Universidade Federal de Pernambuco

Centro de informática

Graduação em Ciência da Computação

Migrando uma Arquitetura Monolítica para uma Arquitetura de Microsserviços Serverless

Matheus José Mota de Oliveira

Trabalho de Graduação

Recife, Março de 2024

Universidade Federal de Pernambuco

Centro de informática

Graduação em Ciência da Computação

Matheus José Mota de Oliveira

Migrando uma Arquitetura Monolítica para uma Arquitetura de Microsserviços Serverless

Trabalho apresentado ao Programa de Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Vinicius Cardoso Garcia

Recife, Março de 2024

Ficha de identificação da obra elaborada pelo autor,
através do programa de geração automática do SIB/UFPE

Oliveira, Matheus José Mota de.
Migrando uma arquitetura monolítica para uma arquitetura de microsserviços serverless / Matheus José Mota de Oliveira. - Recife, 2024.
48 : il., tab.

Orientador(a): Vinicius Cardoso Garcia
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Pernambuco, Centro de Informática, Ciências da Computação - Bacharelado, 2024.
Inclui referências.

1. Cloud Computing. 2. Microsserviços . 3. Engenharia de Software. 4. Serverless. 5. AWS Lambda. I. Garcia, Vinicius Cardoso. (Orientação). II. Título.

000 CDD (22.ed.)

Matheus José Mota de Oliveira

Migrando uma Arquitetura Monolítica para uma Arquitetura de Microsserviços Serverless

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para obtenção do título de bacharel em Ciência da Computação.

Aprovado em: 20 / 03 / 2024

Banca Examinadora:

_____

Prof. Dr. Vinicius Cardoso Garcia (Orientador)

Universidade Federal de Pernambuco

_____

Prof. Dr. Carlos Andre Guimaraes Ferraz (Examinador Interno)

Universidade Federal de Pernambuco

Agradecimentos

Primeiramente gostaria de agradecer imensamente meus pais, por sempre terem acreditado em mim, e por sempre mostrar como a educação e o estudo podem lhe levar mais longe.

A meu avô Laudo que achou que teria mais um engenheiro na família, mas que com certeza vibraria muito com minha formação como cientista da computação. Também aos meus avós Marilene, Vilmar e Maria, e a toda minha família pelo apoio.

A Julia Malheiros, por todo companheirismo e apoio, mesmo nos momentos mais difíceis, quando a conclusão desse ciclo ainda parecia algo distante.

Agradeço profundamente a todo o corpo docente do Centro de Informática da Universidade Federal de Pernambuco, pela dedicação no que faz no propósito de ensinar e criar profissionais mais preparados. Em especial, agradecer ao meu orientador Vinicius Garcia, pelo acompanhamento nesse trabalho, e por todo apoio prestado para que esse trabalho de graduação saísse.

Por último, agradecer a todas as pessoas que passaram por mim no tempo que estive na graduação, e que de certa forma me influenciaram a chegar onde estou, e tornaram o processo mais prazeroso.

Resumo

Apesar dos sistemas monolíticos possuírem certas vantagens, há também algumas desvantagens significativas. Um dos principais desafios associados a esses sistemas é o alto grau de acoplamento entre suas funcionalidades, o que acaba por dificultar a escalabilidade do projeto [1]. Adicionalmente, é comum observar que sistemas monolíticos resultam em um aumento no tempo de execução do programa [2].

Uma estratégia para contornar esses problemas é considerar a migração para uma abordagem baseada em microsserviços serverless [3], aproveitando a integração entre esse padrão arquitetural com a AWS Lambda [4], gerando a possibilidade de trabalhar com o programa como um FaaS (Function-as-a-Service) [15].

Essa transição permite uma maior flexibilidade na arquitetura do sistema, desacoplando as funcionalidades e, assim, facilitando a escalabilidade. Além disso, a utilização de serviços serverless como a AWS Lambda, um serviço de *Cloud Computing* [5], pode otimizar o desempenho e reduzir custos operacionais.

A adoção de uma arquitetura serverless representa uma solução promissora para superar desafios enfrentados pelos sistemas monolíticos, oferecendo maior agilidade, escalabilidade e eficiência no tempo de execução do programa.

Palavras-chave:

Sistemas Monolíticos, migração, AWS lambda, Cloud, microsserviços, serverless

Lista de tabelas

Tabela 1 - Experimento 1 30

Tabela 2 - Experimento 2 31

Tabela 3 - Experimento 3 31

Lista de figuras

Figura 1 - Mercado de serviços de infraestrutura em nuvem, Q1 11

Figura 2 - Mercado de serviços de infraestrutura em nuvem, Q2 2023 12

Figura 3 - Linguagens de programação mais utilizadas 18

Figura 4 - AWS Lambda Cold Start 19

Figura 5 - Diagrama do sistema proposto 21

Figura 6 - Layers da lambda function 24

Figura 7 - GQM do trabalho 29

Figura 8 - AWS Pricing Calculator comparação 32

Sumário

1. Introdução **6**
1.1 Contexto e motivação 6
1.2 Objetivos 8
1.3 Limitações do estudo 8
1.4 Organização do trabalho 9
2. Revisão da Literatura **10**
2.1 Conceitos fundamentais 10
2.1.1 Cloud Computing e AWS 10
2.1.2 Arquitetura de software 12
2.1.3 Serverless 13
2.1.4 Function-as-a-Service (FaaS) 13
2.1.5 Event-driven Architecture 14
2.2 Análise do Estado da Arte 15
2.3 Sumário do Capítulo 16
3. Projeto **17**
3.1 Tecnologias Utilizadas 17
3.1.1 Provedor Serverless 17
3.1.2 Linguagem de Programação 18
3.1.3 Framework Serverless 19
3.2 Solução Proposta 20
3.2.1 Introdução 20
3.2.2 Migração para Lambda 20
3.2.3 Configuração da lambda 22
3.3 Sumário do Capítulo 24
4. Vendor Lock-in **25**
4.1 Introdução 25
4.2 AWS Lambda e Serverless Providers 25
4.4 Sumário do Capítulo 27
5. Metodologia, Experimentos e Resultados **28**
5.1 Goal Question Metric (GQM) 28
5.2 Experimento 29
5.3 Resultados e Análise 30
5.4 Discussão 33
5.5 Sumário do Capítulo 34
6. Conclusão e Trabalhos Futuros **35**
6.1 Conclusão 35
6.2 Possíveis limitações 36
6.3 Trabalhos Futuros 37
7. Referências **39**

1. Introdução

O foco deste Trabalho de Graduação concentra-se na concepção de padrões de arquitetura de software inovadores voltados para aplicações baseadas em microsserviços, com a utilização da AWS Lambda como um FaaS [15].

Este trabalho também explorará a integração da arquitetura serverless com microsserviços, examinando como a AWS Lambda pode ser aproveitada dentro dessa abordagem. Serão investigados os benefícios e desafios associados à adoção da arquitetura serverless para microsserviços, incluindo melhorias no tempo de compilação de funcionalidades, na escalabilidade da aplicação e redução da sobrecarga operacional.

1.1 Contexto e motivação

A comparação entre as arquiteturas monolítica e de microsserviços [6] é uma discussão que continua em constante evidência no campo da engenharia de software [7]. Em meio às diversas opções de arquiteturas de software disponíveis, essas duas abordagens continuam a atrair a atenção de muitas empresas, que buscam determinar o padrão de projeto mais adequado a ser adotado.

No passo que grandes empresas como Netflix, Linkedin, Uber, Amazon, entre outras, migraram seus sistemas monolíticos para microsserviços [8], e mostram as melhorias adquiridas com isso, os mais imediatistas já julgam essa arquitetura como a melhor a ser utilizada independente do sistema que a empresa possua. Porém, se tem visto na contramão dessa tendência muitas empresas voltando para o padrão monolítico [9].

O que se tira como conclusão é que não há uma arquitetura melhor a ser seguida, e sim a que mais se encaixa nas necessidades específicas do projeto e da empresa. A migração de uma arquitetura ou outra, ou a sua implantação, deve vir acompanhada de estudo e planejamento [10], e até de testes e comparações, para se checar qual a mais vantajosa.

Um fator que anda lado a lado da arquitetura do projeto que será escolhido é a decisão sobre o servidor em que ele estará configurado. Como o custo de manutenção e implantação de servidores físicos é algo extremamente elevado, muitas empresas têm escolhido utilizar servidores em provedores de Cloud Computing, uma vez que fica a cargo deles obrigações como manutenibilidade, provisionamento e segurança. Dentre os principais líderes de mercado estão a AWS (Amazon) [11], que será a provedora utilizada nesse projeto, Azure (Microsoft) [12] e Google Cloud (Google) [13].

A vantagem de se utilizar a Cloud Computing vai além, uma vez que as provedoras de cloud geram constantemente novas funcionalidades para lhe ajudar a escalar o seu negócio. Numa mesma plataforma você consegue acesso a máquinas virtuais, armazenamento de arquivos, gerenciadores de bancos de dados, sistemas de notificação e monitoramento para suas aplicações, deploy de Function-as-a-Service (FaaS), e diversas outras.

Por outro lado, como falaremos mais adiante no capítulo 4, nem sempre manter toda a sua infraestrutura em uma única provedora, será algo bom, pois pode levar a um Vendor Lock-in [46], termo utilizado para definir uma situação em que um cliente fica preso a um único fornecedor devido a dependências tecnológicas ou contratuais, tornando difícil ou custoso mudar para outra solução, sendo assim algo necessário de atenção na hora de escolher a provedora e a arquitetura do seu sistema.

Com a pandemia, muitas empresas tiveram que procurar formas de ter seus dados e sistemas em algum lugar que não fosse um servidor físico, e que lhes oferecessem também segurança e integridade nas suas informações. Com a cloud em ascensão já nessa época muitas empresas viram nela um solucionador para esses problemas, com a possibilidade de criação de uma rede de fácil acesso e segura aos seus colaboradores [14], onde você consegue configurar cada usuário para acessar apenas as informações e funcionalidades a que ele for concedida.

Com essa expansão dos ambientes cloud surgiu o conceito de serverless, onde a orquestração da infraestrutura fica a cargo dos provedores de serviço, visando eliminar tarefas de gerenciamento e configuração pelos usuários, e permitindo que se concentrem exclusivamente na aplicação. Juntamente a esse conceito, com o tempo se originou o de Function-as-a-Service, que visava oferecer a possibilidade de deploy de funções em um ambiente em nuvem. utilizando-se de uma arquitetura event-driven, onde as funções são ativadas por eventos, como por exemplo solicitações http.

Citamos aqui alguns exemplos de empresas de tecnologias que migraram seus serviços de monolíticos para microsserviços, reforçando uma tendência nas empresas. Porém, temos exemplos em outras áreas que utilizam essa abordagem, como a Walmart, do setor de varejo [55] e Adidas, do setor de roupas [56]. Temos ainda a McDonald's, uma empresa do ramo alimentício, que juntou essa arquitetura com o potencial da arquitetura serverless, para desenvolver um sistema de recompensas para os seus usuários [57], mostrando que a computação em nuvem está presente em muitas áreas distintas atualmente, se tornando uma necessidade global.

1.2 Objetivos

Como objetivo deste trabalho temos o de promover a migração de um sistema monolítico para um ambiente serverless e analisar as diferenças entre um sistema compilado em uma arquitetura monolítica em um servidor local, e esse mesmo sistema em um ambiente serverless.

