Arquitetura Serverless: contexto histórico, vantagens, desafios e análise prática

Lucas Pereira Carvalho
Departamento de Ciência da Computação
Universidade Federal de Minas Gerais
Belo Horizonte, MG - Brasil
lucaspc20@ufmg.br

Resumo - A arquitetura serverless tem como propósito simplificar o desenvolvimento ao abstrair a gestão de infraestrutura, promovendo vantagens como controle de custo granular e escalabilidade automática. Contudo, essa abstração traz algumas desvantagens, em especial, o *vendor lock-in*. Este trabalho a partir de um referencial teórico, busca entender o contexto que levou ao surgimento da arquitetura serverless, vantagens e desvantagens e características de aplicações desenvolvidas com essa arquitetura. Foi feito também o desenvolvimento prático de uma aplicação de pós-processamento de imagens nos dois maiores provedores de nuvem do mercado: AWS e Azure, somando ao trabalho uma análise e vivência prática dos pontos analisados, além de detalhar informações relevantes durante o processo de desenvolvimento

Abstract - Serverless architecture aims to simplify development by abstracting infrastructure management, offering advantages such as granular cost control and automatic scalability. However, this abstraction introduces some disadvantages, most notably vendor lock-in. Based on a theoretical framework, this work seeks to understand the context that led to the emergence of serverless architecture, as well as its advantages, disadvantages, and the characteristics of applications developed with this architecture. A practical development of an image post-processing application was also carried out on the two largest cloud providers in the market, AWS and Azure, adding a practical analysis and hands-on experience of the analyzed points to the work, in addition to detailing relevant information from the development process.

***Index Terms*—serverless, arquitetura serverless, computação em nuvem, cloud, aplicações serverless, funções serverless, provedor de nuvem**

I. Introdução

Serverless, uma arquitetura de software que tem sido destaque no meio de computação em nuvem devido a sua simplicidade e o baixo nível de gerenciamento necessário. Conforme Li et al., (2023) explicam [1], o foco dos desenvolvedores fica centrado em escrever as funções que serão executadas, utilizando linguagens de alto nível, e disponibilizá-las em plataformas de nuvem, que ficam agora responsáveis pela gerência das aplicações.

Pontos como a diminuição das despesas operacionais, preocupação com configuração do ambiente, cobrança pelo tempo de uso dos recursos computacionais e autoscaling preciso são grandes incentivadores para adoção da computação serverless. Entretanto, dado o alto nível de abstração trazido pelo baixo controle da infraestrutura, surge o problema de alto acoplamento das aplicações a um determinado provedor de nuvem. Conforme [2] apresenta, até mesmo simples casos de uso de serverless acabam lidando com problemas na migração de provedor de nuvem.

Motivado por tais desafios e questões a serem melhor analisadas, este trabalho surge com o intuito de melhor entender a arquitetura serverless - desde o seu surgimento, o contexto histórico, como foi o avanço da computação em nuvem até o surgimento da computação serverless; apresentar vantagens e desvantagens encontradas ao se optar por essa arquitetura, por meio de um referencial teórico. Com estas informações em mãos, é desenvolvido uma aplicação serverless de pós-processamento de imagens em dois provedores de nuvem, AWS e Azure, com todo o processo de desenvolvimento descrito e com uma análise das diferenças, vantagens, desvantagens e desafios visualizados durante o processo.

Este trabalho se organiza da seguinte forma: primeiramente na seção 2, a partir de um referencial bibliográfico, é apresentado uma contextualização e introdução à ideia de computação serverless, desde os conceitos básicos de *cloud computing*; à evolução da computação em nuvem e o devido acompanhamento dos provedores de nuvem; o surgimento da computação serverless - o porquê, seu funcionamento, vantagens e desafios inerentes.

A seção 3 foca em apresentar o desenvolvimento da aplicação de pós-processamento de imagens. São descritos os processos de desenvolvimento, as ferramentas e filosofias que cada provedor de nuvem propõe, medidas e estratégias abordadas para lidar com dificuldades enfrentadas, e comparações entre pontos chave entre ambos provedores.

Na seção 4 é apresentado o que a literatura propõe para o avanço da arquitetura serverless, as principais formas de mitigação de problemas ao se desenvolver uma aplicação serverless, e os principais frameworks e demais ferramentas que existem atualmente no mercado voltado para o desenvolvimento de aplicações serverless.

É apresentado ao final, na seção 5, uma conclusão sintetizando os principais pontos analisados e identificados durante o trabalho, levando-se em consideração os aspectos teóricos juntamente da experiência prática obtida durante o desenvolvimento e implementação da aplicação. A partir da junção teórica e prática, pôde-se obter um conhecimento mais aprofundado desta arquitetura moderna.

II. Referencial Teórico

Ao começo do século XXI, o conceito de *cloud computing* cresceu exponencialmente. Bhagat et al. (2023) [3] afirmam que ainda no século XX a ideia de computação em uma máquina externa e com mais recursos já existia; na década de 1950, nos primórdios da computação, o processamento ocorria em máquinas centrais exorbitantes compartilhadas; já em 1972, IBM apresenta um sistema de máquinas virtuais, surgindo a ideia de virtualização; e em 2006, a Amazon Web Services (AWS), lança o Elastic Compute Cloud (EC2), um grande marco no mercado de tecnologia.

Golden (2007) [4], define virtualização como o compartilhamento e coordenação de recursos de um único hardware, por meio de um sistema operacional *host* (hospedeiro), para diversos sistemas operacionais *guests* (visitantes), de forma que esses *guests*, não percebam esse compartilhamento, ou seja, essa virtualização de acesso aos recursos da máquina.

O mercado acompanha a evolução da computação em nuvem. Borra (2024) [5] apresentou um estudo a respeito dos provedores de computação em nuvem. Destaca-se que nas últimas duas décadas, vários provedores surgiram no mercado, mas poucos conseguiram destaque. Os principais provedores contam com mais de 100 serviços diferentes em seus catálogos. O autor destaca a Amazon Web Services (AWS) - com 32% de market share, Microsoft Azure e Google Cloud Platform (GCP).

Os grandes provedores de nuvem contam com um grande gama de serviços disponíveis para os diversos tipos de clientes, ampliando cada vez mais a adoção da computação em nuvem. Desde serviços de máquinas virtuais, armazenamento de dados, ferramentas de métricas e controle de gastos, ferramentas e plataformas de análise e visualização de dados, balanceamento de carga, entre diversos outros listados em [5].

Os serviços oferecidos pelos provedores de nuvem, podem ser classificados majoritariamente em três modelos de serviço, conforme [3] e [5] descrevem:

- SaaS (Software as a Service): Aplicações disponibilizadas como um serviço por meio da internet, na qual os clientes e os usuários finais podem acessar por meio de um cliente, como um navegador, ou aplicativo móvel. O serviço é mantido e gerenciado pelo provedor, e o cliente geralmente é cobrado por meio de uma assinatura para ter acesso ao serviço desejado. Exemplos seriam Gmail, Outlook e Youtube.
- PaaS (Platform as a Service): É ofertado um ambiente de trabalho pelo provedor de nuvem, no qual os clientes irão desenvolver suas aplicações de maneira facilitada e mais simples, abrindo mão de um nível de configuração mais avançado. Um exemplo de Paas ofertado é o *Google App Engine*.
- IaaS (Infrastructure as a Service): O provedor oferece ao cliente acesso a recursos computacionais, que podem ser diretamente acessados, configurados e orquestrados, assim como a capacidade de configuração e escolha do sistema operacional e programas que serão executados neste ambiente. Um exemplo deste serviço é o *Amazon Elastic Compute Cloud* (EC2).

A computação em nuvem atingiu o potencial e alcance esperado. Pesquisadores da Universidade da Califórnia em Berkeley, em 2009 publicaram o artigo *Above the Clouds: A Berkeley View of Cloud Computing* [6], de referência no contexto de computação em nuvem. Segundo os autores, há três aspectos chave no contexto de computação em nuvem que levaram ao alto uso deste serviço: a ilusão de recursos computacionais infinitos disponíveis sob demanda; a eliminação de um compromisso inicial por parte dos usuários de computação em nuvem; a capacidade de pagar pelo uso de recursos computacionais conforme necessário

Destaque para o terceiro ponto, que permitiu a escalabilidade dinâmica dos recursos computacionais. Segundo Armbrust et al., (2009) [6], "o uso de 1000 máquinas EC2 por uma hora, têm um custo semelhante ao uso de uma máquina por 1000 horas". Essa escalabilidade horizontal mostrou seu poder quando os usuários necessitaram utilizar dos serviços de computação em nuvem, especialmente dos serviços de máquinas virtuais, como o *Amazon Compute Cloud* (EC2) nos anos iniciais da computação em nuvem. Em 2019, pesquisadores da Universidade da Califórnia em Berkeley, publicaram uma revisão do artigo anterior, sob o título de *Cloud Programming Simplified: A Berkeley View on Serverless Computing* [7]. Uma das razões da alta adoção do uso de serviços de máquinas virtuais, como o EC2, segundo os autores, foi devido que nos anos iniciais da computação em nuvem, os desenvolvedores desejavam recriar o mesmo ambiente de desenvolvimento que possuíam em seus servidores particulares, e tais serviços de máquinas virtuais permitem essa configuração mais aprofundada, por se tratar de um serviço sob o modelo de Infraestrutura como Serviço (IaaS).

Seguindo a lógica dos modelos PaaS e SaaS, que promovem a adoção da computação em nuvem, por abstraírem certo nível de complexidade, surge a arquitetura serverless. O foco se desloca do provisionamento de infraestrutura para a execução direta de funções, eliminando a necessidade de gerenciar servidores, além de permitir que o cliente desenvolva a aplicação que deseja, e não simplesmente compre ou assine um serviço pré-estabelecido como ocorre no modelo SaaS. Para Li et al., (2023), toda essa popularidade e adoção pelo mercado pela arquitetura serverless, se dá pela sua simplicidade e facilidade de configuração e disponibilização. Na computação serverless, os desenvolvedores necessitam escrever funções em linguagens de alto nível, como Java, Python e Go, e as enviar para o sistema dos provedores de nuvem, terceirizando o provisionamento, administração e configuração.

Para Jonas et al., (2019) [7], a computação serverless representa uma abordagem que integra dois serviços principais

oferecidos pelos provedores de nuvem: Function as a Service (FaaS) e Backend as a Service (BaaS). A ideia principal de serverless estão nas *cloud functions* (funções da nuvem), no qual as aplicações são construídas por conjuntos de funções que são invocadas pela internet em resposta a eventos, como requisições HTTP, armazenamento de um novo dado em um banco de dados, alarmes agendados, uma nova mensagem em um serviço de mensageria, entre outros. Essas funções são gerenciadas pelos provedores de nuvem, serviço conhecido como FaaS. Além disso, os provedores disponibilizam serviços específicos que essas aplicações necessitam, como bancos de dados, serviços de armazenamento de objetos e gerenciador de aplicações - esse conjunto de serviços criam um ecossistema que suportam a execução das funções é caracterizado como BaaS. Dessa forma, os autores definem a computação serverless como uma combinação de FaaS e BaaS, fornecida aos usuários de maneira integrada.

Em 2014, no evento re:Invent [8], a Amazon Web Services (AWS) apresentou seu novo produto - AWS Lambda. Surge como uma forma de oferecer soluções para problemas que podem ser resolvidos com a execução de uma função de até mesmo uma linha, respondendo a eventos - como exemplificado na apresentação - de escrita, cópia e remoção de dados em um serviço de armazenamento de objetos. Ao longo dos anos, demais provedores de nuvem passaram a oferecer seus respectivos serviços de Function as a Code, como Azure Functions, Google Cloud Functions, CloudFlare Workers, Alibaba Cloud Function, entre outros; isso junto a diversos frameworks e adicionais iniciativas de código aberto.

Embora as aplicações ainda necessitem de um servidor, o nome serverless está relacionado com o fato de que o usuário do serviço de nuvem somente escreve as funções que serão executadas em resposta à eventos, deixando o provisionamento, administração e configuração de distribuição e escalonamento para o provedor de nuvem. Conforme Li et al. (2023) [1] apontam, ao longo do tempo, os modelos de desenvolvimento de computação em nuvem foram gradualmente caminhando para soluções mais automatizadas e com menor nível de configuração. De máquinas virtuais altamente customizáveis e configuráveis, a serviços de plataforma para desenvolvimento, conteinerização, até aos serviços serverless, reduzindo a carga de trabalho dos desenvolvedores em questões de complexidade de configuração e operacional.

