobre como programar o Mark I (JOHN A. PAULSON SCHOOL OF ENGINEERING & APPLIED SCIENCE, 2023).

**Figura 3 - Harvard Mark 1**



Fonte: IBM, 2023

No dia 14 de fevereiro de 1946, durante o período da Segunda Guerra Mundial, foi apresentado ao mundo o projeto *Electronic Numerical Integrator and Computer* (ENIAC), projetado pelos John W. Mauchly e John P. Eckert Jr., ambos da Escola Moore de Engenharia Elétrica da Universidade da Pensilvânia, e este foi o primeiro computador digital eletrônico programável de uso geral no mundo (FREIBERGER; SWAINE, 2023).

Aclamado pelo *The New York Times* como “uma máquina incrível que aplica velocidades eletrônicas pela primeira vez a tarefas matemáticas até então muito difíceis e complicadas para serem resolvidas”, conforme apresentado pela *Penn Engineering* da



Universidade Pensilvânia, o ENIAC foi um computador voltado a solucionar problemas militares, resolvendo cálculos mais complexo muito mais rápidos, pois utilizava de recentes tecnologias mecânicas e tubos de vácuo (PENN ENGINEERING, 2017).

**Figura 4** – Matéria *The New York Times* aclamando o ENIAC



Fonte: THE NEW YORK TIMES, 1946

Segundo Richey (1997), o uso do Eniac nas Forças Militares dos EUA se deu primeiramente na resolução de problemas de energia atômica para o Projeto Manhattan, e ainda no seu primeiro ano foi utilizado na solução de cálculos de trajetórias balísticas para o Departamento de Artilharia, como também para pesquisa e solução dos problemas relacionados a previsão do tempo, estudos de raios cósmicos em astronomia, estudos de números aleatórios e projeto de túneis de vento.

Demais mudanças foram sendo executadas nos anos posteriores, como no ano de 1952 que foi adicionado um câmbio eletrônico de alta velocidade que aumentou em oitenta por cento a performance do computador. Uma unidade de memória central com núcleo magnético de cem palavras foi instalada no ano seguinte pela *Burroughs Corporation*. Sendo o último aperfeiçoamento realizado no ENIAC, concedeu ao supercomputador a capacidade de armazenar maiores quantidades de informação num banco central de memória de acesso aleatório. Anteriormente, o armazenamento temporário era muito pequeno e limitando os tipos de problemas que o ENIAC poderia solucionar, pois ficava no acumulador. E desde a sua criação até obsolescência, durante toda sua época de atuação, o ENIAC foi uma máquina revolucionária (RICHEY, 1997).

**Figura 5-** ENIAC em operação



Fonte: FREIBERGER; SWAINE, 2023

## **2.2. As redes de computadores**

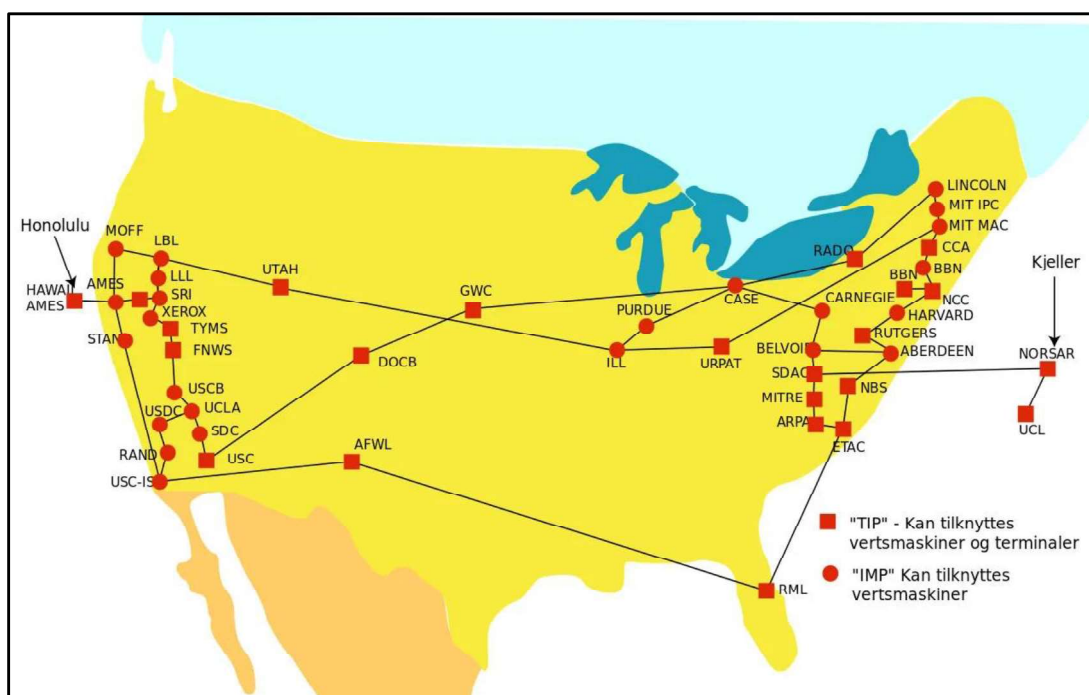
Ao final da década de 60, a *Advanced Research Projects Agency* (ARPA), uma divisão do Departamento de Defesa dos EUA, deu o início ao desenvolvimento da primeira rede de computadores, a *Advanced Research Projects Agency Network* (ARPANET). Este projeto tinha inicialmente como principal objetivo interligar computadores em instituições de investigação financiadas pelo Pentágono através de linhas telefônicas, criando assim uma rede interna de computadores (FEATHERLY, 2023).



Segundo Featherly (2023), mesmo o objetivo não sendo militar e sim acadêmico, a ARPANET avançou em suas instalações graças às conexões acadêmicas, e então, tornou-se uma estrutura militar como o tentáculo, um desejo dos oficiais militares.

Dessa forma, após décadas de trabalhos em projetos de desenvolvimentos de sistemas de comunicações e observações contínuas, o novo diretor da ARPANET, Robert Taylor, tornou público, em outubro de 1967, num Simpósio, um plano para a construção de uma rede de computadores em que ligaria 16 universidades e centros de pesquisas que eram patrocinados pela ARPA nos Estados Unidos. (FEATHERLY, 2023)

**Figura 6** - Representação visual da difusão da ARPANET em 1974



Fonte: FEATHERLY, 2023

Segundo Featherly (2023), o projeto de conectar computadores e pesquisadores, surgiu a partir do interesse em compartilhar informações a grandes distâncias sem a necessidade de conexões telefônicas entre cada computador de uma rede, exigindo a implementação da comutação de pacotes para alcançar este objetivo.

Durante os 10 anos iniciais, a ARPANET foi um hub para desenvolver e testar as inovações. Novos aplicativos e protocolos como *Telnet*, *File Transfer Protocol* (FTP) e

*Network Control Protocol* (NCP) foram constantemente desenvolvidos, testados e implantados na rede. Com o desejo de conecta-se as outras redes, foi pensado em um novo protocolo de comunicação de computador, um *gateway* entre redes, que acabou ficando conhecido como TCP/IP, sendo testado na ARPANET em 1977, e foi maneira pela qual uma rede poderia transferir pacotes de dados para outra rede externa (FEATHERLY, 2023).

O Padrão TCP/IP, foi adotado pelo Departamento de Defesa dos EUA em 1980, e outros órgãos governamentais dos EUA entraram com colaboração, tais como, a *National Science Foundation* (NSF), e a *National Aeronautics and Space Administration* (NASA), através de financiamentos, pesquisas, expansão, e desenvolvimento de outras redes, como *National Science Foundation Network* (NSFNET) para promover uma rede de educação e pesquisa nos Estados Unidos, como também foi criado o *Commercial Internet Exchange* (CIX) para o tráfego de informações comerciais (KAHN; DENNIS, 2023).

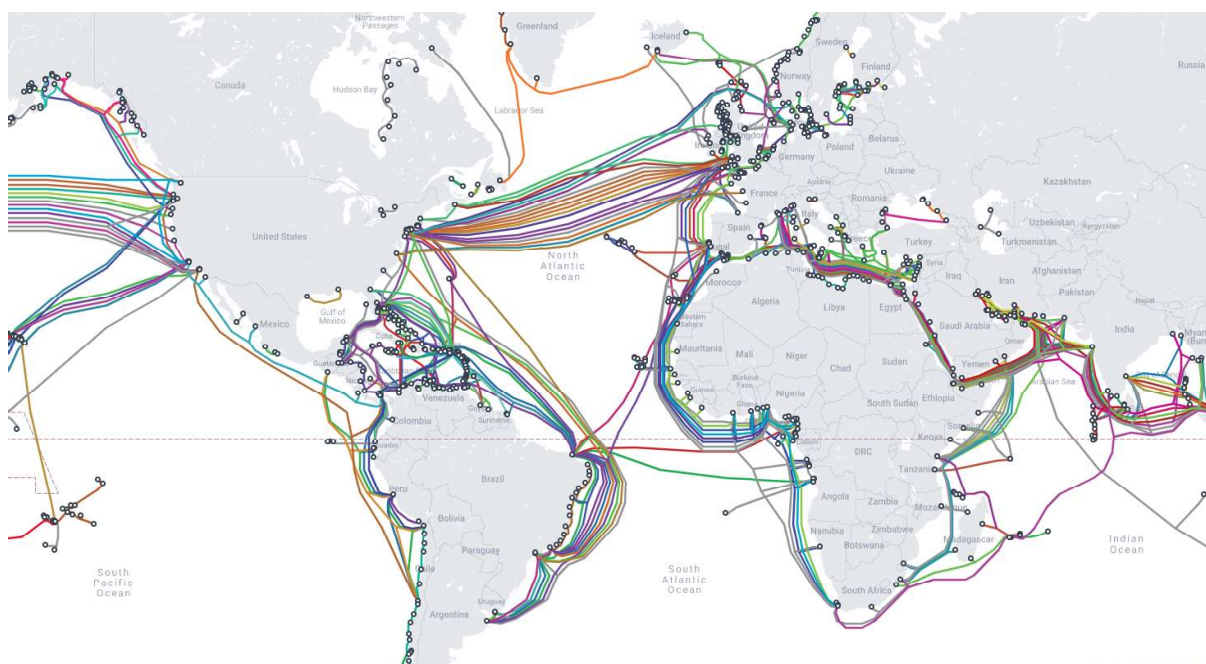
Segundo Kahn e Dennis (2023), Norr (2023), a ascensão dos serviços e aplicações comerciais da internet contribuiu de maneira acelerada para a comercialização da Internet, sendo esse fenômeno um resultado de vários outros fatores. Seguindo estes avanços, a expansão da internet teve um crescimento extraordinário com o desenvolvimento de novos programas que foram disponibilizados como um novo tipo de programa para computador. O Mosaic, conhecido como navegador, que possibilitou o acesso simples através de sua interface “apontar-e-clicar”. O programa então foi melhorado para uma nova versão da internet, o *World Wide Web* (WWW), dando origem aos sucessores *Netscape* e *Internet Explorer* acelerando o crescimento da internet (KAHN; DENNIS, 2023; NORR, 2023).

Ao final de 1990 com a expansão de provedores, a internet teve muitos acessos, e junto a isso os portais de internet conduziram as taxas de publicidades devido ao elevado número de visitantes aos sites, fazendo com que as receitas publicitárias tornassem-se o principal objetivo de muitos sites da Internet, dos quais alguns começaram a especular, disponibilizando serviços gratuitos ou de baixo custo de vários tipos que foram visualmente aumentados com anúncios causando no período entre o final dos anos 90, e início dos anos 2000, o tal colapso, chamado, a *dot-com bubble* (KAHN; DENNIS, 2023; DUGNAN, 2023).

Após esse advento da *dot-com bubble*, surgiu o então chamado “*Web 2.0*”, uma internet voltada para as redes sociais utilizada e pela computação em nuvem, possibilitando novos modelos de negócios para o uso da web semântica. Um modelo de negócios criado nesta nova fase foi os aluguéis de plataformas de computação de *hardware* e *software* pela

internet que foi denominado de *Cloud Computing*. Tomando o *Cloud Computing* como uma tendência futura prevista da computação, prometendo muitos benefícios, como nenhuma despesa de capital, velocidade de implantação de aplicativos, menor tempo de lançamento no mercado, menor custo de operação e manutenção mais fácil para os clientes (KAHN; DENNIS, 2023; GUHA; AL-DABASS, 2010).

**Figura 7 - Mapas de cabos submarinos da conexão global de Internet**



Fonte: TELEGEOGRAPHY, 2023

### 2.3. Cloud Computing

*Cloud computing* ou computação em nuvem são recursos de computação acessados via internet e sob demanda. São aplicativos, servidores (físicos ou virtuais), armazenamento de dados, ferramentas de desenvolvimento, recursos de rede e muito mais hospedados em um data center remoto gerenciado por um *cloud services provider* (CSP), que é um provedor de serviços em nuvem. O CSP disponibiliza esses recursos por uma assinatura mensal ou por um valor cobrado conforme o uso (IBM, 2023).

Segundo o Google Cloud (2023), a computação em nuvem são recursos de computação como serviços na internet disponibilizados sob demanda. Eliminando a necessidade de empresas adquirirem, configurarem ou gerenciarem a infraestrutura, assim os

serviços serão pagos de acordo com o que for utilizado. Na computação em nuvem há três tipos de modelos de serviço: *Infrastructure-as-a-Service* (IaaS) oferece serviços de computação e armazenamento, a *Platform-as-a-Service* (PaaS) oferece um ambiente de desenvolvimento e implantação para criação de aplicativos de nuvem e o *Software-as-a-Service* (SaaS) entrega os aplicativos como serviços.

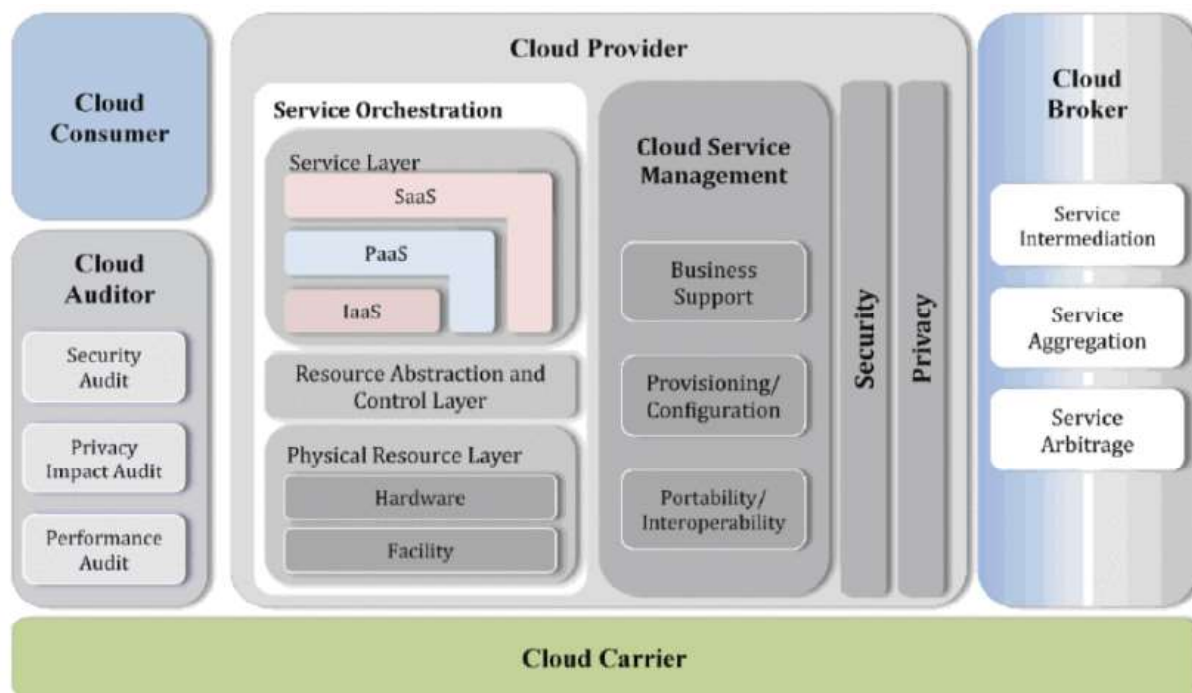
O *National Institute of Standards and Technology* (NIST), define *cloud computing* como “um modelo para permitir acesso de rede universal, adequado e sob demanda a um grupo de recursos de computação configuráveis que são compartilhados, tais como, redes, servidores, armazenamento, aplicativos e serviços podendo ser prontamente provisionados e liberados sem o menor esforço para gerenciar ou interagir com o provedor de serviços” (DONNO et al., 2019; MELL; GRANCE, 2011).

Conforme a RedHat (2023), a computação em nuvem é uma ação de cargas de trabalho em nuvens. As nuvens são consideradas ambientes locais da tecnologia da informação (TI) onde abstraem, agrupam e compartilham recursos escaláveis em uma rede separadamente, nuvens não são tecnologias em si. Anteriormente, nas nuvens públicas, privadas, híbridas e multi-clouds havia diferenças entre elas, pois eram definidas pelo local e propriedades delas.

No período da última década, foi observado uma relevante evolução dos paradigmas da computação. O *Cloud Computing* foi o mais popular e consolidado dentre os outros, sendo um paradigma desenvolvido a partir da necessidade de utilizar-se da “computação como utilidade”, permitindo facilmente o desenvolvimento de novos produtos e serviços de Internet (DONNO et al., 2019; ARMBRUST et al., 2010).

Na *cloud computing* tem-se uma arquitetura de referência que é representada na figura 8. Nela é fornecido uma visão geral de alto nível da nuvem, identificando os principais atores e seus papéis na computação em nuvem. Cada ator corresponde a uma entidade, isto significa que pode representar uma pessoa ou organização, que participa de uma transação/processo, de outro modo, executa algumas tarefas na computação em nuvem. Existem cinco entidades principais: *Cloud Provider*, *Cloud Consumer*, *Cloud Broker*, *Cloud Carrier*, *Cloud Auditor*. (DONNO et al., 2019; LIU et al., 2011).

**Figura 8** - Arquitetura de referência da *cloud computing* NIST



Fonte: DONNO et al., 2019; LIU ET AL., 2011

O *Cloud Provider* é uma entidade que prover um serviço às partes interessadas. O *Cloud Consumer* é uma entidade que consome um serviço e tem um contrato comercial com um ou mais provedores de nuvem. O *Cloud Broker* é uma entidade que atua como intermediário entre provedores de nuvem e consumidores de nuvem, fazendo o gerenciamento o uso, o desempenho e a entrega de serviços de nuvem. O *Cloud Carrier* é um representante que aprovisiona conectividade e entrega de serviços em nuvem dos provedores para consumidores da nuvem. E o *Cloud Auditor* é entidade que administra avaliações independentes da infraestrutura da Nuvem, incluindo serviços, operações de sistemas de informação, desempenho e segurança da implementação da nuvem (DONNO et al., 2019; LIU et al., 2011).