Visamos por meio dessa migração garantir melhorias nas métricas predefinidas, resultando, ao final do processo, em um sistema capaz de extrair de maneira mais eficaz o potencial de uma regra de negócio por trás dele. Para alcançar esses objetivos, são necessários:

- Analisar a funcionalidade de um serviço a ser migrado, a fim de identificar ganhos com a migração para serverless.
- Criação de métricas para identificar os ganhos, e possíveis perdas, com a migração.
- Analisar a arquitetura do sistema antes e após a migração.

1.3 Limitações do estudo

Ao longo do trabalho iremos indicar alguns dos limitadores, que devem ser levados em consideração quando se planeja realizar uma migração entre as arquiteturas, ou que impossibilitaram uma análise mais detalhada e direta entre elas

Entre esses limitadores está o fato de serverless ser uma abordagem relativamente nova, o que gera algumas lacunas em termos de estudos complexos sobre a área. Ainda em cima desse ponto, por ser relativamente nova, ainda não é uma alternativa tão difundida quanto outras, e por possuir alguns desafios técnicos relacionados, pode ser custoso decidir mover para ela com um time que ainda não tenha familiaridade com a ferramenta.

Alguns dos desafios técnicos destacados são a baixa taxa de upload em buckets do s3, comentado na seção 5.3, e que por ser algo central no projeto apresentado no capítulo 3, acabou por limitar uma comparação mais direta entre as arquiteturas, uma vez que ele agiu como um fator externo não previsto quando se foi pensadas as métricas que o trabalho seguiria.

Outra limitação técnica apresentada, é a dificuldade de criação de layers, para a execução de libs externas as que já vem por padrão nas linguagens de programação, o que pode gerar o Vendor Lock-in, aqui já comentado na seção 1.1, bem como aumentar a complexidade dessa abordagem, aumentando o nível técnico necessário para sua implementação.

Esses pontos citados serão discutidos com mais ênfase durante o trabalho, porém, são fatores que devem ser levados em conta na hora de planejar uma migração para serverless.

1.4 Organização do trabalho

- Capítulo 2: Conceitos e tecnologias essenciais para a compreensão deste trabalho.
- Capítulo 3: Apresenta o projeto, suas concepções, tecnologias utilizadas e configurações do mesmo.
- Capítulo 4: Discorre sobre o tema Vendor Lock-in dentro do ambiente Cloud.
- Capítulo 5: Apresenta a metodologia escolhida para o trabalho, assim como as métricas, experimentações e resultados obtidos.
- Capítulo 6: Conclusão, dificuldades encontradas e trabalhos futuros.
- Capítulo 7: Referências bibliográficas.

2. Revisão da Literatura

Este capítulo tem como propósito abordar definições essenciais para a compreensão deste trabalho. Encontra-se dividido em duas seções: a primeira apresenta conceitos-chave relacionados à Cloud Computing e Engenharia de Software, e a segunda apresenta uma análise do estado da arte de Cloud Computing, e o contexto em que o sistema proposto está inserido. Com esses conceitos, conseguiremos entender melhor a solução proposta no capítulo 3.

2.1 Conceitos fundamentais

Nesta seção, visamos esclarecer os conceitos fundamentais de Cloud Computing e Engenharia de Software, visando proporcionar ao leitor uma compreensão maior sobre esse estudo. Aqui selecionamos alguns dos conceitos mais presentes sobre esse tema.

2.1.1 Cloud Computing e AWS

Para entender melhor o que seria uma Cloud Computing, iremos apresentar o conceito desse tema dado pela Google Cloud para depois destrinchar melhor o tema. Apesar desse projeto focar o seu desenvolvimento no provedor de cloud computing AWS [16], é inegável que ambas possuem funcionalidades internas e propostas muito semelhantes.

"A computação em nuvem (cloud computing) é a disponibilidade sob demanda dos recursos de computação como serviços na Internet. Ela elimina a necessidade de as empresas adquirirem, configurarem ou gerenciarem a infraestrutura, assim elas pagarão apenas pelo que usarem." [17]

E o que seria essa disponibilidade sobre demanda? Nada mais é do que ter a infraestrutura necessária, e no tamanho necessário, para implantar o seu projeto. Essa proposta de negócio é conhecida como pay-as-you-go, uma vez que o seu negócio paga apenas pelo que utilizar do serviço cloud. Enquanto uma empresa que necessite de uma infraestrutura pequena e um servidor simples pagará pouco pelo serviço, uma empresa grande com uma infraestrutura robusta pagará um valor mais caro para a provedora.

Antes do surgimento das clouds, caso você necessitasse escalar o sistema de uma empresa você teria que investir caro em um servidor físico para isso. Caso a previsão de escalabilidade não desse os resultados esperados, o investimento no servidor seria perdido.

Em cima desse problema surge o conceito de Elasticidade [8] nas Clouds, que é a capacidade do usuário de alocar dinamicamente os recursos

necessários, com base na necessidade atual do seu sistema, e dimensionando seu sistema sob medida com relação à demanda proveniente, prevenindo assim gastos supérfluos com hardwares.

Como já dito antes, as principais provedoras são a AWS (Amazon), Azure (Microsoft) e Google Cloud (Google). Na Figura 1 podemos ver que a essas 3 controlavam 65% do mercado atual de gastos com provedoras de Cloud Computing no primeiro trimestre do ano. Na Figura 2 conseguimos notar que essa porcentagem se mantém entre essas três, havendo apenas uma diferença na porcentagem da microsoft e google. Já na figura três conseguimos observar uma comparação entre esses gastos num período maior, do último trimestre de 2017, até o segundo trimestre de 2023.

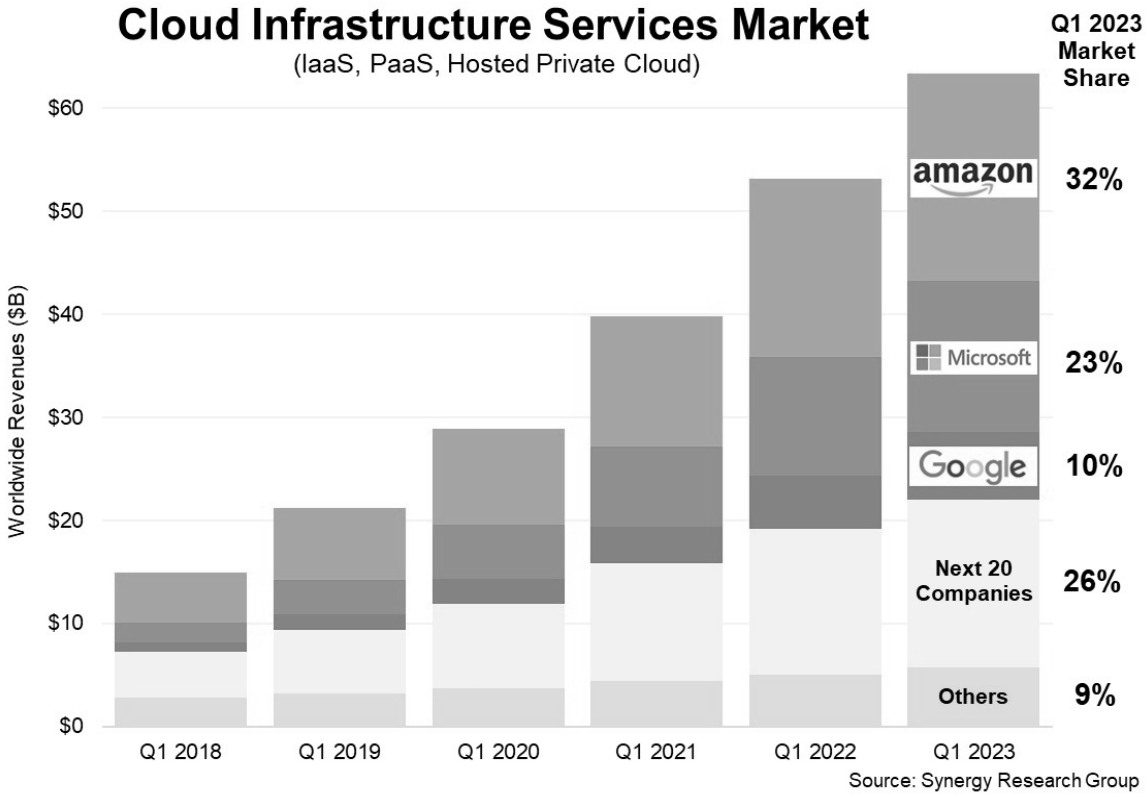


Figura 1 - Mercado de serviços de infraestrutura em nuvem, Q1[61]

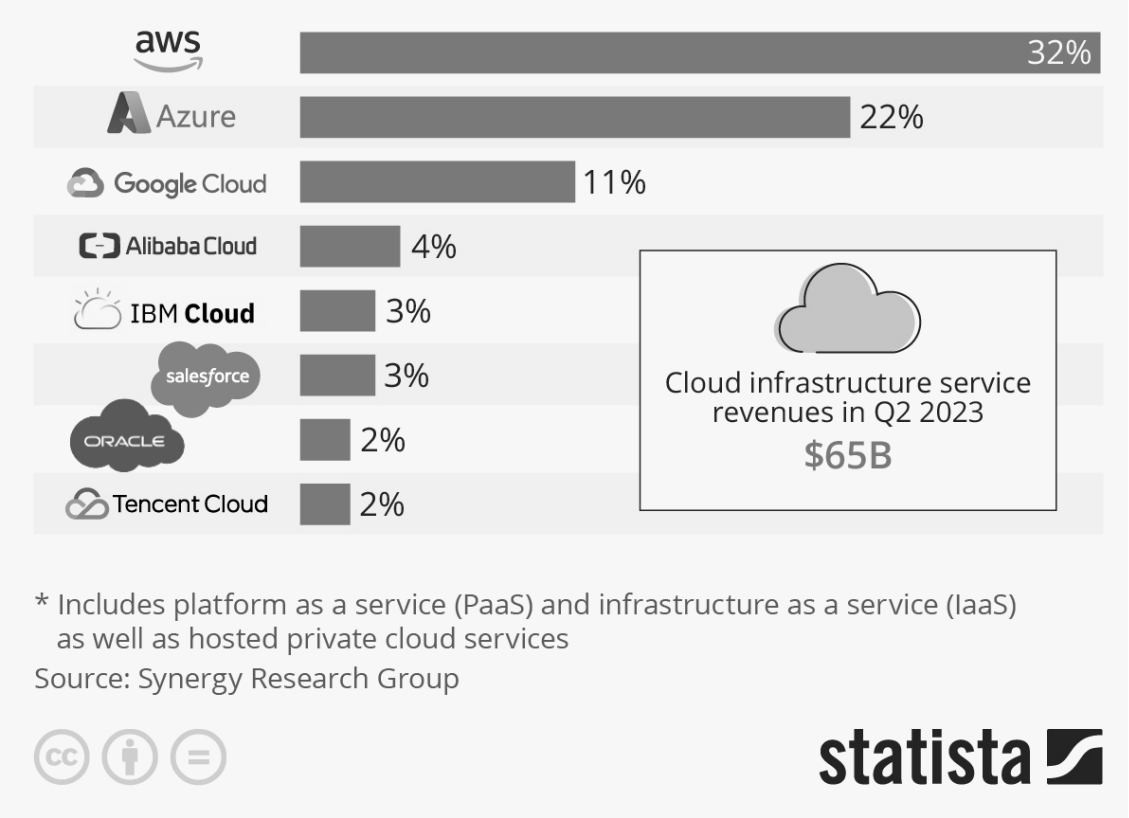


Figura 2 - Mercado de serviços de infraestrutura em nuvem, Q2 2023

2.1.2 Arquitetura de software

Arquitetura de software refere-se à estrutura que um sistema apresenta, e serve como um esquema para a construção da aplicação e sua manutenibilidade. É definida como uma coleção de componentes computacionais juntamente com uma descrição das interações entre esses componentes [18].