A diversa disponibilização de serviços referente à computação serverless pelos provedores de nuvem é justificada pelo mercado. Uma análise do Market.us, publicada em 2024 [9], aponta que para o mercado global de serverless se espera um crescimento considerável nos próximos dez anos. Partindo de 11.5 bilhões de dólares de valor de mercado, para 98.8 bilhões de dólares até 2033 - uma taxa de crescimento anual composta (CAGR) de 24%. Reforçando a ideia de fácil adoção devido ao baixo custo de operação pelas empresas que adotam esse modelo de computação em nuvem.

Junto da redução de complexidade em termos de configuração e disponibilização das aplicações, a computação serverless traz também um controle mais fino e granular em questão de custos. A cobrança é proporcional ao tempo de uso dos recursos computacionais do provedor de nuvem, e não mais referente a alocação desses recursos, que acontece nos modelos tradicionais de computação em nuvem - denominados frequentemente de *serverfull*.

Para Lin et al., (2020) [10], o modelo de cobrança na computação serverless pode ser definido como *pay-as-you-go*, ou seja, a cobrança é sobre o consumo e não sobre a capacidade. Em seu trabalho *Serverless Boom or Bust? An analysis of economic incentives*, os autores buscam entender as vantagens e incentivos que clientes de computação em nuvem têm ao optarem pela computação serverless. Os clientes desejam que suas aplicações sejam disponibilizadas, seja pagando um preço fixo pela alocação dos recursos alocados - como uma máquina virtual - ou pagando pelo tempo de execução da aplicação - como no caso das *cloud functions*. Os resultados do trabalho mostram que a computação serverless ainda é significativamente subutilizada, principalmente nos casos de aplicações que precisam lidar com picos de acessos/requisições, situação na qual o modelo *pay-as-you-go* resulta em custos financeiros menores, que uma alocação de recurso estática, conforme demonstram os experimentos dos autores.

Relacionado ao modelo de cobrança por uso, um outro grande benefício que a computação serverless promove é o autoscale preciso. Eismann et al., (2022) [11] em seu trabalho *The State of Serverless Applications: Collection, Characterization, and Community Consensus*, explica a irregularidade do nível de carga que as funções serverless enfrentam. Verificou-se no trabalho que 81% das funções das aplicações analisadas, lidam com uma carga *bursty* (intermitente), ou seja, percebe-se que a carga em aplicações serverless têm uma característica irregular, lidando com picos de uso e carga. Além disso, outro ponto relevante no trabalho que foi analisado é a quantidade de funções. Somente 7% das aplicações usam mais que 10 funções, e em 82% dos casos, foram usadas 5 ou menos funções. Os autores relacionam isso ao fato do uso de serviços externos reduzirem a quantidade de código escrito pelos desenvolvedores para construir uma aplicação.

A computação serverless lida bem com a característica de carga intermitente, devido a ideia de elasticidade do zero ao "infinito", conforme [12] apresentam. A responsabilidade de disponibilização de mais CPU, memória e outros recursos computacionais ficam voltadas ao provedor de nuvem em períodos de maior necessidade; assim como em períodos em que não é necessário nenhuma computação, os clientes não são cobrados, pois nenhum recurso computacional fica alocado ao cliente - por isso a ideia de escalabilidade do zero ao "infinito".

Uma outra característica da computação serverless é o forte isolamento entre os serviços, promovendo maior compartilhamento de hardware entre clientes. Uma vez que o usuário define as funções a serem executadas, junto da declaração das possíveis bibliotecas utilizadas, realiza-se a disponibilização do código para o provedor de nuvem. Os provedores de nuvem também se beneficiam com tal característica de isolamento da computação serverless. Conforme [13] exemplificam, em um caso no qual uma aplicação recebe poucas requests por minuto, e realiza-se uma computação simples, a média do uso

de CPU é muito baixo; é possível alocar diversas aplicações semelhantes nessa mesma máquina para maximizar o uso da CPU, ao invés de espalhar por várias máquinas diferentes. Isso é possível na computação serverless, mesmo tratando de serviços de diferentes clientes, devido a característica de isolamento das *cloud functions*, ampliando os lucros para os provedores de nuvem. Os clientes por sua vez, tiram proveito da maior segurança proporcionada pelo alto isolamento do funcionamento das diversas funções que são executadas em um mesmo servidor.

Mesmo com tantas vantagens, Eismann et al., (2021) [14] buscaram entender de fato: o porquê de as empresas optarem pela arquitetura serverless; quando a arquitetura serverless é bem aplicada; e como as aplicações serverless são implementadas na prática, a partir do estudo de 89 projetos serverless coletados de fontes comerciais, acadêmicas e open-source.

Primeiramente a respeito do porquê, os autores reforçam a ideia apresentada anteriormente, de que serverless traz uma vantagem de custo devido ao modelo *pay-as-you-go*, especialmente em aplicações com cargas irregulares, que foi o caso em 47% das aplicações. Além disso, um outro ponto citado anteriormente, é a redução de despesas operacionais por parte dos desenvolvedores, apontado em 37% das aplicações.

Em relação ao cenário de uso, ou seja, quando é utilizado a arquitetura serverless, a pesquisa desmonta a ideia de que serverless é adequado apenas para funcionalidades secundárias de sistemas. O estudo revela que 42% das aplicações serverless implementam funcionalidades principais, enquanto 39% são voltadas para funções utilitárias e secundárias. Outro achado relevante foi a questão do tempo de execução e volume de dados. A maioria das aplicações analisadas (69%) lida com volumes de dados inferiores a 10 MB, e 75% das funções possuem tempo de execução na faixa de segundos. Esse resultado reforça que serverless é frequentemente utilizado para tarefas curtas.

A respeito de como as aplicações serverless são implementadas, o estudo mostra que na maioria dos casos é optado por utilizar a AWS, cobrindo 80% das aplicações, seguido por Azure com 10% dos casos. Em relação às linguagens de programação, as mais populares foram JavaScript e Python (42% cada), que os autores inferem essa preferência por serem linguagens interpretadas e terem um tempo de inicialização menor que linguagens compiladas, minimizado os *cold starts*.

Dadas as diversas vantagens descritas que a computação serverless proporciona, surgem também desafios e pontos a serem melhor analisados, e que são o intuito de diversos estudos e pesquisas atualmente. Como é uma arquitetura oposta ao tradicional modelo *serverfull*, muitas tecnologias e soluções estabelecidas lidam com problemas ao se depararem com o modelo serverless. A seguir são apresentados trabalhos relacionados aos problemas enfrentados na arquitetura serverless, em especial o de vendor lock-in.

O trabalho *Serverless Computing: State of the Art, Challenges and Opportunities* de Li et al., (2023) [1] oferece uma análise abrangente sobre os principais desafios e oportunidades no campo da computação serverless. O trabalho destaca questões importantes como latência de inicialização, isolamento, políticas de escalonamento e tolerância a falhas, explorando as soluções existentes na literatura e identificando áreas que ainda necessitam de mais estudos. Fornece uma visão geral do estado da arte e contextualiza as dificuldades enfrentadas no desenvolvimento e operação de aplicações serverless.

Há também o trabalho de Shafiei et al., (2022) [13] - *Serverless Computing: A Survey of Opportunities, Challenges and Applications* - que faz uma análise e revisão sistemática de trabalhos relacionados a computação serverless, agrupando as aplicações serverless em oito principais implementações, e mapeando as motivações e benefícios que impulsionam a adoção desse paradigma em cada um dos tipos de implementação. Os autores também apresentam uma lista de desafios relacionados à computação serverless, identificando lacunas e áreas que estão demandam mais estudos.

Por se tratar de uma arquitetura relativamente nova e que vem ganhando popularidade e uso por parte do mercado, diversos trabalhos como os apresentados anteriormente são publicados, visando apresentar alguns desafios, possíveis soluções e áreas abertas para novas pesquisas.

Um problema já bem discutido tanto na literatura, quanto pelas organizações e desenvolvedores é o de vendor lock-in. Opara-Martins et al. (2016) [15] definem vendor lock-in na computação em nuvem como a situação em que os clientes são dependentes, estão presos (locked-in) a um provedor de nuvem e não conseguem facilmente mover para outro sem gastos significativos, sejam por questões legais ou incompatibilidades tecnológicas. A pesquisa do trabalho, baseada em uma combinação de abordagens qualitativas e quantitativas, identifica os principais fatores de risco associados ao vendor lock-in, como a falta de padronização em interfaces e APIs proprietárias, a ausência de formatos abertos para troca de dados entre diferentes provedores de nuvem. O vendor lock-in foi identificado como uma das principais barreiras na adoção da computação em nuvem pelas organizações do Reino Unido. Segundo os autores, a falta de conscientização por parte das empresas sobre esses riscos é um fator crítico que intensifica o problema, e uma das principais formas de mitigação seria um bom planejamento anterior ao desenvolvimento de aplicações na nuvem para a escolha de quais serviços utilizar, e provenientes de quais provedores de nuvem, assim como uma análise detalhada do contrato de serviço que será fechado.

Weldemicheal (2023) [16] foca em seu trabalho, *Vendor lock-in and its impact on cloud computing migration*, analisar o impacto que o vendor lock-in traz para as empresas. Muitas se veem na posição de migrar para um outro provedor devido a um novo serviço, nova tecnologia ou custos melhores, porém se deparam com a barreira do vendor lock-in ao tentarem realizar a migração. Muitas vezes é necessário fazer uma rearquitetura do sistema quase que por completo. O autor realiza uma pesquisa qualitativa com diversas desenvolvedores que lidam ou lidaram com o problema de vendor lock-in em suas organizações, e busca entender o impacto que tecnologias proprietárias dos provedores de nuvem, o efeito de rede (o uso cada vez maior de serviços internos de um mesmo provedor de

nuvem) e termos de contrato têm em relação ao problema. Os resultados mostram que as medidas tomadas que mais surtiram efeito para a mitigação do problema foi adotar uma estratégia multi-cloud (utilização de serviços de nuvem provenientes de mais de um provedor), negociação de contratos mais flexíveis e com termos de rescisão simplificadas, e o emprego de soluções de código aberto quando possível.

III. Projeto Prático

A fim de se analisar e visualizar na prática como se estrutura uma aplicação serverless, assim como as características citadas ao longo do trabalho, esta sessão apresenta e descreve o processo de desenvolvimento de uma aplicação serverless de pós-processamento de imagens. Foi escolhido este projeto por ter um certo nível de complexidade, necessidade de lidar com biblioteca externa para manipulação das imagens, assim como a necessidade de se integrar diferentes serviços que permitam que usuários realizem upload, edição e download de suas imagens.

Foram selecionados os dois principais provedores e com maior adesão no mercado, AWS e Azure, conforme mostram Eismann et al., (2021) [14]. Possuem documentações extensivas, uma quantidade relevante de conteúdos feitos pela comunidade e fornecem planos gratuitos que permitiram o desenvolvimento deste trabalho sem custos financeiros.

Optou-se por desenvolver a aplicação utilizando a linguagem Python, por tratar-se de uma das linguagens mais utilizadas no contexto de aplicações serverless, como [14] também mencionam. Além disso possui uma sintaxe de fácil entendimento e pouco verbosa, muito relevante para este contexto, pois também como analisado anteriormente, tem-se como objetivo ter funções coesas, curtas e com pouca ou uma única responsabilidade. A versão escolhida foi a 3.11, por ser uma versão recente (lançada ao final de 2022), ambos provedores fornecem um *runtime* compatível, e pela estabilidade da biblioteca Pillow com esta versão - principal biblioteca externa utilizada.

