Associação Propagadora Esdeva
Centro Universitário Academia – UniAcademia
Curso de Sistemas de Informação
Trabalho de Conclusão de Curso - Artigo

Migração de um monolito para uma arquitetura de microsserviços utilizando *serverless*

Rogério Tostes[1]
Centro Universitário Academia, Juiz de Fora, MG

Tassio Ferenzini Martins Sirqueira[2]
Centro Universitário Academia, Juiz de Fora, MG

Linha de Pesquisa: Engenharia de Software

RESUMO

Pode-se considerar que sistemas monolíticos já não são a melhor resposta para o desenvolvimento de grandes sistemas tendo a execução centralizada em um único servidor. Entretanto, esses sistemas têm se tornado de modo crescente, maiores e complexos, dificultando aspectos como manutenibilidade e escalabilidade. Essa difícil realidade vivida por muitas organizações motivou o presente trabalho a trazer uma estratégia de uma arquitetura denominada de microsserviços utilizando *serverless*. Objetivou-se neste trabalho criar uma arquitetura de microsserviços em *serverless*, a fim de viabilizar um sistema menos inflexível e com possibilidades de ajustes, oferecendo assim um sistema mais resiliente, escalável e menos vulnerável. Nessa nova arquitetura, o software é decomposto em pequenas partes que funcionam de modo independente e autônomo, trazendo algumas melhorias em termos de atributos de qualidade de software. Todavia, percebe-se também que o processo de migração de um sistema monolítico para uma arquitetura de microsserviços apresenta alguns desafios que não raro o tornam malsucedido. Os resultados preliminares indicam que uma arquitetura *serverless* elimina a complexidade de gerenciamento de servidores e os custos com sua manutenção, mas dependem de estudos mais detalhados.

Palavras-chave: Arquitetura monolítica. Migração. Microsserviço. *Serverless*.

ABSTRACT

It can be considered that monolithic systems are no longer the best answer for the development of large systems having the execution centralized on a single server. However, these systems have become increasingly larger and complex, making aspects such as maintainability and scalability difficult. This difficult reality experienced by many organizations motivated the present work to bring a strategy of an architecture called

[1] Discente do Curso de Sistemas de Informação do Centro Universitário Academia – UniAcademia. E-mail: rogertostes@yahoo.com.br.
[2] Docente do Curso de Sistemas de Informação do Centro Universitário Academia. Orientador.

microservices using serverless. The objective of this work was to create a serverless microservices architecture, in order to enable a less inflexible system with the possibility of adjustments, thus offering a more resilient, scalable and less vulnerable system. In this new architecture, the software is decomposed into small parts that work independently and autonomously, bringing some improvements in terms of software quality attributes. However, it is also clear that the process of migrating from a monolithic system to a microservices architecture presents some challenges that often make it unsuccessful. Preliminary results indicate that a serverless architecture eliminates the complexity of server management and maintenance costs, but depends on more detailed studies.

Keywords: Software architecture. Migration. Microservice. Serverless.

1 INTRODUÇÃO

A tecnologia, tal qual conhecemos hoje, passou por diversas transformações ao longo dos tempos e continua avançando de forma vertiginosa; onde vivemos em: “um processo dinâmico de melhorias, crescimento e evolução” (NEVES, 2002). Conforme Figueiredo (2013) reforça: “inovações tecnológicas, inovações causadas pelo surgimento de novos negócios, novos *frameworks,* novas linguagens e novos protocolos que surgem em um intervalo cada vez menor de tempo”; o que caracteriza um cenário marcado por uma moderna rede de computadores interoperáveis.

Por trás dos diferentes aplicativos, das ferramentas e sites – entre outros serviços – que usamos sempre que acessamos o mundo digital, existe uma engenhosa estrutura responsável pelo bom desempenho dessas operações. Neste aspecto, é necessário pensar na arquitetura dos sistemas e em suas estruturas com frequência. *Softwares* são criados e desenvolvidos como solução para diversas questões, entre elas (aperfeiçoar processos, melhorar experiências digitais e ampliar mercados).

Monólito, conforme o dicionário *Priberam* significa “obra construída em um único bloco”, é por isso que esse termo é utilizado para definir a arquitetura de alguns sistemas. Sendo assim, uma arquitetura monolítica é um sistema único, não dividido, que roda em um único processo, uma aplicação de *software* em que diferentes componentes estão ligados a um único programa dentro de uma única plataforma. Com o tempo, as estruturas monolíticas vão sofrendo mudanças, suas escalabilidades se tornam inflexíveis e sua manutenção mais complexa. Nesse processo, é interessante pensar em uma arquitetura cujo sistema responda a essas funções com mais flexibilidade.

A arquitetura de microsserviços apresenta uma arquitetura de software diferente do monolito, no qual sistemas grandes e complexos são transformados em componentes individuais. As vantagens dos microsserviços são a sua escalabilidade distribuída, ou seja, não é preciso crescer a aplicação inteira para solucionar um problema, podemos escalar apenas recursos específicos; contrário ao monolito cujo

auto acoplamento dos sistemas dificultam a manutenção e evolução dos sistemas, e a escalabilidade. Nesse ponto, vale pensar em uma arquitetura de microsserviços que se comunicam entre si por meio de redes e, como opção de arquitetura, oferecem diversos modos de resolver problemas que você poderá enfrentar (NEWMAN, 2020).

Serverless é um termo que traduzido literalmente do inglês, significa “sem servidor”, o que não é bem verdade. A arquitetura *serverless* ainda é baseada em servidores. Nela, porém, o desenvolvedor não precisa configurar ou se preocupar com a maioria dos aspectos da infraestrutura em que sua aplicação será rodada. Essas decisões dinâmicas de infraestrutura ficam ocultas para o desenvolvedor ou operador.

A arquitetura *serverless* é um tema em alta pela a computação em nuvem, mas observa-se que, principalmente os sistemas disponibilizados pelo governo ficaram defasados pelo formato monolítico do sistema, por falta de manutenção, pelas dificuldades na evolução dos sistemas e até mesmo por questões de segurança.

A proposta do presente trabalho é mostrar a relevância de se construir uma arquitetura de solução conhecida como *serverless,* partindo da ideia de evolução da arquitetura de um monolito. Para a construção dessa arquitetura neste trabalho está sendo usada a *AWS (Amazon Web Services)*[3] que é uma plataforma de sistema em nuvem.

1.2 OBJETIVOS

Esta seção aborda os objetivos a serem alcançados através deste projeto, citando o objetivo geral e os objetivos específicos, a saber:

1.2.1 Objetivo Geral

- Criar uma arquitetura de microsserviços utilizando *serverless*, partindo da ideia de evolução da arquitetura de um monolito, a fim de viabilizar um sistema menos inflexível e com possibilidades de ajustes. Com tal perspectiva, o objetivo geral se desdobra em objetivos “menores” mais específicos, a saber:

1.2.2 Objetivos específicos

- Possibilitar a criação de um *software* resiliente, visando assim a garantia de que o sistema funcione, mesmo quando outros sistemas dependentes não operem bem;

- Propiciar um *software* escalável de modo a destacar sua capacidade de expansão sem aumentar de forma drástica os custos com recursos técnicos, e como aplicá-lo;

[3] https://aws.amazon.com

- Oferecer um *software* seguro, diminuir a vulnerabilidade de dados dos usuários e demonstrar um exemplo de transformação de monolito em microsserviços utilizando *serverless*.