Segundo Donno et al., (2019) e Armbrust et al., (2010) na *cloud computing* há características essenciais, que estão resumidas abaixo:

➤ **Autoatendimento sob demanda:** Os recursos computacionais podem ser disponibilizados automaticamente de acordo com a demanda, sempre que necessário, sem a necessidade de qualquer intervenção humana entre o consumidor e o provedor de serviços;

➤ Amplo acesso à rede: os recursos de computação estão disponíveis na rede e são acessíveis por meio de vários métodos disponíveis para uma extensa variedade de plataformas de usuários como computadores, *notebooks* e dispositivos móveis;

➤ *Pool* de recursos: Recursos computacionais podem formar agrupamentos para atender múltiplos usuários, sendo alocados e liberados de forma dinâmica conforme a necessidade do usuário. Além disso, a localização dos recursos do fornecedor é irrelevante, pois o consumidor não tem controle e conhecimento da localização exata;

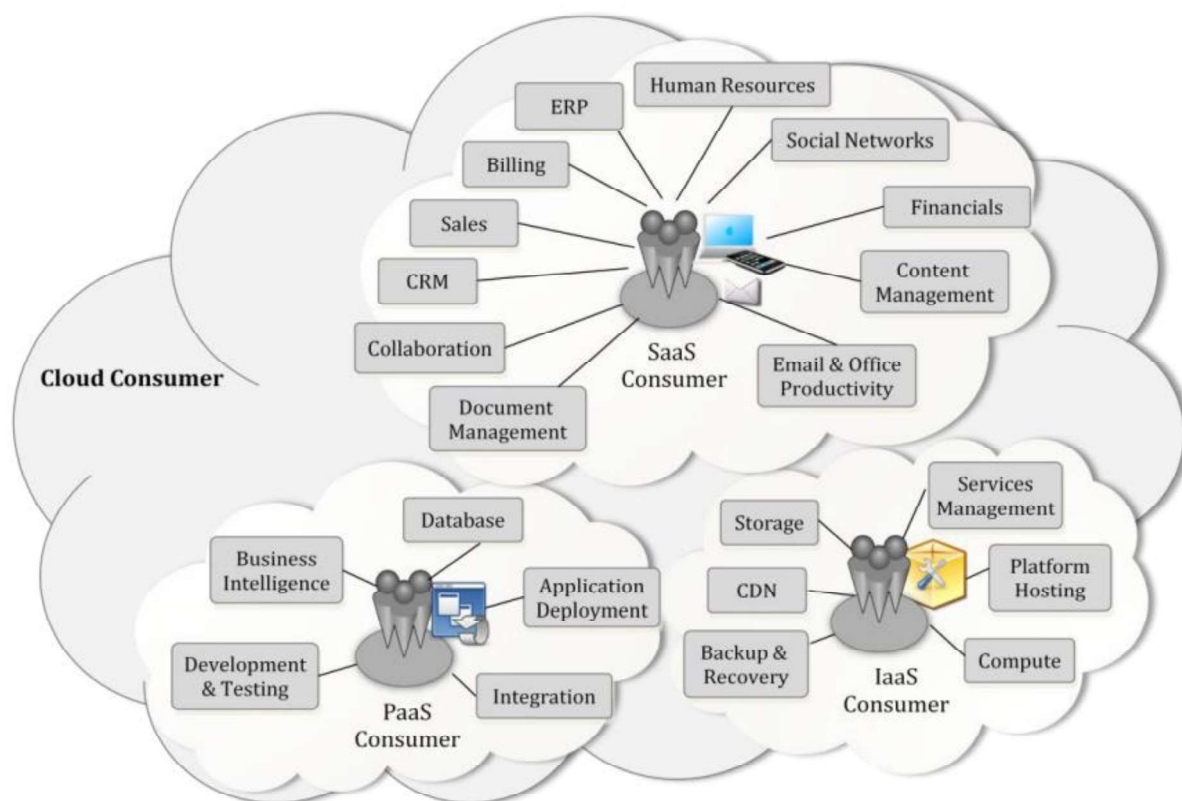
➤ Elasticidade rápida: Recursos computacionais podem ser disponibilizados com flexibilidade para escalar dinamicamente de acordo com a demanda, e liberados quando subutilizados. Assim, o consumidor tem a percepção de recursos computacionais ilimitados e sempre adequadas a sua necessidade;

➤ Serviço mensurado: a utilização de recursos é acompanhada e registrada com base no serviço oferecido. Isso é especialmente significativo em modelos de cobrança por uso, ou pagamento por usuário, garantindo transparência total entre o fornecedor e o cliente do serviço.

Uma infraestrutura em nuvem é uma coleção de *hardware* e *software* que potencializa as características fundamentais da computação em nuvem apresentadas acima.

Nos últimos anos, a demanda por processamento de dados tem-se expandido para diversas áreas, como as de ciências geográficas, engenharia, negócios, finanças, educação e aplicações de saúde. E a *cloud computing* tem sido bastante reconhecida como uma solução apropriada para o processamento de dados, pois é um paradigma de computação de alto desempenho que disponibiliza seus serviços através da internet e executa grandes aplicações científicas, partindo dos três principais tipos de serviço da computação em nuvem, a IaaS, fornece uma ampla plataforma de infraestrutura de hardware de computação e recursos de *software* no modelo de serviços para os usuários da nuvem. A PaaS disponibiliza uma plataforma onde os clientes podem implantar seus aplicativos e utilizar a plataforma existente para desenvolver seus aplicativos, enquanto na nuvem SaaS os usuários apenas executam os aplicativos na infraestrutura de nuvem pela internet (DUBEY et al., 2019; OWENS, 2010; NASR et al., 2018).

**Figura 9** - Exemplos de serviços disponíveis para um consumidor de nuvem



Fonte: LIU ET AL., 2011

## 2.4. Microservices

Martin Fowler e James Lewis, falam que Arquitetura de microsserviços é como uma interface de pequenos serviços, onde cada serviço executa o seu processo e se comunica por meio de pequenos mecanismos, e em várias oportunidades sendo uma API HTTP. O processo de deploy de cada serviço é totalmente automatizado e são realizados separadamente, pois cada serviço é construído de forma segregada com base no seu contexto de negócio. O gerenciamento de cada serviço é minimamente centralizado, podendo ser desenvolvidos com linguagens e tecnologias diferentes, até mesmo os serviços de armazenamento de dados podem ser descentralizados, e cada aplicação tendo o seu armazenamento (FOWLER; LEWIS, 2014).

Os microsserviços são um modelo de arquitetura de aplicações em que um conjunto de serviços, sendo eles segregados, funcionando de maneira independente, se comunicam por

meio de APIs. Na criação de *software*, cada função de uma aplicação pode ser independente, existindo de maneira segregada por contexto. Quando os elementos de uma aplicação são desenvolvidos dessa maneira, as equipes podem trabalhar juntas, não havendo interferência no trabalho da outra, alcançando mais produtividade, pois as equipes conseguiram trabalhar simultaneamente (REDHAT, 2023).

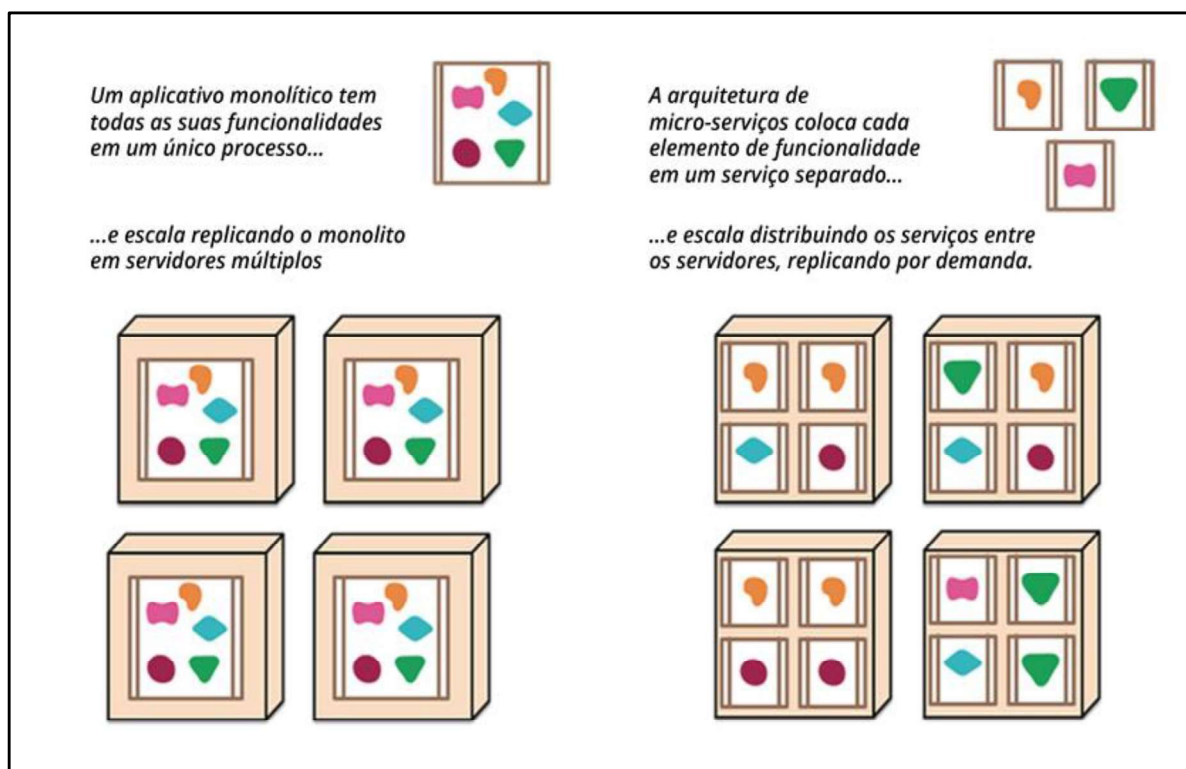
A arquitetura de microsserviços é uma abordagem tradicional no desenvolvimento de *software* que transforma uma aplicação em uma coleção de pequenos serviços, cada um executando seus próprios processos e se comunicando através de mecanismos leves, muitas vezes por meio de APIs com recursos HTTP. Esses serviços são construídos em torno de capacidades específicas de negócios, operando com deploy independente e automático. Há um mínimo de gerenciamento centralizado, permitindo que sejam escritos em diversas linguagens de programação e façam uso de diferentes tecnologias de armazenamento de dados (FOWLER; LEWIS, 2014).

Ao comparar com a abordagem monolítica, em que a aplicação é construída como um único serviço, totalmente centralizado, percebemos que os aplicativos corporativos tradicionais são compostos por uma interface de usuário do lado do cliente, um banco de dados e um aplicativo do lado do servidor. Este último lida com solicitações HTTP, executa lógica de domínio, acessa o banco de dados e gera as visualizações HTML. Entretanto, a estrutura monolítica acarreta ciclos de alterações interligadas, demandando a reconstrução e o redeploy de toda a aplicação (FOWLER; LEWIS, 2014).

Por outro lado, as arquiteturas de microsserviços propõem uma abordagem diferente, onde as aplicações são construídas de forma que cada função principal opera de maneira independente. Essa abordagem permite que as equipes de desenvolvimento criem e atualizem componentes de forma isolada, atendendo às mudanças nas necessidades empresariais sem a necessidade de interrupções globais na aplicação. Essa flexibilidade e independência na execução de funções específicas são características fundamentais das arquiteturas de microsserviços, contribuindo para a agilidade e adaptabilidade das aplicações diante das demandas do ambiente empresarial em constante evolução (REDHAT, 2023).



**Figura 10 - Monólitos e Microserviços**



Fonte: FOWLER; LEWIS, 2014

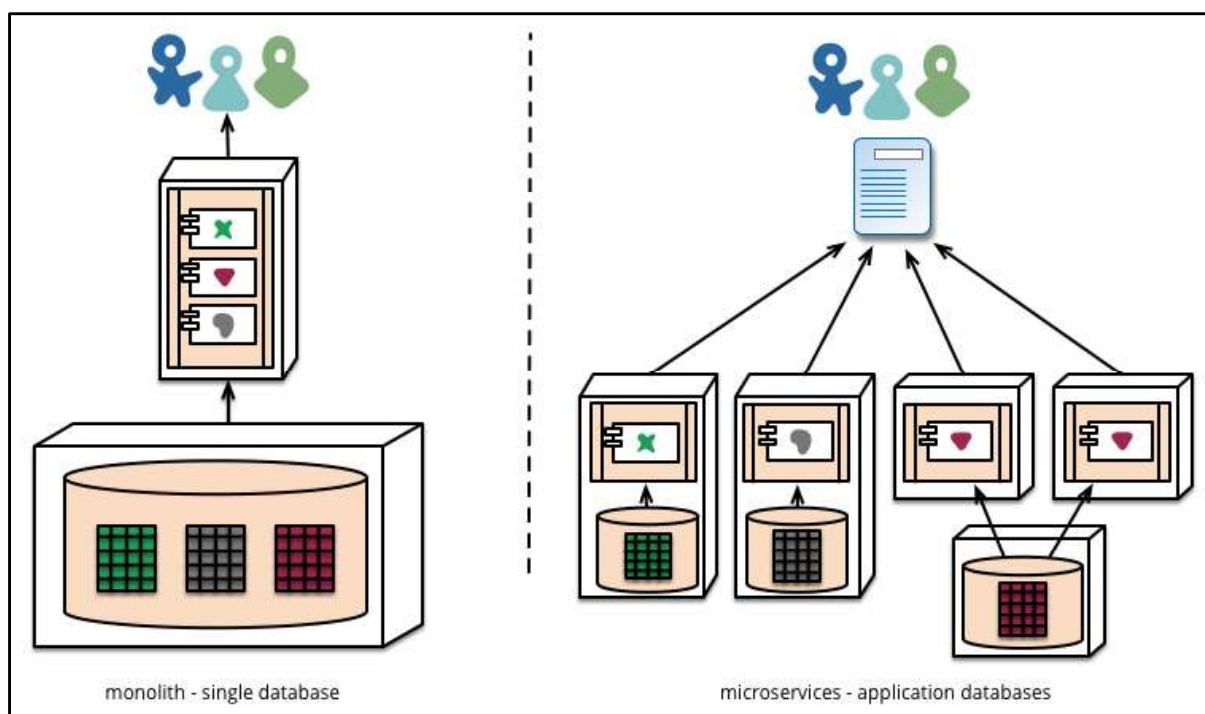
Diante das demandas e da constante evolução com adaptabilidade, os desafios associados aos aplicativos monolíticos, como dificuldade em manter uma estrutura modular coesa, escalabilidade limitada e complexidade nos deploys, motivaram a evolução para a arquitetura de microserviços. Essa abordagem permite deploys independentes, escalabilidade modular, a possibilidade de utilizar diferentes linguagens de programação para serviços distintos e a gestão independente por equipes especializadas (FOWLER; LEWIS, 2014).

Embora os princípios da arquitetura de microserviços tenham raízes antigas, remontando aos princípios de design do Unix, sua aplicação ainda é subutilizada. A proposta de repensar a arquitetura de *software* oferece vantagens como independência de deploy, escalabilidade eficiente e melhor gestão modular, conceitos que muitos desenvolvedores poderiam se beneficiar ao explorar (FOWLER; LEWIS, 2014).

A arquitetura de *microservices* ganhou popularidade, principalmente por sua capacidade de oferecer soluções de *software* escaláveis, resilientes, flexíveis e confiáveis. Consequentemente, diversas corporações que operam *software* extensos e complexos estão buscando por recursos automatizados para transformar suas aplicações monolíticas em microserviços (ABGAZ et al., 2023).

Ao longo do tempo, *softwares* de sucesso tendem a crescer em complexidade devido à implementação de diversas funcionalidades, resultando em componentes altamente acoplados, mas com menos coesão. Nesse contexto, as arquiteturas monolíticas centralizam as funcionalidades em extensos componentes individuais, apresentando limitações inerentes em termos de escalabilidade, manutenção e desempenho de implementação. Por outro lado, as arquiteturas de microsserviços adotam modelos distribuídas, favorecendo a segregação do sistemas em diversos componentes independentes e granulares podendo ser invocados de acordo com a demanda proporcionando benefícios notáveis, como maior escalabilidade e frequência aprimorada de implantação (ABGAZ et al., 2023; DRAGONI et al., 2017; KALSKE et al., 2018; MENDONCA et al., 2021; BALALAIE et al., 2018; AHMADVAND; IBRAHIM, 2016; AUER et al., 2021; CLARKE et al., 2016; ZIMMERMANN, 2016; NEWMAN, 2022; WOLFART et al., 2021; SMITH, 2018).

**Figura 11 - Descentralização de Dados**



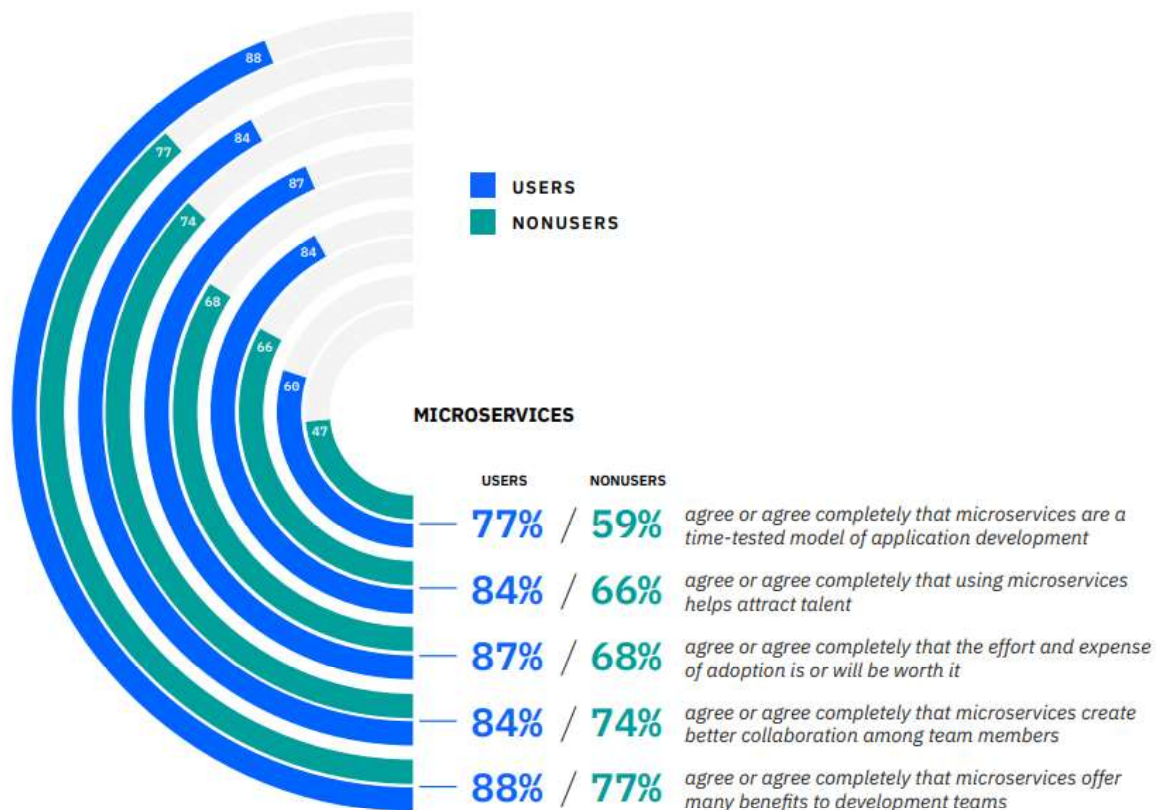
Fonte: FOWLER; LEWIS, 2014

Anteriormente as arquiteturas monolíticas eram geralmente adotadas ao invés da arquitetura de microsserviços. Contudo, ao passo que os *softwares* continuaram a crescer e a demanda por ciclos de implantação ainda mais rápidos, surgiu a necessidade de gerar pacotes das aplicações em pequenos serviços para serem compilados separadamente. Com o advento de inovações baseadas em *cloud*, como SaaS e FaaS, os benefícios das arquiteturas

monolíticas tornaram-se menores, resultando crescente interesse em arquiteturas de microsserviços. Para empresas que contam com um portfólio de aplicações baseados em monólitos, o desafio é segregar (ABGAZ et al., 2023; SCHÜTZ et al., 2013; GROGAN et al., 2020; VILLAMIZAR et al., 2017, LOUKIDES; SWOYER, 2020; IBM, 2021).