A. Cobrança e nível gratuito

Ambos provedores fornecem serviços e planos gratuitos, com algumas diferenças a serem levadas em consideração. Em ambos provedores, os serviços gratuitos se enquadram dentre uma das seguintes categorias:

- Gratuito por 12 meses: serviços que a partir da primeira data de uso, possuem um limite mensal para o qual não são cobrados.
- Sempre gratuito: refere-se a serviços que não possui uma data limite para se iniciar a cobrança, é necessário somente se atentar aos limites mensais de uso

A documentação da AWS a respeito da cobrança de seus serviços é bem detalhada. É fornecida uma página com informações a respeito de cada serviço [1] enquadrado em uma das categorias gratuitas. Além disso, uma ferramenta disponível na AWS foi amplamente utilizada para monitorar e evitar que custos inesperados ocorressem, o *Billing and Cost Management*. A ferramenta lista todo o consumo dos serviços gratuitos, atualizada várias vezes ao dia. É possível estabelecer um *budget* para a conta, ou seja, delimitar um valor em que o usuário é informado que um custo ocorreu, indicando que o limite gratuito de um serviço tenha sido extrapolado.

No contexto da Azure, embora o catálogo de serviços gratuitos seja menor, e a descrição do nível gratuito dos serviços não seja tão descritivo [2], é fornecido aos estudantes um plano de assinatura específico. Nomeado de *Microsoft Azure for students*, o plano oferece a estudantes com acesso a email de uma universidade, além do acesso aos serviços gratuitos, um crédito de 100 dólares por um período de um ano. Para o desenvolvimento deste trabalho foi feita a inscrição e obtenção desta oferta, que se mostrou de grande valia para absorver custos inesperados que ultrapassaram os limites do plano gratuito. Em um dos meses durante o desenvolvimento da aplicação, o limite referente a escrita de dados em um armazenamento do tipo blob foi excedido, e graças ao crédito fornecido, não houve custos financeiros. Existe também na Azure, uma ferramenta chamada de *Cost Management*, que fornece informações de consumo para a assinatura e também fornece um serviço de alerta caso algum serviço tenha o limite gratuito atingido. No caso o alerta ocorreu com alguns minutos de atraso, mas mostrou-se útil e funcional.

Relacionado ao contexto de cobrança, é necessário esclarecer uma diferença fundamental organização interna dos provedores. Na AWS a conta é a unidade fundamental de uma organização, os consumos e cobranças estão vinculadas a uma conta.

Para a Azure existe uma diferença, no qual é aplicado uma lógica de hierarquia. Cada assinatura está associada a um tenant, que funciona como pilar de uma organização. A lógica de um tenant também é relevante no contexto de autenticação dos usuários, citado posteriormente. Cada assinatura funciona como um contêiner lógico para provisionar recursos, com seus próprios limites de faturamento e permissões, similar ao conceito de conta da AWS, porém está sempre associado a um tenant. Essa forma de organização permite que um único tenant tenha assinaturas separadas para diferentes finalidades, como desenvolvimento e produção. No caso deste trabalho, foi mantida uma única assinatura fornecida por meio do plano estudantil.

A tabela I apresenta um comparativo dos limites gratuitos para os serviços utilizados durante o desenvolvimento do trabalho. Os dados foram extraídos das páginas detalhando o nível gratuito de cada provedor. A escolha, necessidade e funcionamento de cada serviço são detalhados nas seções específicas sobre o desenvolvimento.

Outro ponto a se destacar é a questão de regiões. Tanto a AWS quanto a Azure organizam sua infraestrutura global em regiões, que são localidades geográficas distintas (como leste e oeste dos EUA ou América do sul) contendo múltiplos centros de dados isolados. A escolha da região de implementação

[1] aws.amazon.com/free

[2] https://azure.microsoft.com/pt-br/pricing/free-services

Tabela I
COMPARAÇÃO DO NÍVEL GRATUITO: AWS E AZURE

Serviço	AWS	Azure
Funções Serverless	**AWS Lambda:** 1 milhão de requisições por mês. 400.000 GB-segundos de tempo de computação por mês.	**Azure Functions:** 1 milhão de requisições por mês e 400.000 GB-segundos de execução de recursos por mês.
Gateway de API	**Amazon API Gateway:** 1 milhão de chamadas por mês.	**Azure API Management:** 1 milhão de chamadas por mês no plano de Consumo.
Armazenamento de Objetos	**Amazon S3:** 5 GB de armazenamento padrão, 20.000 requisições Get e 2.000 requisições Put, Copy, Post ou List por mês (válido por 12 meses).	**Azure Blob Storage:** 5 GB de armazenamento padrão, 20.000 requisições de leitura e 10.000 de escrita (válido por 12 meses).
Gerenciamento de Identidade	**Amazon Cognito:** 50.000 Usuários Ativos Mensais (MAUs).	**Microsoft Entra ID:** 50.000 Usuários Ativos Mensais (MAUs).
Hospedagem Web Estática	**AWS Amplify:** 1.000 minutos de build por mês, 5 GB de armazenamento e 15 GB de transferência de dados por mês.	**Azure Static Web Apps:** 0,5 GB de armazenamento por aplicação e 100 GB de transferência de dados por mês.

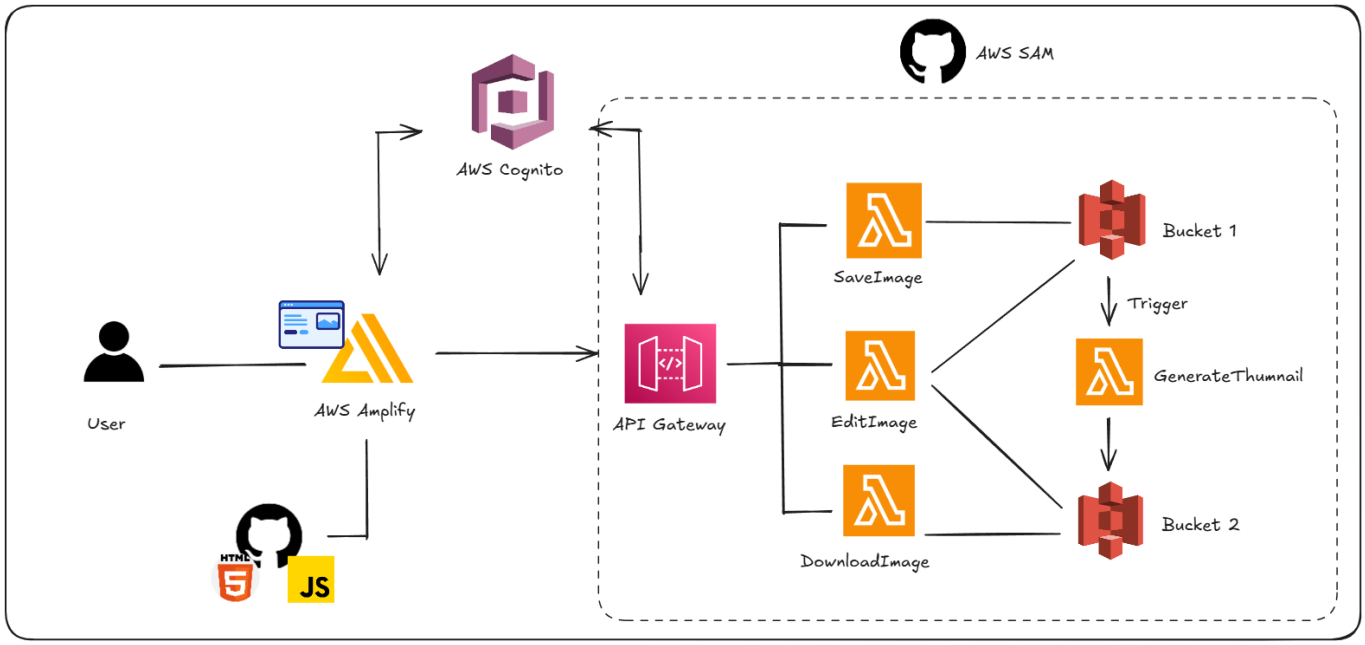

Figura 1. Esquema da aplicação implementada na AWS

está relacionada a 4 fatores principais: latência, custo, disponibilidade de serviços e termos de contrato. A proximidade da região com os usuários finais reduz a latência, enquanto manter todos os serviços em uma única região, como feito neste trabalho, minimiza os custos de transferência de dados, que geralmente são cobrados quando há tráfego entre regiões distintas.

Além disso, nem todos os serviços, especialmente os mais recentes, estão disponíveis em todas as regiões. Optou-se por neste trabalho a implementar a aplicação em uma única região em ambos provedores. Na AWS, a região escolhida foi a *us-east-1*, por ser a principal região e todos os principais recursos estão disponíveis no plano gratuito. Na Azure a principal região é chamada de *East US*, e foi a escolhida.

B. Desenvolvimento e implementação na AWS

Após a criação da conta na AWS, o usuário tem acesso total aos serviços do provedor por meio do Console da AWS acessado pelo navegador. Para interagir programaticamente com os serviços da AWS, é necessário a instalação do AWS CLI. Esta ferramenta é a forma como um desenvolvedores e ferramentas de automação executam as operações na nuvem por meio de comandos.

Para a devida utilização do AWS CLI é necessário credenciais associadas a uma *IAM Role*. O IAM (*Identity and Access Management*) é o serviço central da AWS para controle de acesso, funcionando como o ponto central de segurança para toda a plataforma. É importante ressaltar que esta lógica de permissões não se aplica apenas a usuários, mas também à

comunicação entre os próprios serviços. Por exemplo, uma função Lambda necessita de uma Role com políticas definidas para poder ler ou escrever em um bucket S3. Um outro ponto muito destacado nas documentações da AWS, diz respeito à política de menor privilégio: deve-se relacionar o mínimo de permissões aos usuários e aos serviços, contendo somente as políticas estritamente necessárias.

Com a utilização do framework AWS SAM para desenvolvimento da aplicação (explicado a seguir), o desenvolvedor não necessita criar manualmente cada um dos cargos e políticas, mas é importante entender o seu funcionamento caso problemas de permissão ocorram, comum em aplicações serverless.

Para se obter a devida familiarização com todo o ecossistema AWS, inicialmente um protótipo da aplicação de geração automática de miniaturas de imagem foi implementado baseando-se nesta demo fornecida pela AWS [3]. Foi utilizado a ferramenta *Localstack*, que fornece um ambiente de nuvem local que simula a resposta da APIs do AWS CLI por meio de contêineres Docker que são executados localmente. Embora seja uma ferramenta mantida por terceiros e não seja uma simulação exata do ambiente real de produção da AWS, mostrou-se de grande utilidade para familiarização e entender melhor como funcionam os comandos da AWS CLI, como os serviços funcionam, e como implementar uma aplicação serverless componente por componente e como comunicam entre si.

Após isso, um esquema de aplicação serverless de pós-processamento de imagens foi desenhado. O repositório contendo todo o código está disponível em https://github.com/LucasCarvalho01/serverless-aws. A figura 1 apresenta como a aplicação foi construída de forma integral. Para se ter uma aplicação completa, funcional e segura, uma série de serviços foram utilizados.

Foram necessárias 4 funções serverless, as quais no contexto da AWS são nomeadas de funções Lambda. A responsabilidade e funcionamento de cada uma das funções são descritas a seguir:

- **SaveImage:** lambda executada a partir de um gatilho HTTP, responsável por gerar uma URL pré-assinada no bucket primário. Retorna esta URL a qual o usuário pode utilizar para realizar uma requisição direta ao backet para salvar sua imagem.
- **GenerateThumbnail:** lambda executada a partir de um gatilho S3, ou seja, responde ao evento de inserção de um objeto no bucket primário. A função recupera a imagem recém inserida e a redimensiona em uma proporção de miniatura e a armazena em um bucket secundário.
- **EditImage:** lambda executada por meio de um gatilho HTTP, a qual recebe como parâmetro a chave da imagem armazenada no bucket primário, e um objeto descrevendo as propriedades a serem editadas na imagem.
- **DownloadImage:** lambda também executada a partir de um gatilho HTTP, e que gera uma URL pré-assinada para o bucket secundário, permitindo que o usuário faça o download da imagem pós-processada.

[3]https://docs.aws.amazon.com/lambda/latest/dg/with-s3-tutorial.html

Para se armazenar as imagens, é necessário a utilização de duas unidades de armazenamento de objetos. Para isso a AWS fornece o serviço Amazon S3, no qual denomina cada unidade de armazenamento como um *bucket*. Um serviço que se integra facilmente com outros. Como visto na subseção anterior, trata-se também de um serviço cobrado por uso. No contexto do trabalho, permite um armazenamento de uma quantidade considerável de imagens, principalmente em formatos otimizados, como JPG e Webp. O ponto a se levar em consideração é o limite de requisições mensais de armazenamento e listagem. Algo não especificado e claro pela AWS é que ao se utilizar o Console AWS para visualizar a listagem de imagens, também é contabilizado.