1.3 JUSTIFICATIVA

Atualmente grande parte das aplicações utiliza a arquitetura monolítica como padrão para desenvolvimento de software (NEVES, 2020) uma vez que parte do princípio de que todos os processos dentro de uma mesma solução estão ligados. Isso quer dizer que são acoplados, que estão dentro de uma base de dados similar e executam seus processos como um único serviço. Contudo, em determinados casos esta arquitetura se mostra inviável e propensa a falhas, tais como: a dificuldade de escalar a aplicação; o alto custo com a manipulação e manutenção, entre outras falhas.

Em 2007 o governo federal lançou o software Público Brasileiro (SPB), utilizado como alicerce para definir a política de desenvolvimento, distribuição e uso de *software* pelo setor público do Brasil. Dentre esses sistemas, destaque para o Controle de Marcas e Sinais (CMS) que auxilia o registro e consulta por marcas e sinais utilizados por produtores rurais.

O CMS é um sistema monolítico e a ideia do trabalho, conforme já afirmado, é redesenhar esse programa e transformá-lo em uma arquitetura de microsserviços *serverless* e servir como base para a migração de outros quaisquer sistemas do tipo monolítico, pois os conceitos servem como base de inspiração e estudos.

Em resposta a isso surgiu uma nova arquitetura de *software* pensada a partir da lógica de microsserviços, tornando possível modularizar serviços menores responsáveis por determinada área da aplicação e que podem funcionar de forma independente.

Com base nessas necessidades foi pensado em um *software serverless* fundamentado no modelo de microsserviços que viabiliza sua escalabilidade, resiliência e evolutiva; compreendendo os termos da seguinte forma:

- Escalabilidade: é a capacidade de aumentar o tamanho do *software* ou do seu uso. Em outras palavras, é uma característica desejável em todo sistema, rede ou um processo, que indica sua habilidade de manipular uma porção crescente de trabalho de forma uniforme, ou estar preparado para crescer.

- Resiliente: é aquele sistema capaz de cumprir sua missão mesmo diante de crises e adversidades. Isto é, fornecer os recursos necessários e continuar operante, apesar do estresse excessivo da operação.

- Evolutiva: é a arquitetura que suporta mudanças contínuas e incrementais como um primeiro princípio por meio de vários aspectos.

Com isso, nota-se que a arquitetura de microsserviços apresenta com frequência, “maior flexibilidade e são passíveis de mudança” (Newman, 2015) ou atualizações, tanto como um todo, quanto por meio de atualizações independentes de seus componentes, conforme o autor:

> A natureza independente das implantações da abertura a novos modelos para aumentar a escala e robustez dos sistemas, além de permitir que você misture e combine diferentes tecnologias. Como podemos trabalhar em serviços em paralelo, é possível incluir mais desenvolvedores para resolver um problema sem que atrapalhem uns aos outros. (2015, p. 20).

A possibilidade supracitada se dá pelo fato de a arquitetura ser distribuída e permitir que enquanto seus componentes são atualizados, não seja comprometido o funcionamento da aplicação. Além das vantagens citados anteriormente,

> O isolamento de processos também torna possível diversificar as opções de tecnologia de escolhermos, talvez misturando diferentes linguagens e estilos de programação, plataforma de implantação ou bancos de dados para encontrar a combinação correta. (2015, p.20).

Pode-se inferir, portanto, que as arquiteturas de microsserviços também podem oferecer uma maior flexibilidade.

2 REFERENCIAL TEÓRICO

Todo sistema computacional parte de uma organização, de uma estrutura. Essa organização é também conhecida como uma arquitetura de *software* que é responsável por definir quais componentes farão parte de um projeto, quais suas características, funções e a forma como devem interagir entre si e com outros *softwares*.

Dentro dessa estrutura, é possível entender as diferenças entre as linguagens, sistemas computacionais e ambientes da computação, e por isso mesmo, torna-se uma ideia difícil de se conceituar. Microsserviços e *serverless* são temas relativamente novos em se tratando de discussões teóricas, percebe-se a ausência de fontes mais aprofundadas. Em contrapartida, há muitas referências empíricas no universo tecnológico.

Do mesmo modo, migrar de um sistema para outro também é uma escolha complexa, que envolve a análise de diversos fatores. Desenvolver um *software* mais flexível que visa promover certa “mobilidade” e agregar novas tecnologias exige planejamento e estudo. Sam Newman (2020), traz conceitos, vantagens e desvantagens da migração e referências de melhores práticas para fazer essa migração.

Na criação de uma arquitetura, é fundamental conhecer o domínio, pois é onde reside um conjunto de conceitos, princípios e técnicas necessárias aos desenvolvimentos de sistemas. Eric Evans (2016) apresenta um desenvolvimento de domínios trazendo a modelagem de domínio para o software, visando tirar melhor proveito para o negócio. Pensar em *Domain-Drive Design* (*DDD*) significa desenvolver sistemas de acordo com o domínio relacionado ao problema que estamos propondo resolver.

Um dos primeiros artigos sobre arquitetura de *software* publicado foi de autoria de Edsger Dijkstra, em 1968. Mas, o termo “Arquitetura de software” se consolidou no meio científico no final da década de 1980 (GIROLDO, 2022). Contudo, vale ressaltar que o foco abordado aqui, neste trabalho, debruça-se sobre a questão da criação de uma arquitetura de microsserviços em um sistema *serverless.*

Os estudiosos Alexandre Davi Zanelatto (2019) e Gustavo Matozinho Lima (2022) apresentam trabalhos relevantes com relação ao tema de uma arquitetura *serverless*, entretanto, somente definem conceitos e não apresentam propostas pragmáticas, ou seja, não entregam uma solução evolutiva para um projeto já existente. Buscamos aqui propiciar uma solução de nível corporativo.

Vale salientar ainda que na busca por informações sólidas que pudessem ratificar nossas escolhas resultando na criação do sistema proposto, consultamos o site da IBM-Brasil[4] pensando na relevância da consistência, disponibilidade e partições tolerantes a falhas, o Teorema CAP (que vem do inglês Consistency, Availability and Partition Tolerance)[5] sobre os bancos de dados.

Bancos de dados NoSQL (não relacionais) são ideais para aplicativos de rede distribuída. Ao contrário de suas contrapartes SQL (relacionais) verticalmente escaláveis, os bancos de dados NoSQL são horizontalmente escaláveis e foram criados para serem distribuídos (Sirqueira, Dalpra, 2018)[6]. É possível ajustar sua escala rapidamente em uma rede crescente consistindo em múltiplos nós interconectados. A seguir como se classificam os bancos de dados baseados no Teorema CAP:

- Banco de dados CP: um banco de dados CP entrega consistência e tolerância de partição em detrimento da disponibilidade. Quando uma partição ocorre entre dois nós quaisquer, o sistema deverá desativar o nó não consistente (ou seja, torná-lo indisponível) até que a partição seja resolvida.
- Banco de dados AP: um banco de dados AP entrega disponibilidade e tolerância de partição em detrimento da consistência. Quando ocorre uma partição, todos os nós permanecem disponíveis, exceto aqueles na extremidade errada de uma

[4]< https://www.ibm.com/br-pt/cloud/learn/cap-theorem>. Acesso em 28/08/2022.
[5] <https://pt.wikipedia.org/wiki/Teorema_CAP>. Acesso em 28/08/2022.
[6]Disponível em:<https://www.researchgate.net/publication/327035187_NoSQL_e_a_Importancia_da_Engenharia_de_Software_e da_Engenharia_de_Dados_para_o_Big_Data>. Acesso em 19/09/2022.

partição podem retornar uma versão mais antiga de dados do que outros. Quando a partição é resolvida, os bancos de dados AP geralmente ressincronizam os nós para corrigir todas as inconsistências no sistema.