Para corporações que contam com aplicações baseadas em monólitos, o desafio está na decomposição dos *softwares* com o foco em apoiar os engenheiros de migração na elegibilidade das aplicações a tornarem-se microsserviços, seguindo para implementações coesas fundamentadas em microsserviços, envolvendo técnicas como análise do domínio da aplicação, análise do código-fonte, rastreamentos de execução e informações relacionadas à versão (ABGAZ et al., 2023; DAOUD et al., 2021; LI et al., 2019; CHEN et al., 2017; SAIDANI et al., 2019; FURDA et al., 2018; SELMADJI et al., 2018; GYSEL et al., 2016; JIN et al., 2021; ESKI; BUZLUCA, 2018).

**Figura 12** - Pesquisa IBM Market Development & Insights sobre os desafios e benefícios da adoção de microsserviços

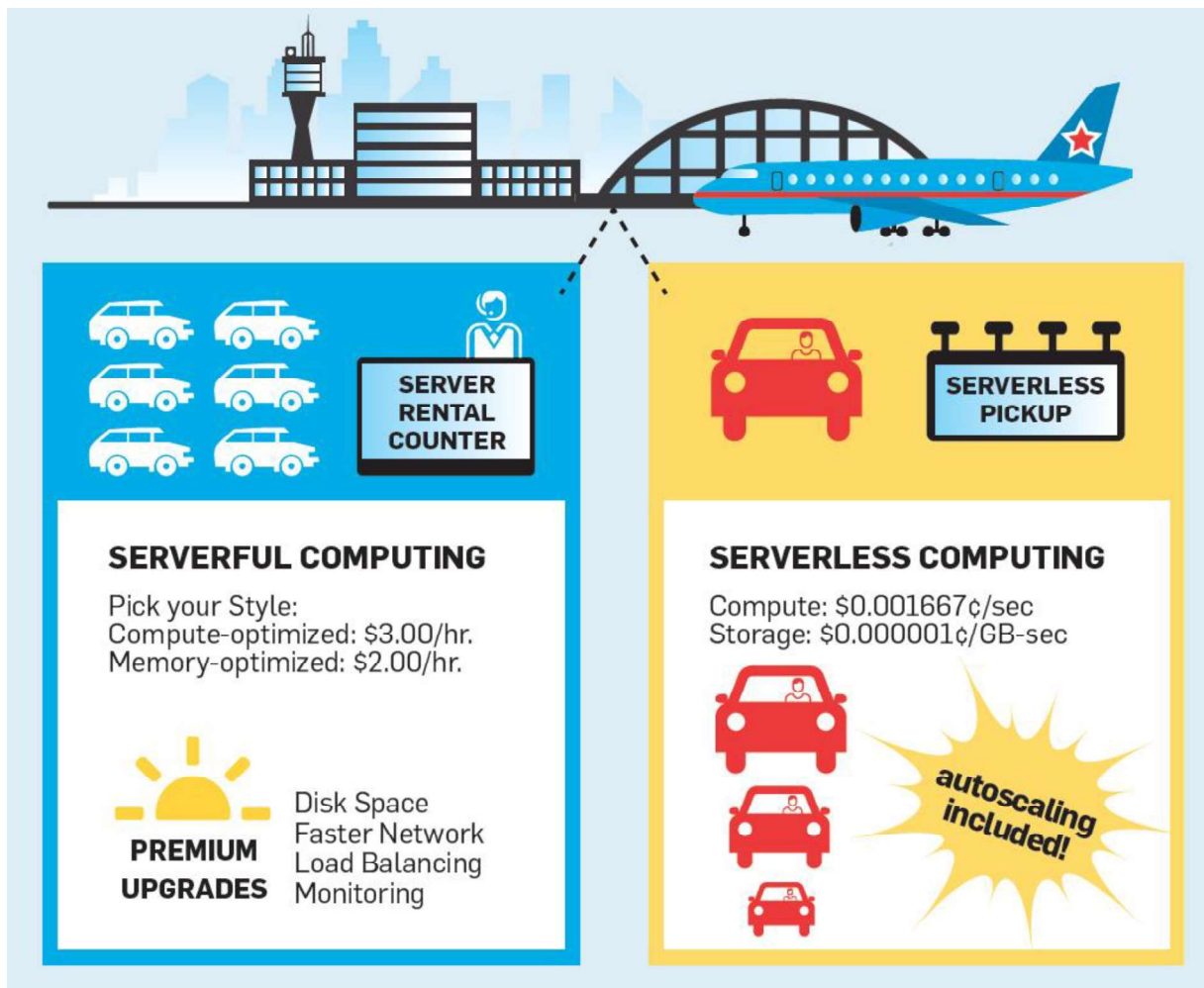


Fonte: IBM, 2021

## 2.5. Serverless

A computação *serverless* é um paradigma de *cloud computing* que concede aos desenvolvedores dediquem-se unicamente na lógica de negócios, enquanto os provedores de serviços em nuvem gerenciam tarefas de gerenciamento de recursos. Aplicações *serverless* baseadas nesse modelo de programação são frequentemente desenvolvidas por várias funções pequenas e granulares do tipo FaaS que implementam processos de negócios complexos por meio de interações mútuas e integração com *Backend-as-a-Service* (BaaS), como bancos de dados (STEINBACH et al., 2022, TAIBI et al., 2021).

**Figura 13** - Abordagens de cloud computing comparadas a viagens do aeroporto: Serverful como aluguel de carro e Serverless como pegar um táxi



Fonte: SCHLEIER-SMITH et al., 2021

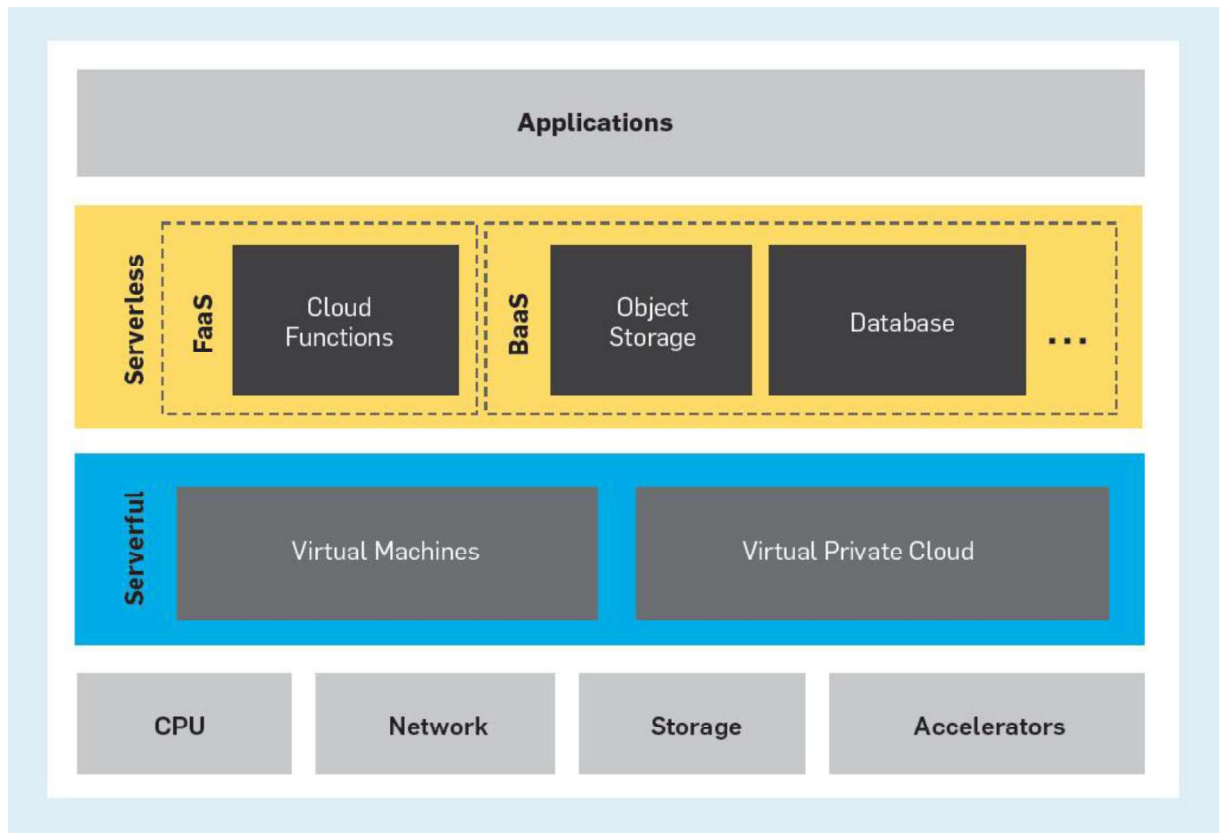
O *serverless* é um padrão de desenvolvimento *back-end* em nuvem para criação e execução de aplicações sem servidor. As atividades de manutenção e serviços de rotina ficam na responsabilidade dos provedores da plataforma em nuvem, sendo assim, gerenciando a capacidade, patching e disponibilidade dos servidores. Com este tipo de arquitetura, as empresas e desenvolvedores criam seus projetos sem ter a preocupação com a escalabilidade, disponibilidade, segurança, gerenciamento, ou com infraestrutura (REDHAT, 2022; VEUVOLU et al., 2023).

Após a implantação dos serviços, as aplicações *serverless* atendem as demandas de acordo com as necessidades, gerenciando a escalabilidade, aumentando ou diminuindo automaticamente. Nos provedores de nuvem pública, as soluções *serverless* tendem ser oferecidas sob demanda por meio de um padrão de execução orientado a eventos, com isso, por parte das funções *serverless* não utilizadas, não haverá cobranças (REDHAT, 2023).

Regularmente, as FaaS e o *serverless* são reconhecidas como sinônimos uma da outra, mas na realidade existem duas definições específicas. Embora, o FaaS seja uma aplicação *serverless*, ele é referente a várias categorias em que o servidor é completamente abstraído, e um subconjunto da computação *serverless* baseado em gatilhos que são orientados a eventos, onde a aplicação é executada em resposta a eventos ou solicitações. Se não houver solicitações orientadas a eventos, o servidor será encerrado, disponibilizando seus recursos para outras solicitações. Após a implantação, o FaaS atende à demanda automaticamente, aumentando e diminuindo conforme necessário. Normalmente, quando uma função *serverless* está inoperante, ela não irá gerar custos, economizando verbas em muitas situações (KING, 2022).

As arquiteturas *serverless* trouxeram vários benefícios para empresas e desenvolvedores que trabalham em soluções de *software* baseadas em *cloud*. Entre estes benefícios estão *pipelines* de desenvolvimento mais simplificadas com bases de código abstraídas de complexidades arquitetônicas. Consequentemente, as plataformas *serverless* são automaticamente escaláveis de acordo com a demanda, resultando em redução de custos para todas as partes envolvidas e menor sobrecarga para os desenvolvedores. Tais benefícios impulsionam as mudanças vistas atualmente, das migrações de arquiteturas tradicionais para “*microservices*” ou soluções baseadas em *serverless* (CHRISTIDIS et al., 2020; FOX et al., 2017).

**Figura 14** - Serverless vs. Serverful: o Serverless oferece uma abstração entre as aplicações e os servidores subjacentes

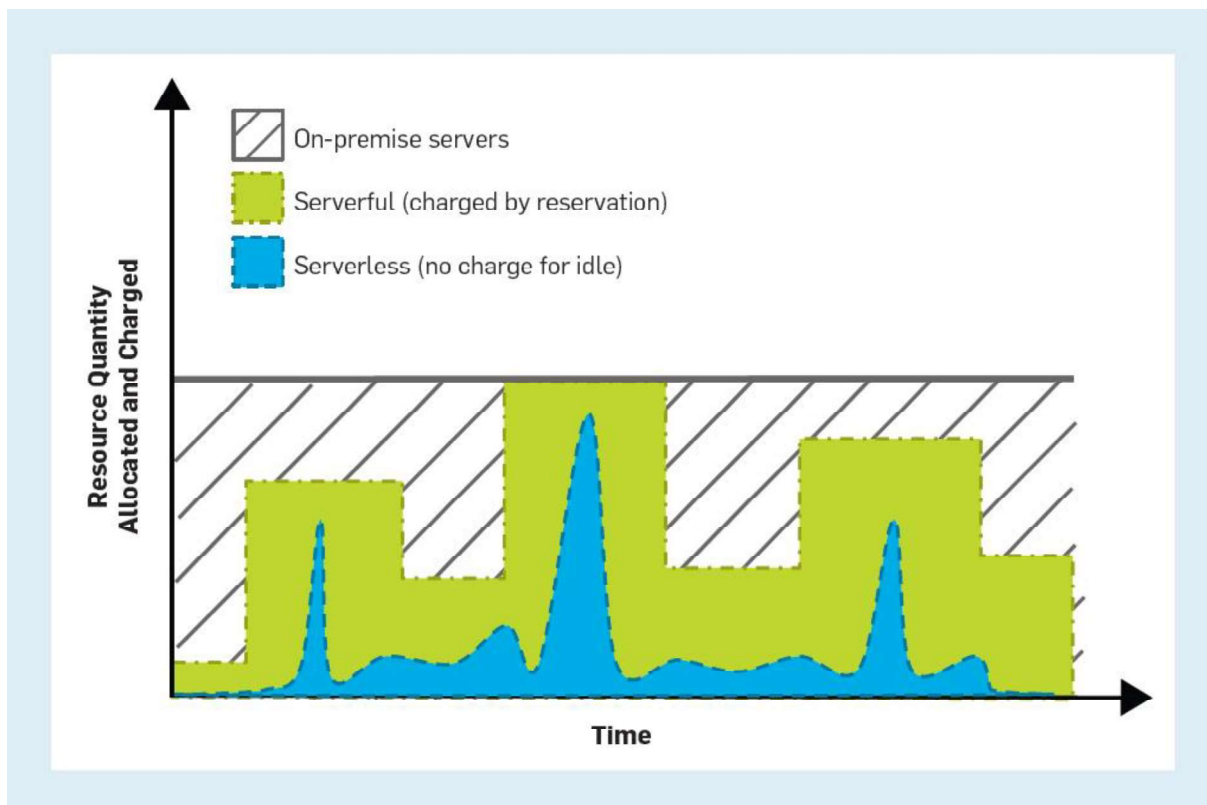


Fonte: SCHLEIER-SMITH et al., 2021

As arquiteturas *serverless* simplificam as complexidades existentes em desenvolvimentos baseados em arquiteturas padrões, como o *client-server*, frequentemente associadas a cargas de trabalho de alto volume em ambientes de produção. Essas arquiteturas priorizam a resolução eficiente dos problemas evidenciados e o desenvolvimento de aplicações de alta qualidade, evitando focar em restrições arquiteturais. O código é encapsulado em pacotes de implantação e executado em um contêiner de computação e sem estado, que são lançados dinamicamente e acionados por eventos. Após as invocações, todos os recursos e dependências são destruídos, fortalecendo o conceito de uma arquitetura sem estado e consequente modelo de faturamento eficiente. Contudo, a crescente complexidade de projetos de desenvolvimento, especialmente, em de trabalho de Inteligência Artificial (IA), pode tornar aplicação incompatível com a abordagem *serverless* (CHRISTIDIS et al., 2020).



**Figura 15** - Serverless vs. Serverful: na computação **Serverless**, os usuários pagam apenas pelos recursos consumidos, não pela capacidade ociosa reservada



Fonte: SCHLEIER-SMITH et al., 2021

Pensando pelo lado de agilidade e recursos, considerar uma arquitetura *serverless* traz diversos benefícios para um projeto de desenvolvimento, pois a equipe não terá que se preocupar com a disponibilidade e provisão de recursos para o serviço. Porém os serviços implementados em um padrão *serverless* haverá limitações no que diz respeito a aplicações que necessitam de grandes processamentos, ou que guardem estados em memória de exceção, pois irão demandar tempo de processamento e espaço em disco, com isso aumentando o provisionamento de recursos de máquina (SEWAK; SINGH, 2018; KOSCHEL et al., 2021).

Com objetivo de garantir os benefícios prometidos, as plataformas *serverless*, como a AWS com a AWS Lambda, cobram algumas restrições aos desenvolvedores quanto ao que e como pode ser implementado. Os Limites são estabelecidos no tamanho físico do pacote de implantação do código-fonte, de até 250MB para o AWS Lambda, na quantidade máxima de RAM alocada, bem como o limite de tempo do ciclo de vida antes que a instância em execução seja interrompida abruptamente. Essas condições definem o ambiente *serverless* do AWS Lambda como uma plataforma com recursos limitados, geralmente destinada a realizar tarefas automatizadas de baixo nível, como transferência programada de dados de um banco

de dados para outro, ou realizar algum processamento simples quando um novo item entra em um meio de armazenamento (FOX et al., 2017; BALDINI et al., 2017; AMAZON WEB SERVICES (AWS), 2023; JANGDA et al., 2019).

Diante disso, a popularidade do *serverless* no modelo FaaS tem expandido e está recebendo uma grande atenção dos desenvolvedores, principalmente após o lançamento do AWS Lambda pela Amazon em novembro de 2014. Recentemente, descobriram que as perguntas sobre *Serverless* no *Stack Overflow*, uma plataforma de perguntas e respostas sobre programação, cresceram 380% de 2015 a 2020. Acredita-se que o mercado FaaS cresça de 3,33 bilhões de dólares em 2018 para 31,53 bilhões de dólares em 2026. No entanto, o *Machine Learning* (ML) tem sido amplamente utilizado na computação em nuvem, essencialmente quando é segmentado em pequenas etapas de *pipeline* de *Machine Learning as a Services* (MLaaS). Considera-se necessário utilizar o *serverless* devido ao alto custo de gerenciamento de recursos em nuvem (BARRAK et al., 2022; WEN et al., 2021; RAJPUT, 2021).

Deste modo, o *serverless* é uma possibilidade considerável para a resolução de pequenos serviços, principalmente quando as empresas não sabem estimar com precisão o tráfego de suas aplicações, da escalabilidade e dos custos. Além disso, existem diversos estudos estão explorando o *serverless* para realizar tarefas de ML, como *training*, *hyperparameter Optimization* e *model deployment* (BARRAK et al., 2022; RIBEIRO et al., 2015; CASTRO et al., 2019; KANAGARAJ; GEETHA, 2021; KAPLUNOVICH; YESHA, 2020; CHAHAL et al., 2021).