Como centralizador e componente essencial na segurança da aplicação, foi utilizado o HTTP API Gateway, um serviço totalmente gerenciado pela AWS e atua como o ponto de entrada para que usuários e outros serviços acessem as ferramentas back-end utilizadas, no caso as funções Lambda e buckets S3.

Como são fornecidos mais de um tipo de API Gateway, a escolha pela opção adequada foi um processo relevante para o trabalho. A seguir estão listados os tipos de API Gateway fornecidos pela AWS e suas principais características:

- HTTP API Gateway: versão com menos complexidade; menores custos (quando ultrapassado os limites do nível gratuito); consegue se integrar facilmente com funções Lambda; fornece autorização via JWT; latência menor
- REST API Gateway: versão com mais funcionalidades e mais complexa. Permite validação de requisições assim como a transformação do corpo de requisições e respostas; funcionalidade de cache integrada; controle granular de cada estágio de desenvolvimento (dev, staging, prod, etc); múltiplos mecanismos de autenticação
- Websocket API Gateway: Voltado para aplicações que funcionam em cima do protocolo Websocket, ou seja, aplicações em tempo real que precisam de comunicação bidirecional, fora do escopo deste trabalho

Diante das opções, optou-se pelo HTTP API Gateway que oferece o balanço ideal entre complexidade, custo e performance em relação ao escopo da aplicação desenvolvida. Sua menor complexidade de configuração acelerou o desenvolvimento e entendimento de como funciona, além de que o suporte nativo à autorização via JWT e a integração simplificada com funções Lambda atendem os requisitos do projeto.

Foi adotada a abordagem de desenvolvimento de Infraestrutura como Código (IaC). Em vez de configurar manualmente cada serviço por meio do AWS CLI ou Console, um processo propenso a erros e de difícil replicação, a abordagem de Infraestrutura como Código permite que toda a arquitetura da aplicação (funções, gatilhos, permissões e gateway), seja definida em um arquivo de código, facilitando principalmente o versionamento e deploys de atualização/correção. A medida em que a complexidade e tamanho das aplicações crescem, manter o padrão de desenvolvimento e deploy manual, componente por componente torna-se uma tarefa com alto custo de tempo e propensa a falhas, identificado logo cedo ao tentar

implementar desta forma por meio do *Localstack*.

A AWS fornece e incentiva o uso do framework *Serverless Application Model*, popularmente referido como AWS SAM. Trata-se de uma extensão do AWS CloudFormation com uma sintaxe mais simplificada e focada em aplicações serverless. O AWS CloudFormation por sua vez é um serviço genérico de IaC voltado para o ecossistema da AWS que permite providenciar e configurar de forma declarativa e por código, qualquer recurso da AWS. O AWS SAM é focado em aplicações integralmente serverless, dessa forma abstrai boa parte da complexidade e verbosidade que existe no CloudFormation.

Além da sintaxe simplificada, o AWS SAM se destaca por fornecer um conjunto de ferramentas por meio do *sam-cli*, que auxiliam consideravelmente no processo de desenvolvimento. Dentre as principais vantagens e facilidades, destacam-se:

- Desenvolvimento e teste local: a capacidade de testar a aplicação localmente, sem a necessidade de um deploy na nuvem para cada alteração. Comandos como `sam local start-api` emulam o API Gateway e as funções Lambda localmente, permitindo a execução dos endpoints HTTP e debug de forma local
- Validação de templates: antes de iniciar um processo de deploy, que pode ser demorado, o comando `sam validate` verifica a sintaxe do arquivo `template.yaml`, capturando erros de formatação e estrutura antecipadamente
- Build automatizado de dependências: O SAM simplifica o empacotamento de dependências. O comando `sam build` analisa arquivos que informam as dependências da aplicação, como o `requirements.txt` e faz o download e os salva de acordo com a estrutura de pastas que o runtime da AWS espera. Para bibliotecas com dependências nativas, como a *Pillow* utilizada neste trabalho, a opção `--use-container` constrói o pacote dentro de um ambiente Docker que espelha o runtime em que uma Lambda é executada, garantindo a compatibilidade.
- Deploy inicial guiado: Para o primeiro deploy, o comando `sam deploy --guided` guia o usuário na configuração da aplicação (nome, região, etc.) e salva essas informações no arquivo `samconfig.toml`. Os deploys seguintes seguem as configurações salvas, facilitando na integração com pipeline CI/CD.

Apesar de diversas vantagens, a utilização do AWS SAM também apresenta desafios e considerações que devem ser levadas em consideração:

- Vendor Lock-in: Embora seja uma ferramenta de código aberto, o SAM gera templates para o AWS CloudFormation. A sintaxe simplificada (ex: *AWS::Serverless::Function*) e a estrutura do projeto são específicas para a AWS, dificultando a portabilidade para outro provedor de nuvem
- Limitações da simulação Local: O ambiente local que o SAM fornece é uma simulação, não uma cópia idêntica do ambiente da nuvem. Ele não replica o comportamento por exemplo, da latência real da rede, ou possíveis problemas de comunicação entre serviços. Testes locais são úteis para testar a lógica da aplicação, mas testes no ambiente de nuvem real continuam sendo necessários
- Complexidade para recursos não-serverless: O SAM é focado para aplicações serverless. Se a aplicação no futuro necessitar de componentes não-serverless (como uma instância EC2, ou um banco de dados relacional), a sintaxe do template precisa seguir o padrão do CloudFormation, diminuindo os benefícios da abstração do SAM

Levando em consideração as capacidades e limitações do SAM, na listagem 1 a seguir, é possível ver um trecho do arquivo *template.yaml* [4]. Trata-se do arquivo principal da aplicação e que define os recursos necessários e como irão se comunicar:

```
Resources:
  BucketReceivedImages:
      Type: AWS::S3::Bucket
      Properties:
        BucketName: bucket-received-images
        CorsConfiguration:
        {...}

  ImageProcessorHttpApi:
    Type: AWS::Serverless::HttpApi
    Properties:
      Auth:
        Authorizers:
          JwtAuthorizer:
            IdentitySource: "$request.header.
                Authorization"
            JwtConfiguration:
            {...}

  SaveImageFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: save_image/
      Handler: app.lambda_handler
      Policies:
        - S3ReadPolicy:
            BucketName: bucket-received-images
```

Listing 1. Trecho do arquivo template.yaml

Neste trecho é possível ver, por meio do uso do SAM, a simplificação do processo de criação dos componentes necessários para aplicação, assim como a configuração da comunicação entre eles, no qual as políticas necessárias são declaradas no mesmo arquivo. Ao mesmo tempo em que é possível ver o quão acoplado é ao ecossistema AWS. Por exemplo, *BucketReceivedImages* (um bucket S3) e a *SaveImageFunction* (uma função Lambda) são definidos com poucas linhas de código. A comunicação e as permissões entre eles são estabelecidas de forma explícita na mesma declaração; a política *S3ReadPolicy* dentro da definição da função, define a permissão necessária de forma clara e localizada. Da mesma forma, a configuração de autenticação do HTTP API Gateway, também é integrada juntamente da sua definição, centralizando toda a configuração em um único código.

[4]template.yaml disponível integralmente em https://github.com/LucasCarvalho01/serverless-aws/blob/main/template.yaml

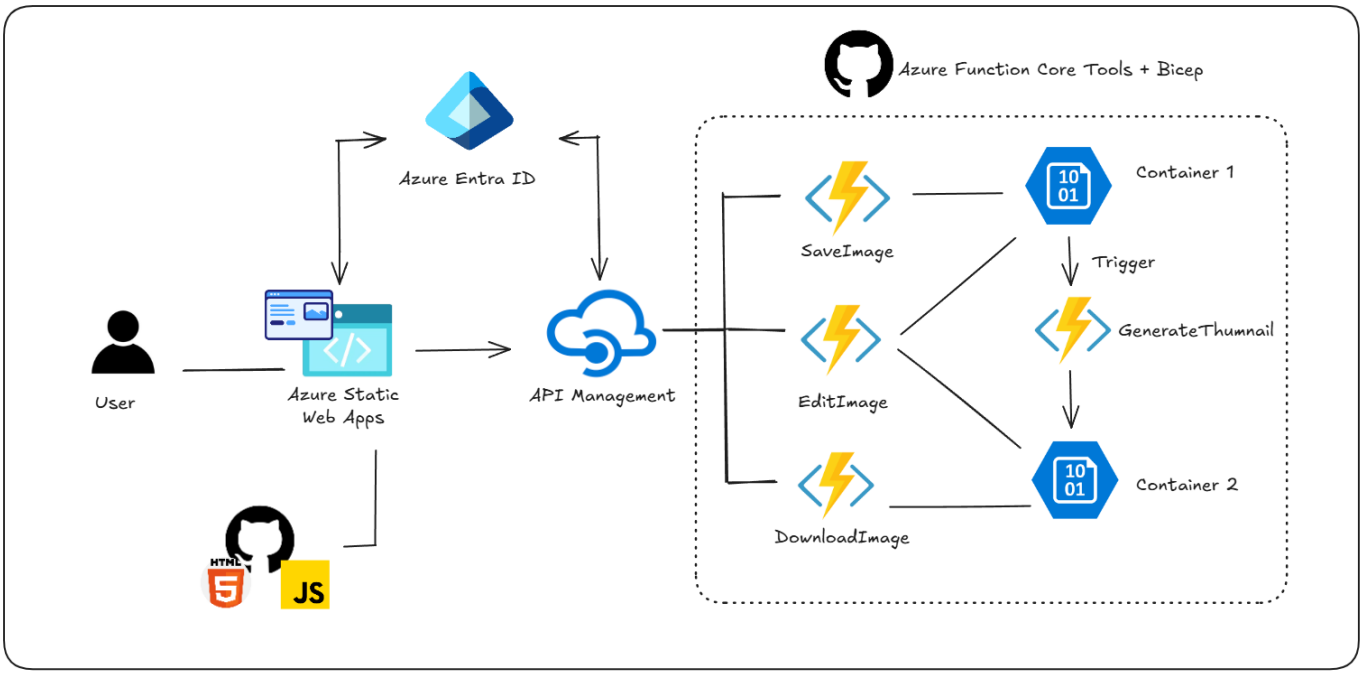

Figura 2. Esquema da aplicação implementada na Azure

A respeito da observabilidade, o serviço AWS Cloud Watch possibilita os usuários verificarem os logs da aplicação de forma simples. Por padrão, todas saídas (incluindo erros e comandos de print) de uma função Lambda é automaticamente transmitida para o CloudWatch Logs, sem a necessidade de configurar bibliotecas. Os logs são organizados em grupos de logs, onde cada função Lambda possui seu próprio grupo, facilitando a localização e o isolamento de registros, muito útil e de ajuda em momentos de debug.

Para se adequar às políticas de DevOps, foi implementado uma pipeline CI/CD via GitHub Actions. Foi configurado um workflow que é acionado automaticamente a cada commit realizado na branch main. Essa pipeline executa uma série de passos de forma automatizada:

1) Autenticação: O ambiente de execução do GitHub Actions autentica-se na AWS por meio de chave de acesso. Foi criado um usuário específico no IAM, seguindo a política de menor privilégio, que possibilita a ferramenta a realizar o deploy
2) Build: Realiza o build levando em consideração todas as dependências como explicado anteriormente
3) Deploy: Em caso de sucesso do build, o comando `sam deploy` é executado, aplicando as mudanças definidas no arquivo *template.yaml*

C. Desenvolvimento e implementação na Azure

Ao se criar uma conta, e vincular com plano de estudante, uma assinatura é vinculada a conta. É válido destacar que a Azure opta por uma organização de agrupamento de recursos. Como citado anteriormente, existe o primeiro nível lógico de separação entre assinaturas. Para se criar novos recursos, primeiramente é necessário criar um *Resource Group* (grupo de recursos), um outro contêiner lógico em que todo serviço precisa estar associado. Uma camada de agregação e de gerenciamento que a Azure fornece, permitindo filtrar acessos dos usuários a recursos em grupos específicos, além de facilitar a visualização e organização de custos. Está um nível abaixo da assinatura, no qual uma assinatura pode possuir mútliplos grupos de recurso. É também mais uma camada de administração, por exemplo, é possível fazer locks impedindo que usuários alterem ou deletem recursos, mover um grupo de recurso para outra assinatura, visualizar métricas a nível do grupo e associar tags.