- Banco de dados CA: um banco de dados CA entrega consistência e disponibilidade em todos os nós. Porém, isso não é possível se houver uma partição entre dois nós quaisquer no sistema, no entanto, e, portanto, não poderá entregar tolerância a falhas.

3 METODOLOGIA

A estratégia de pesquisa do presente trabalho é qualitativa; neste sentido, primeiro se definiu o contexto do projeto e o que precisaria conter de tecnologias, após essa definição realizou-se buscas de referências para entender melhor os conceitos e suas nuances, nesse sentido, averiguou-se informações e referências utilizando as *strings* de busca:(migração, sistemas, arquitetura, microsserviços e *serverless*) na base do *Google Scholar*[7] e para concluir foi efetuada a etapa de análise.

Lembrando que para a construção da arquitetura de microsserviços utilizando *serverless*, neste trabalho está sendo usada a plataforma de computação em nuvem *AWS (Amazon Web Services)*[8]. A *AWS*, oferece ainda o serviço *Amazon Virtual Private Cloud (Amazon VPC)*, que viabiliza o controle total sobre seu ambiente de redes virtuais, incluindo posicionamento de recursos, conectividade e segurança.

Diante das variedades de serviços da *AWS*, vale questionar: como podemos provisionar, gerenciar e subir toda essa infraestrutura? Responderemos adiante.

Posto isso, voltemos nossa atenção para a arquitetura aqui proposta. As aplicações modernas são desenvolvidas primeiro sem servidor, uma estratégia que prioriza a adoção de serviços *serverless*, para que se possa aumentar a agilidade em toda a pilha de aplicações. Tem-se como exemplo o programa público do Governo Federal: o Controle de Marcas e Sinais (CMS).

A aplicação Marcas e Sinais é uma aplicação *Creat Read Update Delete* (*CRUD)*[9] que auxilia o registro e consulta por marcas e sinais utilizados por produtores rurais. Fazendo a análise do código baixado no repositório público abaixo assinalado[10], temos:

---

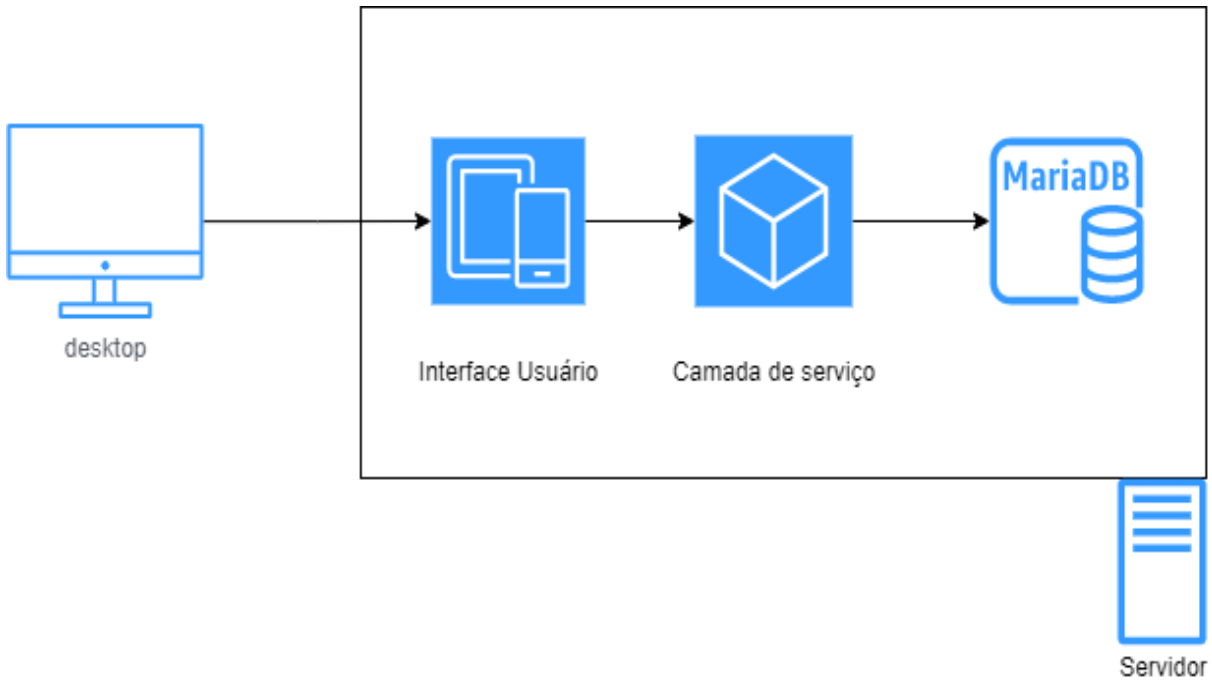
[7]<https://scholar.google.com/scholar?hl=arquitetura%2C+monolitos%2C+microservi%C3%A7os+e+serverless&btng=> Acesso em 28/08/2022

[8] https://aws.amazon.com

[9] Crud (Create Read Update e Delete). Em tradução livre para o português seria Criar, Ler, Atualizar, e Excluir. Crud pode ser compreendido como uma sequência de funções de um sistema que trabalha com banco de dados. *C.f.*:< https://pt.wikipedia.org/wiki/CRUD>. Acesso em: 27/08/2022

[10]< https://softwarepublico.gov.br/gitlab/groups/marca-sinais>. Acesso em: 19/09/2022

Figura 1.



Fonte: Elaboração Própria.

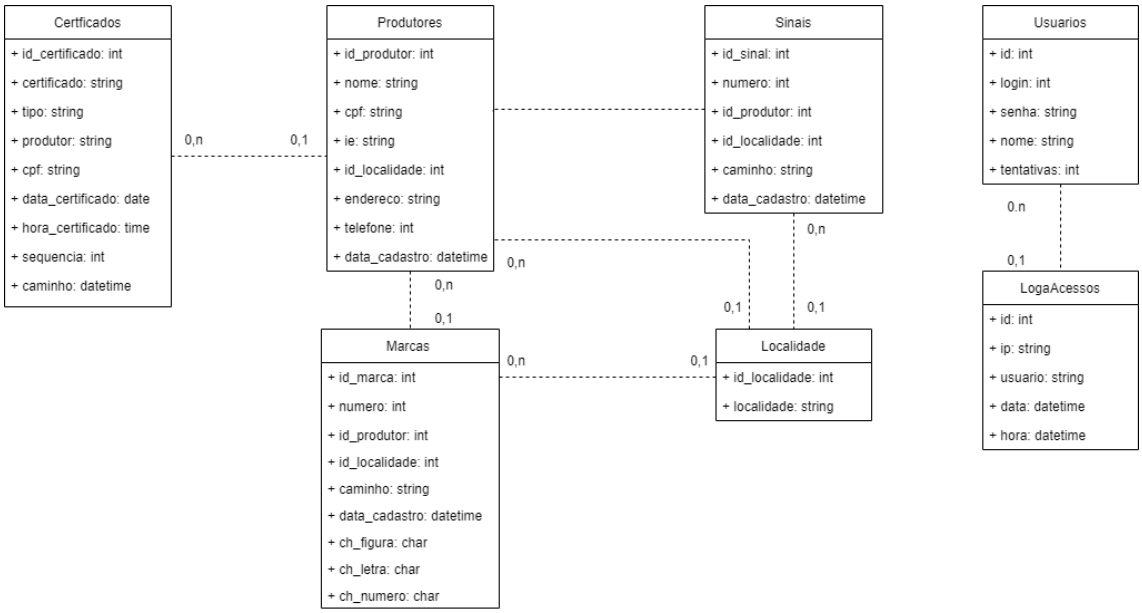
Na figura (1) temos a representação dos sistemas monolíticos que definimos no presente trabalho. A figura 1, representa a arquitetura do tipo monolítica do sistema de Controle Marcas e Sinais (CMS), que é o sistema, em que foi feita a migração, ou seja, a figura nos mostra que todas as peças do sistema rodam em apenas um servidor. Hoje o CMS é feito em *PHP*[11] com banco de dados *MySQL*[12].