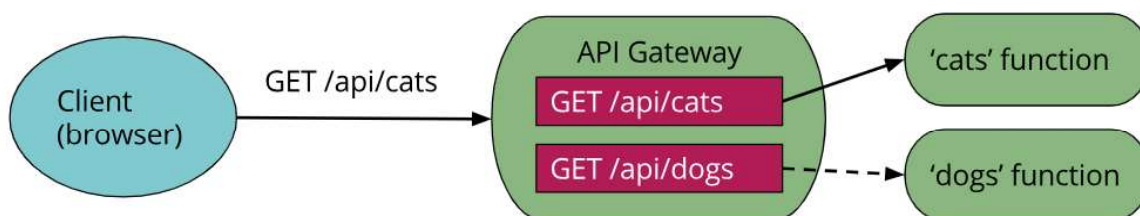
## 2.6. Function as a Services

Desde a criação do *AWS Lambda* em 2014, solução da AWS que adota o modelo FaaS, a computação *serverless* ganhou um grande destaque em diversas áreas, como as de *machine learning*, álgebra linear, tarefas *map/reduce*, dentre outras. Outros principais provedores de nuvem como Google e Microsoft adotaram esta tecnologia incluindo no seu catálogo de serviços como *Google Cloud Functions* e o *Microsoft Azure Functions*. A FaaS, permite que uma aplicação tradicional seja segregada em pequenas funções, sem estado e transitória, executadas individualmente em contêineres com um tempo de execução em uma plataforma FaaS (STEINBACH et al., 2022; CHADHA et al., 2020; CARREIRA et al., 2019; SHANKAR et al., 2020; CHARD et al., 2019; JONAS et al., 2017; CASTRO et al., 2019).



A AWS permite executar um código sem provisionar e gerenciar servidores por meio da *Lambda Function*. O AWS Lambda sendo uma tecnologia *serverless*, executa o código em uma infraestrutura de computação de alta disponibilidade, realizando a administração dos recursos computacionais, inclusive a manutenção do servidor e do sistema operacional, o provisionamento e a escalabilidade automática da capacidade, o monitoramento e o registro de logs do código. O Lambda, permite executar o código para praticamente qualquer tipo de aplicação ou serviço de *back-end*, tornando-se possível a partir da implementação do código em uma das linguagens compatíveis com o Lambda (AMAZON WEB SERVICES (AWS), 2023).

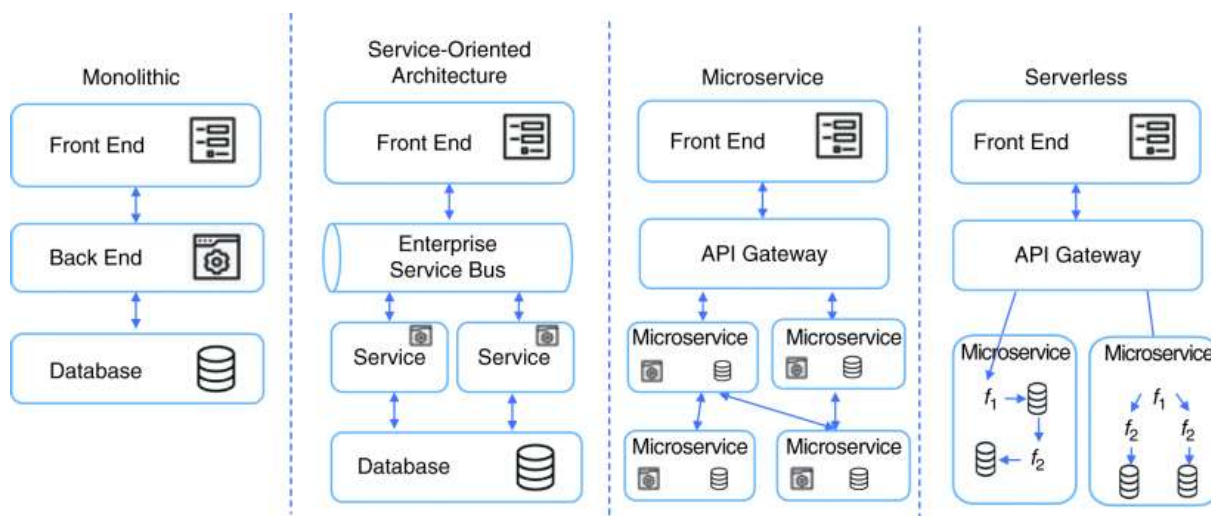
**Figura 16** - Acionamento de Lambda Function por API Gateway



Fonte: ROBERTS, 2018

O fornecimento dos recursos, tais como os contêineres, e o dimensionamento automático das funções são realizados de acordo com a demanda aos serviços, e são de responsabilidade da plataforma FaaS. Baseada em eventos, as funções são acionadas por várias fontes que emitem os eventos, estes são bancos de dados ou filas de mensagens, bem como outras FaaS da aplicação que emitem eventos para atualizações de banco de dados ou fila, além de solicitações HTTP (STEINBACH et al., 2022; CASTRO et al., 2019).

Por possuírem características efêmeras e independentes, as funções trazem benefícios como escalabilidade automática e redução de custos, pois a política *pay-per-use*, de pagar pelo que foi utilizado, se aplica neste contexto. Além disso, a abordagem imperativa na organização do código, proporcionada pela composição de funções, representa uma evolução significativa na forma como as aplicações são desenvolvidas, escaladas e mantidas na era da computação em nuvem, desta forma executar uma aplicação em uma plataforma FaaS pode, portanto, contribuir para redução de custos de uma companhia (STEINBACH et al., 2022; TAIBI et al., 2021).

**Figura 17** - A evolução das arquiteturas de software

Fonte: TAIBI et al., 2021

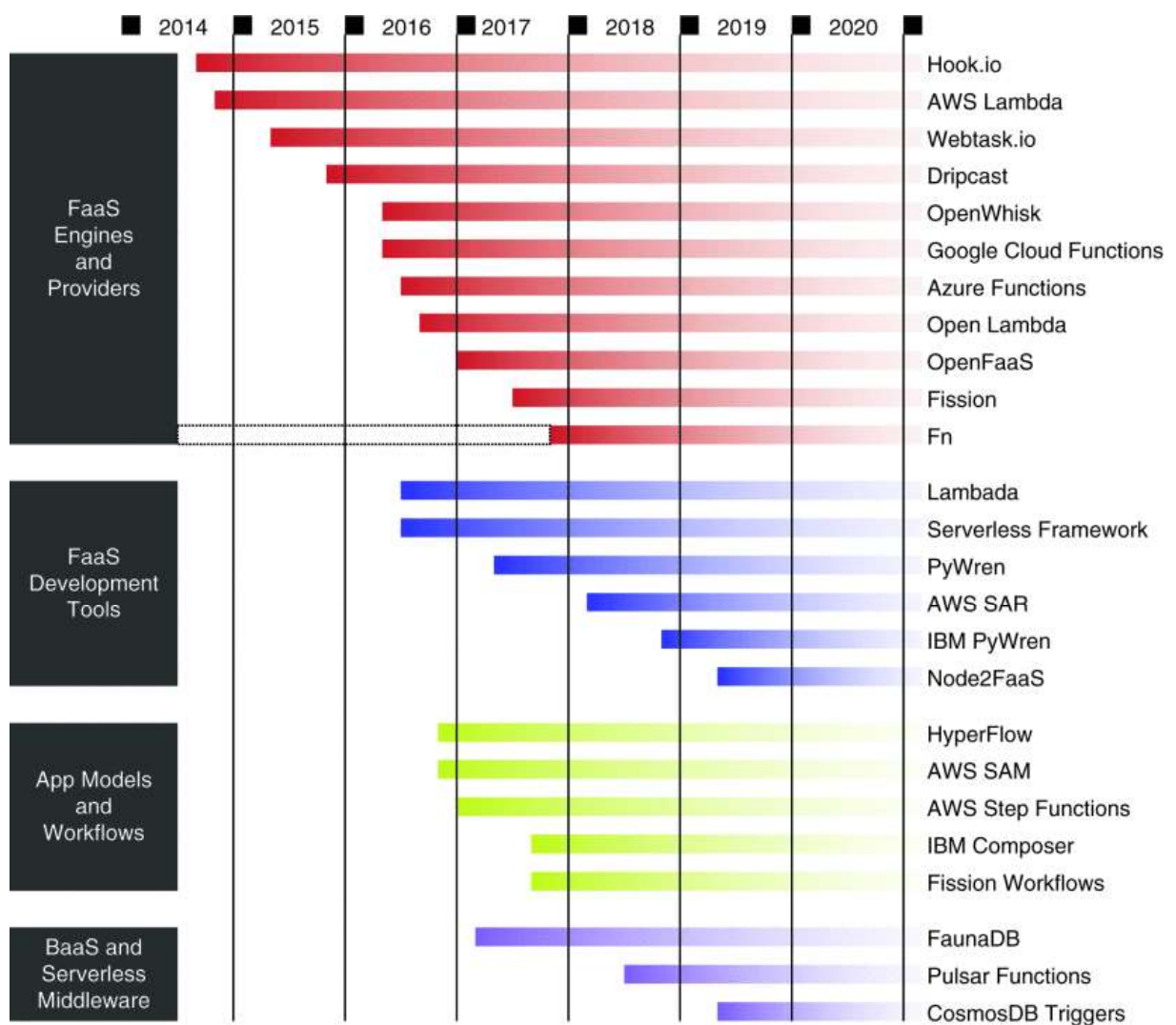
Os desafios do modelo tradicional de gestão de infraestrutura é o alto investimento em instalação e configuração, para que um ambiente possa receber uma aplicação e executá-la adequadamente. A *cloud computing* como solução trouxe o modelo PaaS, entregando um ambiente todo configurado em uma plataforma apta para receber a aplicação. Porém, nesse tipo de serviço eventualmente há necessidade de o desenvolvedor atuar em algumas configurações da infraestrutura para garantir, entre outros aspectos, a elasticidade do ambiente. Além disso, o ambiente precisa ser provisionado durante a implementação da aplicação, aumentando o custo final do projeto. Partindo deste cenário, os provedores de nuvem desenvolveram o FaaS, serviço que cobra apenas pelo processamento efetivo, garantindo uma redução dos custos (CARVALHO; ARAÚJO, 2023; MELL, GRANCE, 2011; CRISTIAN SPOIALA, 2019).

Seguindo o paradigma de *microservices*, o modelo FaaS recentemente tem despertado muito interesse desde sua criação. Independentemente de o FaaS ser utilizado para executar tarefas segregadas, é na composição das aplicações que este modelo de serviço pode proporcionar melhorias significativas de desempenho (CARVALHO; ARAÚJO, 2023; ZIMMERMANN, 2016).

A utilização do modelo FaaS em aplicações monolíticas é um desafio para desenvolvedores, exigindo que se habituem com interfaces específicas de cada provedor em nuvem e uma reestruturação das aplicações. Embora essa transição possa ser desestimulante, pode privar os desenvolvedores dos benefícios intrínsecos de uma arquitetura de

microserviços baseada em *serverless*, como alta disponibilidade, resiliência, redução de custos, entre outros, são significativos. Neste cenário, o *JavaScript* relevante em aplicações web e, especificamente, o *Node.js* surgem como facilitadores, dada sua relevância no desenvolvimento web para execução de aplicações do lado do servidor. A popularidade do *JavaScript* proporciona uma curva de aprendizado menor, aproveitando-se do conhecimento existente, mas reitera a importância de evoluções contínuas dos modelos de desenvolvimento e dos paradigmas, como os de cloud computing para que a progressão dos processos de melhoria se mantenha em um fluxo constante (CARVALHO; ARAÚJO, 2023; FLANAGAN, 2013; FRANKSTON, 2020; SHAH; SOOMRO, 2017).

**Figura 18** - O crescimento do ecossistema de software sem servidor de 2014 a 2020



Fonte: TAIBI et al., 2021

Como um dos benefícios do FaaS é a redução de custos, nem sempre as instâncias dos serviços são mantidas ativas com capacidade de processar uma requisição, sendo comum a

primeira solicitação ou após algum tempo sem ativações, o serviço FaaS apresente um tempo de resposta maior, esse feito é conhecido como “*cold start*”. Outro recurso que pode impactar o desempenho do FaaS é conhecido como “*Spawn Effect*”. Este efeito refere-se à limitação de simultaneidade, sendo mantido apenas um certo volume de solicitações simultâneas, enquanto as requisições posteriores precisam aguardar a liberação de um slot de processamento, resultando na redução da eficiência de execução e ao surgimento de timeouts, ocasionando falhas no sistema (CARVALHO; ARAÚJO, 2023; SCHLEIER-SMITH et al., 2021; RISTOV et al., 2022)

Apesar do desafio do “*cold start*” em ambientes *serverless*, esta modelo ainda se destaca em relação a outras tecnologias em nuvem, como *Virtual Machine* (VM). Diversas estratégias têm sido desenvolvidas para enfrentar esse problema, incluindo o aquecimento periódico de instâncias e o uso de arquiteturas específicas, como a “*switchboard*”, composta por um pacote de *microservices* com seis *lambdas function*, que busca ativar funções subsequentes após a inicialização da primeira. Outras soluções envolvem manter instâncias aquecidas por determinados períodos, e outras demonstrando viabilidade econômica ao otimizar recursos mantendo as instâncias ativas. Complementando, foi proposto a utilização do método *Long-Short Term Histogram* (LSTH) que monitora os períodos de inatividade da aplicação, gerando dois histogramas distintos para representar os padrões de solicitação em intervalos curtos e longos. Esse monitoramento permite a seleção de momentos estratégicos para o aquecimento prévio da função, assegurando a continuidade da instância e minimizando o impacto das partidas a frio, resultando em uma gestão mais eficiente dos recursos (BARRAK et al., 2022; WANG et al., 2018; YU et al., 2021; ZHANG et al., 2020; FOTOUHI et al., 2019; MVONDO et al., 2021; ZHANG et al., 2019; YANG et al., 2022).

O objetivo na adoção da tecnologia *serverless* é essencialmente à redução de custos, diante dos desafios com os altos gastos associados à infraestrutura, como a alocação de máquinas virtuais sem uso total de recursos. Estudos evidenciam que substituir clusters dedicados de IaaS por uma arquitetura *serverless*, através de soluções como *SIREN Machine Learning*, pode resultar em economias nos custos de treinamento em comparação com a arquitetura do *Apache MXNet*. Alcançando até 98% de redução de custos sem comprometer o desempenho, o *AMPS-Inf Autonomous Framework* mostrou-se muito eficiente. Além disso, iniciativas destacam a importância do balanceamento de carga na carga de trabalho de inferência de *Machine Learning* como uma estratégia adicional para otimizar custos, reforçando que a arquitetura *serverless* pode ser uma opção eficaz para minimizar despesas

em projetos ligados à arquitetura de design, computação, implantação de inferência e consultas de leitura/escrita (BARRAK et al., 2022; WANG et al., 2019; JARACHANTHAN et al., 2021; CHAHAL et al., 2020).

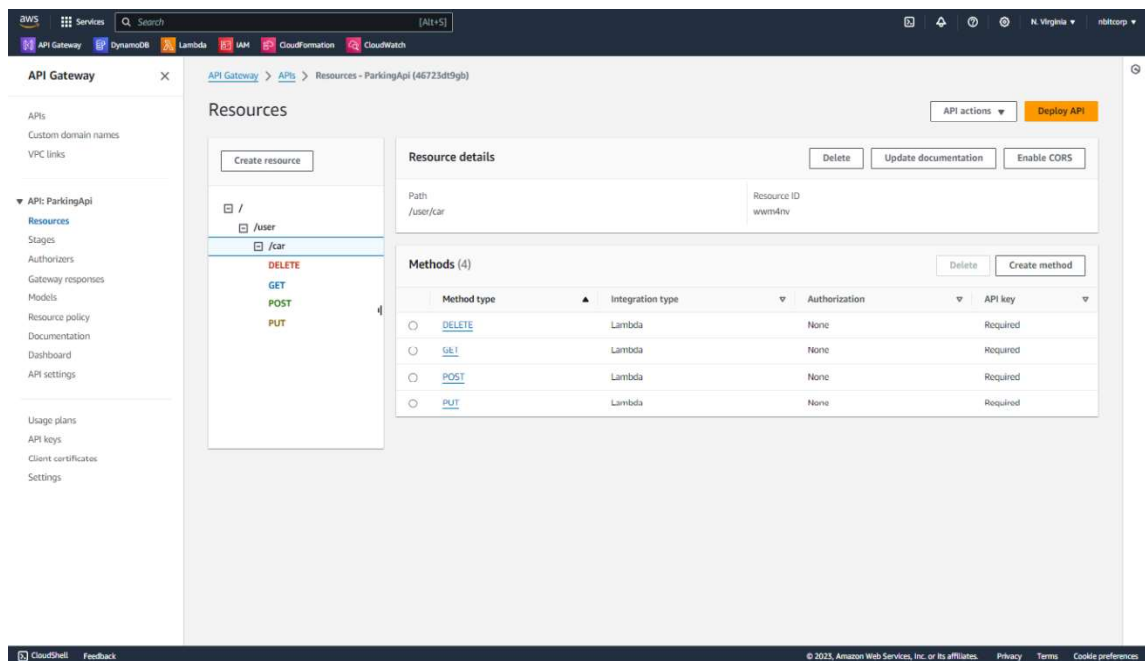
### 3. DESENVOLVIMENTO

Nesta seção será apresentado a no trabalho e porquê da escolha, aprofundando a arquitetura *serverless* com o uso de *AWS Lambda Function*.

#### 3.1. Síntese da proposta

É abordado nesta pesquisa sobre a modernização de *softwares* por meio da implementação de uma solução baseada em serviços *serverless* na *Amazon Web Services* (AWS). As tecnologias principais incluem *AWS Lambda*, *Amazon API Gateway*, *AWS DynamoDB* e *AWS CloudWatch*, com o intuito de garantir uma alta coesão, escalabilidade, confiabilidade, manutenibilidade e segurança para os serviços.

**Figura 19** - API Gateway – Endpoints e Rotas da API RESTful



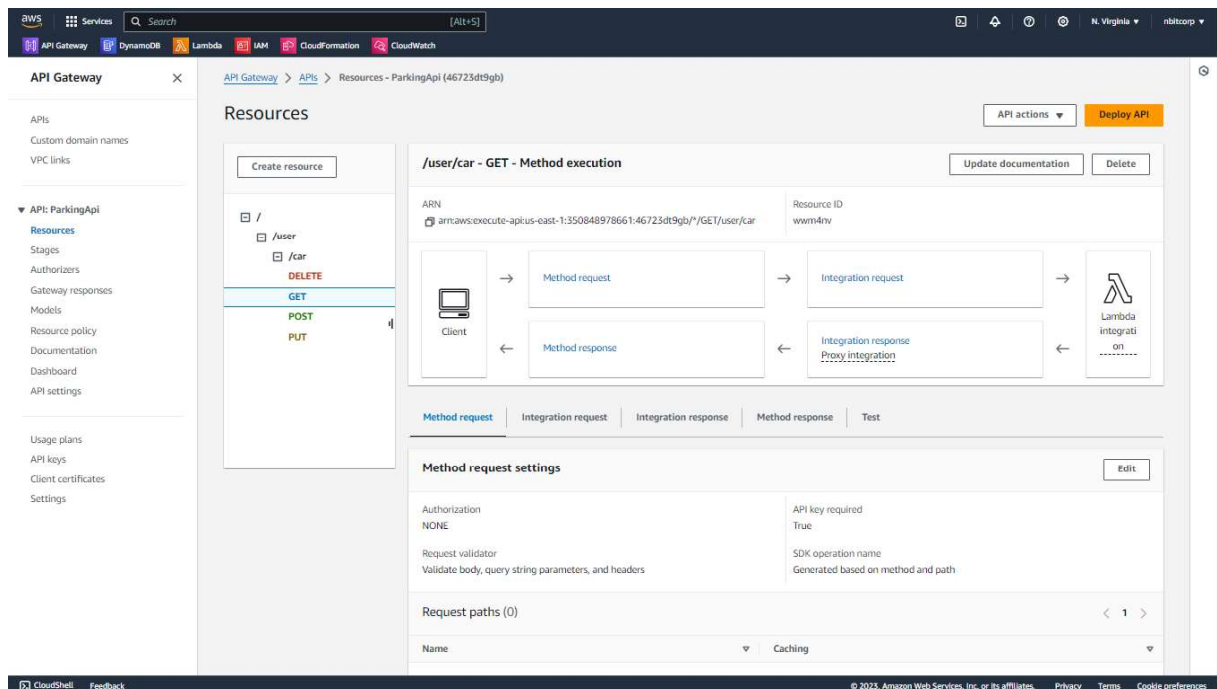
Fonte: Elaborado pelo autor

A crescente demanda por eficiência e escalabilidade em sistemas de modo geral motivou a escolha de tecnologias *serverless* na AWS. A abordagem *serverless* oferece

benefícios significativos, como redução de custos, escalabilidade automática e uma arquitetura mais flexível (TAIBI et al., 2021).