Similarmente ao AWS CLI, existe o Azure CLI, ferramenta de linha de comando que permite o usuário realizar comandos e consultar informações de forma rápida e por meio de seu terminal. Um ponto relevante diz respeito a interface gráfica e organização dos elementos no Portal Azure.

Criou-se também um esquema ilustrando a aplicação, os recursos necessários e como se relacionam, como é possível ver pela figura 2. O repositório contendo todo o código está disponível em https://github.com/LucasCarvalho01/serverless-azure.

Assim como na AWS, utilizou-se também de uma estratégia de desenvolvimento voltada à Infraestrutura como Código, porém não existe uma ferramenta identicamente equivalente ao AWS SAM para a Azure.

A ferramenta *Azure Bicep* trata-se de uma linguagem proprietária e mantida pela Microsoft usada para descrever de forma declarativa o ambiente de desenvolvimento na nuvem. É compilada para *Azure Resource Manager*, arquivos que descrevem os recursos, dessa forma é possível definir quais recursos são necessários, configurá-los, descrever como funciona a comunicação entre si, dentre outros. Porém, possui uma

sintaxe consideravelmente e uma curva de aprendizado mais acentuada.

Diante desse fato, os recursos da aplicação foram provisionados inicialmente através do Portal Azure, que oferece um fluxo de criação guiado e detalhado. Após a criação e deploy inicial das funções e dos demais recursos, Uma funcionalidade poderosa que a Azure fornece é a capacidade de, após a criação de recursos, exportar toda configuração como um template Bicep. Com o template gerado, embora completo, reforçou a percepção da verbosidade do Bicep comparado ao YAML do AWS SAM, como será discutido na seção de análise comparativa.

A respeito das funções serverless, foram implementadas também um total de 4 *Azure Functions*. Possuem responsabilidades e funcionamentos descritos a seguir:

- **GetImageUrl**: função responsável por gerar uma URL SAS (assinatura de acesso compartilhado) ao contêiner primário. A partir dessa URL, o usuário pode realizar uma requisição de escrita direta ao contêiner para salvar sua imagem
- **CreateThumbnail**: função executada a partir de um gatilho blob, ou seja, responde ao evento de armazenamento de uma imagem no contêiner primário, a redimensiona e armazena no bucket secundário
- **EditImage**: responde a um evento de requisição HTTP para edição de uma imagem armazenada pela aplicação
- **DownloadImage**: também responde a um evento de requisição HTTP para gerar uma URL SAS, permitindo que o usuário faça o download da imagem pós-processada pela aplicação.

É possível ver no repositório que cada função está localizada dentro do diretório com seu nome, e em cada diretório existem 2 arquivos: `__init__.py`, arquivo python que contém o código da função; `function.json`, arquivo que descreve suas configurações de forma declarativa, aqui onde ficam diversas configurações, como bindings de entrada e saída, tipo de trigger (HTTP, timer, blob storage), métodos HTTP aceitos, nível de autorização, e outras propriedades específicas da função. As funções que respondem a eventos de requisição HTTP possuem o arquivo `function.json` semelhante, é descrito qual a rota e método HTTP a função deve responder e os possíveis parâmetros de entrada e saída. Para a função `CreateThumbnail`, o arquivo de configuração descreve o tipo do gatilho que a executa (no caso blob), qual o caminho do contêiner que deve ser monitorado e qual a conexão com o armazenamento.

Além disso, outro arquivo relevante trata-se do `host.json`. Neste arquivo ficam as configurações globais das funções. Um ponto de atenção e que foi aplicado no projeto trata-se da configuração de telemetria. Os logs são enviados de forma estruturada e contínua para o serviço *Application Insights*, dessa forma, custos e uma criação excessiva de logs podem ocorrer caso a configuração padrão seja mantida.

O arquivo `requirements.txt` também está presente na raiz, listando todas as dependências Python necessárias para o projeto

Relacionado às funções, novamente existe uma lógica de agrupamento. É necessário criar uma *Function App*, o contêiner lógico que mantém as funções relacionadas e a configuração geral agrupados. Esta ferramenta gerencia todo o ciclo de vida da Function App, desde a criação de projetos baseados em templates até a publicação na nuvem. Em conjunto com o emulador *Azurite*, que simula os serviços de Armazenamento do Azure localmente, foi possível simular um ambiente de desenvolvimento local bem semelhante ao de produção.

Em relação às dependências, a Azure fornece uma opção de build de forma remota no momento do deploy, utilizando um serviço chamado Oryx. Isso abstrai a complexidade do empacotamento de bibliotecas com código nativo, como a Pillow, ponto auxiliar principalmente ao se implementar um fluxo de deploy automatizado, como foi o caso utilizando o GitHub Actions, explicado mais a frente. Entretanto por ser uma camada de abstração do build, em caso de falha, nem sempre é simples de se entender o motivo.

O serviço de armazenamento de objetos na Azure é chamado de *Azure Blob Storage*. Mais uma vez, seguindo a política de divisão e agrupamento da Azure, todos os serviços de armazenamento relacionados precisam de um *Storage Account*. Por meio desse contêiner lógico é possível agrupar os recursos relacionados à armazenamento na Azure, e os que foram utilizados no desenvolvimento da aplicação foram: *container blob* (para armazenamento não estruturado de objetos, referidos como blobs), *tables* (para armazenamento de dados não relacionais) e *queues* (filas). Embora explicitamente a aplicação necessite somente dos contêineres, o funcionamento e lógica por trás do funcionamento de uma Function App demanda estes outros serviços de armazenamento, que também consomem do limite mensal de escrita e leitura.

As *tables* são utilizadas pela Function App para armazenar os logs provenientes de telemetria. As *queues* são necessárias para o controle do fluxo dos eventos - quando um evento configurado como trigger de uma função ocorre, seja uma requisição HTTP ou armazenamento de uma imagem no contêiner, a Function App utiliza uma fila para armazenar estes eventos e os responder em sequência, útil para o caso em que muitos eventos ocorram em um curto intervalo de tempo.

Por padrão, as Azure Functions se integram ao *Application Insights*. Ao invés de serem tratados como texto, os logs são gerados como dados de telemetria estruturados (requisições, exceções, traces). Trata-se de uma ferramenta muito completa e poderosa para análise de performance.

Além disso, ao se configurar uma função com o gatilho do tipo *blob trigger* (o caso da função de geração de thumbnail), para saber se um novo blob foi adicionado, o runtime da Function App precisa escanear o contêiner periodicamente, o que conta como uma operação de listagem. Ao manter a Function App habilitada, mesmo que não haja requisições ou novas imagens sendo salvas nos contêineres, as operações de listagem ainda ocorrem. Caso a aplicação seja mantida habilitada por muitas horas, o limite gratuito de listagem pode

ser atingido, o que foi o caso durante o desenvolvimento do trabalho, mas graças ao crédito de 100 dólares, nenhum custo financeiro real ocorreu. Após o acontecimento deste custo ocioso, uma pesquisa foi feita para o real entendimento do acontecimento, e após verificar um outro usuário com o mesmo problema por uma postagem no StackOverflow [5], foi encontrada uma página na documentação [6] a repeito do funcionamento interno de uma função com um blob trigger configurado.

A respeito do gateway, foi utilizado o serviço *API Management*. Seu funcionamento é bem semelhante ao API Gateway da AWS citado anteriormente. É totalmente gerenciado pela Azure e atua como centralizador da aplicação. A Azure diferencia os diferentes tipos de gateways que fornece em *tiers* (categorias). São ofertadas 8 categorias diferentes, com diferentes limites e preços, entretanto a única categoria que se enquadra no plano gratuito é o *Comsumption Tier*. Diferente dos demais, não existe uma cobrança fixa mensal, é cobrado de acordo com o uso, mas é o modelo com menos funcionalidades. É possível ver em detalhes as diversas opções de forma detalhada no Portal da Azure. [7]. Embora seja a categoria mais simples, o API Management no *Comsumption Tier* forneceu todas as funcionalidades necessárias, em especial integração simples com as funções serverless e contêineres blobs, além de possibilitar a configuração de uma validação JWT, essencial para a segurança da aplicação seguindo uma estratégia semelhante a AWS. Por meio do Portal, é possível facilmente adicionar uma política de validação de token JWT, e após a geração do arquivo bicep descrevendo a infraestrutura, é possível configurá-lo também de forma declarativa.

Com todos os serviços criados e configurados, pôde-se ter acesso ao arquivo bicep descrevendo todo a infraestrutura. Um trecho do arquivo disponível integralmente no repositório [8], é apresentado a seguir na listagem 2

```
param appName string = 'imgproc'
param location string = 'eastus'

resource functionApp 'Microsoft.Web/sites@2023
    -12-01' = {
  name: functionAppName
  location: location
  kind: 'functionapp,linux'
  identity: {
    type: 'SystemAssigned'
  }
  properties: { siteConfig: {
      linuxFxVersion: 'Python|3.11'
      {...}

resource apiManagement 'Microsoft.
    ApiManagement/service@2023-05-01-preview'
    = {
  name: apiManagementName
  location: location
  sku: {
    name: 'Consumption'
    {...}

resource apimPolicy 'Microsoft.ApiManagement/
    service/apis/policies@2023-05-01-preview'
    = {
  parent: apimApi
  name: 'policy'
  properties: {
    format: 'xml'
    value: '''
        <inbound>
        <base />
        <cors allow-credentials="true">
            <allowed-origins>
            {...}
```

Listing 2. Trecho do arquivo main.bicep

Por este trecho, pode-se ver que a ferramenta proprietária da Azure possui uma sintaxe mais verbosa e com maior curva de aprendizagem, quando comparado com o SAM CLI.

A aplicação implementada na Azure também também foi integrada a um pipeline de CI/CD utilizando GitHubActions. Foram configurados dois worklows. O arquivo `deploy.yml` descreve um workflow para deploy da Function App - quando um commit é feito para branch main, a pipeline é executada e atualiza-se automaticamente a aplicação, seja alguma alteração no código de alguma função, ou alguma configuração. Há também o arquivo `deploy-infra.yml`, um workflow dedicado à infraestrutura, o qual verifica por mudanças feitas no arquivo `main.bicep`, e quando ocorre, aplica tais alterações nos recursos.

D. Análise Comparativa

Após o desenvolvimento da mesma aplicação em ambos provedores, foi possível identificar e analisar características em comum entre AWS e Azure, assim como identificar pontos fortes e de destaque de um provedor em relação ao outro.

Um ponto de destaque descrito nas seções anteriores trata-se das ferramentas fornecidas para o desenvolvimento seguindo a prática de Infraestrutura como Código. O AWS SAM é uma ferramenta madura, criada como abstração do AWS CloudFormation. Fornece uma sintaxe simples, curva de aprendizado baixa e muitos exemplos publicados tanto pela AWS quanto pela comunidade. Por outro lado, a Microsoft fornece o Bicep, uma linguagem robusta e proprietária desenvolvida para descrição dos recursos, porém com uma sintaxe muito mais complexa e curva de aprendizado maior.