A arquitetura de software envolve tomar decisões de design para alcançar atributos específicos de qualidade, como desempenho, confiabilidade [19], escalabilidade, manutenibilidade e reusabilidade. Vemos assim uma necessidade em mapear o que é importante no projeto, para que seja dado o foco necessário para que esses elementos se mantenham em boa funcionalidade.

Para um desenvolvedor se tornar um arquiteto, ele precisa ser capaz de reconhecer quais elementos são importantes, reconhecendo quais elementos podem causar problemas sérios caso não seja dada a devida atenção.[20]

Dentro da arquitetura de software, encontramos uma sub-área destinada à padronização dos projetos, conhecida como padrões de arquitetura. Dentro desses padrões estão os já falados nesse artigo, microsserviços e monolíticos,

porém existem diversos outros, entre eles o Orientado a Eventos (Event-Driven).

2.1.3 Serverless

É um modelo de computação, que permite o desenvolvimento e execução de aplicativos em nuvem sem que a infraestrutura, e orquestração do projeto, necessite do gerenciamento do usuário, ficando a cargo do provedor de Cloud essa função.

A intenção desse modelo é fazer o desenvolvedor se preocupar unicamente com o código a ser escrito, que será implantado em contêineres gerenciados por um provedor de serviços em nuvem [21], que cuidará de todo o resto, como a manutenção rotineira da infraestrutura, monitoramento do sistema, alocação de capacidade computacional, entre outras responsabilidades.

Em Serverless, normalmente o preço é baseado no tempo de execução e nos recursos necessários para a uma função ser executada. Os desenvolvedores não pagam por aplicações ociosas, mesmo que ela esteja hospedada em um provedor, mas de forma inativa.

2.1.4 Function-as-a-Service (FaaS)

Faas [15] é um serviço de computação na nuvem, onde funções individuais são executadas em resposta a eventos específicos, como por exemplo o upload de um arquivo em um bucket s3 na AWS ou um usuário realizando alguma ação em um site. Essas ações fazem com que a função seja chamada e executada.

Entre os benefícios dessa abordagem estão:

- Falta de necessidade de gerenciamento de infraestrutura, que fica a cargo do provedor.
- Execução de funções dimensionadas de forma dinâmica e automática, quando a demanda cai, instantaneamente o serviço diminui suas chamadas.
- Pagar apenas pelo uso das funções, onde se uma função é criada porém nunca é chamada, o desenvolvedor não pagará nada por ela, ou quase nada. Enquanto, uma chamada muitas vezes gera um custo maior.
- Inerentemente escalável, os desenvolvedores não precisam se preocupar em implementar medidas para lidar com tráfego

intenso ou alto uso da aplicação, fica a cargo do provedor cuidar de problemas relacionados ao dimensionamento.

Quanto ao fato de ser inerentemente escalável, isso pode se tornar um problema, uma vez que no FaaS você paga pelo uso da aplicação. Caso o desenvolvedor não configure um número máximo de execuções em paralelo para aquela função, se ela for chamada mais vezes do que o desenvolvedor está disposto a pagar por ela, gerando um custo elevado.

O termo Functions-as-a-Service (FaaS) e o conceito de Serverless são confundidos frequentemente [22], no entanto, a distinção reside no fato de que Serverless é empregado para se referir a qualquer categoria de serviço, abrangendo funções, armazenamento, banco de dados, gateways de API, entre outros. Já o FaaS, embora seja possivelmente a tecnologia central em arquiteturas sem servidor, foca no paradigma da computação orientada a eventos, sendo assim o FaaS um subconjunto do Serverless.

2.1.5 Event-driven Architecture

Uma arquitetura orientada a eventos [23] é um padrão de software baseado em três componentes fundamentais: os produtores, os roteadores (também conhecidos como brokers), e os consumidores de eventos. O produtor de eventos é responsável por transmitir um evento para um intermediário, o roteador de eventos, que mantém uma ordem cronológica do evento, seguindo o modelo de fila, ou FIFO (First in, first out) [24], normalmente. Por último, o consumidor processa o evento, sem necessidade que seja em tempo real, para acionar outra ação.

Eventos são ações que ocorrem dentro ou fora de uma organização, como por exemplo interações de clientes com uma plataforma, modificações no status de projetos ou notificações de sistemas. Portanto o ponto chave dessa arquitetura está em como receber, processar e consumir tais ações.

Um ponto de benefício oferecido por esse padrão é o desacoplamento entre serviços diferentes, assim o consumidor e o produtor conhecem unicamente o roteador nesse processo, e não conhecem um ao outro. Essa arquitetura implementa o conceito de comunicação assíncrona [26], o que implica que o remetente e o destinatário não necessitam aguardar um ao outro para avançar para a próxima tarefa, fazendo com que se um dos serviços no processo em algum momento falhar, não será impactada a funcionalidade do outro.

2.2 Análise do Estado da Arte

Em um ambiente empresarial com diversas áreas interligadas, que dependem de um sistema central para obter informações, é crucial entender que um gargalo no início da execução desse sistema pode resultar em um aumento significativo no tempo necessário para a realização de tarefas. Cada área possui um propósito distinto para com as informações desse sistema, tornando essencial a otimização do processo inicial para garantir uma maior eficiência e fluidez na condução das atividades.

A partir do momento, que esse processo inicial é a interpretação e extração de informações de um arquivo, mais propriamente dito, contas de energia, um sistema cada vez mais independente da execução manual e com a capacidade de concorrência, poderia otimizar significativamente o tempo dedicado a essa atividade. Se olharmos ainda mais a fundo o problema, devemos fazer uma comparação entre as duas arquiteturas centrais desse trabalho de graduação, a monolítica e a serverless.

Com a utilização da AWS Lambda e suas triggers conseguimos avançar nessa direção, uma vez que ela nos permite a criação de funções, que serão executadas de forma independente em resposta a chamadas do sistema. Tais chamadas podem ser simples ações, como o upload de um arquivo PDF de uma conta de energia em um bucket no Amazon S3 [25], até gatilhos acionados de uma função para outra, possibilitando execuções em paralelo.

Foi essa característica da Lambda, que fez a McDonald's, citada na seção 1.1, optar por essa tecnologia como uma solução para o sistema anterior. Com isso, eles conseguiram evitar uma execução constante de um microsserviço dedicado, que passava longos períodos sem chamadas, por uma abordagem event-driven, conseguindo também abater o número de erros do sistema.

Em uma empresa que possui um número grande de clientes, cresce também o número de contas a serem lidas, as vezes exponencialmente, pois um cliente pode possuir mais de uma conta. Se pensarmos em um sistema monolítico com execução linear desse processo, o tempo para que essas operações sejam realizadas se torna alto. Nesse cenário, as operações seriam realizadas em etapas, onde cada função ou processo só seria chamado após a conclusão do anterior. Para ilustrar, inicialmente, realizaríamos a leitura de todas as contas, depois extrairíamos as informações e as escreveríamos no banco de dados, somente após essa etapa, procederíamos para o próximo serviço, como o encarregado de aplicar as regras de negócio da empresa, por exemplo.

Com a lambda você consegue gerenciar facilmente o número de threads que uma determinada função pode criar, como iremos mostrar na seção 3.2.3,

evitando um possível custo elevado da aplicação, e tudo isso sem precisar mudar o código desenvolvido no sistema anterior. Nesse caso descrito, ao realizar o upload de 100 arquivos no s3, com 100 threads disponíveis nessa lambda, todas as contas seriam lidas em paralelo, permitindo também a execução em paralelo de funções que se comunicam com esse sistema, conseguindo reduzir bastante o tempo dedicado a essas operações.

2.3 Sumário do Capítulo

Nesse capítulo abordamos conceitos fundamentais relacionados à Cloud Computing, Engenharia de Software e tecnologias emergentes como Serverless e Event-driven Architecture. A análise do Estado da Arte destaca a aplicação prática desses conceitos, especialmente na otimização de processos empresariais, apontando a relevância da adoção de tecnologias como AWS Lambda para aumentar a eficiência e reduzir custos operacionais.

No capítulo a seguir, apresentamos a solução proposta para mitigação de problemas operacionais, bem como as tecnologias utilizadas para o projeto, e as configurações iniciais necessárias.

3. Projeto

Este capítulo detalha as tecnologias e a solução proposta para a transição de um sistema monolítico para uma abordagem serverless. São discutidas as decisões relacionadas às tecnologias que serão utilizadas, e é apresentada uma visão geral do projeto que será desenvolvido.

3.1 Tecnologias Utilizadas

Nessa seção são apresentadas as ferramentas utilizadas para a implementação da solução proposta, oferecendo um contexto para apontar o porquê de sua escolha entre outras existentes.

3.1.1 Provedor Serverless

Como já dito anteriormente nesse trabalho de graduação, entre as principais provedoras de Cloud Computing estão a Amazon Web Services (AWS), Microsoft Azure e Google Cloud. Porém existem diversas outras com uma parcela significativamente menor do mercado, como por exemplo a IBM e a Oracle que mesmo sendo gigantes do setor de tecnologia não conseguem alcançar as concorrentes.

A liderança entre elas fica por parte da AWS, desde sua criação até os dias atuais. Não se pode negar que o fato de que ela foi a primeira empresa a apostar nesse tipo de mercado, com mais de 6 anos de diferença da segunda, a Azure, lhe bota em uma posição de destaque [27], mas a manutenção dessa posição ano após ano não se daria apenas por isso.

Entre possíveis fatores que justifiquem a manutenção dessa liderança está a sua relação com o cliente, pelo aumento sucessivo no fornecimento de seus serviços com base na quantidade de usuários, a vasta opção de ferramentas diferentes que o serviço oferece e a constante criação de novas ferramentas para ajudar diferentes clientes e segmentos de mercados.

Juntando essas qualidades com o fato de ela ser a líder de mercado, fazendo com que qualquer solução criada aqui se encaixe mais facilmente em diferentes empresas, escolhemos ela como provedora de Cloud, logo a AWS lambda como a ferramenta serverless a ser utilizada.

Visto que a linguagem que será utilizada no projeto tem suporte em todas as provedoras de serverless, a escolha ficou mais a cargo dos motivos citados acima.

3.1.2 Linguagem de Programação

Atualmente, existe uma ampla variedade de linguagens de programação, destacando-se Python, Java, Javascript e Typescript, estas duas últimas sendo opções viáveis para o ambiente de execução Node.js. Python em algumas pesquisas se mostra como a linguagem de programação mais utilizada atualmente, como podemos ver na Figura 3.