Pensando na evolução desse sistema para todo o Brasil e, assegurada a dificuldade em se fazer a escalabilidade, controle e até mesmo a segurança, pois ele será exposto para mais pessoas, trouxemos um novo desenho como proposta de solução.

No primeiro passo, para se pensar em uma solução de microsserviços e responsabilidades divididas depois que a decisão de migração e evolução foi tomada, elencamos todos os prós e contras dos domínios que teriam o sistema. Assim cada pequeno serviço ou componente iria trabalhar unicamente para aquele domínio, facilitando assim o crescimento do sistema no futuro.

[11] <https://www.php.net.> Acesso em: 19/09/2022
[12]MySQL. Disponível em <https://www.mysql.com/>. Acesso em: 19/09/2022

Figura 2.



Fonte:Elaboração Própria.

Na figura 2, temos o diagrama UML[13] - das entidades de negócio e seus relacionamentos do sistema CMS.

No CMS temos as entidades vistas na figura 2, fazendo um trabalho de remodelagem de domínio usando o *DDD* e *Event Storming*. O *DDD* é, conforme visto anteriormente, um conjunto de técnicas e ferramentas criadas por Eric Evans visando auxiliar desenvolvedores na criação de softwares mais precisos nas necessidades de negócios. Já o *Event storming* é uma técnica[14] que se configura por sua rapidez de design e que permite engajar o especialista técnico e o desenvolvedor do domínio.

Podemos observar na figura (2) que as principais entidades, Marcas e Sinais, definem o sistema em que o CORE, entre demais entidades, figura apenas como complementos dela.

Considerando as aplicações Marcas e Sinais como as entidades existentes nos microsserviços e, uma vez mapeadas, foi pesquisado na evolução do sistema e de como transformá-la em uma arquitetura *serveless* em microsserviços. Em seguida, passamos a pensar em escalabilidade, resiliência, segurança e em custos, já que toda

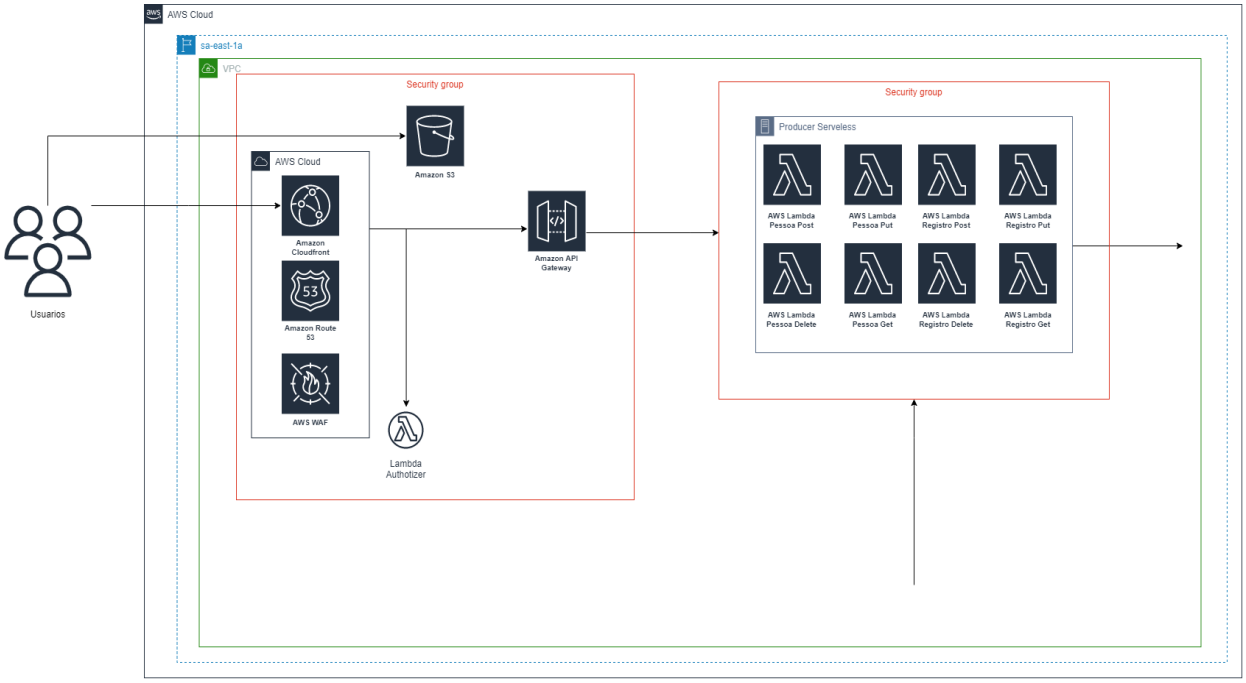
[13] UML (Unified Modeling Language) - Em tradução livre para o português seria Linguagem de Modelagem Unificada. *C.f.*:< https://pt.wikipedia.org/wiki/UML>. Acesso em 02/08/2022
[14] < https://www.eventstorming.com/.> Acesso em 08/08/2022

essa nova infraestrutura ficaria hospedada em domínio de uma empresa terceirizada, a *AWS*.

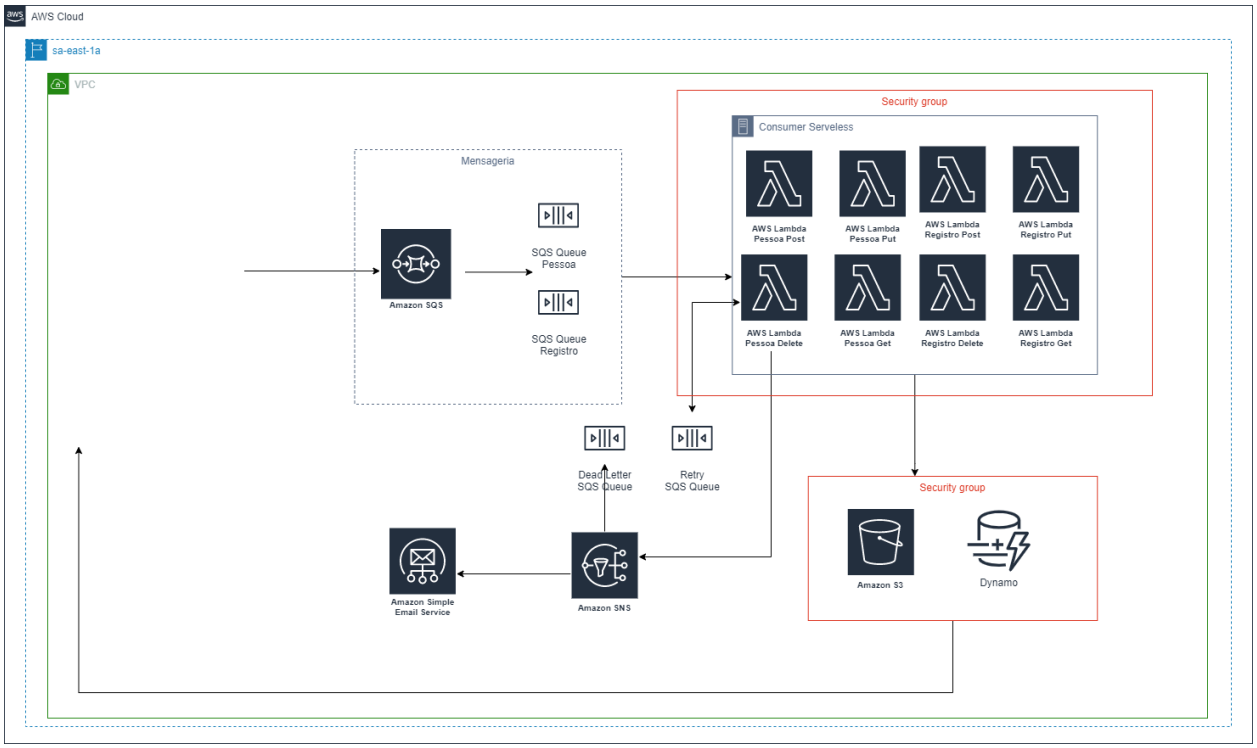
A *AWS* ou *Amazon Web Service* é uma plataforma de computação sem servidor provida pela Amazon[15] e oferece tecnologias para executar código, gerenciar dados e integrar aplicações, tudo isso sem a necessidade de gerenciar servidores. As tecnologias sem servidores contam com escalabilidade automática, alta disponibilidade integrada e um modelo de faturamento pago por utilização para aumentar a agilidade e otimizar os custos. Portanto, para essa migração foi utilizado a nuvem AWS.