A fim de se obter uma melhor visualização da diferença do código entre os dois provedores, é possível visualizar trechos dos arquivos que contêm a função responsável por aplicar as edições. Primeiramente para Azure, na listagem 3 a seguir:

```
from azure.storage.blob import
    BlobServiceClient
import azure.functions as func

SOURCE_CONTAINER = os.environ.get('
    SOURCE_CONTAINER', 'received-images')
```

[5]https://stackoverflow.com/questions/72926339/what-is-lrs-list-and-create-container-operations

[6]https://learn.microsoft.com/en-us/azure/azure-functions/functions-bindings-storage-blob-trigger

[7]https://azure.microsoft.com/en-us/pricing/details/api-management/

[8]https://github.com/LucasCarvalho01/serverless-azure/blob/main/bicep/main.bicep

```
DEST_CONTAINER = os.environ.get('
    DEST_CONTAINER', 'dest-images')
CONNECTION_STRING = os.environ['
    AzureWebJobsStorage']
blob_service_client = BlobServiceClient.
    from_connection_string(CONNECTION_STRING)

def apply_image_edits(image, edits):
    if "brightness" in edits:
        enhancer = ImageEnhance.Brightness(
            image)
        image = enhancer.enhance(1.0 + edits["
            brightness"] / 100.0)
    {...}

def get_image_from_container(container,
    blob_name):
    try:
        blob_client = blob_service_client.
            get_blob_client(container=
            container, blob=blob_name)
        stream = blob_client.download_blob()
        image_data = stream.readall()

        img = Image.open(BytesIO(image_data))
        {...}

def main(req: func.HttpRequest) -> func.
    HttpResponse:
    try:
        req_body = req.get_json()
        image_key = req_body.get('key')
        edit_properties = req_body.get('
            edit_properties')

        if not image_key:
            return func.HttpResponse(
                "Parâmetro 'key' é obrigatório
                    ",
                status_code=400
            )
        {...}
```

Listing 3. Trecho de Função Azure que aplica edições em um blob

A listagem 4 a seguir é possível visualizar trecho do arquivo que contém a para aplicar as edições a uma função na AWS:

```
import boto3

s3_client = boto3.client('s3')
SOURCE_BUCKET = os.environ.get('SOURCE_BUCKET'
    , 'bucket-received-images')
DESTINATION_BUCKET = os.environ.get('
    DESTINATION_BUCKET', 'bucket-dest-images')

def apply_image_edits(image, edits):
    if "brightness" in edits:
        enhancer = ImageEnhance.Brightness(
            image)
        image = enhancer.enhance(1.0 + edits["
            brightness"] / 100.0)
    {...}

def get_image_from_s3(bucket, key):
    try:
        response = s3_client.get_object(Bucket
            =bucket, Key=key)
        image_data = response['Body'].read()

        img = Image.open(BytesIO(image_data))
        {...}

def lambda_handler(event, context):
    try:
        body = event.get('body')
        if body:
            body = json.loads(body)
        else:
            body = event

        image_key = body.get('key')
        edit_properties = body.get('
            edit_properties')

        if not image_key:
            return {
                'statusCode': 400,
                'body': json.dumps({'error': '
                    Parâmetro 'key' é obrigató
                    rio'})
            }
        {...}
```

Listing 4. Trecho de Função Lambda que aplica edições em um objeto

Mesmo se tratando de uma função com um funcionamento simples e com uma lógica de negócio sem muita complexidade, foi possível identificar diferenças consideráveis entre ambos provedores. No caso da função Azure, utiliza-se a biblioteca azure.functions e azure.storage.blob para auxiliar no desenvolvimento da função e acesso aos contêineres de armazenamento, respectivamente; por convenção a função é nomeada como *main*; o parâmetro recebido pela função é a própria requisição HTTP; a sintaxe para conexão e comunicação relacionado ao contêiner possuem um certo padrão; o retorno é um objeto do tipo *HttpResponse*. Além disso, a Azure Function requer a definição de bindings no arquivo *function.json*, que especifica como a função interage com triggers e outros serviços, criando uma camada adicional de configuração específica do provedor. Ao se comparar com o código da função Lambda, percebe-se que uma série de alterações são necessárias - a SDK proprietária para auxílio do desenvolvimento da função e integração com o S3 obviamente é outra, nesse caso o *boto3*; segue um padrão de nomenclatura de *lambda_handler*; os parâmetros recebidos são diferentes; a sintaxe de conexão e comunicação com bucket S3 tem seu próprio padrão; e a função retorna um dicionário Python.

É possível ver que o vendor lock-in está muito relacionado à integração e ecossistema das aplicações serverless. Por serem compostos por pequenas funções, que têm como responsabilidade uma pequena tarefa específica, a lógica de negócio de cada função tende a ser simples, sendo o maior responsável pelo lock-in a integração e utilização de outros serviços do provedor de nuvem. Além disso, os mecanismos de trigger também têm diferenças significativas: a AWS oferece integração direta com uma diversos nativamente, enquanto a Azure depende de bindings específicos configurados declarativamente.

Outro ponto de diferenciação entre os dois provedores diz respeito na organização interna dos serviços. Como mencionado na subseção anterior, a Azure aplica a política de agrupa-

mento em diversos momentos. Desde a lógica inicial de divisão por assinaturas, necessidade de agrupar recursos por meio de um *resource group*, após isso a criação dos contêineres lógicos, como a Function App e Storage Account. Promove uma maior burocracia e complexidade inicial, porém fornece um controle melhor da criação dos recursos e visualização de custos. Por outro lado a AWS segue uma filosofia mais independente em relação aos recursos criados. Pode-se facilmente criar uma função Lambda, criar um trigger HTTP a partir de um API Gateway, conter um código que se comunica com um Bucket de outro projeto, uma grande liberdade fornecida aos desenvolvedores, porém é necessário maior disciplina para se manter a devida organização das aplicações.

Comparando os dois provedores levando em consideração a observabilidade, a AWS se destaca. O serviço da AWS, CloudWatch fornece uma experiência bem mais simples e direta para se visualizar os logs das aplicações. Os logs seguem um padrão textual, e é possível filtar os logs a nível de grupo, em que cada serviço é considerado um grupo, ou seja, para se acessar os logs de uma função lambda específica, basta filtrar pel seu grupo. A interface do CloudWatch apresenta uma navegação intuitiva, onde os desenvolvedores podem rapidamente localizar informações específicas através de filtros por timestamp, níveis de log ou palavras-chave, facilitando significativamente o processo de debugging e monitoramento.

O Application Insights da Azure por sua vez, tem como característica a robustez, e consequentemente, de maior dificuldade de uso. A tarefa de simplesmente visualizar o log de uma execução específica não é tão simples e direta como no AWS CloudWatch. A ferramenta oferece recursos avançados de análise e correlação de dados, porém essa robustez vem acompanhada de uma curva de aprendizado maior. No caso de um simples erro de passagem de parâmetro enfrentado, identificar o log exato da função com problema se mostrou uma grande dificuldade.

E a respeito do gerenciamento de dependências, como citado nas subseções anteriores, em ambos provedores foi possível a utilização da biblioteca Pillow nas funções que dependiam da biblioteca. Entretanto, novamente a AWS forneceu um sistema de melhor usabilidade. Como o AWS SAM permite realizar o build dentro de um contêiner que simula o runtime em que a Lambda é executada, promove um processo de build facilitado e transparente, onde as dependências nativas são automaticamente compiladas para o ambiente de destino sem necessidade de configurações adicionais. Já por outro lado, para realizar o build remoto na Azure, é necessário utilizar o serviço Oryx, que embora eficiente, apresenta uma maior complexidade na resolução de conflitos de dependências nativas. No caso foi enfrentado dificuldades com uma outra biblioteca de manipulação de imagens, e foi necessário optar pela Pillow por este motivo.

E. Aplicação front-end

Para que os usuários consigam acessar e realizar as requisições desejadas ao serviços de pós-processamento de imagens, foi desenvolvido uma aplicação front-end acessada por meio do navegador, para cada provedor. Esta seção aborda em detalhes como ocorreu a implementação das aplicações front-end em ambos provedores, uma vez que se diferenciaram em diversos pontos.

Optou-se por seguir a implementação de uma *Single Page Application* (aplicação de página única), comumente referida como SPA. A Mozilla Developer Network [17] define uma SPA como uma aplicação web que é carregada em um único documento, e o conteúdo da página é atualizado de acordo com a interação do usuário. SPAs frequentemente são desenvolvidas utilizando grandes frameworks e bibliotecas como React e Angular, mas para este caso optou-se por desenvolver a aplicação sem dependências externas, exceto às bibliotecas de autenticação/autorização, para se evitar complexidade desnecessária e não encarecer e dificultar o processo de build e deploy.

Dessa forma, em ambos projetos, tem-se uma arquitetura e organização de arquivos semelhantes e de fácil entendimento:

- *index.html*: O arquivo que contém a estrutura HTML da página
- *style.css*: Arquivo que descreve a estilização da página
- *script.js*: Arquivo que contém o script que lida com autenticação, autorização, envio das requisições, validação das respostas e interação do usuário
- *build.js, config.template.js*: Arquivos referentes ao build da aplicação para se realizar o deploy e arquivo responsável por armazenar as propriedades de autenticação e autorização

Ambos os provedores fornecem serviços para hospedagem de aplicações front-end de forma simplificada e provisionada: para a AWS, o AWS Amplify; para a Azure, o Azure Static Web Apps.

Ambos serviços fornecem um mecanismo de build e deploy simples para o caso deste trabalho, automatizado e integrado com um repositório de código, que no caso foi optado pelo Github. O repositório contendo todo o código hospedado pelo AWS Amplify pode ser encontrado em https://github.com/LucasCarvalho01/front-aws. O código hospedado pelo Azure Static Web Apps pode ser encontrado em https://github.com/LucasCarvalho01/front-azure.

O código de ambos projetos se manteve bem semelhante, uma vez que a estrutura e estilização de ambos foi mantido, assim como a lógica de envio de requisições e tratamento das respostas. O grande ponto de diferença está relacionado ao fluxo de autenticação e autorização, detalhado na próxima subseção, mas que ocorreu devido ao uso de duas bibliotecas diferentes.

1) AWS Amplify: O AWS Amplify [9] é um serviço totalmente gerenciado que simplifica o processo de desenvolvimento, build, deploy e hospedagem de aplicações web. O Amplify oferece hospedagem estática com distribuição global via CDN (*Content Delivery Network*).

O processo de integração com repositórios versionados por meio do Git é nativo, direto e bem guiado pelo Console

[9]https://aws.amazon.com/amplify/

da AWS. Após conectar o repositório GitHub ao Amplify, o serviço monitora mudanças na branch configurada e executa automaticamente a pipeline de CI/CD. O arquivo *amplify.yml* define as etapas deste pipeline:

- **preBuild**: Comandos executados antes do build principal (útil para verificar que o ambiente Node.js está disponível)
- **build**: Execução do script de build da aplicação (no caso, *node build.js*)
- **postBuild**: Comandos pós-build para limpeza ou verificações
- **artifacts**: Especifica quais arquivos devem ser implantados

Um ponto a se levar em consideração é o gerenciamento seguro de variáveis de ambiente. O Amplify permite configurar dados sensíveis (como a URL API Gateway e identificadores de serviços) diretamente no console, evitando sua exposição no código-fonte. O script *build.js* gera dinamicamente o arquivo de configuração *config.js*, adicionando as variáveis de ambiente necessárias.

Relacionado ao fluxo de autenticação e autorização do projeto hospedado na AWS Amplify, a biblioteca utilizada foi a *oidc-client-js*. É recomendada e apresentada como exemplo introdutório nas documentações oficiais da AWS. Além disso, a documentação oficial da biblioteca [10] fornece informações detalhadas a respeito de cada passo para a devida configuração e utilização, e foi devidamente consultada.

2) Azure Static Web Apps: Similar ao Amplify, o Azure Static Web Apps [11] é definido na documentação como um serviço que implementa aplicações web de forma automatizada para a Azure a partir de um repositório. Também fornece os arquivos estáticos das aplicações por meio de CDNs, otimizando o tempo de resposta.

A integração neste caso foi configurada por meio do Github Actions. Durante a configuração inicial no portal Azure, que também é simples e bem guiada, é criado automaticamente um arquivo de workflow *azure-static-web-apps.yml* no repositório. Este workflow define o pipeline de CI/CD executado pelo Github Actions a medida que *commits* ou *pull-requests* são feitos para a branch principal. Os passos executados, descritos como *jobs* no arquivo, realizam as seguintes operações:

- **Checkout**: Baixa o código-fonte do repositório
- **Setup Node.js**: Configura o ambiente Node.js necessário para o build
- **Build and Deploy**: Executa o script de build e realiza a implementação dos arquivos resultantes

O gerenciamento de variáveis de ambiente segue uma abordagem similar ao AWS Amplify, porém utilizando GitHub Secrets. Os dados sensíveis são configurados como segredos no repositório e injetados durante o processo de build, usados na criação do arquivo *config.js*, processos descritos também por meio do script *build.js*.

[10]https://authts.github.io/oidc-client-ts

[11]https://azure.microsoft.com/products/app-service/static

Para autenticação e autorização, a aplicação desenvolvida para Azure utiliza a biblioteca *msal-browser* (Microsoft Authentication Library), que oferece integração nativa com o Microsoft Entra ID. Biblioteca esta, recomendada fortemente pela Microsoft ao se utilizar o Microsoft Entra ID como provedor de identidade, por ser mantida pela mesma organização, fornecendo uma configuração e integração entre os dois serviços de forma simplificada.