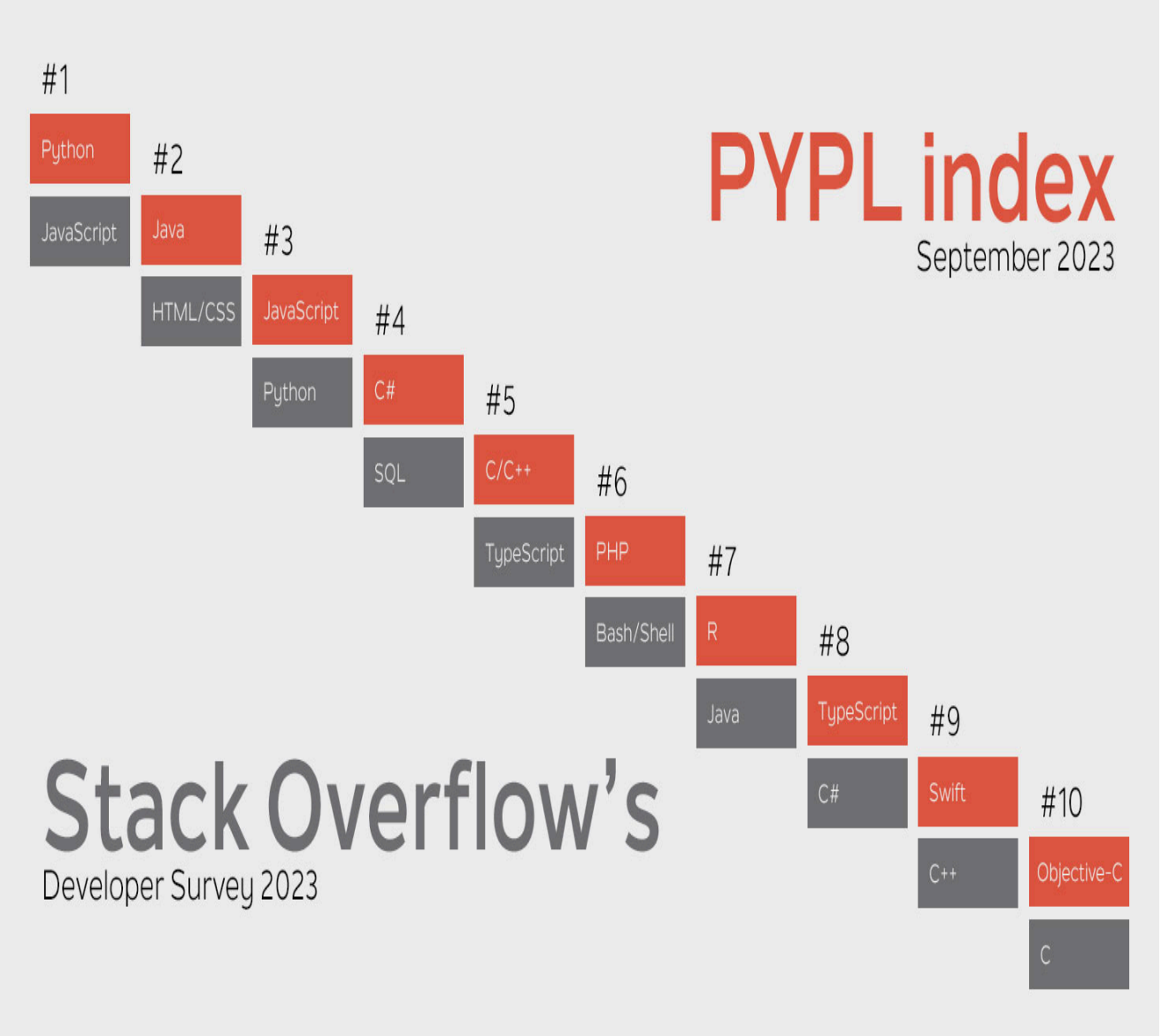


Figura 3 - Linguagens de programação mais utilizadas[62]

Com o tempo, observou-se uma mudança na preferência, e Python deixou de ser a linguagem de programação mais predominante nas funções Lambda, e foi passada pelo Node.js [28]. No entanto, em favor de Python, destaca-se a vantagem do menor tempo de inicialização (cold start) das instâncias, como evidenciado na Figura 4.

Figura 4 - AWS Lambda Cold Start

Portanto, ao considerar a posição de destaque que Python ocupa como uma das linguagens de programação mais utilizada, juntamente com o menor tempo de cold start em comparação com outros runtimes de Lambda, e o conhecimento prévio do autor nessa linguagem, optou-se por sua utilização no projeto.

3.1.3 Framework Serverless

Para simplificar a implementação de funções na AWS Lambda e permitir que os códigos sejam desenvolvidos localmente na IDE de preferência do desenvolvedor, adotamos o framework denominado "serverless" [29]. Este framework oferece suporte a várias linguagens e é compatível com diversos provedores de serviços serverless.

Por linha de comando é possível realizar chamadas ao framework, abrangendo desde a utilização de templates prontos na ferramenta, em diversas linguagens e para diversos propósitos, até o deployment da função na Lambda. Através de um arquivo de configuração do tipo yaml é possível definir propriedades da sua função, como a região de deploy, suas permissões, o número máximo de threads que a função pode criar, entre outras.

3.2 Solução Proposta

Nessa seção apresentamos o projeto que será desenvolvido para migração planejada e os benefícios esperados com a adoção dessa nova arquitetura. Bem como os problemas que levaram à idealização dessa solução.

3.2.1 Introdução

Para mitigar o desafio do gargalo operacional no processo de interpretação de informações associadas a contas de energia e na aplicação de regras de negócios correspondentes em um sistema empresarial, propomos a transição de um sistema monolítico implementado em Python, atualmente hospedado em um servidor local, para uma abordagem serverless utilizando o AWS Lambda. Esta mudança pretende não só melhorar a eficiência operacional, mas também mostrar a possibilidade de escalar esse sistema com base na solução proposta.

A intenção desse projeto é não apenas desenvolver uma arquitetura mais dinâmica e atual, implementando um padrão arquitetural onde anteriormente não tínhamos um claramente definido, mas também obter melhorias com essa migração.

Para ilustrarmos melhor o sistema que tínhamos antes da migração, é relevante destacar que não havia nele a adoção de algum padrão arquitetural específico, como o amplamente utilizado para projetos assim que é o Model-View-Controller [30]. Assim, toda lógica se encontrava em poucas classes, o que causava muita repetição de código, alto acoplamento das funções e também uma violação do princípio da responsabilidade única (Single Responsibility Principle) [31].

Por último, é válido constatar que o sistema anterior possuía a necessidade de compilamento manual, uma vez que não possuímos cron jobs para agendar a execução do código. Se levarmos em consideração o fato de que não obrigatoriamente uma pessoa que acessa e baixa uma conta de energia em uma empresa, vai ter o conhecimento necessário para compilar um código local, o processo como um todo se atrasa mais ainda.

3.2.2 Migração para Lambda

Com a lambda conseguimos resolver boa parte dos problemas citados acima, fazendo com que cada função lambda trabalhe como um microsserviço específico a sua funcionalidade.

Para ilustrarmos o potencial dessa migração escolhemos duas funções que antes se encontravam em um mesmo diretório do sistema, uma delas é referente a leitura das contas de energia em pdf, com utilização de algumas bibliotecas como a PyPDF2 [32] com utilização de regex para capturar as informações necessárias da conta a serem inseridas no banco de dados, e a outra é relativa ao cálculo do aluguel do cliente com desconto, e outras informações necessárias para a regra de negócio da empresa.

Abaixo na Figura 5, conseguimos visualizar o diagrama do sistema como ficaria após o processo de migração. Nela, nós temos primeiramente o processo de upload de um ou mais pdfs em um bucket no s3, que automaticamente já dispara um trigger para a primeira lambda, que chamamos de lateBillsReader, responsável pela leitura de contas de energia. Antes do fim da execução dessa função nós damos um invoke [33] para a outra função que seria a rentCalculation, responsável pelo cálculo de aluguel da energia solar derivada a essa conta contrato. Ambas as funções possuem chamadas para uma instância de banco de dados RDS [34] com Sql Server, como forma de manter a integridade das informações empresariais em um só lugar e de uso compartilhado, no caso da segunda função ela tanto possui comandos de SELECT para a primeira tabela, como de INSERT para a segunda.

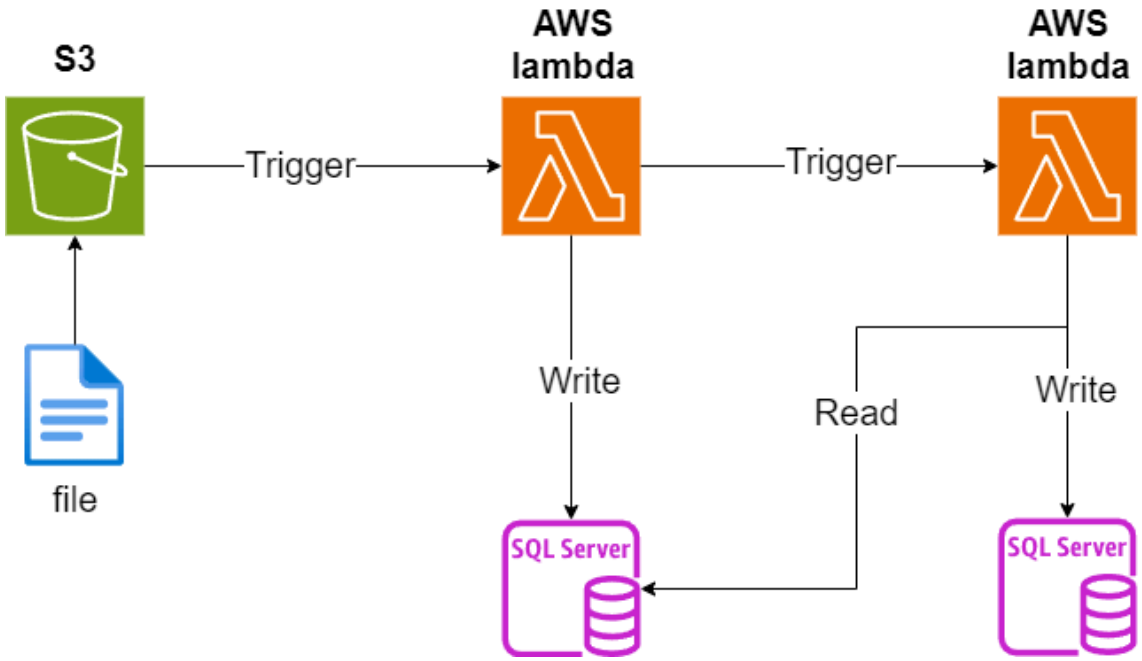


Figura 5 - Diagrama do sistema proposto

Para conseguirmos fazer com que a primeira função lambda invoque a segunda por meio de um trigger, nós devemos utilizar um API Gateway [35] para criar a comunicação entre as duas funções.

3.2.3 Configuração da lambda

Ao utilizar o framework serverless para simplificar o deployment de funções, contamos com um arquivo YAML [36] que permite configurar o serviço e as funções em desenvolvimento. Neste arquivo, podemos especificar detalhes como permissões, gatilhos e recursos necessários. Essa abordagem permite um gerenciamento mais simplificado das suas funções, e possibilita a escrita do código de forma local em alguma IDE.

Uma vez que o foco muitas vezes não é apenas o desempenho, e são levados em conta fatores como o custo da aplicação, nós devemos ter cuidado ao configurar nossas funções. Para que tenhamos um desempenho bom com esse sistema proposto, é importante possuir concorrência em sua execução, porém, se não definirmos o tamanho de máximo de memória que possa ser alocado por ela, as múltiplas threads pode causar um custo bastante elevado para a empresa, e não é isso que queremos com essa migração.