A escolha da plataforma e tecnologia foi motivada pela diversidade de conteúdo e documentações existentes, e isso se dá pela AWS ser uma das plataformas de nuvem pública mais utilizadas e versáteis da atualidade, oferecendo diversos serviços, e tendo em seu portfólio diversos cases de sucessos ao redor do mundo, com grandes corporações no catálogo de clientes (TAIBI et al., 2021; PELLE et al., 2021).

**Figura 20 - Configurações de solicitação de método GET**



Fonte: Elaborado pelo autor

### 3.2. Arquitetura do Sistema

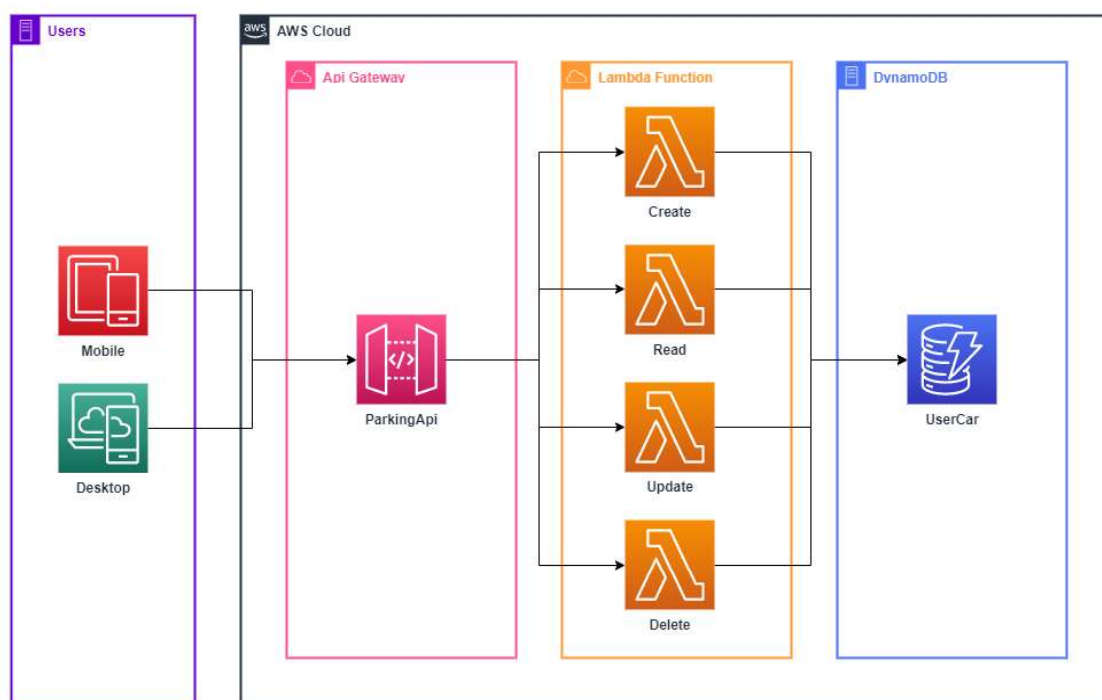
Como visto nos capítulos anteriores, a adoção da arquitetura *serverless* no contexto de aplicações que utilizam *lambda function* é uma alternativa ideal para *workloads* mais curtos, sendo comumente aplicadas em pequenas soluções que contam com uma alta coesão, com o menor risco de falhas e que não necessitam guardar estados. Desta forma, terá um *software* mais performático, autogerenciável, auto escalável e com alta disponibilidade.

A arquitetura do sistema é composta por quatro *lambda functions*, cada uma dedicada a uma operação CRUD específica. A tabela principal, UserCar, construída no *DynamoDB*, é

projetada com a chave de partição sendo o CPF e a chave de classificação sendo a placa do veículo (*License Plate*). Os detalhes do veículo e do usuário são armazenados em um mapa dentro da tabela, contendo as demais informações como modelo, cor, marca, ano, e o e-mail e telefone do cliente para contato.

O projeto foi desenhado pensando no contexto simplificado de um sistema de estacionamento, desenvolvendo um MVP com fornecendo as operações de CRUD buscando custos de operacionais mais atrativos, mas que fosse possível ter controle dos carros dos clientes, garantindo assim a operação do negócio de forma eficiente, gerenciável e de baixo custo (TAIBI et al., 2021).

**Figura 21** - Arquitetura da Aplicação



Fonte: Elaborado pelo autor

### 3.3. API Gateway

O *API Gateway* é um componente da arquitetura que abstrai as funções essenciais nos microsserviços, atuando como a principal entrada para um *microservice*, encapsulando a implementação e interface interna da aplicação. Funcionando como um servidor HTTP, são definidas rotas e *endpoints* específicas designada para cada função, permitindo o mapeamento

de parâmetros da solicitação e transmitindo mensagens em *JavaScript Object Notation* (JSON). Após a execução da lógica, retorna-se um resultado transformando-o em uma resposta HTTP que será devolvida ao solicitante da API. Numa arquitetura *serverless*, esses controladores geralmente são representados por FaaS (ROBERTS, 2018; ZHAO et al., 2018).

A AWS tem seu próprio *API Gateway*, conhecido como *Amazon API Gateway*. O serviço da AWS é um BaaS, sendo um serviço externo configurado que não há necessidade de ser executado ou provisionado por conta própria, e realiza o gerenciamento de tráfego, controle de autorização e acesso, monitoramento e gerenciamento de versão de APIs, e muito mais. (ROBERTS, 2018; AMAZON WEB SERVICES (AWS), 2023).

O *API Gateway* é utilizado para fornecer uma interface de comunicação entre os clientes e as *Lambda Functions*. Cada *Lambda Function* desenvolvida neste projeto é mapeada para um *endpoint* específico, oferecendo uma *API Representational State Transfer Full* (RESTful) para interação com o sistema. Um dos benefícios do *Amazon API Gateway* é um gerenciamento simplificado dos *endpoints* facilitando a configuração de autorizações e políticas de segurança.

A tabela abaixo contém cada *endpoint* e suas rotas configuradas para acionar cada *lambda function*. Cada *endpoint* foi configurando seguindo o padrão RESTful aplicando-se das boas práticas de implementação dos métodos HTTP.

A API está integrada a 4 *Lambdas Function*, sendo distribuídos quatro *endpoints*, na qual os parâmetros relacionados ao cliente e ao veículo são passados nos *headers* nas requisições que não necessitam informar as chaves para realizar a operação, e no *body* para as requisições que precisam informar os dados do cliente e do veículo, como o *endpoint* de cadastro e de alteração. Os *endpoint* seguem a seguinte distribuição:

- 1 *endpoint* para a consultar clientes com 3 eventos específicos de consulta (todos, cliente, cliente veículo);
- 1 *endpoint* para criação do cliente e dos veículos;
- 1 *endpoint* para alterar dados do cliente e/ou do veículo;
- 1 *endpoint* para deletar dados do cliente e/ou do veículo;



**Tabela 1** - Tabela de descrição dos endpoints

Funções	Método Http	Rotas	Cabeçalhos	Corpo da Requisição
Buscar todos os clientes	GET	/ {estágio} /user/car	N/A	N/A
Buscar clientes por cpf	GET	/ {estágio} /user/car	cpf: string	N/A
Buscar clientes por cpf e placa do veículo	GET	/ {estágio} /user/car	cpf: string licenseplate: string	N/A
Criar cliente e veículos	POST	/ {estágio} /user/car	N/A	JSON { "cpf": "00000000000", "licenseplate": "XXX0X00", "details": { "model": "AAAAA", "color": "BBBBB", "brand": "CCCCC", "year": 0000, "email": "xxxxxxxxx@aaaa.bcd" } }
Alterar cliente e/ou dados dos veículos	PUT	/ {estágio} /user/car	cpf: string licenseplate: string	JSON { "details": { "model": "AAAAA", "color": "BBBBB", "brand": "CCCCC", "year": 0000, "email": "xxxxxxxxx@aaaa.bcd" } }
Deletar cliente e/ou dados dos veículos	DELETE	/ {estágio} /user/car	cpf: string licenseplate: string	N/A

Fonte: Elaborado pelo autor

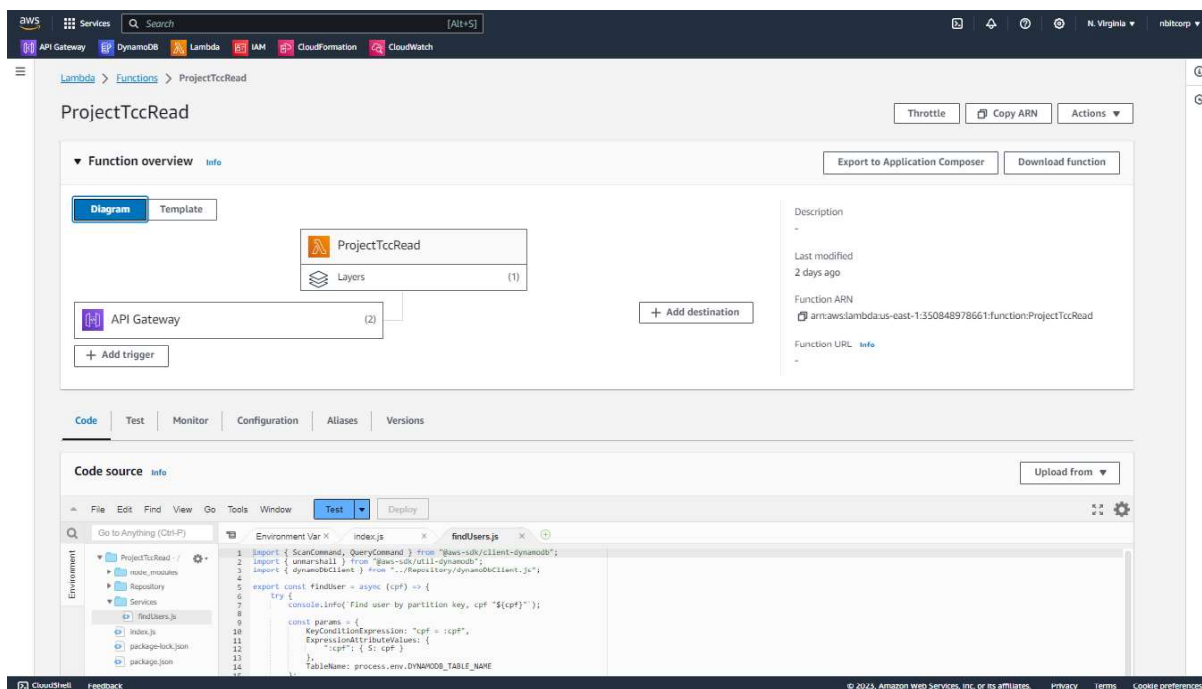
### 3.4. Lambda Function

Cada *lambda function* desenvolvida neste projeto tem como objetivo segregar por contexto a nível de funcionalidades, reduzindo o consumo de funções que são menos utilizadas comumente. Logo o *software* sendo canalizado para cada função a tratar apenas do seu contexto de operacional, reduzirá os custos de manutenção e operação.

No tocante a manutenção, não impactar as demais funcionalidades de um sistema é uma premissa, fazendo com que um time de desenvolvimento possa trabalhar sem correr os riscos de gerar impactos ao *software* como um todo, não impactando a companhia, assim reduzindo os atritos que possam acontecer nas operações, e garantindo melhores testes de regressão, e sendo mais assertivos nos cenários e escopos a serem cobertos. Detalhamento sobre as *lambda functions* desenvolvidas:

- ProjectTccRead (GET): Responsável por buscar todos os clientes, buscar clientes por CPF e buscar clientes por CPF e *License Plate*.
- ProjectTccCreate (POST): Encarregada de criar novos registros de *UserCar*.
- ProjectTccUpdate (PUT): Realiza a alteração dos detalhes do *UserCar* identificado pelo CPF e License Plate.
- ProjectTccDelete (DELETE): Exclui o registro do *UserCar* com base no CPF e *License Plate* fornecidos.

**Figura 22** - Lambda Function ProjectTccRead (GET) e Gatilho via API Gateway



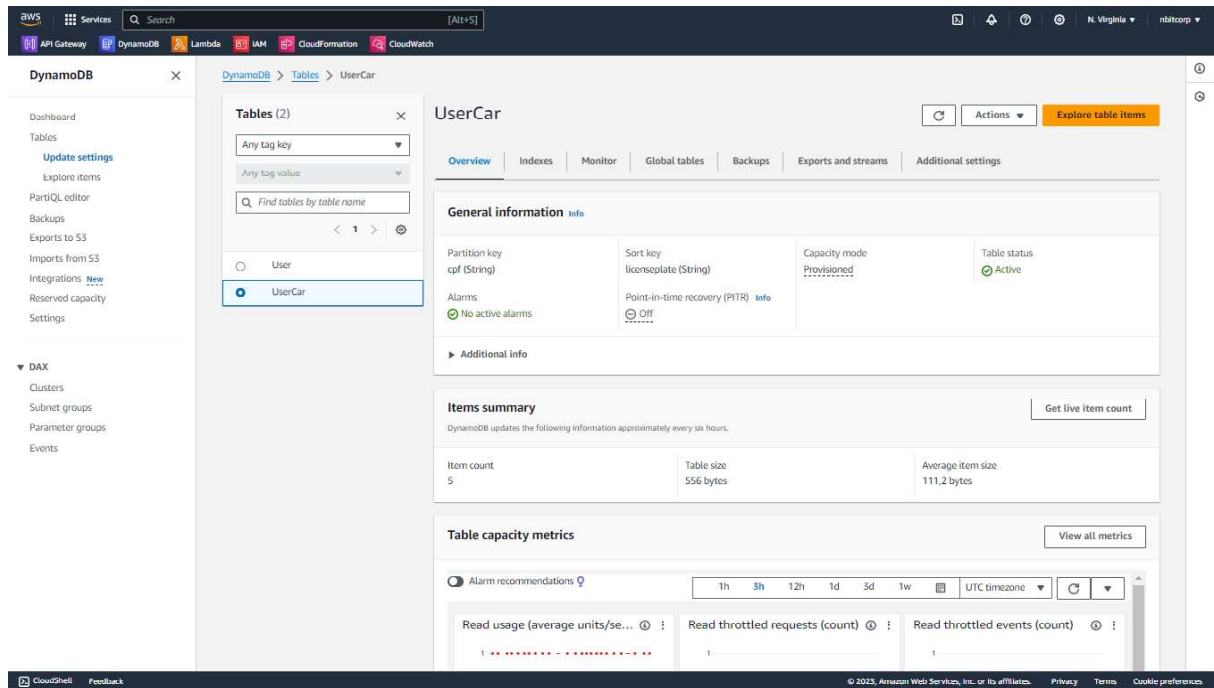
Fonte: Elaborado pelo autor

Diversas companhias podem aderir ao *lambda function* para lidar com *softwares* de baixo volume, reduzindo os custos com recursos de máquina e garantindo coesão, confiabilidade e escalabilidade, mitigando risco que teriam, mantendo sistemas monolíticos, ou até mesmo em microsserviços de contextos maiores.

### 3.5. Banco de Dados NoSQL

A *Amazon DynamoDB* é um banco de dados de NoSQL, totalmente gerenciado, e baseado no modelo *serverless*, sendo projetado para executar aplicações de alta performance grandes escalas, proporcionando segurança, backups contínuos, auto escalável, com armazenamento em cache e ferramentas de importação e exportação de dados, desta forma favorecendo o desenvolvimento de aplicações modernas com redução de custos (AMAZON WEB SERVICES (AWS), 2023).

A escolha do *DynamoDB* como banco de dados NoSQL traz benefícios cruciais para este projeto. A natureza *serverless* do *DynamoDB* se alinha perfeitamente com a arquitetura do sistema, proporcionando escalabilidade automática, desempenho rápido e uma estrutura flexível de chave-valor para armazenar os dados do *UserCar*.

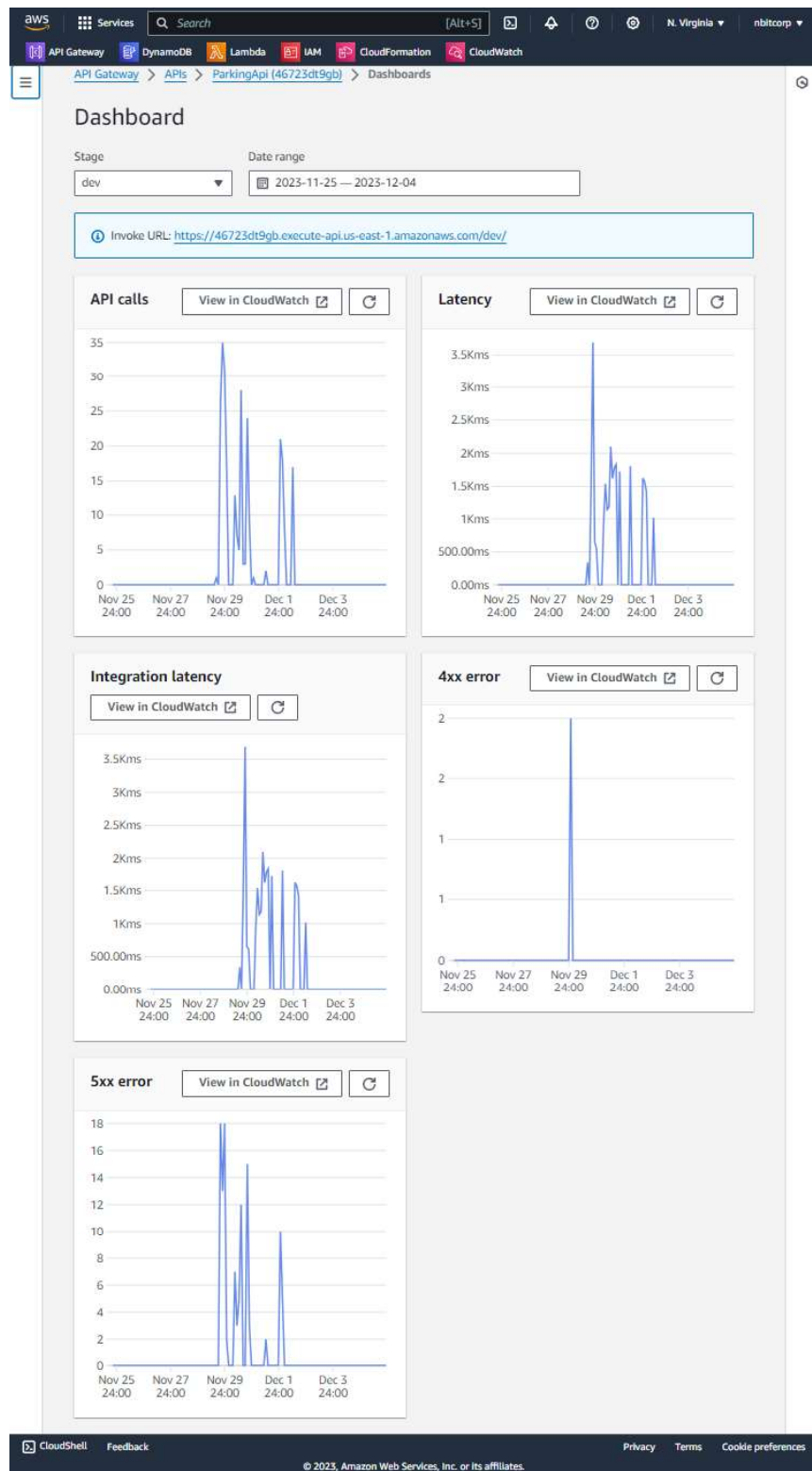
**Figura 23** - Configurações da tabela no DynamoDB

Fonte: Elaborado pelo autor

### 3.6. Monitoramento e Logging

O AWS *CloudWatch* é empregado para monitorar as métricas de desempenho, logs e eventos gerados pelas *Lambda Functions* e pela *API Gateway*. Essa integração permite uma visão abrangente do sistema, facilitando a detecção precoce de problemas e otimização de recursos.