3) Observações: Ao optar por hospedar as aplicações em tais serviços com alto grau de automação e com provisionamento e configurações, em sua maior parte, sob responsabilidade dos provedores de nuvem, pôde-se verificar características de uma aplicação serverless voltada ao escopo de front-end.

Quesitos como configuração de servidores web, gerenciamento de certificados SSL e TLS, configuração de sistemas de backup, monitoramento de recursos computacionais e planejamento de capacidade foram abstraídos, promovendo um desenvolvimento e implementação mais ágil. Assim como no back-end do sistema, o foco ficou em maior parte voltado para a lógica de negócio, uma vez que ambos serviços oferecem uma integração e configuração simples. Além de outras vantagens típicas de uma aplicação serverless, como cobrança pelo uso efetivo, sem cobrança por recursos computacionais ociosos; distribuição de conteúdo otimizado para os usuários por meio das CDNs e facilidade em integrar outros serviços do provedor à aplicação.

Por outro lado, ao se optar por esta implementação, alguns desafios aparecem, tais como intermediário adicional no processo de build e deploy - é necessário agora levar em consideração que não é possível configurar diretamente o ambiente em que a aplicação é executada, além do vendor lock-in, que pelo escopo e complexidade das aplicações não tenha se tornado tão evidente, mas já foi possível identificar individualidades e pontos de alteração caso seja necessário a migração da hospedagem para um provedor de nuvem diferente.

Além disso, um ponto de alta consideração é o limite de requisições. Caso ocorra um ataque mal intencionado, como DDoS, o limite de requisições/largura de banda podem ser atingidos. Embora ambos provedores apliquem políticas padrões de segurança, usuários mal intencionados podem levar a um custo elevado com esse tipo de hospedagem auto provisionada.

Após detalhes de como as aplicações front-end foram desenvolvidas e implementadas, é necessário destaque para o fluxo de autenticação e autorização de todo o sistema, uma vez que é necessário filtrar e permitir somente usuários devidamente autorizados a acessarem a aplicação.

F. Autenticação e Autorização

A aplicação de pós-processamento de imagens pode ser acessada e utilizada tanto pelas aplicações front-end desenvolvidas quanto por meio de requisições HTTP diretas aos gateways (API Gateway e API Management). Dessa forma, é necessário a atenção para a segurança de todos os com-

ponentes. Em especial, a segurança de aplicações front-end acessadas pela web apresentam desafios únicos, pois todo o código da aplicação é executado em um ambiente público e não confiável: o navegador do usuário.

Em resposta a isso, é comum a utilização do protocolo de autorização OAuth 2.0 em conjunto com a camada de identidade OpenID Connect (OIDC). Essa combinação de protocolos de validação de requisições é agnóstica ao cliente, ou seja, a validação é aplicada independentemente se a requisição é proveniente de um navegador, ou de uma ferramenta de desenvolvimento como *Postman*, ou até mesmo de uma ferramenta de script como o *cURL*. O essencial é que a requisição siga as diretrizes dos protocolos: conter um *header* (cabeçalho) *Authorization: Bearer 'token'*, no qual *token* é um JWT válido, não expirado e emitido por uma autoridade de confiança.

Para um melhor entendimento do fluxo em cada provedor, a seguir é apresentando uma visão geral dos conceitos fundamentais do OAuth 2.0 e OIDC para autenticação e autorização de requisições. Embora o foco deste trabalho não seja aprofundar em questões de autenticação e autorização, esta discussão faz-se necessária para que os termos e procedimentos executados pelos provedores de nuvem no fluxo de validação das requisições sejam melhores compreendidos.

O processo se inicia com a autenticação, que é a verificação de identidade do usuário. Em ambas as aplicações, o usuário é redirecionado a uma página de login para se identificar, configurada pelo provedor. Esse fluxo é gerenciado pelo OpenID Connect (OIDC), uma camada de identidade construída sobre o OAuth 2.0. Após o sucesso da autenticação, o provedor de identidade (Amazon Cognito ou Azure Entra ID) gera um ID Token. Este token, em formato JWT, contém informações sobre o usuário, como nome e e-mail, confirmando que a autenticação ocorreu com sucesso.

Em seguida, ocorre a autorização, que é o processo de conceder permissão para que a aplicação acesse recursos específicos em nome do usuário já autenticado. O OAuth 2.0 é o protocolo que define os passos deste fluxo. O provedor de identidade gera um *access token* (token de acesso), também no formato JWT, que representa a autorização concedida à aplicação para acessar os recursos protegidos (neste caso, as funções de processamento de imagem). O front-end, então, anexa no cabeçalho este token a cada requisição enviada aos endpoints da API.

As informações são trocadas seguindo o formato de *Json Web Token* (JWT). Conforme a RFC 7519 [18] define, "JSON Web Token (JWT) é um formato compacto de representação de *claims* (informações/declarações) destinado a ambientes com restrição de espaço, como cabeçalhos de Autorização HTTP e parâmetros de consulta URI. JWTs codificam as *claims* a serem transmitidas como um objeto JSON". Nota-se ser o formato ideal para um cabeçalho de uma requisição e se tornou o padrão para troca de dados entre serviços e aplicações.

Ao se centralizar a validação do token nos gateways, a lógica de segurança permanece desacoplada do sistema. Ao invés de cada função serverless validar a autenticação de cada requisição, isso fica sob responsabilidade do gateway, e caso seja necessário alterar a lógica de validação, isso pode ser feito em um único ponto de alteração. Além disso, evita custos desnecessários, uma vez que a execução da função serverless só ocorre para requisições válidas.

Quando o gateway recebe uma requisição com um *access token*, em ambos os provedores, ele deve executar um processo de validação padronizado antes de redirecionar as requisições às funções serverless. Este processo consiste em quatro verificações principais:

- Validação da Expiração: O gateway irá conferir as *claims* referentes a *exp* (tempo de expiração) e *nbf* (não antes de) para garantir que o token está válido e que esteja ativo.
- Validação da Assinatura: Cada JWT é assinado pelo provedor de identidade (Cognito ou Entra ID) com uma chave privada. O gateway por meio da chave pública correspondente do provedor, verifica a integridade da assinatura do token, dessa forma caso a verificação seja bem-sucedida, o gateway tem a garantia de que o token foi de fato emitido pelo provedor de identidade e que seu conteúdo não foi alterado.
- Validação do Emissor: Verifica-se a *claim iss* para garantir que foi gerado pela autoridade correta (*User Pool* do Cognito e *Tenant* do Entra ID).
- Validação da Audiência: Confere-se também a *claim aud*, para garantir que o token foi gerado para esta aplicação específica (cliente). Impede que um token válido de outra aplicação seja reusado.

Um ponto importante para a segurança da aplicação é a gestão das propriedades de autenticação. Dados como identificadores do cliente, ID do tenant e URL de verificação de autoridade são sensíveis e não é recomendado expô-las em repositórios públicos. Por isso, em ambas as implementações, esses valores não são codificados diretamente nos arquivos. São usadas variáveis de ambiente que são injetadas no processo de build.

Uma diferença relevante entre os dois provedores trata-se da forma da estruturação dos serviços de autenticação e autorização (Amazon Cognito e Azure Entra ID), seus componentes principais (User Pool e Tenant) e características de configuração.

A AWS fornece o Amazon Cognito, serviço central para gestão de identidades. Nesta aplicação, foi utilizado o Cognito User Pool como provedor de identidade. Também é a entidade responsável pela emissão dos tokens JWT, ou seja, o *issuer*. Para que uma aplicação seja capaz de interagir com o Cognito, é necessário realizar o cadastro da aplicação.

A Azure fornece o serviço Entra ID, também responsável pela gestão de identidades, e o provedor de identidade é referido como tenant. Um tenant é como uma organização dentro de uma conta Azure, em que uma ou mais assinaturas podem estar associadas. Também é necessário realizar o registro da aplicação que irá realizar a autenticação por meio do Entra ID. Na Azure, também é obrigatório vincular para cada usuário que terá acesso à aplicação, uma *role*. Os cargos são

as permissões que o usuário possui dentro da aplicação, por exemplo, pode-se ter um usuário ”leitor”e outro ”admin”com mais privilégios. Neste caso foi configurado somente um cargo padrão para se assimilar à configuração feita na AWS.

A experiência do usuário e a lógica de implementação da autenticação no código se diferem entre as duas aplicações. Essas diferenças ocorrem pelas diferenças entre as bibliotecas escolhidas para orquestrar o fluxo de autenticação.

Como a Microsoft disponibiliza e mantém uma biblioteca própria, a *Microsoft Authentication Library* (msal.js), a implementação e configuração na aplicação ocorreu de forma mais simples. A biblioteca permite que seja aberto um pop-up de autenticação, e sendo bem sucedido, a biblioteca gerencia internamente a troca do código de autorização pelos tokens, abstraindo boa parte da lógica de todo o fluxo.

Para a aplicação na AWS, foi utilizada a biblioteca *oidc-client-js*, uma biblioteca agnóstica a provedores. São necessários algumas funções extras para o devido funcionamento, mas a lógica de funcionamento é bem semelhante a msal.js, com a diferença de ser mais generalista. Ocorre um redirecionamento para a página chamada de *Hosted UI* do Amazon Cognito, após o sucesso do login, o usuário é redirecionado para a aplicação, com a troca de tokens ocorrendo manualmente.

A seguir são apresentados detalhes a respeito da implementação da validação de autenticação e autorização em cada provedor.

1) Validação no API Management: No API Management a validação é feita de forma declarativa por meio de uma *policy*. É necessário adicionar uma política chamada *validate-jwt* à seção de processamento de requisições de entrada. Tudo isso é configurado por meio de uma sintaxe XML. Por se tratar de uma funcionalidade muito usada pelas aplicações modernas, a Microsoft fornece documentações e exemplos práticos de como realizar a devida configuração dos serviços relacionados, em especial esta demo [19] usada como base neste trabalho.

2) Validação no API Gateway: Para o API Gateway, a lógica é semelhante, mas aqui o recurso é chamado de *JWT Authorizer*, em que é vinculado a cada rota do API Gateway que se deseja proteger. O AWS SAM abstrai e simplifica muito essa configuração. São necessários somente dois dados: o *issuer* (a URL do Cognito User Pool) e a *audience* (o clientId da aplicação). Com estes dados, o API Gateway abstrai a complexidade restante, realizando automaticamente todas as quatro validações padrões (assinatura, expiração, emissor e audiência) e bloqueando requisições inválidas antes que cheguem às funções lambda.

3) CORS: Por fim, é necessário detalhar um ponto relevante na integração entre uma aplicação front-end e uma API hospedada em um domínio diferente. Embora não esteja diretamente relacionado ao escopo de autenticação e autorização, o gerenciamento da política de *Cross-Origin Resource Sharing* (CORS) é um ponto crucial. Por padrão, os navegadores implementam a *Same-Origin Policy*, que impede que um serviço em uma origem (domínio, protocolo ou porta) faça requisições para outra origem. Para que a aplicação front-end, hospedada no AWS Amplify ou no Azure Static Web Apps, consiga se comunicar com o API Gateway ou o API Management, foi necessário configurar uma política de CORS nos gateways.

Esta política consiste em adicionar cabeçalhos HTTP específicos na resposta do servidor da API ao receber uma requisição OPTIONS do front-end (requisição esta muitas vezes nomeada como *prefilght request*), feita para verificar as políticas de CORS. O navegador verifica a resposta dessa requisição, e permite ou não que a aplicação front-end faça requisições para o serviço localizado em outro domínio. Essa configuração garante que apenas clientes autorizados possam interagir com a API.

Os cabeçalhos *Access-Control-Allow-Origin*, *Access-Control-Allow-Headers*, *Access-Control-Allow-Methods*, *Access-Control-Expose-Headers* e *Access-Control-Allow-Credentials*, definem respectivamente: a partir de quais domínios as requisições são permitidas, quais cabeçalhos são permitidos nestas requisições, quais métodos HTTP são permtidos, quais cabeçalhos customizados a aplicação pode acessar, e se cabeçalhos relacionados a credencias podem ser trocados.