Abaixo podemos ver um arquivo de configuração do tipo yaml para esse serviço, onde limitamos o número de concorrência e a memória alocada pela aplicação, para que a solução além de ter um bom desempenho, tenha baixo custo. Essa foi a configuração inicial utilizada para o projeto.

```
service: late-bills
frameworkVersion: '3'
provider:
  name: aws
  runtime: python3.8
  region: us-east-1
  iamRoleStatements:
    - Effect: Allow
      Action:
        - s3:PutObject
        - s3:GetObject
        - s3:DeleteObject
        - s3:ListBucket
      Resource:
        - arn:aws:s3:::late-bills/*
        - arn:aws:s3:::late-bills
    - Effect: Allow
      Action:
        - lambda:InvokeFunction
      Resource:
        - arn:aws:lambda:us-east-1:[redacted]:function:late-bills-dev-rentCalculation
functions:
  lateBillsReader:
    handler: electric_bill_handler.late_bills_reader
    events:
      - s3:
          bucket: late-bills
          event: s3:ObjectCreated:*
      - http:
          path: source
          method: get
    reservedConcurrency: 40
    memorySize: 512
  rentCalculation:
    handler: rent_invoice_handler.rent_calculation
    reservedConcurrency: 40
    memorySize: 512
```

Outro ponto importante a ser comentado é que na lambda não temos a possibilidade de instalar diretamente libs das linguagens nela, precisamos utilizar essas libs como layers [37] do projeto, onde devemos instalar a lib de forma local e criar um zip com essa lib para que ela seja importada para o seu projeto. Em nosso projeto utilizamos duas layers, uma com o PyPDF2 para leitura de pdf, e a outra com pyodbc e Microsoft SQL Server [38] para as conexões com o banco de dados, como podemos ver na Figura 6.

Name	Layer version	Compatible runtimes	Compatible architectures
PyPDF2	1	python3.8	x86_64
pyodbc-mssql-final	1	python3.8	x86_64

Figura 6 - Layers da lambda function

Uma layer pode ser utilizada em mais de uma função, basta especificar qual layer, ou layers, aquela função deve usar. Todas as layers anteriormente criadas ficam armazenadas, a menos que o usuário opte por excluí-las. No sistema proposto, por exemplo, temos a utilização da layer pyodbc-mssql-final nas duas funções do serviço.

3.3 Sumário do Capítulo

Neste capítulo detalhamos as tecnologias e a solução proposta para a transição de um sistema monolítico para uma abordagem serverless. Inicialmente, são discutidas as decisões relacionadas às tecnologias utilizadas, incluindo o provedor serverless AWS Lambda, a linguagem de programação Python e o framework "Serverless". Em seguida, é apresentada uma visão geral do projeto a ser desenvolvido, abordando a migração planejada e os benefícios esperados com a adoção dessa nova arquitetura.

No próximo capítulo apresentamos melhor o conceito de Vendor Lock-in, e apontamos detalhes da necessidade de planejamento na criação de projetos em um provedor de Cloud Computing, tendo ele em mente.

4. Vendor Lock-in

Neste capítulo iremos explicar com mais detalhes o que seria o Vendor Lock-in [46], e como esse conceito influencia diretamente o planejamento na hora de escolher uma Cloud Provider.

4.1 Introdução

Um termo bastante utilizado no conceito das clouds é o Vendor Lock-in [46], o que na tradução para o português seria algo como “aprisionamento do fornecedor”. Mas o que isso significa?

Esse termo, é utilizado para se referir ao contexto em que uma organização deseja transferir seus negócios entre provedores de cloud, porém, o custo para isso é tão alto que o cliente acaba ficando preso ao fornecedor atual. Os obstáculos para essa mudança podem se dar por diversos motivos, sendo alguns deles:

- Violações de dados e ataques cibernéticos
- Dados localizados em um provedor exclusivo.
- Incapacidade de mudar de fornecedor quando um fornecedor não cumpre os SLAs
- Incapacidade de mover dados e aplicativos para fora dos ambientes de nuvem

Um exemplo para esse último ponto citado, acontece, por exemplo, quando você possui um banco de dados em nuvem, que com o tempo acaba contendo um volume de dados muito grande para ser migrado e você acaba tendo que continuar com a solução atual naquele provedor.

Outro caso comum, que se deve evitar para não cair nesse cenário de lock-in, acontece quando se cria aplicativos nos cloud providers. Ao possuir uma arquitetura muito dependente de tecnologias oferecidas pelo provedor atual, seria necessário buscar uma abordagem nova que se encaixasse no provedor de destino, modificando bastante o aplicativo inicial. Esse, é um problema crítico, uma vez que, além de se gastar mais tempo para que o aplicativo seja configurado, também pode haver limitações técnicas dentro da sua equipe, que por possuírem um conhecimento enviesado para o provedor atual, gere mais dificuldades para que se conclua a transição.

4.2 AWS Lambda e Serverless Providers

Quando falamos em FaaS, a preocupação com o Vendor Lock-in é algo que cresce, uma vez que cada Serverless Provider possui uma abordagem

diferente, e às vezes possuímos microsserviços muito orientados para a plataforma em questão.

AWS Lambda, assim como os outros provedores Serverless, pode gerar esse aprisionamento caso não tenhamos um cuidado especial na hora de configurarmos nossas funções. Portanto, o cuidado deve ser inicial, pois uma vez que já tenhamos os serviços montados e extremamente dependentes de outras funcionalidades do provedor utilizado, ou então que possua muitas particularidades específicas a aquela abordagem, pode dificultar a migração.

Para que isso não aconteça, precisamos enxergar pontos que poderiam gerar um possível lock-in no futuro, e tentarmos evitar, dentro do possível, essas abordagens.

Cada provedor de nuvem possui uma assinatura específica para a escrita das funções, porém isso não é algo que gere uma grande preocupação por não haver diferenças muito grandes entre elas. Do ponto de vista de linguagens de programação, o cuidado já deve ser um pouco maior, uma vez que alguma linguagem pode possuir um tempo de compilamento maior em uma plataforma do que outra, caso o foco seja a eficiência do programa, isso pode se tornar um limitador.

O perigo maior mora a nível de API e na utilização de muitos serviços do provedor atual. A nível de API, o problema está no fato de que cada serverless provider possui chamadas a interface de forma diferente, segundo Dean Hallman, fundador e CTO da WireSoft e criador do Cloudbox, “você precisa ter certeza de que há alguma distância entre você e as APIs que está usando” [47], como uma forma de evitar esses problemas.

Já quando analisamos a utilização de muitas funcionalidades do sistema de origem, podemos utilizar até a solução apresentada na seção 3.2.2. Nessa solução possuímos 3 serviços diferentes da AWS interligados no sistema, são eles as Lambdas, o RDS e o Bucket s3. Se levarmos em conta que a instância do banco de dados utilizada no RDS é a SQL Server, comum em bancos de dados, e que possui uma compatibilidade em diferentes plataformas, isso não seria um problema na hora da migração investigando do ponto de vista das Lambdas. Uma vez que o banco fosse migrado, bastaria mudar a string de conexão para o mesmo dentro da função que faz operações nele.

Já o disparo de triggers a partir do s3 seria algo que teria que se ter mais cuidado. Apesar de outros servidores, como a Azure, possuir funcionalidades semelhantes, onde um evento é disparado a partir do upload de arquivos em serviços de armazenamento, a forma como esse gatilho é disparado e chega até a função muda.

Outra particularidade da AWS, que é falada mais a frente na seção 6.2, e que pode gerar problemas é as layers do lambda, onde se armazena libs das linguagens em que o código é escrito. As formas como as libs são chamadas mudam de serviços para serviços, gerando um retrabalho dependendo do sistema de destino a ser migrado.

Portanto, como falado nessa seção, a questão do lock-in é algo que deve ser investigado na fase de implementação da arquitetura das lambdas, prevenindo assim dificuldades maiores no futuro, porém, não é apenas o uso dessa tecnologia que vai fazer você cair nesse cenário.

Uma vez que consigamos isolar mais as lambdas, com relação a dependências de outras funcionalidades da AWS, que podem estar no fluxo do sistema, conseguimos prevenir um lock-in mesmo utilizando essa abordagem. Uma possível solução para isso, é correr de implementações muito voltadas para clouds específicas, se utilizando de soluções multi-cloud.

Um exemplo para isso usando o sistema proposto na seção anterior, seria remover o banco de dados de dentro do RDS, que é um serviço da AWS, ou então, visando funções que ainda não entraram no sistema, como notificações de clientes por meio de e-mail, deveríamos evitar utilizar o serviço nativo da AWS para notificação por e-mail, o SES.

Outra alternativa bastante comum e com bastante aceitação é o uso do Docker [48] e containers [49]. Docker oferece uma alternativa flexível ao AWS Lambda, permitindo aos desenvolvedores a portabilidade entre diferentes infraestruturas de nuvem.

4.4 Sumário do Capítulo

Este capítulo explora os desafios do "Vendor Lock-in" ao adotar serviços serverless, com foco na AWS Lambda. Também discutimos estratégias para mitigar esse problema na escolha de provedores de serviços em nuvem.

No próximo capítulo entramos na metodologia proposta para avaliar os resultados com a migração entre arquiteturas, bem como os experimentos e resultados obtidos. Por fim temos uma seção de discussão para uma análise mais detalhada dos resultados.

5. Metodologia, Experimentos e Resultados

Neste capítulo, descreveremos a metodologia adotada, os experimentos realizados e os resultados obtidos. Exploraremos o uso do GQM para definir metas, as métricas escolhidas e os experimentos conduzidos para avaliar o desempenho do sistema antes e depois da migração para a arquitetura serverless.

5.1 Goal Question Metric (GQM)

A metodologia escolhida para ser adotada para este projeto é o GQM (Goal-Question-Metric) [39], um sistema de medição orientado a metas. Essa escolha estratégica fundamenta-se na necessidade de estabelecer objetivos claros e mensuráveis, proporcionando uma base para a avaliação do desempenho do projeto.

- Nível conceitual (Goal)

Na fase de definição de objetivos (Goal) no GQM, busca-se estabelecer de maneira clara e abrangente o que se pretende alcançar com as medições. Este objetivo funciona como guia, delineando metas específicas que deseja-se atingir durante o projeto.

- Nível operacional (Question)

Nessa fase, um conjunto de perguntas é utilizado para definir modelos do objeto de estudo e, em seguida, concentra-se nesse objeto para caracterizar a avaliação ou o alcance de um objetivo específico. Tem como objetivo fornecer orientação específica para a coleta de dados. Essas perguntas são formuladas de maneira precisa e alinhadas aos objetivos estabelecidos

- Nível quantitativo (Metric)

Na etapa de métricas do GQM, um conjunto específico de medidas, fundamentado nos modelos estabelecidos, é vinculado a cada pergunta para garantir respostas mensuráveis. Essas métricas são escolhidas com precisão para capturar dados quantificáveis, proporcionando uma base sólida para avaliação.

Na Figura 7 abaixo está o modelo do GQM que será utilizado neste trabalho. O objetivo definido para este projeto é “Aprimorar o desempenho do sistema ao migrar de uma arquitetura monolítica para uma arquitetura de microsserviços serverless”. Duas questões fundamentais foram estabelecidas para avaliar o progresso: a primeira questiona se a migração resulta em melhorias no tempo de execução, medido pela "Taxa de Otimização do Tempo de Execução". A segunda questiona qual arquitetura possui o menor custo para

a empresa, utilizando como métrica os "Custos Operacionais". Essas questões e métricas proporcionam uma abordagem direta para avaliar o sucesso da transição e guiar decisões informadas durante o processo.