Tais recursos de monitoramento com logs detalhados do *CloudWatch* fornecem detalhes sobre cada chamada à API, facilitando a identificação de possíveis problemas. Fornecendo também métricas em tempo real contribuindo com insights instantâneos sobre o desempenho das *Lambda Functions*.

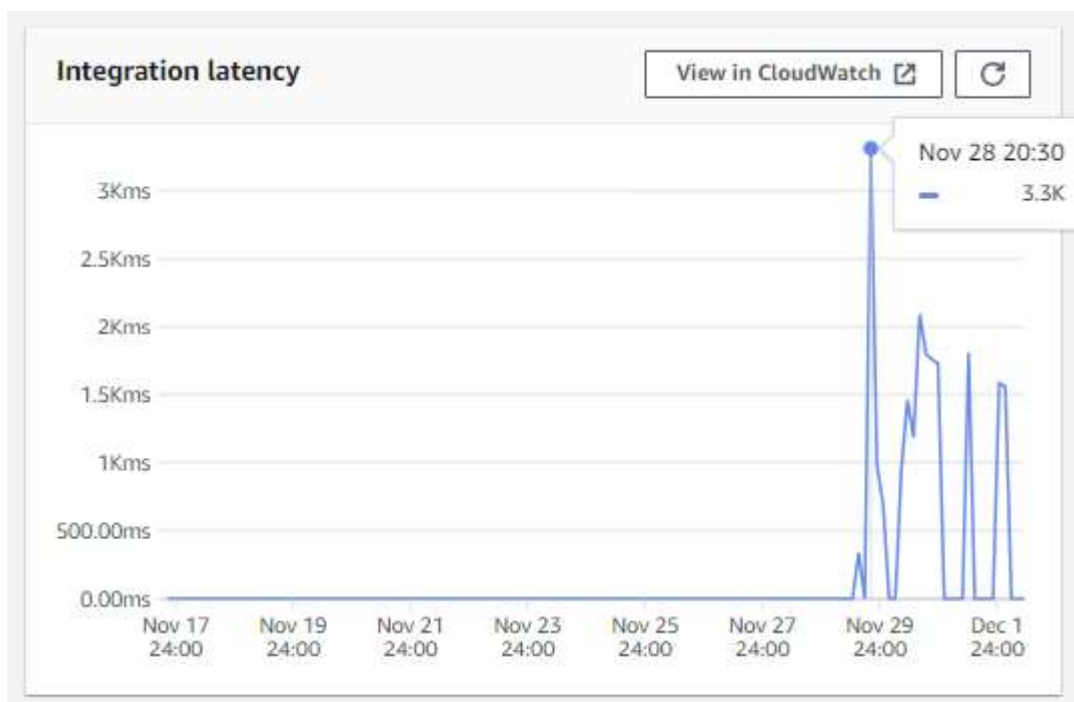
**Figura 24** - Métricas CloudWatch – API Gateway

Fonte: Elaborado pelo autor

A implementação de serviços *serverless* na AWS para um sistema de controle de estacionamento proporciona uma solução moderna, eficiente e escalável. A arquitetura

escolhida, combinando *Lambda Functions*, *DynamoDB*, *API Gateway* e *CloudWatch*, cria uma base sólida para futuras expansões e melhorias, ao mesmo tempo em que otimiza os custos operacionais. Essa abordagem representa um avanço significativo na modernização de sistemas legados de estacionamento, promovendo uma experiência mais ágil e eficaz para os usuários.

**Figura 25** - Efeitos Cold Start no primeiro acionamento



Fonte: Elaborado pelo autor

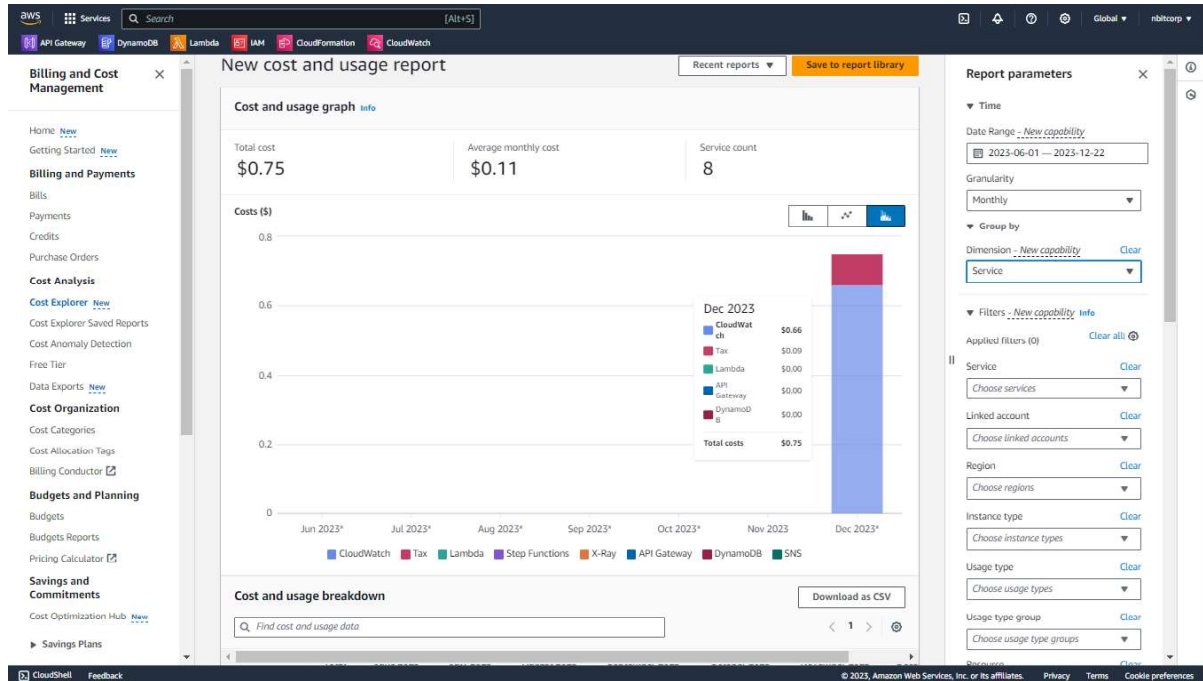
### 3.7. Custos AWS

O Explorador de Custos da AWS tem uma interface amigável e de fácil compreensão permitindo visualizar, entender e gerenciar os custos e o uso da AWS. Com isso, é possível criar relatórios personalizados que analisam dados de custos e uso, com análises dados de resumida ou detalhadas favorecendo a identificar tendências, e determinar os causadores de custos e detectar anomalias. O nível gratuito do *AWS Lambda* fornece por mês 1 milhão de solicitações gratuitas e 400.000 GB/segundo de tempo de computação. (AMAZON WEB SERVICES (AWS), 2023).

Este projeto foi criado utilizando o modelo de conta gratuita. Nesse modelo gratuito o usuário pode desenvolver suas aplicações e fazer a utilização dos serviços presentes no catálogo da AWS, porém alguns serviços são ofertados com determinados limites de utilização. Com isso, pode ser visto que na figura 26 que o limite do *CloudWatch* foi

ultrapassado, gerando uma cobrança na conta, porém o valor a ser cobrado foi um valor de reduzido.

**Figura 26 - Gerenciamento de Custos AWS desta pesquisa**



Fonte: Elaborado pelo autor

### 3.8. Implementação de Código

A adoção do modelo *serverless*, em especial do paradigma FaaS, trouxeram os benefícios de um desenvolvimento simples, favorecendo em códigos limpos e coesos. Estas vantagens também se deram pela linguagem de programação escolhida, o *JavaScript*, fazendo uso do *Node.js*, linguagem suportada pelo *Amazon Lambda Function*.

Neste tópico será apresentado a implementação da *Lambda Function Read*. Logo na figura 27, é apresentado a implementação da função *Read* das operações do CRUD. No código, contém o método *handler*, sendo este o manipulador dos eventos que serão capturados por esta *lambda*, este método irá processar os eventos. Quando uma função é invocada, o *lambda* executa o método do manipulador, e a função será executada até que o manipulador retorne uma resposta, seja encerrando o processo das lógicas de negócio ou atingindo o tempo limite estabelecido nas configurações do *lambda function*.

**Figura 27** - Implementação da Lambda Function Read (GET)

```
import { findUser, findAllUsers, findUserByLicensePlate } from "../Services/findUsers.js";

export const handler = async function (event) {
  try {
    console.log("request:", JSON.stringify(event, undefined, 2));
    let body;

    switch (event.httpMethod) {
      case "GET":
        if (event.headers.cpf != null && event.headers.licenseplate != null) {
          body = await findUserByLicensePlate(event.headers.cpf, event.headers.licenseplate);
        } else if (event.headers.cpf != null) {
          body = await findUser(event.headers.cpf);
        } else {
          body = await findAllUsers();
        }
        break;
      default:
        throw new Error(`Unsupported route: "${event.httpMethod}"`);
    }

    console.log(body);

    return {
      statusCode: 200,
      body: JSON.stringify({
        message: `Successfully finished operation: "${event.httpMethod}"`,
        body: body
      })
    };
  } catch (e) {
    console.error(e);
    return {
      statusCode: 500,
      body: JSON.stringify({
        message: "Failed to perform operation.",
        errorMsg: e.message,
        errorStack: e.stack,
      })
    };
  }
};
```

Fonte: Elaborado pelo autor

Na figura 28, está a camada de domínio e de acesso aos dados que compõem a *Lambda* apresentada. O acesso ao banco *DynamoDB* é gerenciado pela plataforma *Lambda*, sendo de responsabilidade do desenvolvedor informar apenas o nome da tabela, e as operações que serão realizadas no banco.

No código da figura 28 abaixo pode ser visto o método `findUser`, nele será realizado as operações do comando *Query*, sendo montado um objeto contendo os dados da operação, contendo a chave da tabela, que é o CPF, os atributos da consulta e o nome da tabela tornando apto ser executado o comando para retornar os dados do cliente do CPF informado.



**Figura 28** - Implementação dos métodos findUser e findAllUser

```
import { ScanCommand, QueryCommand } from "@aws-sdk/client-dynamodb";
import { unmarshall } from "@aws-sdk/util-dynamodb";
import { dynamoDbClient } from "../Repository/dynamoDbClient.js";

export const findUser = async (cpf) => {
  try {
    console.info(`Find user by partition key, cpf "${cpf}"`);

    const params = {
      KeyConditionExpression: "cpf = :cpf",
      ExpressionAttributeValues: {
        ":cpf": { S: cpf }
      },
      TableName: process.env.DYNAMODB_TABLE_NAME
    };

    const { Items } = await dynamoDbClient.send(new
    QueryCommand(params));
    console.info("GetItem Params:", params);
    console.info("GetItem Result:", Items);

    return Items.map((item) => unmarshall(item));
  } catch (e) {
    console.error(e);
    throw e;
  }
}

export const findAllUsers = async () => {
  try {
    console.log("Get all users");
    const params = {
      TableName: process.env.DYNAMODB_TABLE_NAME
    };


    const { Items } = await dynamoDbClient.send(new ScanCommand(params));
    console.log(Items);
    return (Items) ? Items.map((item) => unmarshall(item)) : {};
  } catch (e) {
    console.error(e);
    throw e;
  }
}
```

Fonte: Elaborado pelo autor

Na figura 28, foi implementado o método findAllUser, método designado a realizar a consulta que retorna todos os usuários e seus veículos. A operação de consulta implementada foi o Scan, seguindo com a construção do objeto de consulta, informando a tabela que será executada o comando, resultando em todos os usuários e seus veículos. O comando Scan

consulta toda a tabela, não sendo muito performático para consultas que necessitem ser filtradas.

**Figura 29** - Implementação do método findUserByLicensePlate



```
import { ScanCommand, QueryCommand } from "@aws-sdk/client-dynamodb";
import { unmarshall } from "@aws-sdk/util-dynamodb";
import { dynamoDbClient } from "../Repository/dynamoDbClient.js";

...

export const findUserByLicensePlate = async (cpf, licenseplate) => {
  try {
    console.log("Find user by license plate");

    const params = {
      KeyConditionExpression: "cpf = :cpf and licenseplate >= :licenseplate",
      ExpressionAttributeValues: {
        ":cpf": { S: cpf },
        ":licenseplate": { S: licenseplate }
      },
      TableName: process.env.DYNAMODB_TABLE_NAME
    };

    const { Items } = await dynamoDbClient.send(new QueryCommand(params));
    console.log(Items);

    return Items.map((item) => unmarshall(item));
  } catch (e) {
    console.error(e);
    throw e;
  }
}
```

Fonte: Elaborado pelo autor

Na figura 29, foi implementado o método que busca um condutor e seu respectivo veículo. O cliente pode ter mais de um automóvel, logo sempre será necessário informar o CPF e a License Plate (Placa de Identificação de Veículos). Para este método foi o utilizando o comando Query, pois ele é mais performático para consultas utilizando as chaves primárias e de classificação.

## 4. RESULTADOS

Neste capítulo são apresentados os resultados obtidos a partir de testes realizados com o desenvolvimento de uma aplicação seguindo o modelo FaaS.

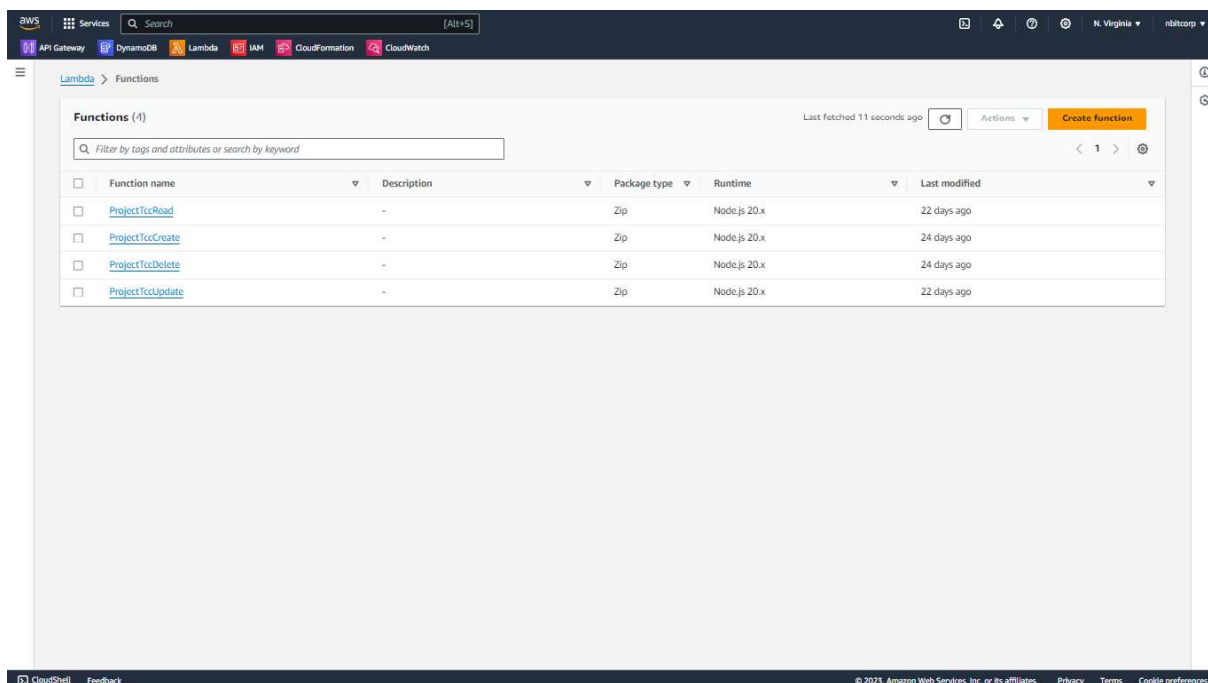
### 4.1. Resultados do desenvolvimento

Como visto neste estudo, adoção de tecnologias *serverless* no desenvolvimento de aplicações utilizando-se do modelo FaaS, tem se tornado uma alternativa para desenvolver sistemas acadêmicos e corporativos, sejam sistemas mais simples ou até mesmo sistemas complexos, pois, implementar *softwares* de forma granular, segregado em funcionalidades menores, específicas para cada contexto, tem-se uma implantação simplificada com menos risco de falhas, tornando o ciclo de vida do *software* mais sustentável, e obtendo um custo reduzido.

Esta pesquisa consiste em implementar um sistema para estacionamento no modelo de MVP, contendo operações básica do CRUD, para criar, listar, alterar e deletar os usuários e seus veículos cadastrados em uma empresa fictícia de estacionamentos de acordo com a operação desejada, utilizando-se do modelo de desenvolvimento de *software* baseado no modelo FaaS proposto neste projeto de pesquisa.

Com a decomposição do *software* por funções operacionais tornou-se possível criar *lambdas function* com suas responsabilidades bem definidas, cada uma sendo executada em uma unidade de processamento e fornecendo seus serviços por meio de uma *API Gateway* integrando os quatro *lambdas functions* com recursos de uma *API RESTful*. Desta forma foi criado um único microsserviço para operação do CRUD, como na figura abaixo.