Aqui o princípio de menor privilégio é aplicado. Em ambos os projetos, os gateways foram configurados para permitir requisições vindas exclusivamente do domínio da sua respectiva aplicação front-end, assim como ter acesso somente aos cabeçalhos necessários, e ter permissão para executar somente os métodos HTTP necessários.

O AWS SAM novamente auxilia nesta configuração. Na Azure, semelhante à validação do JWT, é necessário adicionar uma política CORS, à seção de processamento de requisições de entrada.

IV. Ferramentas e boas práticas

Esta seção traz outro embasamento teórico ao trabalho. A fim de se mitigar e diminuir o impacto das desvantagens inerentes à arquitetura serverless, surgem algumas ferramentas e estudos da literatura focados em lidar com certos pontos dessa arquitetura. Nesta seção são apresentados opções que promovem a mitigação destes problemas, assim como a melhoria no desenvolvimento de aplicações serverless.

Alhosban et al. [20] apresentam por meio de seu trabalho, o *CVL - Cloud Vendor Lock-in Prediction Framework*, uma ferramenta que propõe predizer os custos e riscos de vendor lock-in futuros que uma aplicação de nuvem pode enfrentar, além de auxiliar na escolha de qual provedor de nuvem escolher. Embora o framework tenha sido desenvolvido para aplicações de nuvem como um todo, as funcionalidades descritas são aplicáveis para aplicações serverless. O módulo central do CVL recebe como parâmetro dos desenvolvedores fatores ponderados, como custos de serviço, despesas de transferência de dados, conformidade, segurança e escalabilidade.

Os autores do CVL descrevem que os fatores principais na recomendação são o grau de dependência e os custos, não somente financeiros. Para o grau de dependência, os principais fatores são os termos do acordo de serviço previstos nos

contratos, o formato em que os dados são armazenados nos servidores do provedor, como os serviços se integram e o quão complexo é a usabilidade das ferramentas. A respeito dos custos, os principais fatores são os custos de transferência dos dados, custos de saída e os modelos de precificação. A partir da análise feita levando em consideração todos esses fatores, o CVL propõe fornecer uma análise com dados quantitativos para que uma equipe tome decisões, a partir de uma comparação de custos financeiros e custos gerenciais, e o grau de dependência a um provedor de nuvem de acordo com os serviços escolhidos. É uma ferramenta que pode impactar e trazer muitos benefícios aos desenvolvedores e ao mercado de aplicações serverless.

Um outro ponto muito abordado e já citado anteriormente no trabalho, em [15] e [16], é de se seguir a estratégia multi-cloud, no que diz respeito a não se tornar dependente de um único provedor de nuvem.

Weldemicheal [16] exemplifica as vantagens além de somente mitigar o vendor lock-in ao se abordar uma estratégia multi-cloud. Ao se utilizar serviços de mais de um provedor, empresas podem escolher qual provedor oferece melhores preços e melhores funcionalidades para um determinado serviço. Além disso, essa estratégia facilita caso seja necessário migrar e encerrar o contrato com um provedor, uma vez que já se tem mais informação, preparação e um plano de saída mais elaborado, em comparação com empresas que utilizam serviços de um único provedor. O autor também apresenta as vantagens em se utilizar software, ferramentas e soluções de código aberto quando possível. Serviços de código aberto tendem a ser mantidos por organizações que têm como foco principal a qualidade do serviço e adoção de padronização já existente no mercado.

Para Opara-Martins et al. [15], quando se aborda uma estratégia multi-cloud, as organizações ficam cientes desde cedo da importância da interoperabilidade de seus dados, ou seja, tenham ciência do formato em que seus dados estão sendo armazenados, o custo de se trafegar estes dados para fora dos servidores daquele provedor, e o quão acoplado seus dados estão dos serviços contratados. Mesmo que seja custoso em termos financeiros e de tempo manter uma boa interoperabilidade dos dados, os autores ponderam ser importante, para se garantir o funcionamento dos sistemas caso seja necessário ou vantajoso realizar a troca do provedor de nuvem. Além disso, os autores discorrem também sobre as vantagens da cultura DevOps e de se utilizar mais ferramentas para padronização da gerência e operação de suas aplicações, no contexto de aplicações serverless. O DevOps busca unir o desenvolvimento e a operação das aplicações, promovendo automação, agilidade e padronização. A respeito da padronização, existe o conceito de Infrastructure as Code (IaC), que por meio de ferramentas como Terraform, Chef e Puppet, permitem a descrição declarativa da infraestrutura, garantindo consistência e padronização entre os ambientes de diversos provedores de nuvem.

Existem também ferramentas com alta popularidade no mercado que promovem maior agilidade e auxílio no desenvolvimento de aplicações serverless. Um grande exemplo é o Serverless Framework [21], uma ferramenta open-source de linha de comando, que por meio de configurações declarativas promove maior facilidade no desenvolvimento e implantação de aplicações serverless. Oferece suporte a diversos provedores de nuvem e diversas linguagens de programação, ao abstrair especificidades de cada provedor, e instaurar um padrão de configuração, promovendo maior agilidade no desenvolvimento das aplicações. É um dos frameworks voltados para aplicações serverless mais utilizados, tendo surgido em 2015 e recebendo atualizações frequentes, conta também com a possibilidade de integração de plugins disponibilizados pela comunidade, ou criados de acordo com a necessidade das equipes.

V. Conclusão

Na primeira parte do trabalho, pôde-se verificar que a computação em nuvem é uma realidade muito bem instaurada no cenário atual. Diversas aplicações e empresas utilizam desse modelo de computação. A partir de novas descobertas e inovações a respeito dos poderes da computação em nuvem, novos modelos de negócio foram surgindo com o tempo. Desde modelos que possibilitam um controle maior dos recursos alocados, como IaaS, até modelos que abstraem diversos pontos, como PaaS. Nesse contexto, surge a arquitetura serverless, que introduz uma arquitetura de desenvolvimento focado em abstrair aspectos operacionais e de infraestrutura, permitindo que desenvolvedores concentrem esforços exclusivamente na lógica de suas aplicações, por meio das chamadas funções serverless, e deleguem ao provedor de nuvem questões de configuração e escalabilidade.

Este trabalho explorou desde o contexto histórico do surgimento da arquitetura serverless, a aspectos técnicos e gerenciais, vantagens, como redução nos custos gerenciais e de operação, aplicações mais enxutas e menos complexas, controle de gastos e cobrança mais refinada e a capacidade de autoescalonamento preciso; e desvantagens como código mais acoplado a um provedor de nuvem, denominado de vendor lock-in e latência de inicialização.

Foi descrito também o processo de desenvolvimento de uma aplicação serverless nos dois maiores provedores de nuvem do mercado. A partir do desenvolvimento da aplicação de pós-processamento de imagens, foi possível observar que o vendor lock-in não está associado somente ao código da função. A análise comparativa revelou que o acoplamento está muito relacionado as ferramentas de desenvolvimento, nos SDKs específicos, nos serviços de autenticação e nos modelos de Infraestrutura como Código.

Além disso, com a experiência prática, foi possível lidar com algo inesperado - o custo ocioso gerado pelo mecanismo de polling do blob trigger em uma Azure Function. Isso reforça a ideia de que, mesmo em um ambiente serverless, um entendimento aprofundado do funcionamento interno dos serviços é fundamental para evitar surpresas financeiras.

Outro ponto relevante foi o desenvolvimento das aplicações front-end e a devida configuração de autenticação em ambos. Todo esse processo promoveu também uma maior atenção à

segurança das aplicações, assim como um melhor entendimento do protocolo OAuth2.0.

Como trabalho futuro, surge a possibilidade do desenvolvimento e implementação da mesma aplicação utilizando um framework, como o Serverless Framework para verificar as possíveis mitigações dos desafios enfrentados. Além disso, uma análise comparativa de performance, focada em métricas como latência de inicialização (cold start) e tempo de resposta entre os provedores podem trazer detalhes técnicos relevantes.

O trabalho como um todo forneceu informações valiosas e qualificadas a partir do referencial bibliográfico, e trouxe uma experiência relevante a partir do desenvolvimento prático da aplicação de pós-processamento de imagens. Um melhor entendimento do funcionamento dos provedores de nuvem, como os serviços funcionam e se comunicam, a integração segura de uma aplicação front-end a uma aplicação serverless de forma segura e questões como adequação a prática de Infraestrutura como Código promoveram um aprendizado considerável.

REFERÊNCIAS

[1] Y. Li, Y. Lin, Y. Wang, K. Ye, and C. Xu, "Serverless computing: State-of-the-art, challenges and opportunities," *IEEE Transactions on Services Computing*, vol. 16, no. 2, pp. 1522–1539, 2023.

[2] V. Yussupov, U. Breitenbücher, F. Leymann, and C. Müller, "Facing the unplanned migration of serverless applications: A study on portability problems, solutions, and dead ends," in *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, UCC'19, (New York, NY, USA), p. 273–283, Association for Computing Machinery, 2019.

[3] B. Bhagat, S. Khobragade, and A. Arudkar, "The evolution of cloud computing," *Journal of Network Security*, 2023.

[4] B. Golden, *Virtualization For Dummies*. –For dummies, Wiley, 2007.

[5] P. Borra, "An overview of cloud computing and leading cloud service providers," *International Journal of Computer Engineering and Technology*, vol. 15, 05 2024.

[6] Armbrust, Michael, A. Fox, Armando, Griffith, Rean, Joseph, D. Anthony, R. Katz, R. H, A. Konwinski, Andrew, G. Lee, Gunho, Patterson, D. A, Rabkin, Ariel, Stoica, and Matei, "Above the clouds: A berkeley view of cloud computing," *Berkeley, CA: University of California*, 01 2009.

[7] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson, "Cloud programming simplified: A berkeley view on serverless computing," *arXiv preprint arXiv:1902.03383*, 2019.

[8] Amazon Web Services, "Aws re:invent 2014 - introducing aws lambda." https://www.youtube.com/watch?v=JIQETrFC_SQ, 2014. Acesso em: 30 dez. 2024.

[9] Market.us, "Serverless architecture market size, share, growth analysis report." https://market.us/report/serverless-architecture-market/, 2024. Acesso em: 22 Out. 2024.

[10] X. C. Lin, J. E. Gonzalez, and J. M. Hellerstein, "Serverless boom or bust? an analysis of economic incentives," in *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*, USENIX Association, July 2020.

[11] S. Eismann, J. Scheuner, E. v. Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, and A. Iosup, "The state of serverless applications: Collection, characterization, and community consensus," *IEEE Transactions on Software Engineering*, vol. 48, no. 10, pp. 4152–4166, 2022.

[12] P. Castro, V. Isahagian, V. Muthusamy, and A. Slominski, "The rise of serverless computing," *Communications of the ACM*, vol. 62, pp. 44–54, 11 2019.

[13] H. Shafiei, A. Khonsari, and P. Mousavi, "Serverless computing: A survey of opportunities, challenges, and applications," *Association for Computing Machinery*, vol. 54, Nov. 2022.

[14] S. Eismann, J. Scheuner, E. van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, and A. Iosup, "Serverless applications: Why, when, and how?," *IEEE Software*, vol. 38, no. 1, pp. 32–39, 2021.

[15] J. Opara-Martins, R. Sahandi, and F. Tian, "Critical analysis of vendor lock-in and its impact on cloud computing migration: a business perspective," *Journal of Cloud Computing*, vol. 5, no. 1, p. 4, 2016.

[16] T. Weldemicheal, "Vendor lock-in and its impact on cloud computing migration," 2023.

[17] MDN Web Docs, "Spa (single-page application)." https://developer.mozilla.org/en-US/docs/Glossary/SPA, 2023. Acesso em: 15 de Jun. 2025.

[18] M. B. Jones, J. Bradley, and N. Sakimura, "JSON Web Token (JWT)." RFC 7519, May 2015.

[19] Microsoft, "Api authentication with api management (apim) using apim policies with entra id and app roles." https://github.com/microsoft/apim-auth-entraid-with-approles, 2024. Acesso em: 25 de Mai. 2025.

[20] A. Alhosban, S. Pesingu, and K. Kalyanam, "Cvl: A cloud vendor lock-in prediction framework," *Mathematics*, vol. 12, no. 3, 2024.

[21] Serverless, Inc, "Serverless framework." https://www.serverless.com. Acesso em: 15 Jan. 2025.