Isso posto, evoluímos o desenho de um monolito para uma solução mais robusta na nuvem, conforme mostramos na figura 3, em que abordamos cada fase do desenho.

Figura 3.



[15] https://aws.amazon.com



Fonte: Elaboração própria

Na figura 3 acima está representada a nova arquitetura de microsserviços utilizando *serverless* para o sistema CMS, resultante da migração proposta.

Percebe-se que essa solução, em um primeiro momento, parece ser um pouco maior comparada à figura (1); esse fato se dá pela impossibilidade de entrar e visualizar toda teia existente dentro de um monolito, visto que, como o próprio nome diz: é um bloco fechado. Aqui nós já temos todas as camadas visíveis e claras com cada uma tendo sua própria atribuição.

Na primeira camada a qual, foca-se em segurança, tem-se um bloco responsável por essa incumbência, dessa forma há garantias de autenticação e proteção contra-ataques e outros tipos de falhas. Logo, na primeira camada de segurança observamos os serviços do “Route 53”:

O *Amazon Route 53* é um serviço de *Domaine Name System* (DNS)[16] na nuvem altamente disponível e escalável. Sua projeção foi para oferecer aos desenvolvedores e empresas uma maneira altamente confiável e econômica de direcionar os usuários finais aos aplicativos de Internet, convertendo nomes como *www.example.com* para endereços IP numéricos como 192.0.2.1, usados pelos computadores para se conectarem entre si e facilitarem o trânsito.

[16] Disponível em: <http//:aws.amazon.com/pt/route53>. Acesso em 12/08/2022

Outros serviços com presenças constatadas na figura 3 são, a do *Amazon CloudFront*, um serviço de rede de entrega de conteúdo (CDN) criado para alta performance, segurança e conveniência do desenvolvedor; o *web application firewall* (WAF), um firewall que monitora, filtra e protege as aplicações da *Web* ou *APIs* contra *bots* e *exploits* comuns na Web que podem afetar a disponibilidade, comprometer a segurança ou consumir recursos em excesso; e ainda pensando na camada de segurança, tem-se o JWT, uma chave de autenticação única gerada e validada via o lambda *authorizer*.

JSON Web Token (JWT) é um padrão aberto (RFC 7519) que define uma forma compacta e autocontida para a transmissão segura de informações entre as partes como um objeto JSON[17].

Sobre a camada intermediária temos o *api gateway*, que funciona como uma ponte entre softwares distintos. Ele serve como centralizador e direcionador de todas as chamadas para os serviços corretos. O *Amazon API Gateway* é um serviço gerenciador que permite aos desenvolvedores criarem, publicarem, manterem, monitorarem e protegerem APIs em qualquer escala com facilidade.

E para finalizar a camada intermediária temos o *Amazon S3*, que é um gerenciador de armazenamento de dados com uma infinidade de usos, nesse caso ele irá hospedar o nosso *front end*, pois o (*Amazon S3*) oferece escalabilidade, disponibilidade de dados, segurança e performance líderes do setor.

Em se tratando da segunda camada temos os *producers* que são os produtores de informações onde os lambdas vão fazer as operações solicitadas pelos usuários. O AWS Lambda[18] é um serviço de computação sem servidor e orientado a eventos que permitem executar códigos para praticamente qualquer tipo de aplicação ou serviço de *back-end* sem provisionar e gerenciar servidores.

Nota-se que existe um lambda para cada operação, por boa prática o lambda é usado para apenas uma determinada função e tem uma responsabilidade única, por esse motivo temos um lambda para cada método migrado do CMS. Falando de níveis de produtores e consumidores, temos o pub/sub *pattern*, local em que há publicações de eventos e onde os consumidores também são notificados sobre tais ocorrências. Isso ajuda a manter a resiliência do sistema, que suportará o recebimento e o processamento de várias mensagens. Para enfileirar essas mensagens usamos o *Amazon Simple Queue Service* (SQS).

O *Amazon Simple Queue Service* (SQS)[19] é um serviço de enfileiramento de mensagens rápido, confiável, escalável e gerenciado, que permite o desacoplamento e a escalabilidade de microsserviços, sistemas distribuídos e aplicações sem servidor. Esse serviço nos ajuda a observar a, disponibilidade, resiliência e escalabilidade do sistema, assim podemos usar estratégias de retentativas, avisos e tratamentos de

[17]Disponível em: https://marquesfernandes.com/tecnologia/json-web-token-jwt. Acesso em 15/08/2022
[18]AWS Lambda disponível em: < https://aws.amazon.com/pt/lambda>. Acesso em 06/08/2022
[19] <https://aws.amazon.com >. Acesso em 15/08/2022

erros, além de notificar os usuários e os administradores de sistemas; para isso usa-se o *Amazon Simple Notification* (SNS), um serviço de mensagens totalmente gerenciado para a comunicação de aplicação para aplicação (A2A) e de aplicação para pessoa (A2P).

Com relação à última camada, temos os consumidores das informações; onde se recebe as mensagens dos produtores que, por sua vez, seguiram nossa estratégia de pub/sub, usando o SQS. Nos consumidores os dados irão persistir em nosso banco não relacional[20], o *DynamoDB*.

O *Amazon DynamoDB* foi projetado para executar aplicações de alta performance em qualquer escala, além de contar com recursos integrados de segurança, backup e restauração, bem como armazenamento em cache na memória.

Seguindo o teorema de cap, já citado, o banco de dados vai ser do tipo CP, isso quer dizer com consistência forte e alta disponibilidade.

Conforme questionado anteriormente – após visualização da infraestrutura da nova arquitetura – como iremos gerenciar e provisionar tudo isso? A resposta é usando TerraForm[21], uma ferramenta e linguagem de código livre que implementa a automação de infraestrutura para as mais diversas clouds disponíveis hoje. A intenção dela é implementar um ambiente de cloud híbrida ou *multi-cloud*.

Terraform permite aos desenvolvedores usarem uma linguagem de configuração de alto nível chamada HCL (*HashiCorp Configuration Language*) para descrever a infraestrutura na cloud ou em implementação local de estado final desejada para executar um aplicativo.