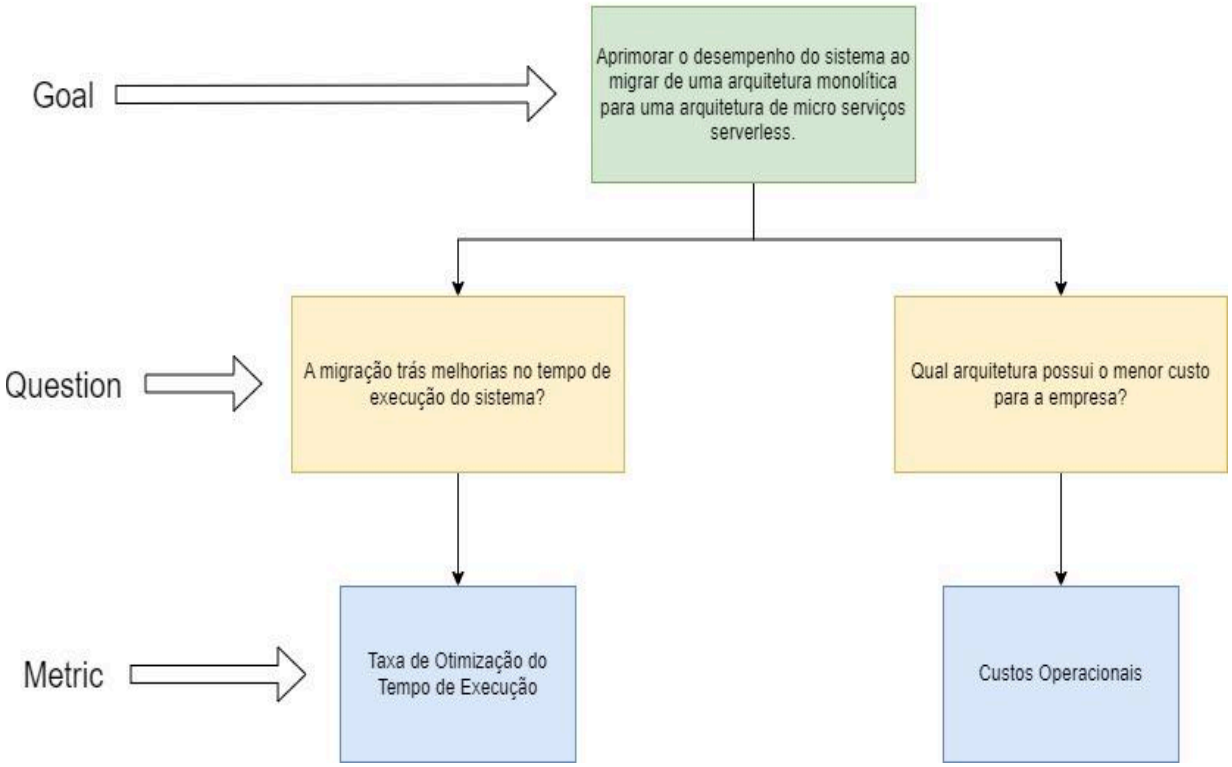


Figura 7 - GQM do trabalho

5.2 Experimento

Para conseguirmos responder as perguntas criadas no GQM, e atingir nosso objetivo, devemos obter métricas para isso.

A primeira métrica escolhida foi a taxa de otimização do tempo de execução. Para calcular o tempo do sistema monolítico antes da migração criamos um decorator [40] de cronômetro para medir o tempo de execução do projeto pré-migração, e fizemos em seguida a medição para 20, 40 e 60 contas, primeiramente, porém essa quantidade se mostrou insuficiente para a avaliação do desempenho do sistema, por limitadores não previstos anteriormente, e aumentamos o número de contas para 240 e 480 para tentar obter resultados mais satisfatórios, do ponto de vista do experimento.

Definimos inicialmente que nossas funções poderiam criar até 40 threads no máximo cada, mas no último experimento utilizamos 100 threads cada para testar a eficiência para um sistema mais robusto, e obter dados que fariam mais sentido para essa possível migração de sistema.

No caso do sistema após a migração para a AWS Lambda, conseguimos metrificar esse tempo de execução a partir do serviço AWS CloudWatch Logs [44], junto com sua funcionalidade Live Tail [41], que nos permite acompanhar em tempo real a execução de nossas funções por meio de logs, e nos facilitar obter uma visão mais clara de que horas a primeira lambda desse processo foi disparada, e que horas foi a última.

A outra métrica escolhida foi o custo operacional de ambos os sistemas, para obter isso, primeiro simulamos a criação de uma máquina virtual na EC2 [42] para que fosse possível rodar o código com arquitetura monolítica de forma local, ou o preço de uma máquina física para realizar essas operações. Para obter uma estimativa do preço da arquitetura serverless e o custo de utilizar a EC2 utilizamos o AWS Pricing Calculator [43], que nos ajuda a estimar o preço de uso de qualquer funcionalidade presente na plataforma, possibilitando o usuário informar as configurações desejadas para cada serviço e assim criar uma estimativa em dólares.

O sistema monolítico, pré-migração, foi compilado em um servidor local com 8gb de memória RAM, processador intel i5-1135G7, ssd de 512mb e sistema operacional Windows.

5.3 Resultados e Análise

Em um primeiro experimento, havíamos dedicado apenas 512mb de memória máxima para as funções, o que se mostrou insuficiente a partir do momento que a taxa de otimização caia consideravelmente com upload de mais arquivos.

Nas tabelas que seguem o experimento, denominamos a coluna de tempo médio da aplicação monolítica de M, a da aplicação serverless de S, por último, calculamos a taxa de otimização fazendo a divisão de S por M, subtraindo 1 do resultado, para obtermos um decimal que representa a porcentagem de ganho de uma em relação a outra.

	M	S	S/M - 1
Quantidade PDF's	Tempo Médio (s) - Monolítico	Tempo Médio (s) - Serverless	Taxa de Otimização
20	25	15	40%
40	47	32	32%
60	71	52	27%

Tabela 1 - Experimento 1

Após isso aumentamos a memória para 1024mb e mantivemos o número de threads para ver se conseguimos algum ganho. Abaixo temos os resultados obtidos. Os dados obtidos no sistema antes da migração foram mantidos, já que o sistema não mudou.

	M	S	S/M - 1
Quantidade PDF's	Tempo Médio (seg) - Monolítico	Tempo Médio (seg) - Serverless	Taxa de Otimização
20	25	14	44%
40	47	29	38%
60	71	49	30%

Tabela 2 - Experimento 2

A partir desse segundo experimento, foi notado um fator que estava influenciando diretamente os resultados, e que não tinha a ver diretamente com o desempenho das funções Lambda propriamente ditas. Que era a velocidade de upload dos arquivos pdf no s3, com taxa de upload em média de 55 KB/s, ou seja, um arquivo por segundo. Uma vez que essas Lambda Functions não possuíam códigos com alto tempo de execução, uma thread ficava disponível antes mesmo do upload de algum arquivo que disparasse essa função.

Como vimos do experimento 1 para o experimento 2, até tivemos uma melhora na taxa de otimização, porém uma melhora pequena para um sistema que dobrou sua capacidade de memória. Na tentativa de contornar essa questão e obter dados relevantes para um sistema de uma empresa que poderia beneficiar-se com essa migração, ou seja, uma empresa que lida com um considerável volume de arquivos, optei por aumentar significativamente o número de contas para 240 e 480, bem como incrementar o número de threads para 100. Essa abordagem visa verificar se haverá melhorias substanciais no desempenho do sistema.

Abaixo temos o resultado desse último experimento.

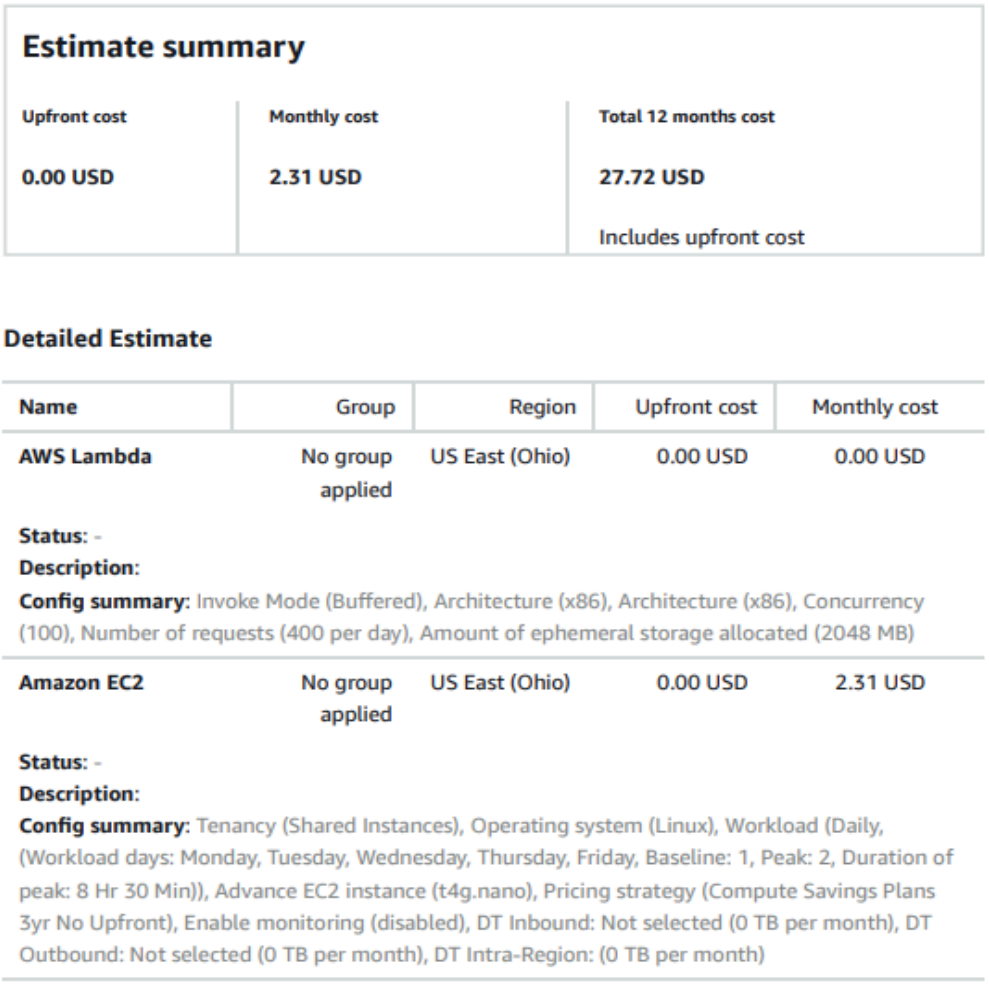
	M	S	S/M - 1
Quantidade PDF's	Tempo Médio (min) - Monolítico	Tempo Médio (min) - Serverless	Taxa de Otimização
240	6:30	3:38	44%
480	12:55	7:00	46%

Tabela 3 - Experimento 3

Todos esses experimentos foram feitos em paralelo com AWS Pricing Calculator, pois, no nosso caso, um aumento significativo no preço das lambdas poderia significar algo não preterido. Porém mesmo na última configuração das funções, dedicando 2048mb e 100 threads para cada, ainda assim não geraria custo nenhum para um fluxo de 400 arquivos por dia, possivelmente até uma lambda com maior poder computacional continuaria sem custo para o usuário.