**Figura 30** - Lista de Lambdas Function por operações do CRUD



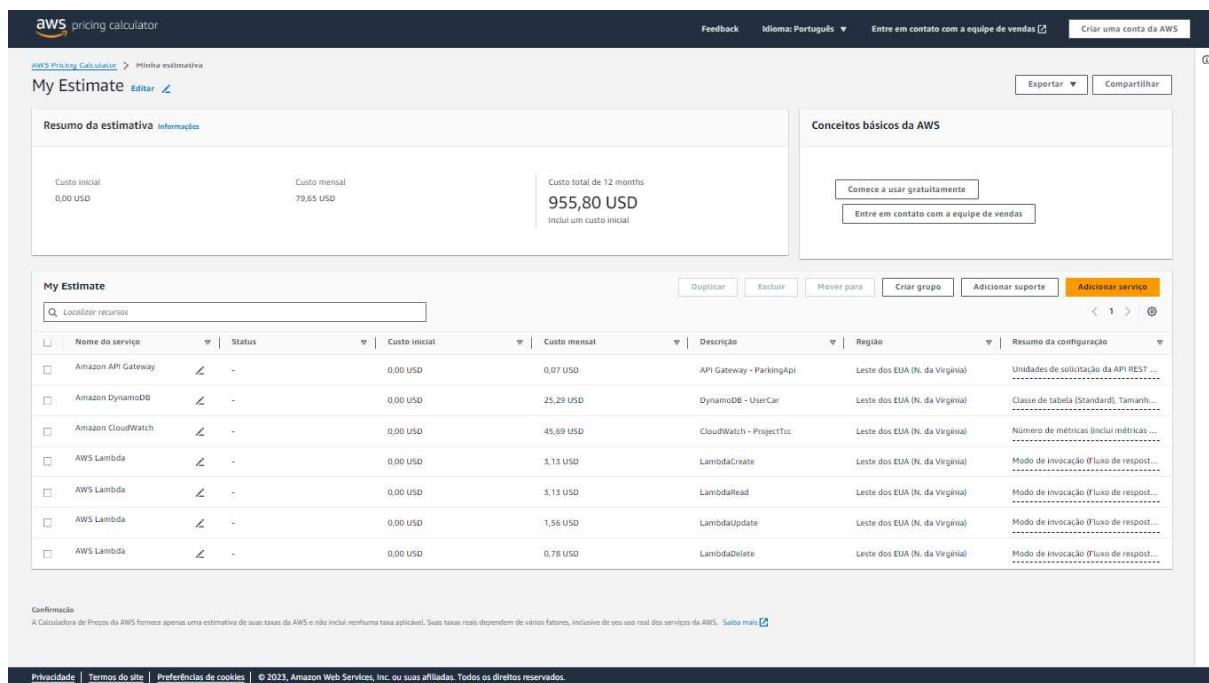
Fonte: Elaborado pelo autor

Neste estudo, foi adotado uma separação por cada operação do CRUD, mas a depender do contexto, pode ser adotado uma segregação por contexto de negócio, onde cada *lambda function* seria responsável por cada modelo de negócio.

O desenvolvimento foi simples e objetivo, por ao segregar em 4 funções, a codificação foi otimizada, sendo criado códigos reduzidos, limpos e coesos, facilitando a leitura e manutenção, e com isso possíveis evoluções de acordo com o modelo de negócio. O resultado desta implementação foi evidenciado na execução, sendo alcançados 3.3 segundo como visto na figura 25, com os efeitos de *cold start*.

Com objetivo de obter custos reduzidos, a tecnologias que usam do paradigma *serverless* foi adotado, alcançando os objetivos pensado para este estudo, sendo evidenciado pela estimativa realizada na calculadora de custos AWS, como na figura abaixo.

**Figura 31 - Cálculo de Custos do Projeto na Calculadora de preços da AWS**



Fonte: Elaborado pelo autor

No levantamento de custos, foi considerado o custo individual para cada serviço utilizado, sendo levantado um custo para atender uma demanda de 20.000 solicitações por mês. Os custos do banco de dados, *DynamoDB* foi considerado 100GB de armazenamento, com 20.000 operações de leitura e escrita, ficando orçado em 25,29 USD.

Os custos com a *API Gateway* foram estimados considerando o volume adotado em toda a aplicação, estimando 20.000 requisições por mês, com um tamanho médio das mensagens de 128 KB, gerando um custo total de 0,07 USD.

As *Lambdas Function* foram separadas em 4 funções, cada uma responsável por uma operação do CRUD. Cada *Lambda* tem alocados 128MB de memória, e 512MB de memória temporária, adotando uma arquitetura x86, sendo considerado também o volume de 1000 solicitações simultâneas, com tamanho médio de resposta de 512KB e provisionado 15ms de tempo médio de duração por solicitação. Por operações de escrita e leitura serem as funcionalidades mais utilizadas num contexto de *software*, foi considerado o volume total adotado na aplicação, de 20.000, para as funções *ProjectTccRead* e *ProjectTccCreate*, pois elas no cenário diário serão as funções mais utilizadas, produzindo um custo de 3,13 USD para cada *lambda* por mês. Para as funções *ProjectTccUpdate* e *ProjectTccDelete*, foram considerados os volumes de 10.000 e 5.000, com 1.000 e 500 de volume de solicitações

simultâneas, adotando os mesmos parâmetros de configurações das *lambdas* de leitura e escrita, logo, foi previsto por mês o preço de 1,56 USD e 0,78 USD, respectivamente.

Os custos mais elevados foram com o serviço de monitoramento e logs, o *Amazon CloudWatch*. Foram alocados 10GB de armazenamentos, e considerado 100.000 eventos, e 30 métricas, podendo essas serem detalhadas ou personalizadas. Foram incluídos métricas considerando o volume de 20.000 requisições da API, de 3.000 eventos por *lambdas* e o valor mínimo de eventos do *DynamoDB*, sendo este de 1 milhão de eventos por mês. Desta forma, o custo total para este serviço foi estimado em 45,69 USD por mês.

Considerado os valores estimados para cada serviço utilizado na aplicação, o valor total previsto para cada mês foi de 79,65 USD, inicialmente com o custo anual programado em 955,80 USD, pois ao se tratar de uma solução *serverless*, os valores serão de acordo com a utilização de recursos de cada tecnologia. Com isto, o objetivo deste estudo foi obtido, com uma implementação simplificada e com custo reduzido, se equiparado a implementação e custo de uma aplicação monolítica e com alocação de recursos de servidores que serão subutilizados.

## 5. CONCLUSÃO

O projeto de desenvolvimento utilizando-se da tecnologia *serverless* com *lambdas functions* trouxe resultados que podem respaldar com efetividade sua aplicação em *softwares* simples com resiliência e eficiência de custos, permitindo manutenibilidade de futuras implementações.

Com os benefícios obtidos na implementação do conceito *serverless*, com a adoção do FaaS, cada *lambda function* aplica sua própria implementação, separadas por funcionalidades, carregando consigo a lógica de negócio para cada contexto. Com isso, torna-se possível a manutenção e execução individual de cada serviço, não impactando as demais funções em casos de falhas, garantido a resiliência de um projeto. Portanto, a independência dos serviços traz performance e robustez para um projeto de *software*, com tempo de requisição e respostas rápidas.

Com a aplicação do FaaS, a redução de custo foi garantida, deste modo, quando os serviços não estiverem sendo utilizados, serão desligados, proporcionando a contenção de gastos elevados. O conceito *serverless* favorece desenvolvedores, acadêmicos e empresas a direcionar novos investimentos e desenvolvimentos motivados pelos custos eficientes alcançados com a aplicação desta tecnologia.

### 5.1. Trabalhos futuros

Para trabalhos futuros, é possível aprimorar os resultados e a performance, implementando melhorias buscando reduzir os efeitos causados pelo *cold start* e prevenir-se ao *Spawn Effect*. Torna-se essencial implementar soluções apresentadas no levantamento bibliográfico sobre *Function-as-a-Services*, ou analisar novas abordagens para mitigar estes desafios presentes nessa tecnologia.

Com o objetivo de aprimorar a eficiência financeira de aplicações baseadas em tecnologias *serverless*, é crucial aprofundar-se em estratégias avançadas de gestão de custos, monitoramento e métricas, evitando custos não previstos para a aplicação. Adicionalmente, uma abordagem mais detalhada e estruturada na análise de padrões de desenvolvimento se mostra essencial para garantir a otimização contínua dessas soluções. É benéfico explorar essas soluções para o desenvolvimento eficiente e sustentável de aplicações neste contexto inovador.

## REFERÊNCIAS BIBLIOGRÁFICAS

ABGAZ, Yalemisew et al. Decomposition of Monolith Applications Into Microservices Architectures: a systematic review. *Ieee Transactions On Software Engineering*, [S.L.], v. 49, n. 8, p. 4213-4242, ago. 2023. Institute of Electrical and Electronics Engineers (IEEE). <http://dx.doi.org/10.1109/tse.2023.3287297>.

AHMADVAND, Mohsen; IBRAHIM, Amjad. Requirements Reconciliation for Scalable and Secure Microservice (De)composition. 2016 *Ieee 24Th International Requirements Engineering Conference Workshops (Rew)*, [S.L.], p. 68-73, set. 2016. IEEE. <http://dx.doi.org/10.1109/rew.2016.026>.

AIKEN, Howard Hathaway et al. Log Book With Computer Bug: computer bug. *Computer Bug*. 1947. Id Number:1994.0191.01 Catalog Number:1994.0191.1 Accession Number:1994.0191. Disponível em: [https://americanhistory.si.edu/collections/nmah\\_334663](https://americanhistory.si.edu/collections/nmah_334663). Acesso em: 11 dez. 2023.

AMAZON WEB SERVICES (AWS) (ed.). Amazon DynamoDB. 2023. Disponível em: <https://aws.amazon.com/pt/dynamodb/>. Acesso em: 14 dez. 2023.

AMAZON WEB SERVICES (AWS) (ed.). Lambda Limits. 2023. Disponível em: <https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html>. Acesso em: 14 dez. 2023.

AMAZON WEB SERVICES (AWS) (ed.). Serverless on AWS: build and run applications without thinking about servers. Build and run applications without thinking about servers. 2023. Disponível em: <https://aws.amazon.com/pt/serverless/>. Acesso em: 14 dez. 2023.

AMAZON WEB SERVICES (AWS) (ed.). What is Amazon API Gateway? 2023. Disponível em: <https://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html>. Acesso em: 14 dez. 2023.

AMAZON WEB SERVICES (AWS) (ed.). What is AWS Lambda? 2023. Disponível em: <https://docs.aws.amazon.com/lambda/latest/dg/welcome.html>. Acesso em: 14 dez. 2023.

AMAZON WEB SERVICES (AWS). AWS Pricing Calculator. 2023. Disponível em: <https://calculator.aws/#/estimate>. Acesso em: 14 dez. 2023.



ARMBRUST, Michael et al. A view of cloud computing. *Communications Of The Acm*, [S.L.], v. 53, n. 4, p. 50-58, abr. 2010. Association for Computing Machinery (ACM). <http://dx.doi.org/10.1145/1721654.1721672>.

AUER, Florian et al. From monolithic systems to Microservices: an assessment framework. *Information And Software Technology*, [S.L.], v. 137, p. 106600, set. 2021. Elsevier BV. <http://dx.doi.org/10.1016/j.infsof.2021.106600>.

BALALAIE, Armin et al. Microservices migration patterns. *Software: Practice and Experience*, [S.L.], v. 48, n. 11, p. 2019-2042, 24 jul. 2018. Wiley. <http://dx.doi.org/10.1002/spe.2608>.

BALDINI, Ioana et al. Serverless Computing: current trends and open problems. *Research Advances In Cloud Computing*, [S.L.], p. 1-20, 2017. Springer Singapore. [http://dx.doi.org/10.1007/978-981-10-5026-8\\_1](http://dx.doi.org/10.1007/978-981-10-5026-8_1).

BARRAK, Amine et al. Serverless on Machine Learning: a systematic mapping study. *Ieee Access*, [S.L.], v. 10, p. 99337-99352, 2022. Institute of Electrical and Electronics Engineers (IEEE). <http://dx.doi.org/10.1109/access.2022.3206366>.

CARREIRA, Joao et al. Cirrus. *Proceedings Of The Acm Symposium On Cloud Computing*, [S.L.], v. 1, n. 1, p. 13-24, 20 nov. 2019. ACM. <http://dx.doi.org/10.1145/3357223.3362711>.

CARVALHO, Antonio Gledson de; PINHEIRO, Roberto B.; SAMPAIO, Joelson Oliveira. Dotcom Bubble and Underpricing: Conjectures and Evidence. 2017. Disponível em: <https://repositorio.fgv.br/items/9be84e46-cdcd-4827-8fc7-66884c0108d5>. Acesso em: 14 dez. 2023.

CARVALHO, Leonardo Rebouças de; ARAÚJO, Aletéia Patricia Favacho de. FaaS-Oriented Node.js Applications in an RPC Approach Using the Node2FaaS Framework. *Ieee Access*, [S.L.], v. 11, p. 112027-112043, 2023. Institute of Electrical and Electronics Engineers (IEEE). <http://dx.doi.org/10.1109/access.2023.3322936>.

CASTRO, Paul et al. The rise of serverless computing. *Communications Of The Acm*, [S.L.], v. 62, n. 12, p. 44-54, 21 nov. 2019. Association for Computing Machinery (ACM). <http://dx.doi.org/10.1145/3368454>.

"CHADHA, Mohak et al. Towards Federated Learning using FaaS Fabric. Proceedings Of The 2020 Sixth International Workshop On Serverless Computing, [S.L.], v. 1, n. 1, p. 49-54, 7 dez. 2020. ACM. <http://dx.doi.org/10.1145/3429880.3430100>.

"

CHAHAL, Dheeraj et al. High Performance Serverless Architecture for Deep Learning Workflows. 2021 Ieee/Acm 21St International Symposium On Cluster, Cloud And Internet Computing (Ccgrid), [S.L.], p. 790-796, maio 2021. IEEE. <http://dx.doi.org/10.1109/ccgrid51090.2021.00096>.

CHAHAL, Dheeraj et al. SLA-aware Workload Scheduling Using Hybrid Cloud Services. Proceedings Of The 1St Workshop On High Performance Serverless Computing, [S.L.], p. 1-4, 25 jun. 2020. ACM. <http://dx.doi.org/10.1145/3452413.3464789>.

CHARD, Ryan et al. Serverless Supercomputing: high performance function as a service for science. Arxiv, [S.L.], v. 1, n. 1, p. 1-13, 2019. ArXiv. <http://dx.doi.org/10.48550/ARXIV.1908.04907>.

CHEN, Rui et al. From Monolith to Microservices: a dataflow-driven approach. 2017 24Th Asia-Pacific Software Engineering Conference (Apsec), [S.L.], p. 466-475, dez. 2017. IEEE. <http://dx.doi.org/10.1109/apsec.2017.53>.

CHRISTIDIS, Angelos et al. Enabling Serverless Deployment of Large-Scale AI Workloads. Ieee Access, [S.L.], v. 8, p. 70150-70161, 2020. Institute of Electrical and Electronics Engineers (IEEE). <http://dx.doi.org/10.1109/access.2020.2985282>.

CLARKE, Paul et al. A complexity theory viewpoint on the software development process and situational context. Proceedings Of The International Conference On Software And Systems Process, [S.L.], p. 86-90, 14 maio 2016. ACM. <http://dx.doi.org/10.1145/2904354.2904369>.

CRISTIAN SPOIALA. Pros and Cons of Serverless Computing. FaaS comparison: AWS Lambda vs Azure Functions vs Google Functions. 2019. Disponível em: <https://assist-software.net/blog/pros-and-cons-serverless-computing-faas-comparison-aws-lambda-vs-azure-functions-vs-google>. Acesso em: 14 dez. 2023.

DAOUD, Mohamed et al. A multi-model based microservices identification approach. *Journal Of Systems Architecture*, [S.L.], v. 118, p. 102200, set. 2021. Elsevier BV. <http://dx.doi.org/10.1016/j.sysarc.2021.102200>.

DONNO, Michele de et al. Foundations and Evolution of Modern Computing Paradigms: cloud, iot, edge, and fog. *Ieee Access*, [S.L.], v. 7, p. 150936-150948, 2019. Institute of Electrical and Electronics Engineers (IEEE). <http://dx.doi.org/10.1109/access.2019.2947652>.

DRAGONI, Nicola et al. Microservices: yesterday, today, and tomorrow. *Present And Ulterior Software Engineering*, [S.L.], p. 195-216, 2017. Springer International Publishing. [http://dx.doi.org/10.1007/978-3-319-67425-4\\_12](http://dx.doi.org/10.1007/978-3-319-67425-4_12).

DUBEY, Kalka et al. A Management System for Servicing Multi-Organizations on Community Cloud Model in Secure Cloud Environment. *Ieee Access*, [S.L.], v. 7, p. 159535-159546, 2019. Institute of Electrical and Electronics Engineers (IEEE). <http://dx.doi.org/10.1109/access.2019.2950110>.

DUIGNAN, Brian. Dot-com bubble: stock market [1995-2000]. stock market [1995–2000]. 2023. Disponível em: <https://www.britannica.com/event/dot-com-bubble>. Acesso em: 14 dez. 2023.

ESKI, Sinan; BUZLUCA, Feza. An automatic extraction approach. *Proceedings Of The 19Th International Conference On Agile Software Development: Companion*, [S.L.], p. 1-6, 21 maio 2018. ACM. <http://dx.doi.org/10.1145/3234152.3234195>.

FEATHERLY, Kevin. ARPANET: united states defense program. United States defense program. 2023. Disponível em: <https://www.britannica.com/topic/ARPANET>. Acesso em: 08 dez. 2023.

FLANAGAN, David. JavaScript: O Guia Definitivo. 6. ed. Porto Alegre - Rs: Bookman Editora, 2013. 1080 p. (8565837483, 9788565837484). Tradução de: João Eduardo Nóbrega Tortello. Disponível em: <https://books.google.com.br/books?id=zWNyDgAAQBAJ&lpg=PR1&ots=IBuey7HeeQ&dq=%20JavaScript%3A%20O%20Guia%20Definitivo&lr&pg=PR1#v=onepage&q&f=false>. Acesso em: 14 dez. 2023.

FOTOUHI, Mohammadbagher et al. Function-as-a-Service Application Service Composition. *Proceedings Of The 5Th International Workshop On Serverless Computing*, [S.L.], p. 49-54, 9 dez. 2019. ACM. <http://dx.doi.org/10.1145/3366623.3368141>.

FOWLER, Martin; LEWIS, James. Microservices: a definition of this new architectural term. a definition of this new architectural term. 2014. Disponível em: <https://martinfowler.com/articles/microservices.html>. Acesso em: 14 dez. 2023.

FOX, Geoffrey C. et al. Status of Serverless Computing and Function-as-a-Service(FaaS) in Industry and Research. Arxiv, [S.L.], v. 1, n. 1, p. 76-100, 2017. ArXiv. <http://dx.doi.org/10.48550/ARXIV.1708.08028>.

FRANKSTON, Bob. The JavaScript Ecosystem. Ieee Consumer Electronics Magazine, [S.L.], v. 9, n. 6, p. 84-89, 1 nov. 2020. Institute of Electrical and Electronics Engineers (IEEE). <http://dx.doi.org/10.1109/mce.2020.3009457>.

FREIBERGER, Paul A.; SWAINE, Michael R.. ENIAC: computer. computer. 2023. Disponível em: <https://www.britannica.com/technology/ENIAC>. Acesso em: 08 dez. 2023.

FURDA, Andrei et al. Migrating Enterprise Legacy Source Code to Microservices: on multitenancy, statefulness, and data consistency. Ieee Software, [S.L.], v. 35, n. 3, p. 63-72, maio 2018. Institute of Electrical and Electronics Engineers (IEEE). <http://dx.doi.org/10.1109/ms.2017.440134612>.