É interessante usar o Terraform em infraestrutura como código, pois favorece na velocidade, confiabilidade de implementações, principalmente, em se tratando de grandes estruturas, além do auxílio na organização e nos testes locais. Além disso, o trabalho com lambdas utiliza o *AWS SAM*[22] (Serverless Application Mode) para fazer seu deploy e algumas infraestruturas da AWS.

---

[20]Um banco de dados não relacionais é qualquer banco de dados que não segue o modelo relacional fornecido pelos sistemas tradicionais de gerenciamento de bancos de dados relacionais (SGBDR). Esta categoria de bancos de dados, é também conhecida como banco de dados NoSQL (Not Only SQL ou Não Apenas SQL). Disponível em: https://blog.debugeverything.com/pt/. Acesso em: 14/08/2022

[21] https://www.ibm.com/br-pt/cloud/learn/terraform. Acesso em: 28/08/2022

[22] Disponível em: <https://docs.aws.amazon.com/pt_br/serverless-application-model/latest/developerguide/serverless-sam-reference.html >. Acesso em: 08/10/2022

A *AWS SAM CLI*[23] é uma ferramenta de linha de comando que opera em um *AWS SAM* modelo e código do aplicativo. A *AWS Command Line Interface* (AWS CLI) é uma ferramenta unificada para o gerenciamento de seus produtos da AWS. Com a *AWS SAM CLI*, pode-se invocar funções do Lambda localmente, criar um pacote de implantação para seu aplicativo sem servidor, no AWS Nuvem e assim por diante.

Em suma, os usuários fazem acesso a *URL* ao *DNS* que está no *Route 53*, acessam ao front que está no *S3*, e *CloudFront* sensibiliza toda essa conexão ao modo de diminuir a latência entre a comunicação e passam pelo o WAF que bloqueia chamadas maliciosas. Após o acesso, temos um *lambda authorizer* que valida o acesso via STS, após o *gateway* recebe as requisições com os cabeçalhos.

Na parte dos *producers* tem-se o lambda que foi invocado pelo usuário, passando pelas camadas anteriores. Falando de uma requisição de cadastro, ela passará pelo *lambda* do seu domínio e irá bater no SQS. Por sua vez, o SQS possui a fila pessoa e Registro e as mensagens são divididas por tipos de filas. Depois da mensagem inserida nas filas o lambda que fica na parte do consumer irá escutar e fazer a inserção no *DynamoDb*. Caso surja algum problema, temos as filas de *retry* que irão tentar mais duas vezes e a *dead letter* para armazenar as mensagens que o tópico de *retry* não obteve sucesso ao enviar novamente, pode ser um erro de infraestrutura, como por exemplo, o banco estar fora ou aquele contrato que está na mensagem pode ser inválido. Caindo na *deadletter* temos o *SNS* que fica ouvindo as mensagens ali e dispara um e-mail.

Uma demonstração do trabalho encontra-se disponível no link: *https://youtu.be/k-1e7FFgtuk*. O repositório de código fonte está disponível para download e consulta em *https://github.com/ces-jf/marcaresinaisserveless* .

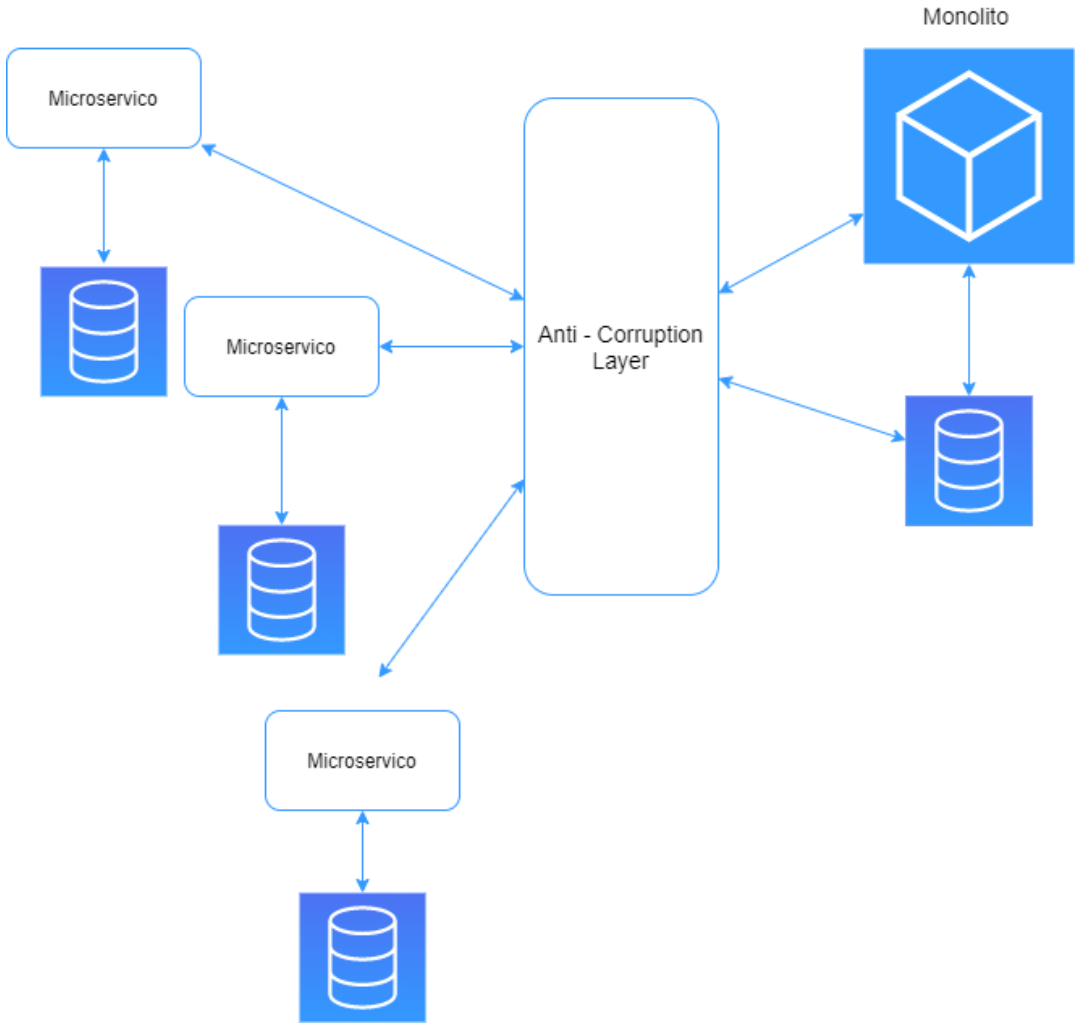
4 RESULTADOS E DISCUSSÃO

Depois de toda a migração do sistema legado e a implementação dessa nova arquitetura, conseguimos entregar um software resiliente, escalável e seguro para o usuário. A intenção desse trabalho além de fazer a migração é também mostrar melhores práticas, conceitos e ferramentas *cloud*.

A migração do sistema proposto foi feita de uma única vez dispensando um padrão para convivência entre a arquitetura antiga e a nova; entretanto, caso seja necessário a convivência entre os dois sistemas, pode-se recorrer ao padrão de camada anticorrupção (ACL).

[23] Disponível em: <https://aws.amazon.com/pt/cli/>. Acesso em: 08/10/2022

Figura 4.