Já ao tentar configurar uma EC2 que realizaria essas funções nós teríamos um custo mínimo de 2.31 dólares por mês. Essa máquina possivelmente seria fraca para realizar ações que envolvessem uma carga grande de arquivos, por possuir apenas 0.5 de Memória Ram e 2 vCpus, o que poderia levar a mais um aumento no custo.

Abaixo, na Figura 8, conseguimos ver um print extraído diretamente do Pricing Calculator informado acima.



Estimate summary

Upfront cost	Monthly cost	Total 12 months cost
0.00 USD	2.31 USD	27.72 USD Includes upfront cost

Detailed Estimate

Name	Group	Region	Upfront cost	Monthly cost
AWS Lambda	No group applied	US East (Ohio)	0.00 USD	0.00 USD
Status: - **Description:** **Config summary:** Invoke Mode (Buffered), Architecture (x86), Architecture (x86), Concurrency (100), Number of requests (400 per day), Amount of ephemeral storage allocated (2048 MB)				
Amazon EC2	No group applied	US East (Ohio)	0.00 USD	2.31 USD
Status: - **Description:** **Config summary:** Tenancy (Shared Instances), Operating system (Linux), Workload (Daily, (Workload days: Monday, Tuesday, Wednesday, Thursday, Friday, Baseline: 1, Peak: 2, Duration of peak: 8 Hr 30 Min)), Advance EC2 instance (t4g.nano), Pricing strategy (Compute Savings Plans 3yr No Upfront), Enable monitoring (disabled), DT Inbound: Not selected (0 TB per month), DT Outbound: Not selected (0 TB per month), DT Intra-Region: (0 TB per month)				

Figura 8 - AWS Pricing Calculator comparação

Se formos levar em consideração o investimento necessário para comprar uma máquina física como a usada para esse experimento, teríamos um custo aproximado de 2500 a 3000 mil reais, com base em anúncios online de máquinas com essa configuração. Provando que para nosso caso de uso a solução mais barata ainda seria a Lambda.

5.4 Discussão

Dentro do ecossistema de serverless, um dos benefícios frequentemente citados é a escalabilidade. Mas o que exatamente isso significa e como se relaciona com o trabalho desenvolvido até aqui no TG?

Um sistema escalável é capaz de lidar com o aumento na carga de trabalho [50], ou seja, a quantidade de processamento e tarefas que ele recebe em um determinado período de tempo. Como dito nas seções 1.1 e 2.1.4, em

serverless o provisionamento de infraestrutura fica a cargo do provedor, o que a torna escalável por padrão, por ser assim, uma abordagem normalmente preparada para suportar um aumento na carga de trabalho.

A capacidade de escalar automaticamente em resposta ao aumento da carga de trabalho pode levar a uma distribuição mais eficiente de recursos, resultando em tempos de execução mais rápidos quando comparados a uma arquitetura monolítica tradicional.

Uma vez que em serverless você é cobrado pelo uso das suas funções, um aumento na carga de trabalho pode gerar problemas se não for algo planejado. Um aumento inesperado na carga pode acabar acarretando em um custo não estimado pela empresa, e por isso há a necessidade de configurar suas lambdas para limitar o número de threads e memória alocada, como fizemos na seção 3.2.3.

O custo de uma lambda também está diretamente ligado ao tempo de execução da mesma, tendo um limite de 15 minutos por chamada para finalizar a sua execução. Isso justifica a quebra de um sistema monolítico, em diversos serviços menores, como forma de abater um custo indesejado com elas, e aproveitar assim o potencial dessa ferramenta.

Dentre as métricas escolhidas, a importância de otimizar custos é algo quase que auto explicável, já que nenhuma empresa quer ter um custo maior para executar a mesma tarefa. Portanto, o aumento do custo de uma aplicação deve vir do abatimento de custos de outra área, por exemplo, o custo operacional de uma atividade. Visto que nossa aplicação além de gerar um abatimento no custo operacional, pelo fato de atingirmos um tempo de execução menor para essa funcionalidade, também se mostra como uma alternativa mais barata que outras soluções, podemos confirmar a preferência de uma abordagem serverless para esse problema.

Para um projeto futuro, a questão de escalabilidade seria uma boa métrica a ser explorada no GQM do capítulo 5. Apesar de termos na literatura exemplos que indicam uma melhoria do serverless em relação a arquitetura monolítica [53], não realizamos testes para indicar se isso também se mantém na aplicação aqui desenvolvida, embora a tendência seja de que de fato essa migração se torne mais escalável do que a abordagem anterior [54].

5.5 Sumário do Capítulo

Neste capítulo detalhamos a metodologia, experimentos e resultados do projeto, tendo GQM como metodologia para definir metas, métricas e conduzir avaliações de desempenho. Nele são feitos experimentos para nos gerar análises de tempo de execução e custos operacionais antes e após a migração

para a arquitetura serverless, além de uma discussão sobre os resultados obtidos.

No próximo capítulo encontramos o fechamento desse trabalho de graduação, onde desenvolvemos uma conclusão sobre tudo que foi falado e obtido até agora, apontando possíveis limitações do trabalho, e trabalhos futuros.

6. Conclusão e Trabalhos Futuros

Neste capítulo fazemos uma síntese de tudo que foi discutido anteriormente nos outros capítulos desse trabalho, apontando também as possíveis limitações do projeto, e trabalhos futuros a fim de melhorar o que foi obtido com esse projeto.

6.1 Conclusão

Portanto, com ambas as métricas sendo positivas para a migração, mesmo com algumas limitações e problemas não esperados, o que resultaria em algumas alterações para otimizar os ganhos com a migração, temos respondidas as duas perguntas levantadas, o que nos indica que conseguimos alcançar a meta preterida.

As experimentações serviram para que conseguíssemos metrificar a melhoria de um sistema para o outro, e conseguirmos comprovar que no caso em que o sistema foi proposto essa mudança na arquitetura geraria ganhos.

A partir da base teórica formada, no início do processo de escrita desse trabalho de graduação, conseguimos guiar a direção em que esse projeto iria, e entender melhor termos e um ambiente em que o sistema estava sendo proposto. No Estado da Arte, conseguimos apontar de forma rápida algumas das dores para que a migração se fizesse necessária. Outros motivos podem aparecer como explicação para isso dependendo do contexto em que os sistemas se encontrem.

A proposta de migração surge em um ambiente ainda não completamente conhecido, por se tratar de uma tecnologia recente, dificultando um acesso fácil a documentação e projetos de outras pessoas nessa tecnologia. Outra dificuldade encontrada no projeto é a criação de Layers para a utilização de libs no projeto, algumas são fáceis de criar e importar, enquanto outras como a lib de pyodbc com sql server já instalado nela exige um trabalho maior, onde apesar de estar usando um banco de dados no RDS, no mesmo provedor de Cloud, não conseguimos obter essa conexão de forma simples.

Por último vale ressaltar que apesar de termos atingido a meta, há possíveis melhorias para que esse sistema aumentasse consideravelmente seu desempenho, em relação ao antes de sua migração.

6.2 Possíveis limitações

- **Enviesamento dos Resultados:**

  Um fator que dificultou os experimentos e também piorou o tempo de execução, e a medição da eficiência do Microsserviço Serverless foi a taxa de upload do S3 baixa. Uma vez que as lambdas criadas são lambdas com funções específicas, montadas pra fazer apenas o que aquela função tem como responsabilidade, seus tempos de execução são baixos, o que faz com que um arquivo às vezes demore mais para ser carregado no bucket do que para a execução da função, o que acaba gerando um resultado mais semelhante com o tempo de upload do que deveria ser.

- **Conhecimento da tecnologia:**

  Por se tratar de uma tecnologia com não muito tempo de criação, não há tantas pessoas com familiaridade para trabalhar nessa plataforma como existem em outras mais legadas.

  O que pode dificultar a obtenção de mão de obra qualificada para essa função, além de diminuir a quantidade de estudos técnicos e acadêmicos na área.

- **Criação de Layers:**

  A necessidade de criação de layers para uso de bibliotecas das linguagens pode dificultar um pouco o processo de criação de funções, uma vez que não é algo trivial como usar linhas de comando para instalação como no python, como pip [57], e no node.js, com o npm [58]. Algumas bibliotecas são trabalhosas para criação de Layers, o que pode desencorajar o desenvolvimento nessa plataforma.

- **Vendor Lock-in:**

  Como dito anteriormente, no capítulo 4, um possível problema em nossa solução surgiria a partir do crescimento do sistema, o que poderia elevar o custo de nossa aplicação, e a descoberta de uma possibilidade de baratear esse custo ao mover a aplicação para um provedor de cloud diferente. Caso a aplicação não fosse previamente projetada para uma migração mais simples, poderia gerar bastante retrabalho para que isso acontecesse, podendo até impedir a migração desejada.

6.3 Trabalhos Futuros

- **Aumento no número de funções Lambda e no número de Microsserviços**

  Conseguimos com esse projeto mostrar a facilidade de chamada de uma lambda para a outra. Uma vez analisadas as funções de um sistema monolítico, que possuam ligação entre si e que sejam candidatas a serem migradas para a lambda conseguimos criar uma rede de funções que se comunicam entre si sem a necessidade de execução manual, uma vez que temos uma arquitetura orientada a eventos.

  O uso de API Gateways facilita a criação de microsserviços, fornecendo uma interface unificada para gerenciar e expor essas funções, simplificando o controle e a acessibilidade desses serviços distribuídos de forma coesa.

- **Método de upload de arquivos no S3**

  Em nosso sistema, o upload de arquivos no S3 seria o trigger inicial para as chamadas das Lambdas, portanto, devemos prover um meio em que o upload não seja um limitador do potencial da funcionalidade.

  Entre as possíveis soluções estaria o upload de um arquivo compactado como zip, e uma lambda anterior nesse fluxo para descompactar as contas.

  Porém uma solução mais indicada seria utilizar o S3 Transfer Accelaration [45], um recurso em nível que possibilita transferências de arquivos rápidas, fáceis e seguras. Nesse caso as transferências se dariam por meio de endpoints configurados para receber os arquivos. Isso pode gerar um custo maior para o seu produto final, porém dependendo do número de requisições esse custo seria mínimo, ou até inexistente.

- **Idempotence**

  Ao desenvolver aplicações em nuvem, é importante compreender esse conceito, que implica que um alvo específico pode receber ou ser invocado por um evento pelo menos uma vez e possivelmente várias vezes, gerando maior previsibilidade, confiabilidade e consistência nos sistemas [59].

Funções idempotentes faz com que, mesmo que um evento seja processado diversas vezes, o resultado permaneça consistente e evite problemas indesejados, contribuindo para a confiabilidade da sua aplicação.

No projeto aqui desenvolvido, um processo de checagem, para saber se uma conta de energia já foi lida, ajudaria a gerar o conceito de idempotence, pois assim evitaria que ela fosse processada e calculado o aluguel, diversas vezes para uma mesma conta de energia, evitando linhas repetidas no banco de dados, o que não é algo desejado no sistema.

- **Uso do auto-scaling**

Como falado na seção 3.1.2, o tempo de cold start é algo importante no conceito Serverless, onde, apesar de Python possuir o menor dos tempos dentro desse runtime, conseguimos diminuir ainda mais esse tempo de inicialização por outros meios.