GOOGLE CLOUD (ed.). What is Cloud Computing? 2023. Disponível em: <https://cloud.google.com/learn/what-is-cloud-computing>. Acesso em: 08 dez. 2023.

GROGAN, Jake et al. A Multivocal Literature Review of Function-as-a-Service (FaaS) Infrastructures and Implications for Software Developers. Communications In Computer And Information Science, [S.L.], p. 58-75, 2020. Springer International Publishing. [http://dx.doi.org/10.1007/978-3-030-56441-4\\_5](http://dx.doi.org/10.1007/978-3-030-56441-4_5).

GUHA, R; AL-DABASS, D. Impact of Web 2.0 and Cloud Computing Platform on Software Engineering. 2010 International Symposium On Electronic System Design, [S.L.], v. 1, n. 1, p. 213-218, dez. 2010. IEEE. <http://dx.doi.org/10.1109/ised.2010.48>.

GYSEL, Michael et al. Service Cutter: a systematic approach to service decomposition. Service-Oriented And Cloud Computing, [S.L.], p. 185-200, 2016. Springer International Publishing. [http://dx.doi.org/10.1007/978-3-319-44482-6\\_12](http://dx.doi.org/10.1007/978-3-319-44482-6_12).

IBM (ed.). IBM's ASCC introduction. 2023. Disponível em: [https://www.ibm.com/ibm/history/exhibits/markI/markI\\_intro.html](https://www.ibm.com/ibm/history/exhibits/markI/markI_intro.html). Acesso em: 08 dez. 2023.

IBM (ed.). What is cloud computing? 2023. Disponível em: <https://www.ibm.com/topics/cloud-computing>. Acesso em: 08 dez. 2023.

IBM. Microservices in the enterprise, 2021: Real benefits, worth the challenges. 2021. Elaborado por: IBM Market Development & Insights. Disponível em: <https://www.ibm.com/downloads/cas/OQG4AJAM>. Acesso em: 14 dez. 2023.

JANGDA, Abhinav et al. Formal foundations of serverless computing. Proceedings Of The Acm On Programming Languages, [S.L.], v. 3, n. , p. 1-26, 10 out. 2019. Association for Computing Machinery (ACM). <http://dx.doi.org/10.1145/3360575>.

JARACHANTHAN, Jananie et al. AMPS-Inf: automatic model partitioning for serverless inference with cost efficiency. 50Th International Conference On Parallel Processing, [S.L.], p. 1-12, 9 ago. 2021. ACM. <http://dx.doi.org/10.1145/3472456.3472501>.

JIN, Wuxia et al. Service Candidate Identification from Monolithic Systems Based on Execution Traces. Ieee Transactions On Software Engineering, [S.L.], v. 47, n. 5, p. 987-1007, 1 maio 2021. Institute of Electrical and Electronics Engineers (IEEE). <http://dx.doi.org/10.1109/tse.2019.2910531>.

JOHN A. PAULSON SCHOOL OF ENGINEERING & APPLIED SCIENCE. Havard University - Collection Of Historical Scientific Instruments (ed.). Harvard IBM Mark I - Automatic Sequence Controlled Calculator: about the mark i. About the Mark I. 2023. Disponível em: <https://chsi.harvard.edu/harvard-ibm-mark-1-about>. Acesso em: 08 dez. 2023.

JOHN A. PAULSON SCHOOL OF ENGINEERING & APPLIED SCIENCE. Havard University - Collection Of Historical Scientific Instruments (ed.). Harvard IBM Mark I - Automatic Sequence Controlled Calculator: how did it work?. How did it work?. 2023. Disponível em: <https://chsi.harvard.edu/harvard-ibm-mark-1-function>. Acesso em: 08 dez. 2023.

JOHN A. PAULSON SCHOOL OF ENGINEERING & APPLIED SCIENCE. Havard University - Collection Of Historical Scientific Instruments (ed.). Harvard IBM Mark I - Automatic Sequence Controlled Calculator: the first programming textbook. The First Programming Textbook. 2023. Disponível em: <https://chsi.harvard.edu/harvard-ibm-mark-1-manual>. Acesso em: 08 dez. 2023.

JONAS, Eric et al. Occupy the cloud. Proceedings Of The 2017 Symposium On Cloud Computing, [S.L.], v. 1, n. 1, p. 445-451, 24 set. 2017. ACM. <http://dx.doi.org/10.1145/3127479.3128601>.

KAHN, Robert; DENNIS, Michael Aaron. Internet: computer network. computer network. 2023. Disponível em: <https://www.britannica.com/technology/Internet>. Acesso em: 08 dez. 2023.

KALSKE, Miika et al. Challenges When Moving from Monolith to Microservice Architecture. Current Trends In Web Engineering, [S.L.], p. 32-47, 2018. Springer International Publishing. [http://dx.doi.org/10.1007/978-3-319-74433-9\\_3](http://dx.doi.org/10.1007/978-3-319-74433-9_3).

KANAGARAJ, K; GEETHA, S. A Hybrid Framework for Effective Prediction of Online Streaming Data. Journal Of Physics: Conference Series, [S.L.], v. 1767, n. 1, p. 012016, 1 fev. 2021. IOP Publishing. <http://dx.doi.org/10.1088/1742-6596/1767/1/012016>.

KAPLUNOVICH, Alex; YESHA, Yelena. Refactoring of Neural Network Models for Hyperparameter Optimization in Serverless Cloud. Proceedings Of The Ieee/Acm 42Nd International Conference On Software Engineering Workshops, [S.L.], p. 311-314, 27 jun. 2020. ACM. <http://dx.doi.org/10.1145/3387940.3392268>.

KING, Brittany. What is FaaS? Function as a Service explained. 2022. Disponível em: <https://www.digitalocean.com/blog/what-is-faas-function-as-a-service-explained>. Acesso em: 14 nov. 2023.

KOSCHEL, Arne; KLASSEN, Samuel; JDIYA, Kerim; SCHAAF, Marc; ASTROVA, Irina. Cloud Computing: serverless. 2021 12Th International Conference On Information, Intelligence, Systems & Applications (Iisa), [S.L.], v. 1, n. 1, p. 1-7, 12 jul. 2021. IEEE. <http://dx.doi.org/10.1109/iisa52424.2021.9555534>.

LI, Shanshan et al. A dataflow-driven approach to identifying microservices from monolithic applications. Journal Of Systems And Software, [S.L.], v. 157, p. 110380, nov. 2019. Elsevier BV. <http://dx.doi.org/10.1016/j.jss.2019.07.008>.

LIU, Fang et al. NIST Cloud Computing Reference Architecture: recommendations of the national institute of standards and technology. Recommendations of the National Institute of Standards and Technology. 2011. Disponível em: <https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication500-292.pdf>. Acesso em: 14 dez. 2023.

LOUKIDES, Mike; SWOYER, Steve. Microservices Adoption in 2020. 2020. Disponível em: <https://www.oreilly.com/radar/microservices-adoption-in-2020/>. Acesso em: 14 dez. 2023.

MELL, Peter; GRANCE, Timothy. The NIST Definition of Cloud Computing: recommendations of the national institute of standards and technology. Recommendations of the National Institute of Standards and Technology. 2011. Disponível em: <https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf>. Acesso em: 14 dez. 2023.

MENDONCA, Nabor C. et al. The Monolith Strikes Back: why istio migrated from microservices to a monolithic architecture. Ieee Software, [S.L.], v. 38, n. 5, p. 17-22, set. 2021. Institute of Electrical and Electronics Engineers (IEEE). <http://dx.doi.org/10.1109/ms.2021.3080335>.

MVONDO, Djob et al. OFC. Proceedings Of The Sixteenth European Conference On Computer Systems, [S.L.], p. 228-244, 21 abr. 2021. ACM. <http://dx.doi.org/10.1145/3447786.3456239>.

NASR, Aida A. et al. A novel water pressure change optimization technique for solving scheduling problem in cloud computing. Cluster Computing, [S.L.], v. 22, n. 2, p. 601-617, 23 nov. 2018. Springer Science and Business Media LLC. <http://dx.doi.org/10.1007/s10586-018-2867-7>.

NEWMAN, Sam. Building Microservices: designing fine-grained systems. 2. ed. Sebastopol, Ca: O'Reilly Media, Inc., 2022. (9781492034025). Disponível em: <https://books.google.com.br/books?id=ZvM5EAAAQBAJ&lpg=PT8&ots=ui4h8E9JYs&dq=%20Building%20Microservices&lr&pg=PP1#v=onepage&q=Building%20Microservices&f=false>. Acesso em: 14 dez. 2023.

NORR, Henry R.. Netscape Communications Corp.: american company. American company. 2023. Disponível em: <https://www.britannica.com/topic/Netscape-Communications-Corp>. Acesso em: 14 dez. 2023.

OWENS, Dustin. Securing elasticity in the cloud. Communications Of The Acm, [S.L.], v. 53, n. 6, p. 46-51, jun. 2010. Association for Computing Machinery (ACM). <http://dx.doi.org/10.1145/1743546.1743565>.

PELLE, Istvan et al. Operating Latency Sensitive Applications on Public Serverless Edge Cloud Platforms. Ieee Internet Of Things Journal, [S.L.], v. 8, n. 10, p. 7954-7972, 15 maio

2021. Institute of Electrical and Electronics Engineers (IEEE). <http://dx.doi.org/10.1109/jiot.2020.3042428>.

PENN ENGINEERING. University Of Pennsylvania. Celebrating Penn Engineering History: ENIAC. 2017. Disponível em: <https://www.seas.upenn.edu/about/history-heritage/eniac/>. Acesso em: 08 dez. 2023.

RAJPUT, Tushar. Function-as-a-Service (FaaS) Market Size Worth USD 31.53 Billion at CAGR of 32.3%, By 2026 - Report and Data. 2021. Disponível em: [https://www.einnews.com/pr\\_news/552783688/function-as-a-service-faas-market-size-worth-usd-31-53-billion-at-cagr-of-32-3-by-2026-report-and-data](https://www.einnews.com/pr_news/552783688/function-as-a-service-faas-market-size-worth-usd-31-53-billion-at-cagr-of-32-3-by-2026-report-and-data). Acesso em: 14 dez. 2023.

REDHAT (ed.). Understanding cloud computing. 2023. Disponível em: <https://www.redhat.com/en/topics/cloud>. Acesso em: 08 dez. 2023.

REDHAT (ed.). What are microservices? 2023. Disponível em: <https://www.redhat.com/en/topics/microservices/what-are-microservices>. Acesso em: 14 dez. 2023.

REDHAT (ed.). What is serverless? 2022. Disponível em: <https://www.redhat.com/en/topics/cloud-native-apps/what-is-serverless>. Acesso em: 14 dez. 2023.

RIBEIRO, Mauro et al. MLaaS: machine learning as a service. 2015 Ieee 14Th International Conference On Machine Learning And Applications (Icmla), [S.L.], p. 896-902, dez. 2015. IEEE. <http://dx.doi.org/10.1109/icmla.2015.152>.

RICHEY, Kevin W.. The ENIAC. 1997. Disponível em: <https://ei.cs.vt.edu/%7Ehistory/ENIAC.Richey.HTML>. Acesso em: 08 dez. 2023.

RISTOV, Sashko et al. Colder Than the Warm Start and Warmer Than the Cold Start! Experience the Spawn Start in FaaS Providers. Proceedings Of The 2022 Workshop On Advanced Tools, Programming Languages, And Platforms For Implementing And Evaluating Algorithms For Distributed Systems, [S.L.], p. 35-39, 25 jul. 2022. ACM. <http://dx.doi.org/10.1145/3524053.3542751>.

ROBERTS, Mike. Serverless Architectures. 2018. Disponível em: <https://martinfowler.com/articles/serverless.html>. Acesso em: 14 dez. 2023.



SAIDANI, Islem et al. Towards Automated Microservices Extraction Using Multi-objective Evolutionary Search. *Service-Oriented Computing*, [S.L.], p. 58-63, 2019. Springer International Publishing. [http://dx.doi.org/10.1007/978-3-030-33702-5\\_5](http://dx.doi.org/10.1007/978-3-030-33702-5_5).

SCHLEIER-SMITH, Johann et al. What serverless computing is and should become. *Communications Of The Acm*, [S.L.], v. 64, n. 5, p. 76-84, 26 abr. 2021. Association for Computing Machinery (ACM). <http://dx.doi.org/10.1145/3406011>.

SCHÜTZ, Sebastian Walter et al. The Impact of Software-as-a-Service on Software Ecosystems. *Lecture Notes In Business Information Processing*, [S.L.], p. 130-140, 2013. Springer Berlin Heidelberg. [http://dx.doi.org/10.1007/978-3-642-39336-5\\_13](http://dx.doi.org/10.1007/978-3-642-39336-5_13).

SELMADJI, Anfel et al. Re-architecting OO Software into Microservices. *Service-Oriented And Cloud Computing*, [S.L.], p. 65-73, 2018. Springer International Publishing. [http://dx.doi.org/10.1007/978-3-319-99819-0\\_5](http://dx.doi.org/10.1007/978-3-319-99819-0_5).

SEWAK, Mohit; SINGH, Sachchidanand. Winning in the Era of Serverless Computing and Function as a Service. 2018 3Rd International Conference For Convergence In Technology (I2Ct), [S.L.], v. 1, n. , p. 1-5, abr. 2018. IEEE. <http://dx.doi.org/10.1109/i2ct.2018.8529465>.

SHAH, Hezbullah; SOOMRO, Tariq Rahim. Node.js Challenges in Implementation. *Global Journal Of Computer Science And Technology*. Framingham, p. 72-83. maio 2017. Disponível em: [https://www.researchgate.net/publication/318310544\\_Nodejs\\_Challenges\\_in\\_Implementation](https://www.researchgate.net/publication/318310544_Nodejs_Challenges_in_Implementation). Acesso em: 15 dez. 2023.

SHANKAR, Vaishaal et al. Serverless linear algebra. *Proceedings Of The 11Th Acm Symposium On Cloud Computing*, [S.L.], v. 1, n. 1, p. 281-295, 12 out. 2020. ACM. <http://dx.doi.org/10.1145/3419111.3421287>.

SMITH, Tom. New Research Shows 63% of Enterprises Are Adopting Microservices Architectures. 2018. Disponível em: <https://dzone.com/articles/new-research-shows-63-percent-of-enterprises-are-a>. Acesso em: 14 dez. 2023.

STEINBACH, Markus et al. TppFaaS: modeling serverless functions invocations via temporal point processes. *Ieee Access*, [S.L.], v. 1, n. 1, p. 9059-9084, 2022. Institute of Electrical and Electronics Engineers (IEEE). <http://dx.doi.org/10.1109/access.2022.3144078>.

TAIBI, Davide et al. Processes, Motivations, and Issues for Migrating to Microservices Architectures: an empirical investigation. *Ieee Cloud Computing*, [S.L.], v. 4, n. 5, p. 22-32, set. 2017. Institute of Electrical and Electronics Engineers (IEEE). <http://dx.doi.org/10.1109/mcc.2017.4250931>.

TAIBI, Davide et al. Serverless Computing-Where Are We Now, and Where Are We Heading? *Ieee Software*, [S.L.], v. 38, n. 1, p. 25-31, jan. 2021. Institute of Electrical and Electronics Engineers (IEEE). <http://dx.doi.org/10.1109/ms.2020.3028708>.

TELEGEOGRAPHY (comp.). Submarine Cable Map. 2023. Disponível em: <https://www.submarinecablemap.com/>. Acesso em: 14 dez. 2023.

THE NEW YORK TIMES (ed.). Electronic Computer Flashes Answers, May Speed Engineering; NEW ALL-ELECTRONIC COMPUTER AND ITS INVENTORS. 1946. T.r. Kennedy Jr.. Disponível em: <https://www.nytimes.com/1946/02/15/archives/electronic-computer-flashes-answers-may-speed-engineering-new.html>. Acesso em: 14 dez. 2023.

VEUVOLU, Rakesh et al. Cloud Computing Based (Serverless computing) using Serverless architecture for Dynamic Web Hosting and cost Optimization. 2023 International Conference On Computer Communication And Informatics (Iccci), [S.L.], v. 1, n. 1, p. 1-6, 23 jan. 2023. IEEE. <http://dx.doi.org/10.1109/iccci56745.2023.10128286>.

VILLAMIZAR, Mario et al. Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS Lambda architectures. *Service Oriented Computing And Applications*, [S.L.], v. 11, n. 2, p. 233-247, 27 abr. 2017. Springer Science and Business Media LLC. <http://dx.doi.org/10.1007/s11761-017-0208-y>.

WANG, Hao et al. Distributed Machine Learning with a Serverless Architecture. *Ieee Infocom 2019 - Ieee Conference On Computer Communications*, [S.L.], p. 1288-1296, abr. 2019. IEEE. <http://dx.doi.org/10.1109/infocom.2019.8737391>.

WANG, Liang et al. Peeking Behind the Curtains of Serverless Platforms. 2018 Usenix Annual Technical Conference (Usenix Atc 18). Boston, Ma, p. 133-146. jul. 2018. Disponível em: <https://www.usenix.org/conference/atc18/presentation/wang-liang>. Acesso em: 15 dez. 2023.

WEN, Jinfeng et al. An empirical study on challenges of application development in serverless computing. *Proceedings Of The 29Th AcM Joint Meeting On European Software*

Engineering Conference And Symposium On The Foundations Of Software Engineering, [S.L.], p. 416-428, 18 ago. 2021. ACM. <http://dx.doi.org/10.1145/3468264.3468558>.

WOLFART, Daniele et al. Modernizing Legacy Systems with Microservices: a roadmap. Evaluation And Assessment In Software Engineering, [S.L.], p. 149-159, 21 jun. 2021. ACM. <http://dx.doi.org/10.1145/3463274.3463334>.

YANG, Yanan et al. Replication package for INFless. Artifact Digital Object Group, [S.L.], p. 768-781, 18 fev. 2022. Association for Computing Machinery (ACM). <http://dx.doi.org/10.1145/3462314>.

YU, Minchen et al. Gillis: serving large neural networks in serverless functions with automatic model partitioning. 2021 Ieee 41St International Conference On Distributed Computing Systems (Icdcs), [S.L.], p. 138-148, jul. 2021. IEEE. <http://dx.doi.org/10.1109/icdcs51616.2021.00022>.

ZHANG, Chengliang et al. MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving. 2019 Usenix Annual Technical Conference (Usenix Atc 19). Renton, Wa, p. 1049-1062. jul. 2019. Disponível em: <https://www.usenix.org/conference/atc19/presentation/zhang-chengliang>. Acesso em: 15 dez. 2023.

ZHANG, Michael et al. STOIC: serverless teleoperable hybrid cloud for machine learning applications on edge device. 2020 Ieee International Conference On Pervasive Computing And Communications Workshops (Percom Workshops), [S.L.], p. 1-6, mar. 2020. IEEE. <http://dx.doi.org/10.1109/percomworkshops48775.2020.9156239>.

ZHAO, J T et al. Management of API Gateway Based on Micro-service Architecture. Journal Of Physics: Conference Series, [S.L.], v. 1087, p. 032032, set. 2018. IOP Publishing. <http://dx.doi.org/10.1088/1742-6596/1087/3/032032>.

ZIMMERMANN, Olaf. Microservices tenets. Computer Science - Research And Development, [S.L.], v. 32, n. 3-4, p. 301-310, 16 nov. 2016. Springer Science and Business Media LLC. <http://dx.doi.org/10.1007/s00450-016-0337-0>.