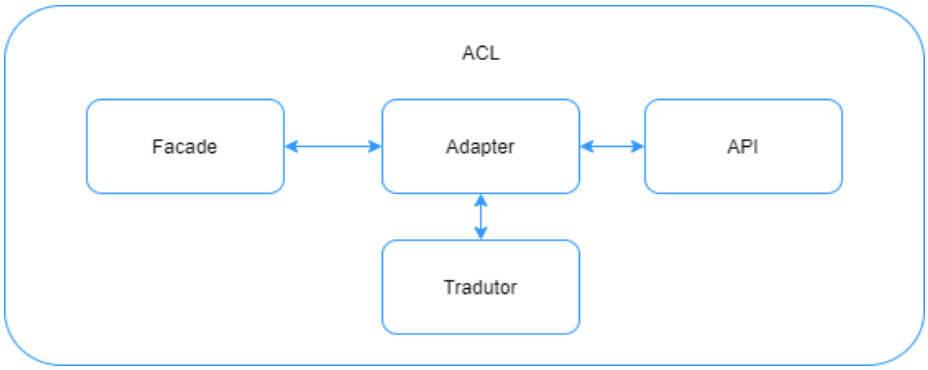


Fonte: Elaboração própria

Uma camada anticorrupção pode ser descrita como uma base entre diferentes subsistemas que não compartilham a mesma semântica. Com o padrão (ACL) é criado uma base entre os dois sistemas que irá fazer a comunicação entre eles, possibilitando essa convivência. Geralmente o ACL é uma camada serviços que irá fazer a comunicação entre sistemas na qual temos algumas implementações para a tradução de modelos do sistema antigo para o novo utilizando alguns padrões de projetos como *Facade* e *Adapter*[24].

Figura 5.

[24] Disponível em:< https://home.cs.colorado.edu/~kena/classes/5448/f12/lectures/08-facadeadapter.pdf>. Acesso em: 18/10/2022



Fonte: Elaboração própria

"Adapter é um padrão de design estrutural que permite que objetos com interfaces incompatíveis colaborem"[25]. Basicamente é o padrão que irá juntar as informações dos sistemas diferentes e fazer a tradução. Já o "*Facade* é um padrão de projeto estrutural que fornece uma interface simplificada para uma biblioteca, uma estrutura ou qualquer outro conjunto complexo de classes"[26]. Em resumo, é criar uma interface para diminuir a complexidade do monolito. Sobre a questão, Eric Evans diz:

> Quando sistemas baseados em modelos diferentes são combinados, a necessidade de que o novo sistema se adapte à semântica do outro sistema pode levar a uma corrupção do próprio modelo do novo sistema[27].

Na migração aqui proposta, usamos a arquitetura *serverless,* mas vale questionar até onde é válido segui-la sem servidor? Hoje a lambda tem TPS de até no máximo dez mil requisições por segundo, se a sua aplicação precisar ter um número maior ou próximo disso é bom seguir por outro caminho. Outro ponto relevante que foi observado: sobre os custos dos serviços Lambda.

Conforme consta no site do *Amazon.com*, com o AWS *Lambda*, paga-se somente o que for usado; cobra-se pelo número de solicitações e durações de suas funções, hoje cada chamada de um lambda custa em torno de vinte centavos de dólar por 1 milhão de solicitações[28] e 0,0000000333 de USD em média por duração a cada milissegundo, conforme tabela[29] abaixo.

---

[25] Disponível em: <https://refactoring.guru/design-patterns/adapter>. Acesso em 10/10/2022

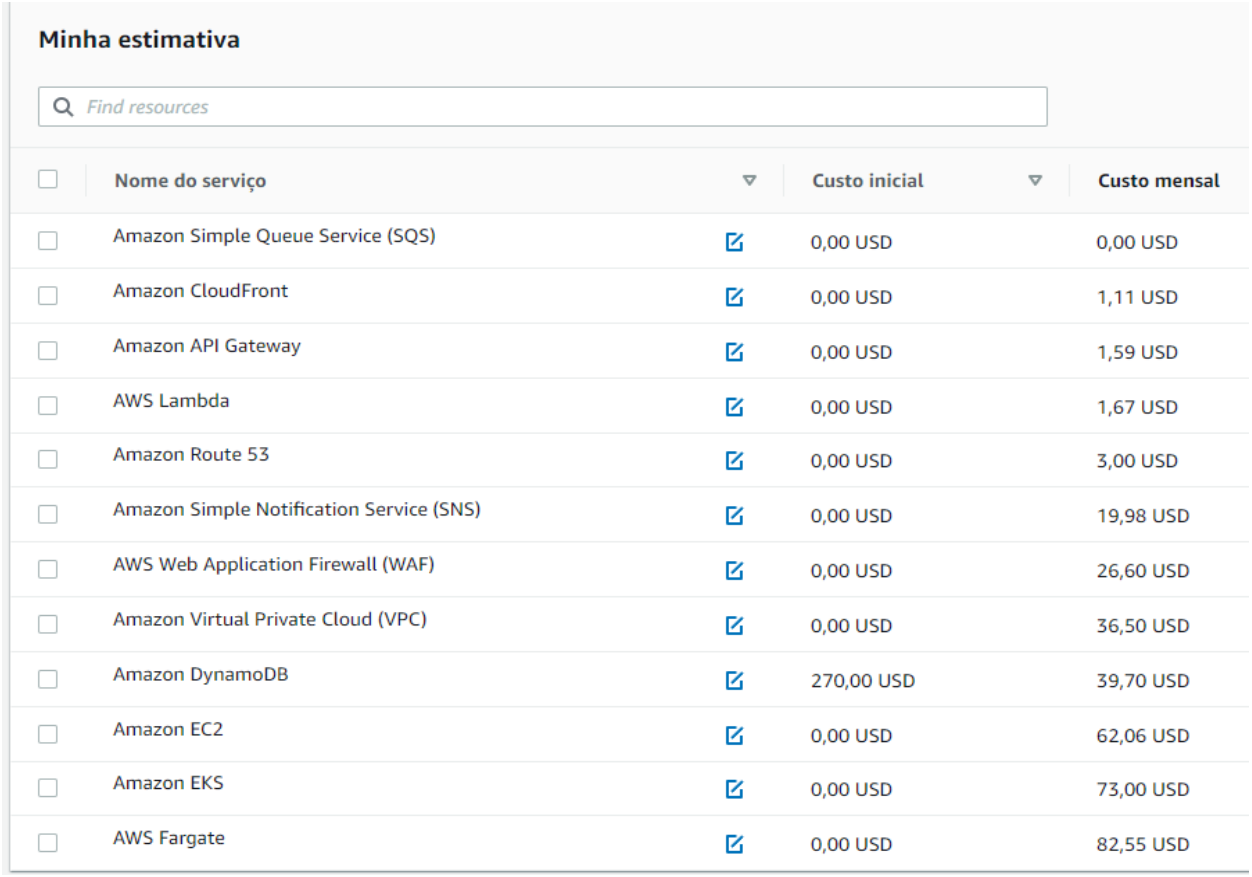
[26] Disponível em: < http://refactoring.guru/design-patterns/facade>. Acesso em 10/10/2022

[27] Do original: "When systems based on different models are combined, the need for the new system to adapt to the semantics of the other system can lead to a corruption of the new system's own." C.f.: Eric Evans. Domain-driven design: talking complexity in the heart of software, 2004, p.348

[28] Disponível em: <https://aws.amazon.com/pt/lambda/pricing/>. Acesso em:10/10/2022

[29] Disponível em:https://calculator.aws/#/estimate?id=5a4002ba37d1a3b5cb0bcde4d50aa066fc73ea19.Acesso em 18/10/2022

Figura 6.