Uma das soluções, é manter essa função aquecida com requisições programadas para manter ela sempre ativa, quando não se tem muita previsibilidade nos horários de execução da lambda, que é o nosso caso atualmente.

Num futuro, devemos chegar numa previsibilidade maior, para horários de obtenção dessas contas, e uma solução mais indicada para esse cenário seria o uso da funcionalidade fornecida pela AWS, chamada de Provisioned Concurrency, que nos permite com base em uma programação previsível, aumentar a quantidade de simultaneidade durante períodos de alta demanda e diminuí-la quando a demanda diminuir [60].

7. Referências

[1] Blinowski, G., Ojdowska, A., Przybyłek, A. 2022. Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation.

[2] Tapia, F.; Mora, M. Á.; Fuertes, W.; Aules, H.; Flores, E.; Toulkeridis, T. 2020. From Monolithic Systems to Microservices: A Comparative Study of Performance.

[3] IBM. Serverless Computing. Acesso em: 01/06/2023. Disponível em: <https://www.ibm.com/br-pt/topics/serverless>.

[4] Djemame, K. 2023. Serverless Computing: Introduction and Research Challenges.

[5] Amazon Web Services, AWS. What is Cloud Computing. Acesso em: 01/06/2023. Disponível em: <https://aws.amazon.com/pt/what-is-cloud-computing/>.

[6] Gos, K., Zabierowski, W. 2020. The Comparison of Microservice and Monolithic Architecture.

[7] Michigan Technological University. What is Software Engineering? Acesso em: 07/06/2023. Disponível em: <https://www.mtu.edu/cs/undergraduate/software/what/>.

[8] Medium. Microservices vs. Monolithic Architecture. Acesso em: 12/06/2023. Disponível em: <https://medium.com/startlovingyourself/microservices-vs-monolithic-architecture-c8df91f16bb4>.

[9] Google Cloud. What is Microservices Architecture? Acesso em: 15/06/2023. Disponível em: <https://cloud.google.com/learn/what-is-microservices-architecture>.

[10] Ponce, F., Márquez, G., Astudillo, H. Migrating from monolithic architecture to microservices: A Rapid Review. 2019.

[11] Amazon Web Services, AWS. What is AWS? Acesso em: 05/11/2023. Disponível em: <https://aws.amazon.com/pt/what-is-aws/>.

[12] Microsoft Azure. What is Azure? Acesso em: 05/11/2023. Disponível em: <https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-azure/>.

[13] Google Cloud. Acesso em: 15/11/2023. Disponível em: <https://cloud.google.com>.

[14] Arunkumar, R., Shantakumar, R., Karthick, R. A Study on Cloud Computing: A Review. In: Materials Today: Proceedings, 2020.

[15] IBM. Functions as a Service. Acesso em: 15/11/2023. Disponível em: <https://www.ibm.com/br-pt/topics/faas>.

[16] Google Cloud. What is Cloud Computing? Acesso em: 17/11/2023. Disponível em: <https://cloud.google.com/learn/what-is-cloud-computing?>.

[17] Amazon Web Services, AWS. What is Cloud Computing? Acesso em: 17/11/2023. Disponível em: <https://aws.amazon.com/pt/what-is-cloud-computing/>.

[18] Shaw, M., Garlan, D. 1996. Software architecture: perspectives on an emerging discipline.

[19] Universidade Federal de Alagoas. Acesso em: 25/11/2023. Disponível em: <https://www.repositorio.ufal.br/handle/riufal/1721>.

[20] Incrys. Software Architect's Career Path. Acesso em: 30/11/2023. Disponível em: <https://www.linkedin.com/pulse/software-architects-career-path-incrys/>.

[21] IBM. Serverless Computing. Acesso em: 10/12/2023. Disponível em: <https://www.ibm.com/topics/serverless>.

[22] BMC. What is Serverless Computing? Acesso em: 10/12/2023. Disponível em: <https://www.bmc.com/blogs/serverless-faas/>.

[23] Amazon Web Services, AWS. Event-Driven Architecture. Acesso em: 10/12/2023. Disponível em: <https://aws.amazon.com/pt/event-driven-architecture/>.

[24] Totvs. FIFO, FEFO e LIFO: Entenda a importância dessas metodologias. Acesso em: 06/01/2024. Disponível em: <https://www.totvs.com/blog/gestao-para-rotas/fifo-fefo-e-lifo/>.

[25] Amazon Web Services, AWS. Amazon Simple Storage Service (S3). Acesso em: 09/01/2024. Disponível em: <https://aws.amazon.com/pt/s3/>.

[26] Mendix. Asynchronous vs. Synchronous Programming. Acesso em: 06/01/2024. Disponível em: <https://www.mendix.com/blog/asynchronous-vs-synchronous-programming>.

[27] InfoWorld. Why AWS leads in the cloud. Acesso em: 06/01/2024. Disponível em: <https://www.infoworld.com/article/3596813/why-aws-leads-in-the-cloud.html>.

[28] Datadog. State of Serverless. Acesso em: 06/01/2024. Disponível em: <https://www.datadoghq.com/state-of-serverless>.

[29] Serverless. Acesso em: 07/01/2024. Disponível em: <https://www.serverless.com>.

[30] DigitalOcean. What is MVC? Acesso em: 17/01/2024. Disponível em: <https://www.digitalocean.com/community/tutorials/what-is-mvc>.

[31] DigitalOcean. S.O.L.I.D: The First Five Principles of Object-Oriented Design. Acesso em: 18/01/2024. Disponível em: <https://www.digitalocean.com/community/conceptual-articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design>.

[32] PyPDF2. Acesso em: 18/01/2024. Disponível em: <https://pypdf2.readthedocs.io/en/3.0.0/>.

[33] Amazon Web Services, AWS. Invoke. Acesso em: 18/01/2024. Disponível em: <https://docs.aws.amazon.com/pt_br/lambda/latest/dg/API_Invoke.html>.

[34] Amazon Web Services, AWS. Amazon Relational Database Service (RDS). Acesso em: 18/01/2024. Disponível em: <https://aws.amazon.com/pt/rds/>.

[35] Amazon Web Services, AWS. API Gateway. Acesso em: 21/01/2024. Disponível em: <https://aws.amazon.com/pt/api-gateway/> 21/01

[36] Red Hat. What is YAML? Acesso em: 21/01/2024. Disponível em: <https://www.redhat.com/pt-br/topics/automation/what-is-yaml>.

[37] Amazon Web Services, AWS. AWS Lambda Layers. Acesso em: 22/01/2024. Disponível em: <https://docs.aws.amazon.com/lambda/latest/dg/chapter-layers.html>.

[38] Microsoft. What is SQL Server? Acesso em: 18/01/2024. Disponível em: <https://learn.microsoft.com/en-us/sql/sql-server/what-is-sql-server?view=sql-server-ver16>.

[39] Van Solingen, Rini et al. Goal question metric (gqm) approach. Encyclopedia of software engineering, 2002.

[40] Refactoring Guru. Decorator Design Pattern. Acesso em: 22/01/2024. Disponível em: <https://refactoring.guru/design-patterns/decorator>.

[41] Amazon Web Services, AWS. CloudWatch Logs LiveTail. Acesso em: 22/01/2024. Disponível em: <https://docs.aws.amazon.com/pt_br/AmazonCloudWatch/latest/logs/CloudWatchLogs_LiveTail.html>.

[42] Amazon Web Services, AWS. Amazon Elastic Compute Cloud (EC2). Acesso em: 23/01/2024. Disponível em: <https://aws.amazon.com/pt/ec2/>.

[43] Amazon Web Services, AWS. AWS Pricing Calculator. Acesso em: 23/01/2024. Disponível em: <https://docs.aws.amazon.com/pricing-calculator/latest/userguide/what-is-pricing-calculator.html>.

[44] Amazon Web Services, AWS. What Is Amazon CloudWatch Logs? Acesso em: 23/01/2024. Disponível em: <https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/WhatIsCloudWatchLogs.html>.

[45] Amazon Web Services, AWS. Amazon S3 Transfer Acceleration. Acesso em: 24/01/2024. Disponível em: <https://docs.aws.amazon.com/pt_br/AmazonS3/latest/userguide/transfer-acceleration.html>.

[46] Faction. Vendor Lock-In. Acesso em: 05/02/2024. Disponível em: <https://www.factioninc.com/blog/vendor-lock-in/>.

[47] TechBeacon. Serverless vendor lock: Should you be worried? Acesso em: 07/02/2024. Disponível em: <https://techbeacon.com/enterprise-it/serverless-vendor-lock-should-you-be-worried>.

[48] Docker. Docker Overview. Acesso em: 09/02/2024. Disponível em: <https://docs.docker.com/get-started/overview/>.

[49] Docker. What is a Container? Acesso em: 09/02/2024. Disponível em: <https://www.docker.com/resources/what-container/>.

[50] Weinstock, B., Goodenough, C., On System Scalability. 2006.

[51] IBM. Workload. Acesso em: 01/03/2024. Disponível em: <https://www.ibm.com/topics/workload>.

[52] Amazon Web Services, AWS. New for AWS Lambda – Predictable Start-Up Times with Provisioned Concurrency. Acesso em: 02/03/2024. Disponível em: <https://aws.amazon.com/pt/blogs/compute/new-for-aws-lambda-predictable-start-up-times-with-provisioned-concurrency/>.

[53] Manousian, J. Investigating differences in performance between monolithic and serverless based architectures, 2022.

[54] Split. Monoliths vs. Microservices vs. Serverless. Acesso em: 03/03/2024. Disponível em: <https://www.split.io/blog/monoliths-vs-microservices-vs-serverless/>.

[55] McDonald's Technical Blog. Enhancing Loyalty Rewards: How McDonald's leverages AWS Lambda for Microservices. Acesso em: 04/03/2024. Disponível em: <https://medium.com/mcdonalds-technical-blog/enhancing-loyalty-rewards-how-mcdonalds-leverages-aws-lambda-for-microservices-67e2a243275f>.

[56] RisingStack Blog. How Enterprises Benefit from Microservices Architectures. Acesso em: 04/03/2024. Disponível em: <https://blog.risingstack.com/how-enterprises-benefit-from-microservices-architectures/>.

[57] Python Package Index (PyPI) - pip. Acesso em: 05/03/2024. Disponível em: <https://pypi.org/project/pip/>

[58] npm Documentation. About npm. Acesso em: 05/03/2024. Disponível em: <https://docs.npmjs.com/about-npm>

[59] InfoQ. Achieving Idempotency in AWS Serverless Architectures. Acesso em: 05/03/2024. Disponível em: <https://www.infoq.com/articles/idempotence-aws-serverless-architecture/>.

[60] Amazon Web Services, AWS. New for AWS Lambda – Predictable Start-Up Times with Provisioned Concurrency. Acesso em: 05/03/2024. Disponível em: <https://aws.amazon.com/pt/blogs/compute/new-for-aws-lambda-predictable-start-up-times-with-provisioned-concurrency/>.

[61] Wire19. Cloud Infrastructure Spending Continues to Surge. Acesso em: 25/03/2024. Disponível em: <https://wire19.com/cloud-infrastructure-spending/>

[62] Stackscale. As Linguagens de Programação Mais Populares em 2024. Acesso em: 25/03/2024. Disponível em: <https://www.stackscale.com/blog/most-popular-programming-languages/>