Minha estimativa

Find resources

Nome do serviço	Custo inicial	Custo mensal
Amazon Simple Queue Service (SQS)	0,00 USD	0,00 USD
Amazon CloudFront	0,00 USD	1,11 USD
Amazon API Gateway	0,00 USD	1,59 USD
AWS Lambda	0,00 USD	1,67 USD
Amazon Route 53	0,00 USD	3,00 USD
Amazon Simple Notification Service (SNS)	0,00 USD	19,98 USD
AWS Web Application Firewall (WAF)	0,00 USD	26,60 USD
Amazon Virtual Private Cloud (VPC)	0,00 USD	36,50 USD
Amazon DynamoDB	270,00 USD	39,70 USD
Amazon EC2	0,00 USD	62,06 USD
Amazon EKS	0,00 USD	73,00 USD
AWS Fargate	0,00 USD	82,55 USD

Fonte: Elaboração própria

Portanto, entende-se que se o sistema tiver um alto número de requisições por segundo e uma longa duração da execução do lambda, usar a arquitetura *serverless* pode não ser a melhor opção.

Como pode ser constatado, além dos benefícios citados, a arquitetura *serveless* é válida na redução de custos, conforme se observa na figura (6), essa arquitetura tem custo baixo, para sistemas de médio e pequeno porte ou de uso esporádico é uma arquitetura bem interessante em se tratando de custos. Conforme já citado, a Lambda oferece escalabilidade e disponibilidade sem maiores esforços, sendo uma ótima escolha para esse trabalho. Outro ponto utilizando serviços disponibilizados pela AWS como SQS, conseguimos manter baixo custo e resolver problemas de alta requisições no sistema, assim garantimos de forma assíncrona a integridade da solução e caso tenhamos problemas já temos o ambiente de monitoramento da AWS para nos ajudar a entender os problemas, como *Cloud Watch* e Cloud *Trail e X-ray.*

- O AWS X-Ray rastreia as solicitações do usuário enquanto percorrem todo o aplicativo. Ele agrega os dados gerados por serviços e recursos

individuais que compõem o aplicativo, oferecendo uma visão completa do seu desempenho.

- O AWS *CloudTrail* monitora e registra a atividade da conta por toda a infraestrutura da AWS, oferecendo controle sobre o armazenamento, análise e ações de remediação.

- O Amazon *CloudWatch* permite coletar, acessar e correlacionar esses dados em uma única plataforma a partir de todos os seus recursos, aplicações e serviços da AWS em execução nos servidores da AWS.

Com isso concluímos que a migração para a nuvem por ser benéfica em vários sentidos listados no trabalho apresentado.

5 CONSIDERAÇÕES FINAIS

Monolito é um sistema com vários elementos trabalhando em uma única unidade implantável de execução.

Esses sistemas começam pequenos, mas tendem a crescer à medida que novos elementos são agregados. Entretanto, com o passar do tempo e com a adição de novos recursos o sistema monolítico pode começar a apresentar problemas resultantes de algumas dificuldades tais como, de acesso para plena manutenção, na questão da resiliência e até mesmo no quesito vulnerabilidade.

Para a pesquisa e desenvolvimento deste trabalho usou-se os recursos oferecidos pelas ferramentas que a AWS disponibiliza, possibilitando a elaboração de uma arquitetura *serverless* flexível.

O trabalho de análise e entendimento do sistema legado seguindo os conceitos apresentados no trabalho, fizemos o desenho de uma nova solução e fizemos a implementação de um novo modelo.

No presente trabalho observou-se que a alteração do sistema ficou mais ágil, pois este ficou dividido em vários microsserviços, além de o sistema ter ficado desacoplado possibilitando a novas expansões no futuro.

Por fim, cabe lembrar que a arquitetura de software e soluções estão em plena expansão. Novos conceitos, técnicas e ferramentas mudam ou surgem a todo momento. Com este trabalho esperamos contribuir com estudos futuros, apresentando como atualmente a AWS pode auxiliar nessa transformação.

REFERÊNCIAS

BERTALANFY, L. Von. **Teoria Geral dos Sistemas**. Rio de Janeiro: Ed. Vozes, 1975.

BRANDOLINI, Alberto. **Introducing Event Storming**. Disponível em: http://ziobrando.blogspot.com/2013. Acesso em:19/09/2022

CARDOSO; MEFFE; MARTINS, S.P. O software Público Brasileiro. Revista Linux Magazine, nº6, 2011.

EVANS, Eric. **Domain-driven design**: talking complexity in the heart of software. Ed.: Addison-Wesley. 2004.

FERNANDES, Henrique Marques. **O que é sistema/aplicação Monolito/Monolítica?** Disponível em:https://marquesfernandes.com/tecnologia/o-que-e-um-sistema-aplicacao-monolito-monolitica/ Acesso em: 15/07/2022

FIGUEIREDO, Elaine G. M. de. **Arquiteturas de sistemas Web 3.0**. 2013. Disponível em: https://www.devmedia.com.br/arquiteturas-de-sistemas-web-3-0. Acesso em:15/07/2022

GERTEL, Lucas. **Padrões de Arquitetura Web**: Monolítica ou Micro Serviços?. 2019. Disponível em: <https://medium.com/@lgertel/padrões-de-arquitetura-web-monolítica-ou-micro-serviços-7b3f0c9394fe>. Acesso em:27/08/2022

GIROLDO, Bruna. **Mercado da arquitetura de software. 17**, Novembro 2020 Disponível em: <https://posdigital.pucpr.br/blog/mercado-da-arquitetura-de-software>. Acesso em:27/08/2022

HAQ, Siraj Ul. **Introduction to Monolithic Architecture and MicroServices Architecture**. 2018. Disponível em: <https://medium.com/koderlabs/introduction-to-monolithic-architecture-and-microservices-architecture-b211a5955c63>. Acesso em:27/08/2022

LEWIS, James; FOWLER, Martin. **Microservices: a definition of this new architectural term.** Disponível em:<https://martinfowler.com/articles/microservices.html>. Acesso em:27/08/2022

LIMA, Gustavo Matozinho. **CAPI**: uma abordagem de desenvolvimento usando tecnologias serverless . Disponível em:<https://https://repositorio.ifes.edu.br/handle/>. Acesso em:13/09/2022

MENDES, Antonio. **Arquitetura de Software**: Desenvolvimento orientado para arquitetura. Disponível em: <https://www.devmedia.com.br/arquitetura-de-software-desenvolvimento-orientado-para-arquitetura/8033>. Acesso em:27/08/2022

NEVES, Joana. **História geral** – a construção de um mundo globalizado. Ed. Saraiva, 2020.

NEWMAN, Sam. **Building Microservices**. 1. Ed. O'Reilly Media, Inc, Gravenstein Highway North, Sebastopol, USA, 2015. Cap. 2, p. 19.

NEWMAN, Sam. Migrando sistemas monolíticos, para micro serviços. Ed.Nova tec, 2020.

Revista de engenharia de software magazine 63. Disponível em:<https://www.devmedia.com.br/revista-engenharia-de-software-magazine-63/29391>**.** Acesso em:27/08/2022

SIRQUEIRA, Tassio; DALPRA, Humberto. **NoSQL e a Importância da Engenharia de Software e da Engenharia de Dados para o Big Data.** Disponível em: <https://www.researchgate.net/publication/327035187> Acesso em: 18/10/2022

ZANELATTO, Alexandre Davi. **Arquitetura Serverless Baseada em Eventos Para Aplicações WEB Utulizando a AWS.** Disponínel em <https://repositorio. animaeducacao.com.br/bitstream/ANIMA/10963/2/TCC>. Acesso em:13/09/2022.